



PORTLETS AND APACHE PORTALS

Stefan Hepper
Peter Fischer
Stephan Hesmer
Richard Jacob
David Sean Taylor
Mike McCallister

 MANNING

Preface	4
About this book.....	4
Roadmap.....	8
Code conventions and downloads	9
Acknowledgements.....	9
About the Authors.....	10
Chapter 1 My first portlet	13
1.1 The current state of the Java Portlet Specification	13
1.2 Getting started	15
1.4 Writing the portlet code.....	18
1.5 Creating the portlet deployment descriptor	20
1.6 Building and packaging the portlet with Ant.....	21
1.7 Deploying the portlet application.....	26
1.8 Portlets in action: running Hello World.....	33
1.9 Summary.....	34
Chapter 2 The Big Picture	35
2.1 What is a portal?.....	35
2.2 And what is a portlet?.....	42
2.3 Creating portlets.....	47
2.4 Going beyond servlets: the portlet architecture.....	50
2.5 Running Java portlets remotely	62
2.6 Summary.....	63
2.7 References	63
Chapter 3 Building Portlets by example	65
3.1 The Bookmark portlet example.....	65
3.2 The Calendar portlet application example.....	75
3.3 Adopting Portlet best practices	92
3.4 Future features: next generation portlets.....	98
3.5 Summary.....	101
3.6 References	101
Chapter 4 Building Portlets with Eclipse and other tools.....	102
4.1 Using Eclipse for portlet development.....	102
4.2 Developing the Bookmark portlet in Eclipse	108
4.3 Commercial development tools	123
4.4 Summary.....	128
4.5 References	128
5 Using JavaServer Faces with Portlets.....	129
5.1 Using JavaServer Faces	129
5.2 Converting existing JSF Applications to Portlets	132
5.3 Running JSF Portlets.....	137
5.4 Portlet Concepts applied to JSF.....	141
5.5 The whole example	147
5.6 Summary.....	151
5.7 Resources	151
5.8 Further Reading	152
Chapter 6 Exploring a sample portal architecture	153
6.1 Understanding the generic portal architecture	153
6.2 Defining the portal components.....	155
6.3 Understanding the component interaction	159
6.4 Portal programming model	169
6.5 Summary.....	174
Chapter 7 Working with the Pluto Open Source Portlet Container	175
7.1 What is Pluto?.....	175
7.2 The portlet container architecture	178
7.3 How to set-up Pluto.....	188
7.4 How use Pluto Container in your own Portal	193
7.5 Summary.....	205
7.6 Resources	205

Chapter 8 Working with the Jetspeed Portal	206
8.1 Introducing the Jetspeed Enterprise Portal	206
8.2 Portlet Development with Jetspeed	209
8.3 Installing the Jetspeed-1 Enterprise Portal	210
8.4 Using standard portlets with Jetspeed-1	210
8.5 Getting started with the new Jetspeed-2	215
8.6 Building the book's portlet samples with Maven	217
8.7 Summary	218
Chapter 9 Working with WSRP	219
Understanding web services	220
Understanding Service Oriented Architecture	221
Introducing WSRP	223
Working with the Actors in the WSRP World	230
Handling Basic WSRP Interactions	232
Generating URLs	242
Managing Sessions	246
Providing Portlet Customization	247
Summary	251
References	251
Chapter 10 WSRP in Action: WSRP4J	252
Understanding the Architecture	252
Getting Started: WSRP4J Prerequisites	256
Setting up the Producer	257
Setting up the Test Portlet	259
Setting up the Swing Consumer	261
Setting up the Proxy Portlet	265
Summary	271
References	271
Chapter 11 Developing Portlets Using Existing Web Frameworks	272
Bridging two worlds: the Web and Java portlets	272
Developing portlets using the JSP Bridge	273
Developing portlets using the JSF Bridge	275
Developing portlets using the Struts Bridge	281
Introducing the Bridges Framework	291
Developing Portlets with Spring and Portals Bridges	292
Developing portlets using the Velocity Bridge	307
Summary	315
Chapter 12 Customizing the Jetspeed Portal	316
Getting to know the default Jetspeed-2 portal	316
Introducing the Waxwing Benefits Custom Portal	323
Setting up the Portal Site structure: Folders and Pages	324
Aggregating the Page Content with Jetspeed Layouts and Decorators	327
Securing your portal	331
Creating Content for the Portal Pages	333
Planning Your Portal Integration Points	341
Summary	343
References	343
Chapter 13 Implementing Database Access	344
About the Waxwing Benefits member portlets	344
Database Access using JDBC: the Claim Status Portlet	345
Accessing user attributes	350
12.2 Implementing MVC with the Deductible Status Portlet	352
Browsing databases: the claims browser and claims detail portlets	359
Using inter-portlet communication: the claims browser and claims detail portlets	371
Summary	377
Chapter 13 Implementing Database Access	378
About the Waxwing Benefits member portlets	378
Database Access using JDBC: the Claim Status Portlet	379

Accesssing user attributes.....	384
12.2 Implementing MVC with the Deductible Status Portlet	386
Browsing databases: the claims browser and claims detail portlets	393
Using inter-portlet communication: the claims browser and claims detail portlets	405
Summary.....	411
Appendixes.....	412
Appendix Portal market	414
Appendix B. Introduction to Web services	419
Appendix C Enabling technologies.....	426
Appendix D Table of Acronyms.....	432
Appendix E References	434
Further readings	436

Preface

This has been a long process. We thought, after creating the Java Portlet Specification and the Web Services for Remote Portlets and providing implementation for both portlet standards at Apache, “we have now done everything to enable people writing portlets by themselves and get started in portlet development.”

We soon realized, based on questions we answered in the different news groups, and portlet code we saw that specifications are not that easy to read and understand. We therefore wanted to provide people with something close to hand that explains the most important concepts:

- The portlet programming model
- Portlet best practices
- Commonly used frameworks in portlet programming
- Different Apache projects that center around portlets

We also wanted to provide you with many sample portlets to enable you learning by seeing running code and being able to play around with the code to get a better and deeper understanding.

We hope to have provided all this information in this book for you so that you can start developing your own portlets and avoid many of the pitfalls that first-time portlet writers may get themselves into. So enjoy the new and exciting world of portlets!

About this book

Portals are Web sites that serve as a starting point to information and applications on the Internet or from an intranet. To accommodate the aggregation and display of diverse content in a dynamic manner, a portal server must provide a framework that breaks the different portal components into pluggable parts. One class of these parts are portlets.

This book explains how you can develop Java and Web Services for Remote Portlet compliant portlets and how to test and deploy these portlets on Apache open source software. The book is split into three main parts: the first part introduces the portlet technology; the second part explains portals and the Apache open source software available for portlet development. The third part gives more portlet examples and shows how to set-up a complete portal.

Part I - Fundamentals

In the first chapter we step right away into the development of a simple JSR168 portlet, namely the world’s most famous portlet: the Hello World Portlet. We will develop some portlet code, package the portlet into a web archive and deploy the Hello World Portlet on the Pluto Portal. This hands-on chapter will give you an introduction into the portlet world and show how simple portlet development can be.

In the second chapter we will broaden our view a bit and take a look at the overall picture. What is a portal? How do portlets fit into portals? We will also discuss how portlets are used, starting with proprietary portlet APIs, moving to the standard Java portlet API to creating complete portlet applications, either from scratch or based on converting already existing web applications to portlet applications.

Part II – Portlets in Action

After covering all the basics in the previous part we will now create a complete and function-rich portlet example: the Bookmark portlet, in Chapter 3. In this example we’ll show you how to leverage what you’ve learned in the first part: create a portlet based on the Model-View-Controller pattern, support localization and

use the portlet preferences in order to store persistent data. Next we'll create a more complex example including two portlets and sharing of data between these portlets. The example will include a calendar portlet, where you can select a specific calendar date and a todo portlet that allows you entering todo's for the selected date. We'll also give you an example on how to leverage the portlet render parameter in order to store view state, like the current month that the calendar portlet should display, in the URL. We will also discuss our favorite portlet best practices in order to enable you creating maintainable and scaleable portlets. The chapter ends with a section about an outlook where portlets are heading.

While you can develop portlets with nothing more than a plain text editor, chances are you will want something more sophisticated. We will cover Eclipse for portlet development in Chapter 4. You'll learn which portlet parts Eclipse will automatically generate for you, and which parts you need to fill in yourself. After that, we will take a look at advanced tooling topics that will really make portlet application development very simple, including support for frameworks like Struts or JSF and allows you to create your portlet application visually instead of typing in code. In Chapter 5 we will cover a very popular framework that can ease portlet development a lot: the Java Server Faces (JSF) technology. We will explain how JSF can be used inside portlets and how the different portlet concepts can be applied to JSF. We will also demonstrate how you can easily convert an existing JSF application with minor effort into a portlet application. Finally we put the learned into practice and let the JSF demo applications that we converted into portlets run on Pluto and Jetspeed-2 and use the already well-known bookmark example to demonstrate how to leverage the JSF technology inside of a portlet.

Part III - A Closer Look: Portlets, Portals and Open Standards

If you ever wondered how a portal is built, read Chapter 6. In Chapter 6 we will explain the basic building blocks that define a portal and how these different components interact. We will further discuss how portal architecture relates to the J2EE architecture and dive into more details about the major portal components and the programming model of these components.

In Chapter 7 we will cover the Java Portlet Specification reference implementation, Pluto. Pluto is an Apache Open Source project and Chapter 7 will cover both a high level overview and explaining the big picture and a dive into the architecture of Pluto. We will show you hands-on step-by-step what to do to get the Pluto project set up on your system – you will download and install Pluto, then get it running under Tomcat. Finally, for advanced users, we will demonstrate how to use the Pluto portlet container in your own portal.

Chapter 8 introduces you to the Apache Portals open source community and the Jetspeed Enterprise Portal project. You will learn how to get the Jetspeed portal downloaded, installed and configured.

Part IV – Advanced topics

In this part we'll cover the more advanced topics, like remote portlets and integration of existing web frameworks into portal.

In Chapter 9 we'll introduce you to the distributed world of Web Services for Remote Portlets. We will talk about the WSRP vision and how the service oriented architecture can be key to enable this vision. Then we will step deeper into the protocol and show you how Consumers interact with remote portlets.

After you are familiar with the WSRP concepts we'll take a look at the open source WSRP implementation WSRP4J at Apache in Chapter 10. We will talk about its architecture and how to build and install the various components. We will create a sample WSRP producer and consumer and test both samples.

In Chapter 11 we will cover existing web frameworks, like Struts, Velocity and Spring and how to leverage these from your portlets. We deliver an introduction to the bridges available in Apache Portals. We are mainly covering the Jetspeed Portal's useful sample portlets based on the 'bridges' technology. These sample covers amongst others SSO integration, RSS, and Web Services.

Part V – Putting it all together: The Waxwing Benefits Portal

This final part of the book is about hands on portlet application development and creating a custom portal with the Jetspeed-2 portal. In Chapter 12 we'll start by customizing the Jetspeed-2 portal and create a complete portal site, which includes security considerations. We will also discuss some best practices on how to plan your portal integration points in order to integrate components, like backend databases, content management systems, search engines and RSS feeds.

Next we'll implement different database access portlets to access backend database systems in Chapter 13. We'll start with a simple portlet that uses plain JDBC in order to get familiar with the basic concepts. After that we'll create a more complex sample using the Velocity framework. We'll end this chapter with integrating portlet messaging into our sample that consists of one portlet to select data and one portlet displaying the data.

In the last chapter, Chapter 14, we'll implement various advanced portlets. We will start by integrating the content management system Graffito and the search engine Lucene into our sample portal. Next we'll create a portlet displaying RSS news feeds. And finally we'll use a web services in our stock quote portlet.

Who Should Read This Book?

"Portlets in Action" is intended for architects, portlet programmers and people with Java and web programming skills who want to learn how to program portlets. Independent Software Vendors (ISVs) can also use this book to develop high quality portal applications once and make them available in portals from different vendors.

Software architects will find Chapter 2 (explaining the big picture of portals and portlets) and Chapter 6 (detailing the portal architecture) of special interest. Java developers who want to create portlets will see that Chapters 3, 4, and 5 explain everything from the basic portlet programming model to using Eclipse for portlet development and leveraging Java Server Faces for the user interface of your portlet. Chapters 11-14 offer additional and more complex portlet samples.

If you are interested in running a portal we will guide you through the different Apache portal projects in Chapters 7, 8, and 10. These include:

- Pluto, the Java portlet reference implementation
- Jetspeed, a full-blown portal: Jetspeed.
- WSRP4J, a Web Service for Remote Portlet implementation, WSRP4J

You will find a complete portal sample based on Jetspeed in Chapter 12.

People involved in the creation and maintenance of the portal often have different roles. Typical roles are portal administrators, web designers, system integrators, portlet developers, system architects, and managers. The larger the enterprise, the more people can be assigned to roles. Often in a small portal deployment, small teams have to learn two or more roles in managing the portal.

This book tries not to limit itself to small or large portals. Our goal is to demonstrate how to manage a portal whether it is a small, medium or large deployment. Who should read this book? Anyone involved in creating portlets, portlet application or hosting a portal at his or her organization or company. Or if you are just interesting in learning about portlets and portals, we believe this book is a great way to learn in a "hands on" manner. This book is geared towards those who need to see how things work, not just read about theory.

Java and JSP Developers

Java enterprise developers should read this book to learn to write portlet applications and how to integrate their enterprise into a portal. This is done through writing standard Java portlets and grouping them into a standardized deployment known as a portlet application.

The portals in this book are written in the Java programming language. Although there are examples in this book of writing portlets in other languages, this book has a strong Java flavor; Java is the de facto standard for developing enterprise applications. Open committees of experts have standardized the Java 2 Enterprise Edition (J2EE), a software development platform for enterprise-scale Java applications. The example portals in this book, such as Apache Jetspeed, are founded on one of those standards: the Java Portlet API standard. Throughout this book we pack many useful examples of integrating Jetspeed with other open Java standards such as JAAS, JMX, JMS and EJB. The examples are based on portlets. The portlets are integrated with J2EE technologies. This is a crucial role of portals: enterprise integration, which we will discuss in more detail throughout this book.

Java and JSP developers fall under the roles of portlet developers, system integrators and system architects. A primary role in a portal is a portlet developer.

The portlet developer writes portlet code in Java and JSP. Portlets are user-facing web components that are largely concerned with user interaction via the portal such as displaying dynamic content and receiving and processing input. This book teaches the Java and JSP developer how to program to the Java Portlet API standard. But it also goes beyond that in giving insight into the best practices for programming portlets, such as separating business logic from presentation. Following these best practices, Java and JSP developers will learn advanced hands-on portlet programming techniques and portal system design.

As the system integrator, developers often have to know when to write their own code and when to leverage code that has already been written. In a portal this is very much true. A well-known anti-pattern is where developers try to write everything themselves without first looking to see if it has already been done. As software systems mature, system integration is becoming a larger and larger part of the development process. We will teach best practices to the Java and JSP developer for finding and extending existing portlets, portlet applications and web services.

Portal Administrators

Portal Administrators will want to read this book to learn how to manage and secure the portal. The role of the portal administrator is to manage, or administer, the live portal site. Portals always come with a built-in portlet application for administering the portal

Identity Management is about managing people or identities that come to visit the portal. In a portal, users are authenticated via standardized single sign-on authentication, by providing identification and credentials. Users can have different roles in a portal. A *security role* is a privilege granted to users or groups of users. Security roles allow you to grant or restrict access to portal resources for several users at once. A user can have different roles depending on the context of the portal. This context is called the security realm.

Portal Security is also managed by the administrator, and works hand in hand with Identity Management. The administrator can assign permissions to principals, authorizing users by assigning roles to users in the context of a security realm. Portal security is all about securing access to portal resources by granting and denying privileges to principals based on security rules. With the Jetspeed Portal Administration portlet application, the administrator controls security policies dynamically.

User databases are often stored in identity management systems such as corporate LDAP databases. Portals are often integrated with the JAAS (Java Authentication and Authorization Service) standard to provide pluggable authentication of user principals. For example, the Jetspeed portal provides a default JAAS Security Policy out of the box.

Web Designers and Content Managers

Web Designers and Content Managers will find this book useful for learning how to package and integrate web content for use within a portal using standard packaging and integration techniques. With most portals, the look and feel, or style, of the portal can be dynamically customized. The collection of markup, styles and content that make up this look and feel is known as a theme. Themes define how your portal looks to the end user. Themes can be changed on the fly at runtime. Different users or groups of users can see different themes over the same content. Themes are based on common standards including Cascading Style Sheets (CSS).

Enterprise Consultants and Managers

Consultants and managers will find valuable information in this book regarding portal architecture, systems integration, security and overall benefits of using a portal in your enterprise. Portals are designed for high volume traffic, and can be configured to run in web farms and balanced across two or server nodes. Consultants need to have a strong understanding of portal application architectures.

Roadmap

People who just want to dive in with a pertinent example, Chapter 1 is for you. You'll create your standard "Hello World" as a portlet and see that it's not so mysterious and complex.

In this book we will introduce you to portlets, explain the basics of portlet programming and walk with you through a lot of examples. We will also explain to you the Apache Portals project that hosts a lot of the software you need in order to develop, test and run portlets.

We will start as simply as possible with the famous "Hello World" sample code in order to give you a first feeling for what portlets look like. After this sample, you should understand that portlets are not that mysterious and complex so that we can then take an in depth look into portlet programming and help you understand how portlets differ from servlets and other J2EE components.

After covering the theoretical part of the portlet programming model it is time to look into some examples again. We'll do this with a Bookmark portlet and a Calendar application consisting of two portlets. After using our favorite text editor in order to create our samples to this point, now it is time to use a visual development tools. We will use the Open Source tool Eclipse and show how to create and debug portlets with this tool.

After covering all the basics of portlet development it is time to take a closer look on how to implement more complex user interfaces with portlets. We'll use the Java Server Faces (JSF) framework for this and show you how to use JSF in your portlets. JSF provides you with a UI component model and the ability to use predefined UI components in your portlet, like lists, tables, and radio buttons.

Next we'll introduce you to the world of portals and the different Apache Portals projects. As the Java Portlet Specification is currently available as version 1.0 it only covers the most common use cases; more advanced usages, like inter-portlet communication are not part of this first version. Until these more advanced functions become part of the Java Portlet Specification the portlet programmer needs to use portal-specific extensions. Thus it is important to know how portals are built internally and what the capabilities of portals are.

First we'll describe the architecture on which most of today's commercial and open source portals are based upon so that you get a better understanding on the major blocks a portal is build of and the mechanics inside the portal. One very important building block of a portal is the portlet container. The portlet container runs the portlets and is, therefore, important for portlet developers to understand. As a concrete example we will use the Apache Pluto portlet container, which is the reference implementation of the Java Portlet Specification.

Next we'll introduce you to a complete open source portal that you can use for testing of portlets, or even for building portals that can be used in production -- the well known Apache Jetspeed portal. Now you can run your own portal and see your portlets in action and real-world use.

After covering the basics of portals we will go one step beyond Java and introduce you to Web Services for Remote Portlets (WSRP). WSRP allows you to offer your Java portlet to everyone as a WSRP service, while at the same time you can consume a WSRP service other people provide into your portal. With this technology you can make your portlets available to a lot more consumers as they don't need to deploy your portlet on their local machine. Next we'll put what we learned about WSRP into practice and use the Apache WSRP4J project to create our own consumer and producer.

We'll end this advanced part with telling you how to connect your portlet with currently existing web frameworks, like Struts, Velocity or Spring. If you already know any of these frameworks, we will help you use that knowledge in the portlet world for your portlets.

In last two parts we will take another step towards having created a complete portal solution out of different portlets. Together we will set up the Jetspeed portal and create a real portal site. You will learn how to configure security and customize the look and feel of the portal. We'll then create the different portlets to run in the portal, starting with more simple portlets to display account information and benefits up to more advanced portlets, dealing with accessing content management systems and connecting to web services in order to retrieve their data or integrate search engines, like Lucene. We will also cover a portlet that is able to display RSS (Really Simple Syndication) news feeds.

Code conventions and downloads

There are many code examples in this book. These examples will always appear in a code font. Any class name, method name, or XML fragment within the normal text of the book will appear in code font as well.

Many Apache open source communities are very active and thus produce new versions of their software at a rapid pace. This also includes the Apache projects included in this book: Pluto, Jetspeed, and WSRP4J. If not stated differently we used in this book: Pluto V 1.0.1 rc2 (besides Chapter 1, where we used V 1.0.1 rc3 which includes new administration portlets), Jetspeed 1 V1.6, Jetspeed 2 V 2.0, and WSRP4J based on the code of end of May 2005 (at that point in time WSRP4J was still in the incubator phase at Apache and did not have a published release).

Any newer versions may have different install instructions and may need more or less additional libraries. Please check the Apache web sites on the install instructions if you use newer versions.

Complete source code for the examples found in the book can be downloaded from the publisher's web site at <http://www.manning.com/portlets>. This includes also a running Jetspeed portal with all examples installed and running, so you can test them out on your local machine.

Acknowledgements

To work on this book while at the same time being involved in real-life customer engagements and further development of the technologies described was a great opportunity and challenge. Without that we would have not been able to add this many insights to the book that we gained from our day jobs. We would like to thank our management and colleagues from IBM WebSphere Portal for providing us with that opportunity.

Several people have provided reviews, feedback, suggestions about the content of the book, or supported us. We are indebted to Ara Abrahamain, Riccardo Audano, Chris Bono, George Franciscus, Walter Hänel, Jack Herrington, Berndt Hamboek, Stefan Liesche, Kito Mann, Brendan Murray, George Peter, Thomas

Schäck, Larissa Schoeffing, Keyur Shah, Gary Sprandel, and Dirk Wittkopp. Special thanks to Doug Warren for his really extensive comments.

We would also like to thank our technical writer Mike McCallister and our editors from Manning, Jackie Carter and Lianna Wlasiuk, for making a consistent and easy to read book out of all these different chapters written by different authors. Thanks for all your help on this and your patience!

Personal acknowledgements

Peter Fischer--

I would like to thank my fiancé Katja for supporting me during writing this book especially during times of low motivation and low creativity. Furthermore I would like to thank Stefan Hepper for pushing whenever one of us was about to fall behind.

Richard Jacob

My very special thanks go to my wife Nicole for cheering me up during the hard phases while writing on this book and suspending me from my housekeeping duties during this time. I hereby promise to be more active again in this special area in the future. I also thank my baseball teammates and coaches for allowing me to still be on the starting lineup although I was missing a lot of practice sessions.

Stefan Hepper

I would like to thank my wife and my kids for all the patience and support while spending many nights and weekends writing this book. Special thanks to my wife Tina for helping me on many of my drawings, for all the proof-reading and last, but not least, for keeping my spirits up!

Stephan Hesmer

I would like to thank my girlfriend for her patience and support while spending many long days, evenings and weekends writing this book. Special thanks to my good friend Larissa for all the proof-reading.

David Sean Taylor

I would like to thank my sons, Alexi and Nicolas, for allowing me the time to write this book instead of playing soccer with them. Special thanks to my wife Magali, who stood by me through thick and thin. Also would like thank Ate Douma for help with the Struts Bridge section, and the entire Jetspeed-2 team for creating a great open source portal.

Mike McCallister

My thanks go to Jackie Carter and Stefan for bringing me into this most interesting project. Thanks also to my wife Jeanette for endless patience, especially when my door was closed.

About the Authors

Peter Fischer

After finishing his studies at University of Cooperative Education Dresden as top of the class in 1999, Peter Fischer has been a software engineer for IBM in the development lab at Boeblingen, Germany.

He started as a developer of the Component Broker OS/390 product which then became the WebSphere Application Server z/OS. 2001 Peter joined the WebSphere Portal team where he worked as the WSRP product architect and technical team lead of the WSRP team, responsible for the implementation of the

WSRP specification in WebSphere Portal v5.x. He also started the Apache WSRP4J project. Just recently he became responsible for the Integrated Solutions Console (ISC) work in the WebSphere Portal foundation.

In his spare time Peter likes to spend time with his girl friend, to go mountain biking, and to drive and repair vintage cars.

Richard Jacob

Richard graduated from University of Stuttgart, Germany and received a Diploma of Computer Science in 2000. Besides his studies in the late 90's, Richard worked for a German telecommunications company where he gained first experiences in networking, distributed systems, systems management and security. During these years, he was responsible for the architecture and development of the networking infrastructure and a distributed systems management solution based on HP OpenView.

In 2000 Richard joined the IBM Böblingen Development Laboratory where he first worked on the development of the IBM zSeries I/O subsystem management. After two years, in 2002, Richard joined the IBM WebSphere Portal development team. Richard is responsible for the WSRP architecture in WebSphere Portal and for the WSRP standardization and specification. He represents the IBM development team at the OASIS Web Services for Remote Portlets technical committee, where he is a core member. Richard chairs the OASIS WSRP Interoperability subcommittee as well as the OASIS WSRP Publish/Find/Bind subcommittee. Richard co-founded and started the Apache WSRP4J project.

In his spare time he enjoys spending time with his wife and friends. Richard plays baseball in the German Baseball league, likes skiing and playing bass guitar with his Heavy Metal band.

Stefan Hepper

Stefan Hepper received a Diploma of Computer Science from the University of Karlsruhe, Germany, in 1995. After graduating, he worked for three years in the Research Center Karlsruhe in the area of medical robotics and component-based software architectures for real-time systems. In 1998 he joined the IBM Böblingen Development Laboratory where he worked with Java Cards in the areas of security and card management. After working on several pervasive computing standards, like Java Card Forum and SyncML Stefan joined the WebSphere Portal development team. Stefan is the responsible architect for the WebSphere Portal programming model and public APIs and was co-leading the Java Portlet Specification JSR 168. Stefan also started the Pluto project at Apache that provides the reference implementation of JSR 168.

Stefan has delivered a number of lectures at international conferences, like JavaOne, published various papers, and was co-author of the book Pervasive Computing (Addison-Wesley 2001). His research interests are component-based software architectures, pervasive infrastructures, and of course portals and portlets.

Besides working and being with his family and friends, he likes skiing, motorcycling, and driving his old Porsche 914.

Stephan Hesmer

Stephan received a Diploma of Information Technology from the University of Cooperative Education Stuttgart, in 2000.

After graduating, he joined the IBM Böblingen Development Laboratory to work in the WebSphere Portal Team as Portlet Runtime team lead. Currently, he is responsible architect of the Portlet Standard area in WebSphere Portal. Additionally, he is responsible for the integration of WebSphere Portal with its base product WebSphere Application Server. Stephan also helped in a lot of aspects related to the JSR 168 Java Portlet Specification and designed and implemented the initial version of the JSR 168 Reference Implementation, Pluto.

Stephan has worked with C++ and Java for many years. Outside work he enjoys spending time with his girlfriend, hiking, squashing and snowboarding.

David Sean Taylor

A long time ago, David studied Computer Science in Engineering at Arizona State and Arizona Technical Institute. David is now the owner of Bluesunrise, a software consulting company. He is an open source developer and member at the Apache Software Foundation. David is the founder of Apache Portals and Jetspeed-2 open source enterprise portal. He has been involved in developing open source portals at Apache for over 4 years. He is also a member of the Java Portlet API Expert Group. His professional software development spans over 15 years, as a team member in the creation of quite a few products including AutoCAD, Poet Object Database, DST Scan and Workflow engine, H&R Block Electronic Filing, and more recently a number of enterprise portals and syndication engines at various companies.

David has worked with C++ and Java for many years. Outside work he enjoys spending time with wife and two sons, hiking, coaching and playing soccer.

Mike McCallister

Mike has been using Linux and writing about it for several years. He has been using and writing about Windows for even longer. His technology stories have been published in places like Java Developers Journal, Linux Journal, Linux Business Week, and Isthmus.

He is a member of the Society for Technical Communication, the Milwaukee Linux Users Group and the National Writers Union. He writes a technology weblog, Notes from the Metaverse, at <http://radio.weblogs.com/0124049>. Mike is also the author of the Computer Certification Handbook (Arco: 2000). When not typing at his keyboard, he reads a lot of history and walks around the city of Milwaukee and environs.

Chapter 1 My first portlet

In the first days of mass access to the World Wide Web, it was all about search. We all went to places like Yahoo and Lycos and Excite, first to see what we could learn about a topic, and sometimes just to see what was new and cool on the Internet on any given day. As more news organizations came on the web, it became possible to keep up with what was happening in the world at large, or perhaps just the tech corner of it. These search sites mentioned above came to offer more personalized features and started calling themselves portals as they became our window to the Web at large. Today, portals can be broadly focused, like My Yahoo, or focused tightly on a business sector, or just the home page for the corporate Intranet. They are created by assorted proprietary technologies, and, increasingly, on open-source, standards-based tools.

Building a worthwhile and user-focused portal can be a quite a challenge for Web developers. Besides dealing with the content and application logic on their own, developers need to invent technologies for personalization, customization of single components, enablement of a common look and feel for all components, etc. In short, they need to develop a complete infrastructure enabling a personalized and focused end-user experience. This is where portlets come into play. They allow developers to concentrate on their content and their application logic without the need to think about the overall framework. This book aims to make it easier to rise to these challenges by showing you how to create good portlets, the building blocks for effective portals.

In this chapter we step right away into the development of a simple JSR168 portlet, namely the world's most famous portlet: the Hello World Portlet. We will develop some portlet code, package the portlet into a web archive and deploy the Hello World Portlet on the Pluto Portal. This hands-on chapter will give you an introduction into the portlet world and show you how simple portlet development can be.

If you prefer to get a little grounding first in how portlets fit in with portals and the standards that they all rely on, start with Chapter 2, and come back here later.

1.1 The current state of the Java Portlet Specification

Ok, don't get nervous if, as you read this section, you start feeling lost among the many new terms we will mention. The objective here is to get you started as soon as possible with writing your first portlet. In chapter 2, where we will cover portals, portlets and their relationship in detail, it will all become clear. For now, we're just going to provide you with a bare bones introduction to portals and portlets.

Portals are Web sites that serve as an entry point to information and applications on the Internet or from an intranet. To accommodate the aggregation and display of diverse content in a dynamic manner, a portal (also called a portal server) must provide a framework that integrates the various pluggable parts into a consistent user interface for the portal user. One class of these parts are portlets. Other parts, which are not covered in this book, include themes and skins that define the look and feel of the portal page.

Portlets are Java technology based web components that process requests and generate dynamic content. From an enduser's perspective, a portlet is a small window or mini application on the portal page that provides a specific service or information, for example, a calendar or news feed portlet. From an application developer's perspective, portlets are pluggable modules that are designed to run inside a portlet container. The portlet container manages the life cycle of portlets and invokes methods on portlets targeted by an end-user interaction with the portal page.

A portal takes one or more portlets and implements them as pluggable user interface components that provide the presentation layer to Information Systems. This type of deployment allows you to utilize the functionality of portlet components from just about anywhere. By adding portlets to your portal server as described, you can provide users with customizable access to an unlimited variety of applications. To give you a real-life example, suppose someone has written a portlet that displays news. You can take this portlet, deploy it on your portal, and give your users access to this portlet on their portal page. The user, in turn, can customize the portlet further, for example, to display the specific news topics of interest.

So far so good, but how can you be sure that your portlet will run on any portal server that people have installed? What you need is a standard for portlets that is supported by all portal server vendors. This is where the Java Portlet Specification (JSR168) comes into play. It defines a contract between the portlet container and portlets and provides a convenient programming model for portlet developers. The Java Portlet Specification V1.0 was standardized as JSR (Java Specification Request) 168 in 2003.

The JSR 168 process was co-led by IBM and Sun and had a large Expert Group that helped to create the final version now available. This Expert Group consisted of Apache Software Foundation, Art Technology Group Inc. (ATG), BEA, Boeing, Borland, Citrix Systems, Fujitsu, Hitachi, IBM, Novell, Oracle, SAP, SAS Institute, Sun, Sybase, Tibco, and Vignette. As you can see by that impressive list, the Java Portlet Specification was quickly and widely supported. Today many commercial and open source implementations supporting JSR 168 are available (see Appendix A).

The goal of the first version of the Java Portlet Specification was to address the 60 % use cases for portlets. It introduces the basic portlet programming model with two phases of action processing and rendering in order to support the Model-View-Controller pattern (see Section 2.4.3 for more details on the Model-View-Controller pattern).

- portlet modes, enabling the portal to advise the portlet what task it should perform and what content it should generate
- window states, indicating the amount of portal page space that will be assigned to the content generated by the portlet
- portlet data model, allowing the portlet to store view information in the render parameters, session related information in the portlet session and per user persistent data in the portlet preferences a packaging format in order to group different portlets and other J2EE artefacts needed by these portlets into one portlet application which can be deployed on the portal server.

You will find more details for all of these concepts in Section 2.4 of this book.

Due to the fact that the Java Portlet Specification is currently available as version 1.0 and only covers the most common use cases; more advanced usages, like inter-portlet communication are not part of this first version. This means that for the more advanced use cases where portlets would like to interact with other portlets you'll need to use vendor specific extensions for now. In Part 5, we'll show you how to do this with the Open Source Jetspeed portal. However, help is coming soon with the next version of the Java Portlet Specification that, among others, will address the inter-portlet communication use case. We expect that this new JSR will be started in 2005. More information on what the future of the Java Portlet Specification may bring can be found in Section 3.4 of this book.

Now let's get started with our Hello World portlet in the next section.

1.2 Getting started

We will now create our first portlet, the famous Hello World portlet. In order to achieve this we need to

- set up our development environment in order to have all the tools available we later on require.
- organize our project in different folders.
- write the actual Hello World portlet.
- create the deployment descriptor which describes in a XML file to the portal the capabilities of the portlet.
- compile the Java code and package everything together into one archive. We'll use the Ant tool to support us for this task.
- deploy the Hello World portlet on the Apache Pluto portal. We'll show you two ways how you can achieve this: first using a graphical user interface, which is actually also a portlet running on Pluto, and second editing all the required files by hand.
- OK, the final step is to run the portlet and see the output on the screen

So let's start with setting up the environment on our machines.

1.2.1 Setting up your environment

To get the sample running you need a development environment and a runtime environment. Let's start with the development environment. Basically the development environment is very simple. All you really need is a Java Development Kit and your favorite source editor or integrated development environment like Eclipse.

But aren't there tools that help us in the development process? Yes, there are; and we will discuss them thoroughly in chapter 4. But for better understanding it is often better to do things manually, so you understand the basic steps involved.

The development environment for this sample consists of:

Table 1.1: The Development Environment for the "Hello World" sample used in this chapter.

Tool	Description
JDK	For this example we used JDK 1.4.
Editor	Choose your favorite here. In fact we used Eclipse and generated a Java project.
Apache Ant	For the build environment, i.e. compiling the source, packaging the application, etc. we provide you a simple Ant build.xml file. We used the built-in Ant installation in eclipse to run our builds. Of course you can do the build and packaging manually, too if you wish.
Apache Maven	(optional) Pluto uses Maven as the project management and comprehension tool. You need to install Maven 1.0 in order to being able to build and install Pluto using the latest CVS source. Also, Pluto uses Maven for portlet deployment. However, as of writing this book, new administration capabilities found their way into Pluto, which make this manual deployment unnecessary.
Apache Pluto	(optional) We need also the Pluto source for two purposes. First, we need to use Pluto's deployment tool to deploy our portlet on the Pluto portal. As said above, in the meantime administration portlets were developed which ease the deployment procedure a lot. Second, we need the portlet-api-1.0.jar file to compile our project. We bundled the jar file with our example code, so there is no need for you to get the Pluto source and build Pluto from scratch at this point.

Note: You don't need Maven or Pluto for this example, but you can add them to your development environment if you intend to set up everything manually from scratch or want to be able to do more sophisticated things

Once we developed our portlet we would like to test and run it in a portal environment. Since our portlet is written against the standard Java Portlet API (defined by the JSR168) we could deploy our portlet on any JSR168 conformant portal. In our example we used Apache Pluto. The Pluto project implements the JSR168 portlet container and provides a simple portal environment. Pluto runs on top of the Tomcat servlet container.

Ordinarily, the easiest way to install and run Pluto is to use the binary distribution of Pluto found at the Pluto homepage. You can alternatively download the binary distribution from <http://www.manning.com/portlets>. For all examples in this book, we used Pluto 1.0.1 RC3. Release Candidate 3 introduced administration portlets which make portlet deployment very easy. Because Pluto 1.0.1 RC3 was not available yet as a binary distribution when we finished this book, we packaged our own Pluto binary distribution which you can download from the web site mentioned above. This distribution comes with a bundled Tomcat 5.0.28 and a preconfigured Pluto web application ready to run. To install the environment simply extract the binary distribution archive to a directory of your choice. We will refer to this installation directory as TOMCAT_HOME.

The installation process is also described on the Pluto homepage. Chapter 7 provides you with more detailed installation instructions for the Pluto portal. If you encounter any problems in the installation process, please refer to that chapter.

The runtime environment consists of:

Table 1.2: Runtime environment for the Hello World sample

Tool	Description
Apache Tomcat	The servlet container on which the Pluto portal is running. In our example we used the bundled Tomcat 5.0.28.
Apache Pluto	The reference implementation of the Java Portlet Specification. You can download the binary distribution to have a convenient way to run Pluto. In order to being able to use the Pluto portlet deployment administration portlets you should use our binary release coming with the samples. At the time you read this book, the Pluto project might already have released a new binary distribution containing the admin portlets. We used Pluto 1.0.1 RC3 for our examples

1.2.2 Organizing the project source files

Let's start developing our Hello World Portlet project by setting up the directories containing our source files. Don't hurry; you do not need to create these directories manually. We will describe how to setup the project in a few seconds. You can also download the complete example presented in this chapter from <http://www.manning.com/portlets>. But first, let's take a look at the simple source tree:

```
HelloWorldPortlet
├── build
│   └── build.xml
├── lib
│   └── portlet-api-1.0.jar
├── src
│   ├── com
│   │   ├── manning
│   │   │   ├── portlets
│   │   │   │   ├── chapter01
│   │   │   │   │   └── hello
│   │   │   │   │       HelloWorldPortlet.java
└──
```

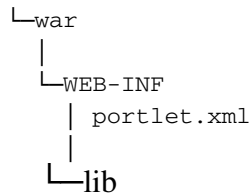


Table 1.3 describes each project directory:

Table 1.3: Directories used for the Hello World sample

Directory	Purpose
root	The root directory is HelloWorldPortlet. It contains all files related to our project.
build	The build subdirectory contains the build.xml file. We use this file to feed Apache Ant with build targets for our project.
lib	The lib subdirectory contains all code libraries maintained in jar files we need to develop our project. In our simple Hello World example we only need the portlet-api-1.0.jar. This file contains the Java Portlet API interfaces and classes. You can obtain this jar file from the Pluto project. Once you successfully have built Pluto, you can copy the jar file from the /jakarta-pluto/api/target directory. For convenience our build.xml file will contain an ant target that copies the Java Portlet API jar file from the Pluto source tree to our lib directory. We will discuss the build and packaging process in section [Build and Packaging]. If you choose to download the sample for this chapter, the lib directory will already contain the portlet-api-1.0.jar.
src	The src subdirectory will contain all of our Java code for the portlet application. The Hello World Portlet project has only a single Java file HelloWorldPortlet.java in the package com.manning.portlets.chapter01.hello.HelloWorld.
war	The war subdirectory holds the contents of the Web archive file package (war file) that we will build. Usually this directory contains all resources needed by the portlet application like images, JSPs, HTML documents, jars, etc. It also contains the portlet.xml deployment descriptor.

1.2.3 Setting up the project

Let's now setup our project. All you need to do is to create the project's root directory along with the build subdirectory. Once you have done this, you can copy the build.xml file into the build subdirectory. We provide this file for you and discuss it in section 1.6.

If you are using Eclipse you can use the following ".project" and ".classpath" files (Listing 1.1 and 1.2) in the project's root directory for the Eclipse Java project. You can then simply import this project into your Eclipse environment. Notice that you don't need eclipse to develop and run this sample.

Listing 1.1: Eclipse project definition

```

<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
  <name>HelloWorldPortlet</name>
  <comment></comment>
  <projects>
  </projects>
  <buildSpec>
    <buildCommand>
      <name>org.eclipse.jdt.core.javabuilder</name>
      <arguments>
      </arguments>
    </buildCommand>
    <buildCommand>
      <name>com.ibm.etools.validation.validationbuilder</name>
      <arguments>
      </arguments>
    </buildCommand>
  </buildSpec>
</projectDescription>

```

```

        <name>com.ibm.etools.ctc.serviceprojectbuilder</name>
        <arguments>
        </arguments>
    </buildCommand>
</buildSpec>
<natures>
    <nature>org.eclipse.jdt.core.javanature</nature>
    <nature>com.ibm.etools.ctc.javaprojectnature</nature>
</natures>
</projectDescription>

```

Now that we have the project defined we also need to tell the Eclipse Java compiler which libraries it needs to include for compiling our Hello World portlet: the Java portlet API jar.

Listing 1.2: Eclipse classpath definition

```

<?xml version="1.0" encoding="UTF-8"?>
<classpath>
    <classpathentry kind="src" path="src"/>
    <classpathentry kind="con" path="org.eclipse.jdt.launching.JRE_CONTAINER"/>
    <classpathentry kind="lib" path="lib/portlet-api-1.0.jar"/>
    <classpathentry kind="output" path="bin"/>
</classpath>

```

Our build.xml file contains an ant target “init” which will setup the necessary directories for us prior to development. Simply invoke ant with this target. We will discuss that process in detail in Section 1.6.

1.4 Writing the portlet code

As mentioned above, our Hello World Portlet will be coded against the Java Portlet API. The Java Portlet API defines the javax.portlet package. The main abstraction of the API is the javax.portlet.Portlet interface. In general all portlets need to implement this interface. This can be done either directly by developing your own class that implements this interface or more conveniently by extending a class that already implements this interface.

The Java Portlet API offers an abstract class javax.portlet.GenericPortlet that implements the Portlet interface and provides default functionality already. It is a good practice to start your portlet development by extending this class.

You can download the Java Portlet specification as well as the Java Portlet API JavaDoc from <http://jcp.org/en/jsr/detail?id=168> to become familiar with the specification and the javax.portlet.* interfaces.

1.4.1 Working with the portlet interface

Let’s take a quick look at the Portlet interface. Similar to servlets which run in a servlet container, portlets run in a portlet container. The Java Portlet Specification defines a contract between the container and portlets. It defines methods the container needs to call during a portlet’s life cycle. Portlet developers implement these methods to provide the desired functionality. Exactly these methods are defined in the Portlet interface:

Table 1.4: Portlet interface methods

Method	Purpose
init()	Called by the container when the portlet is instantiated. This method contains logic that prepares the portlet to serve requests. For example the portlets could initialize backend connections or perform other costly one-time operations.
destroy()	Called by the container when the portlet is taken out of service. This method implements logic that cleans up portlet resources like backend connections. The container usually calls this methods when it shuts down but can also decide to destroy the portlet object at some other point in time depending on the container implementation.
processAction()	Usually called by the container when the user submits changes to the portlet. This method is intended to process input from a user action; it allows the portlet to change it state.
render()	Called by the container when the portlet needs to be redrawn on the portal page. In this method portlets generate content to be displayed to the end-user based on their current state.

The `GenericPortlet` abstract class implements the `render()` method and provides finer grained methods to which it delegates when the container calls `render()`. These methods generate content based on the portlet mode.

But what are portlet modes? The portlet mode is one state item that is maintained by the container for each portlet. The portlet mode indicates which function a portlet is performing. Depending on the mode portlets may behave differently and generate mode-specific content. Each portlet has a current portlet mode, which is passed by the container to the portlet in the request. The Java Portlet Specification defines three modes:

- view,
- help
- edit.

We believe you can figure out straight away what these modes are for, right? We will discuss portlet modes in chapter 3 and give you a more detailed insight there.

Now that we have identified what modes are, and understand that `render` behaves differently depending on them we can look at the lower level methods, which are dispatched in the `GenericPortlet` abstract class in the `render` method based on current mode and overridden by each portlet implementation sub-class:

Table 1.5: GenericPortlet render methods

Method	Purpose
doView()	Called by <code>render()</code> when the portlet is in view mode. The intent is to display the main view of the portlet.
doHelp()	Called by <code>render()</code> when the portlet is in help mode. Usually portlets display a help page that explains the functions of the portlet.
doEdit()	Called by <code>render()</code> when the portlet is in edit mode. Usually portlets display a customization screen here, which allows users to modify the portlet. For example a weather portlet's edit mode could allow the user to change the zip-code or the measurement units for the information it displays in the view mode.

So is this all the `GenericPortlet` does for us? No, there are more helper methods, which allow us to obtain the portlet's configuration data, resource bundles etc. We won't bother about them in our basic example.

1.4.2 Hello World, the portlet way

Let's start coding! Here is our first portlet example:

Listing 1.3: HelloWorld.java source code

```
package com.manning.portlets.chapter01.hello;

import java.io.IOException;
import java.io.PrintWriter;
import javax.portlet.GenericPortlet;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;
```



```
import javax.portlet.PortletException;

public class HelloWorldPortlet extends GenericPortlet
{
    protected void doView(RenderRequest request, RenderResponse response)
        throws PortletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<h1>Hello World</h1>");
    }
}
```

HelloWorldPortlet
extending GenericPortlet

Setting the output content type

Obtaining the output writer object

Writing the content to the output stream

Our HelloWorldPortlet class extends the GenericPortlet and implements the doView() method. The doView() method receives two parameters - the RenderRequest and the RenderResponse objects.

RenderRequest encapsulates all information about the client data, request parameters and attributes, the requested portlet mode, windows state, security attributes, etc. In short, it provides all the information that portlets need to process the request accordingly.

RenderResponse encapsulates all information returned from the portlet to the container. The portlet can set the content type, dynamically change the portlet title, obtain writer objects to write the actual content, etc.

Our HelloWorldPortlet first sets the content type of the response - here we simply hard code the content type to text/html, since we will return simple HTML. Then we obtain the PrintWriter object where we can send our content. And finally we write our content.

1.5 Creating the portlet deployment descriptor

The Java Portlet Specification defines packaging and deployment of portlet application as part of standard Web Archive (WAR) files. Besides portlets these WAR files can contain other resources like JSPs or servlets. In addition to the well-known web.xml file in WAR files, the portlet.xml is required to be part of the deployment descriptor. The portlet.xml contains all information related to portlets and their settings.

Since our Hello World portlet application does not contain any additional web resources, we can omit the web.xml file in our case. But doesn't the portlet specification require a web.xml containing the minimal definition of the portlet application name and description? Yes, it does. But we will use the Apache Pluto deployment procedure, which will generate a web.xml for our application during the deployment process. Therefore we can safely ignore it here.

So all we need is a portlet.xml describing our portlet application. Edit the source of the portlet.xml in the HelloWorldPortlet/war directory as shown in listing 1.4 :

Listing 1.4: Portlet Deployment Descriptor portlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-app xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd"
version="1.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd
http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd">
    <portlet>
        <description>This is my first portlet.</description>
        <portlet-name>HelloWorldPortlet</portlet-name>
        <display-name>Hello World Portlet</display-name>
        <portlet-class>
```

Start of the

Start of the portlet definition

```

com.manning.portlets.chapter01.hello.HelloWorldPortlet</portlet-class>

    <supports>
    <mime-type>text/html</mime-type>
        <portlet-mode>VIEW</portlet-mode>
    </supports>

    <portlet-info>
    <title>Hello World Portlet</title>
    <short-title>Hello</short-title>
    <keywords>portlet, first, hello world</keywords>
    </portlet-info>
</portlet>
</portlet-app>

```

The root of our XML document is the `portlet-app` element. The `portlet-app` element contains all portlets belonging to our portlet application. Since we have only one portlet in our application there is only one `portlet` element in the descriptor.

The `portlet` element describes our portlet in the following way:

Table 1.6: Portlet deployment descriptor elements

Element	Purpose
description	This element provides obviously the description of our portlet. It can be used by the portal environment to provide this information to the end-user.
portlet-name	The value of this element uniquely identifies the portlet within the portlet application.
display-name	The display name can be used by the portal when presenting a list of available portlets to the user.
portlet-class	This element contains the fully qualified class name of the portlet. This is the class implementing the <code>Portlet</code> interface that becomes the entry point for the portlet logic. The container will use this class when invoking the portlet life-cycle methods.
supports	The <code>supports</code> element provides information about what portlet modes are supported for each content type. In our example the portlet supports only the <code>text/html</code> mime type and limits the portlet modes to view only.
portlet-info	Here information about the portlet (usually localized and used in resource bundles) is contained. title: This is the static title of the portlet. It is usually displayed in the portlet decoration on a portal page. Note that the <code>Portlet</code> API allows portlets to change this title dynamically, too. short-title: The short title is intended to be set on devices with limited display capabilities like hand-helds or mobile phones. keywords: The keywords are used by portals which offer search capabilities for their users based on these keywords. This way, users can easier find the desired portlets in a portlet catalogue.

Now that we have successfully created the portlet code and the deployment descriptor for the portlet we take a deeper look into compiling the portlet and packaging everything together into one archive file in the next section.

1.6 Building and packaging the portlet with Ant

Now that we successfully edited our two source files, it's time to compile the code and package our application into a WAR file. To accomplish these tasks we use our `build.xml` file in the `HelloWorldPortlet/build` directory. This file is used as input to Ant and defines targets for compiling the code, creating a JAR file and packaging the application into a WAR file.

If you are not familiar with Ant let's quickly explain the structure of the Ant `build.xml` file. An Ant build file contains exactly one project definition which in turn may contain multiple target definitions. Targets consist of executable tasks which are processed once a target is invoked. Being that said a target within a project can be considered as a set of tasks which are being executed to achieve a distinct goal represented by the target.

For example we have the target “init” in our build.xml file. The purpose of this target is to create the directories we need for our build process. Another target “env” is supposed to display the name of our portlet application we intend to build. Note that the “env” target is dependant on the “init” target. By using the “depends” attribute on a task we can chain the portions of work that need to be done to achieve a goal. This way we can create modular and reusable chunks each intended to achieve a certain goal.

For convenience we provide you the full build.xml (Listing 1.5) first and then step into the targets separately. If you want to get started quickly, you can simply invoke Ant with the default target build. This target will take care of all necessary steps.

Listing 1.5: build.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="HelloWorld" default="build" basedir=".">

    <!-- Some settings we need for the build -->
    <property name="AppName"      value="HelloWorld"/>
    <property name="debug"        value="on"/>
    <property name="dir.build"     value="../build"/>
    <property name="dir.src"       value="../src"/>
    <property name="dir.classes"   value="../classes"/>
    <property name="dir.lib"       value="../lib"/>
    <property name="dir.war"       value="../war"/>
    <property name="dir.war.lib"   value="${dir.war}/WEB-INF/lib"/>
    <property name="dir.driver"    value="../driver"/>
    <property name="portlet.api"   value="portlet-api-1.0.jar"/>
    <property name="pluto.home"   value="PathTo/pluto"/>

    <!-- Build our sample application -->
    <target name="build" depends="build.war"/>

    <!-- Clean all up and rebuild all from scratch -->
    <target name="rebuild" depends="clean,build"/>

    <!-- Initialize the environment-->
    <target name="init">
        <mkdir dir="${dir.src}"/>
        <mkdir dir="${dir.build}"/>
        <mkdir dir="${dir.classes}"/>
        <mkdir dir="${dir.lib}"/>
        <mkdir dir="${dir.driver}"/>
        <mkdir dir="${dir.war.lib}"/>
        <copy file="${pluto.home}/api/target/${portlet.api}"
            todir="${dir.lib}" failonerror="no"/>
    </target>

    <target name="env" depends="init">
        <echo message="Portlet Application: ${AppName}" />
    </target>

    <!-- Compile our source -->
    <target name="compile" depends="env">
        <javac srcdir="${dir.src}"
            destdir="${dir.classes}"
            includes="**/*.java"
            debug="${debug}" />
    </target>
```

Project properties
containing reusable
variables

← Target starting the build process

← Target to rebuild all from scratch

← Target to initialize build environment

← Target to print environment information

← Target to compile the source code

```

        <classpath>
            <fileset dir="${dir.war.lib}">
                <include name="**/*.jar"/>
            </fileset>
            <fileset dir="${dir.lib}">
                <include name="**/*.jar"/>
            </fileset>
        </classpath>
    </javac>
</target>

<!-- Build the JAR file -->
<target name="build.jar" depends="compile">
    <jar jarfile="${dir.classes}/${AppName}.jar"
        compress="true">
        <fileset dir="${dir.classes}">
            <exclude name="*.jar"/>
        </fileset>
    </jar>
</target>

<!-- Build the WAR file -->
<target name="build.war" depends="build.jar">
    <jar jarfile="${dir.driver}/${AppName}.war">
        <fileset dir="${dir.war}">
            <exclude name="**/*.${$}*"/>
        </fileset>
        <zipfileset dir="${dir.classes}"
            prefix="WEB-INF/lib">
            <include name="*.jar"/>
        </zipfileset>
    </jar>
</target>

<!-- Clean up the whole lot -->
<target name="clean">
    <delete dir="${dir.classes}"/>
    <delete dir="${dir.driver}"/>
</target>
</project>

```

Target to package the class files to jar file

Target to build assemble the war file

Target to clean the environment

Let's take a closer look what the above file contains: First we define the project along with some properties, like directory names. Please note that you need to set the Pluto source code root directory for the property "pluto.home". We use this property to automatically copy the necessary portlet-api-1.0.jar file from the Pluto directory. Furthermore we assume that you have successfully built Pluto. If you obtained the jar file from another source and copied it to the lib subdirectory (by downloading it from our web site for example), you can ignore this. The copy step will then just issue a warning but will not stop the overall build process.

Now that we've seen the complete file, let's break it down into the major parts and explain in detail what we did. We'll start with the project directories in the next section.

1.6.1 Setting up the project directories

The first task of our development is to create the directories and copying the necessary Java library portlet-api-1.0.jar our code is dependant on. If you haven't done this manually, the target init will take care of this. It creates all required directories and copies the Java Portlet API jar file into our lib directory. Simply invoke ant with the target init:

Listing 1.6: target init

```
<target name="init">
  <mkdir dir="${dir.src}"/>
  <mkdir dir="${dir.build}"/>
  <mkdir dir="${dir.classes}"/>
  <mkdir dir="${dir.lib}"/>
  <mkdir dir="${dir.driver}"/>
  <mkdir dir="${dir.war.lib}"/>
  <copy file="${pluto.home}/api/target/${portlet.api}"
    todir="${dir.lib}"
    failonerror="no"/>
</target>
```

Now that we've create all the necessary directories let's try to compile the Java portlet code and package everything together into one Java archive file.

1.6.2 Compiling and Packaging the Java Code

Besides compiling our source code, we also want to generate a JAR file containing our compiled classes. This JAR file will later be placed into our WAR file's lib directory. The target compile takes care of compiling the source.

Listing 1.7: compile target

```
<target name="compile" depends="env">
  <javac srcdir="${dir.src}"
    destdir="${dir.classes}"
    includes="**/*.java"
    debug="${debug}">
    <classpath>
      <fileset dir="${dir.war.lib}">
        <include name="**/*.jar"/>
      </fileset>
      <fileset dir="${dir.lib}">
        <include name="**/*.jar"/>
      </fileset>
    </classpath>
  </javac>
</target>
```

← Source file directory definition

← Class file output directory definition

← Classpath definition, contains all jar files to be included in the compilation

The target compile invokes the Java compiler. The source dir is our HelloWorldPortlet/src directory. The compile step includes all Java source files found in this directory and its subdirectories. The output .class files will be copied into our destination directory HelloWorldPortlet/classes. You don't need to worry about creating this directory. The dependent target env and init will take care of this.

To successfully compile we also need to set up the correct class path. We include all JAR files in our HelloWorldPortlet/lib and HelloWorldPortlet/war/lib directories.

Once you invoked the compile target you should see the compiled class file in the classes subdirectory.

Listing 1.8: build.jar target

```
<target name="build.jar" depends="compile">
  <jar jarfile="${dir.classes}/${AppName}.jar" compress="true">
    <fileset dir="${dir.classes}">
      <exclude name="*.jar"/>
    </fileset>
  </jar>
</target>
```

The build.jar target creates the HelloWorld.jar file and stores it into our HelloWorldPortlet/classes target directory. It includes all files excluding other JAR files in the HelloWorldPortlet/classes directory.

After invoking this target you should see the created JAR file in the classes subdirectory. You don't need to worry about first compiling the source. This target will first invoke the compile target if necessary.

1.6.3 Packaging the portlet application

Similar to other Web applications our portlet application will be packaged into a WAR file ready to deploy. Our WAR file structure is as follows:

```
HelloWorld.war
|
└─WEB-INF
   |
   └─portlet.xml
      |
      └─lib
         |
         └─HelloWorld.jar
```

The WEB-INF directory contains our portlet.xml. The lib subdirectory contains our HelloWorld.jar file storing our compiled portlet code.

To create the WAR file we use our build.war target.

Listing 1.9: target build.war

```
<target name="build.war" depends="build.jar">
  <jar jarfile="${dir.driver}/${AppName}.war">
    <fileset dir="${dir.war}">
      <exclude name="**/*.${$}*"/>
    </fileset>
    <zipfileset dir="${dir.classes}"
      prefix="WEB-INF/lib">
      <include name="*.jar"/>
    </zipfileset>
  </jar>
</target>
```

This target creates the HelloWorld.war file and stores it in the HelloWorldPortlet/driver directory. We include all source files from our web archive source directory HelloWorldPortlet/war, as well as all JAR files from our HelloWorldPortlet/classes directory.

Once you have invoked the target you should find our portlet application WAR file in the driver subdirectory. Similar to the other targets you don't need to pre-invoke the compile or build.jar target. If necessary, this will be done for you.

1.7 Deploying the portlet application

We've made it! Our portlet is coded and the portlet application is packaged into a WAR file ready to deploy. In this section, we will use the Apache Pluto portal to deploy and, in section 1.7, run our portlet. We assume you have successfully installed Tomcat and deployed the Apache Pluto web application to Tomcat, so that your Pluto portal is running. If you haven't done so, better hurry up since we're taking off. The easiest way to accomplish this is to download the binary Pluto portal distribution. If you encountered problems with the installation, please refer to chapter 10 for detailed instructions.

As of Pluto 1.0.1 RC3 (bundled with our downloads), there is a new easy way to deploy portlets. We will first introduce you to the new administration portlets and then show you what happens behind the scenes as we manually deploy the portlet and modify the Pluto portal page structure.

1.7.1 Deploying the portlet using the pluto administration portlets

One easy way of deploying portlets is to use the Pluto administration portlets. In order to use them, you need to start Tomcat first and invoke the Pluto portal URL in the browser. By default this is <http://localhost:8080/pluto/portal>. You now should see the initial Pluto portal page.

To invoke the admin portlets, click on the Admin page URL on the left hand side, and you will see the administration portlets, as shown in Figure 1.1.

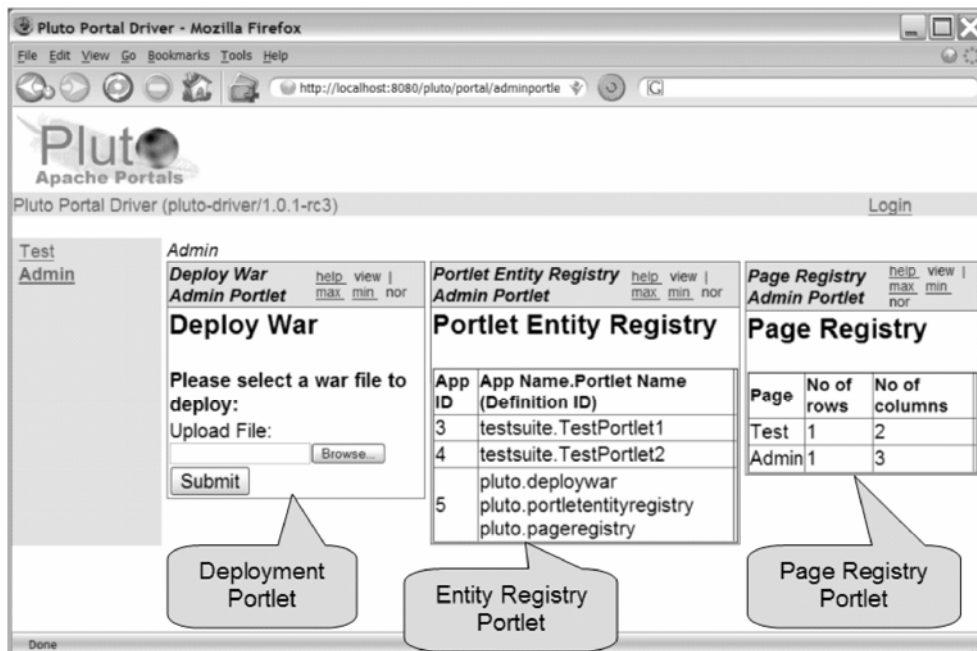


Figure 1.1 Pluto portal administration page with three portlets: the deployment portlet for deploying new portlet applications, the current portlet entity registry and the current page registry.

The Admin page contains the following three portlets:

- **Deploy War Admin Portlet.** Allows you to deploy a new portlet application. We will use this portlet later on in order to deploy our Hello World portlet application.
- **Portlet Entity Registry Admin Portlet.** Shows you information about the currently installed portlets. As you can see, our default Pluto portal distribution has three portlet application installed. The first two are the Pluto test portlets. The third one contains the three administration portlets.
- **Page Registry Portlet** Shows you which pages are currently defined in the Pluto portal. There are 2 pages defined by default. The first one is the test page holding the two test portlets. You might explore

these if you want to. The second page is our Admin page we are currently viewing.

Let's start our portlet deployment. The process is very simple:

- provide the war file we want to deploy
- define a new page to which our portlet will be added
- place the portlet on the new page
- restart Pluto

Providing the war file we want to deploy

The first step is to provide the WAR file we want to deploy. To do this, click on the Browse button in the Deploy War Admin Portlet on the left column of the page. A dialog should appear allowing you to browse your local file system. Locate the WAR file in your HelloWorldPortlet/driver directory, select it and click OK.

The path to the war should now appear in the Upload File input field, as in Figure 1.2.. Click on the Submit button to start the deployment process.

When successfully deployed, you should see a new rendition of the portlets and a message saying that our portlet has successfully been added to portlet registry in the left portlet. To verify this, look on the Portlet Entity Registry Admin Portlet. Our portlet application consisting of the single HelloWorldPortlet now appears on the list of installed portlets. (see Figure 1.3)



Figure 1.2 Select war file of portlet to deploy, in our example the HelloWorld.war we've created.

Defining a new page to which our portlet will be added

The second step is to define a new page where our portlet will be added to. We will define a page with the title "HelloWorld" and the description "Hello World Example". To do this, enter the data to the input fields now presented in the Deploy War Admin Portlet.

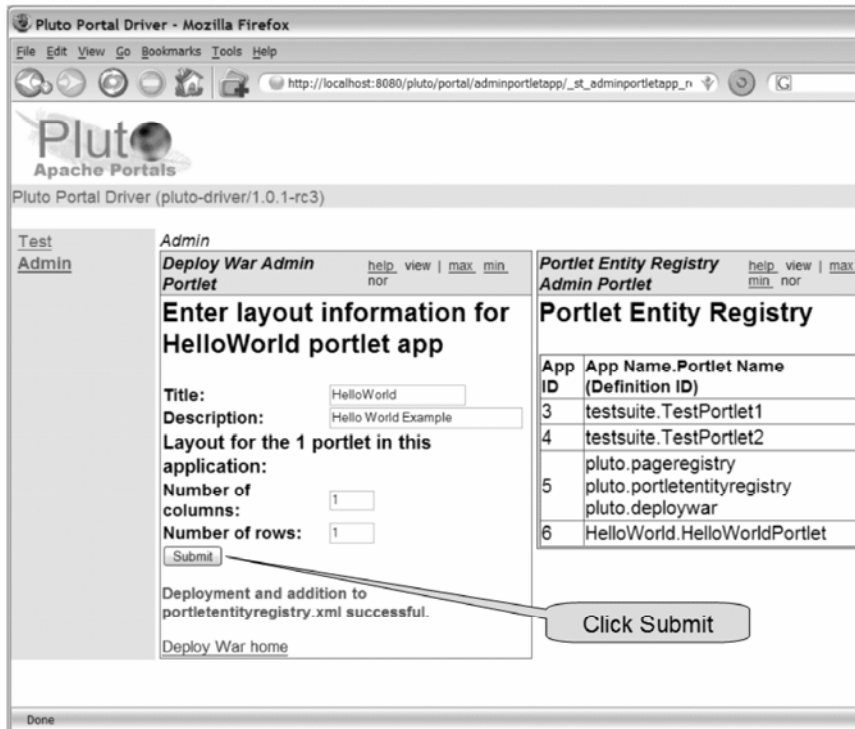


Figure 1.3 Define the Hello World sample page

Since we have only one portlet to deploy, we define the page to consist of one single column and one single row. This will be forming the portlet window for our portlet. When you entered the data, hit the Submit button again. This will bring up the next page of the Deploy War Admin Portlet, as shown in Figure 1.4.

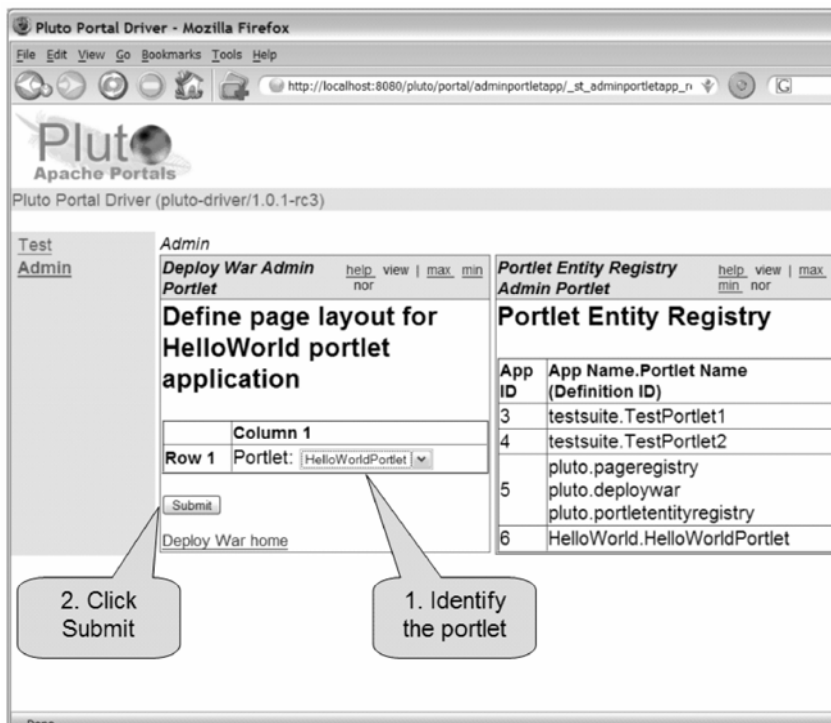


Figure 1.4 Place portlet on the page in the Deploy War Portlet by entering the name of the portlet.

Placing the portlet on the new page

Now that we defined the page with its title, description and layout, the next step is to place our portlet on the page. The current page of the Deploy War Admin Portlet will allow you to do that. Since we have only one cell (one column and one row) and a single portlet, the dialog is already correctly filled in. Hit the Submit button again to complete the process.

The final screen, shown in Figure 1.5, should report the successful deployment of the portlet. Notice that the Page Registry Admin Portlet now shows our new “HelloWorld” page as being defined.

Finally, to make the changes effective, you need to restart the Pluto portal.

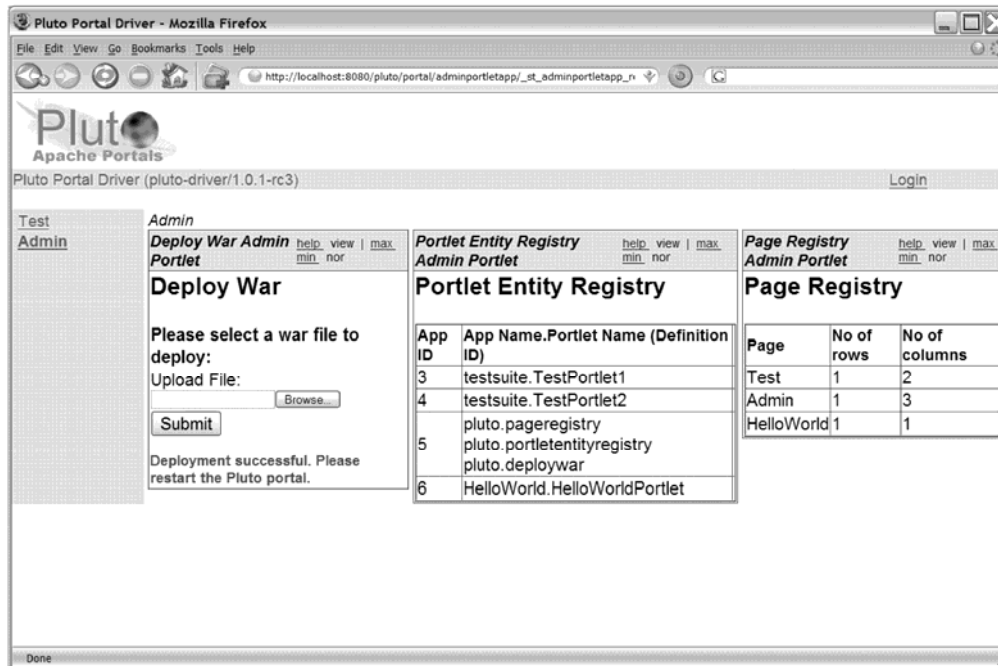


Figure 1.5 Successfully deployed portlet

1.7.2 Behind the Scenes: Deploying the portlet manually

Now that we have deployed our portlet using the admin UI, let's walk through the deployment process manually. Completing the following steps will give us a closer look at how deployment and page definition work in detail:

- deploy our portlet as a web application to Tomcat
- make the portlet known to the Pluto portal and add it to a page
- add the portlet on a portal page
- restart Pluto

Note: If you intend to follow these manual steps make sure to use another clean copy of our binary Pluto portal distribution which has only the default portlets and pages included. Also be sure to install the optional parts like Maven und the Pluto source. If you don't intend to do this extra exercise you can skip to section 1.7 und run our portlet.

Deploying our portlet as a web application to Tomcat

First we need to deploy our portlet as a web application to Tomcat. Every portlet application is handled a separate web application in the application server. To do this, we use the Pluto portlet deployment process. Note that you must first change Pluto's build.properties in the Pluto source root directory to point to your

TOMCAT_HOME directory where you installed Pluto. Initiating the portlet deployment is straightforward. Simply change to the root of your Pluto source and run:

```
maven deploy -Ddeploy=fullPathTo/HelloWorld.war
```

The deployment process extracts our war file into the TOMCAT_HOME/webapps directory. The name of the application directory is the name of the war file. When successfully run, you should see a HelloWorld sub-directory in your TOMCAT_HOME/webapps directory.

Since our portlet application will be handled like any other web application we need a web.xml. The deployment takes care of it and generates a servlet wrapper for our portlet. You can view the contents of this file in the TOMCAT_HOME/webapps/HelloWorld/WEB-INF subdirectory.

Making the portlet known to the Pluto portal and add it to a page

Principally, our portlet is now deployed and ready to run. The next step is to make the portlet known to the Pluto portal and add it to a page. For this purpose, Pluto maintains two configuration files: the portletentityregistry.xml and the pageregistry.xml. The portlet entity registry stores all portlet definitions known to the Pluto portal. Once a portlet is registered, it can be put on a page. You can find these files in the TOMCAT_HOME/webapps/Pluto/WEB-INF/data directory.

We've edited Pluto's entity registry and added a new application containing our Hello World Portlet as can be seen in Listing 1.10:

Listing 1.10: portletentityregistry.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-entity-registry>
  <application id="5">
    <definition-id>pluto</definition-id>
    <portlet id="1">
      <definition-id>pluto.portletentityregistry</definition-id>
    </portlet>
    <portlet id="2">
      <definition-id>pluto.pageregistry</definition-id>
    </portlet>
    <portlet id="0">
      <definition-id>pluto.deploywar</definition-id>
    </portlet>
  </application>
  <application id="4">
    <definition-id>testsuite</definition-id>
    <portlet id="1">
      <definition-id>testsuite.TestPortlet2</definition-id>
      <preferences>
        <pref-name>TestName4</pref-name>
        <pref-value>TestValue4</pref-value>
        <read-only>true</read-only>
      </preferences>
    </portlet>
  </application>
  <application id="3">
    <definition-id>testsuite</definition-id>
    <portlet id="1">
      <definition-id>testsuite.TestPortlet1</definition-id>
      <preferences>
```

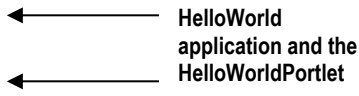
← Pluto administration application and its portlets

← Pluto test applications and their portlets

```

        <pref-name>org.apache.pluto.testsuite.BOGUS_KEY</pref-name>
        <pref-value>notTheOriginal</pref-value>
        <read-only>false</read-only>
    </preferences>
    <preferences>
        <pref-name>TestName4</pref-name>
        <pref-value>TestValue4</pref-value>
        <read-only>true</read-only>
    </preferences>
    <preferences>
        <pref-name>TEST</pref-name>
        <pref-value>TEST_VALUE</pref-value>
        <pref-value>ANOTHER</pref-value>
        <read-only>false</read-only>
    </preferences>
</portlet>
</application>
<application id="6">
    <definition-id>HelloWorld</definition-id>
    <portlet id="0">
        <definition-id>HelloWorld.HelloWorldPortlet</definition-id>
    </portlet>
</application>
</portlet-entity-registry>

```



Our portlet entity registry contains Pluto's default applications and our new HelloWorld portlet application definition. We assign the application id 6 to it – this is important once we come to adding a portlet on a page. Since our application contains only one portlet we define only one portlet entity (with id 0) in this application.

What are the definition-ids for the application and the portlet and how do they relate to our portlet application? The definition-id of the application corresponds to the web application name – in our case HelloWorld (do you remember, our WAR file name is HelloWorld.war and gets expanded to the HelloWorld subdirectory in the Tomcat webapps directory?). The definition id is a concatenation of the web application name and the portlet name defined in the portlet.xml. Take a quick look at code listing 1.4. We assigned our portlet the unique name HelloWorldPortlet. And this is exactly the part of the id after the `.`.

Adding the portlet on a portal page

Once we defined our portlet entity registry, there is one last step left: add the portlet on a portal page. To do this, we need to edit the pageregistry.xml file. We've edited Pluto's default page registry and added a new page to it:

Listing 1.11: pageregistry.xml

```

<?xml version="1.0"?>
<!--
Copyright 2004,2005 The Apache Software Foundation
Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

```

Unless required by applicable law or agreed to in writing, software

distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.

See the License for the specific language governing permissions and
limitations under the License.

-->

<portal>

```
    <fragment name="navigation"
class="org.apache.pluto.portalImpl.aggregation.navigation.TabNavigation">
    </fragment>
```

```
    <fragment name="test" type="page">
```

```
        <navigation>
```

```
            <title>Test</title>
```

```
            <description>...</description>
```

```
        </navigation>
```

```
        <fragment name="row" type="row">
```

```
            <fragment name="col1" type="column">
```

```
                <fragment name="p1" type="portlet">
```

```
                    <property name="portlet" value="3.1"/>
```

```
                </fragment>
```

```
                <fragment name="p2" type="portlet">
```

```
                    <property name="portlet" value="4.1"/>
```

```
                </fragment>
```

```
            </fragment>
```

```
        </fragment>
```

```
    </fragment>
```

```
    <fragment name="adminportletapp" type="page">
```

```
        <navigation>
```

```
            <title>Admin</title>
```

```
            <description>Used for deploying portlets to Pluto</description>
```

```
        </navigation>
```

```
        <fragment name="row" type="row">
```

```
            <fragment name="col1" type="column">
```

```
                <fragment name="p1" type="portlet">
```

```
                    <property name="portlet" value="5.0"/>
```

```
                </fragment>
```

```
                <fragment name="p2" type="portlet">
```

```
                    <property name="portlet" value="5.1"/>
```

```
                </fragment>
```

```
                <fragment name="p3" type="portlet">
```

```
                    <property name="portlet" value="5.2"/>
```

```
                </fragment>
```

```
            </fragment>
```

```
        </fragment>
```

```
    </fragment>
```

```
    <fragment name="HelloWorld" type="page" >
```

```
        <navigation>
```

```
            <title>HelloWorld</title>
```

```
            <description>Hello World Example</description>
```

```
        </navigation>
```

```
        <fragment name="row1" type="row">
```

```
            <fragment name="col1" type="column">
```

```
                <fragment name="p1" type="portlet">
```

Pluto test page definition
containing one row and two
portlets in the columns

Pluto admin page
definition containing
one row and three
portlets in the columns

HelloWorld page definition
containing one row and one
HelloWorld portlet

```

        <property name="portlet" value="6.0"/>
    </fragment><!-- end of portlet frag -->
</fragment><!-- end of col frag -->
</fragment><!-- end of row frag -->
</fragment><!-- end of 'page' frag -->
</portal>

```

Our page registry is simple. It contains three pages, the “test” and “adminportletapp” pages coming with Pluto and our new page “HelloWorld” consisting of one row and column which holds our portlet. We define the page with the name “HelloWorld” and assign it the title “Hello World” used for display.

Within this page definition, we define one row which has exactly one column. Into this table cell we place our Hello World Portlet. To do this we add a property with the name “portlet”. The value attribute identifies our portlet from the portlet entity registry. The value is a concatenation of the application id and the portlet id separated by a `.`. Take a quick look again at code listing 1.9. We defined the application id “6” and we defined the portlet id “0” in this application for our Hello World portlet.

With these changes we are basically done. You need to restart Pluto in order to make the changes effective.

1.8 Portlets in action: running Hello World

Regardless of which route you took to get here, the UI-based deployment of the portlet or the manual steps, we are now ready to go. Let’s start Tomcat and invoke the Pluto portal URL in the browser. By default this is <http://localhost:8080/pluto/portal>. You now should see the initial Pluto portal page.



Figure 1.6 Pluto Portal Main Page

In the figure above you see the main Pluto portal page. The navigation bar on the left presents all defined pages from the page registry we can navigate to. In our example it shows the “Test” and the “Hello World”

pages we defined in Figure 1.3 for the UI-based deployment or Listing 1.10 for the manual deployment..**Error! Reference source not found..**

Click on the “Hello World” link. When Pluto processes this link it renders the page and all portlets on the page. In our case, this is our Hello World Portlet.



Figure 1.7 Hello World Portlet

And here it is, live and in action: the Hello World Portlet! As you can see the defined portlet title (from our portlet.xml) is displayed in the decoration as well as our basic HTML markup fragment.

1.9 Summary

In this chapter we introduced the Java Portlet Specification and stepped right away into the development of a simple JSR168 portlet. We discussed the Portlet interface as the main abstraction of the specification and as the main handler of the container-to-portlet contract. We used the GenericPortlet abstract class as a convenient default implementation of the Portlet interface as a starting point for our code.

We provided you with a basic source tree to start with and showed you how to build and package a portlet application. In this context we introduced the basic information contained in the portlet deployment descriptor portlet.xml.

Also we introduced you to the Apache Pluto portal project as the reference JSR168 implementation and showed you how to deploy and run portlets in this environment.

Hopefully we were able to prove that writing portlets is not only easy, but a viable way to write pluggable visual components for portals. As long as your portlets conform to the Java Portlet Specification, they are deployable in a vendor-independent environment. This means you can run it on any portal server that supports the Java Portlet Specification. This allows you to let other people also use your portlet, even if they run a different portal server and it also will save you from any portlet re-writing in case you consider to switch to a different portal server .

In the next chapter we will broaden our view and step away from the pure portlet developer perspective. We will provide you an overview of portals and portlets and explain how they fit together and what their purpose is. After that we take a look at relevant technologies in the portal domain.

Chapter 2 The Big Picture

In Chapter 1, we wrote our own Hello World portlet and learned how easy it is to create portlets. In this chapter we will broaden our view a bit and take a look at the overall picture to include portals. What is a portal? How do portlets fit into portals? What are the standards that apply to portal and portlet development? Why use portlets instead of servlets?

We will also discuss portlet development, starting with using proprietary portlet APIs, moving to the standard Java Portlet API, and creating complete portlet applications, either from scratch or converting existing web applications to portlet applications.

Finally we make a dive into the architecture of a portlet, and show how portlets can be run remotely.

To really understand the relationship between portlets and portals, we need to start by looking at how a portal is defined and how a portal benefits you.

2.1 What is a portal?

Portlets by themselves are quite useless without a portal to plug into. Put another way, as letters are put together to form words, and words are put together to create sentences, portlets are put together to create one portal page. We must look at portals to understand what portlets are and where they come into play. To do otherwise is like trying to explain why letters are such a great thing without explaining that you can aggregate them together to create words and sentences.

In this section we will take a brief look at the different types of portals that exist and how portlets fit in. Then, we will take a closer look at the runtime environment of portlets: the portlet container. As the servlet container provides the infrastructure for running servlet components in the servlet world, so the portlet container provides the infrastructure for running portlets.

To get started, let's first explain what a portal is and what benefits portals provide the user, namely the integration of several independent applications on one screen.

2.1.1 What does a portal look like?

Let us first take a look at a real portal page before we go into more details of the above definition. Figure 2.1 depicts a typical portal page. The portal consists of several pages, like Welcome, My Workplace, and My Finances.

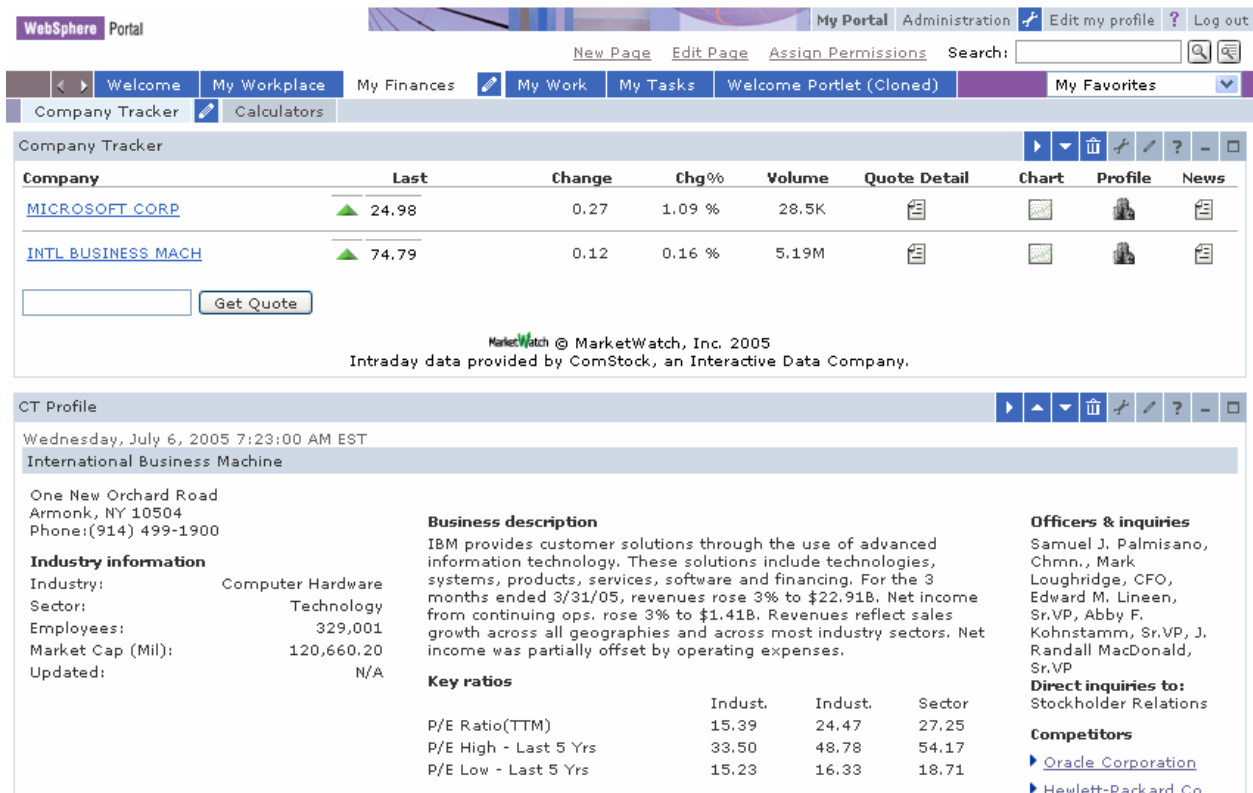


Figure 2.1: This example of a real-life portal page has five portlets that interact with each other. Select a company in the Company Tracker and the Profile portlet display the information for that company.

The figure shows the My Finances page of the portal.. The navigation bar at the top of the page allows the user to navigate between portal pages. The administration links on the top right side allow the user to login or logout of the portal, access the user profile or get help. Two finance-related portlets are shown below this upper bar:

- The Company Tracker portlet - displays different companies and their stock quotes and trends,
- The CT Profile portlet - displays details on a specific company,

The previously mentioned portlets are placed on the page and represent the applications the user can interact with. These different applications are now integrated onto one page with one consistent API and managed centrally. This has advantages for the user, who now can access different applications consistently and in one place, and for the portal administrator, who can manage access to backend services for a specific user in one place.

If we look at this more closely, you can see what a portal really offers. Portals allow users, even when they are using the web, to work more as they do on their desktop, with different applications on one screen that can be interconnected and exchange data.

2.1.2 What does the specification say?

So how is a portal defined? There are quite a lot of definitions available that all center on aggregation and integration. We will take the following definition from the Java Portlet Specification [1]:

“A portal is a web based application that –commonly– provides personalization, single sign on, content aggregation from different sources and hosts the presentation layer of Information Systems. Aggregation is the action of integrating content from different sources within a web page. A portal may have sophisticated

personalization features to provide customized content to users. Portal pages may have different set of portlets creating content for different users.”

This section will shed some more light on the different parts of the portal definition that all center on the main portal theme: application integration.

2.1.3 The overall portal theme: application integration

All main characteristics of a portal have their roots in the main function that a portal provides: integration of different applications on one screen. In this section we will go into more detail on how a portal makes this happen.

Figure 2.2 shows how a portal works in principle: the user interacts with the portal and the portal interacts with different backend (portlet) applications, integrating them all in one browser window for the user. This is often called integration at the glass, as all the applications are integrated into one screen that shows up on the user’s monitor. In today’s world, with a widely fractured IT infrastructure within companies, this is a very powerful concept: Instead of dealing with different IT systems, each with potentially different user interfaces, the user now only needs to deal with one consistent and integrated view of the IT system, that is, the portal.

In both the portlet you created in Chapter 1 and in the portal in Figure 2.1, you can easily see the integration of portlets and portals on the level of presentation. That is, portlets coexist nicely on a single portal page without overlapping each other. What you may not realize is that you can achieve integration on the application level. Advanced portals allow wiring together different applications, gaining synergies that were not possible in the isolated IT application scenario. For example, in Figure 2.1, all the applications are aware of each other. If you select a specific company in the Company Tracker portlet all other portlets on that page get notified about this selection and adapt their content and display details, the stock chart, and news about the selected company.

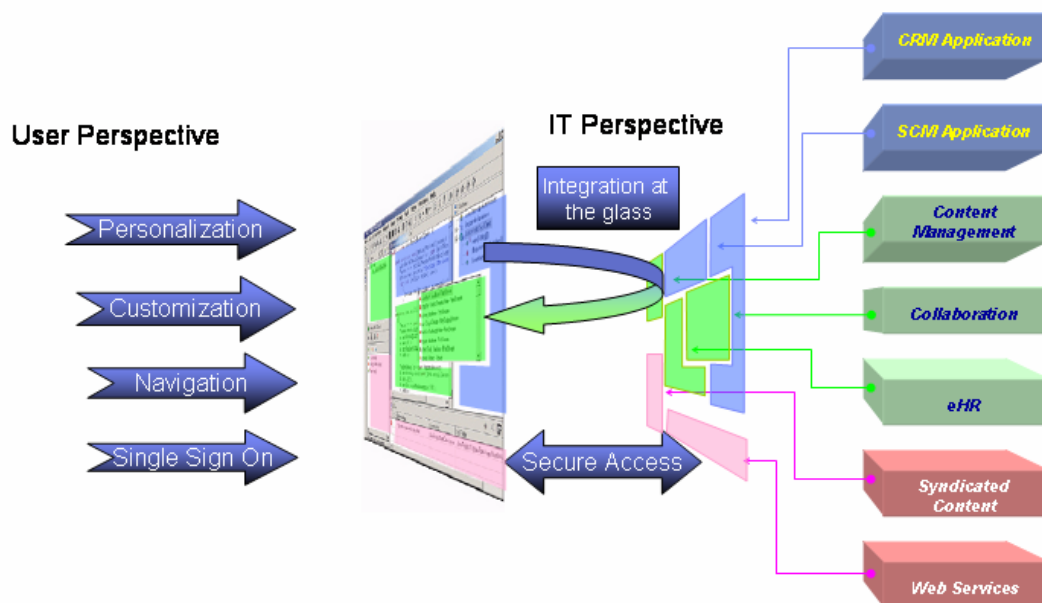


Figure 2.2: Application integration at the glass via a portal. All the backend applications listed on the right come together and display as one entity to the user.

Now let's get back to the more abstract view in Figure 2.2 and take a closer look on how this picture relates to the major points of the Java Portlet Specification's portal definition:

- User personalization – allows portal users to personalize the content they see on a portal page. For example, a user could set specific news topics she is interested in and the Syndicated Content component would filter the content accordingly and only present items matching these topics.
- Back end customization – allows portal administrators to customize given components to the concrete portal environment. An administrator may set the source for the content management system, or the collaboration server, so that the normal user does not need to know this information.
- Single sign on – enables portal users to access different applications that are surfaced. If a user has to login once to get her personalized content, a second time to read a news item from one source and a third time to bring up a calendar application, she will not be coming back often. It's just too much trouble. With single sign-on, the portal feels like it really is integrated, as she now only needs to login to one system and not all the different backend systems with different login names and passwords.
- Content aggregation – is the task that the portal performs in order to provide the user one screen consisting of different applications. The portal also provides navigation elements on the screens in order to enable the user to navigate between different applications on different screens.
- Portlet component concept – allows application providers to produce components that plug in seamlessly into the portal. They are the basic building blocks, like LEGO pieces, that can be plugged together and thus provide the end user with the all the applications that she needs integrated on one screen.

All of these aspects will show up all through the rest of the book and will be explained in more detail and filled with life in examples. Often portals are compared to ordinary websites. Even though a website can be included into a portal, comparing the two is misleading and incorrect. The most important differences between the two are highlighted in table 2.1.

Table 2.1 Differences between ordinary web sites and portals

Web Site feature	Portal feature
Predefined home page	Personalized, task oriented information
Static	Information interaction
Stand alone	Integrated with business processes
No transaction support	Fully transactional
Browser only	Multiple methods of delivering information including PDA's, phones and browsers

For now let us stick a bit more to the foundations and take a look at the different portal types that exist.

2.1.4 Different portal types

As portals became more and more widespread they were also applied to very different segments and have a wide variety of targeted users. This variety makes it very confusing for someone who would like to provide applications for portals, as different portal scenarios have different requirements.

In this section we will first cover briefly the traditional portal classifications, like the Business-to-Employee portal and the difference between vertical and horizontal portals. Then we will come to a size-based classification with Nano, Micro, and Macro portals that provides more information about the requirements on portlets deployed on these portals for portlet application providers.

Vertical and horizontal portals

In the early days of portal deployment, portals were focused towards specific tasks and user groups. Therefore the traditional portal classification is based on the relation between the portal host and the portal user. The basic assumption of this classification is that different users will have different needs for the portal.

Most commonly portals are categorized into

- Business-to-Consumer (B2C) – which is a portal that is tailored towards supporting customers of a specific company. The functionality of the portal may include product availability, ordering status, product information and manuals or customer support. Often these portals are front-ends of Customer-Relationship-Management (CRM) backend systems.
- Business-to-Business (B2B) – is a portal tailored towards companies that deal with the company hosting the portal. These portals typically provide a front-end for Supply-Chain-Management (SCM) backend systems and offer functions like access to purchase orders, invoices, confirmations and information on billing and manufacturing processes.
- Business-to-Employee (B2E) – is an intranet portal that provides employees of a company with unified access to all company IT systems, which may include company news, search engines, travel planning, and collaboration facilities like discussion groups and team rooms. It may also include workflow processes, like in the travel application that may require different steps of approval before being able to book the travel arrangement.
- Internet– is a portal that can be accessed by everyone and thus has a very large user group. Examples for this kind of portals are Yahoo, Google or Excite, which offer the users different services, such as email, news, or search capabilities.

These categories are going to fade away, because one user may be an employee one moment and a consumer the next. Users should not have to learn different methods of relating to portals based on their role at a given moment, but should expect to do all their work in one place.

This is where the terms horizontal and vertical portal come into play. The traditional view represents the vertical portal that is focused on a specific application, like CRM or SCM, or maps to a organization sub-structure, like a department. Horizontal portals are integration portals, also sometimes called Über-portals, which aggregate the content of other portals into a new portal. This allows the end user to only deal with one portal that syndicates all the content and applications of the different vertical portals the user is interested in.

Macro, micro and nano portals

Portals can be integrated in the IT infrastructure on different levels. While this book concentrates on what we call the macro level with the vertical and horizontal portals, we can see more specialized uses for portals that are in use today on a small scale, and may become popular in the future. These other types may demand different portlet applications than the macro portals.

So, what are the different scales that exist today in the portal space?

- Macro portals – this is the scale we have been covering up to now in the book. These kinds of portals integrate existing back end applications into one consistent user experience and are accessed by a large number of users. Examples of this kind of portal include web portals like Yahoo or company portals.
- Micro portals – are one scale below the macro portals and are only used by a single user or a small group of users. This kind of portal is also often used in online and offline mode as the portal is very small and can be installed on a portable device like a laptop computer or a car. Examples of this kind of portal include homeportals, intermittently connected portals (client portals), or branch portals. Often micro portals are intermittently connected to macro portals and synchronize data held locally on the micro portal with the main macro portal. An example for this is a sales representative who can download company data from the intranet portal to her laptop and work offline at the customer with this data. As soon as the sales representative has access to the intranet again she can upload the new sales data to the intranet portal.
- Nano portals – are portals at the smallest level; they are integrated into electronic devices and act as user interface for these devices that may consist of other sub-components. Examples for these kind of portals are home or telephone controllers, portals integrated in refrigerators or other kitchen appliances like a slow cooker, or machines like CNC milling machines.

Figure 2.3 and Figure 2.4 display how these three portal levels may be integrated into one IT infrastructure and provide a very flexible, robust and user friendly interface.

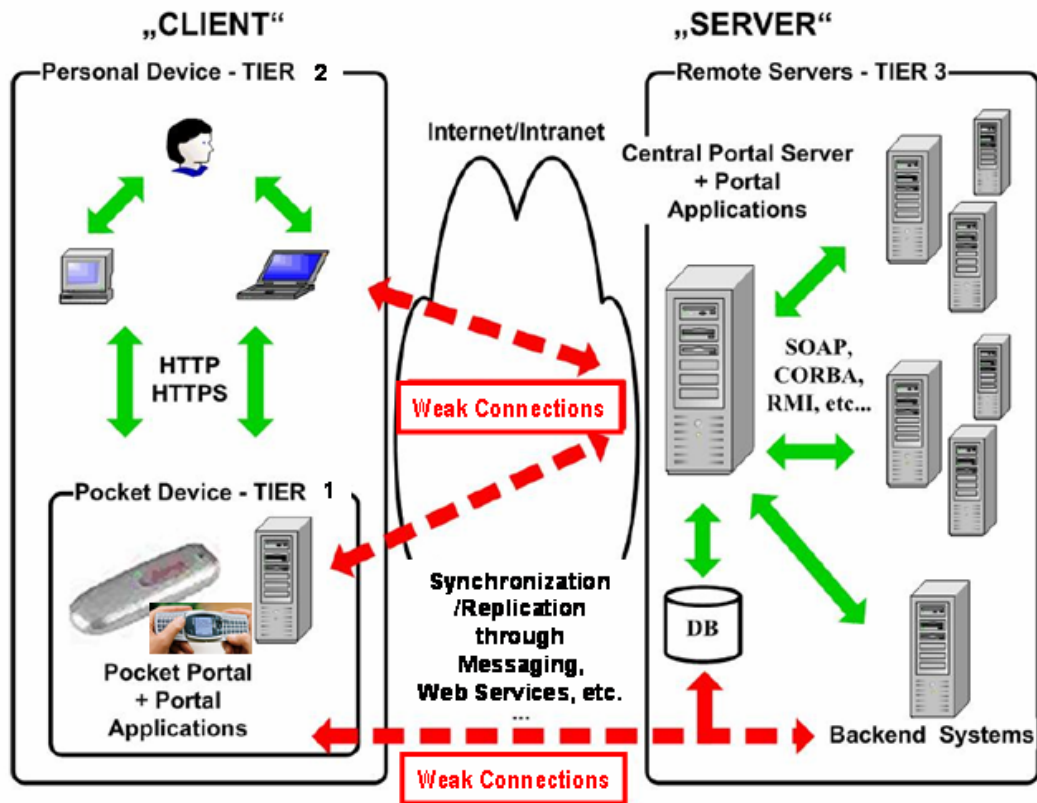


Figure 2.3: This model shows how to integrate portals on different scales into the IT infrastructure. Through various methods, servers containing information can communicate via portals to PCs, laptops and cell phones.

Figure 2.4: User interface of a portal that includes access to micro and nano portals. This portal displaying the status of an assembly line behaves like an assembly line itself, delivering information both to a screen and via email.

In Figure 2.3, you can see how portals of different scale work together. The nano portals on the first tier work together with the micro portals on the second tier via a direct HTTP connection or via a weak connection with the macro portal in tier three. The micro portal itself is also connected to the macro portal via a weak connection in order to replicate data and access services on the macro portal.

Figure 2.4 shows a running portal that integrates all these different levels, and provides the desktop-like experience referred to earlier. The portal displays information about a robotic assembly line. Each robot on the line has an interface (a nano portal) that reports its status to the micro portal that, in turn, displays the status of all the nano portals on the upper left side. This macro portal also contains an email portlet on the lower half that delivers email status messages to the person responsible for this line. The email portlet can synchronize the email between the micro portal and the macro portal running the email server. In some ways, the portal itself can be seen as an assembly line, pulling together information from each robot to give the human supervisor a product that can help him make decisions.

Designing portlets for the target portal type

Of course portals on different scales impose different requirements for portlet application providers (the companies that sell portlet applications). Here are some requirements to consider when building portlet applications on the micro and nano scale:

- Prepare to work in an intermittently connected mode. Micro portals are often used in an environment where a backend connection is not always available. The portlet application still needs to provide meaningful data to the user even if no backend connection is available. Often this is achieved with caching data on the micro portal and writing back the data to the backend system when the connection is available again. This also leads to the second requirement:
- Integrate with existing synchronization infrastructures. As mentioned before, micro portals are often running in a disconnected mode and macro portals integrating these micro portals therefore commonly provide a synchronization infrastructure allowing the portlets to work offline with a local data store and synchronize back the changes when connected again.
- Small runtime footprints. As nano portals have very restricted hardware capabilities they impose restrictions on the memory and computation power that the portlet requires during runtime. Thus in many cases the portlet need to be tailored towards these restrictions and may not provide the same functionality as when running on a micro or macro portal.

Now that we have covered the different kind of portals let's get back on our main theme: portlets. Let's take a look on how portlets are run inside the portal in the next section.

2.1.5 The portal architecture: the portlet container

Now that we have talked a lot about the portal and how portals fit into the IT infrastructure, let's come back to portlets and try to understand where portlets fit into the overall portal.

Portlets are run by a component called a portlet container. This container provides the portlet with the required runtime environment. The portlet container manages the life cycle of all the portlets and provides persistent storage mechanisms for the portlet preferences, allowing portlets to produce user dependent markup. The portlet container passes on requests from the portal to the hosted portlets. The portlet container does not aggregate the content produced by the portlets; that is the portal's job. Figure 2.5 shows the overall portal architecture.

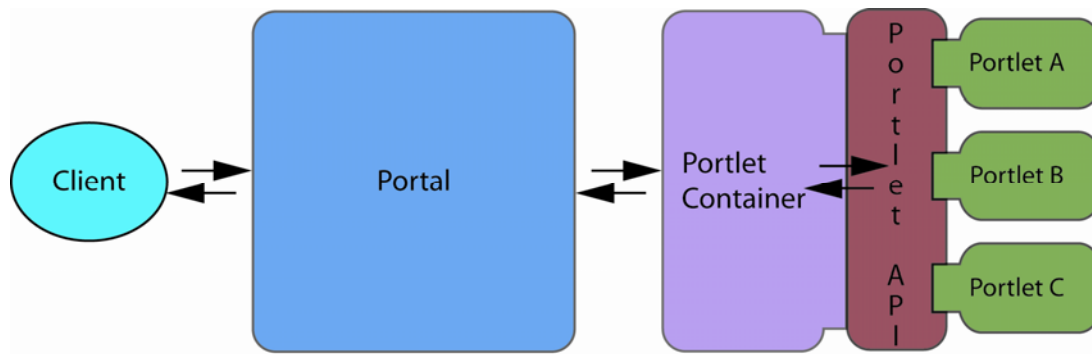


Figure 2.5: Overall portal architecture with the portal aggregating the content and the portlet container running the portlets.

Here's how it works:

1. A registered user (client) opens the portal, the portal application receives the client request and retrieves the current user's page data from the portal database.
2. The portal application then issues calls to the portlet container for all portlets on the current page.
3. The portlet container, which holds the user's preferences, then calls the portlets via the portlet API requesting the markup fragment from each portlet and returning this fragment back to the portal.
4. The portal aggregates all markup fragments together into one page, which the portal finally returns to the client/user, giving them the integrated, useful interface they are used to on the desktop.

More details about the portal architecture can be found in chapter 6 where we explain the components and their sub-components in more detail. For now it should be sufficient to keep in mind that there are the two major components: the portal itself and the portlet container. After learning this much about portals, let's take a closer look at what portlets really are and what their role is in the big picture.

2.2 And what is a portlet?

Now that we know how a portal functions, it is time to take a closer look at portlets and their role in this environment. So let's start with a definition.

A portlet is a Java-based Web component that processes requests from a portlet container and generates dynamic content. The content generated by a portlet is called a fragment, which is a piece of markup (e.g., HTML, XHTML, WML) adhering to certain rules. A fragment can be aggregated with other fragments to form a complete document, called the portal page.

One could ask why portlets were invented and specified in the Java Portlet Specification at all. Why were existing J2EE concepts, namely the servlet, not sufficient? As we have already seen that would lead to challenges in creating a consistent user experience. But what else is there that justifies creating a new component? Table 2.2 lists the reasons why we think portlets are a separate component:

Table 2.2 Portlets vs. servlets as portal components

5. Servlets

7. Web clients interact directly
9. Each servlet assumes it is the only responding component, and produces a complete document.
11. Directly bound to a URL
13. Less refined request handling
- 15.
- 17.
- 19.
- 21.
- 23.

6. Portlets

8. Web clients interact with portal. Portal acts as mediator, provides infrastructure.
10. Portlets assume other portlets are responding to the portal's request, and produce markup fragments. Portal coordinates response to client, handles character set encoding, content type, and setting HTTP headers.
12. Only addressed via portal.
14. Request handling includes action processing and rendering options.
16. Portlets have predefined modes and window states that indicate the function the portlet is performing and the amount of real estate in the portal page available to the portlet.
18. Numerous portlets exist on a portal page and therefore require concepts like the portlet window and portlet entity.
20. Portlets need means for accessing and storing persistent configuration and customization data on a per-user basis. As portlets need to be plugged into an existing portal these storage functions need to be provided by the portal infrastructure for the portlet and thus need to show up in the portlet API.
22. Portlets need access to user profile information to generate user-specific output.
24. As portlets are plugged into portal systems, portlets need URL rewriting functions for creating hyperlinks within their content, allowing the creation of URL links and actions in page fragments to be created independent of the specific portal server implementation.

All these requirements made it a cleaner choice to introduce a new component, portlets, instead of bending the servlet definition to also fulfill these requirements. However, the Java Portlet Specification is very closely aligned with J2EE concepts in order to reuse as much as possible of the existing J2EE infrastructure already available. This close alignment is reflected in

- Portlet applications are packaged as WAR files with an additional portlet deployment descriptor (portlet.xml) for the portlet component and can be deployed using the existing J2EE web application infrastructure for WAR files.
- Portlet applications reuse the standard HttpSession and thus portlets can share data via the session with other J2EE artifacts like servlets and JSPs.
- Portlets can access the web application context via the portlet API and share data with other J2EE artifacts on the context level.
- Portlets can access web application initialization parameters defined in the web.xml via the portlet context.
- Portlets can include servlets and JSPs via a request dispatcher.
- Portlet J2EE roles defined in the portlet.xml can reference J2EE roles defined in the web.xml enabling a unified role mapping between portlets and servlets.

In order to leverage the existing J2EE infrastructure for portlets today, they can be wrapped as servlets and deployed in the web container with the portlet container running on top of the web container (see Figure 2.6). This approach is taken by the Pluto JSR 168 portlet reference implementation as well as in the Jetspeed portal.

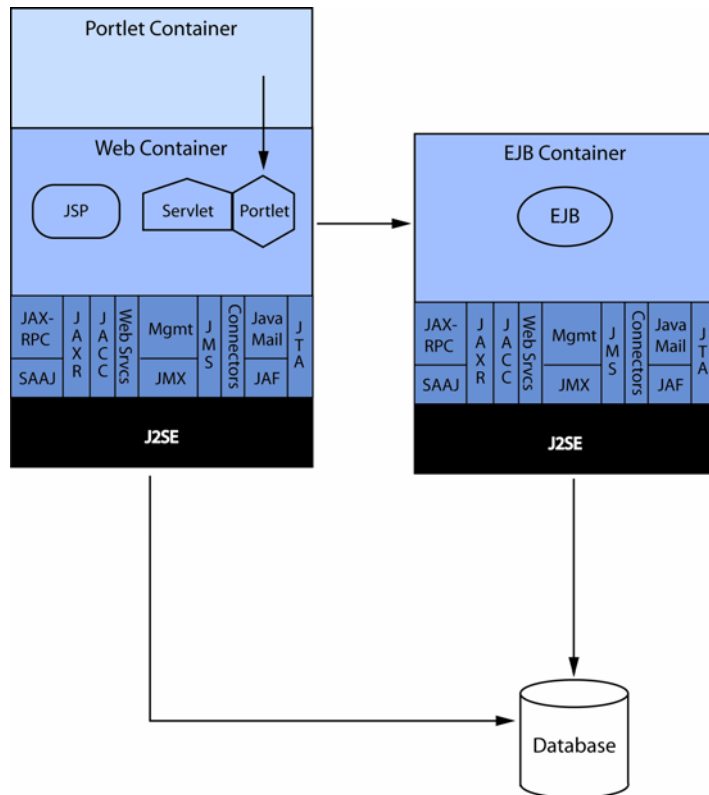


Figure 2.6: How portlets relate to J2EE. Today, while using as many J2EE concepts as possible, portlets are not an integral part of J2EE. Thus the portlet container is running on top of the servlet container.

The plan is to keep future portlet specification aligned with the next J2EE versions. The goal is to integrate the Portlet Specification into J2EE in the future. This would allow treating portlets as first class J2EE citizens and they would be supported by the whole J2EE infrastructure, like application management, monitoring, deployment, and authorization. When this happens, a portlet container will be part of the application server and leverage the server's entire infrastructure, including administration and performance tuning. Portals would then be web applications running on the application server and leveraging the different containers provided by the application server. Figure 2.7 shows the portlet container as an independent container besides the servlet and EJB container. More information concerning the J2EE specification can be found in [2].

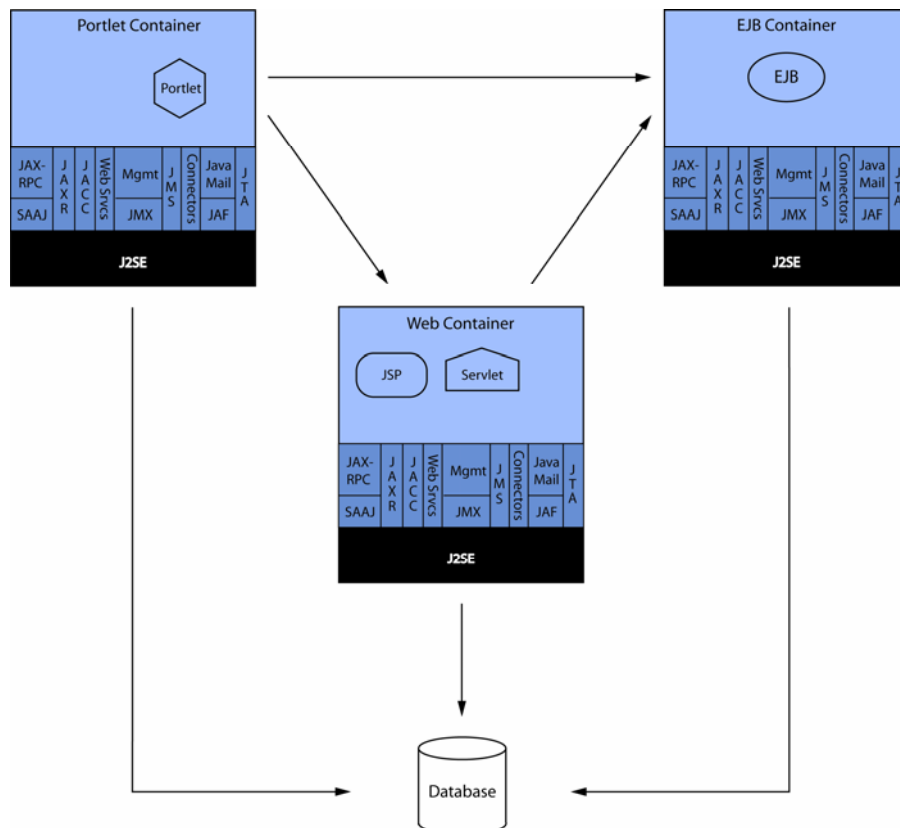


Figure 2.7: Future scenario when portlets are part of J2EE and the portlet container is another J2EE container, like the servlet and EJB containers.

The Java Portlet Specification is also aligned with another upcoming important J2EE technology: Java Server Faces, which enables server-side user interfaces for web components. For more information about JSF and portlets see Chapter 7.

So much for theory. Now it's time to see a portlet in the real world.

2.2.1 Portlets in practice

OK, so what does this definition mean in practice? We've already seen a real life portal page in Figure 2.1, so Figure 2.8 depicts the basic structure of such a portal page. The markup fragments produced by portlets are embedded into a portlet window as portlet content. In addition to the portlet content, the portlet window also has a decoration area that can include the portlet title, controls to influence the window state and the mode. The user can control the size of the portlet window via the portlet window controls, from minimized (only the title is displayed) to normal to maximized (only portlet on the page). The portlet modes influences the requested function of the portlet. A portlet may offer help in a help mode, or allow customizing the behavior in an edit mode.

Several portlet windows may be aggregated by the portal to produce a complete portal page. This means that the portlet only produces the portlet content and the portal produces all the rest visible on the portal page, like the portlet window, the portlet window controls and the layout of the page.

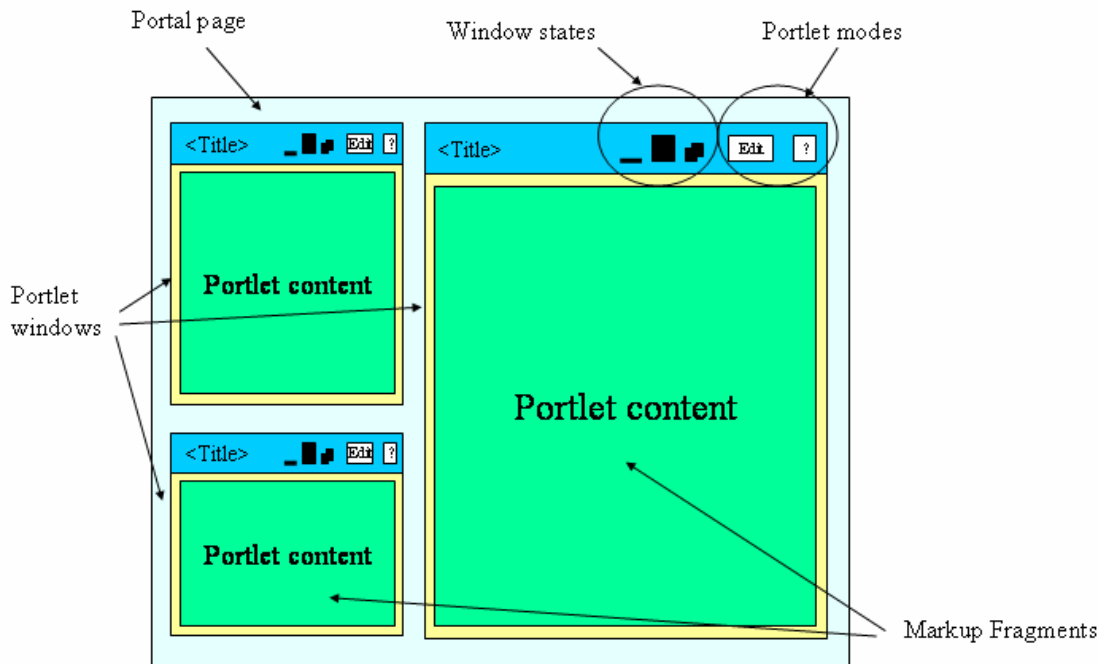


Figure 2.8: Portlet windows on a page have several basic elements: In addition to the markup fragments that provide the information, there is a standard state-change area (minimize/maximize/close), and the Edit Mode option.

Until now all our examples are HTML-based, however portlets are not restricted to HTML. Figure 2.9 depicts an example of a portlet that can produce different markups for different devices: for desktop browsers, it produces HTML markup and for WML devices, like mobile phones, it produces WML markup. This multi-device support offered by portlets allows the users to access the same applications regardless of the device they use.

Now that we know how portlets fit into the portal world, let's take a closer look on how portlets are used in the next section.



Figure 2.9: Examples for different portlet markups: on the left side, a portlet has produced HTML markup in a desktop browser; on the right side, it has produced WML markup on a mobile phone.

2.3 Creating portlets

Until now we have learned a lot about what portals and portlets are. In this section we'll start making portlets. There are three main scenarios for this:

- Creating a new portlet-based application project.
- Migrating a portlet-based application from a proprietary portlet API to the standard portlet API (Java Portlet Specification)
- Transforming an existing web application into a portlet-based application.

We will cover all of these scenarios and start with creating new portlet applications from scratch.

2.3.1 Creating portlet applications from scratch

So, why would someone write portlet-based applications? That's a good question and one that we will answer in this section.

Previously we have defined a portlet as a Java-based Web component that processes requests and generates dynamic content. Thus our question should have been more precisely: why would someone write portlet-based web applications? In the days before portlets appeared the web programming model consisted of servlets and Java Server Pages (JSPs). We will explain this programming model in more detail below, but for now we can concentrate on their major characteristics:

- Adhere to a request/response paradigm. Web applications communicate with the web client by having the client send a request and the web application responds back to the client.
- Self-contained application. A web application comes with all needed components and does not interact with other applications installed on the server running the web application. This means that you get one monolithic, consistent application that solves a specific problem, like an Internet store.

While the first bullet also holds true for portlet applications, the second does not. In fact, this monolithic structure of web applications was the reason why portlets were invented. The monolithic structure of a complete application still has its usage scenarios in a world of portlets, like a self-contained online shop that doesn't need further customizations. However, more and more web applications are becoming portal-like to allow users, as we have said, to work more in a desktop manner with different applications on one screen that can be interconnected and exchange data.

As the demand for more portal-like applications grows, the need to manage multiple applications in a single portal becomes critical. In all likelihood, the many applications that are typically included in a portal are created by more than one provider or group. Therefore each web application must be modular enough to fulfill several new requirements:

- Portlet applications must easily "plug in" to an existing portal to provide the portal with new functionality.
- Portlet applications must adhere to some rules in order to play well together with other portlet applications in the same portal
- The portal must provide a unified user interface across multiple portlet applications.

These kinds of rules and integration points are not provided by the servlet API and therefore the new portlet API was created that supports building modular web applications that can be plugged into portals and produce content that is aggregated with content from other portlet applications into one page.

So the answer to our question at the beginning of this section is: You would write portlet applications if they will be used inside portals or if they need to be modular and integrate with other applications.

But what if you have already created portlets using a proprietary portlet API? Let's consider this.

2.3.2 Moving from proprietary APIs to standard APIs

Here's another problem. Perhaps you were a portlet early adopter, but no standard existed when you started, or you found the first version of the Java Portlet Specification too restrictive. For these reasons, you programmed your portlets against a proprietary portlet API. Now that the standard is in place, it is a good time to move from the proprietary portlet APIs to the standard portlet API. This will make your portlets vendor independent and thus broaden the market for your portlets or allow you to move later on from your current portal vendor to a different one without throwing away all your portlet applications.

If you have written a portlet to a proprietary API, your situation looks like the one depicted in Figure 2.10. The portlet is very tightly coupled to the available portal infrastructure as all the portlet APIs available are specific to the portal it was developed for.

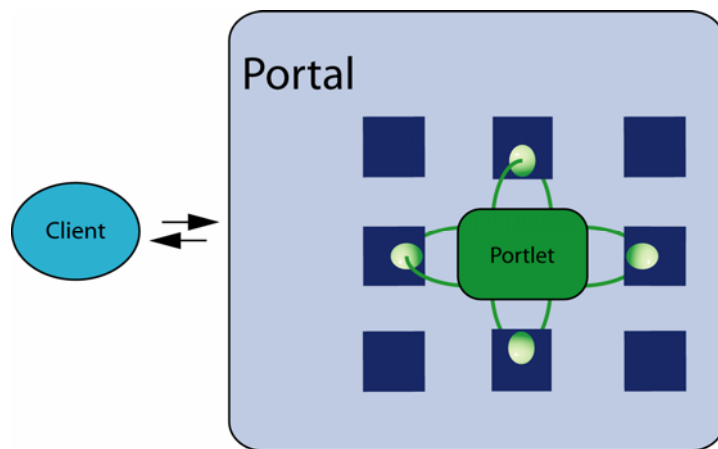


Figure 2.10: Portlets written against a proprietary portlet API are tightly coupled with the surrounding portal infrastructure. The task in migrating them is to open them up.

When moving to the standard API there are two different cases to look at: in the first, the proprietary and standard APIs each support the same feature set; in the second, the proprietary API has a more robust feature set than the standard API..

Migrating supported features

The first one is the easy one. In this case the functionality that the portlet uses in the proprietary API is also available in the standard API and thus the portlet can be re-written against the standard portlet API. This way you end up with a portlet that is completely vendor independent, guaranteed to run unchanged for years as new versions of the portlet standard will be backwards compatible and easier to maintain, as even years from now people will know how to program against V1.0 of the standard portlet API.

Migrating non-supported features

So let's come to the more complex case. In this case your portlet uses some functionality that is currently not available in the standard portlet API, like sending events between portlets. Even if this is the case it may still be beneficial to move to the standard API and only use vendor-specific extensions for the functions that are not available in the standard API. You can then program your portlet to query the portal at runtime for supported extensions. If the portal does not have the extension(s) you want, the portal can still run with degraded functionality.

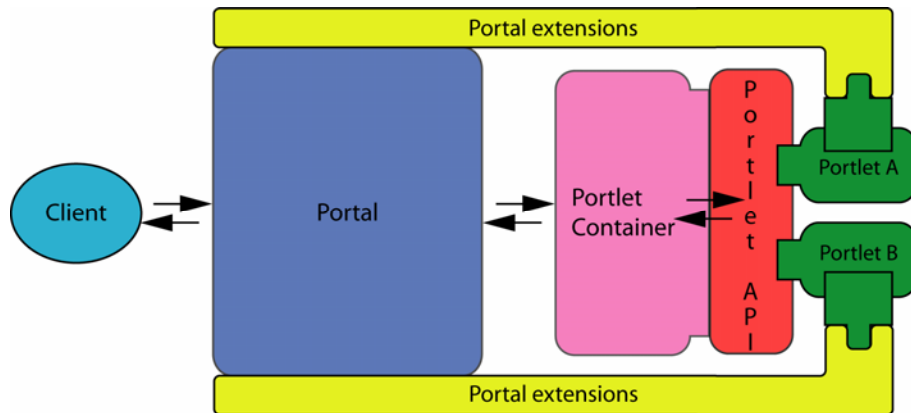


Figure 2.11: Portlets written against the standard portlet API, but use vendor extensions when available are decoupled from the portal and can still run in different portals. If a portlet runs on a standard portal without the vendor extension, it still runs with degraded functionality.

Figure 2.11 depicts such a scenario. As you see, the portlet plugs into two different APIs, the Java Portlet API and the Portal Extension API. Now the portlet is decoupled from the portal infrastructure and can also run in a standard environment and just offer less functionality.

Now that we have covered creating portlet applications from scratch and converting portlets written against proprietary portlet APIs we will take a look at transforming web applications into portlet applications and thus handle the last usage scenario for portlets.

2.3.3 Transforming web applications to portlet applications

This last scenario deals with another common case. You already have an existing, servlet-based web application and now you've read all this exciting stuff about portlets and want to leverage the advantages of portals and portlets. Of course, you would like to avoid throwing away your existing code and starting again from scratch. It would be rather nice to put your existing web application into a tool, where the tool transforms it into a portal application. Unfortunately, reality is not so easy and only limited automated tool support for transforming web applications to portal applications is coming in the latest tool versions.

Let's try to look at the different ways your web application is implemented to see how easy or complicated it is to transform into a portal application.

As with the last scenario, one implementation method is easier to transform than the other. If you were lucky enough to have written your web application from scratch without using any Model-View-Controller (MVC) framework like Struts, there are just a few considerations. It largely depends on how modular your web application is and if you can easily factor out and remove the parts that do not fit in the portlet programming model that we will discuss in the next section. These parts include:

- using HTTP error codes or error JSPs. These parts now need be delegated to the portal.
- mixing of state-changing code and rendering code. If these two are not separated you will have a hard time moving to the portlet programming model (or any MVC-based framework)
- has the code parts that deal with protocol handling and markup selection not cleanly separated, as these parts are handled by the portal in the portlet case.

However, if you stick to the Sun guidelines for J2EE web applications [3], which recommend the MVC pattern or have used any MVC-based framework like Struts it should be a doable effort to transform your web application into a portlet application. You still have only done the first step with this, as your transformed portlet application most likely does not leverage the full power that the portlet programming model provides. Using the complete data model the portlet programming model provides will take some effort, but will make your portlet application faster and easier to use.

But what is this portlet programming model? In this next section, you will learn more. We'll give you even more details on the portlet programming model and the data model in Chapter 3.

2.4 Going beyond servlets: the portlet architecture

“OK, we may need the idea of portlets,” you may ask, “but why do we need to have a standard defining this new portlet component?” With the emergence of an increasing number of enterprise portals, various vendors have created different APIs for portlets. This variety of incompatible interfaces generates problems for application providers, portal customers, and portal server vendors. Portlets written to one portal server could not run on other portal servers from other vendors. This means that application providers need to rewrite their applications for each portal product he would like to support. This very much slowed down the portal development market, as large companies were very hesitant to invest large amounts of money in such a proprietary environment.

To overcome these problems, Java Specification Request 168 (JSR 168), the Java Portlet Specification, was started to provide interoperability between portlets and portals. The JSR 168 process was co-led by IBM and Sun and had a large Expert Group that helped to create the final version now available. This Expert Group consisted of Apache Software Foundation, Art Technology Group Inc. (ATG), BEA, Boeing, Borland, Citrix Systems, Fujitsu, Hitachi, IBM, Novell, Oracle, SAP, SAS Institute, Sun, Sybase, Tibco, and Vignette. As you can see by that impressive list, the Java Portlet Specification was quickly and widely supported. Today many commercial and open source implementations supporting JSR 168 are available (see Appendix A). Pluto, the reference implementation of JSR 168, consists of a simple portal implementation in order to allow portlet developer to test their portlets. Pluto is discussed in detail in Chapter 7.

So, what exactly does the Java Portlet Specification define? Does it define a whole portal system? No, actually the JSR 168 defines a contract between the portlet container and the portlet, including a portlet API and the packaging and deployment of portlets. Everything that lies beyond the portlet container, like the portal system with its aggregation engine is not part of JSR 168 (see Chapter 6 for how portal architectures look like).

In this section, you will learn about portlet architecture, which ranges from the basics of the portlet specification, like the portlet lifecycle and the request/response flow, to the portlet data model, and finally packaging and deployment of portlets.

2.4.1 Starting with the basics

So let us start with the basic requirements for a portlet that are not present in the servlet web programming world. The Java Portlet Specification defines the following basic capabilities:

- A portlet must support the lifecycle, specifying the request / response flow and the lifecycle of a portlet
- A portlet must support modes and window states, defining the function and available real estate for a portlet window
- A portlet must support several customization levels, allowing customizing of portlets on different levels

Let's take a closer look at each of these requirements.

Portlet lifecycle

Every portlet must implement the portlet interface, or extend a class that implements the portlet interface. The portlet interface defines the basic portlet life cycle. This lifecycle consists of:

1. initializing the portlet and putting the portlet into service (init method)
2. responding to requests from the portlet container (processAction and render methods)
3. destruction of the portlet when the portlet needs to be put out of service (destroy method)

The portlet container manages the portlet life cycle and calls the corresponding methods for each life cycle on the portlet interface. The request handling in the second step of the lifecycle is divided into two phases:

- action handling – the portlet container calls the portlet’s processAction method to notify the portlet that the user has triggered an action on this portlet. Only one action per client request is triggered. During an action, a portlet can issue a redirect, change its portlet mode or window state, or modify its state.
- rendering – the portlet container calls the portlet’s render method to request the markup fragment from the portlet. For each portlet on the current page, the render method is called, and the portlet can produce markup that may depend on the portlet mode or window state or other state information or backend data. The portal attaches a list of valid MIME types that must have at least one element, which the portlet can choose for rendering the markup. The MIME types that the portal sends the portlet must be a subset of the MIME types that the portlet has declared as supported in the deployment descriptor. The portlet sets the chosen MIME type on the response and generates the markup either by directly writing to the output stream or by including a servlet or JSP that generates the markup.

Figure 2.12 depicts these different request categories and shows the difference between the action call that is only issued to the portlet that the user interacts with and the render calls that are issued to all portlets on the current page.

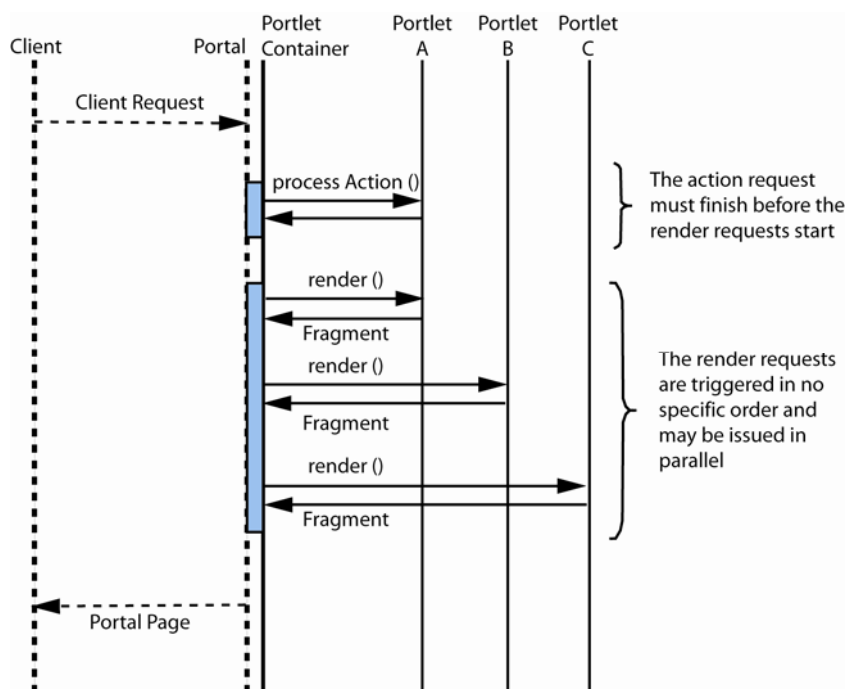


Figure 2.12: Portlet request flow for rendering one page: the user has triggered an action on portlet A, which results in a processAction call for portlet A and the portal page for this user consists of portlets A, B, C, which results in render calls for all these portlets in order to produce the page markup.

Figure 2.12 also depicts the parts of the request flow that are defined within the Java Portlet Specification, namely, as mentioned before, the interaction between the portlet container and the portlets, and the parts that are not defined in the specification (dashed lines): the interaction of the client with the portal and the interaction between the portal and the portlet container.

Here is the actual portlet interface with all its methods:

Listing 2.1 Portlet interface

```
public interface Portlet
{
    public void init(PortletConfig config) throws PortletException;

    public void processAction (ActionRequest request, ActionResponse
        response)      throws PortletException, java.io.IOException;

    public void render (RenderRequest request, RenderResponse
        response)      throws PortletException, java.io.IOException;

    public void destroy();
}
```

As you can see it really is a simple interface and consists of even fewer methods than the servlet interface! Now that we know the basic lifecycle of the portlet, let us take a look at what portlet modes and window states are good for.

Portlet mode and window state

The portlet mode advises the portlet what task it should perform and what content it should generate. This is the function it will perform for the current request. As explained before, this additional information is needed because the portlet does not directly interact with the user, but the portal is between the end user and portlet. Through these portlet modes and window states the portal can enforce a consistent look and feel for all portlets on a page.

Usually, portlets execute different tasks and create different content depending on the functions they are currently performing. When invoking a portlet, the portlet container provides the current portlet mode to the portlet. Portlets can also programmatically change their mode when processing an action request.

There are three different categories of portlet modes:

- **required modes** are modes every portal must support. These modes are Edit, Help, and View. A portlet must support the View mode used to render markup for a page. The Edit mode is used to change per-user portlet preferences to customize the portlet markup, and the Help mode is used to show a help screen.
- **optional custom modes** are modes that are listed in the appendix of the portlet specification and that a portal may support. However, as these are optional portlet modes, portlets cannot count on being called in one of these modes. Therefore if your portlet supports an optional portlet mode your portlet should still work if it is never being called in this mode. The optional modes include the About mode to display an "about" message, the Config mode letting administrators configure the portlet, the Edit_defaults mode lets an administrator preset the Edit mode's values, the Preview mode enables a preview of the portlet and the Print mode renders a view that can easily be printed.
- **portal vendor-specific modes** are not defined in the portlet specification.

A window state indicates the amount of portal page space that will be assigned to the content generated by the portlet. When invoking a portlet, the portlet container provides the current window state to the portlet. The portlet may use the window state to decide how much information it should render. Portlets can also programmatically change their window state when processing an action request. The following window states are defined:

- **Normal** – indicates that a portlet may share the page with other portlets or that the portlet is rendered on a device with a small screen. This is the default window state.

- Maximized – indicates that a portlet may be the only portlet on the portal page or that the portlet has more space compared to other portlets in the portal page and can therefore produce richer content than in a normal window state.
- Minimized – indicates that the portlet should only render minimal output or no output at all.

A portlet can be called in any of these three window states, but is free to produce the same markup for all three states. In addition to these window states, the portal may also define vendor-specific window states.

Customization levels

The beauty of a portlet is that the developer doesn't impose his idea of what he thinks the user wants on the user. Portlets are designed to be customized. Normally users customize the portlets in order to get content for their specific needs. For example, the user can customize a stock quote portlet to only show desired stock quotes, or a news portlet to just show the news topics of interest.

In Java there are different ways to customize a Java object. The simplest way is to create a copy of that object and customize the new object. This has the advantage for the programmer that it is easy to understand and that the object can store data internally in instance variables. However, it also has a couple disadvantages: creating a new object is an expensive operation since it requires memory, and you cannot inherit settings from the original object. Inheritance of settings plays an important role as we will see below.

To provide scalable portals for many millions of users, portlets use the flyweight pattern (see [4]), to implement customization. In this pattern, only one very light Java object exists and the different data sets are passed into the object. There exists only one Java portlet instance, even if that portlet is used by a million users with different customization settings for this portlet and inheritance of different levels of customization data can easily be implemented. However, as always in life, this comes at some cost: now the portlet must store the customization data separately and cannot store these data in instance variables. The Java Portlet API provides the `PortletPreferences` in order to allow the portlet to store these data. The portlet also needs to be programmed thread-safe as several threads may call the same portlet code at the same time to generate markup for different users.

You see that making portlets customizable has very dramatic consequences on the portlet programming model. The next issue is at which levels portlets can be customized.

Figure 2.13 depicts the different levels that influence the data that the portlet operates on. Besides the `portlet.xml` that serves as the initial seed there are three levels: the portlet definition, the portlet entity and the portlet window. It is important that data between these levels are inherited, which means, for instance, that a change in the portlet definition will be visible in all portlet entities created from this definition. As you may remember, this is one advantage that the flyweight pattern offers: the data are not stored in instance variables but outside the portlet and can be merged by the portal infrastructure easily without being visible to the portlet. Several of these portlet definitions may be created out of one portlet definition in `portlet.xml` and the administrator may set different configuration settings and thus provide new inheritance trees.

Typically a new portlet entity is created out of a portlet definition for each user to represent the user-specific preference data. As these portlet entities may show up on one or more pages of the user, one or more portlet windows are created pointing to this portlet entity. So we have the portlet definition of News, and each user selects the sources or types of news, which are stored in the portlet entity. These news items can appear in different portlet windows on different pages as the user explores the site.

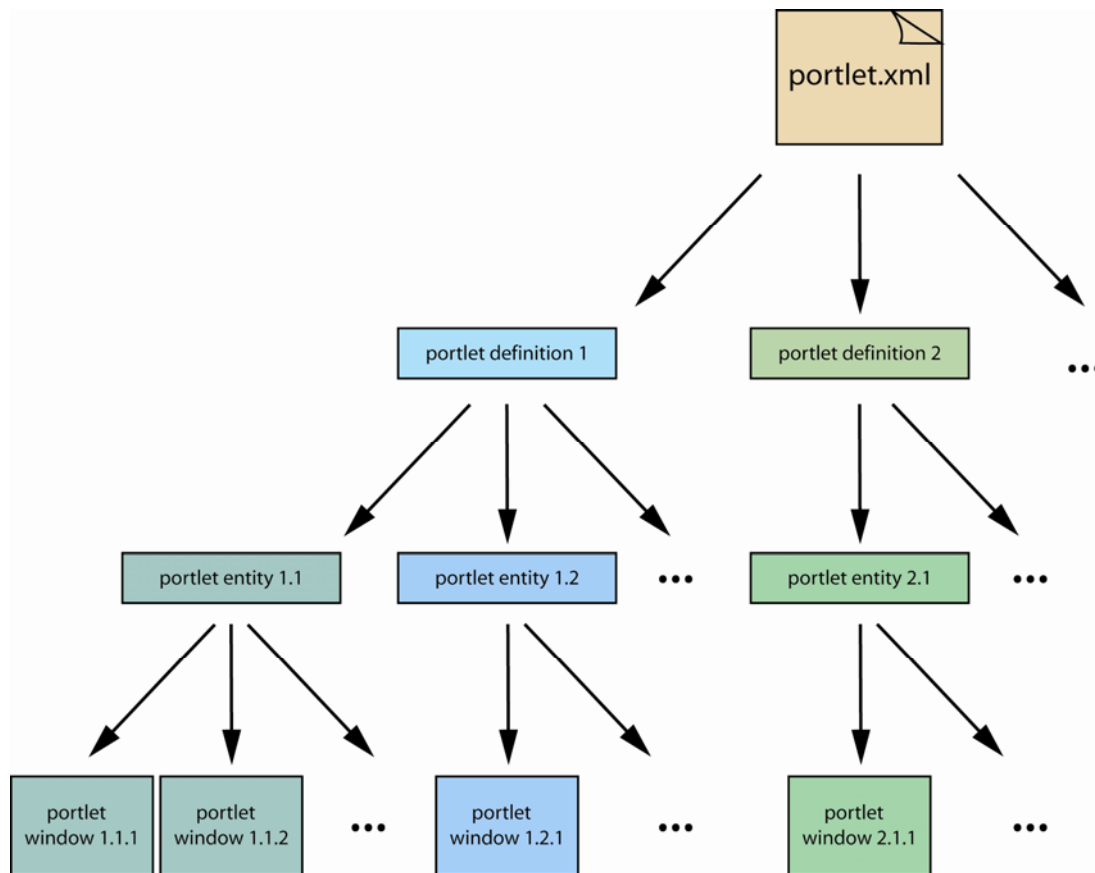


Figure 2.13: Relations between the different portlet customization levels: Each level (portlet.xml, portlet definition, portlet entity, and portlet window) can define or provide context for the next level down. Each level inherits data from its next level up.

Now how do these different levels of customization show up in the portlet? Each level has a different representation in the portlet:

- Portlet definition – provides the portal administrator with the ability to customize portlet settings, like the stock quote server of a stocks portlet. The settings for the portlet definition can therefore be changed by the administrator in the optional Config mode. The portlet can access these settings via the read-only preferences. They are called read-only, because they cannot be changed outside of the Config mode and are thus read-only for users that are not administrators. A typical example of such a setting is a server address of a news portlet where the portlet retrieves its news. More details on the portlet preferences are explained below in the data model section.
- Portlet entity – provides the end-user with the ability to customize the portlet. New entities are created for each user that puts a portlet on her/his page. The user can then further customize the writable portlet preferences in the Edit mode. An example for a portlet entity setting is the news groups of a news portlet that a specific user is interested in.
- Portlet window – is the window in which the markup fragment that the portlet produces is displayed. A portlet window is created whenever a user attaches a portlet entity to a specific portal page. This decoupling of the portlet window from the portlet entity allows having different windows pointing to the same portlet entity. This enables the user having the same news portlet or calendar portlet on different pages. The portlet window also comes with data (navigational states) attached to it that enable the portlet to render a specific view of the data. Navigational data is represented in the portlet API in the portlet mode, the window state and the render parameters. Contrary to the previously mentioned customization levels this one normally is not stored persistently, which means that if a user logs on again the portlet may not be in the same portlet mode or window state as the last time the user was logged in.

We have already started to get into the data model and which kind of data the portlet has access to. The data model is a very important part of the portlet programming model so let's take a closer look at it.

2.4.2 Understanding the Portlet data model

As we have seen in the previous section the portlet has access to different data sets that are configured by different user roles: the administrator and the end user. It is very important to understand this data model when you are writing portlets, as otherwise you could confuse administrators and end users if the portlet does not adhere to this data model. For example, if the news portlet provided a section in the View mode in which the news server can be changed, this would lead to an inconsistent user experience, as the administrator would look for a Config button on the portlet window and would not find one. Thus she would conclude that she cannot customize this portlet as she would not see the usual Config button at the usual place in the portlet window. The portlet data model clearly separates the different roles of administrators and end users and thus allows the portal to create different, but consistent interfaces for each user type. Portlets that do not adhere to this pre-defined schema, like the example mentioned, would thus create their own user interface for administrators and would break the user experience of the whole portal.

This section describes in detail the data model provided by the Java Portlet Specification that the portlet should use for storing and retrieving customization data. All the customization data together defines the state of the portlet. As you may remember from the last section, the portlet programming model uses the flyweight pattern, where the Java object itself does not store any state, but for each processing request gets the data from outside. We will cover both the persistent and transient state that the Portlet Specification defines.

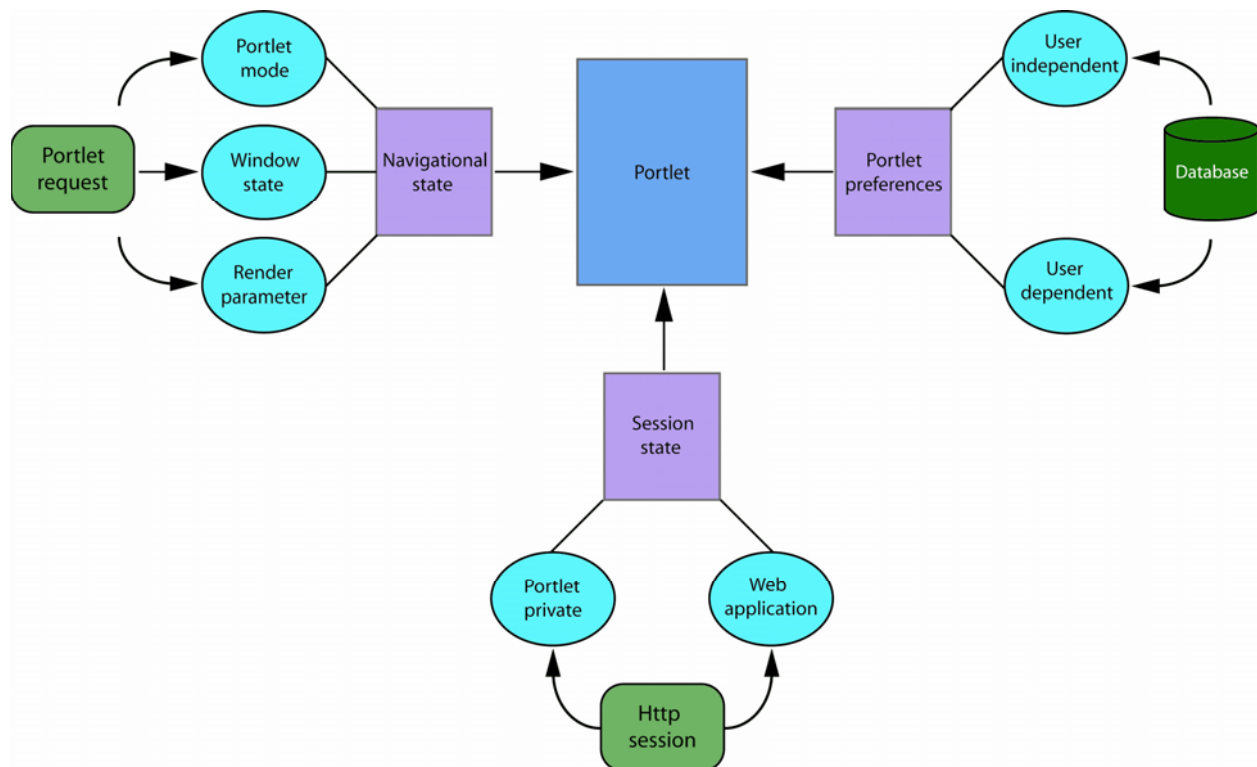


Figure 2.14 The portlet data model shows different types of state information that a portlet has access to. This includes the transient navigational and session state and the persistent portlet preferences. The portlet uses all this information to display a consistent, personalized user interface.

Figure 2.14 summarizes all these different states that the portlet can access in one picture: the persistent state on the right that is available to the portlet via the portlet preferences and the transient state that has a part that depends on the request and one that is session dependent. We will explain this figure in more detail in the next sections.

Persistent state

The portlet can access two different types of persistent data, the initialization parameters and the portlet preferences. The initialization parameters are read-only data that can be specified in the portlet deployment descriptor and are the same for all portlet entities created from this portlet definition. They can be used to set basic portlet parameters, like the names of the JSPs that are used to render the output.

The portlet can also access persistent data via the portlet preferences. There are two different categories of portlet preferences:

1. *user-independent (Administrative) preferences* – these preferences are declared in the portlet deployment descriptor as read-only and cannot be changed by the user. Only an administrator can change these settings, e.g. by using the portlet Config mode or an external tool. Examples for this kind of data include server settings and billing information. All user-independent preferences must be listed in the deployment descriptor as read-only preferences.
2. *user-dependent preferences* – these preferences can be changed by the user to customize the portlet. This is normally done using the Edit portlet mode. Examples for this kind of data include the companies a user is interested in seeing stock quotes for or topics the user would like to get news for. New user-dependent preferences can be created by the portlet at run-time.

Both kinds of preferences are merged into one interface that offers this data to the portlet as can be seen on the right side of Figure 2.14. Keep in mind that preferences can only be read and written in the action phase, and read in the render phase. This restriction was introduced in order to enforce the programming model that changes to the portlet state must be done in the action phase and the render phase is idempotent and replayable. The portlet preferences can be either strings or string array values associated with a key of type String. Preferences can be preset with default values in the deployment descriptor.

In order to validate that the portlet preferences conform to application specific restrictions, like number of news articles per page needs to be greater zero, the portlet should specify a preference validator. A preference validator is a class that implements the PreferencesValidator interface and is called automatically by the portlet container each time before any changes are made persistent. Supplying the validator as a separate class has the advantage that other tools and generic administration portlets can change the preferences of the portlet and the validator is called each time to ensure that the preferences are consistent. The portlet needs to define in the deployment descriptor the class that implements the PreferencesValidator interface under the tag preference-validator (see the deployment descriptor explanation below).

Transient state

The portlet has access to two different kinds of transient state: session state and navigational state. This distinction reflects different lifetimes of the data stored in these states. The session state reflects the lifetime of a user session, where a user logs on to a portal, performs several operations and finally after some time logs out or is automatically logged out due to a long period of inactivity. The navigational state represents a more short-term store that is bound to the request and may change from one request to the next. Therefore, the portlet accesses these different states through different APIs, as can be seen in Figure 2.14 on the left and bottom.

The session state is state that is available for the portlet per portlet application. The session concept is based on the HttpSession defined for Web applications. Portlet applications are Web applications and thus use the same session as servlets. In order to allow portlets storing temporary data private to a portlet entity, the

default session scope is the portlet scope. In this scope, the portlet can store information needed across user requests that are specific to a portlet entity. Attributes stored with this scope are prefixed in the session by the portlet container to avoid two portlets (or two entities of the same portlet definition) overwriting each other's settings. The second scope is the Web application session scope. In this scope, every component of the Web application can access the information. The information can be used to share transient state among different components of the same Web application (e.g., between portlets, or between a portlet and a servlet).

The navigational state defines how the current view of the portlet is rendered and is specific to the portlet window that the portlet is rendered in. Navigational state can consist of data like the portlet mode, window state, a sub-screen id, etc. The navigational state, unlike portlet mode and window state, is represented in the portlet API via the render parameters. The portlet receives the render parameters for each render call and they can only be changed in an action or via clicking on a render link with new render parameters. The portlet receives these parameters at least as long as the user interacts with the current page. After executing a bookmark or switching to another portal page and returning to the old portal page the portlet may not get the old render parameters anymore. This behavior depends on the portal implementation and is not specified in the Java Portlet Specification. Therefore, as portlet programmer, you should expect such behavior in your portlet code and let the portlet act gracefully if all previous render parameters are lost in the next request.

2.4.3 Programming patterns for portlets: the Model-View-Controller pattern

We now know what portlets look like, that they have different phase processing actions and rendering content, that they have portlet modes and window states and a quite complex data model for storing and retrieving data. Now what are the programming patterns for portlets so we can start writing them?

Implementing application logic and presentation markup together

The simplest way to implement portlets that generate dynamic content is by generating content completely inside the portlet (see Figure 2.15). So, how would the client request be processed in this pattern implementation?

The client sends a request (1) to the portal server. The portal server identifies the portlets on this page and sends a request for each portlet to the portlet container (2). Now the portlet container sends a request to the targeted portlet (3). The portlet executes the controller logic and responds back to the portlet container (4). The portlet container generates a response to the portal based on the portlet response (5) and finally the portal creates a response that is sent back to the client (6) consisting of the whole portal page with the content of all aggregated portlets.

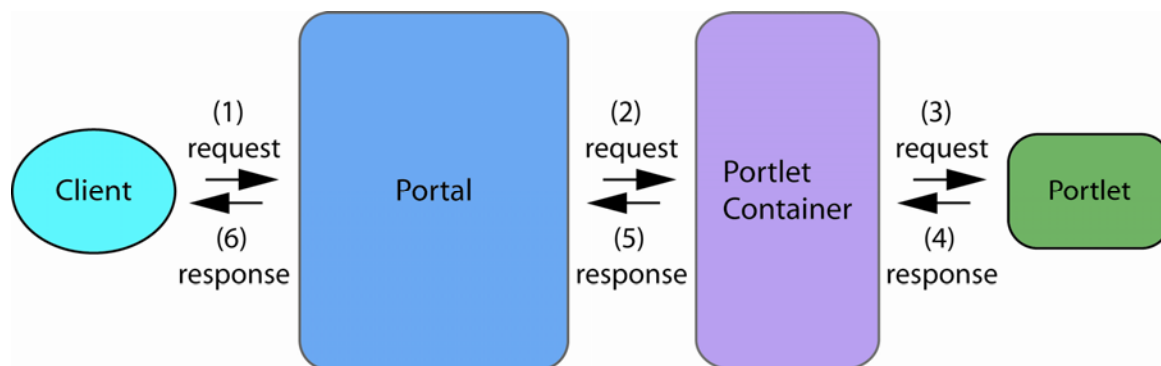


Figure 2.15: Simple portlet generating the markup in the portlet. Requests travel from the client (browser), through the portal and portlet container to the portlet itself. The portlet uses the data model to respond to the client's request, providing a consistent interface through the portal.

This concept of implementing application logic and markup generation in one piece of code is appropriate only for very simple applications. It is too inflexible for real-life applications and does not provide appropriate separation of responsibilities between Java programmers and content designers.

Separating logic and markup

Fortunately there is a pattern that is still quite simple and resolves many of the shortcomings of the simple solution: the Model-View-Controller (MVC) pattern. The MVC pattern depicted in Figure 2.16 offers much more flexibility and provides a clear separation of responsibilities. Here is how the request is processed with this pattern:

The portlet now acts as controller, receiving all incoming requests (3) and controls their execution. The new state that results in this business logic execution is stored in the model (4). The model, normally a Java bean, is responsible for storing the application data needed to produce the view. The controller then calls the appropriate view to create the response (5). The view, in most cases a Java Server Page (JSP), is therefore responsible for generating the markup and uses the state stored in the model in order to create state dependent output (6).

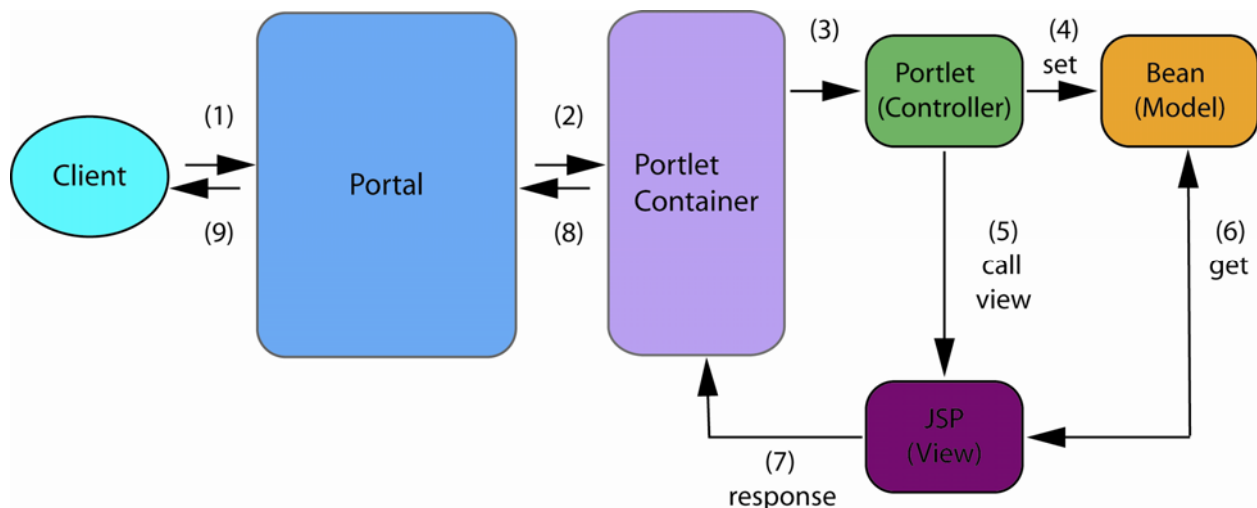


Figure 2.16: For all but the most simple client requests, portlets should generate markup using the Model-View-Controller pattern. Here the portlet controls the process where a Java bean holds the data model, and a Java Server Page generates the markup.

This separation of responsibilities allows programming and maintaining the controller code and the view code independently and by different persons (e.g. the portlet programmer and the content designer).

The Model-View-Controller pattern has proven its usefulness for quite some time now in the servlet world, and the Java Portlet Specification has decided to design the MVC pattern directly into the portlet programming model. Portlets support this pattern in various ways, like the distinction between action processing and rendering, allowing state changes only in action, the ability to include JSPs, and a JSP tag library.

The separation between action processing and rendering allows encapsulating state changes in the action processing and only performing the view selection in the render method. In the action processing phase the portlet does not have access to the output stream and therefore is unable to generate content. In the action phase the portlet can set new persistent state, e.g. via the preferences, new transient state, e.g. portlet mode, window state, render parameter, session state, or can forward the action processing to another web resource.

In the render phase the portlet has access to the output stream in order to produce the markup; however the access to state information, like portlet preferences or the navigational state, is now read-only. In order to

delegate the rendering of the content to JSPs the portlet API provides a request dispatcher allowing the portlet to include servlets and JSPs.

JSPs can access the portlet-specific objects and data via the portlet tag library that the Java Portlet Specification defines. These tags give access to the render request and response, the portlet context and allow creating action and render links via tags.

In Section 3.1, we will provide a concrete portlet example and explain it in detail. More information about the MVC pattern can be found in [5]; the JSP specification can be found at [6] and further information on portlet frameworks is provided in chapter 7.

2.4.4 Packaging and deploying portlets

Now that we have covered all the programming model parts of portlets there is one thing left that stands between us and a running portlet on a portal server: somehow we need to package all our portlet code, JSPs and other resources we have created together into one unit that we can then deploy on a portal server. So how is this packaging done?

A portlet application's resources, portlets and deployment descriptors are packaged together in one Web application ARchive (WAR file). This is a very convenient thing, as WAR files are well known from the servlet world. This means that there are already many tools out there that will help you to generate such WAR files. In contrast to servlet-only Web applications, portlet applications consist of two deployment descriptors: one to specify the Web application resources, the `web.xml`, and one to specify the portlet resources, `portlet.xml`. All Web resources that are not portlets must be specified in the `web.xml` deployment descriptor and all portlets and portlet-related settings must be specified in `portlet.xml`. As a result of the two deployment descriptor files, portlet application deployment is a two-step deployment process that deploys the Web application into the application server and the portlets into the portal server. The details of this deployment procedure are vendor-specific.

Figure 2.17 and 2.18 depict the schema of the `portlet.xml` file. Figure 2.17 shows the settings that can be applied on the application level. These settings include the support of custom portlet modes or window states, user attributes (like the first or last name of the user, see also chapter PLT.4 [1] for the most common user attribute names) and security constraints when accessing the portlet (like always use a SSL connection when accessing this portlet).

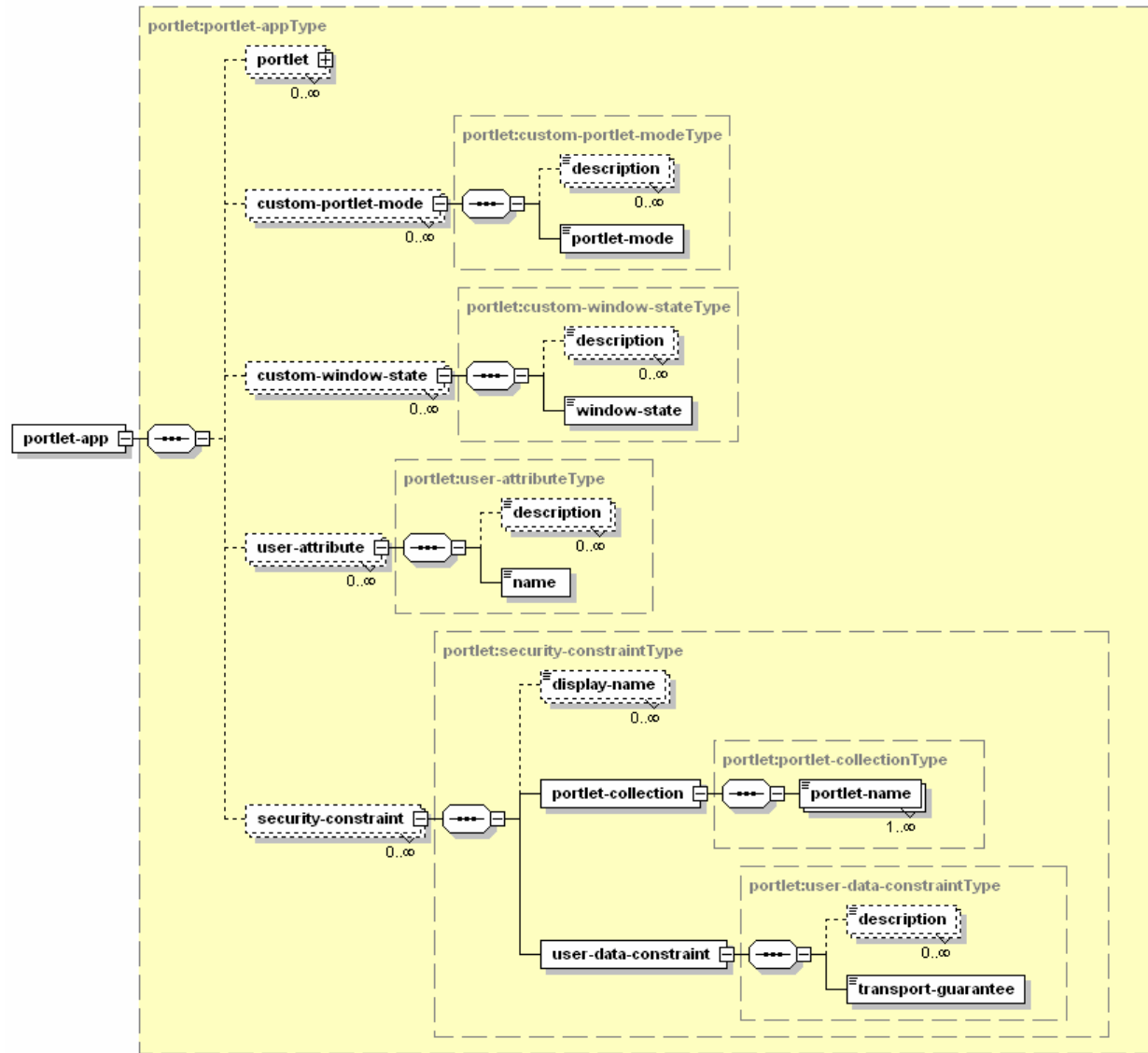


Figure 2.17: First part of the deployment descriptor schema showing the application level settings. These include custom portlet modes and window states, user attributes, and security constraints

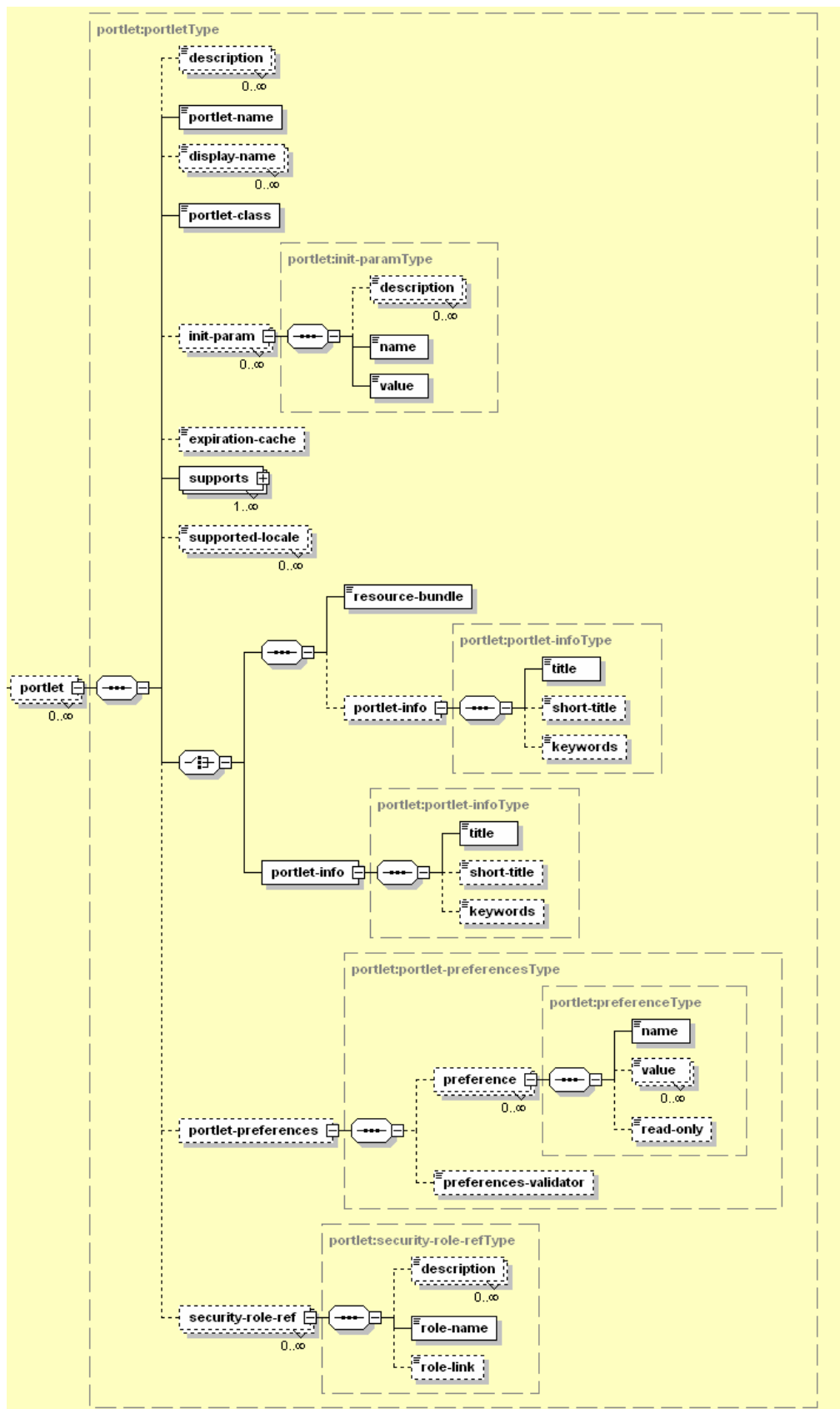


Figure 2.18: Second part of the deployment descriptor schema showing the portlet-level settings. These settings include the portlet's name and class, initialization parameters, cache settings, supported MIME types, locales, persistent preferences and J2EE roles.

Figure 2.18 depicts the settings on the portlet level, which include name and class of the portlet, initialization parameters, cache settings, supported markup MIME types (e.g. text/html) the portlet can produce, supported locales, persistent preferences the portlet uses, and J2EE roles the portlet accesses. Luckily tools, like Eclipse, are available that will generate the portlet deployment descriptor for you as we will see in Chapter 4.

2.5 Running Java portlets remotely

Running portlets on remote systems has a lot of benefits, ranging from distributing the work load, to allowing departments that contribute content to the overall company portal to run and manage their own portal system.

What do I need in order to run my Java portlets on remote machines? Web Service for Remote Portlets (WSRP, see [7]) is the answer, as WSRP provides the capability to aggregate content produced by portlets which are running on remote machines using different programming environments like J2EE and .NET. WSRP services are presentation-oriented, user-facing web services that plug and play with portals or other applications. WSRP is covered in more detail in chapters 9 and 10 of this book; this section will only show the alignment between the concepts of JSR 168 and WSRP (Table 2.3). The close alignment of the specifications allows JSR portlets to act as WSRP services without requiring you to specifically program the portlets for WSRP. This means you get your remote portlets for free once you have a JSR 168 portlet written! It is just an administrative task in the portal to publish a JSR 168 portlet as WSRP service.

Table 2.3: Alignment of the major programming concepts between WSRP and JSR 168

Concept	WSRP	JSR 168	Comment
Portlet Mode: indicates portlet in which mode to operate for a given request	View, edit, help + custom modes	view, edit, help + custom modes	full alignment
Window State: the state of the window in which the portlet output will be displayed	minimized, normal, maximized, solo + custom window states	minimized, normal, maximized + custom window states	full alignment ("solo" is missing in the JSR, but can be implemented as a custom state)
URL encoding to allow re-writing URLs created by the portlet	defines how to create URLs to allow re-writing of the URLs either on consumer or producer side	encapsulates URL creation via a Java object	full alignment (the implementation of the Java object can implement the WSRP URL rewriting rules)
Namespace encoding to avoid several portlets on a page conflicting with each other	defines namespace prefixes for consumer and producer side namespacing	provides a Java method to namespace a String	full alignment (the JSR namespace method can implement the WSRP namespace behavior)
User – portlet interaction operations	performBlockingInteraction: blocking action processing getMarkup: render the markup	action: blocking action processing render: render the markup	full alignment (action invocations carried through performBlockingInteraction, render carried through getMarkup)
View state allows the current portlet fragment to be correctly displayed in sub-subsequent render calls	navigational state	render parameter	full alignment (WSRP navigational state maps to JSR render parameters)
Storing transient state across request	session state concept implemented via a sessionId	utilizes the Http web application session	full alignment (the WSRP sessionId can be used to reference the JSR session)
Storing persistent state to personalize the rendering of the portlet	allows properties of arbitrary types	provides String-based preferences	full alignment (JSR String preferences can be mapped to WSRP properties)

Information about the portal calling the portlet	RegistrationData provide information of the consumer to the producer	PortalContext provide a Java interface to access information about the portal calling the portlet	full alignment (all data represented through the PortalContext to the JSR portlet are available in the RegistrationData)
--	--	---	--

Table 2.3 shows important concepts in the portlet space and how they are realized by both the WSRP and JSR specification. As can be seen from this table there is a mapping of all these concepts between JSR 168 and WSRP. Due to this alignment it is also possible to include any WSRP service via a JSR 168 proxy portlet into a JSR 168 compliant portal. This way a J2EE portal can easily integrate portlet services from other platforms like .NET. Both of these abilities, to integrate WSRP services via a JSR 168 proxy portlet and to publish any JSR 168 portlet as WSRP service are show later on in this book.

2.6 Summary

In this chapter, we covered the details of portals and portlets. First we explained what a portal is and showed different applications of portals. The main point here is that portals are an integration point that integrates other applications into one consistent end-user application, so that users can work more in a desktop manner,

Next, we covered the role that portlets play in the portal environment, noting they are the central UI components that are rendered by the portal and that allow developers to extend the portal. We covered how portlets currently fit into existing J2EE architectures, why servlets are not enough to provide portal components and how portlets may be even more tightly integrated in future versions of J2EE. If portlets would become part of J2EE this would be a major achievement and would put portlets on par with servlets and enable portlets to leverage the complete J2EE infrastructure seamlessly.

We covered the technologies used for developing portlets. We explored three different scenarios how you would be able to create a portlet application: you can start from scratch and develop your portlet application using the samples and best practices of this book. Or you can take an existing portlet application that is written to a proprietary portlet API and migrate this portlet application to the standard Java Portlet API. And finally you could take an existing web application that was written using UI frameworks like Struts and migrate them to portlet applications.

Finally, we introduced the portlet architecture with its basic concepts and the major portlet programming pattern: the Model-View-Controller pattern.

After covering the programming model, the packaging and data model in detail we also briefly explained how to get remote WSRP portlets out of your Java portlets.

Now that we have laid down all the groundwork let's get real in the next chapter by developing portlets that are more complex than the simple Hello World portlet in Chapter 1.

2.7 References

- [1] A. Abdelnur, S. Hepper: JSR 168: The Java Portlet Specification, URL: <http://jcp.org/en/jsr/detail?id=168>.
- [2] Java 2 Platform Enterprise Edition Specification, v1.4, URL: <http://www.jcp.org/en/jsr/detail?id=151>
- [3] Designing Enterprise Applications with the J2EE Platform, Second Edition, URL: http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/index.html
- [4] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Addison-Wesley, 1995
- [5] Java BluePrints, Model-View-Controller, URL: <http://java.sun.com/blueprints/patterns/MVC-detailed.html>
- [6] D. Coward, Y. Yoshida: Java Server Pages 2.0 Specification; URL: <http://www.jcp.org/en/jsr/detail?id=152>
- [7] WSRP 1.0 standard specification document: <http://www.oasis-open.org/committees/download.php/3343/oasis-200304-wsrp-specification-1.0.pdf>

Further readings

- D. K. Fields, M. A. Kolb, S. Bayern: Web Development with JavaServer Pages, 2nd Edition, 2001, Manning Publications
- J. Falkner, K. Jones: Servlets and JavaServer Pages: The J2EE Technology Web Tier; URL: <http://www.theserverside.com/books/addisonwesley/ServletsJSP/index.tss>

Chapter 3 Building Portlets by example

Now that we have written our first simple Hello World portlet in Chapter 1 and taken a look in Chapter 2 at the overall environment to see what a portal is and on which technologies a portal is built upon, let's turn our attention again to the smallest component that makes portals so special: the portlet. Portlets are the cornerstone of every portal and the component that the end user interacts with most. Now let's get our hands covered with some more dirt and create some complete and more function-rich portlets.

In this chapter, we'll build two example portlet applications: the first is a simple Bookmark portlet and the second is a more complex Calendar portlet. The Bookmark portlet displays a list of user-defined links to take the user to other places on the Web. The Calendar portlet application actually consists of two portlets, one showing a calendar and another one displaying todo's of the day based on the day selected in the calendar portlet. We also discuss our favorite portlet best practices. The chapter ends with a section about the outlook for the future: where portlets are heading.

After reading this chapter, we hope to have you convinced that portlets are the next step in web application programming beyond servlets, as they enable modular and user-centric web applications.

3.1 The Bookmark portlet example

When we look under the hood you can see how simple it is to write a portlet; you just need an editor, a Java compiler and the JSR 168 reference implementation (Pluto) to run the portlet. As an example, let's create a portlet that shows user-defined bookmarks. The bookmark portlet shows a list of bookmarks in its View mode that you can click on, and the browser will take you to the bookmark destination. In Edit mode you can add new bookmarks or remove current ones.

Table 3.1 shows the different tasks that we need to perform in order to create our Bookmark portlet.

Table 3.1 Making a Bookmark Portlet
Bookmark Portlet Task

Writing Portlet Code

Writing JSP Code

Including Deployment Descriptors and Resource Bundles

Running the Portlet

How To

Implement action processing to store or remove new bookmarks
Generating Markup by including the JSPs for rendering the View mode displaying the available bookmarks, or the Edit mode allowing to add or remove bookmarks
Accessing Portlet Preferences in order to read the current bookmarks
Displaying Bookmarks in View mode
Managing Bookmarks in Edit mode
The web application descriptor, web.xml
The portlet descriptor, portlet.xml
Resource Bundles
Displaying Bookmarks
Editing Bookmarks
Changing the default language

More specifically, we will:

- use JSPs for rendering the markup and the portlet preferences to store the data, implementing the MVC pattern, as we have separated the controller (portlet) from the model (portlet preferences) and from the view (JSPs).
- customize the portlet output taking user-specific data into account, which we will store in the portlet preferences.
- handle actions to allow the user to change these preferences.
- use the VIEW mode for displaying the list of bookmarks, and EDIT mode to manage the bookmark list.
- provide localized output.

In this example we will cover all parts that make up the bookmark portlet application. First we will explain the portlet code, and then the JSPs that are called from the portlet followed by the portlet deployment descriptor needed to successfully deploy the portlet and finally the resource bundles that provide the localization support. At the end screenshots of the running portlet in Pluto are provided.

3.1.1 *The portlet code*

The portlet code implementing the controller part of the MVC pattern consists of two sub-parts: the first part is the action processing in the portlet and the second part is generating the markup. In the action processing part, the portlet needs to react to specific user actions. The sample portlet recognizes two different actions that can be triggered in the Edit mode:

- remove – to remove a bookmark from the bookmark list. This is done by deleting the preference with that name from the portlet preferences and writing back this change with store.
- add – to add a bookmark to the bookmark list.

In order to keep the code small and easily browsable only minimal error handling is inserted into the code. We will break the code in three parts: the action handling and the rendering for both View and Edit mode.

Portlet Action Processing

Let's start with listing 3.1, which shows the action handling of the portlet.

Listing 3.1 process action handling of the bookmark sample

```
package com.manning.portlets.chapter03;

import java.io.IOException;

import javax.portlet.*;

public class BookmarkPortlet extends GenericPortlet {

    public void processAction (ActionRequest request,
                              ActionResponse actionResponse)
        throws PortletException, IOException
    {
        String removeName = request.getParameter("remove");
        if (removeName!=null) {
            PortletPreferences prefs = request.getPreferences();
            prefs.reset(removeName);
            prefs.store();
        }
        String add = request.getParameter("add");
```

← Remove an existing bookmark

Add a new bookmark

```

if (add!=null) {
    PortletPreferences prefs = request.getPreferences();
    prefs.setValue(request.getParameter("name"),
                  request.getParameter("value"));
    prefs.store();
}
}

```

So, what does this code fragment do? First of all our Bookmark portlet extends the GenericPortlet which is a good idea in most cases, as you get the dispatching of the render call to the different modes for free. The action implementation looks at the request parameters for a known action. First it checks for the remove action. If the “remove” action is found, the preference specified as request parameter id removed from the preferences and the preferences are stored again. If an “add” action is found, the name and value of the new bookmark are read from the request and stored in the preferences.

Rendering Markup

The second part, in listing 3.2, shows the implementation of the render method. For convenience the GenericPortlet provides dispatching to the standard modes VIEW, EDIT, and HELP. In this simple example we will not use the HELP mode and thus only overwrite the doView and doEdit methods. The first part implements the VIEW mode and sets the returned content type, retrieves the path of the view JSP from the portlet configuration and includes the JSP to render the view markup.

Listing 3.2 rendering part of the bookmark example: the View mode

```

public void doView (RenderRequest request,
                   RenderResponse response)
    throws PortletException, IOException
{
    response.setContentType("text/html");
    String jspName = getPortletConfig().getInitParameter("jspView");
    PortletRequestDispatcher rd =
        getPortletContext().getRequestDispatcher(jspName);
    rd.include(request, response);
}

```

The last part of the portlet code, in listing 3.3, implements the EDIT mode. Like in the doView method, the MVC pattern is used and the markup generation is forwarded to the edit JSP. As an example of how to transfer data between the portlet and the JSP, the two action URLs needed in the Edit JSP are set as attributes in the render request. In a real-life portlet the JSP would produce these URLs by itself, using tags defined by the Portlet Specification.

The sample code uses two different kinds of URLs: an action URL for the add URL to ensure that the processAction method is called and the portlet can process the bookmark that should be added to the list, and a render URL to cancel the EDIT mode that does not result in an action processing, as this is not required for only changing the portlet mode. Action URLs trigger the processAction method and allows the portlet to change state, whereas render URLs only set new render parameters and do not trigger a processAction call. More details on the action and render phases can be found in Chapter 2 (section 2.4), where we discuss the portlet architecture.

Listing 3.3 rendering part of the bookmark example: the Edit mode


```

public void doEdit (RenderRequest request,
                   RenderResponse response)
    throws PortletException, IOException
{
    response.setContentType("text/html");
    String jspName = getPortletConfig().getInitParameter("jspEdit");
    // generate action urls
    PortletURL addUrl = response.createActionURL();
    addUrl.setPortletMode(PortletMode.VIEW);
    addUrl.setParameter("add", "add");
    request.setAttribute("addUrl", addUrl.toString());
    PortletURL cancelUrl = response.createRenderURL();
    cancelUrl.setPortletMode(PortletMode.VIEW);
    request.setAttribute("cancelUrl", cancelUrl.toString());
    PortletRequestDispatcher rd =
        getPortletContext().getRequestDispatcher(jspName);
    rd.include(request, response);
}

} // closing of BookmarkPortlet

```

create the action URL for adding a new bookmark and switch back to View mode

create the render URL for canceling the Edit mode and switching back to View mode without any changes

include the Edti JSP

Now we have the complete controller code that handles the actions and dispatches to different JSPs based on the current portlet mode. Details about the included JSPs are explained in the next section.

3.1.2 The JSP code

As mentioned in the portlet code section the example consists of two JSPs: one rendering the VIEW mode and one rendering the EDIT mode. The View mode is the rendered list of the user's bookmarks. The Edit mode allows the user to add and remove items on the list.

The VIEW JSP

The View JSP shown in Listing 3.4 accesses the portlet preferences and iterates through the list of bookmarks that are defined and renders these bookmarks with their name and URL. The code also uses the resource bundle to output the localized text. There are also two things to note in the beginning of the JSP. First, it declares that no new session should be created for this JSP, which is always a good thing to put in portlet JSPs, as sessions should be created in the controller part. Second, it includes the portlet tag library with the taglib tag that allows using the portlet tags in the JSP. In a real-world JSP, one would also include the Java Standard Tag Library (JSTL) that provides tags for the conditional checks and the loops. That way, the amount of Java code in the JSP could be further reduced.

Listing 3.4: The View JSP

```

<%@ page session="false" %>
<%@ page import="javax.portlet.*"%>
<%@ page import="java.util.*"%>
<%@ taglib uri='http://java.sun.com/portlet'
    prefix='portlet' %>
<portlet:defineObjects/>
<%
ResourceBundle myText = portletConfig.getResourceBundle(request.getLocale());
%>
<B><%=myText.getString("available_bookmarks")%></B><br>
<%
    PortletPreferences prefs = renderRequest.getPreferences();

```

```

Enumeration e = prefs.getNames();
if (!e.hasMoreElements())
{
    <%myText.getString("no_bookmarks")%><BR>
}
while (e.hasMoreElements())
{
    String name = (String)e.nextElement();
    String value = prefs.getValue
        (name, "<" +
        myText.getString("undefined") + ">");
    <A HREF=<%=value%>><%=name%></A><BR>
}

```

← if no bookmarks are defined give out a corresponding text message

← if bookmarks are defined loop through all available bookmarks

← get the name of the bookmark

← get the URL of the bookmark

← write out name and URL of the bookmark

The EDIT JSP

The Edit JSP in Listing 3.5 is a bit more complex as it needs to provide a table with the current bookmarks, an action for removing this bookmark, and a form for submitting a new bookmark. As explained in the portlet code section, the URL to add a new bookmark and to cancel the edit screen are stored in the request as an example of transferring request-scoped data from portlet to a JSP. These URLs are now accessed in the JSP via the standard useBean tag. Next the form for adding a new bookmark with the add URL generated by the portlet is set up and the table heading is written using localized values for the name and the URL of the bookmark. Then the bookmark name and the bookmark URL are written to the output stream. Now we need to create the different buttons on the page for removing a bookmark, adding a bookmark and canceling the operation. The remove button is created using an action URL via the portlet tag library tags. For adding a new URL input fields are defined and the add action URL is put in the top of the form as the form action to execute. In order to cancel the Edit screen and return to the View screen, another form is set up that reuses the cancel URL generated by the portlet code. Using a render URL in forms, like in this form, is normally not encouraged, as form parameters may not be reflected as render parameters. However, as this render URL does not use any parameters it can be safely used and thus save an action call.

Listing 3.5 : The Edit JSP

```

<%@ page session="false" %>
<%@ page import="javax.portlet.*"%>
<%@ page import="java.util.*"%>
<%@ taglib uri='http://java.sun.com/portlet'
    prefix='portlet' %>
<jsp:useBean id="addUrl" scope="request"
    class="java.lang.String" />
<jsp:useBean id="cancelUrl" scope="request"
    class="java.lang.String" />
<portlet:defineObjects/>
<%
ResourceBundle myText = portletConfig.getResourceBundle
    (request.getLocale());
%>
<B><%=myText.getString("available_bookmarks")%></B><br>

```

← Portlet tag providing the request/response variables

create surrounding form for submitting a new bookmark

```

<FORM ACTION="<%=addUrl%>" METHOD="POST">
<TABLE CELLPADDING=0 CELLSPACING=4>
  <TR>
    <TD>
      <B><%=myText.getString("name") %></B>
    </TD>
    <TD>
      <B><%=myText.getString("url") %></B>
    </TD>
    <TD>
    </TD>
  </TR>
<%
PortletPreferences prefs = renderRequest.getPreferences();
Enumeration e = prefs.getNames();
while (e.hasMoreElements())
{
  String name = (String)e.nextElement();
  String value = prefs.getValue(name,
                                "<" +
                                myText.getString("undefined")
                                + ">");
%>
  <TR>
    <TD>
      <%=name%>
    </TD>
    <TD>
      <%=value%>
    </TD>
    <TD>
      <portlet:actionURL var="removeUrl">
        <portlet:param name="remove" value="<%=name%>" />
      </portlet:actionURL>
      <A HREF = "<%=removeUrl.toString()%>">
[<%=myText.getString("delete") %>] | #8
    </A>
    </TD>
  </TR>
<%
}
%>
<TR>
  <TD>
    <INPUT NAME="name" TYPE="text">
  </TD>
  <TD>
    <INPUT NAME="value" TYPE="text">
  </TD>
  <TD>
    <INPUT NAME="add" TYPE="submit"
      value="<%=myText.getString("add") %>"
    </TD>
  </TR>
</TABLE>

```

← create a table to format the output of all current bookmarks

← loop through all current bookmarks

← write the name of the bookmark

← write the URL of the bookmark

Portlet tag creating an action URL to remove the current bookmark and assigning it to the variable removeUrl

write the remove URL

← create input field to enter the name of the new bookmark

← create input field to enter the URL of the new bookmark

← create the add button

```

</FORM>
<FORM ACTION="<%=cancelUrl%>" METHOD="POST">
<INPUT NAME="cancel" TYPE="submit"
  VALUE="<%=myText.getString("cancel")%>">
</FORM>

```

create the cancel button

After we have now created all the JSPs we need for our portlet we need to put them at the correct location so that our previously created portlet can find them and include them,

Organizing the JSP code in your project

In order to organize our files we will put the two JSPs in a folder called “jsp” in the WAR folder of the Bookmark Portlet project so they are easy to locate. Also we need to store the directory information in the portlet init parameters in the portlet deployment descriptor so that our render code will find the correct JSP when calling the request dispatcher include. This “jsp” directory will therefore show up again in Listing 3.7.

Now that we have all the code fragments together, we need to create the corresponding deployment descriptors and resource bundle and package all parts together into a portlet application. This task is described in the next section.

3.1.3 Deployment descriptors and resource bundles

As mentioned above, a portlet application consists of two deployment descriptors: the web.xml for all resources besides portlets and the portlet.xml for portlet resources.

The web application deployment descriptor

The web.xml for the example simply defines an application name and description as shown in Listing 3.6.

Listing 3.6: The web application deployment descriptor (web.xml)

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <display-name>Bookmark Portlet Application</display-name>
  <description>Bookmark application for maintaining a list of
    bookmarks
  </description>
</web-app>

```

Note that the web.xml itself is required by the servlet specification for a correct WAR file, even if it is empty. The web.xml could of course have additional information about other J2EE parts of the portlet application that are not portlet related, like additional servlets or EJBs.

Let’s take a look at the portlet deployment descriptor that contains the portlet specific parts of our portlet application.

The Portlet Deployment Descriptor

The portlet.xml deployment descriptor of the example, shown in Listing 3.7, provides a description of the bookmark portlet, the portlet name and portlet class name.

Listing 3.7: The portlet application deployment descriptor (portlet.xml)

```

<?xml version="1.0" encoding="UTF-8"?>
<portlet-app xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd"
version="1.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd
http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd">
  <portlet>
    <description>Bookmark Portlet</description>
    <portlet-name>BookmarkPortlet</portlet-name>
    <display-name>Bookmark Portlet</display-name>
    <portlet-class> com.myCompany.portlets.BookmarkPortlet
      </portlet-class>
    <init-param>
      <name>jspView</name>
      <value>/jsp/view.jsp</value>
    </init-param>
    <init-param>
      <name>jspEdit</name>
      <value>/jsp/edit.jsp</value>
    </init-param>
    <expiration-cache>0</expiration-cache>
    <supports>
      <mime-type>text/html</mime-type>
      <portlet-mode>VIEW</portlet-mode>
      <portlet-mode>EDIT</portlet-mode>
    </supports>
    <supported-locale>en</supported-locale>
    <supported-locale>de</supported-locale>
    <resource-bundle>portlet</resource-bundle>
    <portlet-preferences>
      <preference>
        <name>Pluto Homepage</name>
        <value>http://portals.apache.org/pluto</value>
      </preference>
    </portlet-preferences>
  </portlet>
</portlet-app>

```

Defines JSP names for the View and Edit mode

Defines MIME type support HTML for our bookmark portlet

Declares supported locales – English and German

Defines a default bookmark, the Pluto homepage

In this listing, we define the JSP names as initialization parameters to allow changing the JSP names and locations without necessarily recompiling the portlet (1). The code further declares the supported markup MIME type and portlet modes for this MIME type (2). The different locales that the portlet supports are also declared with the corresponding resource bundle (3). Finally, the deployment descriptor defines a default bookmark (4).

Version Tracking with manifest.mf

In order to keep track of the version of the portlet application the WAR file should always contain a MANIFEST.MF file in the META-INF directory. For our example application the manifest file in Listing 3.8 defines the application name, version, and vendor.

Listing 3.8: Manifest file containing the version information of the portlet application (MANIFEST.MF)

```
Manifest-Version: 1.0
```

```
Implementation-Title: BookmarkPortletApplication
Implementation-Version: 1.1.1
Implementation-Vendor: Manning
```

Resource Bundles

In the resource bundle property files the portlet title, the short title for small screens and keywords are defined. Furthermore the portlet-specific keys with the corresponding text are specified. These text parts are accessed from the JSPs to provide localized output.

The first file in Listing 3.9 shows the resource bundle for German:

Listing 3.9: Resource bundle in German (portlet_de.properties)

```
javax.portlet.title = Lesezeichen Portlet
javax.portlet.short-title = Lesezeichen
javax.portlet.keywords = Lesezeichen, Bookmark
available_bookmarks = Verfügbare Lesezeichen
no_bookmarks = Keine Lesezeichen verfügbar. Bitte benutzen Sie den Edit-Mode, um neue
Lesezeichen hinzuzufügen.
undefined = nicht definiert
name = Name
url = URL
add = Hinzufügen
delete = löschen
cancel = Abbrechen
```

The second file in Listing 3.10 shows the resource bundle for English:

Listing 3.10: Resource bundle in English (portlet_en.properties)

```
javax.portlet.title = Bookmark Portlet
javax.portlet.short-title = Bookmark
javax.portlet.keywords = Bookmark
available_bookmarks = Available bookmarks
no_bookmarks = no bookmarks available. Please use edit mode to add bookmarks.
undefined = undefined
name = Name
url = URL
add = Add
delete = Delete
cancel = Cancel
```

In order to allow the classloader locating these property files we will put them in the classes folder under the Web-inf folder.

Now that we have all the pieces together we can start creating the WAR file and deploy the WAR file on a portal (see Chapter 8 for how this is done on Jetspeed and chapter 7 for the Pluto reference implementation).

3.1.4 Running the example

Now that we have explained all the code and other files that form this example application it is time to see the example running. The example was deployed and executed using the JSR 168 reference implementation Pluto. You can also download all book examples and a portal environment from

<http://www.manning.com/portlets>. Besides the portlet container, Pluto provides a simple portal, which allows testing portlets (see Chapter 7 for more details about Pluto). The first screenshot Figure 3.1 depicts the portlet in the view mode displaying the predefined Pluto bookmark from the `portlet.xml` file.

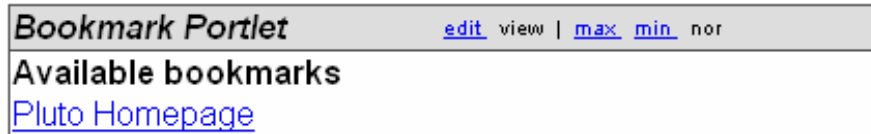


Figure 3.1: The user sees the Bookmark portlet in View mode, sized normally. From here, the user can click the bookmark to visit the Pluto homepage, resize the portlet with the Max and Min links, or go to Edit Mode to make changes to the Bookmark list.

The second screenshot Figure 3.2 shows the example portlet in Edit mode after clicking on the Edit link in the titlebar of the portlet, where the user can delete existing bookmarks or add new bookmarks.

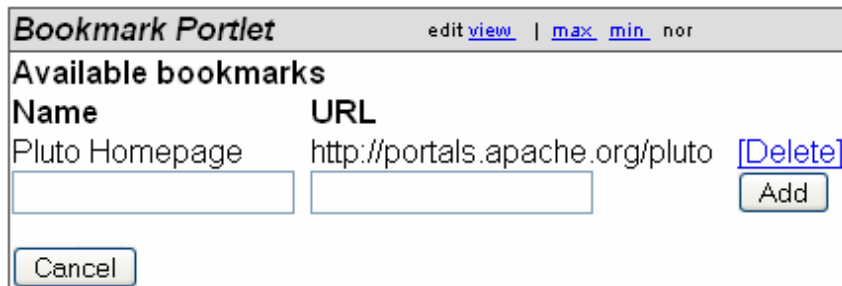


Figure 3.2: In Edit mode, the user can add new bookmarks with a name of their choosing, or delete existing bookmarks by clicking the Delete link.

Adding new bookmarks

We will now add two more bookmarks for Jetspeed and WSRP4J. The result of this operation is displayed in Figure 3.3.

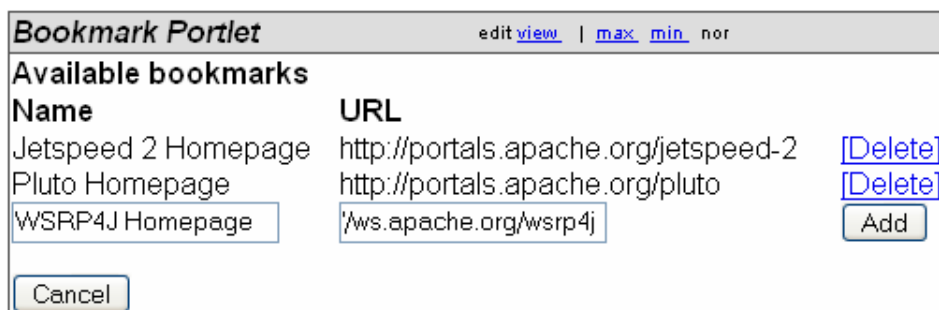


Figure 3.3 Adding two new bookmarks to the portlet in Edit mode. When you have copied the URL into the appropriate edit box, name the bookmark and click Add.

When pressing the “Add” button we will automatically switch back to the View mode and the portlet is rendering the new list of available bookmarks as HTML links. Additionally, to show that the National Language Support (NLS) enabling of the portlet really works, we switch the preferred language in the browser from English to German. As can be seen in Figure 3.4 the two new bookmarks are added to the list and the output changed to German.



Figure 3.4 Final view with language changed to German. Note that the user-defined names for the bookmarks do not change.

3.2 The Calendar portlet application example

After covering the basics of how a portlet is developed, including portlet code, JSPs, and internationalization, we will now create a more complex application, the Calendar portlet application. The Calendar portlet application consists of two portlets: a calendar portlet displaying a specific month where the user can select a day and a Dilbert display portlet that renders the selected day's To-dos in the calendar portlet.

In this section we will walk through how to create the entire calendar portlet application. The main tasks are listed in Table 3.2:

Table 3.2 Building a Calendar Portlet Portlet Application Task

Portlet Application Task	How To
Enable data sharing between portlets	Edit the Calendar portlet to store selected date Edit the Todo portlet to retrieve selected date
Create a calendar using JSTL tags	Add JSTL tags to Calendar portlet
Store and retrieve selected date using render parameters	Set render parameters in calendar portlet Set render parameters in viewDate JSP
Write portlet and JavaBean code	Write calendar portlet Write Todo portlet Write the calendar bean Write the Todo bean
Write JSP code	Write the viewDate JSP for calendar portlet Write the view JSP for Todo portlet Write the Edit JSP for Todo portlet
Package application	Create the deployment descriptor Create the file structure
Deploy and run application	Run application and add new To Do items

In this example we'll learn to program portlets by

sharing data between portlets for communicating the selected date from the Calendar portlet to the Todo portlet

- using JSTL (Java Standard Tag Library) tags in the JSPs in order to remove scriptlet code from the JSPs
- leveraging render parameters for paging through the different month in the Calendar portlet

Let's start with looking into the sharing of data between portlets.

3.2.1 Sharing data between portlets

In order to communicate the selected date from the Calendar portlet to the Todo portlet we need some inter-portlet communication means. The first version of the Java Portlet Specification only has limited support for

inter-portlet communication; real eventing between portlets is not yet supported (see also section 3.4 on future enhancements). Thus we need to share the data via the portlet session.

Storing the selected date

Portlets can share data via the session application scope that is accessible for all components in the same web application (see section 2.4.2 in Chapter 2). Therefore the Calendar portlet will set the currently selected date into the application scope session. The corresponding code snippet is listed in Listing 3.11.

Listing 3.11: CalendarPortlet storing the selected date in the session

```
public void processAction(ActionRequest request,
                        ActionResponse actionResponse)
    throws PortletException
{
    // get the selected date
    String date = request.getParameter(DATE);
    String month = request.getParameter(MONTH);
    if (date != null)
    {
        request.getPortletSession().
setAttribute(DATE,date,PortletSession.APPLICATION_SCOPE);

        actionResponse.setRenderParameter(MONTH, month);
    }
}
```

store the new date in the application session scope

← set the new month as render parameter

The current month is encoded as a request parameter by the viewDate JSP. We will cover this in more detail in section 3.2.3. For now, we will concentrate on the session sharing part. The portlet retrieves the current date and stores the date in the application scope session under the key DILBERT_DATE (see Section 3.2.5 and 3.2.6 for the complete portlet and JSP code).

Retrieving the selected date

Now the displaying portlet, called TodoPortlet, needs to retrieve the date from the session again. Listing 3.12 shows the corresponding code lines in the TodoPortlet for retrieving the date.

Listing 3.12: TodoPortlet retrieving the selected date from the session

```
protected void doView (RenderRequest request,
                      RenderResponse response)
    throws PortletException, IOException
{
    response.setContentType("text/html");

    TodoBean bean = new TodoBean();
    bean.setDate((String) request.getPortletSession().
getAttribute("date",PortletSession.APPLICATION_SCOPE));
    request.setAttribute(TODO_BEAN, bean);

    PortletRequestDispatcher rd = getPortletContext().
getRequestDispatcher("/jsp/view.jsp");
    rd.include(request,response);
}
```

get selected date from the application session

include the View JSP to ewnder the ToDo portlet view

The current selected date is retrieved from the session and set in a bean in order to allow the view JSP to retrieve the date. Another alternative would have been to directly access the session in the JSP, however that would have required some scriptlet code (see Section 3.2.4 and 3.2.5 for the complete portlet and JSP code).

After achieving the data sharing let's take a close look at how we can simplify the JSPs by using JSTL tags in the next section.

3.2.2 Applying JSTL to portlet JSPs

In our first example, the Bookmark portlet, we needed some Java scriptlet code, for example, for looping through all preferences. This is normally something you should avoid as it makes JSPs hard to read and mixes the concepts of UI design and programming in Java. Therefore we will use JSTL in the Calendar example in order to get rid of scriptlet code in the JSPs.

As we will see later on, you can't use the JSTL Expression language inside of the Java Portlet Specification tag library. This is due to the fact that the Java Portlet Specification taglib is based on J2EE 1.3, and thus is not compliant with the JSP Expression Language (EL) leveraged by JSTL. This restriction will be lifted as soon as a follow-on version of the Java Portlet Specification is defined.

Creating the calendar with JSTL

Listing 3.13 shows the JSP view of the CalendarPortlet, which uses JSTL tags in order to create the calendar.

Listing 3.13: The CalendarPortlet viewDate JSP parts with JSTL tags

```
<%@ taglib uri='http://java.sun.com/portlet' prefix='portlet'%>
<%@ taglib uri='http://java.sun.com/jstl/core' prefix='c'%>
<jsp:useBean id="bean" class="com.manning.portlets.chapter03.calendar.CalendarBean"
scope="request" />

<table><tr>
    <td>Sun</td><td>Mon</td><td>Tue</td><td>Wed</td><td>Thu</td>
    <td>Fri</td><td>Sat</td>
</tr><tr>
<c:forEach var="link" varStatus="status" items="${bean.dateLinks}">
    <td><c:out value="${link}" escapeXml="false"/></td>
<c:if test="${(status.count % 7) == 0}">
</tr><tr>
</c:if>
</c:forEach>
</tr></table>
```

← create a table for the calendar

loop through all days in the current month

every 7 days create a new table row

Note the escapeXml as the output is a URL that must not be escaped

The CalendarBean contains all links for the current month and via the forEach tag we loop through all links stored in the bean. We stored the complete link, which we prefabricated in the Calendar portlet, in the bean in order to highlight that all JSTL tags produce XML escaped output per default. For normal markup, you need to do this to comply with XML-based markups like XHTML. However, there are a few cases where this is not true, and one of them is outputting URLs that the user should be able to select. For these cases you can specify the escapeXml=false attribute and JSTL does not escape the output of this tag. Finally we test in the loop if we need to close the current table row and start a new one, as we've already put seven days in one row.

Now that we have produced a calendar for a single month, let's take a look at how we can leverage render parameters to allow switching between different months in the calendar.

3.2.3 Using render parameters

As mentioned in Chapter 2, a portlet can store its view state in render parameters. These render parameters are provided with each subsequent render request. New render parameters can be set either in the `performAction` method or via render links. The big advantage of render links is that they require less server computing power and they enable crawlability as search engines can follow the render links without fear of triggering an action that changes some backend state.

In our Calendar portlet we use render links in order to switch to either the previous or the next month. The render parameter therefore contains the current selected month. If no render parameter is set we use the month of the current date.

Setting render parameters in the Calendar portlet

Listing 3.14 shows the Calendar portlet snippets and Listing 3.15 the JSP snippets that implement the render parameter handling.

Listing 3.14 The render parameter handling in the CalendarPortlet

```
public void processAction(ActionRequest request,
                        ActionResponse actionResponse)
    throws PortletException
{
    String date = request.getParameter(DATE);
    String month = request.getParameter(MONTH);
    if (date != null)
    {
        request.getPortletSession().
            setAttribute(DATE, date, 3
PortletSession.APPLICATION_SCOPE);

        // set new render parameter
        actionResponse.setRenderParameter(MONTH, month);
    }
}

protected void doView (RenderRequest request,
                        RenderResponse response)
    throws PortletException, IOException
{
    ...
    Calendar today = Calendar.getInstance();
    CalendarBean bean = new CalendarBean();

    int month = today.get(Calendar.MONTH);
    int year  = today.get(Calendar.YEAR);

    if ( request.getParameter(MONTH) != null )
    {
        month = Integer.parseInt(request.getParameter(MONTH));
        today.set(Calendar.MONTH, month);
    }
    int prevMonth = (month > 0 ? month - 1 : 0);
    int nextMonth = (month < 11 ? month +1 : 11);
    ...
}
```

← get the selected date

← get the selected month

store the selected date in the application scope session

As an action resets all render params we need to set the new render param here; the selected month

store the selected date in the application scope session

get the selected date

← calculate the previous month

← calculate the next month

```

bean.setNextMonth(Integer.toString(nextMonth));
bean.setPreviousMonth(Integer.toString(prevMonth));
....
}

```

As you can see, in the `doView` method we try to get the current selected month from the request as render parameter. If we don't succeed, and no render parameter is available we take the current month as default. In order to be able to create the render links to the previous and next month we set these two values in the bean that is attached to the request and can be accessed in the JSP.

Setting render parameters in the `viewDate` JSP

In Listing 3.15 you can see how the render links are created using these bean values and how the render parameters are set through render links.

Listing 3.15 The render parameter handling in the `viewDate` JSP

```

...
<table><tr>
<td>Sun</td><td>Mon</td><td>Tue</td><td>Wed</td><td>Thu</td><td>Fri</td><td>Sat</td>
...

<portlet:renderURL var="myPrev">               ← create the render URL for
    <portlet:param name="month" value="<%=bean
    .getPreviousMonth()%>" />
</portlet:renderURL>
<portlet:renderURL var="myNext">
    <portlet:param name="month" value="<%=bean.getNextMonth()%>" />
</portlet:renderURL>

<table>               ← create the render URL for the next month
<tr>
<td>
<form action="<%=myPrev%>" method="GET">       ← render the button to get to
    <input name="prev" type="submit" value="Previous month">
</form>
</td>
<td>
<form action="<%=myNext%>" method="GET">       ← render the button to get to
    <input name="next" type="submit" value="Next month">
</form>
</td>
</tr></table>

```

Via the portlet tags for render URLs (1), we generate the render URLs and set the render parameter `month` with the values we have put into the bean in the portlet. Next we create a table (2) with two forms that only have one button. Note that we use the HTTP method GET here to allow search engines to follow this link and caching systems to cache the content.

Now that we have covered the major functions of the Calendar portlet application it is time to take a look at the complete portlet code.

3.2.4 *The complete portlet code*

As mentioned before, this portlet application consists of two parts:

1. the Calendar portlet displaying a calendar with links in order to select a specific date
2. the Todo portlet that displays the Todo's of the selected date

Writing the Calendar portlet code

Let's first take a look at the Calendar portlet code in Listing 3.16. The main task of the calendar portlet is to fill the Calendar bean with the links for all days of the current selected month.

Listing 3.16: The CalendarPortlet

```
package com.manning.portlets.chapter03.calendar;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Calendar;

import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;
import javax.portlet.GenericPortlet;
import javax.portlet.PortletException;
import javax.portlet.PortletRequestDispatcher;
import javax.portlet.PortletSession;
import javax.portlet.PortletURL;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;

public class CalendarPortlet extends GenericPortlet
{
    // key under which to store the shared data in the session
    public final static String DATE = "date";
    public final static String MONTH = "month";

    private final static String CALENDAR_BEAN = "bean";

    public void processAction(ActionRequest request,
                             ActionResponse actionResponse)
        throws PortletException
    {
        // get the selected date
        String date = request.getParameter(DATE);
        String month = request.getParameter(MONTH);
        if (date != null)
        {
            // store the new date in the application session scope
            request.getPortletSession().setAttribute(DATE,
            date,
            PortletSession.
            APPLICATION_SCOPE);
            // set new render parameter
            actionResponse.setRenderParameter(MONTH, month);
        }
    }
}
```

```

protected void doView (RenderRequest request,
                        RenderResponse response)
throws PortletException, IOException
{
response.setContentType("text/html");

try
{
    ArrayList links = new ArrayList();

    Calendar today = Calendar.getInstance();
    CalendarBean bean = new CalendarBean();

    int month = today.get(Calendar.MONTH);
    int year  = today.get(Calendar.YEAR);

    if ( request.getParameter(MONTH) != null )
    {
        month = Integer.parseInt(request.getParameter(MONTH));
        today.set(Calendar.MONTH, month);
    }
    int prevMonth = (month > 0 ? month - 1 : 0);
    int nextMonth = (month < 11 ? month +1 : 11);
    int maxDays = today.getActualMaximum(Calendar.DAY_OF_MONTH);
    today.set(Calendar.DAY_OF_MONTH, 0);

    for (int i=0; i<today.get(Calendar.DAY_OF_WEEK); i++)
        links.add("");| #1

    for (int i = 1; i <= maxDays; i++)
    {
        String dateString = getDateString(year,month+1,i);
        PortletURL portletURI = response.createActionURL();
        portletURI.setParameter(DATE, dateString);
        portletURI.setParameter(MONTH, Integer.toString(month));

        links.add("<a href=\"" + portletURI.toString
() + "\">" +
                dateString.substring(0,2) + "/" +
dateString.substring(3,5) +
                "</a>");
    }

    bean.setDateLinks(links);
    bean.setNextMonth(Integer.toString(nextMonth));
    bean.setPreviousMonth(Integer.toString(prevMonth));

    request.setAttribute(CALENDAR_BEAN, bean);

    PortletRequestDispatcher rd = getPortletContext().
getRequestDispatcher("/jsp/viewDate.jsp");
    rd.include(request, response);

```

← Add empty links to have the correct offset for the calendar table

Create the action link for the date

← Store all links in a bean and put bean in the

Render the calendar

```

        } catch (Exception exc)
        {
            this.getPortletContext().
log("CalendarPortlet: An error occurred ", exc);
            throw new PortletException("CalendarPortlet: An error
occurred");
        }
    }
}

```

```

private String getDateString (int year, int month, int day)
{
    StringBuffer date = new StringBuffer();

    if (month < 10) date.append(0);
    date.append(month);
    date.append("/");

    if (day < 10) date.append(0);
    date.append(day);
    date.append("/");

    date.append(year);

    return date.toString();
}
}

```

Helper method to create the correct date format

We now have the date links for our calendar ready and are able to receive actions on selected dates and store this date into the portlet session. Next, we need to create a To-Do list portlet that will allow us to store todo's for the selected date.

Writing the Todo portlet code

Second we created the TodoPortlet, which displays the To-Do list for the selected date in the view mode and allows you to add or remove items for the selected date in the edit mode. The Todo portlet is shown in Listing 3.17.

Listing 3.17: The TodoPortlet

```

package com.manning.portlets.chapter03.calendar;

import java.io.IOException;

import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;
import javax.portlet.GenericPortlet;
import javax.portlet.PortletException;
import javax.portlet.PortletPreferences;
import javax.portlet.PortletRequestDispatcher;
import javax.portlet.PortletSession;
import javax.portlet.RenderRequest;

```

```

import javax.portlet.RenderResponse;

public class TodoPortlet extends GenericPortlet
{
    private final static String TODO_BEAN = "bean";

    public void processAction(ActionRequest request,
                             ActionResponse actionResponse)
        throws PortletException, IOException
    {
        if (request.getParameter("remove") != null) // remove
        {
            PortletPreferences prefs = request.getPreferences();
            String date = (String) request.getPortletSession().
getAttribute("date", PortletSession.APPLICATION_SCOPE);
            String[] currentValues = prefs.getValues(date, new
String[0]);
            String[] newValues = new String[currentValues
.length-1];
            for (int i=0, j=0; i < currentValues.length; ++i, ++j)
            {
                if (currentValues[i].
equalsIgnoreCase(request.getParameter("todo")))
                    i++; // remove this item
                if (i >= currentValues.length)
                    break;
                newValues[j] = currentValues[i];
            }

            prefs.setValues(date, newValues);
            prefs.store();
        }

        if (request.getParameter("add") !=null ) // add
        {
            PortletPreferences prefs = request.getPreferences();
            String date = (String) request.getPortletSession().
getAttribute("date", PortletSession.APPLICATION_SCOPE);
            String[] currentValues = prefs.getValues(date, new
String[0]);
            String[] newValues = new String[currentValues.length+1];
            for (int i=0; i < currentValues.length; ++i)
                newValues[i] = currentValues[i];
            newValues[currentValues.length] = request.getParameter
("todo");
            prefs.setValues(date, newValues);
            prefs.store();
        }
    }

    protected void doView (RenderRequest request,
                           RenderResponse response)
        throws PortletException, IOException
    {

```

Only remove
the selected
todo from the
array of todo's

← Add new toda at the end of
the array of existing todo's


```

        response.setContentType("text/html");

        TodoBean bean = new TodoBean();
        bean.setDate((String) request.getPortletSession().
getAttribute("date", PortletSession.APPLICATION_SCOPE));
        request.setAttribute(TODO_BEAN, bean);

        PortletRequestDispatcher rd = getPortletContext().
getRequestDispatcher("/jsp/view.jsp");
        rd.include(request, response);

    }

    protected void doEdit(RenderRequest request,
                          RenderResponse response)
        throws PortletException, IOException
    {
        response.setContentType("text/html");

        TodoBean bean = new TodoBean();
        bean.setDate((String) request.getPortletSession().
getAttribute("date", PortletSession.APPLICATION_SCOPE));
        request.setAttribute(TODO_BEAN, bean);

        PortletRequestDispatcher rd = getPortletContext().
getRequestDispatcher("/jsp/edit.jsp");
        rd.include(request, response);
    }
}

```

The only thing worth noting in this portlet is that we've chosen to store the todo's with the date as key and all todo's for a specific date as array of Strings. Therefore we need to get all existing todo's in the processAction method and scan for the todo we want to remove or add the new todo at the end of the list.

Writing the calendar JavaBean code

We also have the two beans that are used by the Calendar and the Todo portlets. They are listed in Listing 3.18 and 3.19 for completeness. They are plain Java beans without any portlet-specific functionality.

Listing 3.18: The Calendar bean

```
package com.manning.portlets.chapter03.calendar;
```

```
import java.util.Collection;
```

```
public class CalendarBean {
    private Collection dateLinks = null;
    private String NextMonth = null;
    private String PreviousMonth = null;
```

store all links that select a current
day of the month as collection

store the next month

store the previous month

```

/**
 * @return Returns the dateLinks.
 */

```

```

public Collection getDateLinks() {
    return dateLinks;
}
/**
 * @param dateLinks The dateLinks to set.
 */
public void setDateLinks(Collection dateLinks) {
    this.dateLinks = dateLinks;
}
/**
 * @return Returns the nextMonth.
 */
public String getNextMonth() {
    return NextMonth;
}
/**
 * @param nextMonth The nextMonth to set.
 */
public void setNextMonth(String nextMonth) {
    NextMonth = nextMonth;
}
/**
 * @return Returns the previousMonth.
 */
public String getPreviousMonth() {
    return PreviousMonth;
}
/**
 * @param previousMonth The previousMonth to set.
 */
public void setPreviousMonth(String previousMonth) {
    PreviousMonth = previousMonth;
}
}

```

The Calendar bean stores the links of each day of the current month and the previous and next month. The Calendar bean is filled by the Calendar portlet and accessed by the viewDate JSP.

Writing the Todo JavaBean code

The Todo bean stores the current selected date and is filled by the Todo portlet and accessed by the view and edit JSPs.

Listing 3.19: The Todo bean

```

package com.manning.portlets.chapter03.calendar;

public class TodoBean {
    private String Date = null;
    /**
     * @return Returns the date.
     */
    public String getDate() {
        return Date;
    }
    /**

```

← store the current date as string

```

    * @param date The date to set.
    */
    public void setDate(String date) {
        Date = date;
    }
}

```

As next step we'll take a look at the complete JSP code in the next section.

3.2.5 The complete JSP code

Our Calendar portlet application consists of three JSPs: one viewDate JSP for the Calendar portlet and one view and one edit JSP for the Todo portlet.

Writing the viewDate JSP code for the calendar portlet

Let's start with the viewDate JSP, which is shown in Listing 3.20.

Listing 3.20: The viewDate JSP of the Calendar portlet

```

<%@ taglib uri='http://java.sun.com/portlet' prefix='portlet'%>
<%@ taglib uri='http://java.sun.com/jstl/core' prefix='c'%>
<jsp:useBean id="bean" class="com.manning.portlets.chapter03.calendar.CalendarBean"
    scope="request" />

<table><tr>
    <td>Sun</td><td>Mon</td><td>Tue</td><td>
        <td>Fri</td><td>Sat</td>
    </tr><tr>
<c:forEach var="link" varStatus="status" items="${bean.dateLinks}">
    <td><c:out value="${link}" escapeXml="false" /></td>
    <c:if test="${ (status.count % 7) == 0}">
        </tr><tr>
    </c:if>
</c:forEach>
</tr></table>

<p></p>

<portlet:renderURL var="myPrev">
    <portlet:param name="month" value="<%=bean.
        getPreviousMonth() %>" />
</portlet:renderURL>
<portlet:renderURL var="myNext">
    <portlet:param name="month" value="<%=bean.getNextMonth() %>" />
</portlet:renderURL>

<table><tr>
    <td>
<form action="<%=myPrev%>" method="GET">
    <input name="prev" type="submit" value="Previous month">
</form>
</td>
<td>
<form action="<%=myNext%>" method="GET">
    <input name="next" type="submit" value="Next month">

```

← create a table for the calendar

← loop through all day links for the selected month

← create a table for the next and previous month buttons

← create the previous month button

← create the next month button

```

</form>
</td>
</tr></table>

```

We already explained the JSTL tag and the render links in the previous sections. One thing worth noting here is that we need to create the render links accessing the bean via the `<%= %>` scriptlet code and not via the JSTL EL code `${bean.nextMonth}`. As mentioned before, this is due to the fact that the current Java Portlet tag library does not support the expression language.

Writing the view JSP code for the Todo portlet

Next we take a look at the view JSP of the Todo portlet in Listing 3.21.

Listing 3.21: The view JSP of the Todo portlet

```

<%@ taglib uri='http://java.sun.com/portlet' prefix='portlet'%>
<%@ taglib uri='http://java.sun.com/jstl/core' prefix='c'%>
<%@ page import="javax.portlet.*"%>
<jsp:useBean id="bean" class="com.manning.portlets.chapter03.calendar.TODOBean"
scope="request" />

<portlet:defineObjects/>

<P>
Current selected date: <c:out value="${bean.date}"/>
</P>
<P>
<P>
<B>ToDo's for the current date:</B><br>
</P>
<c:set var="prefsMap" value="${renderRequest.preferences.map}"/>
<c:if test="${empty prefsMap[bean.date]}">
    No ToDo currently defined for this date.<br>
    To add a ToDo please press "Edit ToDo's".
</c:if>
<c:if test="${not empty prefsMap[bean.date]}">
    <c:forEach var="todo" items="${prefsMap[bean.date]}">
        <c:out value="${todo}" /><BR>
    </c:forEach>
</c:if>
</p>

<form action="<portlet:renderURL portletMode="edit"/>" method="GET">
    <input type="submit" name="action" value="Edit ToDo's">
</form>

```

← get all todo's from the portlet preferences

← if no todo is defined for the selected date output a note how to add todo's

← if todo's are available for the selected date render the todo's

← loop through all todo's of the selected date

← add a button to switch to the Edit mode in order to allow the user to add or remove todo's

Here we also tried to use JSTL tags wherever possible. First we retrieve the preferences map from the request and then get the array of todo's for the current selected date. First we check if the array is empty, which means that there are no todo's defined for the selected date. If the array is not empty we'll output all todo's. Finally we provide a convenience link to the edit mode where the user can add or remove todo's.

Writing the edit JSP code for the Todo portlet

Listing 3.22 shows the last JSP, the edit JSP of the Todo portlet that shows the current todo's in a table and allows adding a new todo or remove an existing todo.

Listing 3.22: The edit JSP of the Todo portlet

```
<%@ taglib uri='http://java.sun.com/portlet' prefix='portlet'%>
<%@ taglib uri='http://java.sun.com/jstl/core' prefix='c'%>
<jsp:useBean id="bean" class="com.manning.portlets.chapter03.calendar.TODOBean" scope=
    "request" />

<portlet:defineObjects/>

<P><B>Defined Todo's</B></P>
<P>For date: <c:out value="${bean.date}" /> </P>

<c:set var="prefsMap" value="${renderRequest.preferences.map}" /> ← get the portlet
                                                                    preferences that
                                                                    contain the todo's

<form action="<portlet:actionURL portletMode="view" />"
    method="post"> ← create a form to add or remove a todo
<table> ← create a table to display all currently defined
        <tr>                                todo's for the selected date
        <td>
            <B>Current Todo's</B>
        </td>
        </tr>
        <c:set var="prefsMap" value="${renderRequest.preferences
.map}" />
        <c:if test="${not empty prefsMap[bean.date]}">
        <c:forEach var="todo" items="${prefsMap[bean.date]}"> ← loop through all
            <tr><td><c:out value="${todo}" /></td>                currently defined todo's
            </tr>                                                for the selected date
        </c:forEach>
        </c:if>
        </table>
<p></p>
<table> ← create a table for the add
        <tr>                                and remove buttons
        <td>
            <input name="todo" type="text">
        </td>
        <td>
            <input name="add" type="submit" value="Add Todo">
        </td>
        <td>
            <input name="remove" type="submit" value="Remove Todo">
        </td>
        </tr>
        </table>
</form>
<form ACTION="<portlet:actionURL portletMode="view" />" METHOD="POST"> | #6
    <input name="cancel" type="submit" value="Cancel"> ← #7 create the cancel button
</form>
```

Note that we needed to have a separate input field for removing a todo and could not add a link behind each todo in the table, as we wanted to avoid using scriptlet code. This is also rooted in the fact that the JSTL

variable defined in the `forEach` loop cannot be used for creating portlet links and thus there is no way currently to encode the `todo` stored in the `#{todo}` variable in a portlet remove link.

Now that we have all the code together let's bundle everything together into a portlet application in the next section.

3.2.6 Packaging the portlet application

In order to allow the Calendar and the Todo portlets to access the shared data in the session, both portlets need to be packaged into one portlet application, as each portlet application, like every web application, had its own session.

Creating the deployment descriptor for the calendar portlet

First we need to create the portlet deployment descriptor, which is shown in Listing 3.23.

Listing 3.23 The calendar portlet application deployment descriptor (portlet.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-app xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd"
version="1.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd
http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd">
  <portlet> ← define the calendar portlet
    <portlet-name>CalendarPortlet</portlet-name>
    <display-name>Calendar Portlet</display-name>
    <portlet-class>com.manning.portlets.chapter03.calendar.CalendarPortlet
  </portlet-class>
    <expiration-cache>-1</expiration-cache>
    <supports> ← the calendar portlet only supports the View mode
      <mime-type>text/html</mime-type>
    <portlet-mode>VIEW</portlet-mode>
    </supports>

    <supported-locale>en</supported-locale>

    <portlet-info>
      <title>CalendarPortlet</title>
      <short-title>Calendar</short-title>
      <keywords>Calendar</keywords>
    </portlet-info>
  </portlet>

  <portlet> ← define the todo portlet
    <portlet-name>TodoPortlet</portlet-name>
    <display-name>Todo Portlet</display-name>
    <portlet-class>com.manning.portlets.chapter03.calendar.TODOPortlet
  </portlet-class>
    <expiration-cache>-1</expiration-cache>
    <supports> ← the todo portlet supports the View and Edit mode
      <mime-type>text/html</mime-type>
    <portlet-mode>VIEW</portlet-mode>
    <portlet-mode>EDIT</portlet-mode>
    </supports>

    <supported-locale>en</supported-locale>
```

```

        <portlet-info>
        <title>TodoPortlet</title>
        <short-title>Todo</short-title>
        <keywords>Todo</keywords>
        </portlet-info>
    </portlet>
</portlet-app>

```

In order to make the deployment descriptor as short as possible, we added only the bare minimum. See the first bookmark example on how to implement a portlet application that supports multiple languages and help screens.

Creating the file structure

Now we need to place all files in the correct location.

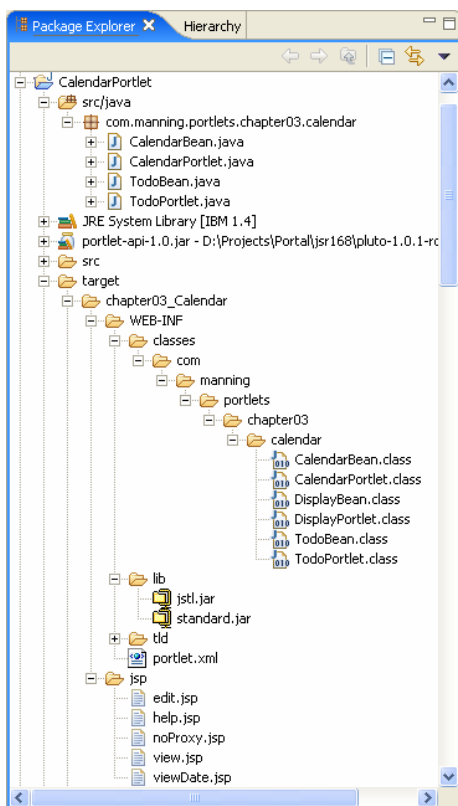


Figure 3.5 The file structure of the Calendar portlet application

Under the target directory we put everything we need in the portlet application WAR file. Chapter 7 explains in detail how to create WAR files and deploy them on Pluto. You can also just zip the content without compression using a zip tool or the jar tool from Java JDK and rename it to a WAR file with the appendix .war. As an alternative you can go to <http://www.manning.com/portlets> and download all samples of this book as source and in a running portal environment.

After we have everything together for our application let's see how the application is running on Pluto.

3.2.7 Deploying and Running the calendar portlet example

Before being able to run the example we need to deploy it on our Pluto sample portal. We'll do this with the simple maven command available in Pluto (example for the Windows environment):

```
d:\>maven deploy -Ddeploy=target\chapter03_Calendar.war
```

For more details on how to set up Pluto and deploy portlet applications, please see Chapter 7. After deploying the Calendar portlet application on the Pluto sample portal we get the result displayed in Figure 3.6

The screenshot shows the Pluto Apache Portals interface. At the top is the Pluto logo and the text "Apache Portals". Below this is a navigation bar with "Pluto Portal Driver (pluto-driver/1.0.1-rc2)" on the left and a "Login" link on the right. The main content area is divided into two portlets. The "Calendar" portlet on the left has a title bar with "Calendar" and a link. It contains a "CalendarPortlet" table with columns for days of the week and dates from 01 to 31. The date 05/28 is highlighted. Below the table are "Previous month" and "Next month" buttons. The "TodoPortlet" on the right has a title bar with "TodoPortlet" and links. It shows "Current selected date: 05/28/2005" and a list of "ToDo's for the current date": "Prepare Party" and "Go shopping". There is an "Edit ToDo's" button at the bottom.

Figure 3.6: The Calendar portlet application view

As you can see in the Todo portlet we selected the 28th of May 2005 as the date in the Calendar portlet. You can also see the two buttons with the render links in the Calendar portlet that sets the month to the previous or next month. In the Todo portlet the current todo's for the 28th of May are displayed.

Adding a todo item

Now we want to add a new todo for this day and thus click on the “Edit Todo’s” button (see Figure 3.6). The Todo portlet now renders the edit screen and we enter the new todo “Call Peter” in the input field as shown in Figure 3.7:

The screenshot shows the Pluto Apache Portals interface in edit mode. The "Calendar" portlet is the same as in Figure 3.6. The "TodoPortlet" now has a title bar with "TodoPortlet" and links. It shows "Defined Todo's" and "For date: 05/28/2005". Below this is a section for "Current ToDo's" with the same list: "Prepare Party" and "Go shopping". At the bottom, there is an input field containing "Call Peter", and "Add ToDo", "Remove ToDo", and "Cancel" buttons.

Figure 3.7: Adding a new Todo in the Todo portlet

The final step is to click on the “Add ToDo” button. Figure 3.8 shows that the Todo portlet now displays the new todo item at the end of the todo list.

[Calendar](#)

Calendar

[help](#) [view](#) | [max](#) [min](#) [nor](#)

Sun	Mon	Tue	Wed	Thu	Fri	Sat
05/01	05/02	05/03	05/04	05/05	05/06	05/07
05/08	05/09	05/10	05/11	05/12	05/13	05/14
05/15	05/16	05/17	05/18	05/19	05/20	05/21
05/22	05/23	05/24	05/25	05/26	05/27	05/28
05/29	05/30	05/31				

[Previous month](#)
[Next month](#)

TodoPortlet

[edit](#) [help](#) [view](#) | [max](#) [min](#) [nor](#)

Current selected date: 05/28/2005

ToDo's for the current date:

Prepare Party
Go shopping
Call Peter

[Edit ToDo's](#)

Figure 3.8: View of the Calendar portlet application with a new todo item

We have now covered two portlet application examples and gained some experience in developing portlets. It's time to look into some portlet best practices to help you create high quality portlets right from the start.

3.3 Adopting Portlet best practices

Now that we have covered the architectural concepts of portlets and seen a real portlet coming into life, we will provide some best practices in this section that will enable you to write more efficient, maintainable, and better quality portlet code. However, guidelines are always for the 90% case, and there are always exceptions where one of these rules may not apply. Therefore, use these best practices as a rule of thumb, but not as something that is carved in stone.

We organized the guidelines in the following broad categories:

- portlet coding
- JSP coding
- use of persistent data and transient data
- internationalization
- portlet application packaging.

3.3.1 Coding portlets

Beyond the normal Java related best practices, like using JavaDoc comments, there are some things that are specific to portlet development that we will cover in this section.

Use MVC pattern and frameworks

Use the Model-View-Controller pattern, either directly from the portlet by passing data to the view (JSP) as a bean in the request and including the JSP, or via additional frameworks, like Struts or Java Server Faces (JSF). This will result in maintainable and extensible applications and save you a lot of time and money later on. Additional frameworks are very beneficial for larger portlet projects and are covered in detail in Chapters 5 and 6.

Store data in Instance variables

Portlets exist as a singleton instance within the JVM of the portlet container. Therefore, a single memory image of the portlet processes all requests and must be thread-safe. Data stored in instance variables will be accessible across requests and across users and can collide with other requests. Data must be passed to internal

methods as parameters. There are other means of storing data across requests as explained before, like using the session or the render parameters.

Include Version information

In order to enable portal administrators to differentiate between different portlet application versions and to provide update support, declare the version of your portlet application in the META-INF/MANIFEST.MF using the 'Implementation-Version' attribute. Apply the recommendation of the Java Product Versioning Specification for the version string with major.minor.micro, where

- Major version numbers identify significant functional changes.
- Minor version numbers identify smaller extensions to the functionality.
- Micro versions are even finer-grained versions.

These version numbers are ordered with larger numbers specifying newer versions. This way you can provide updates of your portlets that get correctly deployed as new versions of your portlet.

Avoid Using J2EE roles

Only declare J2EE roles in your portlet application if absolutely necessary. J2EE roles are separate from the portal roles and need to be managed separately from the portal roles. Therefore J2EE roles should only be used to perform access control in portlet applications if the user profile information is not sufficient.

Always Use P3P user profile attributes

The Platform for Privacy Preferences (P3P) defines ways how users can control the privacy of personal information in the Internet. One part of this specification defines attribute names for all common user information. Use these P3P user profile attributes whenever possible for accessing user attributes (see [2] for more details about the P3P specification). When the portlet needs to access user profile attributes, like the user name or address, it should always use the keys that P3P defines for these attributes. This reduces the administration effort when deploying the portlet as the P3P attributes are automatically mapped to the attributes in the current user directory, whereas attribute names not in the P3P list need to be manually mapped by the portal deployer. These are also listed in the Java Portlet Specification Appendix D.

Use URL encoding

Use URL encoding for resources inside the portlet application WAR file in order to allow the portal to proxy resources inside the portlet application WAR file. Resource links should thus always be encoded using the `encodeURL` method.

However, for resources outside the portlet application WAR file do not use the URL encoding. URLs outside the portlet application should not be encoded in order to let the client directly access the resource outside the portlet application WAR file. This reduces the workload on the portal server as the server hosting the resource can directly serve the resource without any involvement of the portal server.

Optimize Parallel portlet rendering

In order to allow the portal rendering the portlet in parallel with other portlets on the page in the render phase the portlet should:

1. Expect `IOExceptions` when writing to the `OutputStream` or `PrintWriter` and act accordingly. If a portlet takes too much time for rendering its content the portal may cancel the rendering of this portlet and this will result in an `IOException` when the portlet tries to write to the `OutputStream` or `PrintWriter` after the portal has canceled the rendering of this portlet.
2. In methods that are expected to take many computation cycles the portlet should periodically check if the `flush` method of the `OutputStream` or `PrintWriter` throws an `IOException`. If the `flush` method throws

such an exception the portal has canceled the rendering of the portlet and the portlet should terminate its current computation.

Render URLs and form POSTs

The render phase should not change any state, but provide a re-playable generation of the markup. Thus HTTP POST requests that submit forms should always be handled in an action via creating an ActionURL and not in render. This also ensures that the portlet can be used as WSRP service, as WSRP does not support new parameters in a render request. The only exception from this rule is when the form does not consist of any parameters (e.g. a cancel button).

Ensure minimal portlet functionality for unsupported vendor-specific portal extensions

If your portlet uses extensions specific to a portal vendor, it should also be coded to run in a plain JSR 168 environment with no extensions. Degraded functionality is acceptable when running in a plain JSR 168 environment. The portlet should check at runtime the support extensions of the calling portal via the PortalContext and act accordingly.

Avoid Naming confusion between portlets and servlets

Don't name portlets and servlets the same in one web application. Tools and portals may use the portlet name to identify the portlet in a web application and may get confused if there is also a servlet with the same name in this web application.

3.3.2 Coding JSPs

This section contains coding guidelines that need to be considered when writing JSPs that are included from portlets. The emphasis on the guidelines is on HTML; however similar rules apply to other markup languages as well.

Honor portlet specific markup restrictions

The portal server aggregates the markup of several different portlets into one document that is sent back to the client. Thus, JSPs should only contain markup fragments in order to allow this aggregation. Therefore, all markup tags that belong to the document cannot be used by the portlet. See section PLT.B of the Java Portlet Specification for more details about tags that the JSP is not allowed to use.

Use standard style classes

Use the style classes recommended by the Java Portlet Specification section PLT.C in order to give the portal page a consistent look-and-feel across portlet applications provided by different parties. Only if portlets use these pre-defined styles will the whole pages that are aggregated out of different portlets from different portlet providers be displayed to the portal end user with a consistent look-and-feel.

Namespace portlet specific resources on a page

URIs, HTML element name attributes, and JavaScript resources must be namespace-encoded. Since many portlets can exist on a page, and it is possible that more than one portlet will produce content with like-named elements, there is a risk of namespace collision between elements, causing functional problems with the page. Use the <portlet:namespace/> tag to encode such resources with the portlet instance name.

For example, to a FORM name:

```
<FORM method=POST name="javascript:<portlet:namespace/>form1" action="<%=actionUrl%>">
```

Be careful with JavaScript code

Minimize dependencies on JavaScript code since JavaScript implementations and behavior differ widely between browser types and versions. The more your portlet depends on JavaScript, the more browser-dependent your portlet becomes in the end.

Avoid pop-up windows

Do not use pop-ups, as interactions within the portal are state-based, which means that the portal tracks your trail through the pages and portlets. By using pop-ups, the portal is no longer in a position to track the user action and cannot provide a consistent user experience.

Use tag libraries

Use tag libraries whenever possible. Encapsulating Java code within taglibs not only allows common view functions to be easily reused, it keeps the JSPs clean and makes them more like normal HTML pages, which allows the page designer to concentrate on layout and decoration and reduces the possibility of breaking the embedded Java code. For example, use the Java Standard Tag Library (JSTL) instead of Java code. JSTL defines a lot of commonly needed tags for conditions, iterations, URLs, internationalization and formatting.

Use iFrames only as last resort

Use IFRAMEs with caution. IFRAMEs are an easy way to include external content within a portlet, but undermine the whole portlet principle as the portlet API is just tunneled. Thus IFRAMEs should only be used for very special cases, like surfacing legacy applications. Otherwise you will end up re-creating a second portal in your IFRAME and all the money you spend on your portal system in the first place is wasted as the portal integration ends at the IFRAME. This means that all links rendered in the IFRAME would not contain the navigational state of the other portlets on the page and thus clicking on these links would result in losing the state for all other portlets on the page.

Endure cross-site scripting protection

In order to protect the portal site from malicious code that a user can enter in an input field of your JSP be sure to encode entered data of the user before writing them to the output stream again. If you don't do this a user could enter some JavaScript code that is executed when written to the output stream without encoding it. This kind of attack is not new and is known for all kind of web applications and was reported as cross-site scripting by CERT in 2000 (see [3]). A simple solution to protect against this attack is to use the Java URLEncoder class to encode user input before writing it to the output stream.

3.3.3 Persistent and transient data usage

As mentioned earlier, portlets have access to different kinds of states. The portlet programmer should, very carefully and early on, decide the category of information for each state in order to most efficiently use these different categories.

Save Navigational state conservatively

Minimize navigational state information wherever possible, as the navigational state of all portlets on the current page need to be aggregated and is normally stored in the URL. Therefore, the navigational state that the portlet stores in the render parameters need to be minimized in order to keep the URLs small. Also, most small devices only support a very limited URL length. Of course storing this information in session is not an option as this is even more expensive. However, maybe some of the data is not pure navigational state, but can be recreated by the portlet. This kind of data should be stored in a cache.

The portlet should also be programmed in such a manner that it takes the request nature of the navigational state into account. This means that with the next request to the portlet all previous navigational state may not be re-transmitted to the portlet and the portlet should still produce some meaningful output.

Manage Session state issues

Limit the use of the portlet session for storing state information. The portlet session is a convenient place to store global data that is user- and portlet-specific and that spans portlet requests. However, there is considerable overhead in managing the session, both in CPU cycles and heap consumption. Only data should be stored in the portlet session if the data is user-specific and cannot be recreated by any other means, such as portlet state information. For example, parsed configuration data (from portlet preferences) should not be stored in the portlet session, since it can be recreated at any time. Data that can be recreated should be stored using a cache service and not in the session.

Prevent temporary sessions from being generated in the JSP. Add the JSP page directive `<%@ page session="false" %>` to the JSP to prevent temporary sessions from being created by the JSP compiler, if none already exist. This will help guard against attempting to use the session to store global data if the session will not exist past the current request. You will need to be sure the Portlet Session exists before trying to use it.

Be aware of session timeouts. As each portlet application is a separate web application, each portlet application will have its own session. This results in different timeouts for different portlets on a page, as the user may interact with some portlets more frequently than with other portlets.

Use attribute prefixing for global session scope. Portlets can write into the web application session without any prefixing of the portlet container using the application scope portlet session setting. This can be used to share data between a portlet and other portlets or servlets of the same web application and/or between several entities created out of the same portlet. The portlet must take into account that there may be several entities of it on the same page and that the portlet may need to prefix the global setting to avoid other entities overwriting this setting. One convenient way to do this is provide a read-only portlet preference entry called session-prefix that the administrator can set in the Config mode.

Persistent state

Use portlet initialization parameters for storing static information not meant to be changed by administrators or users. These data are specified in the portlet deployment descriptor using the `init-param` tag and are read-only for the portlet. The portlet can access these data via `PortletConfig.getInitParameter`. An example is declaring the JSP names and directories that are used for rendering the different modes. This allows changing the JSP names and directory structure without needing to re-compile the portlet.

Use read-only portlet preferences for storing configuration data. Configuration data are user-independent and should only be changed by administrators in the Config portlet mode (e.g. the name of the news server). These data should be declared in the portlet deployment descriptor using the preference tag together with the read-only tag. The portlet should also provide one or more Config screens to allow the administrator to change this setting.

Use writeable portlet preferences for storing customization data. Customization data are user-specific data that the user can set to customize the portlet output (e.g. news he/she is interested in) via the Edit mode. These data should be declared in the deployment descriptor with some meaningful default values as writeable portlet preferences. The portlet should provide one or more edit screens to allow the user to change these values.

Write persistent data only in the action phase. The portlet must only change persistent state information in the action phase. The render phase must be completely re-playable to avoid issues with the browser back button and bookmark ability.

Use String arrays for preference lists, as preferences can be either Strings or String arrays. Use String arrays if you have a list of preference values for one key as this makes it easier to manage.

3.3.4 Internationalization

If you stick to the recommendations of this section, you will enable the portal to fully leverage the internationalization provided by your portlet.

Use resource bundles

Use a resource bundle per portlet to internationalize the portlet output and declare this resource bundle in the portlet deployment descriptor. All displayable strings should be stored in resource bundles and fetched using the ResourceBundle Class or in JSPs via the JSTL internationalization tags. The resource bundles should be organized by language under the portlet's WEB-INF/classes directory, and the portlet should always provide a default resource bundle with only the base name to enable the fallback mechanism for locales not supported by the portlet. This will make your portlet very robust and easily maintainable in the future.

Instead of using the local system's locale for referencing a resource bundle, always use the locale specified on the render request in order to honor the portal-wide language settings that the user has defined.

Reference Preference data

Define preference names as reference names to the localized name in the resource bundle. The names of the preferences in the portlet deployment descriptor should be references under which the localized name for this preference is stored in the resource bundle. The portlet should use the naming convention defined in the Portlet Specification section 13.3.1 for the resource bundle entries. This will also allow external tools or generic portlets for changing the preference data to display the preferences in a localized manner.

Define Supported locales

Define the locales that the portlet supports in the deployment descriptor. In the portlet deployment descriptor the portlet should define all locales it supports using the <supported-locale> element. This enables the portal to show users only portlets for locales that they have selected.

3.3.5 Portlet application packaging

As mentioned above, portlets and their resources are packaged together in portlet applications. However, portlet applications should define a meaningful set of portlets in order to leverage the provided application concept.

Use the library pattern

One common pattern that also applies to portlet applications is the library pattern. This means that you should make common functions available externally to portlets. If the portlet contains common functions that are replicated across several portlets, consider isolating them, making them externally accessible by the portlets. The easiest way to do this is build another JAR file of the common classes and place the JAR file either in each portlet application or a location that is in each portlet application classpath, such as the shared directory of the application server.

Group portlets in a meaningful way

Another very important rule is to group portlets that operate on the same backend data into one portlet application. This allows for sharing configuration settings, like the backend server name, or userid / password, and data like the current date that the user is interested in, between the portlets of this application.

Packaging portlets this way allows separating portlets from each other via the web application sandbox that don't relate to each other and enable related portlets to leverage sharing mechanisms, like the session, provided inside the web application.

3.4 Future features: next generation portlets

As the JSR 168 was the first version of the Portlet Specification many more complex functionalities or use cases were not addressed that exists already in today's proprietary portlet vendor-specific APIs. However, this should not stop portlet programmers from moving to the standard API, as the standard API will also evolve and very likely vendors will provide their additional functionality as extensions to the standard API.

Therefore, this section covers areas that are not addressed in the first version of the Portlet Specification, but are on the list of future versions of the portlet specification. This should also help you to distinguish between current proprietary extensions to the standard API that will soon make it into the standard and ones that are further away from that goal and thus should be used with more care.

Proprietary extensions should always be encapsulated with specific guards that check if the extension is in available in the current environment (e.g. by using the `PortalContext` to determine this) or with a try-catch block to handle any `ClassNotFoundExceptions` that may occur. Always program your portlet in such a manner that it can run in a plain JSR 168 environment with degraded functionality.

3.4.1 Inter-portlet communication

Inter-portlet communication provides means for sending and receiving events between portlets. This provides a more integrated end-user view, as different portlets of different portlet applications can be wired together to provide a consistent user experience. A calendar portlet may, for example, send out an event containing the current date the user is looking at and a todo portlet may take this event as input and display all todo's due on this date. Inter-portlet communication also allows structuring the user interface in a more modular manner and therefore enables applications like selection and browser portlets. In the selection portlet the user can select a topic of interest out of a list, such as a news article she/he is interested in and the browser portlet will then display the complete news article. These portlets can now be placed on the page wherever the user likes them to be, even on different pages.

So inter-portlet communication is really something that is needed for a lot of portlet applications and is therefore on the list of the next version of the portlet specification.

3.4.2 Client-capabilities access

Access to client capabilities would give the portlet the ability to take special client capabilities, like screen size, into account when generating the markup. For client capabilities the standard Composite Capabilities/Preferences Profiles (CC/PP) from W3C allows defining device capabilities and the User Agent Profile (UAProf) that provides specific attributes definitions for individual devices.

The CC/PP standard is now available as a Java API that was specified in the JSR 188 (see [4]). However, as JSR 188 was developed in parallel with the portlet specification it does not take portlets into account and the factory in CC/PP that allows you to get the profile of a client takes only a `HttpServletRequest` as input parameter and not a `PortletRequest`. Therefore either the CC/PP API needs to be extended to also take a `PortletRequest` as input parameter, or the portlet specification needs to add a method that directly returns a CC/PP client profile.

3.4.3 Advanced caching

Currently, the caching options for portlets are restricted to only expiration-based caching. However, this is a very static approach that for many applications is too coarse-grained. Advanced caching capabilities would enable the portlet to provide caching information that goes beyond this simple expiration mechanism of the first version.

One enhancement could be to distinguish between content that is user-specific and needs to be cached per user and content that is shared between users and therefore would only need to be cached once. This would dramatically reduce the caching storage needed for portlets that are not personalizable by users, like a company-wide announcement portlet that would display the same message for all users.

Another area of improvement is providing a more fine-grained control to the portlet via validation mechanisms allowing the portlet to check based on a validation tag if the given content is still valid or not. This allows only generating markup for the views that have expired as a result of the action and not all output the portlet has produced.

3.4.4 Portlet entities and portlet windows

Portlet entities and portlet windows are currently not reflected in the portlet API. This means that the portlet does not get notified if a new entity or a new window is created, cloned or deleted. The portlet may want to react on all of these events, e.g. initialize and destroy services needed at portlet entity creation and deletion time, or removing special values from the preferences, like userid and password, when being cloned.

This lack of notification currently restricts the portlet programming model, as the portlet cannot claim resources on a per-entity level, as it does not get notified when new entities are created or the current one is destroyed. This will also be hopefully addressed in the next version of the portlet specification.

3.4.5 Contributing content to other page parts

Currently portlets are restricted to producing only content that is displayed inside the portlet window. One step to weaken this restriction is already done in the current version of the portlet specification, as a portlet can specify a portlet title, which is rendered in the title bar of the portlet. However, currently the portlet can only produce text output to be included in the title, no icons or sounds.

A general concept of allowing the portlet to contribute to other page parts would therefore include the document header, the navigation area, title bar, or a related links section. Figure 3.9 depicts these different areas. On the navigation area the portlets could provide specific links to functions that are available or sub-views available. On the title bar the portlets may also provide icons and sound, instead of text only, and in the related links sections, the portlets could provide links to related information.

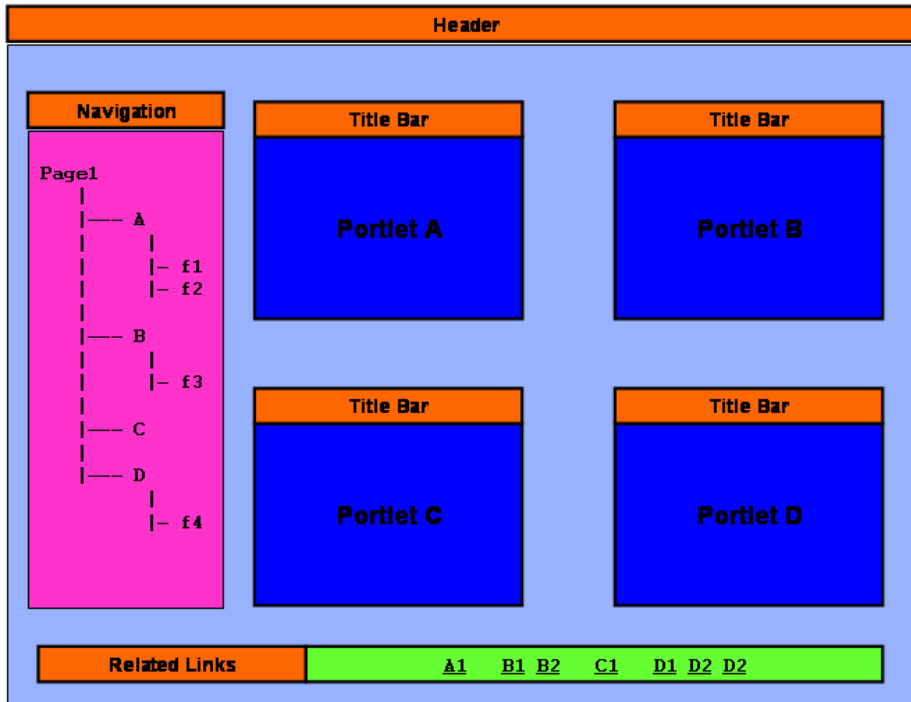


Figure 3.9 Areas where portlets may want to contribute content to besides their markup in the portlet window (in orange)

This would allow you to create a much more consistent user experience, as new portlets would not only seamlessly plug into a portal window, but also in the overall content and layout structure of the portal page.

3.4.6 Next version of the Java Portlet Specification

As of the writing of this book, in April 2005, no follow-on version to JSR 168 has been submitted, but we expect that this will happen soon. The content of this follow-on version to JSR 168 will be of course defined by the Expert Group of this new JSR, still there are some items which are very likely to be integrated in this new version as they were deferred from the first version. Therefore following items are likely to be addressed in this version:

- Support of J2EE 1.4 in addition to J2EE 1.3. This would allow portlets to use J2EE 1.4 enhancements, like JSP 2.0.
- Access to client capability date (CC/PP) via the API defined by JSR 188. We already covered the reasons why this is important above.
- Enhanced caching capabilities like the ones explained above.
- Alignment with WSRP 2.0 [1] that is also currently under development. The goal is, like for the Java Portlet Specification 1.0 and WSRP 1.0, being able to publish remote Java portlets and creating proxy Java portlets that can proxy WSRP remote portlets into J2EE portals.
- Inter-portlet communication, like mentioned above.
- Define portlet entity and portlet window concepts to further clarify the scope of the navigational state and session state and allow for use cases like pop-up windows. This may also include the above mentioned lifecycle listeners for portlet entity and portlet window creation and deletion.

With this additional functionality in place, especially the inter-portlet communication and the enhanced caching, nearly all complex scenarios can then be realized with the Portlet API without the need of additional, vendor-specific, extensions.

3.5 Summary

In this chapter we provided all the information you need to get started with developing portlets. First, we showed you two portlets in action. The simple bookmark portlet demonstrates every needed component: the portlet code, JSPs, deployment descriptors, and resource bundles. This example also covered many of the previously explained concepts, like the MVC-pattern, portlet preferences, action and render phase, and different portlet modes. Next, we covered a more complex example with sharing of data between two portlets. This example used render links to store view state, and JSTL to simplify the JSPs.

After getting familiar with the portlet code in the examples, a section with best practices for portlet development provided a comprehensive list that helps you avoid many of the pitfalls that one can get into when starting with portlet development.

Finally, we shed some light on future developments in the portlet space and some of the features the next version of the Java Portlet Specification will include.

The Java Portlet Specification will enable interoperability of portal servers and Java portlets and thus create a healthy ground for growing the portal market, and encouraging the growth of portlet applications from third parties and tools.

Now that we have laid out all the groundwork of portlet development and created our examples by hand with a simple text editor, we will take a look at how tools can help us make this task easier.

3.6 References

- [1] Web Services for Remote Portlets ; URL: <http://www.oasis-open.org/committees/wsrp/>
- [2] L. Cranor, M. Langheinrich, M. Marchiori, M. Presler-Marshall, J. Reagle: The Platform for Privacy Preferences 1.0 (P3P1.0) Specification; URL: <http://www.w3c.org/TR/P3P>
- [3] CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests; URL: <http://www.cert.org/advisories/CA-2000-02.html>
- [4] CC/PP Processing; URL: <http://www.jcp.org/en/jsr/detail?id=188>

Further readings

- S.Hepper, M.Lamb, Best practices: Developing portlets using JSR 168 and WebSphere Portal V5.02; URL: http://www-106.ibm.com/developerworks/websphere/library/techarticles/0403_hepper/0403_hepper.html
- www.theserverside.com – for discussions around J2EE topics
- <http://www.ibm.com/developerworks/java> – for additional technical articles around servlet and portlet programming

Chapter 4 Building Portlets with Eclipse and other tools

In the previous chapter, we showed you how to program portlets by making a bookmark portlet just using a text editor, so we could focus on the different parts that make up a portlet application. If you did not yet read Chapter 3 it would be a good idea to do this now before reading this chapter as it reuses the example explained in Chapter 3.

Now you may ask, right, but how would I develop a portlet in real-life? As we showed in Chapter 1, you do not have to abandon your nice integrated development environments (IDEs) to develop portlet applications.

In this chapter, we will write the same bookmark portlet using Eclipse, the most common open source IDE. You'll learn which parts Eclipse will automatically generate for you, and which parts you need to fill in yourself. After that, we will take a look at advanced tool features that really make portlet application development very simple. These advanced tools include support for frameworks like Struts or JSF and the ability to create your portlet application visually instead of typing in code.

4.1 Using Eclipse for portlet development

This section will introduce how to use Eclipse in order to develop JSR 168 compliant portlets. While there are lots of different commercially available IDEs that support JSR 168, like BEA WebLogic Workshop, IBM Rational Application Developer, Oracle JDeveloper, and Sun ONE Studio, the number of offerings in the open source community really comes down to one: Eclipse (see [1]). Eclipse has broad industry support behind it: IBM, Borland, QNX Software Systems, Rational Software, RedHat, SuSE, Fujitsu and others. In fact, Eclipse is the most popular Java IDE in North America according to Evans Data Research [2].

Therefore, we will provide an overview of portlet development with Eclipse and the Pluto plug-in that allows you to create JSR 168 portlets. First, we will give you a basic introduction to Eclipse, then we will show you how to use the Pluto plug-in. The plug-in includes the Pluto JSR 168 reference implementation and saves you some time by writing some of the necessary stub code. In section 4.2 we will use Eclipse in order to re-create the Bookmark portlet example.

4.1.1 Quick introduction to Eclipse

This book is not about Eclipse. However for those of you new to Eclipse, a quick introduction can help you get started. For those of you with Eclipse experience, skip over this section. This introduction covers the basic Eclipse features for Java development and are not related to the Pluto plug-in. The Pluto plug-in offers additional functionality and is explained in section 4.1.2.

Understanding Eclipse Perspectives

Eclipse is based on workspaces and projects. Workspaces are containers for holding collections of projects. Projects hold the actual Java source code, build configurations, and resources. You can view a project's source code using Eclipse's Perspectives. The most important Eclipse perspectives for Java developers are Resource, Java, Java Browsing, and Debug.

In the Resource perspective, the Eclipse Navigator provides a hierarchical view of all files in your project, like the one displayed in Figure 4.1.

Here we see the basic layout of the typical portlet project explained in Chapter 3, the bookmark portlet. The src directory has java and webapp. The java directory holds the portlet Java code. The webapp directory holds JSP templates and deployment information.

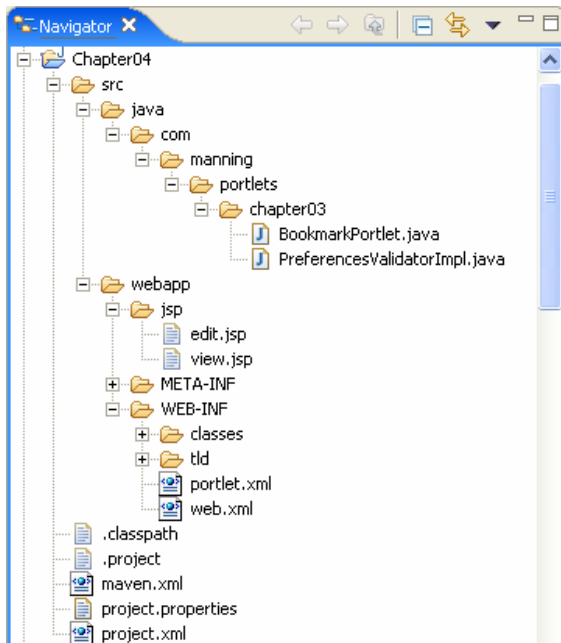


Figure 4.1: Eclipse's Resource Perspective displays a tree of every file in your project. To open a file in the editor, just double-click.

Another important Java perspective is Java Browsing. The Java Browsing perspective allows you to get a great overall view of the Java classes in your projects as displayed in Figure 4.2.

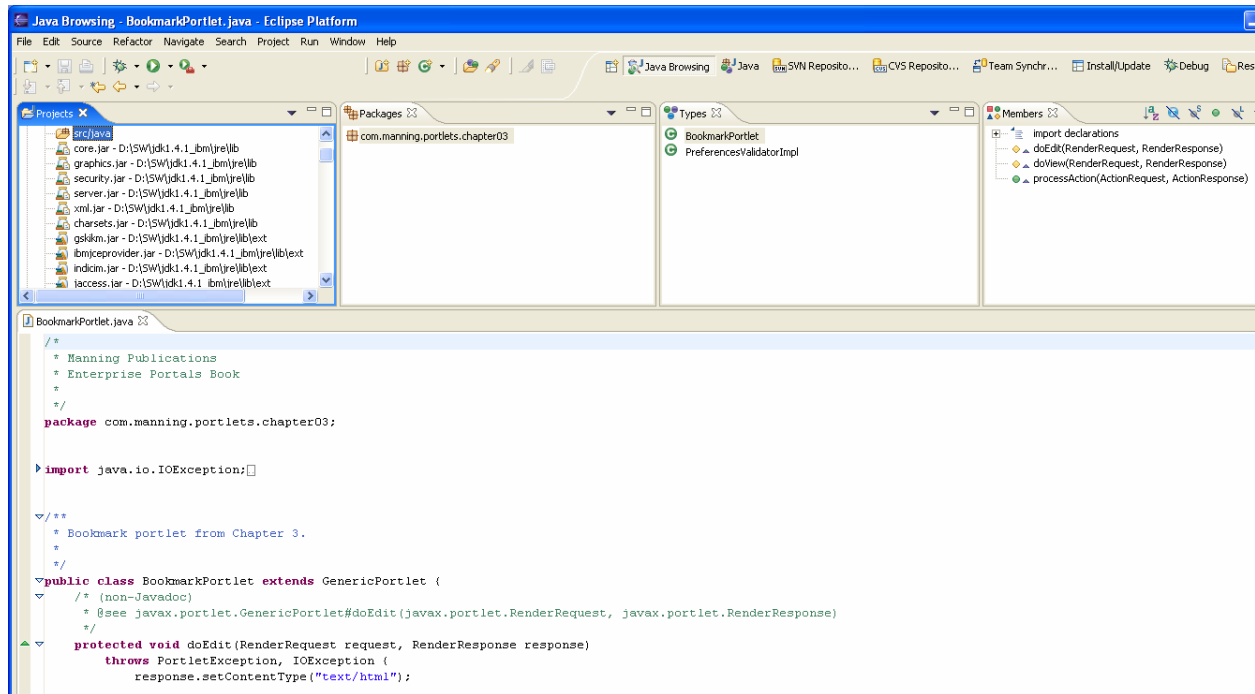


Figure 4.2: Eclipse's Java Browsing Perspective gives you an overview of the Java classes in your project. Tabs show your projects, packages, types and members, and the editor view at the bottom lets you edit your code directly.

Notice that from the package window, we can see all the packages in a project. From the Outline window, we can see the methods, imports, and class data members for a class. Above, we see the standard portlet methods provided by a portlet. Clicking on any of these will navigate to the source file declaration in the main source window. For example, clicking on “doEdit” brings you to the declaration of the method shown on the right side.

Finally, Eclipse makes editing Java code easy. The editor is sensitive to your typing, and will automatically help you enter class elements. For example, type “this.” with a portlet will display a drop-down list of legal options for you to choose from (Figure 4.3):

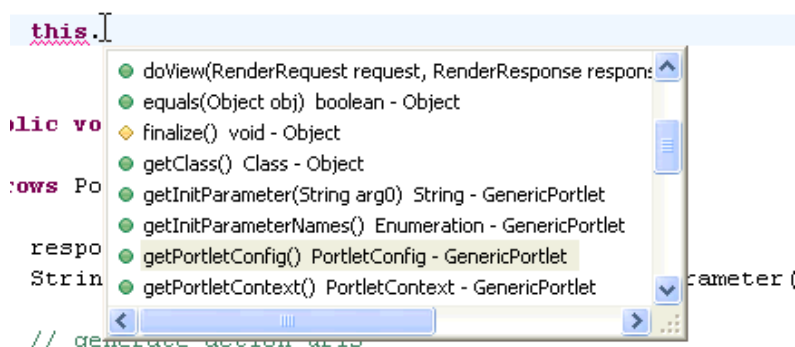


Figure 4.3: Eclipse makes Java editing easy with context-sensitive completion.

If you choose the portlet request instead you will see all accessible methods on the portlet request:

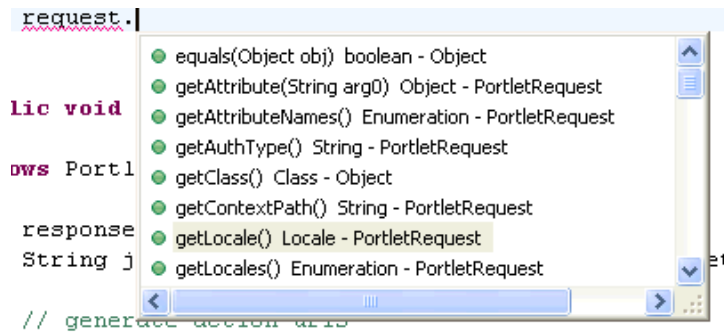


Figure 4.4: When typing a Portlet Request, the Eclipse editor knows what comes next.

In short, Eclipse is a flexible and very useful Java-based tool that will simplify your portlet coding tasks. Once you have completed coding, Eclipse can also help you debug, which we will see in the next section.

Debugging your portlet application with Eclipse

As mentioned before, portlets are running in the portlet container and are accessing data that the portlet container provides, like the portlet preferences. Therefore it is very helpful being able to run portlets in debug mode and be able to introspect the different variables at runtime. In order to debug your portlet application, it first needs to be deployed to the portal server.

To demonstrate a few debugging features, we will use Pluto as the portal server to run the First, we will give you a basic introduction to Eclipse, then we will show you how to use Pluto bookmark portlet just like we did in Chapter 3. Once the portlet application is deployed and the Pluto portal is running, you can then connect Eclipse to the JVM running the portal under Tomcat.

First, switch Eclipse into Debug Perspective. Either use the Perspective icon in the upper right corner of the Workbench, or use the Window menu to bring up the Perspective menu, which can be seen in Figure 4.5:

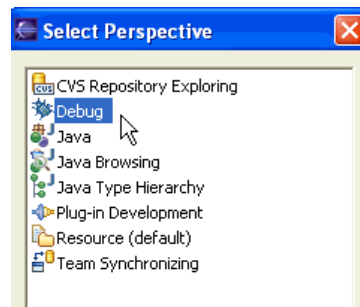


Figure 4.5: Switching to the Debug Perspective. Perspectives in Eclipse are preset views designed to complete a specific task, like coding in Java, debugging, and connecting to a CVS repository. You can choose from the default selections, or build your own perspective.

Next, attach the Eclipse debugger to the running instance of Tomcat. Select 'Remote Java Application' and then press New and then Debug. After that the screen shown in Figure 4.6 appears where you can configure the debugging options.

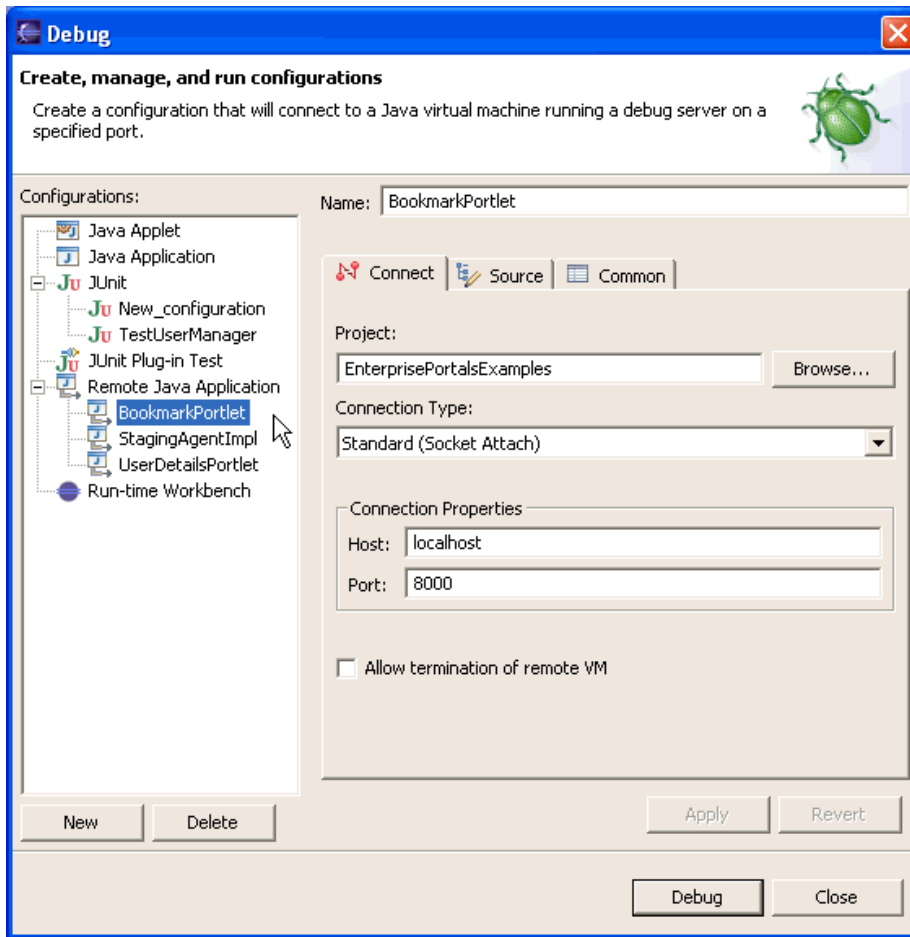


Figure 4.6: Attaching Eclipse to the remote Tomcat server running Pluto portal. This allows the debugger to check your code.

After pressing Debug, you will then see your debugger connect to the remote JVM. Let's explore how to put breakpoints in your portlets. Navigate down to the Bookmark Portlet found under "src/java/com/manning/portlets/chapter3" and open the Java source file. Scroll down to the doView method, and click in the margin outside the main source window, as in Figure 4.7:

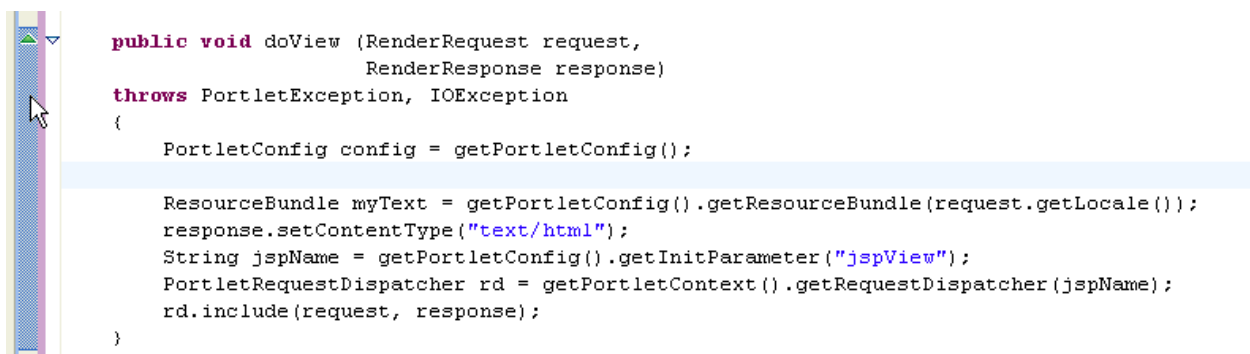


Figure 4.7: Double click in the margin to create a breakpoint

Now, with your web browser, navigate to the page where you placed the Bookmark Portlet. When the portal goes to render the page holding the Bookmark Portlet, the breakpoint in Eclipse will interrupt the application

server's JVM, and allow you manually step through your code with the debugger (Figure 4.8). Notice how the line where you placed the breakpoint is highlighted.

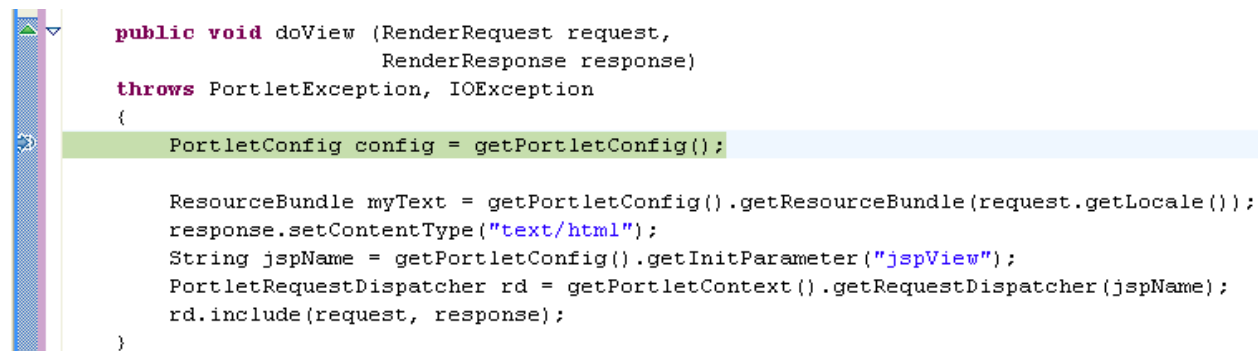


Figure 4.8: The Eclipse Debugger lets you step through each line of code.

The debugger allows you to step through each line of code. There are several options from the Run menu option along with function keys as shown in Figure 4.9:

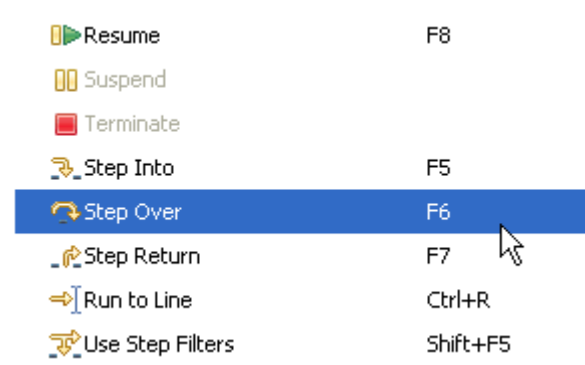


Figure 4.9: Debugging Step Options

Finally, examining your variables is dead easy. Just take a look at them in the Variables window as depicted in Figure 4.10:

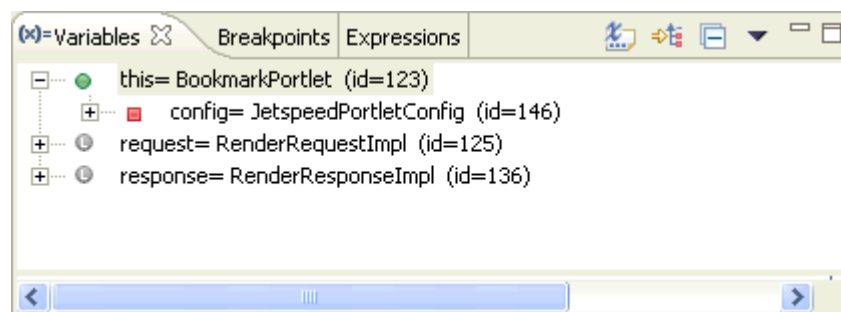


Figure 4.10: Examining Variables

This quick overview of debugging in Eclipse will give you a good foundation for debugging portlets. We'll come now to a more convenient way on how to create portlets in Eclipse: the Pluto plug-in that already creates a lot of code for you automatically.

4.1.2 Eclipse and the Pluto plug-in

As of the writing of this book, `org.eclipsefan.pluto.ui` at Sourceforge is the only plug-in for JSR 168 portlet development available in the open source space for Eclipse (see [3]). This plug-in is called “Pluto” because it comes with a version of the Pluto JSR 168 reference implementation for testing the created portlets on a running portal. Pluto consists of a portlet container running the portlet and a simple sample portal so that you can see a portal page displaying your portlet. We’ll discuss the Pluto reference implementation in more detail in chapter 7 of this book.

The name Pluto for the plug-in may be a bit confusing, as the plug-in really is a JSR 168 portlet development environment that provides you with wizards to create your portlet stubs and the deployment descriptor. The Pluto plug-in, therefore, saves you from having to type in all this boring boiler-plate code that is very similar in every project.

Downloading and Installing the Pluto Plug-in

Installing the Pluto plug-in is a snap. To download the Pluto plug-in, go to [3], click Installation Instructions, then click the download link. Extract the complete zip archive into the `/eclipse/plugins` directory, then open Eclipse.

You should now be able to follow the instructions in section 4.2.1 to set up your portlet application.

Limitations of the Pluto Plug-in

There are a couple of problems with the Pluto plug-in, though. The current implementation of this plug-in does not allow you to run the Pluto portal inside Eclipse. This means that after creating your portlet application with the Pluto plug-in, you need to copy the portlet application to the Pluto portal directory, and start the portal implementation outside of Eclipse. Finally, you must then fire up your web browser to access your portlet. Hopefully this restriction will be overcome in the future and a round-trip development completely inside Eclipse will be possible, which would be much more convenient, less error prone and faster.

One other restriction of the current version 1.0 of this plug-in needs to be mentioned: the plug-in creates a `web.xml` deployment descriptor that is tailored especially for Pluto. This makes it easier to deploy the portlet application directly on Pluto with just a copy, instead of an additional Pluto deployment command. The disadvantage is that you need to hand-edit the `web.xml` if you want to deploy your portlet application on any other portal. This is another glitch that will hopefully be fixed in one of the next versions of the plug-in.

This plug-in already generates a lot of stub code that you no longer need to write yourself. So let us dive into that part now and implement the Bookmark example from chapter 3 with Eclipse and version 1.0 of the Pluto plug-in.

4.2 Developing the Bookmark portlet in Eclipse

Let’s begin by re-creating the Bookmark example of Chapter 3 with Eclipse and the Pluto plug-in. By using the same portlet sample you can easily see the parts that we created by hand in Chapter 3 are now created for us automatically by Eclipse. Besides one new code fragment, the preferences validator, the code you’ll see in the screen shots is exactly the same as in Chapter 3.

So what do we need to get started? First you need to have Eclipse version 2.1.1 (it also runs on Eclipse 3.0, but this is currently not officially supported), which you can download from [1]. Next you need to download the Pluto plug-in from [3] and follow the installation instructions listed in section 4.1.2. After you have done this, you have the basic development environment set-up.

4.2.1 Creating the project

Now we can start by creating the bookmark project as a Pluto Portlet Application. From the Eclipse Workspace, click New, then Project from the menu. Depending on your version of Eclipse, you should see something like the New Project window seen in Figure 4.11.

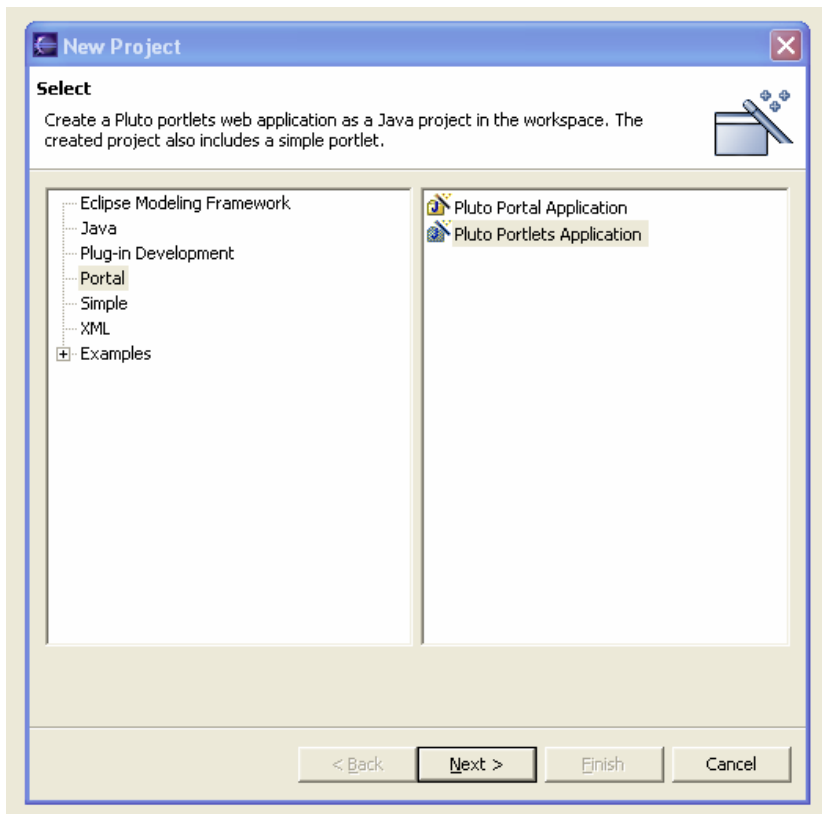


Figure 4.11: Creating a new portlet project in Eclipse. You must have the Pluto Eclipse plug-in installed for the Portal option to be included in the menu.

In the left pane, select “Portal” as the type of project you are creating. The right pane will display two options. The first is the Pluto Portal Application which the Pluto plugin can use to install the Pluto reference implementation. In this chapter the Pluto reference implementation will run our sample. As we discussed in section 4.1.2, Pluto provides a JSR 168 compliant portlet container and a simple sample portal for displaying the portlets. You can learn much more about the Pluto reference implementation in chapter 7.

Select the second option to create the portlet application and click Next. On the next page, fill in the Project Name for our bookmark sample in the wizard, and the plug-in will create a new portlet application.

On the left side of Figure 4.12 the created basic project can be seen. The project already contains a link to the portlet API library and an empty source folder src and the web-root folder under which the web application parts will be stored.

After we successfully created the portlet project we need to create the Java package for the portlet code. From the Java perspective, go to File, then New, then Package. The New Java Package dialog box appears, as shown on the right side of Figure 4.12. Create the package com.myCompany.portlets to hold the Bookmark portlet by typing this in the Name box. Click Finish to create the package. You should now see the new package listed in the Package Explorer under the src folder.

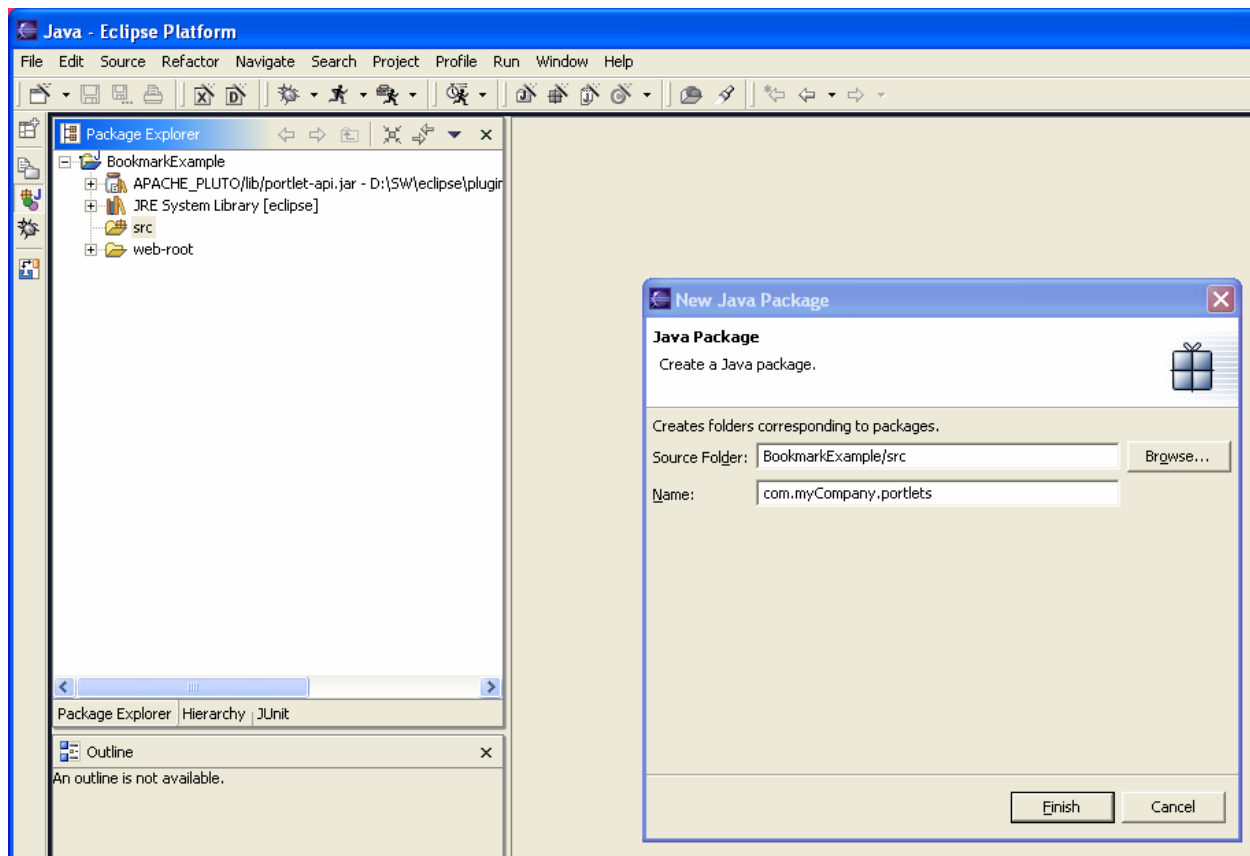


Figure 4.12 Creating the java package for the portlet code. The Package Explorer seen in the Java perspective lists the generic code created by default by the Pluto plug-in, but you still must do the specific coding, including the package creation.

Now we will create the bookmark portlet class `BookmarkExample` and add `GenericPortlet` as superclass of the bookmark portlet. Extending the `GenericPortlet` is in most cases a good idea, as it already dispatches different portlet modes to different methods. Go to the New menu again and click Class. The dialog box in Figure 4.13 appears.

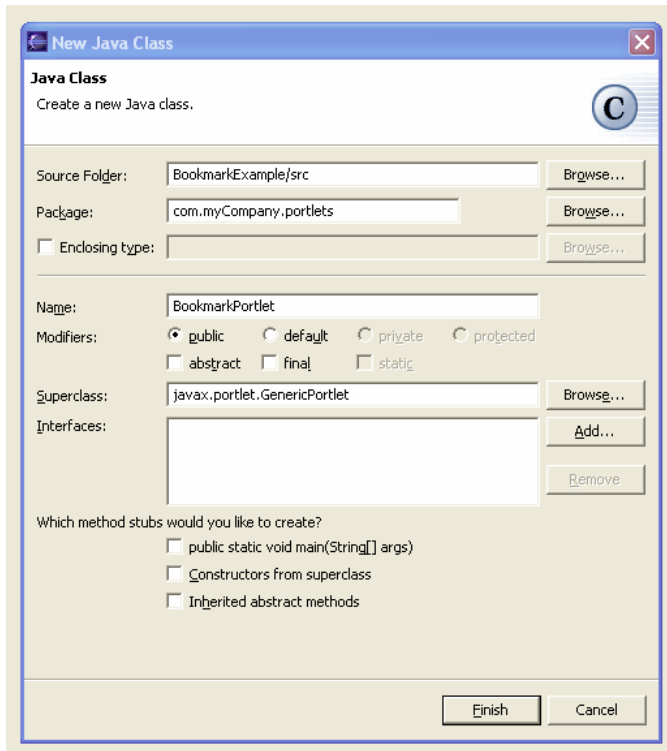


Figure 4.13: Creating the bookmark portlet class.

You may need to browse for the correct package and Superclass, or just type them in yourself. Type the name `BookmarkPortlet` in the appropriate box. No interfaces need to be added here, nor do you need any of the method stubs listed in the dialog box. You will define new methods in the next section. Click **Finish** to create your new class.

In Figure 4.14 below, you can see on the left side the newly created `BookmarkPortlet.java` file showing up in the project browser and on the right side behind the dialog box the automatically generated code for the `BookmarkPortlet`.

4.2.2 Creating the portlet part

The next step is to create the methods for the action handling and handling of the View and Edit mode. We will use the Eclipse wizard to create the method stubs for us with the dialog depicted in Figure 4.14. Go to the **Source** menu and select **Override/Implement Methods**. Select the `doView`, `doEdit` and `processAction` methods from the menu. Click **OK** to add these methods.

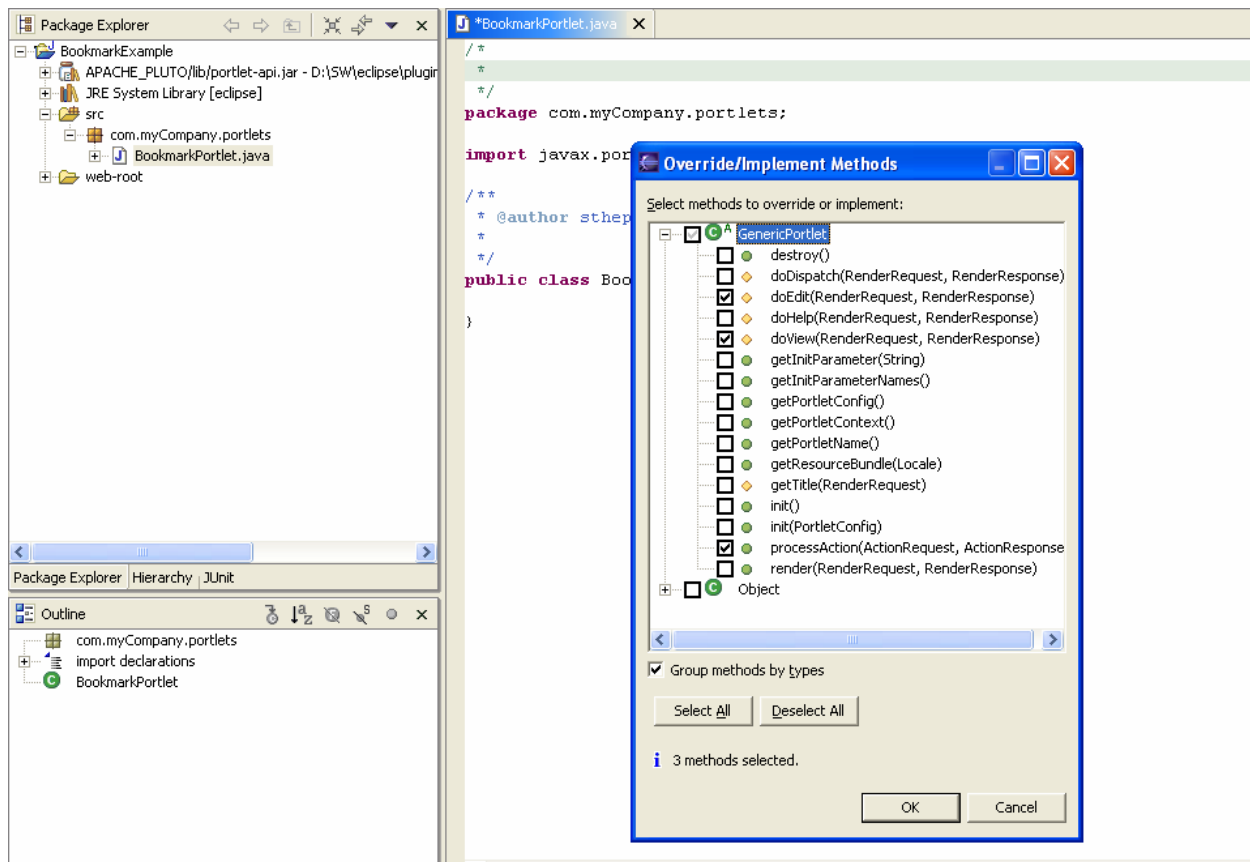


Figure 4.14: Adding the method stubs for action processing, Edit mode rendering, and View mode rendering

Eclipse adds the following code to `BookmarkPortlet.java`:

Listing 4.1: Autogenerated `BookmarkPortlet` stub code

```
public class BookmarkPortlet extends GenericPortlet {

    /* (non-Javadoc)
     * @see javax.portlet.GenericPortlet#doEdit(javax.portlet.RenderRequest,
     javax.portlet.RenderResponse)
     */
    protected void doEdit(RenderRequest request, RenderResponse response)
        throws PortletException, IOException {
        // TODO Auto-generated method stub
        super.doEdit(request, response);
    }

    /* (non-Javadoc)
     * @see javax.portlet.GenericPortlet#doView(javax.portlet.RenderRequest,
     javax.portlet.RenderResponse)
     */
    protected void doView(RenderRequest request, RenderResponse response)
        throws PortletException, IOException {
        // TODO Auto-generated method stub
        super.doView(request, response);
    }

    /* (non-Javadoc)
```

Autogenerated stub for
the Edit mode

Autogenerated stub
for the View mode

```

    * @see javax.portlet.Portlet#processAction(javax.portlet.ActionRequest,
    javax.portlet.ActionResponse)
    */
    public void processAction(ActionRequest request, ActionResponse response)
        throws PortletException, IOException {
        // TODO Auto-generated method stub
        super.processAction(request, response);
    }
}

```

**Autogenerated stub for the
action handling code**

Figure 4.15 shows what this code looks like in the Eclipse editor. As you can see in the main window the Bookmark portlet now consists of three stub methods for doView, doEdit and processAction that we need to customize further.

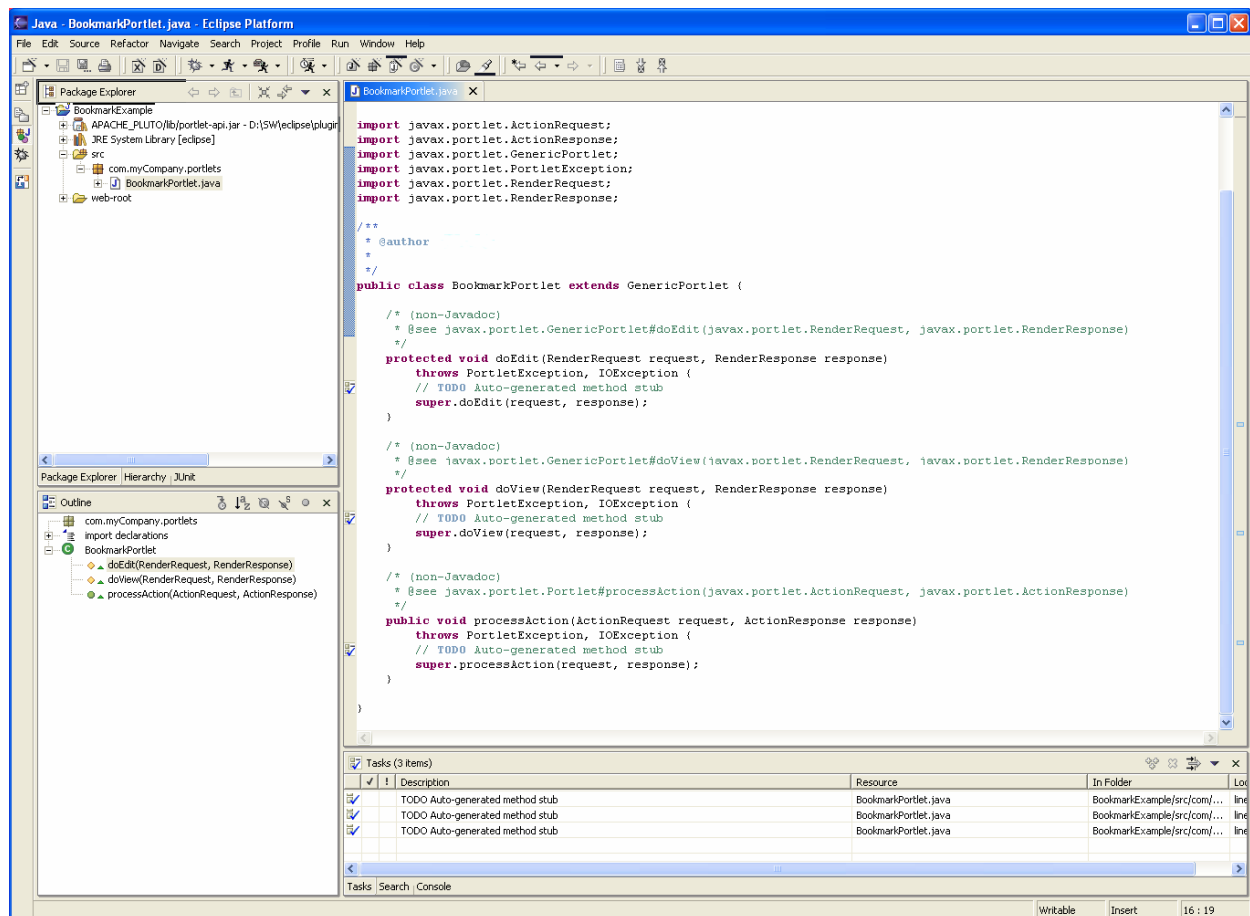


Figure 4.15: Portlet code with method stubs created for doEdit, doView, and processAction

Now, we add the code for these three methods, as we did back in Chapter 3 in Listings 3.1, 3.2 and 3.3..

The code in the doEdit method will create the “add” and “cancel” URLs and include the Edit JSP. The doView method includes the View JSP and the processAction method stores the new bookmark in the portlet preferences, or removes an existing one from the portlet preferences, depending on the user action. Figure 4.16 shows the Bookmark portlet after we add this code.

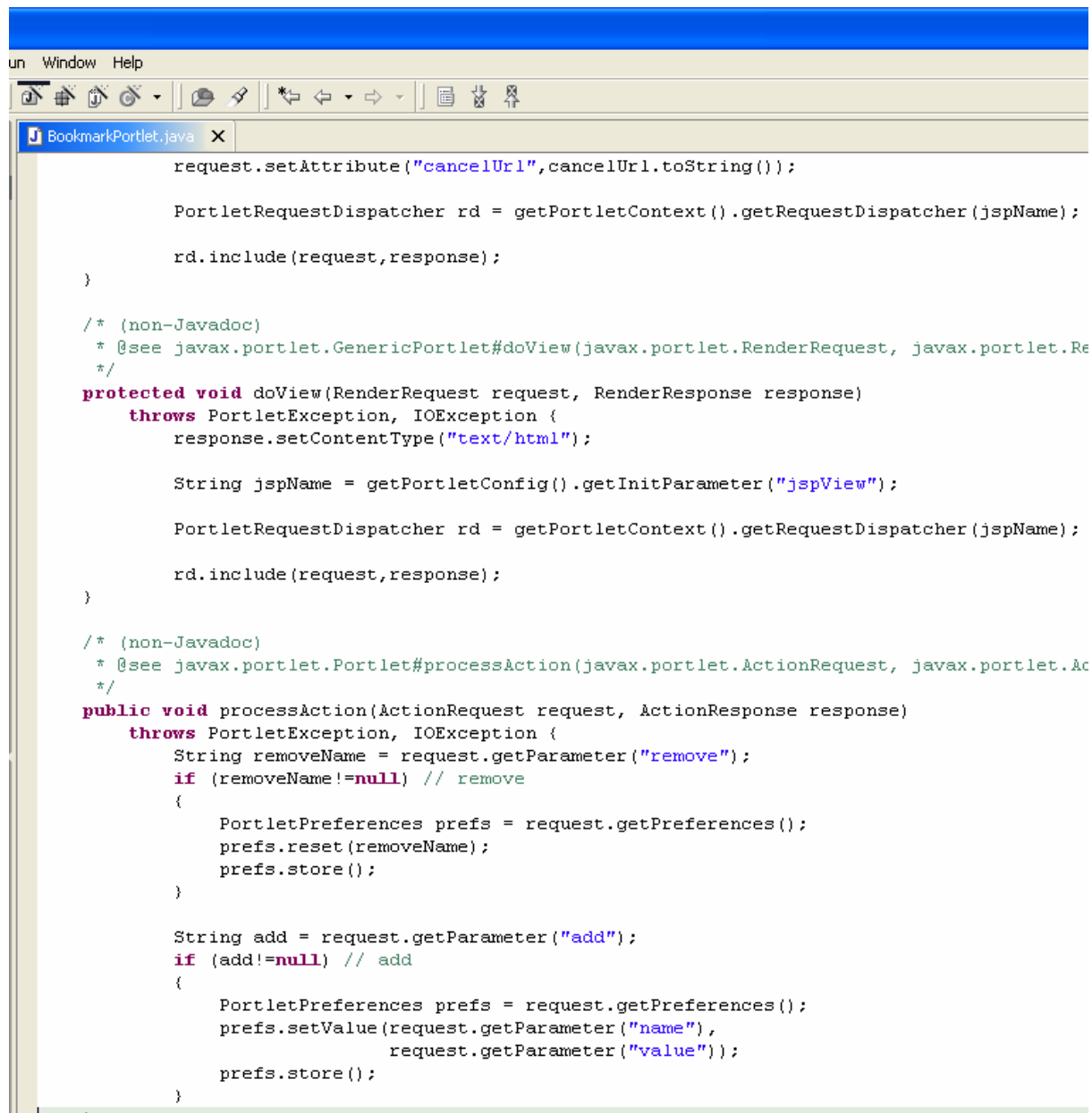


Figure 4.16: Complete portlet code for the Edit mode, View mode, and action processing

4.2.3 Adding a preference validator

Now we will introduce a new part that we didn't implement in the Bookmark example in chapter 3 in order to keep the example small. To ensure that the preferences are always consistent and adhere to certain restrictions, the portlet should also provide a preference validator. This validator will be called from the portlet container before the store method for the portlet preferences is executed, or it may be called by external systems that may populate the preferences. We will again create a new class, this time by clicking the green C icon in the toolbar. Figure 4.17 depicts the Eclipse class wizard that will help us create this validator class. Type the Name in: PreferencesValidatorImpl. Then click Add next to the Interfaces section. Under Choose Interfaces, type javax.portlet.PreferencesValidator, then click OK. Click Finish to create the new class.

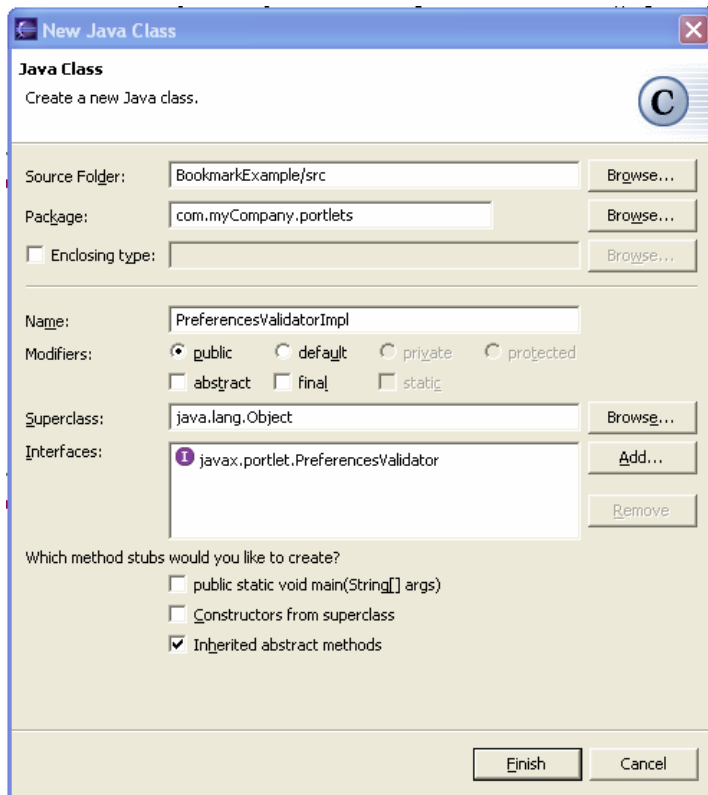


Figure 4.17: Eclipse class wizard for creating the preference validator class

Eclipse generates the following code:

Listing 4.2: Autogenerated Preference Validator code

```
public class PreferencesValidatorImpl implements PreferencesValidator {

    /* (non-Javadoc)
     * @see
     * javax.portlet.PreferencesValidator#validate(javax.portlet.PortletPreferences)
     */
    public void validate(PortletPreferences preferences)
        throws ValidatorException {
        Collection failedKeys = new ArrayList();
        Enumeration names = preferences.getNames();

        String[] defValues = {"no values"};

        while (names.hasMoreElements())
        {
            String name = names.nextElement().toString();
            String[] values = preferences.getValues(name, defValues);

            for (int i=0; i<values.length;i++)
            {
                if (!values[i].equalsIgnoreCase(values[i].trim()))
                {
                    failedKeys.add(name);
                    i = values.length;
                }
            }
        }
    }
}
```

← loop through all the preferences

← go through all values of a preference

← check that the value das not contain any white spaces at the beginning or end


```

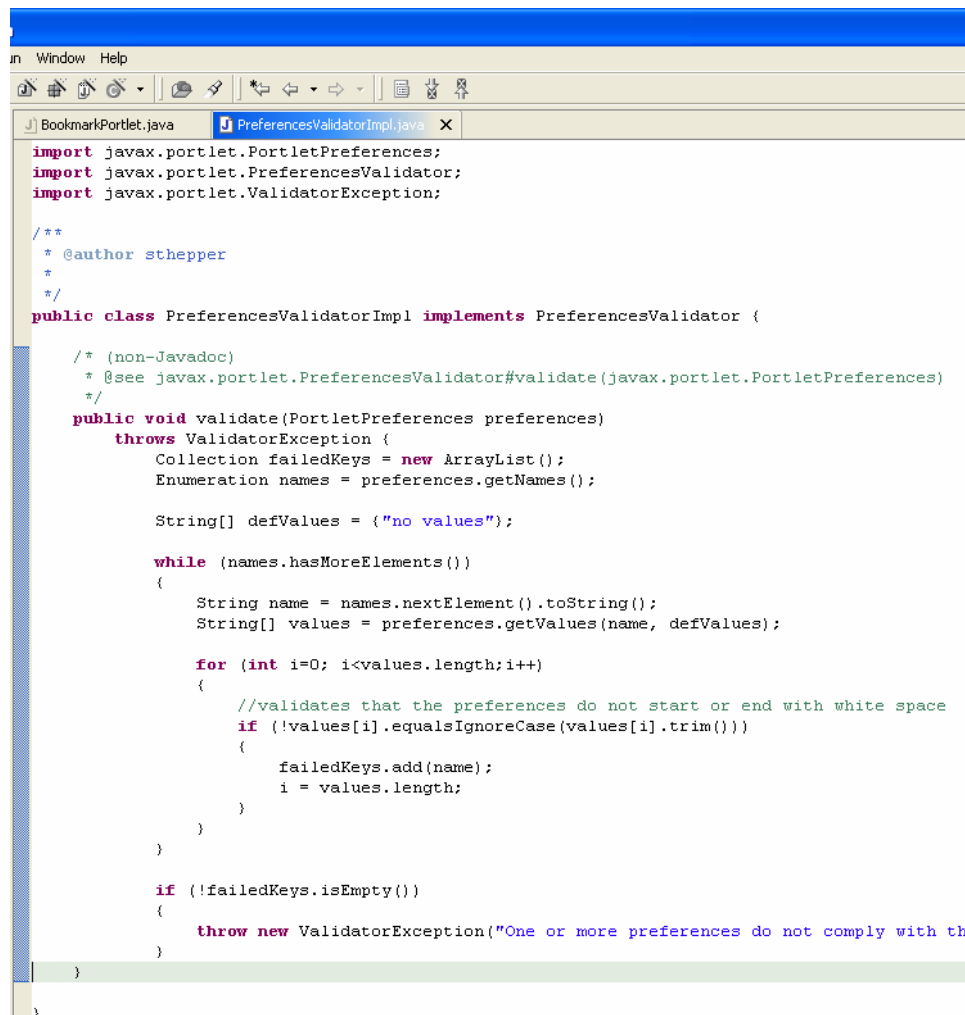
    }
}

if (!failedKeys.isEmpty())
{
    throw new ValidatorException("One or more preferences do not comply
with the validation criteria",failedKeys);
}
}

```

if some values failed the validation
throw an exception with the keys these
values belong to

Figure 4.18 shows the automatically generated code for the validator. The validator checks for whitespace at the beginning or end of the bookmark URLs and generates an error if one or more URLs do not comply with this rule. This is where IDEs really pay off, as the deployment descriptor is also updated automatically and the reference in the portlet.xml to the validator is added. Of course, you can now add more validation code specific for your application, like checking length restrictions or more complicated relations between different settings, such as having a country field and then a zip code field for the city, where the length of the zip code will depend on the selected country.



```

import javax.portlet.PortletPreferences;
import javax.portlet.PortletPreferencesValidator;
import javax.portlet.ValidatorException;

/**
 * @author sthepper
 *
 */
public class PreferenceValidatorImpl implements PortletPreferencesValidator {

    /* (non-Javadoc)
     * @see javax.portlet.PortletPreferencesValidator#validate(javax.portlet.PortletPreferences)
     */
    public void validate(PortletPreferences preferences)
        throws ValidatorException {
        Collection failedKeys = new ArrayList();
        Enumeration names = preferences.getNames();

        String[] defValues = {"no values"};

        while (names.hasMoreElements())
        {
            String name = names.nextElement().toString();
            String[] values = preferences.getValues(name, defValues);

            for (int i=0; i<values.length;i++)
            {
                //validates that the preferences do not start or end with white space
                if (!values[i].equalsIgnoreCase(values[i].trim()))
                {
                    failedKeys.add(name);
                    i = values.length;
                }
            }
        }

        if (!failedKeys.isEmpty())
        {
            throw new ValidatorException("One or more preferences do not comply with th
        )
    }
}

```

Figure 4.18: Preference validator code. The Preference Validator checks for whitespace at the beginning and end of bookmark URLs and generates an error if the URL is mal-formed.

Now we have finished the Java parts of our portlet and have the stub code for the View and Edit mode and the preferences validator. Next we'll create the JSPs that will produce the markup that our portal can render.

4.2.4 Creating the rendering by using JSPs

Now we will start creating the JSPs needed to render the content of the Edit and View mode. The Pluto plug-in automatically detects that we have overridden the `doEdit` and `doView` mode and therefore creates an Edit and View JSP stub, and places them in the `web-root/jsp` folder in the Package Explorer. The generated code looks like this:

Listing 4.3: Autogenerated JSPs for View and Edit mode

```
<%@ page session="false" %>
<%@ page import="javax.portlet.*"%>
<%@ page import="java.util.*"%>
<%@ taglib uri="/WEB-INF/tld/portlet.tld" prefix='portlet'%>
<portlet:defineObjects/>
```

This a very simple Portlet

Hope that the plugin made it easier to get started with writing JSR-168 portlets

Current Portlet Mode: <%=renderRequest.getPortletMode()%>

Current Window State: <%=renderRequest.getWindowState()%>

Figure 4.19 shows the Edit JSP stub that the Pluto plug-in has created.

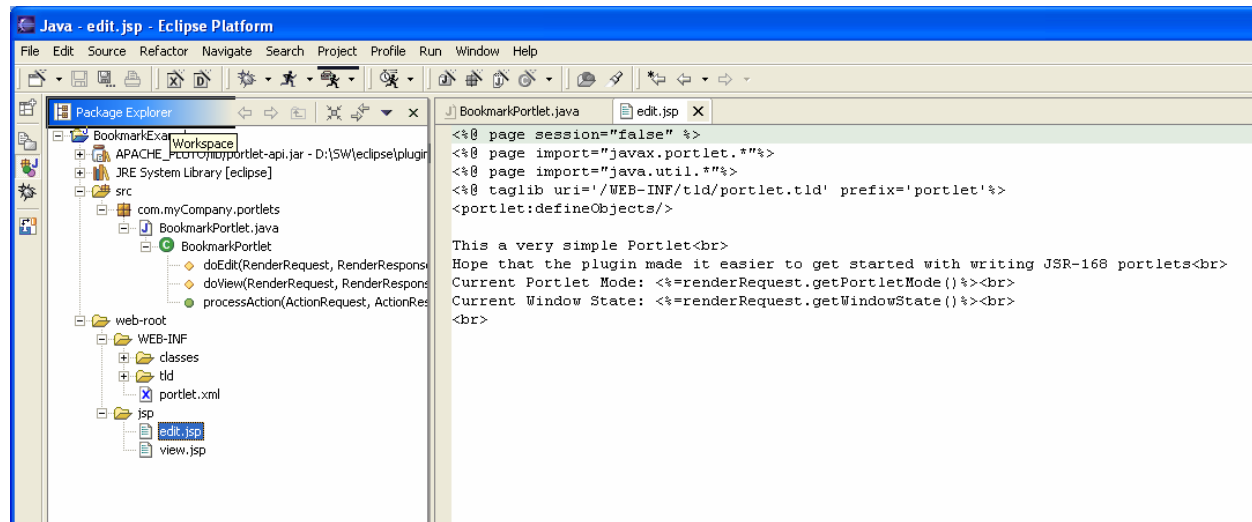


Figure 4.19: Edit JSP stub created by the Pluto plug-in

We now replace the Edit stub code with the code from our Bookmark sample to create the table with the current bookmarks, and the buttons to delete an existing bookmark, add a new bookmark, or cancel the edit operation. Figure 4.20 displays the final Edit JSP.

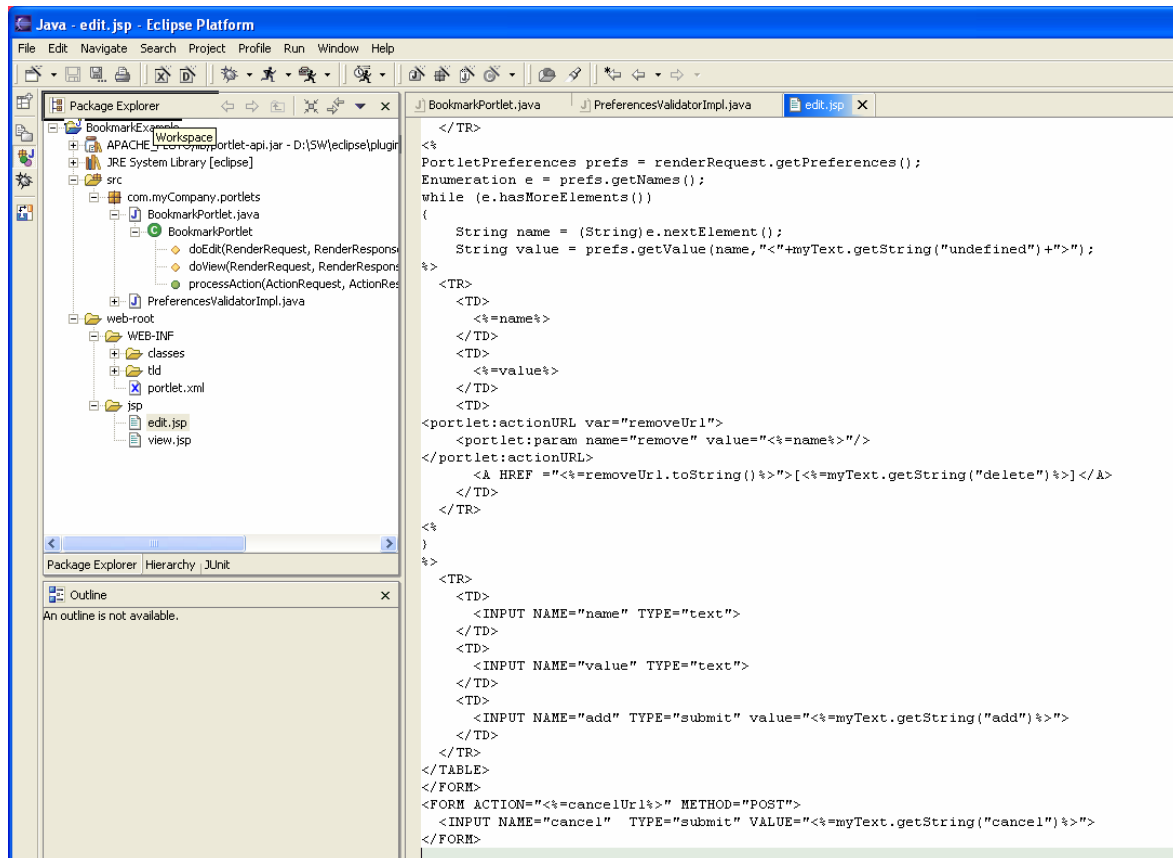


Figure 4.20: Edit JSP code when you have added the BookmarkPortlet code from Listing 3.5.

After we have finalized the Edit JSP we now need to replace the View JSP stub code with the code from Listing 3.4 to render the table of the current bookmarks. The View JSP code in Eclipse is depicted in Figure 4.21.

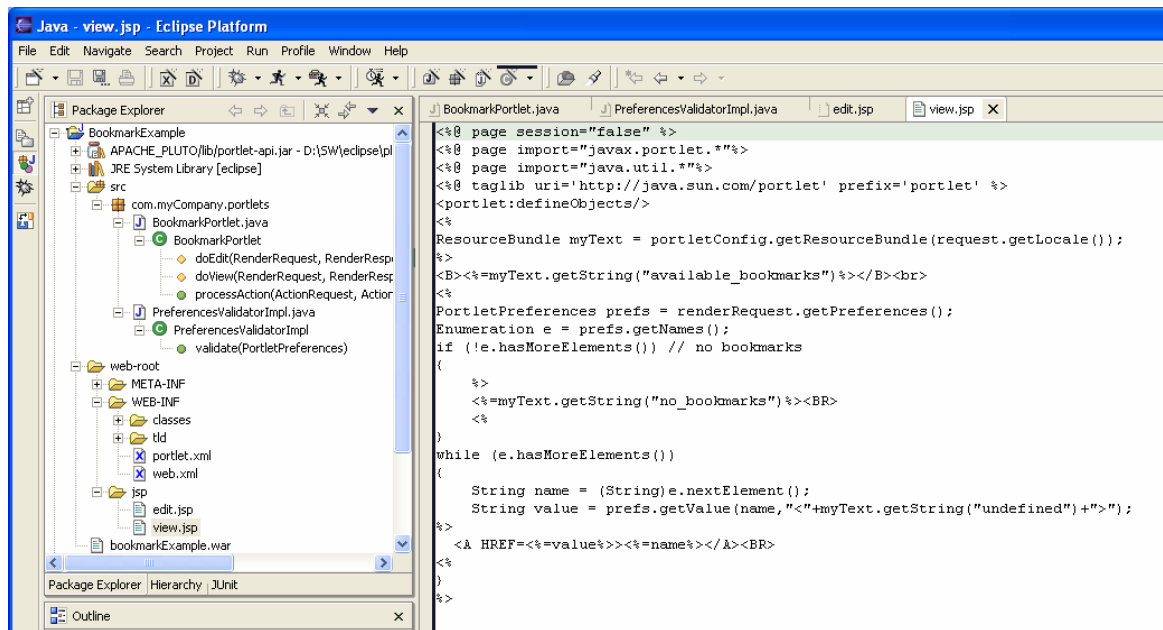


Figure 4.21: View JSP code

Now you've seen what Eclipse can do to simplify your coding process. We still need to create the necessary files to deploy our portlet.

4.2.5 Deploying the portlet

After we implemented the portlet code and the JSPs we will now create the deployment descriptors, web.xml and portlet.xml, and resource bundle files as we did in Chapter 3.. Currently, the Pluto plug-in does not support auto-generated resource bundles, therefore we need to add this by hand. First we will modify the auto-generated portlet.xml and add the missing information for the resource bundle support. The final deployment descriptor is shown in Figure 4.22, where we have added the supported locales English (en) and German (de) and the name of the resource bundle (portlet), as in Listing 3.7. We also added some other missing information we need in the resource bundle: the init parameters with the locations of the Edit and View JSP and a default bookmark (Pluto Homepage).

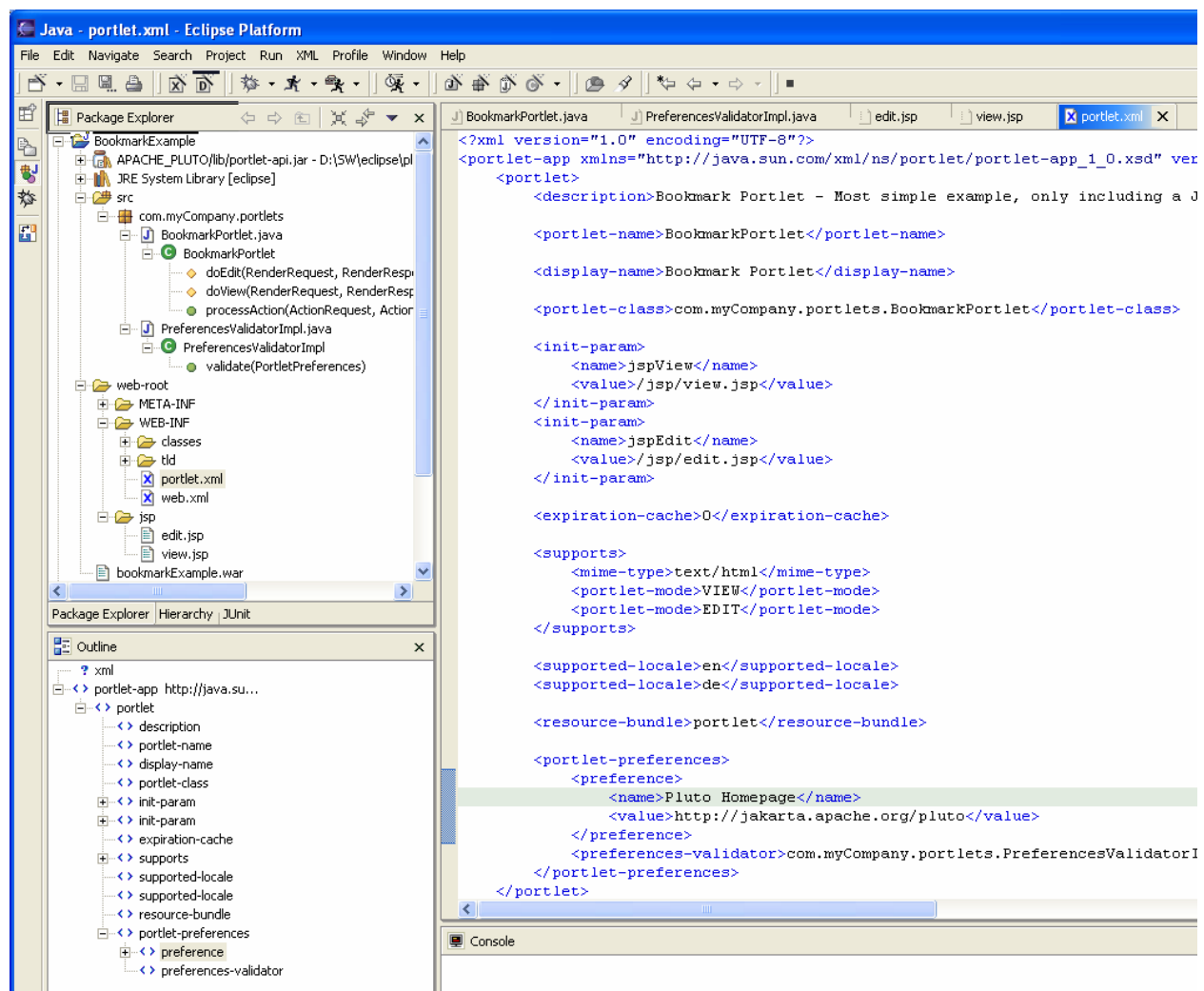


Figure 4.22: The portlet.xml deployment descriptor

Next, we will create the MANIFEST.MF and include the version of our bookmark portlet allowing portal servers to keep track of different versions of our bookmark portlet application. We need to do this manually, as the Pluto plug-in does not currently support this.

First, we need to create the META-INF directory in which the MANIFEST.MF file needs to reside, and then the file itself. Use the drop-down menu from the New toolbar icon to make a New Folder under web-root. Name it META-INF, and click Finish. Repeat the process to make a New File in the META-INF folder, called MANIFEST.MF. Enter the information from Listing 3.8 in the General Information box in the resulting page, and save the file to create your MANIFEST.MF file. .

The outcome of these steps is depicted in Figure 4.23.

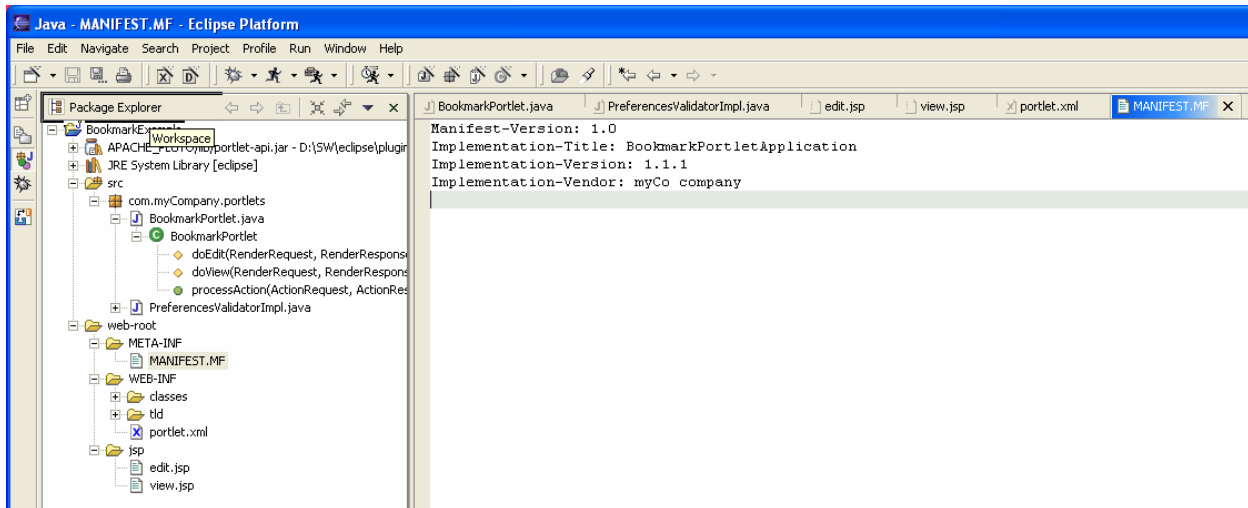


Figure 4.23: The manifest file containing the version number of the bookmark portlet application

On the left, in the package explorer, you can see the new folder META-INF with the new file MANIFEST.MF and on the right, the content of the MANIFEST.MF file is shown with the name, version, and vendor of this portlet application. As explained in Chapter 3 this version information is very important for portal deployers in order to update existing portlet applications with newer versions.

Now we have finalized the portlet.xml deployment descriptor and the MANIFEST.MF Meta information file. However, in order to provide the localization support, we need to provide the resource bundle files with the localized texts. Create the new files as we did with MANIFEST.MF. If it is not already there, create a classes folder under WEB-INF, then create the portlet_en.properties file. Figure 4.24 depicts the Eclipse dialog for creating the resource bundle for English.

We will do the same thing for the German resource bundle and call the file portlet_de.properties. For the portlet container to successfully find the resource bundles, the resource bundle file needs to start with the name specified in the portlet.xml deployment descriptor, in our example with “portlet.” After that the Java resource bundle naming conventions need to be followed, which means that the name is followed by an underscore and the country code (“en” and “de” in our example) and the file extension is “.properties”.

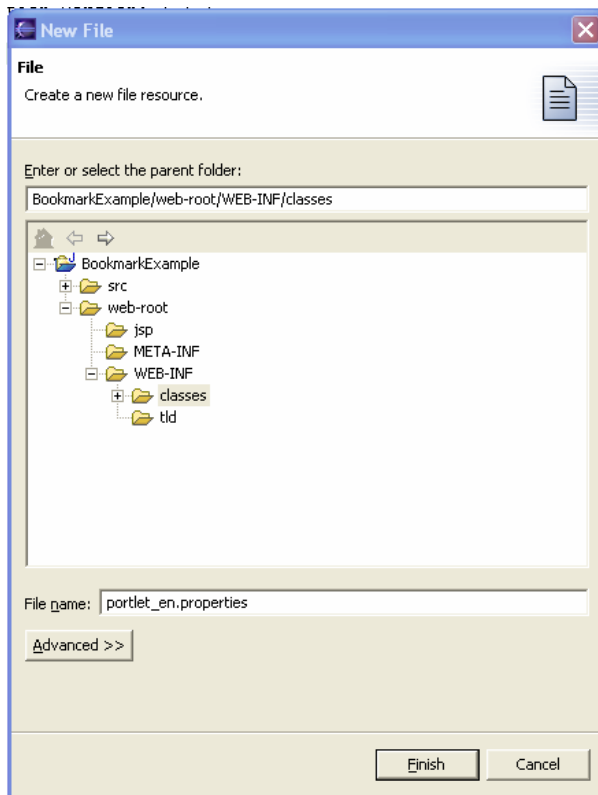


Figure 4.24: Creating the English resource bundle file

Now we will insert the title, keywords and the localized texts of the portlet. Figure 4.25 shows the English version, whereas Figure 4.26 shows the German resource bundle. See Listings 3.9 and 3.10 for the code.

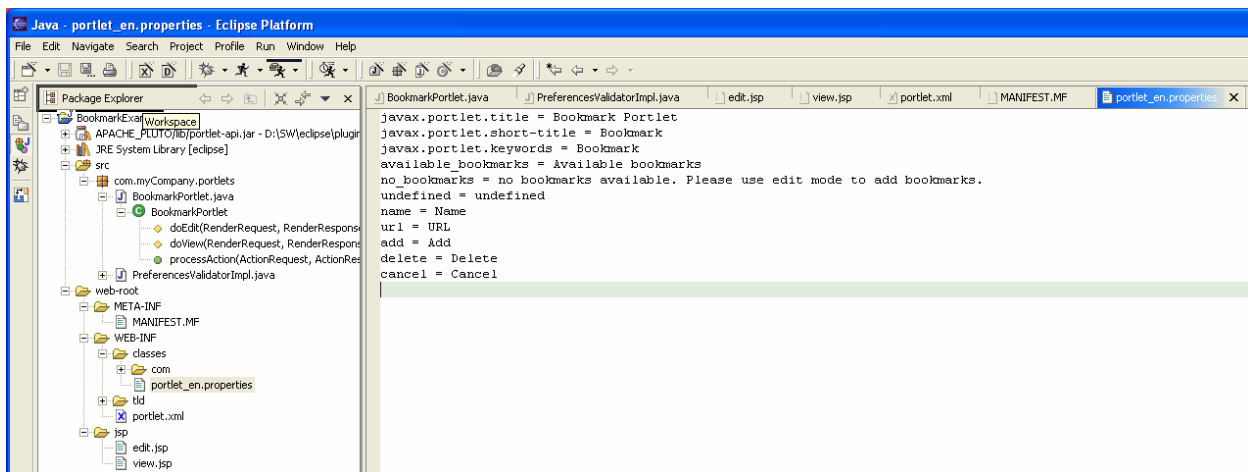


Figure 4.25: The English resource bundle file

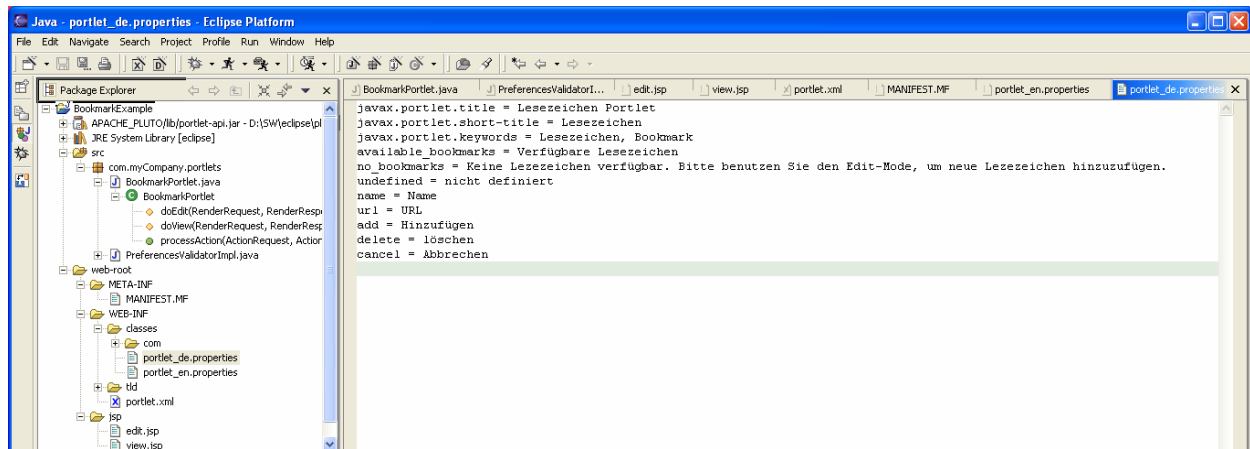


Figure 4.26: The German resource bundle file

The Pluto plug-in also auto-generates the web.xml file with the Pluto wrapper servlet. To create a portlet application that can be deployed on any JSR 168 compliant portal we will create our own web.xml and use the Pluto Maven script for deploying the portlet application (see Chapter 9 for how to deploy portlets on Pluto). We therefore remove everything from the web.xml deployment descriptor besides the web application name and description, as shown in Figure 4.27.

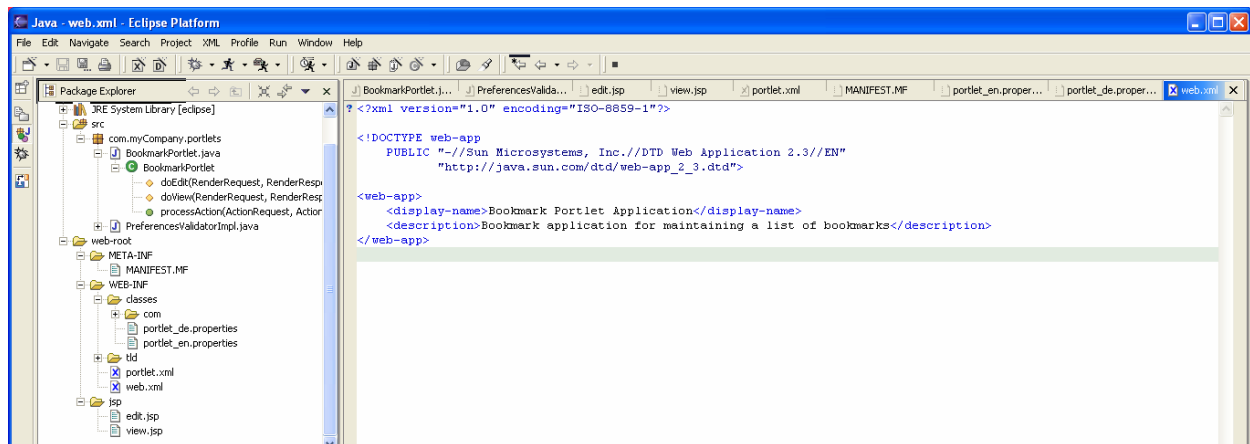


Figure 4.27: The web.xml file, without the Pluto-specific deployment code

Since the current version of the Pluto plug-in does not create a WAR file, we need to do this by hand. Everything below the web-root directory belongs to the WAR file, so we simply need to execute the following command in the web-root directory to create the WAR file:

```
jar cf BookmarkPortlet.war *
```

This WAR file can now be deployed on all JSR 168 compliant portals, like the Pluto reference implementation (see Chapter 7) or Jetspeed. (see Chapter 8).

In this section we explained how to create the Bookmark portlet example with Eclipse and version 1.0 of the Pluto plug-in. As you have seen it still required some tweaking by hand in order to end up with a portlet application WAR file that can be deployed on any portal. This will very likely change when the plug-in evolves or other plug-ins become available. To get an idea of what is possible, we will next take a look at what is already available in today's portlet development tools.

4.3 Commercial development tools

In the previous section, we re-created the Bookmark example of Chapter 3 using Eclipse and version 1.0 of the Pluto plug-in. As you have noticed in going through this example this first version of the plug-in has some limitations that will hopefully be removed in future versions of the plug-in. However, if you need a more powerful tool right now there is another option you may chose: use a commercial development environment

There are a variety of commercial tools available for portlet development, like BEA WebLogic Workshop, IBM Rational Application Developer, Oracle JDeveloper, Sun ONE Studio, and others. We will choose the IBM Rational Application Developer V6 (see [4]) for a closer look at advanced portlet development features as this one is also Eclipse-based and will thus show you what can be achieved inside the Eclipse framework.

Even if you use such a commercial tool you can deploy the created portlet application on any JSR 168 compliant portal server. For example, if you created your portlet application with Rational Application Developer, you can still run it on the Open Source Jetspeed portal. This is another great benefit that standardization brings you: the freedom of choice in tools and the run-time environment.

Now let us take a closer look at some features that IBM Rational Application Developer provides in addition to the functionality of the Pluto Eclipse plug-in.

4.3.1 Deploying portlets on-the-fly

All commercial tools have the corresponding portal and application server integrated. In the case of Rational Application Developer, these include the WebSphere Application Server and WebSphere Portal. This allows the developer to deploy new or updated portlets with one click on the integrated portal server without the need to stop the portal, deploy the portlet application and restart the portal. This dramatically reduces the time needed to test the portlet application. One click of the mouse will package the portlet application as a WAR file, start the portal server, deploy the WAR file on the portal server, and put the portlets in the portlet application on a test page.

This feature could also be implemented for open source tools. For example, with Eclipse, there could be a portlet development plug-in that adds tomcat and the Jetspeed portal and also allow this on-the-fly portlet deployment.

4.3.2 Support for MVC frameworks like Struts and JSF

As explained in Chapter 3, larger portlet application projects should base their applications on Model-View-Controller frameworks to keep the application maintainable and extensible. Commercial tools provide wizards that create stub code for integrating your portlet application with popular MVC frameworks. Right now there are two very popular MVC frameworks: the open source Apache Struts framework and Java Server Faces (JSF) that provide server-side UI components. JSF is explained in more detail in Chapter 5.

Let us take a look on how this MVC support looks in Rational Application Developer for JSR 168 portlets and JSF. Figure 4.28 depicts the Rational Application Developer wizard that allows creating portlet stubs. Note that you select the Faces portlet option to have your stub based on JSF. The other choices just create plain portlets. The IDE will now create a portlet that includes JSF for rendering and will allow you to put JSF components on the corresponding portlet JSPs.

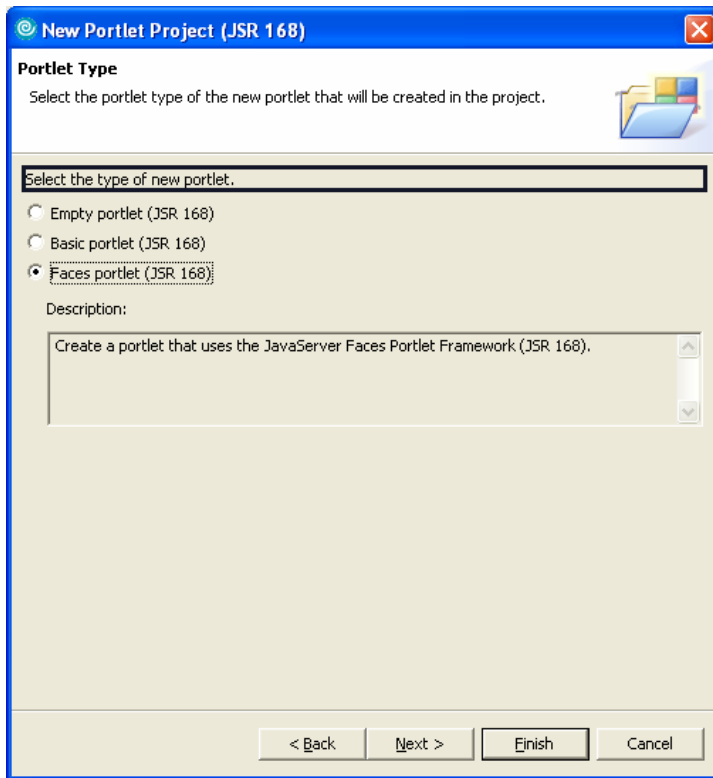


Figure 4.28: Creating a JSF-based portlet with Rational Application Developer

In order to show you the advanced tooling functionality we will create a simple example. The bookmark example is ported to JSF in Chapter 5 as a more complex JSF-based portlet application.

4.3.3 Developing portlets visually

Developing portlets visually is very attractive together with using a MVC framework, as these frameworks normally provide a lot of predefined visual components. This means that you can create portlets by dragging predefined components (e.g. JSF components) onto a screen and wire them together. This ability combined with wizards for the most common tasks beside the visual part result in quite complex portlets created without actually writing any Java code.

Figure 4.29 shows what the edit screen of our portlet example may look like in the visual development tool of Rational Application Developer. It offers special JSF components for portlets, like the text area and the command button that triggers a portlet action. In this example you can add an arbitrary text sting in the edit mode, which is stored in the portlet preferences and displayed in the view mode.

All in all you can create your different JSPs very easily by using this drag-and-drop feature and thus no longer need to type in the HTML code for the table, the portlet tags for creating the action and render links and Java code for accessing the portlet preferences.

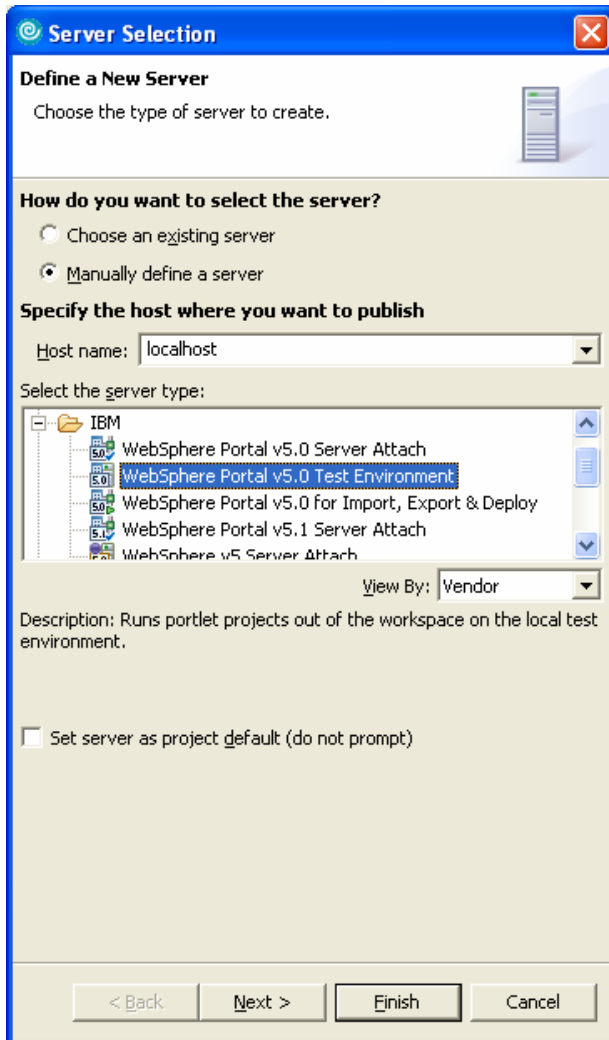


Figure 4.30: Running a portlet in the IDE with an integrated portal

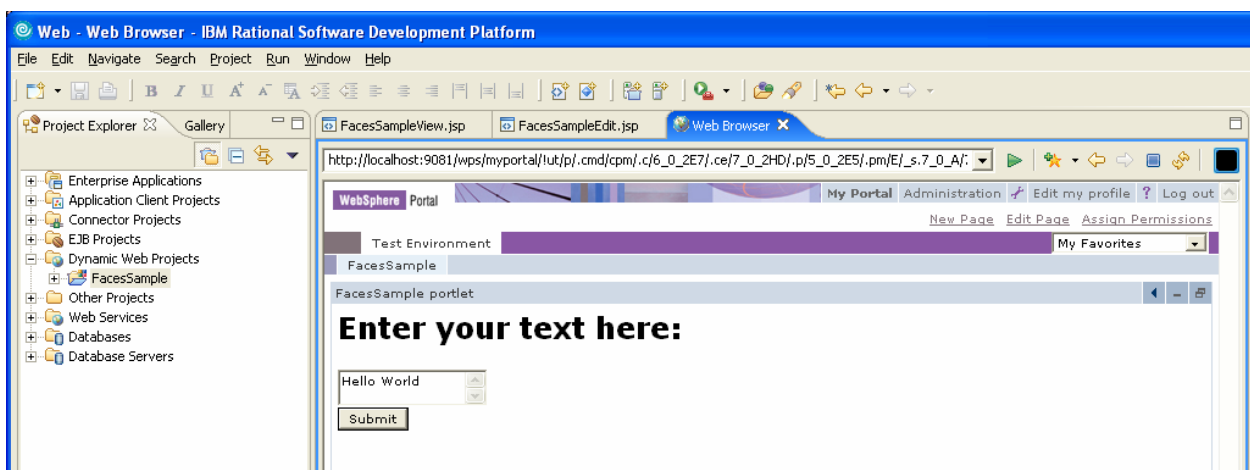


Figure 4.31: The bookmark example running as JSF portlet in Rational Application Developer

Until now we always talked about portlet applications. In the next section we will broaden our scope a bit and take a look at what you can do in Rational Application Developer to develop complete portal applications.

4.3.4 Creating portal applications

As the Java Portlet Specification only covers the portlet applications and no portal-related artifacts, whole portal application development is vendor-specific and not portable. If you have larger portal applications including several portlet applications that group related portlets, and/or include data access or other services that the portal needs, it can be very convenient to pre-package and test this portal as a whole. Similar portal application development tools would be something nice to have for Open Source portals, like Jetspeed. Maybe there will be some Eclipse plug-in available sometime that would allow this.

Most commercial tools for portlet development are targeted towards a specific portal, and some of them also allow you to create complete applications for that portal: you can create pages and content hierarchies together with your portlet application. These pages and artifacts can then be exported and imported into WebSphere Portal, if you are using Rational Application Developer. Portal designers may also create themes and skins of portal pages and export them.

Figure 4.32 shows how portal application development looks in Rational Application Developer.

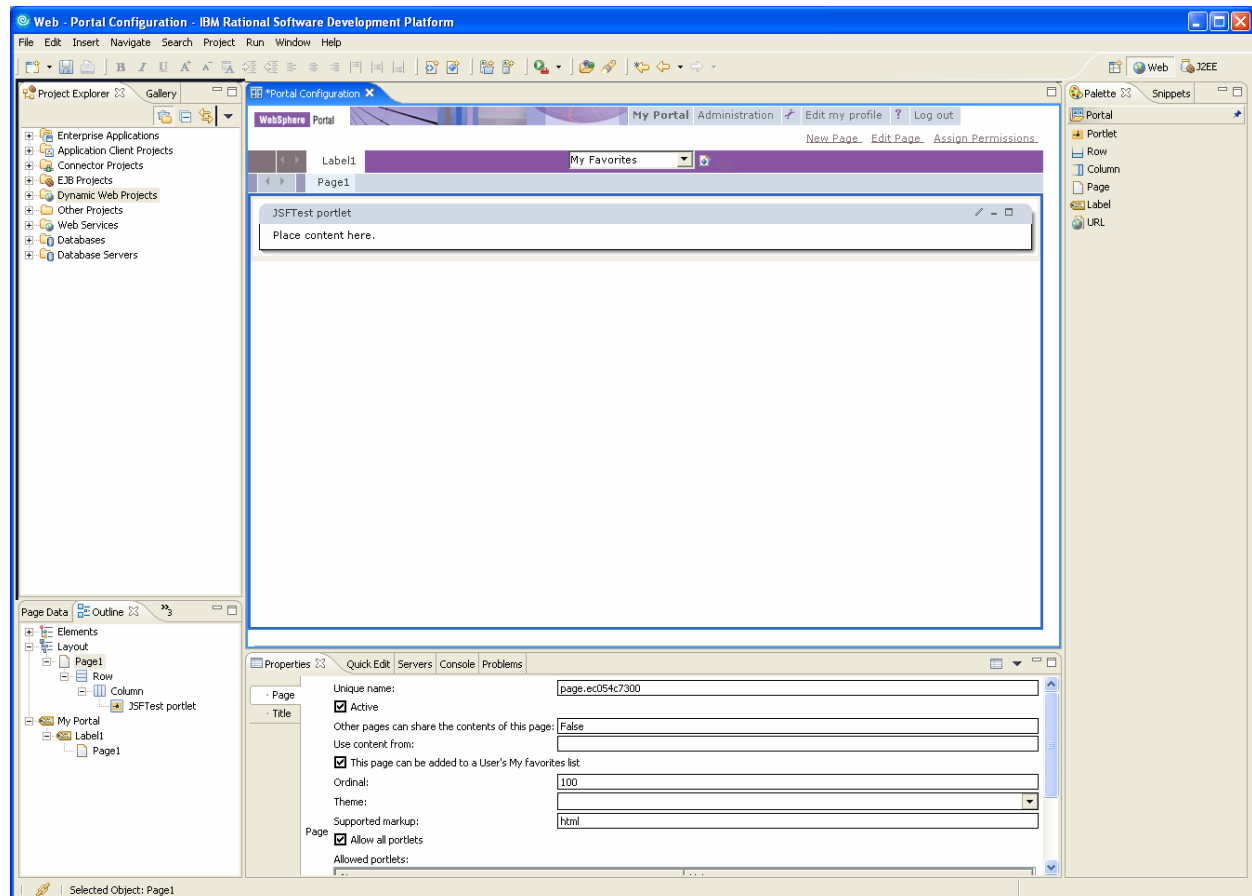


Figure 4.32: Portal application development with Rational Application Developer

The left side shows the navigator view of the portal project and the running application can be seen in the middle. Below the main window in the middle there is a window that allows customizing the current page attributes, and on the right side are portal components, like portlets, row or column containers, that can be added to the current page via drag-and-drop.

You can see that commercial tools can be a powerful addition to your portlet development arsenal. They cover many more advanced use cases and help you creating complete portal applications.

4.4 Summary

In this chapter, we showed you how tools can help you develop portlets more efficiently. We have done this using the most widely adopted Open Source Java development environment, Eclipse, and the Pluto plug-in that adds portlet development support. We reused the Bookmark example from Chapter 3 to show the differences between development with just an editor and one with an IDE where a lot of wizards already generate the standard code fragments needed.

Another big advantage of tools-based development is the fact that the tool can automatically keep all the deployment descriptors in sync with the functionality your portlet deploys, which can be quite a challenge when doing this by hand.

Then we showed very useful additional functionality that commercial portlet development tools, like Rational Application Developer, already support. This includes on-the-fly deployment of the developed portlet application in the IDE and support for visual development and Model-View-Controller frameworks, like JSF. By using simple example again and creating it as JSF-based portlet application in the visual development mode, we showed how easy and fast this can be realized.

Now that we have covered the portlet programming model in Chapter 3 and tools that allow creating your portlet applications efficiently we will take a closer look in Chapter 5 on how the Model-View-Controller framework JavaServer Faces (JSF) fits well into the portlet model. As already shown in the advanced tooling section of this chapter, these frameworks provide a lot of pre-defined visual components that make developing your JSPs much easier. But there is more to it—these frameworks also help you with the control flow of your application.

4.5 References

3. [1] Eclipse; URL: <http://www.eclipse.org/>
4. [2] Evans Data Research about popularity of Eclipse; URL: http://www.evansdata.com/n2/pr/releases/EMEAAPAC04_01.shtml
5. [3] Eclipse plug-in for Apache Pluto; URL: <http://plutoeclipse.sourceforge.net>
6. [4] IBM Rational Application Developer; URL: <http://www.ibm.com/software/awdtools/developer/application/index.html>

5 Using JavaServer Faces with Portlets

In this chapter, we explain how the JavaServer Faces (JSF) technology can be used inside of portlets and how the different portlet concepts can be applied to JSF. We start by first bringing you up to speed about JSF and by briefly touching on portlets.

In the second section, we will demonstrate how you can easily convert an existing JSF application into a portlet application. In fact, we will use several demo applications that come along with the two different JSF implementations: MyFaces and the SUN Reference Implementation.

In the third section we put the learning into practice and run the demo applications that we converted into portlets on Pluto and Jetspeed-2.

So that you can use more sophisticated portlet features inside your JSF Portlet, we will iterate through most of the important portlet concepts in the fourth section, and explain their relation to the JSF technology. This will show us whether we can simply reuse the JSF technology or if we need to adapt it to portlets.

In the end we use the already well-known bookmark example to demonstrate how to leverage the JSF technology inside of a portlet.

5.1 Using JavaServer Faces

Applications based on JSPs can typically be developed easily and in a short amount of time. However, as soon as the application becomes more complex, such as when adding enhanced navigation or extending the business logic, it quickly becomes difficult to manage.

As a means to manage this complexity, the J2EE enterprise application community invented the Model 2 Architecture, an extension to the MVC architecture. There are many open-source frameworks available for MVC, including Struts and Velocity, however until recently a standard was lacking. The solution arrived with the new standard called JavaServer Faces.

This section briefly introduces JSF by giving an overview and explaining the advantages of the new technology. There are already a lot of JSF books on the market that you can use to familiarize yourself better with the technology [11]. At the end of this section we will learn that JSF can also be used for portlet development helping to solve several issues concerning maintenance and development process.

Please note: if you are familiar with JSF you can directly jump to section 5.1.3 since we introduce JSF and its features in the next two sections. These sections provide a brief overview of selected concepts of the JSF technology that are related to portlets and do not claim to be a complete JSF description. It is recommended that you read additional background material if you are not familiar with JSF.

5.1.1 JSF Overview

JavaServer Faces (JSF) technology is a user interface framework for J2EE applications. It is particularly suited, by design, for use with applications based on the MVC architecture. JSF began as Java Community Process

(JSR-127), and was finalized in March 2004. A reference implementation is available for download at the JavaServer Faces Technology Download page [4]. An open-source implementation called MyFaces is also available from SourceForge [2] [3]. JSF is still not yet a standard part of J2EE, however it is already supported by all major platform vendors.

The scope of the JSF technology is ambitious – some of the features are highlighted in the following list:

- Managed Bean Facility for translating input events to server-side behavior
- Validation Facility to validate input values
- Rich and Extensible Component Library also designed for easy tools integration
- Pluggable render kits to support multiple client types and markups
- Navigation in Response to Specific User Events
- Preserving Application State Across Requests

Knowing that JSF has these features is necessary to understand its purpose and the issues it addresses. But how do you actually use this technology inside of your web application?

The answer is simple: With a servlet. JSF defines a controller servlet that, in fact, is part of the JSF API and therefore part of the distribution package. The developer references this servlet inside of a web.xml. Additional configuration parameters can be defined as well by binding configuration parameters to either servlet or context. Listing 5.1 shows the part of the web.xml that needs to be added for JSF.

Listing 5.1: Web.xml snippet of a JSF web application.

```
<!-- Faces Servlet -->
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.faces</url-pattern>
</servlet-mapping>
```

This listing shows plain JSF and how it is being used in a plain web application. Later within this chapter we will start looking into how JSF can be combined with portlets.

In the next section we dive into some theoretical details of JSF that are very important to know for this chapter.

5.1.2 Request Lifecycle

One of the main concepts of JSF is the lifecycle and eventing system, which is pretty much the foundation of JSF. The lifecycle of a JSF request is processed in six phases as depicted in Figure 5.1. Built within these six phases an eventing system allows the developer so send and receive events. Its purpose is to allow JSF components to exchange messages between each other.

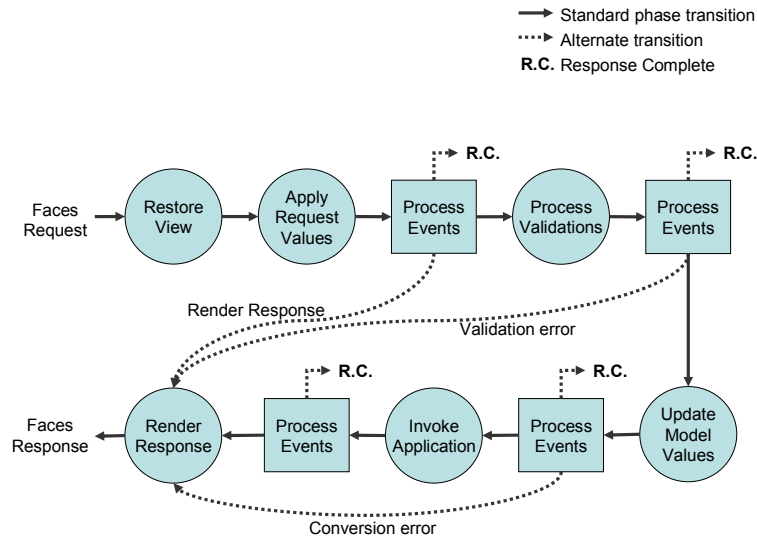


Figure 5.1: Lifecycle of a JavaServer Faces request.

A more detailed description about the different lifecycle phases is given in Table 5.1:

Table 5.1 The Phases of the JSF Lifecycle

Phase	Description
Restore View	A JSP in a JSF application is represented by a component tree. The Restore View phase starts the Lifecycle request processing by constructing this tree. The identifier of this component tree is the path information portion of the request URL and therefore to the view. This information is then saved and passed on to following request processing phases.
Apply Request Values	In the Apply Request Values phase, the local value of each component in the component tree is updated from the current incoming request. A value can come from a request parameter, a header, etc. During this phase, a component may queue events, which will then be processed during the process event steps. In case of an error the request phase is interrupted and proceeds directly with the Render Response phase.
Process Validations	After the local value of each component is updated, those values are validated if necessary in the Process Validations phase. A component that requires validation must provide an implementation of the validation logic. If any value does not pass the validation the request phase is interrupted and proceeds directly with the Render Response phase.
Update Model Values	In this phase, the Lifecycle object updates the application's model data. During this phase, a component may again queue events, such as conversion errors.
Invoke Application	During this phase, the JSF implementation handles any application level events, such as submitting a form or linking to another page.
Render Response	In this phase, the JSF implementation renders the response to the client.

The table discussed all the lifecycle phases of a JSF request. In JSF this is the most important concept since it define the whole concept in which all other JSF techniques work. All other artifacts of JSF are based on JSF components which are called by each phase and therefore can react on each with a different logic.

The next section will discuss how JSF can be applied to portlets and which problems to expect.

5.1.3 JSF applied to Portlets

In the previous sections, we talked about JSF in general, how it relates to web applications, and how to actually use it inside of a web application or servlet. In this section we will expand the topic to include portlets as well.

Portlets are normal web artifacts that reside in web applications, like servlets. They have the same advantages and disadvantages as any web application. One disadvantage is that the application becomes less and less maintainable the bigger it gets, the overview is easily lost and there is no role separation allowing

multiple people to work on one big application. All these reasons clearly show that we not only need to apply frameworks (such as JSF) that solve these problems to servlets, but also to portlets.

Interfacing with Portlets

As we already learned in this book, the Java Portlet Specification explicitly tried to, and succeeded in, aligning portlets with servlets and reusing as much existing J2EE technology as possible. The JSR 168 and JSF expert groups interlocked with each other and enabled us to use JSF not only inside servlets, but also inside portlets. As the result of the interlock, the JSF Expert Group introduced the concept of an *External Context* which allows embedding JSF into nearly every technology, such as portlets. Please see section 6.1.2 of the JSF Specification [4] for more details.

The External Context provides access to all of the components defined by the portlet runtime within which the JSF web application is deployed. This class must be implemented along with the `FacesContext` class, and must be accessible via the `getExternalContext` method in `FacesContext`. As described above, the JSF Specification only distributes a servlet controller – therefore we need to implement a portlet controller additionally to both contexts.

Luckily, we don't need to implement all of this ourselves, as there are already implementations available from different groups. In this book we will leverage the open-source implementation of Jetspeed-2 working with MyFaces. The following listing shows the code that we need to add to the `portlet.xml` file to enable JSF support inside of a portlet.

```
<!-- Faces Portlet -->
<portlet>
  <portlet-name>BookmarkPortlet</portlet-name>
  <portlet-class>
    org.apache.portals.bridges.myfaces.FacesPortlet</portlet-class>
  ...
</portlet>
```

The portlet `FacesPortlet` of the MyFaces implementation takes care of handling the `ExternalContext` and therefore we can simply rely on this implementation.

Applying the Lifecycle Phases

In the previous section we explained the Lifecycle phases of JSF and what they are used for. By now you also know that the Java Portlet Specification introduced two phases: action and render. The two phases are less sophisticated but nevertheless map onto the JSF phases quite well.

The Java Portlet Specification defines that model and state changes have to occur during the action phase and that the render phase is only supposed to be used to render the view of the underlying model. Therefore, a faces controller for portlets, such as the one from Jetspeed-2, executes all phases besides the Render Response phase inside of the portlet action phase. And in the portlet render phase it executes the Render Response Phase only.

You now know a little bit about JSF and how to integrate JSF into your portlet structure. But what if you already have some JSF content? Can you make that into a portlet? Yes, you can, and we'll do it in the next section.

5.2 Converting existing JSF Applications to Portlets

So far we have learned a little bit about JSF and more important how it all relates to portlets. In the upcoming sections we delve deeper into the portlet concepts and we will see how they apply to JSF.

Our first step into the JSF-Portlet world is based on already existing JSF applications that we will convert into portlets. This is a very common scenario that, hopefully, you can use and leverage a lot in your beginnings of using JSF Portlets and later as the need arises. If you currently have existing JSF servlet applications that you want to convert into a JSF portlet application, this section provides you with exactly the information you need to perform this step.

By way of example, we will use the JSF-CarDemo application, which is being distributed with the SUN Reference Implementation, to show how to convert a JSF servlet application into a portlet. Download JSF-CarDemo from the JSF Homepage [4]. Figure 5.2 shows how the CarDemo looks like.

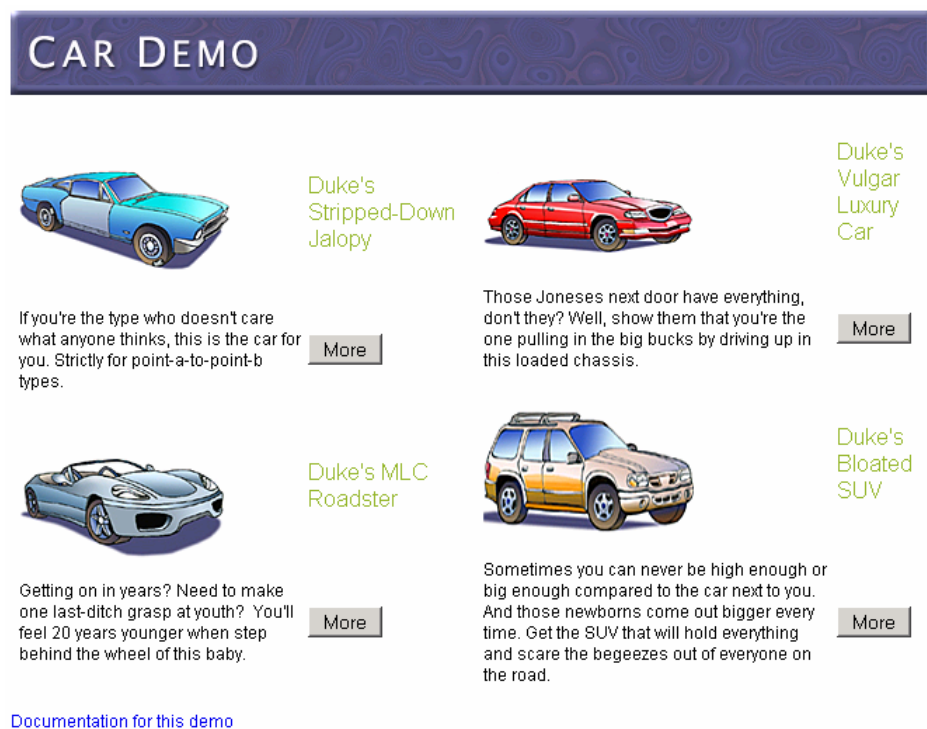


Figure 5.2: JSF-CarDemo example distributed with the SUN Reference Implementation. In this section, we will convert this JSF content into a portlet.

Table 5.2 lists the steps that you need to complete to convert a JSF servlet application into a portlet:

Table 5.2: Converting JSF Content to a Portlet	
Step	Description
Setting up	Backup the WAR file
Preparing to run on MyFaces	Change Web Deployment Descriptor Repackage JAR files
Becoming a Portlet	Add Portlet Deployment Descriptor Repackage JAR files
Finishing and Cleaning Up	Remove Document Tags Jetspeed-2 Faces Bridge Constraints

Now, let's look closer at each step.

5.2.1 Setting Up

To start the conversion we duplicate the WAR file of the original JSF servlet application to create a backup. In case any step would not succeed as anticipated we would still have a working original that we could start from again.

In this example we copy the WAR file jsf-cardemo and rename it to jsf-cardemo-portlet.

5.2.2 Preparing to run on MyFaces

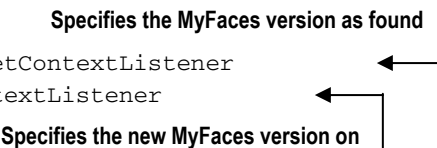
Since we will not leverage the JSF SUN Reference Implementation inside of portlets, we first replace the existing SUN JSF implementation with the MyFaces implementation. MyFaces can be downloaded at their homepage [2]. Please note that, as we are writing this, the MyFaces project pages are being moved from the Sourceforge repository to the Apache Software Foundation. More information can be found at the Apache incubator [3].

Proceed with the following steps to convert the demo application to MyFaces.

Change the Web Deployment Descriptor

Add the content shown in the following listing to the web.xml of the jsf-cardemo-portlet application. The listener tag needs to be added before the servlet tags within the web.xml. MyFaces uses this technique to plug itself into the JSF technology. Only one of the class names is required!

```
<listener>
  <listener-class>
    net.sourceforge.myfaces.webapp.StartupServletContextListener
    org.apache.myfaces.webapp.StartupServletContextListener
  </listener-class>
</listener>
```



Repackage the JAR files

Next, we replace the Sun JSF Reference Implementation with MyFaces. In order to do this, remove the following JAR files from the WEB-INF/lib directory:

- jsf-api.jar
- jsf-impl.jar

Then, add these JAR files from the MyFaces distribution:

- myfaces-jsf-api.jar
- myfaces.jar
- myfaces-components.jar
- commons-codec-1.2.jar

This is the minimal set that is required to run the CarDemo application. Depending on the richness of the functionality that is used inside of a JSF application, you might need to copy more JAR files from the MyFaces distribution into the WEB-INF/lib directory.

Once you've done this, it would be a good time to test the converted application as is on Tomcat 5 (or any other application server) to make sure that the application still works purely based on servlets.

Note: If you have trouble running CarDemo, you'll appreciate having created a backup WAR file when we got started!

The next steps will turn the servlet based JSF application into a portlet.

5.2.3 Becoming a Portlet

Now that we have a working CarDemo application running with MyFaces we will convert it into a portlet. In section 5.3 we will show you how to run the converted application.

Add Portlet Deployment Descriptor

Finally we get to the heart of the conversion, creating the portlet.xml. With this core piece we will turn the servlet into a portlet that can be consumed by portals such as Jetspeed-2. Listing 5.2 shows the portlet.xml that we used for the CarDemo application. Create the file portlet.xml with the following content in the WEB-INF directory:

Listing 5.2: Portlet.xml to be added.

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-app xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd"
version="1.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd
http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd">
  <portlet>
    <portlet-name>CarDemoPortlet</portlet-name>
    <display-name>CarDemoPortlet</display-name>
    <portlet-class>
      org.apache.portals.bridges.myfaces.FacesPortlet
    </portlet-class>
    <init-param> 2
      <name>ViewPage</name>
      <value>/chooseLocale.jsp</value>
    </init-param>
    <expiration-cache>-1</expiration-cache>
    <supports>
      <mime-type>text/html</mime-type>
      <portlet-mode>VIEW</portlet-mode>
    </supports>
    <supported-locale>en</supported-locale>
    <portlet-info>
      <title>CarDemoPortlet</title>
      <short-title>CarDemoPortlet</short-title>
      <keywords>Car, Demo, JSF, Portlet</keywords>
    </portlet-info>
  </portlet>
</portlet-app>
```

**Generic portlet class
that is being provided
by the Jetspeed-2 Faces
_...**

**Initial view to use when
the portlet mode view is
displayed**

There are two aspects of the listing that are worth pointing out. As a portlet class we use the generic portlet class that is being provided by the Jetspeed-2 Faces Bridge.(1) The init parameter (2) defines the initial view that the JSF implementation is supposed to use when the portlet mode view is displayed.

We nearly made it. We have switched to MyFaces and the portlet is converted, now we need to put everything together again in one application.

Repackage JAR files

Since we reference the Jetspeed-2 Faces Bridge inside of the portlet.xml, we also require the corresponding JAR files to be available in the application. Therefore, we copy the following JAR file into the WEB-INF/lib

directory. You can find them in the Jetspeed-2 build directory under portals-bridges/common/target and portals-bridges/myfaces/target.

- portals-bridges-common-0.1.jar
- portals-bridges-myfaces-0.1.jar

The main work of the conversion is now done. There are still a few minor miscellaneous items to finish before we can successfully run the application. We will finish in the next section.

5.2.4 Finishing and Cleaning Up

The last section of the JSF conversion guides you through some important miscellaneous steps. Without them the whole CarDemo application would not run or run appropriately.

Remove Document Tags

JSF-CarDemo produces a document as written. Portlets do not deliver documents in response to a portal request, they deliver markup fragments. Since our application is now a portlet we also need to consider how to return a markup fragment and not a document. Fortunately, this does not require much effort. Therefore, we remove or comment out the following tags inside of the index.jsp:

- html
- header
- body

The following listing shows the new adapted index.jsp.

```
<!-- html -->
  <!-- head -->
  <!-- /head -->
  <!-- body -->
  ...
  <jsp:forward page="chooseLocale.faces" />
  <!-- /body -->
<!-- /html -->
```

Work around Jetspeed-2 Faces Bridge Constraints

The Jetspeed-2 Faces Bridge is still in its beta phase and therefore a couple of shortcomings are quite normal. In our scenario we need to work around two of these.

Modify Web Deployment Descriptor—Once again we apply a minor change to the web.xml of the demo application. This special change is required by the Jetspeed-2 Faces Bridge that we are using to enable our portlet for JSF. It requires the extension *.jsf rather than *.faces. Table 5.3 shows the change that needs to be applied.

Table 5.3 We need to modify the web.xml file for the Jetspeed-2 Faces Bridge

Old servlet mapping	New servlet mapping
<pre><servlet-mapping> <servlet-name>Faces Servlet</servlet- name> <url-pattern>*.faces</url-pattern> </servlet-mapping></pre>	<pre><servlet-mapping> <servlet-name>Faces Servlet</servlet- name> <url-pattern>*.jsf</url-pattern> </servlet-mapping></pre>

Note: This change is required because of an assumption that has been made in the Jetspeed-2 Faces Bridge. In the future it is likely to be changed to either *.faces, or to a configurable value. If you get an

exception where the system complains, for instance, that it does not find `chooseLocale.faces`, then it has been changed and the pattern needs to be `*.faces` again. Please see the Jetspeed-2 documentation about the current status.

Adapt all references to the `outputLink` tags—The second problem we need to work around is the `outputLink` tag that currently is not working together with the Java Portlet Specification. There are two ways on how this problem can be solved. The first one is by simply not using this tag anymore and commenting it out. Here's how to comment out the `outputLink` tag.

```
<!-- h:outputLink value="javadocs" -->
    <!-- f:verbatim>Documentation for this demo< /f:verbatim -->
<!-- /h:outputLink -->
```

The second way is to prefix the value attribute of the `outputLink` tag with the name of context root.

```
<h:outputLink value="/jsf-cardemo-portlet/javadocs">
    <f:verbatim>Documentation for this demo</f:verbatim>
</h:outputLink>
```

Note: This problem is very likely to be solved in the near future. Please see the Jetspeed-2 documentation about the current status.

This section provided us with a step-by-step guide that allows us to convert any JSF servlet application into a JSF portlet application. The example used has been the CarDemo application of the SUN Reference Implementation that we converted into a portlet that we can deploy and run on any portal in its present form. The next section will show how to successfully run the CarDemo portlet application inside of Pluto and Jetspeed-2.

5.3 Running JSF Portlets

In this section we will take the CarDemo JSF portlet application that we just created and let it run on Pluto and Jetspeed-2. We will quickly walk through the installation steps required for a portlet on each platform and talk about the peculiarities of each platform.

The best approach to successfully walk through this exercise is to start by making sure that all the required software and applications are running without errors. Table 5.4 describes the setup you need.

Table 5.4: System Requirements for Running JSF Portlets

Application	Installation Instructions
Tomcat 5.0.28 (or later)	http://jakarta.apache.org/tomcat/
Pluto (latest version or version 1.0 if available)	Chapter 7
Jetspeed-2 (latest version)	Chapter 8
Maven	Chapter 7 and http://maven.apache.org/

Note: Throughout this section, we'll refer you to chapters 7 and 8 whenever we feel that you might need to know more about Pluto and Jetspeed to run the CarDemo application. Although you might find this organization cumbersome, we find JSF an exciting topic and wanted to expose you to its benefits early in this book.

5.3.1 Using Pluto

Like examples we worked with in Chapter 1, 3, and 4 we will use Pluto, which provides not only the reference implementation of the portlet container, but a simple portal as well, to quickly test out our portlets. To complete this section, you will need a running Pluto. Please jump to Chapter 7.3 for an easy-to-follow description of the Pluto install. If you're new to Pluto, you'll also learn more about its feature set in Chapter 7.

Note: The CarDemo application works as described only when running on Tomcat 5.

After you have set up a working Pluto, please proceed with the following steps to install and run the CarDemo portlet application.

Modify Web Deployment Descriptor

The current implementation of Pluto has a couple of shortcomings with regard to the portlet deployment process. Therefore, to be able to deploy our CarDemo portlet application at all into Pluto the security constraint section of the web.xml must be modified. The following listing shows the changes required.

```
<security-constraint>
...
  <!-- commented out because the pluto deployment cannot handle it -->
  <!-- display-name>Restrict access to JSP pages< /display-name -->
...
</security-constraint>
```

Note: The display name needs to be commented out in the web.xml file, otherwise the portlet deployment in the next step will fail. This problem is very likely to be solved in the near future. Please see the Pluto documentation about the current status.

Deploy Portlet Application

To deploy portlets into Pluto, use the following command line.

```
maven deploy -Ddeploy=<PORTLET_WAR>
```

The current directory has to be the home directory of Pluto. In order to deploy portlets into Pluto we need to use the Maven tool. Please see chapter 7 for more information on how to setup Maven.

The portlet web application option must include the complete path, such as shown in Figure 5.3.

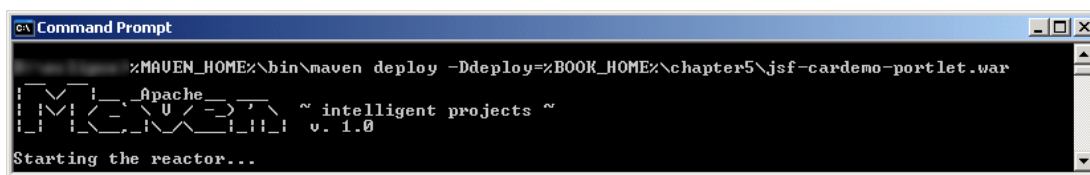


Figure 5.3: Deploying the CarDemo portlet application with Maven.

When Maven returns with the message successful, we can proceed with the next step. Please note that you have to build Pluto first in order to pass this step. Chapter 7 describes how to achieve this.

Pluto Constraints

In this section, we basically need to re-add the <listener> tag that we added to the web.xml file back in section 5.2.2. Here's why: the Pluto deployment process still has a few shortcomings. The shortcoming we

described in section 5.3.1 (“Modify Web Deployment Descriptor”) results in stripping out the listener tag (we showed you this listener element in the listing in section 5.2.2) during the deployment process.

So, to workaroud this shortcoming, we, once again, add the listener tag to the web.xml, but this time, we add the tag to the web.xml of the **already deployed** portlet application. The web.xml file will most likely reside in the tomcat webapp directory, such as (%TOMCAT_HOME%/webapps/jsf-cardemo-portlet/WEB-INF/web.xml).

After changing the file we have a working portlet application and can add this portlet to a page.

Note: This problem is very likely to be solved in the near future. Please see the Pluto documentation about the current status.

Add Portlet to Page

The last step in order for us to view the CarDemo as portlet is an easy one – we put the portlet on a page using the Pluto framework. This is comprised out of two separate steps:

- we create a portlet entity that references our deployed CarDemo portlet application,
- we create a portlet window that references our newly existing portlet entity.

The result will be that we will see the CarDemo portlet application on a page within Pluto. We need to modify two files: the portletentityregistry.xml and pageregistry.xml.

Note: Please see the Pluto Homepage for more information about the structure of these files.

First, let’s create the Portlet entity that references our portlet application. Therefore we use the portlet tag as shown in the following listing with the id 1 to create the portlet entity. It references the portlet definition by using the definition-id tag. Since the entity resides inside of the application tag, the parent child relation is already given. The following listing shows the snippet that we added to the portletentityregistry.xml of Pluto. The file can be found in the WEB-INF/data directory of the Pluto web application. Please note that the application id has to be unique within the whole file. Otherwise you are free to choose any numeric id you want.

```
<application id="1">
  <definition-id>jsf-cardemo-portlet</definition-id>
  <portlet id="1">
    <definition-id>jsf-cardemo-portlet.CarDemoPortlet</definition-id>
  </portlet>
</application>
```

Next, let’s create a portlet window that references our newly existing portlet entity. The next listing shows the snippet that we added to the pageregistry.xml of Pluto. This snippet creates a new page with exactly one portlet on it, the CarDemo Portlet.

```
<fragment name="CarDemo" type="page">
  <navigation>
    <title>CarDemo</title>
  </navigation>
  <fragment name="row" type="row">
    <fragment name="col1" type="column">
      <fragment name="p1" type="portlet">
        <property name="portlet" value="1.1"/>
      </fragment>
    </fragment>
  </fragment>
```



```
</fragment>
</fragment>
```

After changing this last file we finally can start Pluto and see how our portlet looks like inside of a portal. Enter the following address to access Pluto: <http://localhost:8080/pluto/portal>. The resulting page should look like Figure 5.4.

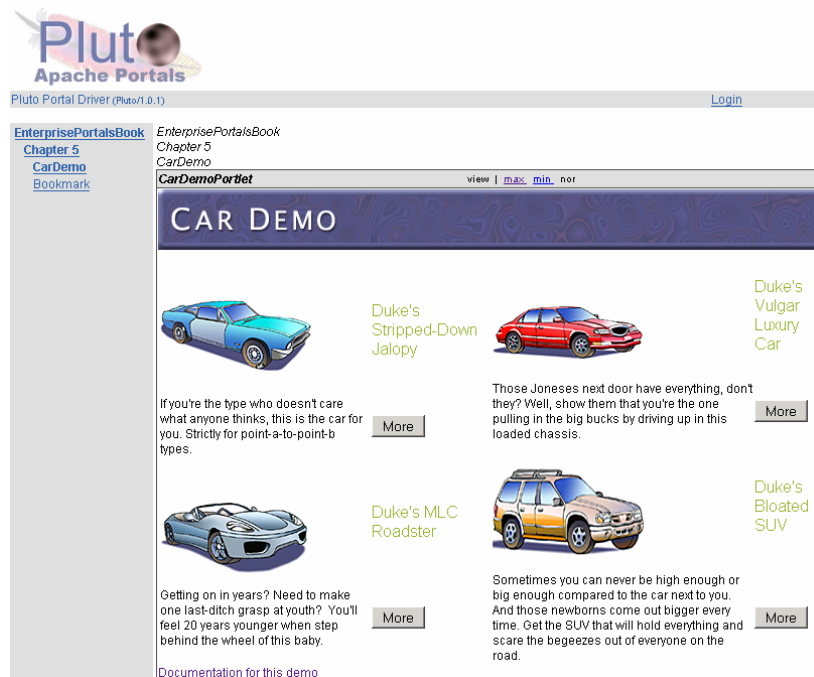


Figure 5.4: JSF CarDemo portlet application running inside of Pluto.

5.3.2 Using Jetspeed-2

As with Pluto in the last section, we need a working Jetspeed-2 in order to complete this section. Please follow the instructions in Chapter 8 to get Jetspeed-2 running.

Note: this example will only work as described when running on Tomcat 5.

After you have set up a working Jetspeed-2, please proceed with the following steps to install and run the CarDemo portlet application.

Deploy Portlet Application

In contrast to Pluto, the deployment in Jetspeed-2 is not triggered with a Maven command, but rather by dropping the file into a predefined directory. This behavior is adapted from Tomcat, where one can simply drop WAR files in the webapps directory to install a web application. In Jetspeed-2 the directory is called `deploy` and resides in the Jetspeed-2 web applications `WEB-INF` directory, i.e. `<TOMCAT-HOME>/webapps/jetspeed/WEB-INF/deploy`.

After copying the portlet WAR file into this directory you only need to start Jetspeed-2; during startup Jetspeed-2 automatically deploys the portlet.

Add Portlet to Page

The next step to look at our newly deployed portlet is to put it on a page. Jetspeed-2 uses psml files that define the page layout and content. Each tab inside of Jetspeed-2 corresponds to one psml file. These files are located in `<J2-HOME>/WEB-INF/pages` and defines the different portlet fragments that are rendered on the

page. A portlet fragment is the output of a portlet itself. It is called fragment since it is not a document that can stand for itself, such as an HTML document. In order to see our portlet we create a new psml file with the text editor and point one portlet fragment to our CarDemo application. The following listing shows a demo psml that we used to test the portlet.

```
<?xml version="1.0" encoding="UTF-8"?>
<page id="/chapter-5.psml" hidden="false">
  <title>Manning Enterprise Portals Book Examples, Chapter 5</title>
  <defaults layout-decorator="jetspeed" portlet-decorator="jetspeed" />
  <fragment id="manning-5.0" type="layout"
    name="jetspeed::VelocityOneColumn">
    <fragment id="manning-5.1" type="portlet"
      name="jsf-cardemo-portal::CarDemoPortlet"/>
  </fragment>
</page>
```

Note: You'll learn more about PSML in Chapter 12.

After changing this last file we can restart Jetspeed-2 and see how our portlet looks like inside of a portal. Enter the following address to access Jetspeed-2: <http://localhost:8080/jetspeed/portal>. The resulting page should look like Figure 5.5.

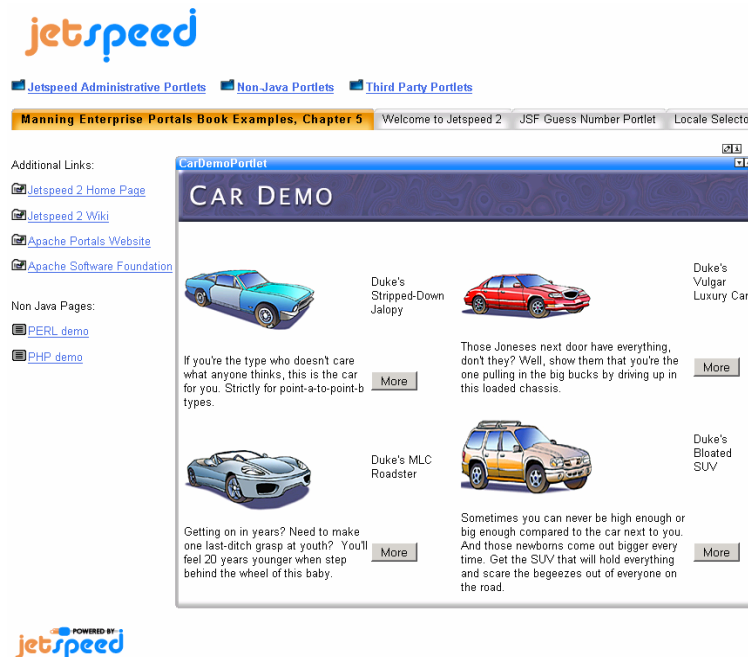


Figure 5.5: JSF CarDemo portlet application running inside of Jetspeed-2.

5.4 Portlet Concepts applied to JSF

This is a very important if not the most important section of this chapter. In this section we will walk through the portlet concepts and learn how they can be used within a JSF application. This is particular interesting since this enables you to use the JSF and portlet capabilities by combining the two technologies. Of course, this brings the best of both worlds together and allows to selectively pick the best capabilities as required. So for instance, you can reuse nearly all the JSF components that are written for JSF within a portlet and extend them to leverage the Portlet API objects such as PortletPreferences.

In this section we will use the Bookmark Portlet that we used previously in this book to show how portlet capabilities can be used together with JSF. We will not cover all of JSF, but only the parts that are interesting for portlets. In section 5.5 we will show the whole example and see what we accomplished. Figure 5.6 shows how the Bookmark Portlet looks like with JSF.

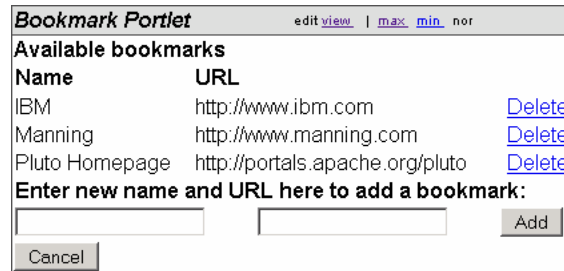


Figure 5.6: JSF Bookmark Portlet in portlet EDIT mode.

5.4.1 Portlet Window handling

The Java Portlet Specification introduces the term Portlet Window. It is defined as a means to include more than one portlet, maybe even the same, multiple times on a page. *The window is the view into the portlet.* More information on portal pages and portlet windows can be found in Section 2.2.

Technically, the portlet windows allow us to scope all available data of a portlet and ultimately assign it to one specific window. This allows the user to put the same portlet more than once on her pages without the portlet window A overwriting the state of portlet window B. The data scoped by the portlet window are:

- PortletSession
- Navigational state

You can find a more detailed discussion about the different portlet states and data in Section 2.4.1. In addition, a portlet can access a request scoping via the `PortletResponse.getNamespace()` method. This is supposed to be used from a portlet to guarantee that returned markup will be unique on a page. For example, you could access a namespace to prefix a JavaScript method, but if you have the same portlet twice on a page, JavaScript would generate a duplicate method name.

For all cases, you can use the `ExternalContext` from JSF to get to the respective objects such as `PortletSession` or `PortletResponse`, which then in turn automatically scope your data. The following listing shows how this is achieved when you have a `FacesContext` at hand. If you do not have a faces context you can use the static `getCurrentInstance()` method to get a handle of it.

```
ExternalContext extCtx = facesContext.getExternalContext();
RenderRequest request = (RenderRequest)extCtx.getRequest();
RenderResponse response = (RenderResponse)extCtx.getResponse();

PortletSession session = Request.getPortletSession();
String prefix = response.getNamespace();
```

Please note, that there is a slight misalignment between the Java Portlet Specification and JSF in this regard. In a lot of cases JSF tags write JavaScript methods into the output stream but will not leverage the `getNamespace()` method in the portlet scenario. As a result a JSF portlet cannot be displayed more than once on a page.

To overcome this problem, it might be possible to namespace the id attributes of all JSF tags that write JavaScript into the response stream. This is possible because JSF uses the id of the actual UI element to namespace the JavaScript method.

5.4.2 Action and Render

As discussed earlier, the Java Portlet Specification defines two methods, action and render, whereas JSF defines six phases. We also discussed how action and render is being mapped into these phases to enable us to use JSF inside of a portlet at all. Namely all JSF phases, besides Render Response phase, which is mapped to the render phase, are mapped to the action phase.

However, this puts us into another position where we do not have direct access to the `ActionRequest/Response` nor `RenderRequest/Response`. So, where do we put all the code that previously has been in the `processAction()` or `render()` method? For instance if you look at the action method of the bookmark portlet from chapter 3, you'll see that we use this method to add and remove bookmarks from the `PortletPreferences`. We need some way to do this in a JSF based portlet application.

The answer is easy for the `render()` method, since programming guides and patterns suggest delegating the rendering to a view such as a JSP. In the portlet case we also have access to the `RenderRequest/Response` inside of a JSP either programmatically or via tags. In a non-JSP environment the Java Portlet Specification also provides a way to access the `Request/Response` by defining fixed keys for the Portlet Objects that are available in the `ServletRequest`.

The action logic is a bit more complicated and requires us to move the logic into a different class. *We recommend moving the logic into a JSF action listener that logically replaces the `processAction()` method in this scenario.*

Looking at the original `BookmarkExample`, the `processAction()` method fulfilled two purposes. First, it handled adding new bookmarks, and second, it handled deleting existing ones. In the converted `BookmarkExample`, we changed this behavior and moved these two functions into two different action listeners that can be called from different buttons or links. The following listing shows how we referenced the action listener for the add button.

```
...
<h:commandButton
    value="{myText.add}" action="add" actionListener="#{bookmark.add}"/>
...
```

The corresponding action listener implementation is shown next.

```
public void add(ActionEvent event) throws AbortProcessingException {
    Object _request =
        FacesContext.getCurrentInstance().getExternalContext().getRequest();
    ...
    ActionRequest request = (ActionRequest)_request;
    PortletPreferences prefs = request.getPreferences();

    ...
    prefs.setValue(name, URL);
    prefs.store();
    ...
}
```

← Name and URL are known from a `BookmarkBean`. See section 5.5 for the whole example.

5.4.3 Validation

In general, we can say that both technologies, JSF and the Java Portlet Specification, define a way to validate settings that have been entered by the user. However, conceptually they are designed in different ways for

different purposes. Of course, both have the same purpose to validate settings, the difference is more subtle and to understand it we need to take a look at the main focus of each specification again.

JSF mainly concentrates on UI aspects and therefore provides a way to generate a rich user experience. The validation concept provided by JSF allows us to validate each single UI element individually and return an error message for each single element that did not pass validation.

The Java Portlet Specification, on the other hand, is very similar to the Java Servlet Specification and concentrates on a Model/Controller concept and doesn't define anything related to UI. The portlet validation is based on a single validator that is checking all preferences at once whenever the preferences are stored. If the validation did not succeed, we can return one error message for all invalid references. This may include preferences that have correlations, like zip code and city and may only be validated with additional information from some backend system.

As a result, we are unable to combine these two technologies for validation, but rather have to decide on a case by case basis which validator to use. In our example we did not touch the validator introduced in Chapter 3.

5.4.4 PortletPreferences

PortletPreferences are solely a portlet concept that has no overlap with either Servlets or JSF. In this and the following sections we will explain how we can work with portlet-only concepts.

An important building block to work with PortletPreferences has already been laid in the previous sections by explaining how to get hold of the PortletRequest/Response pairs. This enables you to access any portlet object such as PortletPreferences. Looking at the listing in section 5.4.2 again, you can also see how we retrieve the preferences and store our bookmark.

Still missing is the part about rendering PortletPreferences inside of a view or JSP. JSF provides an html tag called dataTable that allows displaying a table with any data as its content. Unfortunately for portlet developers, the out-of-the-box dataTable does not support Maps as one of its acceptable data formats. JSF itself provides a couple of ways to solve this problem. Two solutions leveraging the JSF architecture could look like this:

7. Create a new renderer to generate the table.

In this case we cannot leverage the already existing table renderer since it is not written in an extensible way. This may depend on the JSF engine that you are using.

8. Create a new UIData Model and leverage the existing table.

In this case we have to define a little bit more, such as a new tag and configuration files, but besides that the existing table renderer could display a map.

Since explaining all the JSF features is outside the purview of this book, we found a simpler third option of the JSF architecture to enhance. In our example we convert the preferences map into a List for every request by using a JSF-managed bean. Since the portlet preferences are actually managed from the portlet container, it would be wrong to store the bean into the session. Therefore we fill the bean for each request with the current portlet preferences. This is achieved by using a phase listener that creates and initializes the bean at the first convenient opportunity.

Listing 5.3 shows the bean responsible for converting the map into a list.

Listing 5.3: Converting the PortletPreference Map into a List.

```
public class PreferencesBean
{
    private ArrayList convertedPreferences;
    private String defaultValue = "";
    ...
}
```

```

public void setPreferences(PortletPreferences preferences) {
    convertedPreferences = new ArrayList();
    Enumeration enum = preferences.getNames();
    while (enum.hasMoreElements()) {
        String name = (String)enum.nextElement();
        String value = preferences.getValue(name,defaultValue);
        BookmarkBean entry = new BookmarkBean();
        entry.setName(name);
        entry.setURL(value);
        convertedPreferences.add(entry);
    }
}
}

```

In Listing 5.4 the PreferencesRestore listener is used to create and fill the bean.

Listing 5.4: Displaying the PortletPreferences in a table.

```

public class PreferencesRestore implements PhaseListener
{
    public void beforePhase(PhaseEvent event)
    {
        Object _request =
            event.getFacesContext().getExternalContext().getRequest();
        if ((_request instanceof PortletRequest)) {
            PortletRequest request = (PortletRequest) _request;
            PreferencesBean bean = 1
                (PreferencesBean) request.getAttribute("preferences");
            if (bean == null) {
                bean = new PreferencesBean();
                bean.setDefaultValue("undefined");
                bean.setPreferences(request.getPreferences());
                request.setAttribute("preferences", bean);
            }
        }
    }
    public void afterPhase(PhaseEvent event) { }
    public PhaseId getPhaseId() { return PhaseId.ANY_PHASE; }
}

```

Check if bean
already exists

Create bean and fill it with
the portlet preferences

Set bean into the request so
that it gets picked up by JSF

Now that we have learned how PortletPreferences can be used in a JSF application we proceed in the following section with learning how portlet URLs are generated within JSF.

5.4.5 URL Generation

Another portlet-only concept that we will discuss in this section is the generation of URLs pointing to portlets. More specifically, we will look at how to leverage the additional properties that the Java Portlet Specification provides inside of URLs, such as PortletModes, WindowStates and RenderParameters.

Normally, when we create a URL inside of a portlet, it looks like shown below.

```

PortletURL addUrl = response.createActionURL();
addUrl.setPortletMode(PortletMode.VIEW);
addUrl.setWindowState(WindowState.NORMAL);
addUrl.setParameter("add", "add");

```

As you can see we have the ability to create a URL that directly points to a specific portlet mode or window state and that may contain additional parameters. With the current JSF Specification 1.1 we lose all these features, including the ability to create RenderURLs. This means that JSF can be used inside portlets, but does not leverage the render links that allow searchability and performance optimization.

To work around the shortcomings of not being able to specify portlet mode, window state and render parameters, we developed a special action listener that is able to handle all these portlet-specific properties. The properties are mapped to specific attributes that are retrieved from the action listener and then set on the `ActionResponse`. As a result, we used another way to set `PortletModes`, `WindowStates` and `RenderParameters` that is provided to us by the Java Portlet Specification. The following listing shows how this action listener can be used with a JSP.

```
<h:commandButton value="#{myText.cancel}" action="cancel" immediate="true">
  <f:actionListener
    type="com.manning.portlets.chapter05.PortletActionHandler"/>
  <f:attribute name="PortletMode" value="view" />
  <f:attribute name="WindowState" value="normal" />
  <%-- f:attribute name="RenderParameters"
    value="[name,value],[name2,value2]" / --%>
</h:commandButton>
```

Please note, we actually did not implement the feature for render parameters, but you can enhance this example easily to also support them by yourself.

5.4.6 State Handling

Every application usually requires some kind of state in order to work correctly. The state that we will discuss in this section is responsible for defining the view of an application, sometimes also called navigational state. In general we want any state to be available on the next request so that it has to be carried over from request to request in some fashion to not get lost in between.

A normal web application has multiple ways to carry this state over, the typical and most obvious being cookies, `HttpSession` and hidden parameters within a FORM. Every method of course has its characteristics - the `HttpSession`, for instance, is easy to use but has two major disadvantages: First, it is stored on the server side and therefore claims expensive memory on the server, which you should always keep as low as possible for performance reasons. Second, the session is only a bucket that has no reference to the actual page that is currently displayed. Especially for view state it is desirable to bind it to the browser page, otherwise you would not get any previous state when you press the back button inside of the browser and display the previous browser page or view into the application.

For all the reasons that we displayed above, JSF chooses to use hidden parameters inserted in FORMs to transport its view state. Thus, as soon as the user presses the back button in the browser, the FORM of the previous browser page would be displayed and any following request issued by any link would contain the previous view state.

Applying this technique to the portal scenario requires a couple of considerations. In the beginning of the book we explained the concept of a portal. In a nutshell, it means that we have an aggregation of portlets on one page, and therefore a couple of players that contribute to the resulting page: The portal application itself as well as all portlets that are displayed on the current page. As you can see, we have now multiple applications contributing to the page result as contrast to the web application scenario where only one application returned the page result.

In the end, this means that the view state of the result page comprises the view state of every single application on the page. It also means that one task of a portal consists of aggregating each single view state into one overall state including the portal's state. Looking at the FORMs technique it seems pretty difficult to

solve the problem of nested forms where we send all data of all forms on one browser page. Actually, there is a solution to this problem using JavaScript and DOM manipulations that we will not describe here as this is beyond the scope of this book.

One elegant way to solve this problem is to store the view state inside of a URL. This guarantees that the view state is bound to the browser page. By pressing the back-button the previous view state would be resend and the browser page – the page could even be bookmarked. Of course you also have to deal with the problem that the view state may become very large and therefore would not fit inside of a URL anymore. Portal vendors may consider this fact and take steps to reduce the size with different compression techniques.

Unfortunately, the JSF Specification 1.1 does not provide enough flexibility to change the behavior of using URLs instead of FORMs. As a result, a JSF application will lose its model state that is stored inside of a hidden FORM parameter when clicking on any link outside of the portlet itself – unless you implement a complex solution that involves JavaScript and DOM manipulations. Yes, albeit an unsatisfying conclusion, the JSF Spec doesn't currently support this concept. The good news is that this is a shortcoming of the current version and might be addressed in version 1.2.

So basically with JSF V 1.1 you only have limited choices to support navigational state and the browser back button. For now you need to count on portal vendor extension and hope that the next JSF version will address this shortcoming.

5.5 The whole example

Now let's take a look at the complete example, the Bookmark Portlet converted to JSF. We will see all the single pieces coming together to create one meaningful picture that we can easily use as a template to generate more complex applications. Once you know the basics and worked through one complete example you will see how easy it is to extend it with your own functions and capabilities. Figure 5.7 depicts the portlet in its VIEW mode, and Figure 5.6 in its EDIT mode.



Figure 5.7: JSF Bookmark Portlet in portlet mode VIEW.

The portlet application comprises out of a couple of elements and is packaged as a WAR file. Table 5.5 lists the content of the war file with a description for each element.

Table 5.5: WAR file content of the BookmarkPortletJSF

Directory	Filename	Description
/jsp	view.jsp	JSP rendering the VIEW mode
/jsp	edit.jsp	JSP rendering the EDIT mode
/WEB-INF	faces-config.xml	configuration file of JSF
/WEB-INF	portlet.xml	Portlet deployment descriptor as required by the Java Portlet Specification
/WEB-INF	web.xml	Web deployment descriptor as required by the Java Servlet Specification
/classes	portlet_*.properties	properties files in different languages
/classes/...	BookmarkBean.java	JavaBean representing one Bookmark and also containing action listeners for adding and deleting bookmarks
/classes/...	PortletActionHandler.java	ActionListener to switch between portlet modes or window states

/classes/...	PreferencesBean.java	Bean representing the PortletPreferences, required for display
/classes/...	PreferencesValidatorImpl.java	Portlet Validator
/lib	*.jar	Required Libraries

For the remainder of the section we will show the most interesting, but not all, elements of the BookmarkPortlet in its completeness so that you gain deeper understanding of the whole example and its solutions. Listing 5.5 shows the view.jsp displaying all the bookmarks that have been entered in the EDIT mode and stored in portlet preferences.

Listing 5.5: view.jsp used to display all bookmarks as hyperlinks

```
<%@ page session="false" %>
<%@ taglib uri='http://java.sun.com/portlet' prefix='portlet' %>
<%@ taglib uri="http://java.sun.com/jstl/core" prefix='c' %>
<%@ taglib uri='http://java.sun.com/jsf/html' prefix='h' %>
<%@ taglib uri='http://java.sun.com/jsf/core' prefix='f' %>
<f:loadBundle basename="portlet" var="myText"/>
<portlet:defineObjects/>
<f:view>
  <B><h:outputText value='#{myText.available_bookmarks}' /></B><br>
  <c:set var="prefsMap" value="${renderRequest.preferences.map}" />
  <c:choose>
    <c:when test="${empty prefsMap}">
      <h:outputText value='#{myText.no_bookmarks}' />
    </c:when>
    <c:otherwise>
      <c:forEach var="pref" items="${prefsMap}">
        <A HREF='<c:out value="${pref.value[0]}" />'>
          <c:out value="${pref.key}" /></A><BR>
        </c:forEach>
      </c:otherwise>
    </c:choose>
  </f:view>
```

Load JSF taglibraries

Load JSTL taglibrary, since we use JSTL in this example to display the preferences

Retrieves the preference map from the render

Writes a message when no preference is available

Iterates of the preferences and write hyperlinks to each element

This listing showed how to easily display a list using the PortletPreferences as data element. We use the JSTL technique to iterate and access the elements since it is more intuitive with JSTL instead of JSF. In the upcoming listing we will use JSF to see how different this is.

Listing 5.6 shows the edit.jsp displaying all available bookmarks as well as allowing us to delete and add them.

Listing 5.6: edit.jsp used for deleting available bookmarks or adding new ones.

```
<%@ page session="true" %>
<%@ page import="java.util.*"%>
<%@ taglib uri='http://java.sun.com/portlet' prefix='portlet' %>
<%@ taglib uri="http://java.sun.com/jstl/core" prefix='c' %>
<%@ taglib uri='http://java.sun.com/jsf/html' prefix='h' %>
<%@ taglib uri='http://java.sun.com/jsf/core' prefix='f' %>
<f:loadBundle basename="portlet" var="myText"/>
<portlet:defineObjects/>
<f:view>
  <h:form>
    <B><h:outputText value='#{myText.available_bookmarks}' /></B><br>
    <h:dataTable value="#{preferences.convertedPreferences}"
```

Use converted preference ArrayList to display dataTable

```

        var="bookmark2">
<h:column>
    <f:facet name="header">
        <h:outputText value='{myText.name}' />
    </f:facet>
    <h:outputText value='{bookmark2.name}' />
</h:column>
<h:column>
    <f:facet name="header">
        <h:outputText value='{myText.url}' />
    </f:facet>
    <h:outputText value='{bookmark2.URL}' />
</h:column>
<h:column>
    <h:commandLink action="delete" actionListener="#{bookmark2.delete}">
        <h:outputText value="{myText.delete}" />
    </h:commandLink>
</h:column>
</h:dataTable>
</h:form>
<h:form>
    <h:panelGrid columns="3">
        <f:facet name="header">
            <h:outputText value='{myText.addTitle}' />
        </f:facet>
        <h:inputText id="name" value="#{bookmark.name}" required="true" />
        <h:inputText id="url" value="#{bookmark.URL}" required="true" />
        <h:commandButton value="{myText.add}" action="add"
            actionListener="#{bookmark.add}" />
        <h:message styleClass="validationMessage" for="name" />
        <h:message styleClass="validationMessage" for="url" />
        <h:outputText value='' />
    </h:panelGrid>
</h:form>

<h:form>
    <h:commandButton value="#{myText.cancel}" action="cancel"
        immediate="true">
        <f:actionListener
            type="com.manning.portlets.chapter05.PortletActionHandler" />
        <f:attribute name="PortletMode" value="view" />
    </h:commandButton>
</h:form>
</f:view>

```

← Display preference name

← Display preference value, which is a URL in this case

← Create link that can be used to delete this row (preference) of the dataTable

← The next form including panelGrid is used for adding new bookmarks

← Create input fields for name and URL of the new bookmark

← CommandButton calling the actionListener with the name add at the bookmark bean

← CommandButton for returning into the VIEW mode of the portlet

In this listing we used JSF to create a list of bookmarks. Behind each bookmark a link is created to delete it. At the bottom of the list we created two forms to add a new bookmark and to return from edit mode into view mode again.

The last interesting piece of the Bookmark Portlet that we will look at is the BookmarkBean. The bean functionality itself is not the interesting one; it is rather the two action listeners that come along with the bean. Listing 5.7 shows the BookmarkBean.

Listing 5.7: BookmarkBean representing one bookmark and

public class BookmarkBean implements java.io.Serializable

```
{
    private String name;
    private String URL;
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getURL() { return URL; }
    public void setURL(String url) { URL = url; }

    public void add(ActionEvent event) throws AbortProcessingException {
        Object _request =
            FacesContext.getCurrentInstance().getExternalContext().getRequest();
        if (!(_request instanceof ActionRequest)) {
            throw new AbortProcessingException("Expected ActionRequest/Response, but
received "+_request);
        }

        ActionRequest request = (ActionRequest)_request;
        PortletPreferences prefs = request.getPreferences();

        try {
            prefs.setValue(name, URL);
            prefs.store();
        } catch (ReadOnlyException e) {
            throw new AbortProcessingException("Name/URL is read-only: "+name, e);
        } catch (ValidatorException e) {
            throw new AbortProcessingException("Validation not successful", e);
        } catch (IOException e) {
            throw new AbortProcessingException("Exception happend while storing", e);
        }
    }

    public void delete(ActionEvent event) throws AbortProcessingException {
        Object _request =
            FacesContext.getCurrentInstance().getExternalContext().getRequest();
        if (!(_request instanceof ActionRequest)) {
            throw new AbortProcessingException("Expected ActionRequest/Response, but
received "+_request);
        }
        ActionRequest request = (ActionRequest)_request;
        PortletPreferences prefs = request.getPreferences();

        try {
            prefs.reset(name);
            prefs.store();
        } catch (ReadOnlyException e) {
            throw new AbortProcessingException("Name/URL is read-only: "+name, e);
        } catch (ValidatorException e) {
            throw new AbortProcessingException("Validation not successful", e);
        } catch (IOException e) {
            throw new AbortProcessingException("Exception happend while storing", e);
        }
    }
}
```

Normal JavaBean
representing one
bookmark

Retrieve
ActionRequest from
ExternalContext

Retrieve
PortletPreferences

Set new bookmark with name and URL

Store preferences

```

        throw new AbortProcessingException("Exception happend while storing", e);
    }
}
}

```

The add method is linked with the “Add” button of the edit.jsp so that it is called when the user presses this button. Within the method we first get a pointer to the PortletPreferences object of the Portlet API and then use the name and url variable to store them in a preference.

The delete method is linked with the “Delete” button of the edit.jsp so that it is called when the user presses this button. As with the “Add” button we first get a pointer to the PortletPreferences object of the Portlet API and then use the name variable to delete the preference.

The complete example can be downloaded on the homepage of this book, <http://www.manning.com/portlets>.

5.6 Summary

In this chapter we learned how the JavaServer Faces technology can be integrated into portlets. We started off with a quick introduction to JSF especially tailored towards portlets.

Continuing into the next section, we first took the approach of converting any existing JSF application into a portlet application. This is a very important part of the chapter, since it enables you to convert any JSF application that you already built independently into a portlet application and therefore use it inside of any portal. Then, we installed the resulting JSF portlet on Pluto and Jetspeed-2 in order to show you that this portlet can really run on different portals.

The most important section of this chapter described how the different portlet concepts relate to the JSF technology. We covered the complete palette from perfectly matching concepts to problems and issues with the current JSF technology that cannot really be matched to portlets. We illustrated this by using the Bookmark Portlet example that we already used throughout the book. As a result we have a portlet that is completely converted to use JSF.

Please note that we now not only covered a JSF application into a portlet but also vice versa – you know how to convert an existing portlet to use JSF and leverage its advantages.

At the end, we showed the major parts of the Bookmark Portlet so that you can see all the changes that we did in the previous section in one place and therefore see the whole picture. You can use this chapter as a template and reference for your own JSF portlet project.

In the next chapter, we will cover the portal architecture and we will explain the basic building blocks that define a portal and how these different components interact.

5.7 Resources

- [1] JSF Central, Popular Java Server Faces Community, URL: <http://www.jsfcentral.com>
- [2] My Faces, Free Java Server Faces Implementation, URL: <http://www.myfaces.org>
- [3] My Faces, Moved to Apache, currently in incubator, URL: <http://incubator.apache.org/myfaces/>
- [4] JSF Reference Implementation from SUN, URL: <http://java.sun.com/j2ee/jaserverfaces/index.jsp>
- [5] Introduction to Java Server Faces, URL: <http://java.sun.com/j2ee/jaserverfaces/jsfintro.html>
- [6] Jetspeed 2, Homepage of Jetspeed 2, URL: <http://portals.apache.org/jetspeed-2/>
- [7] Pluto, Reference Implementation of JSR 168, URL: <http://portals.apache.org/pluto/>
- [8] The Java Portlet Specification, URL: <http://jcp.org/en/jsr/detail?id=168>

- [9] The JavaServer Faces Specification, URL: <http://jcp.org/en/jsr/detail?id=127>
- [10] Tomcat, Homepage of Tomcat, URL: <http://jakarta.apache.org/tomcat/>
- [11] The JavaServer Faces Specification 1.2, URL: <http://jcp.org/en/jsr/detail?id=252>

5.8 Further Reading

Kito D. Mann: JavaServer Faces in Action, Manning, October 2004

Chapter 6 Exploring a sample portal architecture

As we already discussed in the previous chapters, one of the main functions of a portal is to integrate information and applications and present them to the end users. If you ever wondered how a portal must be built to achieve this, keep on reading because in this chapter we will explain the basic building blocks that define a portal and how these different components interact. We will further discuss how the portal architecture relates to the J2EE architecture and dive into more details about the major portal components and the programming model of these components. With that background knowledge you will be able to better understand the full potential as well as the limitations of portals using portlets and get the most out of it.

Although this chapter shows several screenshots of a Jetspeed portal this chapter goes beyond the limitations of existing open source projects (as far as they are known to us) and shows an architecture that can fulfill the most common requirements of portals today.

If you are further interested in portal architecture and want to see one brought to life in the Jetspeed open source project, chapter 8 is waiting for you further below.

6.1 Understanding the generic portal architecture

Let's start with the question: What makes up a portal from the highest level architectural point of view? Figure 6.1 displays that there are basically three tiers in a portal architecture and how they interact.

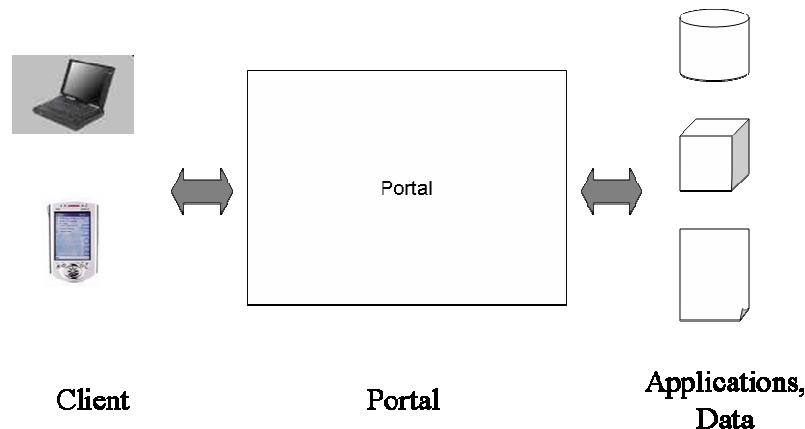


Figure 6.1: A very high level representation of the tiers of a portal architecture

The first tier makes up the means to access the portal – a client device. This device could be, for example, a desktop, a phone, or a PDA. Then the center tier is the portal itself which we are going to discuss in more detail during this chapter. And last but not least, there is the content the portal aggregates. This tier could be applications or content of any kind.

But how could the portal tier be built up in more detail? To find out let's take a look at the basic requirements of a portal. And we find that the two main requirements of a portal are:

1. It needs to know where to get the information and how to call the applications.
2. It needs to aggregate the collected data and display that information in a compact manner.

To fulfill these requirements, our architecture would consist of two components as depicted in Figure 6.2. The first is an aggregation component and the other is an information and application handler component – let's just call it portal core for now.

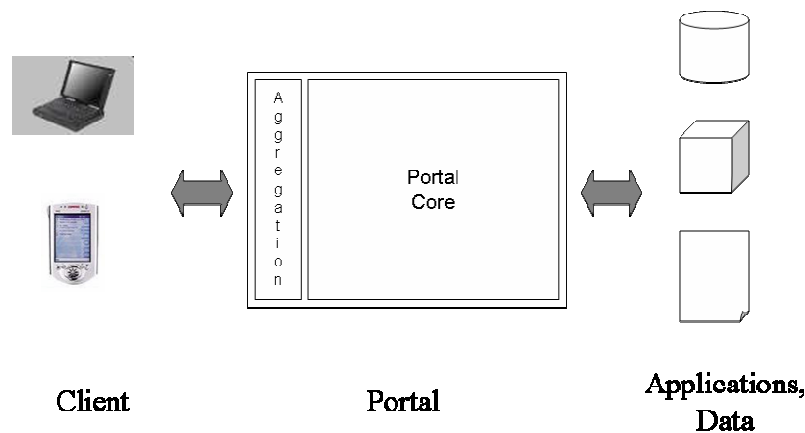


Figure 6.2: Portal architecture overview with the two main components, an Aggregation and a Portal Core component for the portal itself

As you can see in Figure 6.2, the client interacts with the Aggregation component, or to be more precise, the client receives the aggregated markup of all integrated applications from the Aggregation. The portal core on the other hand takes care of applications and data and delivers the results of this interaction to the Aggregation. If we think again about what the portal core should provide then we want it to enable you to add and remove information and applications easily – at best even during runtime. But how can our portal core possibly know where to get that changing information from, how to retrieve it, and how to call all the various applications?

To do all this, we add a well defined layer via which it can call any application or retrieve any kind of data via the same interface(s) or the same API. So our component does not handle the data itself but manages and calls pieces of code hidden behind the new layer and becomes a container for those pieces. Those pieces of code now finally are the ones who know where and how to get that data either as they are coded that way or can be configured accordingly. Those small pieces making up the whole portal are – as you guessed it – the portlets. The portlet core is therefore the portlet container as shown in Figure 6.3:

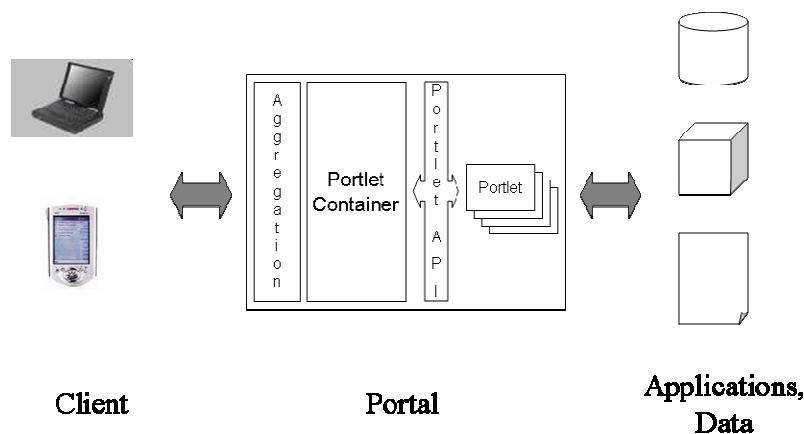


Figure 6.3: Portal architecture overview part III showing how portlets and portlet container interact via the portlet API

This is the most high-level architectural overview of a portal that is common to most, if not all, portal implementations. How those basic portal components could be constructed in more detail and what further components are contained in a portal architecture will be discussed in the following section.

6.2 Defining the portal components

While the previous section introduced portal architecture with a generic and therefore hardly applicable architecture, it's time to build a more complex and more specific sample architecture. Pieces of the architecture and some of the components, we will come up with during this section are taken from the Apache Jetspeed projects 1 and/or 2 (covered in chapter 8) or are at least strongly related while others are part of commercial products or not part of any portal yet.

6.2.1 Portal Requirements

Before we start building the architecture we should think about what the most common requirements are and therefore what our creation should be able to provide in the end. The one most basic requirement is probably to display portlets. Then of course we want to add, or deploy portlets and we should also be able to remove portlets later again.

To be able to display those newly deployed portlets and also for optimizations of the content it should be possible to customize the portal's layout. This includes the creation and deletion of pages, and the modification of a page's content, such as adding and removing portlets. Then we want portlets to be displayed not only in English, but in any preferred language, so we need national language support (NLS) too.

When introducing several pages with portlets we also need to be able to navigate through the portal. This should not only be possible for HTML browsers but for multiple markups. To get the best user experience for a markup we need support for multiple devices in order to tailor the markup to the client device.

Another feature that is very important for a portal is the ability to customize the look and feel by setting skins and themes. Themes refer to the code and data required for the portal's decoration and navigation, including the images and stylesheets that are used. Skins refer to the content part of the page. The skin defines how the frames around a portlet look, covering the color or the form (angular, rounded or transparent) of the frame.

To be able to customize the portal to our specific personal needs it first has to know us - so authentication is required. To support authentication we need some means to manage users. When the portal already knows me, due to authentication, we could use this for personalization; so that the news portlet shows the sports

news to me first and puts the politics to the end. With the portal knowing the user we could protect the portal and add authorization. And we want to achieve a real single sign-on (SSO) experience. Single sign-on means that you sign into the portal once (single) and then you can access all data or applications without having to provide credentials again. Therefore the portal has to be able to store all credentials of a user to be used to authenticate at other systems. Most of the requirements we collected so far produce lots of data that needs to be persisted in some datastore. This includes data for the users, deployed portlets, layout, etc.

Our portal should definitely support remote portlets. Remote portlets are not locally installed but a proxy calls the portlet that resides on a remote server via a remote portlet protocol. This means that we don't have to get the portlet code written in the correct language against an API our portal supports and deploy it as alien code in our portal. We can simply add new instances of the proxy portlets to our portal. The proxy portlet then calls the actual portlet residing on a remote site written in any language against any API on any platform. This multiplies the services we can add to our portal without much effort. But to enable an administrator or an end-user to cope with such a lot of functionality we need some administration facility.

Besides those visible functions we need base, technical functionality like logging and tracing to be able to maintain the portal, access to configuration for all portal components to find out how they are supposed to behave, and an event dispatching mechanism that enables a component to sign up for a notification if a specific event in the portal takes place (e.g. to do some clean-up if the user logged out).

This means our list of requirements that we want to use in order to create our sample portal architecture below looks like::

- | | | |
|------------------------------|----------------------|---------------------------|
| 1. Display portlets | 8. Skins and Themes | 15. Remote Portlets |
| 2. Deploy / remove portlets | 9. Authentication | 16. Administration GUI |
| 3. Customize portal layout | 10. User Management | 17. Logging and Tracing |
| 4. Navigate | 11. Personalization | 18. Configuration Service |
| 5. National Language Support | 12. Authorization | 19. Event Dispatcher |
| 6. Multiple markup support | 13. Credential Store | |
| 7. Multiple device support | 14. Datastore | |

Now that we have the basic portal requirements, let's try to come up with an architecture that can fulfill every single one of them in the next section.

6.2.2 Grouping requirements into components

But how should we group the requirements into components? First we need to decide which requirements need a new and separate component and which requirements can be grouped and handled together by a single component. But this cannot be done following some cookbook. The decision is mostly based on your experience and your gut feeling. This doesn't sound very scientific but this is part of what makes software engineering an art.

A possible option to group the requirements and create corresponding components would be like those listed below:

Authentication

Authentication should definitely be handled by a separate component to be able to support various authentication products and authentication mechanisms. Thus we can easily exchange the implementation that handles the authentication when needed.

User Management

The same is true for user management. Probably every company already has some database or directory that contains employee or customer data. To be able to reuse the existing data we need a component that is the plug point for that functionality in the portal.

Datastore

Once again the same reason for a new component: We want to support various products and mechanisms. The datastore component must be able to store our data, for example, as XML files or in some relational database.

Engine

This component is responsible for displaying portlets, navigation, personalization, (customize) portal layout, NLS, multiple markups, and skins and themes. The handling of the corresponding information and data is the most basic portal functionality and more or less the brain or the engine of the portal. Although the data comes from different separate components or modules this is where it is brought together and is computed to produce the requested portal page.

Aggregation

The actual markup and therefore the view is then put together and rendered by our aggregation component based on the data computed by the engine. This is a separate component as there are new aggregation implementations required to support further markups or devices based on the same input data delivered by the engine. So we can easily extend our portal to support new client devices or browser by adding a further new aggregation implementation.

Deploy / remove portlets

The module that is responsible for the whole lifecycle of our portlets is of course the portlet container. This covers deployment, instantiation, invocation, destruction and removal of portlets. All of these functions depend on the types of portlet we want to support. Therefore it makes sense to cover them by one component to have all dependencies on the type put into one pluggable and/or exchangeable unit. This enables us to support several portlet types or standards (e.g. Jetspeed-1 and JSR 168 portlets) by supporting several portlet container implementations.

Remote Portlets

Depending on what we want to support here this can be either one or two components. To be able to integrate remote portlets into the portal to be displayed there we need a remote portlet client component or “consumer” component. This is more or less a very complex special kind of portlet calling remote portlets.

To provide local portlets to other portlets for remote invocation, a remote portlet server component or “producer” component is required. This is another kind of access point to the portal that delivers markup and therefore a special kind of an aggregation.

Administration

As we are building a portal that should enable everybody to do anything via the browser our choice for the implementation of an administrative GUI is, of course, to build admin portlets.

For an offline or automated administration, a script or XML based administration makes sense as well.

Credential Store

The credential store is a separate component to be able to support and plug in existing vault implementations perhaps already containing the users' credentials. Moreover this is the kind of functionality that should be available to all other components and therefore become a portal service.

Logging and Tracing

Again we want to support various implementations perhaps based on existing logging and tracing mechanisms. And more than anything else this should be a service to be used by all other components.

Configuration

As some of the behavior of your components should depend on the configuration of the overall portal and to allow for one single place of configuration the components need a consolidated way to find out about the configuration of the portal. How could we handle this better than by yet another portal service. that is available to all other components.

Event Dispatching

In such a pluggable and dynamic architecture a means is required that allows the exchange of information without having to know what other components APIs (or even what other components) exist. Via an event dispatcher, components can subscribe to be informed about specific events they are interested in and act accordingly.

Portal Service Infrastructure

To be able to easily access such base functionality like the credential store, logging and tracing, configuration and event dispatcher we should pull in some portal service infrastructure and implement these portal components as portal services that are available to all other portal components.

6.2.3 Putting it all together

Now we went through the process of finding and listing the requirements for our sample portal and grouped these somehow to get separate, exchangeable components. The reason is that using a lot of smaller but strongly specialized components makes our portal architecture as modular as possible and still easy to understand. This is still true although we came up with a new component almost for each requirement as you can once again see in figure 6.4 which shows from a high level point of view on how they come together.

As you can see our architecture has already become quite complex by now. How everything plays together now and how the components work and how they interact is covered now in the following section.

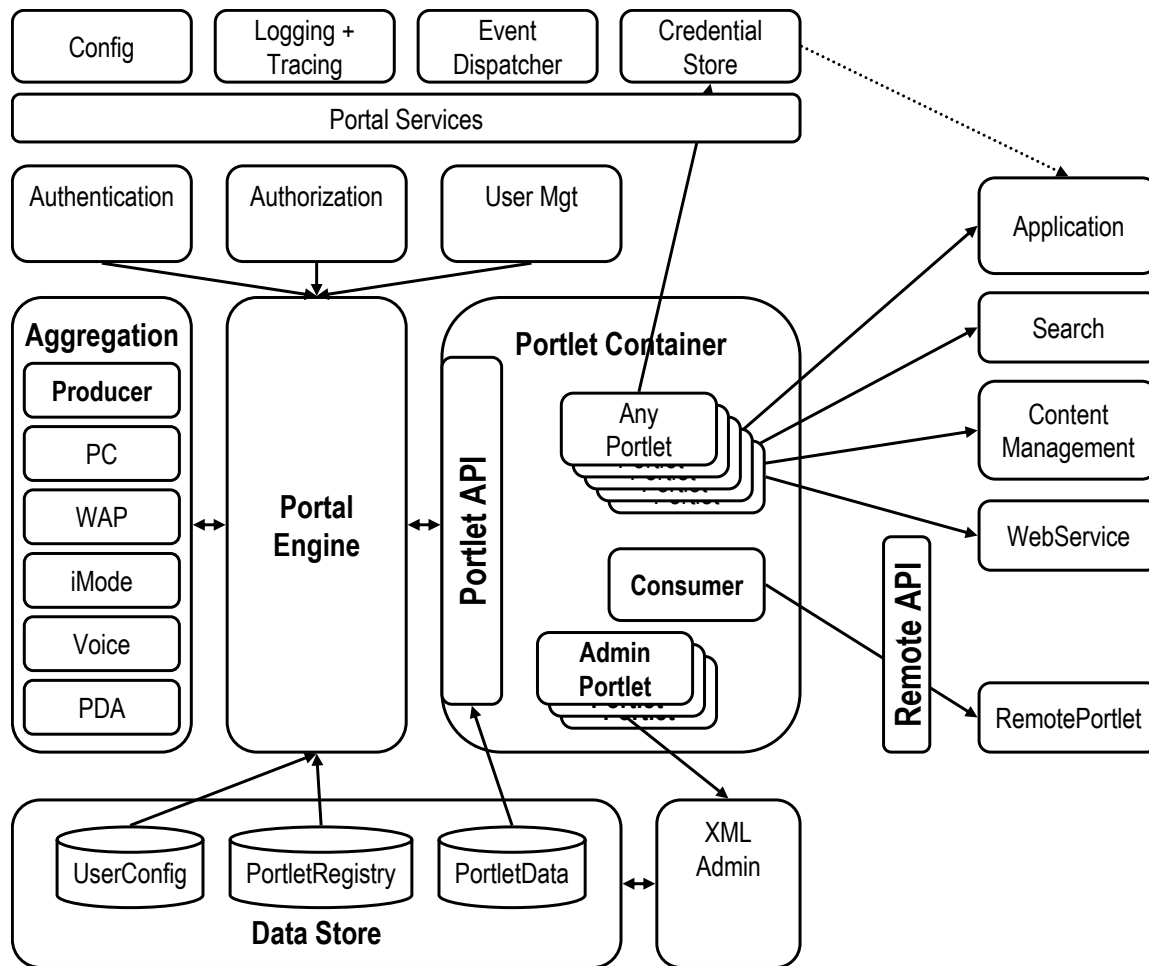


Figure 6.4: Our Portal sample architecture brings together all the necessary components.

6.3 Understanding the component interaction

Looking at our architecture it raises the question what do those components do and especially how do they interact? To get an idea for at least most of the components let's try to investigate the following scenarios and the corresponding flows:

- A user accesses the portal for the first time without authentication
- A user logs in
- A user clicks an action link in the portlet

While following the flow for these scenarios every time we come to a new part of a component that comes into action we will explain how it could possibly be designed and implemented in our J2EE based sample portal.

6.3.1 First scenario: a user accesses the portal without authentication

Let's first take a look at what happens if a user accesses the portal for the first time without providing any authentication information. For example, he can do this by entering the portal's URL in a browser, causing a new request to be generated and sent to the portal. But what component of the portal is the request sent to

and how can it receive requests? The corresponding flow is depicted in Figure 6.5 but the explanation is build up step by step in the following sub-sections

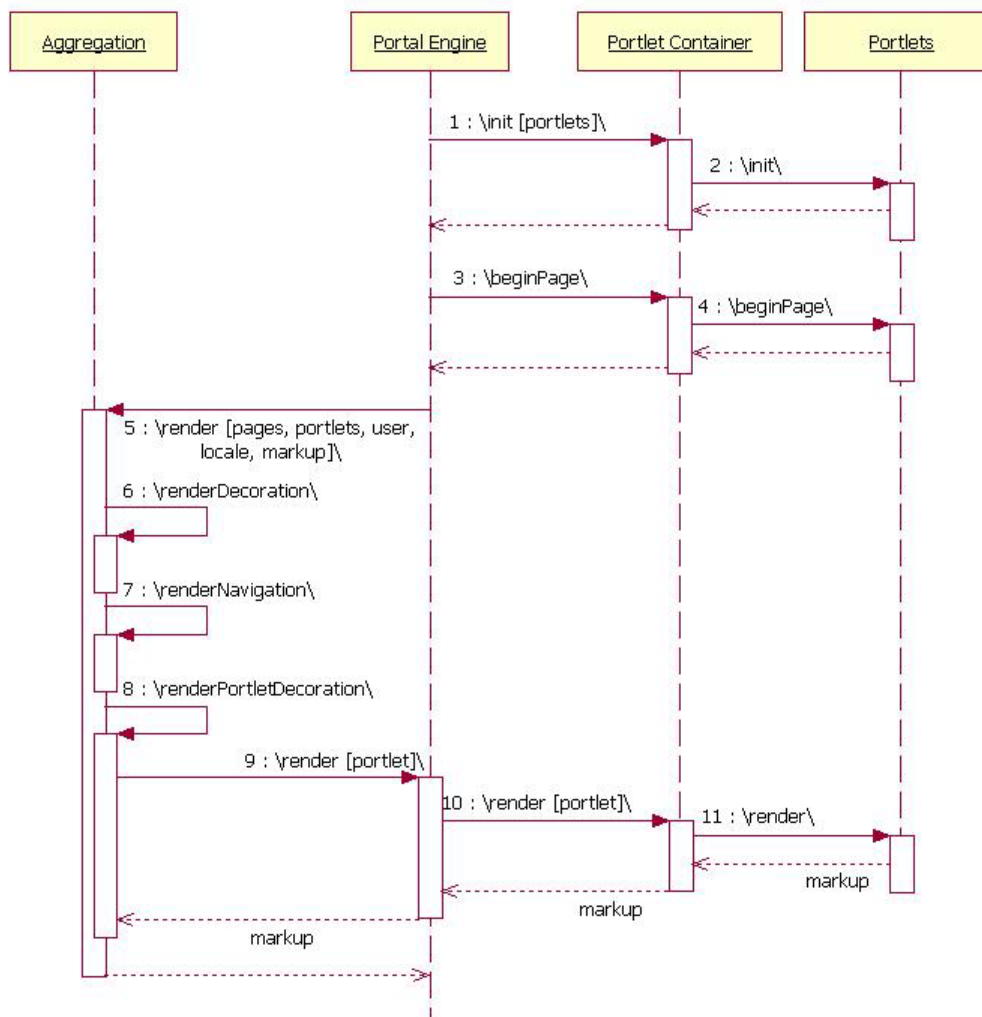


Figure 6.5: Flow for the first scenario: Render the page

Portal Engine: part I

The component that is responsible for the main logic in the portal is the portal engine. It is not only the heart of the portal but also provides the access point to the portal. As we are talking about a J2EE sample portal the access point we choose is, of course, a servlet. So the engine's servlet can take the request and start to dispatch it to other components to fulfill a portal's tasks.

First this request is given to the authentication component which inspects it to check if there are credentials included in this request to be able find out who has sent the request.

Authentication: part I

An implementation of the authentication component could be based on existing authentication proxies or reuse the functionality of the underlying application server. Let's assume we want to reuse the authentication mechanisms of the J2EE application server. Then we would simply have to configure the application server to accept the authentication mechanisms we want to support. The application server will then provide us with a

Java Authentication and Authorization Service (JAAS) subject for each request in the current thread (more details about J2EE security are discussed in the appendix C.). All the authentication component has to do then is to map the JAAS subject to create some kind of an authentication user object.

In our scenario the authentication could of course not find any credentials and therefore creates a null or “unauthenticated” user object or something similar. The next step then is to give this user object to the authorization component to do a first check if the user is allowed to access the portal before we spend any further computing power.

Authorization: part I

The authorization component determines whether a given user is allowed to perform a given action on a given resource. In our case this would be: Is our unauthenticated user allowed to view the portal?

As this specific sample is based on a J2EE server, the best implementation to handle that first step of authorization is to reuse the authorization mechanisms of the underlying application server. By protecting the portal’s entry point (e.g., a servlet) the authorization of the application server will refuse the request and send a response indicating that access to the resource is not permitted if that is the portal’s setup. This means that our authorization component does not have to do anything because if our user’s been able to call portal code he must be authorized to do so. Normally you would specify two entry points to the portal: One that is protected and one that is not protected, if you want to show information to unauthenticated users as well. As our user does not send any credentials he is only allowed to access the freely accessible access point to the portal. As we got this far in the flow he must have entered the correct URL and our authorization component will let him pass.

Further requests to the authorization component cannot be handled that easily and require some kind of table(s) via which it can look up the requested authorization information. The authenticated user ID (which is in our case a “not authenticated” user ID) can now be used to get more information about the user via the user management component.

User Management

The user management component provides access to an existing database or directory containing user data like a corporate LDAP (lightweight directory access protocol – but LDAP usually refers to the directory product that is accessible via that protocol). The user management basically finds information about a user in the directory and creates a user object containing at least the attributes required in the portal environment. This user object can then be used all over the portal. You can find more information about the user object in further down this chapter in the programming model section.

As our “unauthenticated” user ID cannot be found via the user management something like another “unknown” personalization user object is returned.

Before we go on let’s see how the flow until that point would look like in a diagram. This is shown in the flow diagram shown in figure 6.6:

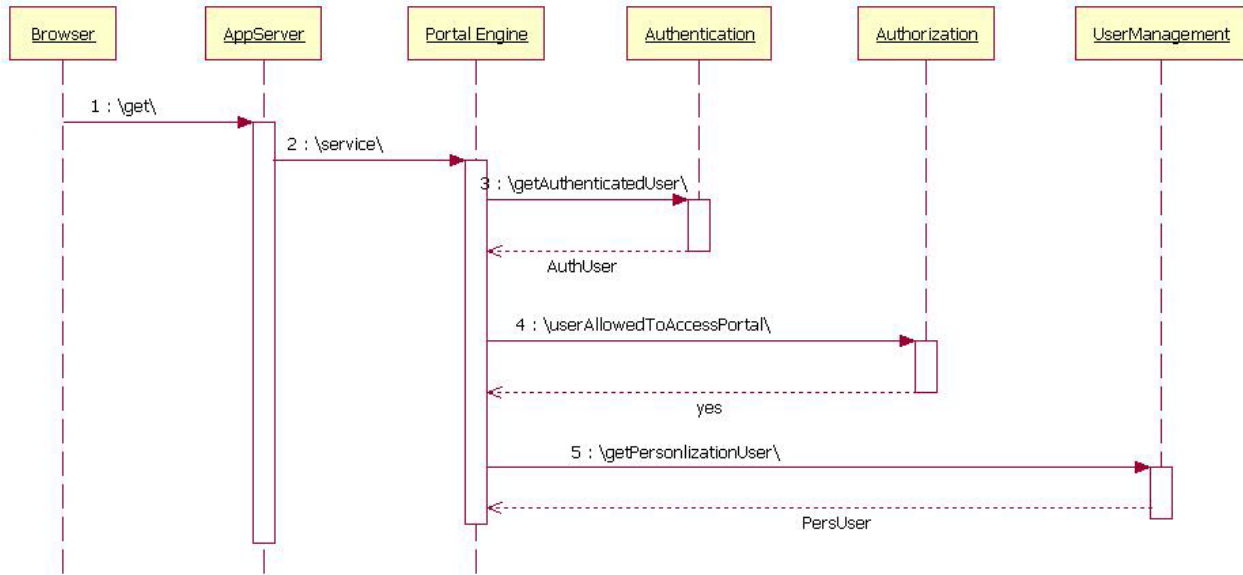


Figure 6.6: Flow 1st scenario – access user information

Now that the portal - or at least the portal engine - knows that the user is actually allowed to access some of the portal's information, it starts to create a page for him.

Portal Engine: part II

Now that the aggregation knows whose page to create, it must know for what environment this should happen. It inspects the request to find out what markup will be accepted by the device that was used to create and send the request. The required markup language can now be determined through the content type in the request header. In addition to the kind of markup, another important input parameter for a correct aggregation is the requested locale, which is also available via the request.

With all these parameters available, the aggregation queries the data store component for the pages that support the required markup.

Data Store

The datastore component provides interfaces through which you can easily query for specific objects. The implementation could, among other things, query relational databases via SQL for the requested information and wrap the returned data into the corresponding portal objects.

After querying the datastore for the pages, we now have a set or tree of pages, but is our unauthenticated user prevented from accessing possibly sensitive information? To do that, we need to access our authorization component again. This time we loop through the hierarchy of pages and ask for each page whether we should keep it in our list of pages for this user. One criterion is if our user is allowed to view them. Only the pages he may see are kept. If we keep it, then we try to find out if it supports the requested locale or at least a configured fallback language (or a set of fallback languages); otherwise it is omitted.

For the remaining set of pages, we pick the one to be displayed and the data store is queried again to find out about the content of the page. To keep it simple let's say we only get back a list of portlets for this page. One could imagine arbitrarily complex sub-structures and references to dynamic remote sub-trees or to remote aggregation components, or other advanced technologies that could build up the page instead.

The result of these queries and the filtering is a set of objects representing all pages that our user is allowed to see and able to understand plus the corresponding portlets for the current page. The corresponding flow is presented in figure 6.7:

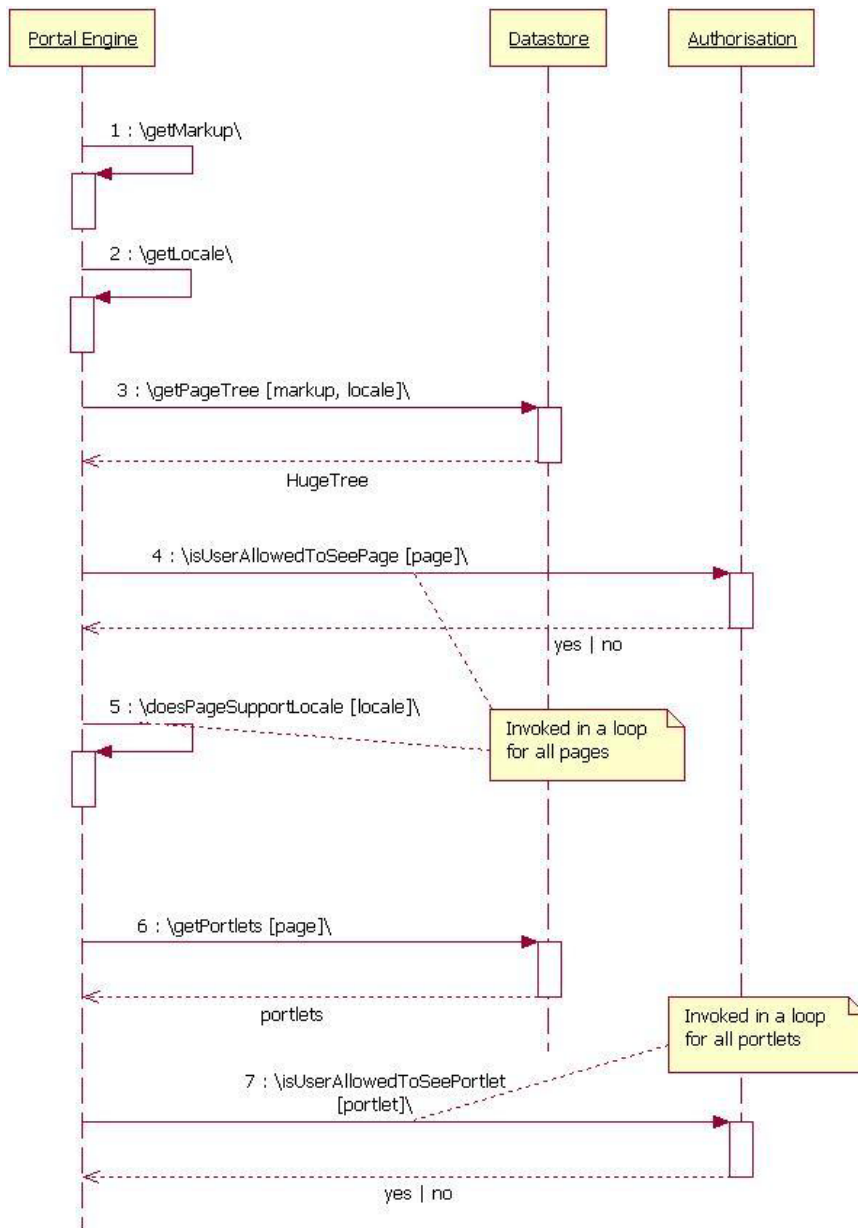


Figure 6.7: Flow 1st scenario – get page content

Before the engine hands off these objects and the information about the device, the markup and the locale to the aggregation to be rendered, the engine first has to take care of something else: The portlets' lifecycle. It could be that some of our portlets in the list of the current page have not been initialized yet. Therefore the engine takes care of this by invoking the corresponding method of the container.

Portlet Container

Depending on the portlet API, our container implements different lifecycle concepts that must be supported. One example for a Java portlet API could be the Jetspeed API, another one the later JSR 168 portlet API. The JSR 168 was first available via the Pluto reference implementation and is quite widespread and supported by various different Java-based portal vendors by now. The Pluto implementation is discussed in detail in chapter 7, Jetspeed in chapter 8, and the JSR 168 portlet API is explained in chapters 2 and 3.

The lifecycle concept could cover

- initialization of the portlets,

- beginning of the portal page to enable the portlets to add information to the beginning of the page like JavaScript,
- portlet actions,
- portlet rendering,
- destruction of portlets,

Basically the container must provide methods for each step of the lifecycle, taking the required information to implement the supported portlet API. During one lifecycle step the affected portlets are invoked compliant to the portlet API and the results (that is, markup) are returned to the caller. You can learn more details about how a portlet container could look in chapter 7, which covers the Pluto open source project.

After the initialization of the portlets, the engine still does not start to render the page. First the portlets are given the opportunity to contribute to the beginning or heading of the page by the engine. The portlets can then provide relevant data for this part of the page, like JavaScript declarations. This happens again by invoking the corresponding lifecycle method at the portlet container. This is, for example, part of the Jetspeed portlet API.

Now we finally take the pages and portlets with their properties to the aggregation component to produce something that makes sense to our user and (not to forget) to his device.

Aggregation: part I

The aggregation is responsible for generating the actual markup for the portal. Our portal – at least from a user point of view – consists of a decoration part, a navigation part and the portal content as shown in Figure 6.8.

The decoration displays some greeting, information about the portal itself like a company name or some label and links to basic portal functionality like login or help, etc.

The navigation could be represented as a tree structure for example or as tabs like Figure 6.8 shows.

The portal content then is the big area where the actual information is placed. In our case the information is only aggregated via portlets.

But back to how Aggregation puts these pieces together. Let's start with the portal decoration: This part of the portal is relatively static, although it depends on parameters like device, markup, user and language. Depending on the device and the markup, a set of corresponding Java Server Pages could be included for the decoration. We choose JSPs to be able to easily exchange the look and feel of the portal by just replacing or modifying the JSPs.

These decoration-specific JSPs then will have to be implemented in a way that they produce the required markup in a fashion that is suitable for the device. The language and user could then be made available to these JSPs via tags or beans. That way a JSP is able to produce the markup in the correct language and greet the user by name – or if no user object (or an unauthenticated user object) is available, the JSP could display a means to log in as shown in figure 6.8.

The next part to be rendered is the navigation: The navigation is created based on the set of objects representing the pages and the portlets. To be as flexible as possible again the rendering is done by including JSPs that display the navigation - again depending on the markup, device and language. In a simple navigation model for an HTTP browser, for each page represented by a corresponding object, a tab could be rendered. Each tab could contain a link referencing the corresponding page as shown in figure 6.7.

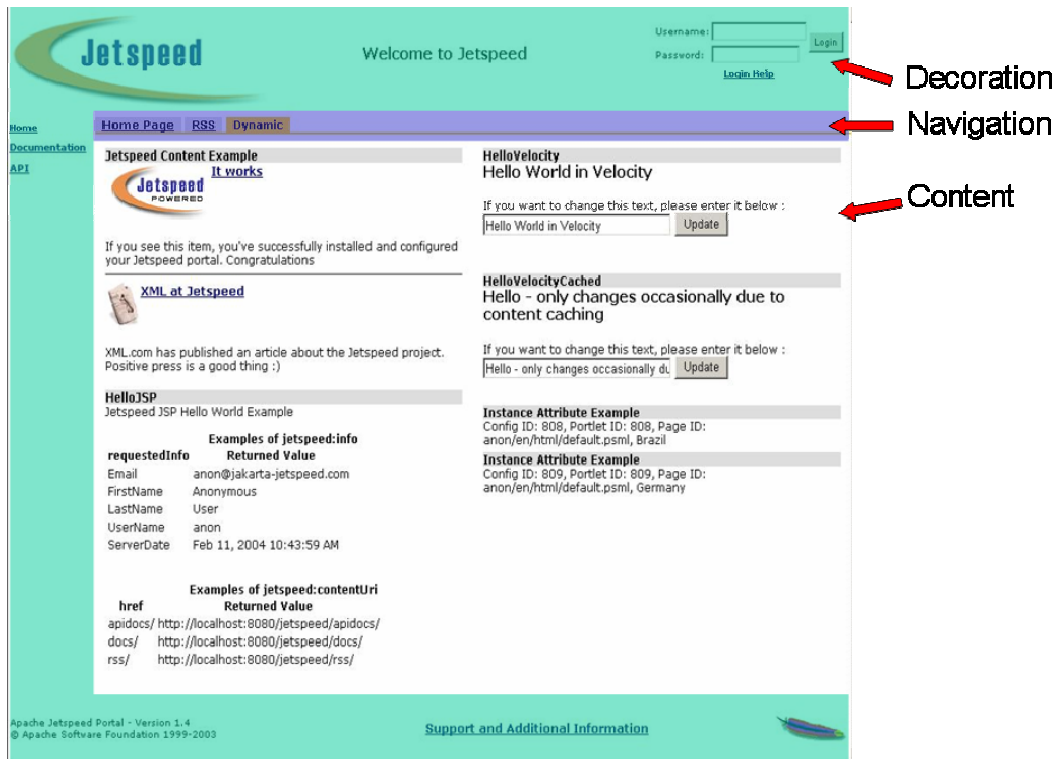


Figure 6.8: A Jetspeed-1 portal page. The aggregation generates the markup for this page, including the decoration, navigation tools, in addition to the portal content.

Finally we come to the content: The content is also based on the set of objects that stand for the pages and the portlets. This time, only the currently displayed page and the portlets belonging to this page are interesting to us. With the list of portlets available, the portlets can now be invoked and their markup displayed.

This means that the aggregation component starts now to call the portlet container. But to remove the interdependencies between our components, the aggregation does not call the portlet container directly but calls it via the engine. That way the engine can add information to the calls to the container that is not accessible to the aggregation. So the aggregation triggers the invocation of the portlets via the engine to finally get the content rendered.

Aggregation: part II

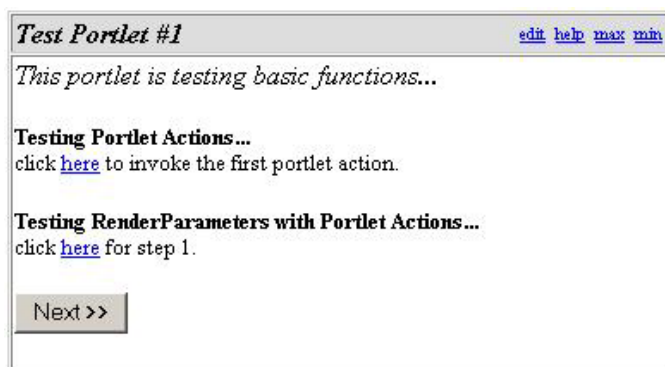


Figure 6.9: A Pluto test portlet confirms that decorations and content are being generated.

Now we can start to invoke the portlets, right? Not quite yet! First the portlet's decoration must be created before its markup can be added to the page. The portlet decoration is the markup containing a portlet's title and the links to change its window state (such as to maximize it) and to change its mode (such as into an edit mode). Sample decoration for a test portlet is shown in figure 6.9.

What kind of states and modes each portlet supports is known to the aggregation via the object that represents the portlet and only the links in the decoration for the supported modes and states for the current markup are rendered. This happens again by including some JSP to easily exchange the look and feel of the decoration.

Now that we have the decoration of the portlet we can finally call the container to invoke the portlet.

When the aggregation finishes rendering all portlets to the content pane of the portal, it again has to take care of the lifecycle of the portlets affected – this time it informs the portlets via the engine and the container that the rendering of the page is finished.

And that's already it! We have stepped through the complete process how a portal renders a page as already shown in the flow in figure 6.5. But of course there are several things we did not touch yet, which we will cover during the following scenarios.

6.3.2 Second scenario: A user logs in

The first scenario covered an unauthenticated user who could see the public portal page or pages. But to be able to access further information that may be personalized to his requirement he decides to log in which results in the flow shown in figure 6.10 that is explained in the following sub-sections.

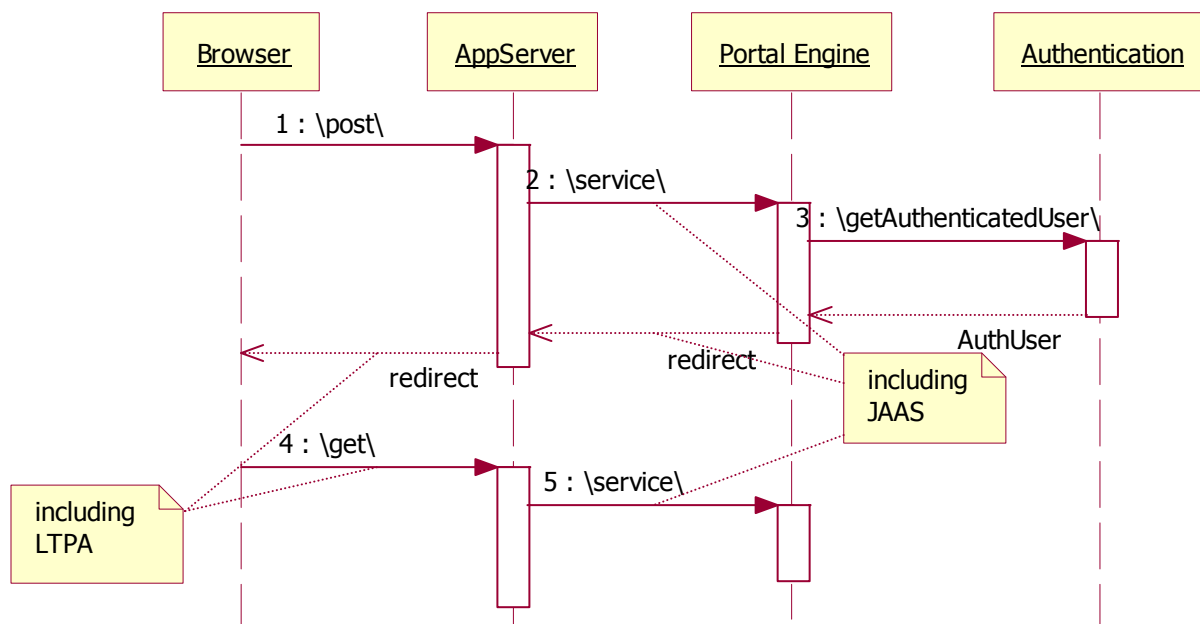


Figure 6.10: Flow for second scenario: A user logs in, and the portal responds.

The whole flow is triggered by the user by filling in the form in the decoration as shown in figure 6.11 and submitting it via the “Login” button.



Figure 6.11: The login form of Jetspeed-1. When the user logs in with this form, the portal validates the login information.

Submitting this form triggers the validation of that form based authentication in the application server. As the correct user id and password have been entered the validation succeeds and the application server creates a JAAS subject for the current thread.

Again the request then goes to the portal engine servlet. The engine again first asks the authentication for a user object.

Authentication: part II

The authentication this time finds the JAAS subject in the thread and takes this information about the user to access to the LDAP directory. With these attributes retrieved from LDAP it can now create a user object.

With such a valid user object returned, the engine knows that the user successfully authenticated at the portal.

Portal Engine: part III

But looking at the request the engine has to find out that it was not directed at a secured resource. Of course the login form is not secured, as anybody must be able to send their authentication information without being authenticated. For this special login case, the engine does not render a portal page but returns a redirect response pointing to the secured part of the portal.

In addition to what the portal adds to the response, the application server could add some information about the authenticated user via a cookie; for example, a Lightweight Third Party Authentication (LTPA) token. This means that the browser automatically sends a request to the redirected URL containing the LTPA token cookie without the user having to trigger it.

This request is inspected by the application server, which finds the authentication cookie and again creates a JAAS subject. Moreover it finds that the request was directed towards a secured resource. But with the authentication information available, the access to this resource (the secured engine servlet) is granted.

From now on the flow is almost the same as for the unauthenticated page: The portal engine receives the request, but this time when it asks for the authentication for the user object it gets back an object for this authenticated user, not an authenticated user object. The engine finds more pages and portlets that are accessible to this user. The portlets are again initialized as far as necessary. But now with the user available, another lifecycle method is invoked that informs the container (and therefore the portlets) that this user did log in.

Now the information about the pages and portlets is given to the aggregation and the portal page is rendered.

6.3.3 Third scenario: A user clicks an action link in the portlet

Now with the authenticated page displayed, the user is going to interact with one of the portlets displayed. For example, he could retrieve his mail by clicking a link in the corresponding mail portlet. This link is intended to modify the state of the portlet and is therefore an action link. The corresponding flow is depicted in figure 6.12 and again explained in the following sub-sections.

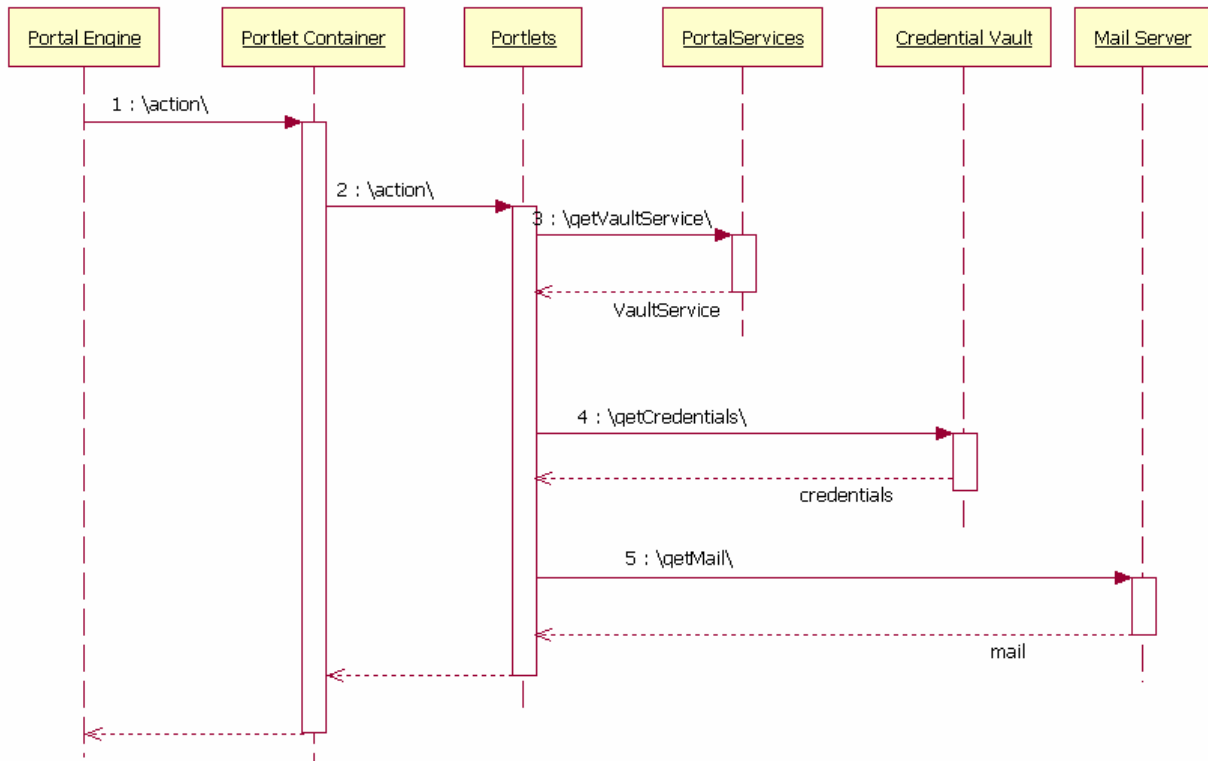


Figure 6.12: Flow third scenario: Perform a portlet action

The corresponding request contains information about what kind of action for what portlet should be executed. The engine again takes the request, gets the user object, retrieves the pages and portlets and invokes lifecycle methods.

Portal Engine: part IV

Before rendering the portal page, it invokes the action at the container for the corresponding portlet as encoded in the request. During this action the portlet must now access the mail server to retrieve new messages using our user's credentials. It retrieves the credentials from the credential store component, which again is accessible via a portal service infrastructure.

Portal Services

The portal service infrastructure provides access to the portal services by, for example, providing a set of static methods. Via these methods you can enter some identifier (like a name or the interface's class) for the type of service you want to have and in return you get an instance of this service similar to what a home provides in the EJB world. In our case we request the credential store service and get back the corresponding object.

Credential Store

The credential store could store a user's credentials in a relational database or in a specific vault implementation. Then the service could either handle the login, knowing the authentication mechanism is returning a corresponding context, or (to keep it simple) just return the credentials for a specific service. To guarantee that nobody else gets a user's credentials the credential store could either access the user's JAAS subject in the thread or explicitly take the user's principal object.

The complete flow for this usage scenario is depicted in Figure 6.12 starting with the Portal Engine that triggers the action that finally leads to a retrieval of mails from the Mail Server.

We already know much about the components we hit and their interaction from the three previous scenarios. We still don't know how they are build from a programming model point of view. But the next section will talk about the portal programming model and hopefully answer your questions.

6.4 Portal programming model

When you hear something like “portal programming model” you probably think immediately of pluggable modules called portlets and the corresponding APIs. But having looked into only some of the portal usage scenarios (still on a quite high level), it becomes clear that the programming model and the customization points of a portal cover not only portlets. Discussing the programming model of a whole portal, which would be all its APIs and how they are supposed to be used would fill a book itself. So we will reduce the scope of this discussion to some basic concepts and how customization points could be implemented.

Some of the later chapters in this book will cover specific parts of the portal programming model:

- **Portlets in the Portlet Container:** The programming model used between portlets and the portlet container is standardized via the JSR 168 and is being discussed in the corresponding chapters that covers the standard (chapters 2 and 3) and the implementation in the Apache Pluto subproject (chapter 7)
- **Portlet Container and the Portal Engine:** A good example for how a programming model being used between the portal engine and the portlet container could look is also the Apache Jakarta Pluto project discussed in chapter 7 .

6.4.1 Programming model principles

For the programming model of a portal server implementation, the same rules apply as for any other server software. But portals are, on the one hand, huge web applications and, on the other hand, integration points for heterogeneous environments. Therefore we will only concentrate on the two most important aspects that are required specifically in the portal area and are as important for applications you may want to build that will have to run on top of a portal.

Ensuring scalability

Our portal must be able to handle up to thousands or even millions of users. So scalability is everything. This is strongly dependent on the application server and the database being used, but our portal must add its part as well.

First of all the portal must be absolutely threadsafe. This can be achieved by keeping the scope of any objects used as local as possible. All APIs and SPIs must be designed accordingly. This mostly prevents the necessity of any kind of semaphores and thus increases the performance and at the same time makes the maintenance of the code easier. Keeping to this rule means that any kind of implicit APIs must not be introduced or used, such as “global” object containers. Unfortunately that cannot always be prevented in J2EE environments. For example, if you want to pass information from one web module to another, you will have to use request attributes or similar mechanisms. But never use untyped parameters anywhere else, as far as it can possibly be prevented. You never know who else is able to access such objects and corrupt your data.

Managing extensibility

As an integration point our portal must be as pluggable and dynamic as possible. This is already true if you think of portlets. But to enable your portal to easily integrate in heterogeneous environments, further customization points are required. To achieve that, you must try to build specialized and therefore easily

exchangeable components in the architecture. But also the design of the components' APIs and of the used object models is crucial.

Basically we must achieve absolute clear contracts between the components and not rely on any implicit data or side effects. This is of course the biggest challenge in software development, as we can only define syntax while the semantics are only implied by the naming of the syntax and additional documentation. But interfaces should be used for API, SPI and object model as far as possible, implicit objects must be prevented and again the scope of the objects should be as local as possible.

6.4.2 Customizing the look and feel of a portal via skins and themes

As said before, the main customization of the portal will be performed by the portlets you deploy in your portal. But another very important aspect of the portal that must provide customization is the look and feel of your portal. The colors, layout, fonts, images, etc. that are displayed must be adapted to reflect the corporate design and therefore the corporate identity of your business.

Themes

As already discussed above we usually distinguish between skins and themes. Themes cover the decoration or banner part of your portal page. There usually the company's name is advertised use the colors and fonts of the company's corporate design.

Jetspeed-1 (covered in chapter 8), for example, does not provide administration for themes, as you most probably will adapt the theme to your needs only once and not change it on a regular basis.

To modify the theme of your Jetspeed-1 portal you can modify the markup-specific JSP files in the /WEB-INF/templates/jsp/navigations directory located in your Jetspeed-1 installation.

For example the html directory contains four JSP files that make up the theme for that markup:

- top_default.jsp
- top_loggedIn.jsp
- left.jsp
- bottom.jsp

By modifying these JSP files you can adapt the parts of your portal's page as shown in figure 6.13. To modify the top part of the portal's pages to be visible by default, i.e., the top_default.jsp must be adapted before logging in.

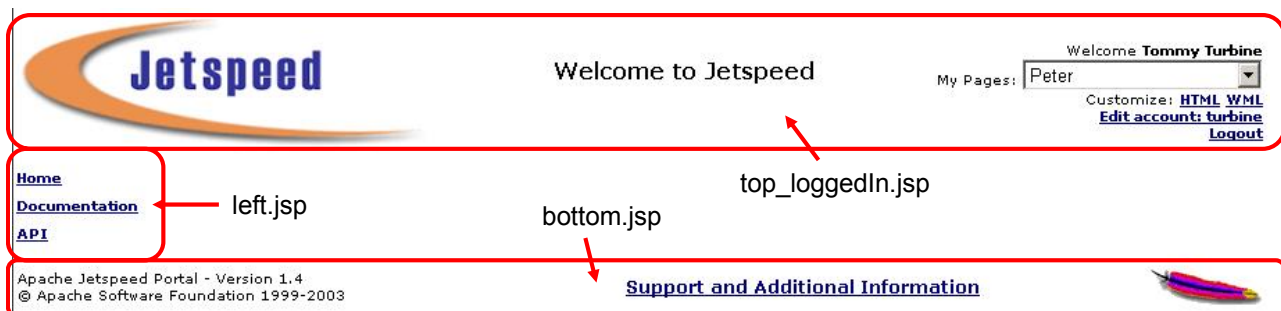


Figure 6.13: A Jetspeed-1 theme consists of three JSP files: Top (logged-in as above, or default if the user has not yet been authenticated), Left and Bottom.

Skins

Skins are dealing with, as already explained above, the content part of your portal where the portlets are displayed. This is also true for the Jetspeed-1 portal. Specific for Jetspeed-1 is that in addition to skins you have the possibility to customize the content pane of your portal via decorations.

Jetspeed-1 allows you to set skins and decorations via the administrative GUI. You can set skins on a per-page level, while you can set different decorations on a per-portlet level. For details about how to configure pages or how to implement your own skins and decorations, please refer to the Jetspeed-1 project at <http://www.apache.org>.

In figure 6.14 you can see a content area of Jetspeed-1 using the default skin containing two portlets with the default decorations.

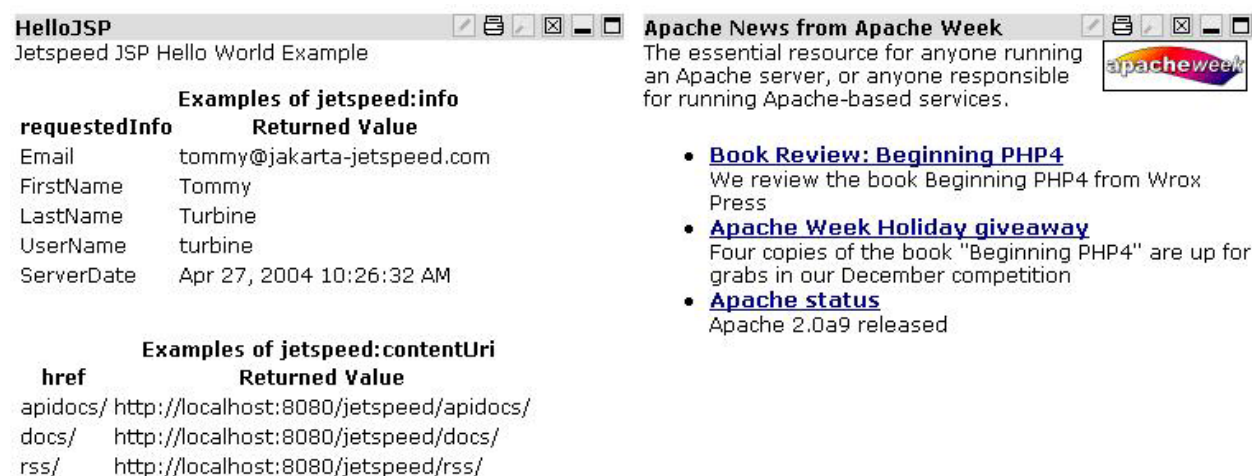


Figure 6.14: Jetspeed-1 portlets with default skin and decoration

In figure 6.15 you can now see the same part of the page but with a new skin called 'MetalSkin'. As you can see the new skin influences the look of all portlets on this page.

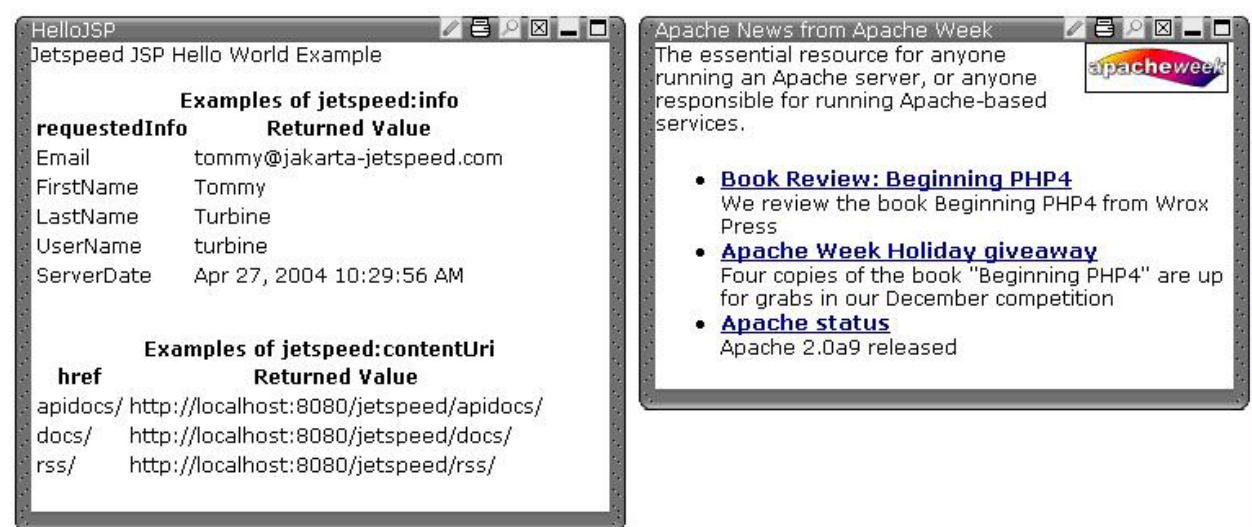


Figure 6.15: Jetspeed-1 portlets with 'MetalSkin' skin and default decoration


In figure 6.16 we kept the same skin but changed the decoration of the portlet on the left hand side to 'boxed' and the decoration of the portlet on the right hand side to 'clear'. And as you can see, these decorations are not using the resources of the skin and are that way overriding the look of the skin. Moreover these decorations do not render any title bar with any controls. That way it is possible to configure your portal's page with individual looks and accessibility for each single portlet.

Jetspeed JSP Hello World Example

Examples of jetspeed:info	
requestedInfo	Returned Value
Email	tommy@jakarta-jetspeed.com
FirstName	Tommy
LastName	Turbine
UserName	turbine
ServerDate	Apr 27, 2004 10:28:35 AM

Examples of jetspeed:contentUri	
href	Returned Value
apidocs/	http://localhost:8080/jetspeed/apidocs/
docs/	http://localhost:8080/jetspeed/docs/
rss/	http://localhost:8080/jetspeed/rss/

The essential resource for anyone running an Apache server, or anyone responsible for running Apache-based services.



- [Book Review: Beginning PHP4](#)
We review the book Beginning PHP4 from Wrox Press
- [Apache Week Holiday giveaway](#)
Four copies of the book "Beginning PHP4" are up for grabs in our December competition
- [Apache status](#)
Apache 2.0a9 released

Figure 6.16: Jetspeed-1 portlets with 'MetalSkin' skin as well as 'boxed' and 'clear' decoration

There are many other ways of customizing your portal besides themes and skins. Let's take a look at them now.

6.4.3 Further customization points in the portal model

The reason why you need a well defined programming model is, on the one hand, to achieve extensibility via well defined contracts between the exchangeable components. But this automatically allows not only for extensions but also for customization by exchanging component and such gaining different, custom behavior. Our sample architecture shows that virtually every component of the portal is a customization point. Perhaps except for the portal engine itself you can plug in component implementations of your choice and that way customize your portal. Of course the implementations must apply to the components contracts like the APIs and SPIs and therefore use the corresponding object model.

The importance of a well-designed programming model becomes even clearer if we take a look at following types of object models of the portal that influence several of the numerous customization points of our sample portal architecture:

User handling

One of the most important objects in the whole portal is the user. If you take a look at the basic functionality portals provide, it is all related to the user. Everything must be customized, personalized, internationalized, restricted or accessible, etc. on a per-user level. The user is interesting to the complete portal rendering flow. Even the authentication and authorization in the application server before entering the portal already deals with the user. Also the portlet, as the last component in the chain, requires user information to personalize its content in the user's favorite language This means that virtually every component must be able to handle user data.

If we think twice there are basically two use cases for user data. One is security-related, and it checks whether this user is allowed to perform specific actions. The other one is related to personalization in order to optimize content for the user's needs.

As these two aspects of the user are orthogonal we could even decide to create two separate user objects for our portal object model. This would make the semantics of many components clearer, as you will always know that the component requires the user for authorization or for personalization. The authenticated user would be produced or created by the authentication component. As this component can now concentrate on this specific task and reduced set of user data, it is very easy to plug in custom implementations using authentication mechanisms perhaps already available in an enterprise's environment.

The same is true for the personalization part: It is much easier to implement a plug point that fills a user object with the user attributes retrieved, for example from your company's LDAP directory without having to care about anything else.

With such a programming model you could even realize a scenario where you have one authenticated user object used for several personalization-related user objects: Imagine a one-to-one trust relationship between two portals 'A' and 'B,' both with lots of end-users. If portal 'A' uses portlets of portal 'B' remotely, it authenticates always as the same, single portal 'A'. But to enable personalization for its end-users, portal 'A' sends the personalization data for the current user for each request. That way portal 'B' always has the same authenticated user (portal 'A'), but receives lots of different personalization users. Therefore, it can enable its portlets to produce personalized content on a per-user granularity as shown in figure 6.17.

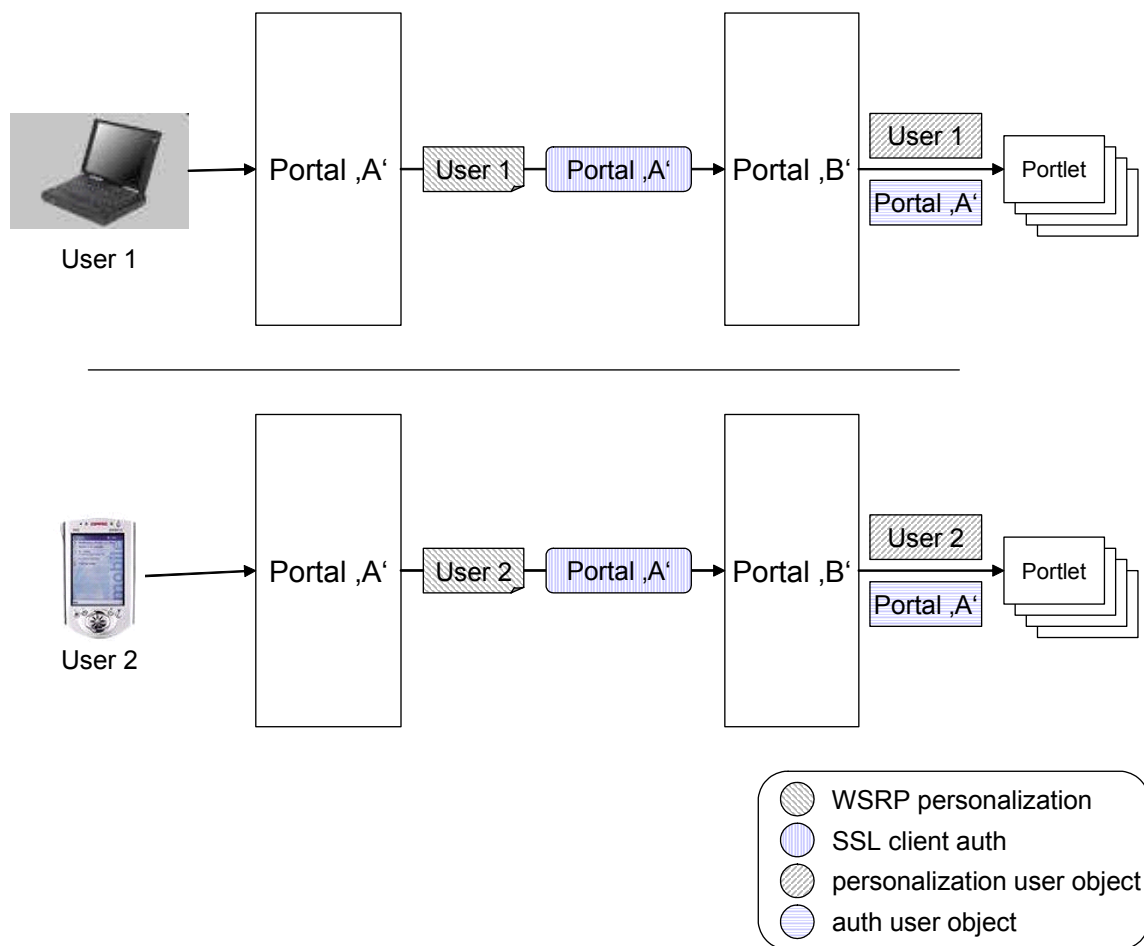


Figure 6.17: Two aspects of a user: The personalization data can be independent from the authentication data

Object identification

Another thing that can be crucial for the success of your portal is the way how “entities” are identified. With entities we mean any kind of “thing” in the portal that needs to be identified or addressed. This could be a portlet, a column on a page, or a credentials entry in the credentials service.

One use case would be to encode the IDs in URLs and that way associate a request with one or more entities like the information about which page should be rendered for this request. To be able to export and import such entities for backup reasons, it is sufficient that their IDs are locally unique. But to enable staging of a portal from a test to production environment, these IDs should be globally unique.

To be able to easily and quickly compare two or more entities for equality, it makes sense to use complex objects instead of Strings for the IDs. With such a object ID interface in place the interfaces between your portal’s module can now be easily defined. And with nice interfaces between the portal and any exchangeable component, the components themselves have become customization points and part of the portal’s programming model.

6.5 Summary

In this chapter we introduced you into the world of portal architecture by building a sample architecture.

We started with finding the most high level architecture on the tier level, then breaking down the portal tier into more high level components.

We started creating the sample architecture by thinking about the requirements for the portal. With a given list of requirements we tried to find architectural components that can fulfill our requirements connecting them in our sample architecture.

To discuss and understand the created architecture, we started looking at use cases for our portal. By following the flows for three use cases, we learned how the components of the architecture interact. Furthermore whenever we hit a new component in our flow, we discussed the tasks of this component and how it could be built up. This way you learned how the components of our architecture work and how they interact.

To go beyond this component level view, we finally started a discussion of the portal’s programming model. Here we focused on parts of the portal’s programming model that are not covered by other chapters, like the portlet API.

Now that you know how a portal is basically constructed and how it works from a high-level point of view, we can start looking into more specific parts of the portal world as covered in the following chapters especially chapter 8 will show many of the architecture concepts brought to live.

Further readings

<http://portals.apache.org/> for portal related Apache projects

- <http://portals.apache.org/jetspeed-1/> for the Jetspeed-1 Apache project
- <http://portals.apache.org/jetspeed-2/> for the Jetspeed 2 Apache project

Chapter 7 Working with the Pluto Open Source Portlet Container

In this chapter, we explain the Apache Open Source project Pluto by familiarizing ourselves with the architecture and design as well as installation and setup. Apache is a software foundation supporting open source software projects. The Apache group characterizes themselves not simply as a group of projects sharing a server, but rather a community of developers and users. This attitude positively affects all projects in a sense that each project, with their developers and users, seems like a small family.

First, we introduce you to the Pluto project by giving a high level overview and explaining the big picture. In the second section, we dive into the architecture of the Pluto project. It will give you a deep understanding of the components comprising Pluto. It is a main building block that we use to build upon in the following sections.

The third section slightly breaks the flow of the chapter to move from the theoretical level to the practical level. It shows you hands-on step-by-step what to do to get the Pluto project set up on your system – from download, installation, through running.

After the first three sections, you will have gained a lot of experience with the Pluto project, as well as a sense for the different parts and components, and you will know how to configure and install it. This knowledge will help you to understand the last section, which demonstrates how to use the portlet container of Pluto in your own portal.

7.1 What is Pluto?

Pluto is the reference implementation - from now on called RI - of the Java Portlet Specification defined in JSR 168 through the JCP process. That process specifies that a JSR must comprise a specification document, a reference implementation and a compliance test suite. Each part of the JCP serves a special purpose. The specification is a document that defines in a very detailed manner the JSR, the reference implementation shows the feasibility of it, and the test suite makes sure that an implementation fulfills all required parts of it.

JSR 168 specifies the contract of the portlet container to the portlet. It does not define any interface or relation with the portal, nor does it define any aggregation pattern for the portal. Analogous to the emphasis of the specification on the portlet container, the RI focuses on the portlet container implementation. However, in order to show portlets on a page and display them in a browser, we need some piece of code that uses the portlet container to render portlets. Therefore Pluto doesn't only contain the implementation of the RI, but also a test implementation of a portal which enables us to view and aggregate portlets in a very basic fashion. Figure 7.1 depicts the big building blocks of the Pluto Project.

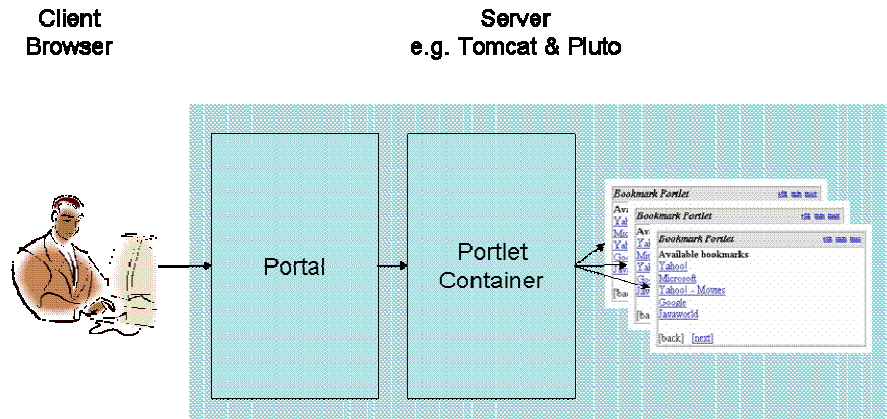


Figure 7.1: The Big Picture

Definition: Pluto is the reference implementation of the portlet container. The Portal piece of Pluto is only a required byproduct.

In the next section we learn why we started the Pluto project and how it relates to its sister projects on Apache.

7.1.1 The Apache portlet project

Even though Pluto started with JSR 168 and its single need to prove the feasibility of the Java Portlet Specification, it now serves a diversity of objectives.

From the beginning of the Java Portlet Specification and its reference implementation, the people working on this topic (Stefan Hepper and Stephan Hesmer amongst them) felt that it would take more than just writing code and making it available for download in order to be accepted by the Java community. They were creating something new, and only if it would be used and supported by others, could they really make a difference. To achieve this aggressive goal, they decided to open source Pluto on Apache in the first step. By doing this, the community is able to participate in the project, shape it to their wishes, and embed Pluto into other projects as well. The second step was to make the portlet container architecture as pluggable as possible so that it can be leveraged from many portals.

Jetspeed is another project on Apache, which in contrast to Pluto, focuses on the portal functionality, not the portlet container. The next generation version of Jetspeed, called Jetspeed-2, uses Pluto as its portlet container. The next chapter focuses on using Jetspeed as an enterprise portal.

Figure 7.2 shows how the Pluto portlet container supports many different portal applications.

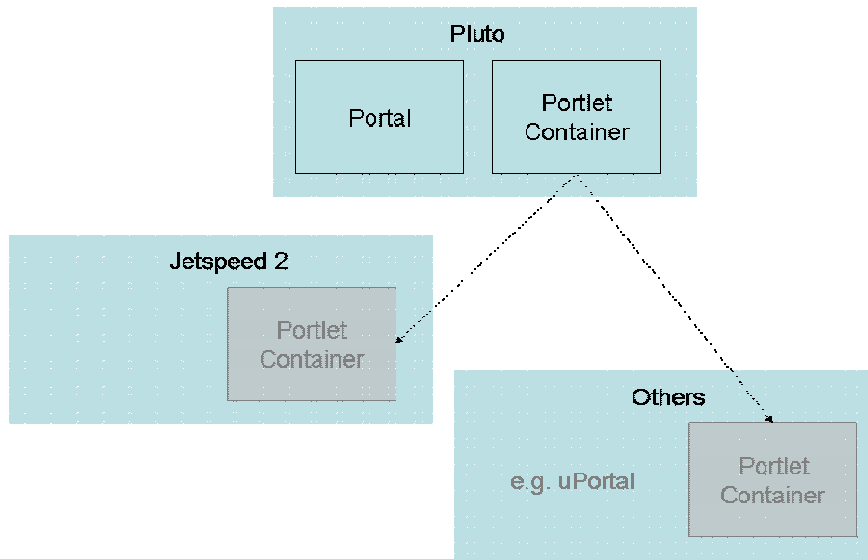


Figure 7.2: Pluto's Portlet Container can be reused in many other projects

One common misperception of the Pluto project is that it also provides portal capabilities. It only provides a test portal implementation to be able to run the portlet container and pass the compliance test suite. If one is looking for a rich featured open portal with aggregation, user management and so forth, Jetspeed is the better choice.

7.1.2 Portlet Container

The portlet container is the heart of Pluto. It is responsible for executing portlets and providing them with the necessary runtime context. Additionally, a portlet container is responsible for the portlets lifecycle – it can start and stop a portlet when required – as well as handling runtime services such as session management. More details follow later in the chapter.

In order to be able to plug the portlet container in other portals, the portlet container defines interfaces to all necessary components, such as portal or database. These interfaces are depicted in Figure 7.3.

One of the interfaces to the portal is called `PortletContainerInvoker`. Among other things, it provides the ability to render portlets. Additionally, the portlet container defines some service provider interfaces that need to be implemented by the portal to retrieve information about the portal. The most important service provider is called `InformationProvider` and handles request based information like the current locale.

The interface to the database is called `PortletObjectModel` and defines an abstraction layer for the data that the portlet container is working with. A portal's persistence layer may store that data in any way as long as the interface complies with the requirements.

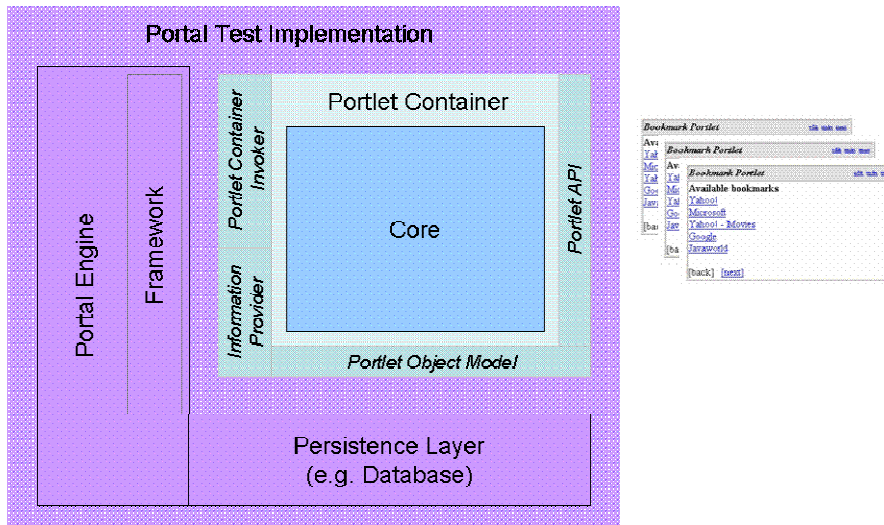


Figure 7.3: Interfaces of the Portlet Container

All portlet container interfaces and components will be described in more detail in the architecture section of this chapter.

7.1.3 Test Portal Implementation

The Test Portal component of Pluto is the entry point to the portlet container. It is used to

- access portlets via a browser,
- run the compliance test suite and
- demonstrate how the portlet container is integrated into a portal

A portlet container can never run without being integrated into a portal because it needs a portal context. The context comprises capabilities such as URL-Generation or resource handling. Moreover, the portal provides an entry point to the portlet container by providing a graphical user interface via the browser in which portlets are displayed.

The portal is implemented as a servlet that can be accessed via any browser or HTTP client. To render portlets, the portal calls the portlet container's `PortletContainerInvoker` interface and passes the servlet request and response. As a consequence, the portlet container can only be called from portals or clients that have a servlet request and servlet response at hand.

7.2 The portlet container architecture

In this section, we explain the architecture and underlying concepts of the portlet container inside Pluto. We start with a look at all of the components of the portlet container, as well as showing their purpose and role within the system. Then we focus on the interfaces that allow integrating the portlet container into other portals. Later in this section we look at the relation with J2EE and how the portlet container leverages the capabilities provided by the application server. In the last section we learn how the portlet deployment is being done and which hurdles we have to overcome.

7.2.1 Components

A normal application consists of a set of components. Each component fulfils a different task, some only support and help the main application in its tasks, while others take care of the application's purpose.

In this section, we will dive into the depths of the portlet container and reveal the secrets of each component. To get a first impression and a good overview over the portlet container's components, we put together Figure 7.4 and Table 7.1:

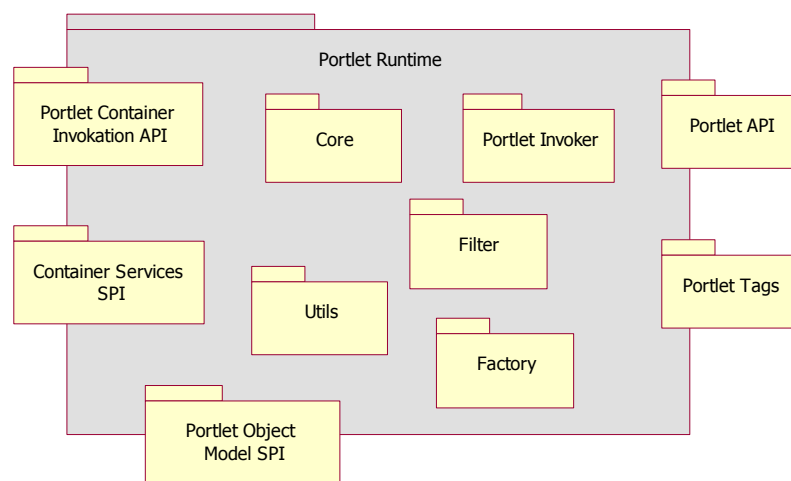


Figure 7.4: Pluto's components at a glance

With the overview in the following table, you gain an initial idea of the components comprising Pluto.

Table 7.x Pluto Components

Component	Description
Portlet Container Invoker API	This is the entry point of the portlet container. It is called from the portal to either render portlets or call the action method of portlets. This interface is described in more detail in section 7.2.3.
Container Services SPI	Several Interfaces enhance the portlet container with functionality of the portal. A portal implements these interfaces and provides instances of those to the portlet container. These interfaces are described in more detail in section 7.2.3.
Portlet Object Model	The Portlet Object Model represents an abstraction layer of portlet-related data necessary to run the portlet container. A portal can freely choose how to store the data in a backend system. This interface is described in more detail in section 7.2.3.
Core	This package contains the implementation of Portlet API.
Portlet Invoker	This is the equivalent to the Servlet Request Dispatcher for the portlet world. The Portlet Invoker is scoped to one portlet and provides methods to call action and render on a portlet.
Factory	The Factory Pattern is vastly used within Pluto. All objects are created by using factories to allow maximum flexibility and freedom, for example, performance improvements.
Portlet Tags	The Java Portlet Specification defines portlet tags that support portlets using Java Server Pages (JSP). This package contains the implementation of them.
Filter	Pluto provides a Portlet Filter capability in this package. Portlets or Portlet Container extensions can use this already to extend or modify the behavior of portlet methods.
Utils	Miscellaneous classes and utilities that make the life of a developer easier.
Portlet API	This package contains the API that is defined in the Java Portlet Specification and used by portlets. This interface is described in more detail in section 7.2.3.

Next, we will pick out the most important components and explain them in more detail. We will not discuss the obvious components that do not contribute a lot to the concept of a portlet runtime like utils.

Core

We consider the core component of Pluto as the most important one, as it takes care of the implementation of the Portlet API defined by the Java Portlet Specification. Figure 7.5 shows which interfaces are inherited by the Portlet API implementation. These interfaces are divided into three categories, the portlet capability (Portlet API), the servlet capability (Servlet API) and Pluto internal capabilities.

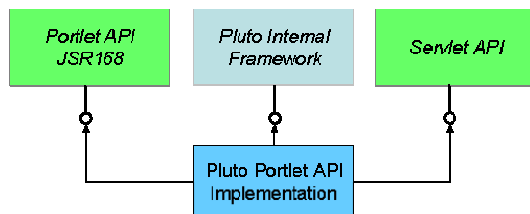


Figure 7.5: Inheritance of the Portlet API Implementation

Of course, the core implementation inherits from the Portlet API, as the main task of Pluto is to provide the functionality defined by the Java Portlet Specification. Additionally, it implements two more interfaces: Servlet API and an Internal Framework.

The Servlet API is used to align the Portlet Container with the Servlet Container. By doing this, we are able to reuse most of the Servlet Container's functionality without the need to re-code every aspect of the infrastructure already available in an application server. For instance, the implementation of `PortletRequest` implements `HttpServletRequest`, the `PortletResponse` implements `HttpServletResponse`, and so forth. More details will be explained in section 7.2.3.

The other interface is an internal framework of Pluto meant to generalize the access to internal objects and state, such as accessing the underlying Servlet API objects, as well as the corresponding Portlet Object Model. The framework helps to stay independent of any implementation classes throughout Pluto, as all other parts of Pluto do not need to know about any implementations, but only about any object implementing the given interface of the internal framework. For instance, the `InternalPortletRequest` interface allows access to the `HttpServletRequest` as well as the `PortletWindow`.

The Internal Framework comprises not only interfaces, but also an additional helper class. The helper allows access to the internal framework by simply passing any Portlet API object. This mechanism even supports filters, as described later in this section. Figure 7.6 gives an overview about the Internal Framework including the helper class.

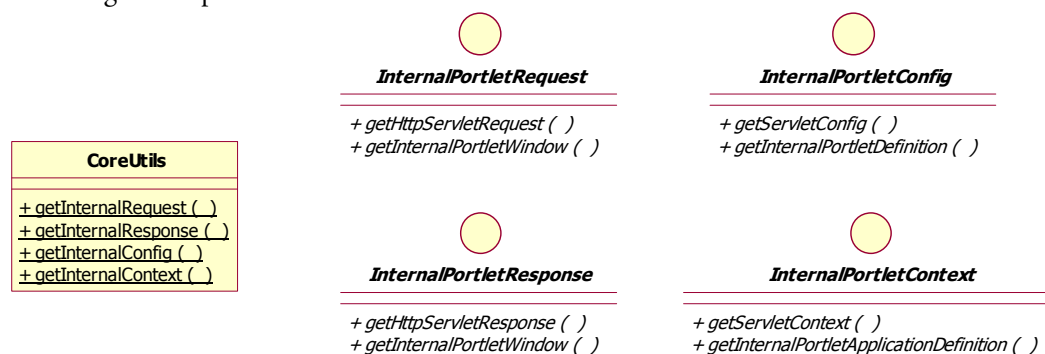


Figure 7.6: Pluto Internal Framework and Accessor Class

Portlet Invoker

This is a small component of Pluto, but rather important, because it provides functionality that is yet missing in the Java Portlet Specification and might get in some day. It corresponds to the Servlet Request Dispatcher for the portlet world, and provides methods to call action and render a portlet. Figure 7.7 shows the interface definition of the Portlet Invoker and its helper class. The helper class is necessary as there is no hook point in the existing Portlet API for the Portlet Invoker as for the servlet request dispatcher at the servlet context.

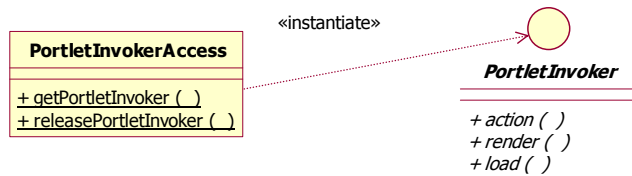


Figure 7.7: OO-Diagram of the Portlet Invoker

In order to retrieve a portlet invoker you need a portlet definition defined in the portlet object model. Additionally, the corresponding request/response pair is required to call a portlet method, such as action or render. Here is an example .of how to use the Portlet Invoker

```

PortletInvoker invoker;
invoker = PortletInvokerAccess.getPortletInvoker(portletDefinition);
invoker.render(renderRequest, renderResponse);
PortletInvokerAccess.releasePortletInvoker(invoker);
  
```

We see the Portlet Invoker as the equivalent functionality of the Request Dispatcher of the Servlet API. The following Table 7.2 compares both concepts and tries to show the similarities between them.

Table 7.2: Comparing Servlet Request Dispatcher with Portlet Invoker

Compared Item	Servlet Request Dispatcher	Portlet Invoker
Purpose	Used to call a method on another servlet	Used to call a method on another portlet
Creation	By passing the servlet name	By passing the portlet definition
Methods	Provides access to the service method	Provides access to each method of portlet interfaces

In the future, a Portlet Invoker, as we already have in Pluto, is likely to be standardized in a new version of the Java Portlet Specification.

Factory

The factory pattern is frequently used in Pluto. Object creations will always be handled through a factory. However, since Pluto is a pluggable component that is supposed to be used within a portal, it does not oppose any factory implementation of the portal. Instead, Pluto defines a container service providing factory pattern capabilities called FactoryManager, which describes all requirements of the portlet container implementation. Any portal integrating Pluto's portlet container only needs to provide an implementation of the factory manager to be able to host the portlet container.

Figure 7.7 illustrates how the factory pattern is used in the Pluto project. A class accesses a new object via FactoryAccess (1), which then passes on the call over the container service FactoryManager (2) to the portal engine (3). The factory implementation retrieved from the portal (4) is instantiating the accessed object (5) which is returned to the caller.

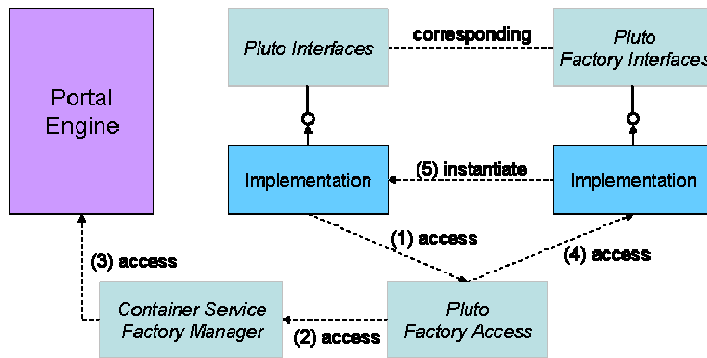


Figure 7.7: Shows how the factory pattern is used in Pluto

Filter

Portlet Filters are another bit of functionality not yet defined in the Java Portlet Specification. Regardless of that fact, we see this as a very important aspect of portlet and portlet container development. Therefore, we already introduced an implementation of portlet filters in Pluto. In a future release, they can be changed to the ones defined in a new Java Portlet Specification.

The concept of Portlet Filters is aligned with the concept of Servlet Filters. They allow wrapping the request, as well as the response object, which allows doing all sort of things. It can be as simple as to log each call, or more complex by modifying the behavior of the existing methods.

In Pluto, we only provide a subset of the capabilities of filters. A portlet developer can only wrap the request and response method; there is no lifecycle for filters, nor is there a registration mechanism that can be used to describe the filter class in any deployment descriptor. This will be implemented as soon as the Java Portlet Specification defines Portlet Filters as part of the standard. This is one topic that is desired by a lot of enterprises participating in the standardization effort, so that we most likely will see this functionality very soon.

Figure 7.8 depicts the portlet filter concept in Pluto. The wrappers are shown on purpose between the portlet container and the portlet, because the existing mechanism allows filters to be written by the portlet container, the portal or the portlet. Literally every component in the request flow is able to write a filter as long as they use the filter classes provided by Pluto.

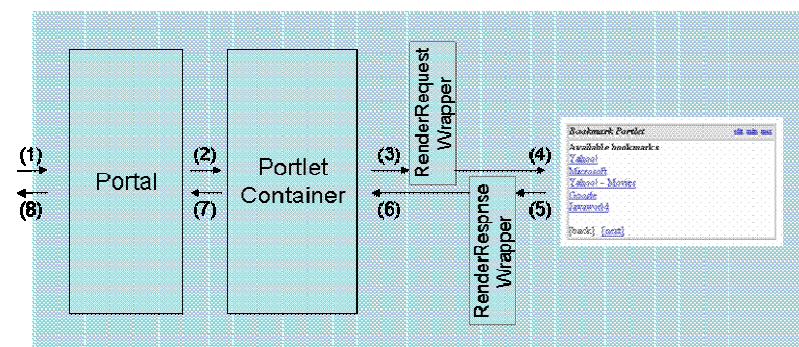


Figure 7.8: Portlet Filter concept in Pluto

7.2.2 Programming Model Interfaces

In this section, we explain the purpose of each interface, as well as which role they play in the overall picture. This lays the groundwork for the last section of this chapter, where we will describe how to use the portlet container in your own portal.

Figure 7.9 depicts the interfaces around the portlet container core, and gives an idea on how they relate to other components. Portlet Container Invoker, Container Services and Portlet Object Model are facing the portal and either are accessed or implemented by it. All other interfaces are internal to the portlet container. Therefore, we turn our attention in the next paragraphs on the first three interfaces, Portlet Container Invoker, Container Services and Portlet Object Model.

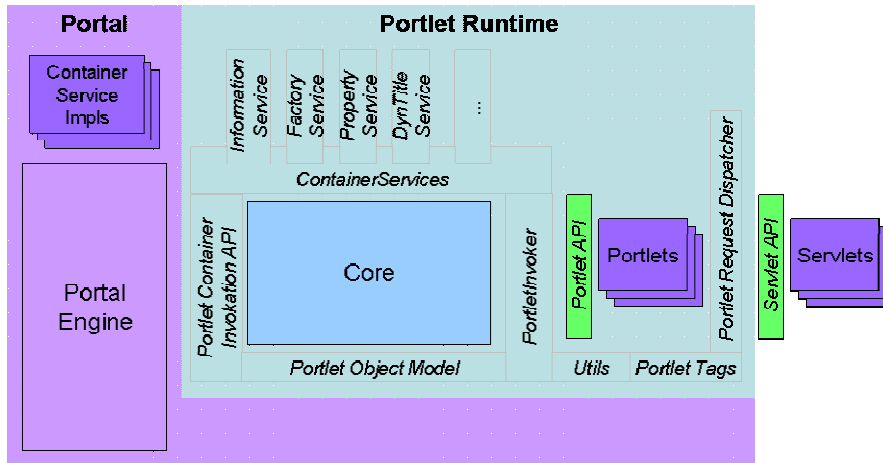


Figure 7.9: Interface surrounding the portlet container core

Portlet Container Invoker API

This is the entry point of the portlet container. In contrast to the other two interfaces described in this section, this is an application programming interface, and not a service provider interface. Consequently, this is the only interface of the portlet container that is actively called from the portal. The other interfaces have to be implemented by the portal and passed as parameter via the Portlet Container Invoker API to provide the portlet container with an instance of these interfaces.

The interface mainly allows a portlet container to call either the action or render method of a portlet. The portlet is defined by passing an object defined in the Portlet Object Model as parameter.

Container Services SPI

Container Services are a generic plug-in concept that allows extending the core portlet container with additional functionality. A container service is defined by an interface, accessed by the portlet container and provided by the portal. The portlet container mainly uses the services to transfer information between the portal and portlet container. In a few cases, the data flows in the opposite direction, from portlet container to portal.

The Container Service concept makes the portlet container independent of portal functions so that it might be used by different portals, and furthermore, new services can be plugged in to get a richer portlet container experience.

Let's take a closer look at two of the predefined container services.

InformationProvider—The InformationProvider is a callback mechanism for the portlet container into the portal, to get hold of necessary request-based information that can only be known by the portal, like URL generation or portlet mode and window state handling. The provider decouples the Portlet API layer of the portlet container that makes the Portlet API Java Portlet Specification conform to the basic information

source provided from the portal. The information source only needs to focus on providing information, and is not required to know the whole Java Portlet Specification but only the contract of the Pluto interfaces.

PropertyManager—The Java Portlet Specification defines a mechanism to transfer data between portlet container and portlets. The data is represented by a name/value pair of strings. It is meant to allow portals to transport proprietary information to portlets.

The portlet container extends this mechanism to the portal by providing a container service, called PropertyManager. This interface allows you to associate properties with the portlet request, which can be read from portlets. It also allows retrieving properties that might have been set at the PortletResponse by the portlet.

Portlet Object Model

The Portlet Object Model allows a portal to stay totally independent from the underlying implementation—it is transparent for the container if it is streamed into an XML file, a database, or even if there is a caching layer built in between. It represents the data of the portlet deployment descriptor that is processed and read by the portal during portlet deployment.

The portlet container only defines the interfaces that are necessary to execute it. A portal is free to extend this object model for its own needs as long as the requirements for the portlet container are fulfilled.

7.2.3 Relation to J2EE

Figure 7.10 gives an idea to what extent the portlet container reuses the functionality that is provided by the J2EE Specification through any Application Server. The Java Portlet Specification explicitly encourages this behavior and embeds itself into the J2EE Specification, since we want to reuse the given technology at hand. The areas reused by Pluto are:

- Classloader Handling
- Servlet Lifecycle Management
- HttpSession
- JSP Engine
- Deployment

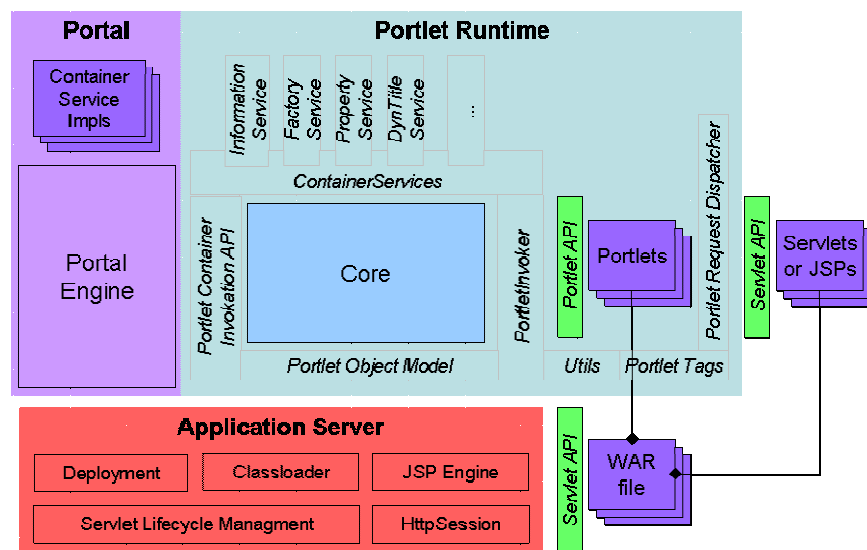


Figure 7.10: Relation of Portlet Container with J2EE and Application Server

In order to reuse the functionality provided by the application server, a portlet application needs to live as a web application. This means that first it has to reside within a web archive (WAR file). This is defined by the Java Portlet Specification, and every portlet container can work on this assumption. The second step to reach this goal is to embed each portlet into the servlet container by wrapping it with a servlet.

In the architecture of Pluto, we specify that each portlet has to be wrapped with a servlet. Figure 7.11 shows the scenario and its flows.

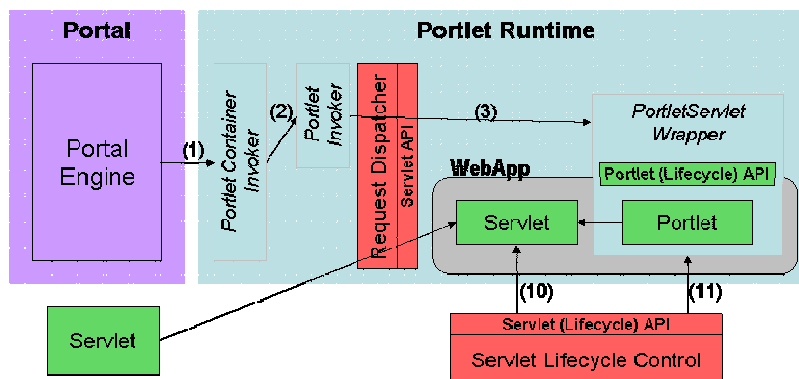


Figure 7.11: Detailed diagram of a Portlet embedded in a Servlet

In the following section, we describe how both architectural directions combined allow us to reuse the given functionality.

Servlet Lifecycle Management

Since each portlet is wrapped within a servlet, the servlet container instantiates the servlet as soon as the corresponding web application is started or when it is accessed. It also takes the servlet out of service when it is not required anymore. This can happen though a server shutdown or a failure condition. In figure 7.11 it is shown with the numbers 10 and 11.

The wrapper servlet only needs to instantiate the wrapped portlet in the `init()` method and destroy it in the `destroy()` method.

ClassLoader Handling

The J2EE Specification says that each module must have its own classloader. In this context, the WAR file is a specialized module named web module. The Java Portlet Specification defines that a portlet application always resides in a valid web module. Consequently, each portlet application must live in the very same classloader as web artifacts defined in the web deployment descriptor and controlled by the servlet container.

In this case, we get the classloader handling for free as each portlet is wrapped by a servlet, and therefore the portlet is instantiated in the correct classloader.

HttpSession

The Java Portlet Specification says that a portlet must have access to the `HttpSession`. This allows transferring data between a servlet residing in the same web module and itself.

One of the flows shown in figure 7.11 is a request flow (Number 1, 2 and 3). The Portlet Invoker calls the portlet indirectly via the wrapper servlet by using the servlet request dispatcher. This is necessary for a couple of reasons. The servlet container switches to the context of another web application and thus also to another classloader. Additionally, it switches to the corresponding `HttpSession` of the other web application.

The wrapper servlet has then access to the correct `HttpSession` and provides it to the portlet.

JSP Engine

Each portlet can include other web artifacts, such as a servlet or JSP, by calling the portlet request dispatcher provided via the Portlet API. The portlet container uses the servlet request dispatcher under the hood to call the given web artifact. This approach ensures that the built-in JSP Engine of the application server kicks in as soon as a JSP is included.

Deployment

Since each portlet application must reside in a web module, we can easily reuse the deployment mechanism of the application server. It recognizes all web artifacts that are defined in the web deployment descriptor and processes them. This, in turn, means that the application server cannot process the portlets as they are defined in the portlet deployment descriptor. These have to be parsed by the portal and made available to the portlet container through the portlet object model. On the other hand, there is some information that has to be passed on to the servlet container, such as tag library definition or all wrapper servlets.

The next section describes the portlet deployment in more detail.

7.2.4 Portlet Deployment

In the previous section, we discussed portlet deployment in general, and how we are able to reuse the application server's deployment mechanism. In this section, we show you how this has been realized in the Pluto portal, as well as the problems that have to be overcome with this approach.

There are three phases involved during portlet deployment to be able to successfully deploy a portlet, which are described in detail in the next paragraphs.

Phase 1: Pre-Deployment

In this phase, we prepare the WAR file for portlet deployment. As described earlier, there is some information that needs to be passed on to the servlet container, such as the tag library definition. In Pluto portal, the web deployment descriptor is modified and the necessary part is added into the descriptor. Consequently, the application server deployment in phase two is able to recognize the tag library and provides it through the JSP Engine to any portlet JSP.

Another important piece of information necessary for the application server is the wrapper servlet definition per portlet. In Pluto portal, one servlet definition per portlet is added to the web deployment descriptor, so that the application server is able to call the wrapper servlet, which in turn can call the portlet.

Figure 7.12 depicts the flow of the pre-deployment phase. First, the deployment is issued by any user (1), then additional classes are added to the web module (3), as well as some modifications to the web deployment descriptor (4). Last, the prepared WAR file is ready to be deployed (2).

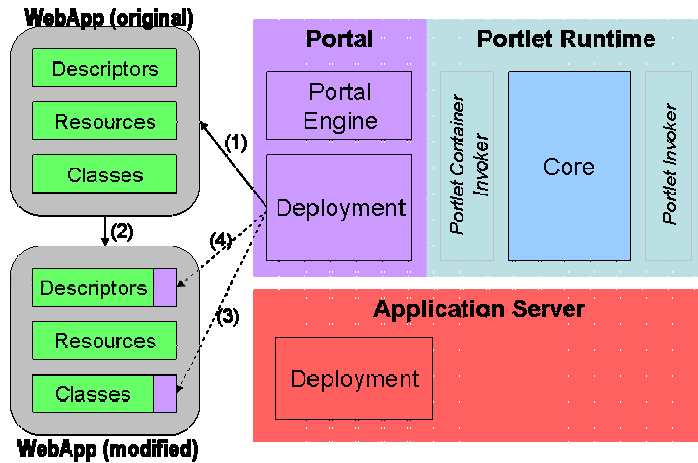


Figure 7.12: Prepare phase of portlet deployment

Phase 2: Deployment

From a portlet and portal perspective, this phase is the simplest and easiest one, as we only need to pass the prepared WAR file to the application server's deployment. The application server takes care of everything else in this phase.

Figure 7.13 demonstrates the flow of this phase. First the portal deployment starts the application server's deployment (1). Then the application server takes the prepared WAR file (2), and deploys it into the system (3).

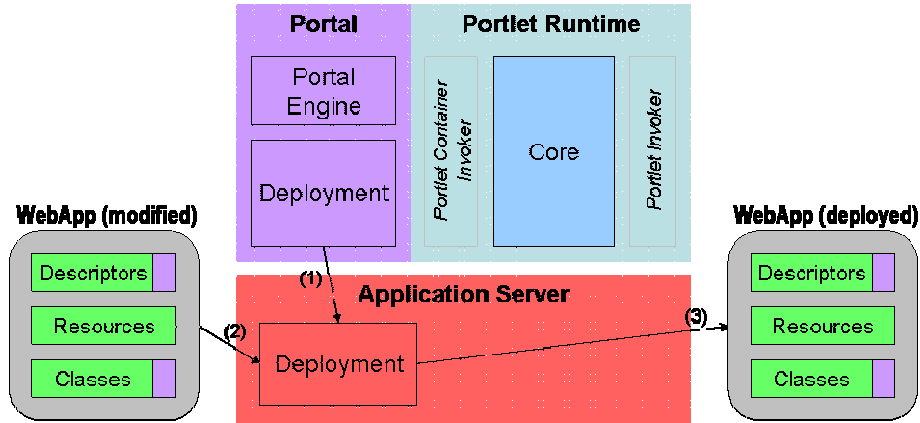


Figure 7.13: Application Server Deployment

Phase 3: Post-Deployment

After the WAR file is deployed into the application server, the last phase takes care of any remaining deployment tasks that are portlet-specific. These are parsing of portlet deployment descriptors and providing the gathered information to the portlet container via the portlet object model.

Figure 7.14 demonstrates the flow of this phase. First, the portal parses all portlet deployment descriptors (1), and then stores this information in the portlet registry (2). The registry is a means to provide the portlet information via the portlet object model to the portlet container (3).

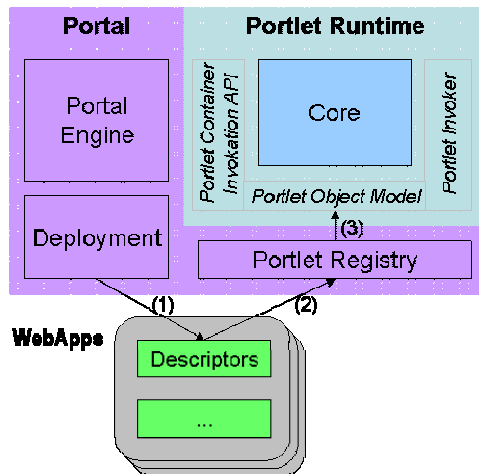


Figure 7.14: Post-Deployment phase of portlet deployment

The previous sections described Pluto in detail on a theoretical level. In this section, we will get practical and set up Pluto hands-on by showing you step-by-step how to get Pluto running on your system. While going through the different steps, you will also learn how to develop Pluto in Eclipse.

7.3 How to set-up Pluto

Pluto is a sub-project of the Apache Portals project. The Portals project defines itself as a collaborative software development project dedicated to providing robust, full-featured, commercial-quality, and freely available Portal-related software on a wide variety of platforms and programming languages.



More information about Apache Portals in general can be found at <http://portals.apache.org>. The starting point for the following sections is <http://portals.apache.org/pluto>.

7.3.1 Prerequisites

Apache Pluto is a server-side Java application that extends the J2EE environment with portlet capabilities. Therefore it has the following prerequisites:

- Java Development Kit (JDK) 1.3 or higher, to compile the application
- Tomcat 4, to run the application
- Servlet API 2.3, if another Application Server is used rather than Tomcat 4
- Maven 1.0 or higher for the build system of Pluto

7.3.2 Retrieve Sources

There are different ways of getting the sources of Pluto. The easiest way is to download them from the Apache Mirror System. They are stored as compressed archives, for instance as zip archive, and contain one snapshot, from either a nightly build or stable release, of the CVS repository. The Portals Project has a download page that amongst others provides a link to the Apache Mirror System. It can be found at <http://portals.apache.org/download.html>.

Another and more convenient way for developers to download the sources for Pluto is to directly access the CVS repository and download it from there. We will demonstrate in the remainder of this section how to setup and download the source from the CVS at the example of Eclipse. For all who already know the process, we summarized the necessary settings in Table 7.3.

Table 7.3: Required settings for accessing Pluto’s cvs repository.

Item	Description
Connection type	pserver
User	Your username or “anoncvs” for anonymous access
Password	Your password or leave empty for anonymous access
Host	cvs.apache.org
Port	Default
Repository path	/home/cvs or /home/cvspublic for anonymous access
Module	portals-pluto (Please note that even though the Pluto project has been moved from jakarta.apache.org to portals.apache.org, the name of the module was still jakarta-pluto at the time this book was written)

To get a project with the sources of Pluto in Eclipse, we will show you through a couple of steps. First, you need to switch to the “CVS Repository Exploring” perspective in Eclipse, and create a new repository location by using the context menu of this perspective as shown in Figure 7.15

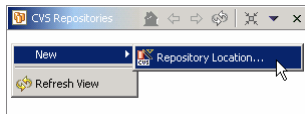


Figure 7.15: Create a new repository location by going to New, then Repository Location from the context menu.

In the dialog box, enter the settings listed in Table 7.3 as shown in Figure 7.16 and then finish the dialog:

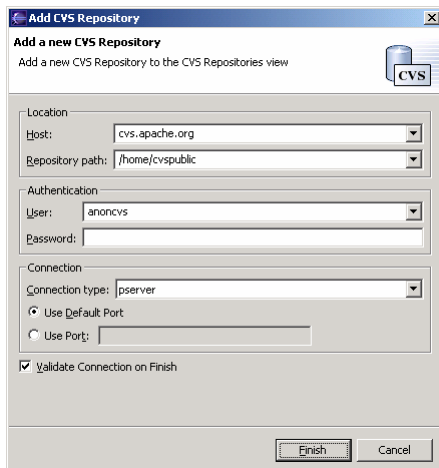


Figure 7.16: Enter all required settings to finish creating the new repository location.

After creating the new repository location, you will have one entry in the “CVS Repositories” view of the perspective. Next, we check out Pluto into Eclipse as a Java project. Then, open the just-created repository, and then its child “HEAD.” This is the main development stream of any CVS project. The list comprises all projects that are part of this repository, one of them being portals-pluto (or Jakarta-pluto). By using the context menu, you can easily check out any CVS project as a Java project into Eclipse. This is shown in Figure 7.17.

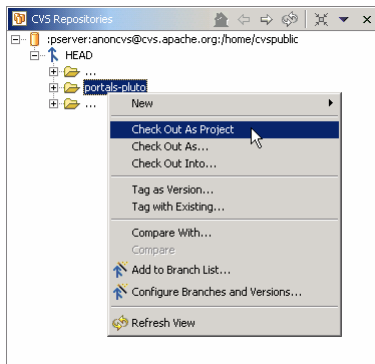


Figure 7.17: Check out portals-pluto as Java project into Eclipse. Right-click, then select Check Out As Project.

As soon as you select the “Check Out As Project” entry, Eclipse will create a new Java Project, download all files part of that CVS project, and store them in the new Java Project. Additionally, Eclipse shares this new project automatically, which means that the local Java project is linked with the remote CVS project so that it is very easy to synchronize the files with the CVS repository.

We have now successfully downloaded all source for Pluto, and created an appropriate project in Eclipse. You will notice that there are some errors showing up in the Eclipse Task view that we will address in the next section.

7.3.3 Build Environment

The build environment of Pluto is based on Maven. We will show you how to use Maven in order to get the build environment set up and thus get Pluto compiled. However, we do not describe Maven in detail, as there is already enough material out there that you can look up, such as it can be found on the maven homepage <http://maven.apache.org>.

First of all, please make sure that you have Maven installed, and the required system environment variables set, such as MAVEN_HOME. As soon as this is finished, we can start building and deploying Pluto. Open a command line window, and go to the root directory of Pluto. Start Maven with the command line option `fullDeployment` to build and deploy Pluto in one step, as shown in Figure 7.18. This will take care of everything for you, including copying shared jars, and deploy the base Pluto portal along with the Portlet Test Suite. Since we did not specify the home directory of Pluto, the build will end with an error when it tries to install Tomcat. This is expected, as we will show you how to install Pluto in the next section.

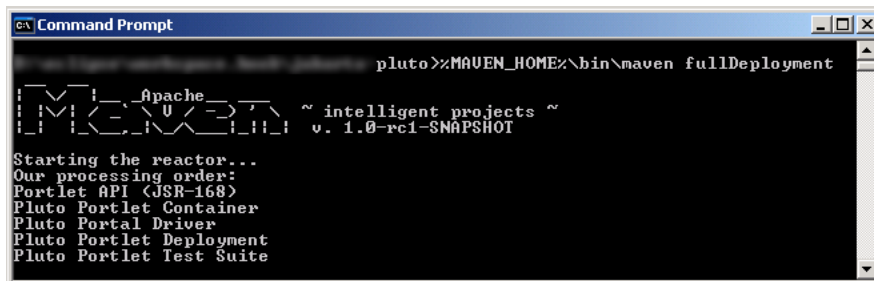


Figure 7.18: Building and deploying Pluto.

If you only want to build a subproject such as the container, switch to its subdirectory and call maven.

```
cd /container
%MAVEN_HOME%\bin\maven maven
```

After we successfully compiled Pluto with Maven, we can start using the Eclipse Development Environment. It was not possible earlier because one must run the Maven build at least once to setup the Maven repository. This is required to successfully compile in Eclipse.

Earlier we had some error messages showing up after we checked out Pluto from the CVS repository. Eclipse complains about a missing variable called MAVEN_REPO. We solve this problem by adding a classpath variable to the Java Environment of Eclipse. Therefore, open the Preferences dialog by selecting Window|Preferences in the menu. Then, go to Java|Classpath Variables in the left tree and press the button New. In the following dialog enter MAVEN_REPO as name and your actual Maven repository location as path. Hint: Normally your repository is located in your user home directory with the additional subdirectory /.maven/repository.

The result should look like Figure 7.19.

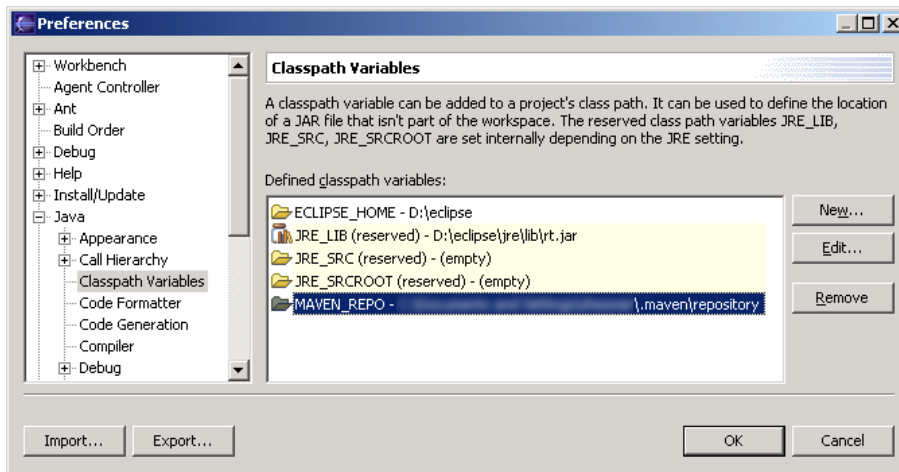


Figure 7.19: Add the classpath variable MAVEN_REPO.

7.3.4 Installation in Tomcat

In the previous sections, we have learned how to successfully compile Pluto with Maven and Eclipse. In this section, we will take the next step and finally get Pluto running in Tomcat. The Pluto community provides a script with Maven that automatically takes care of all the required installation tasks. However, when you start developing portlets or work on Pluto, it is an advantage to know what is going on behind the scenes. Therefore, we will show you all manual steps that are required to install Pluto in Tomcat as well.

In order to install Pluto, you need a file named build.properties in the root directory of the Pluto project. It is the same directory where you can find the maven.xml and project.xml. You can use the build.properties.sample as a template. In this file, add a property with the name maven.tomcat.home, and as value, the directory of the Tomcat installation. The build environment uses this property to locate Tomcat. There is also a property with the name maven.tomcat.version.major that specifies the version of Tomcat. Pluto supports version 4 and 5, at the time this book was written.

After setting up the environment correctly, we are able to start the automatic installation by starting Maven with the command line option fullDeployment. This time the build will not end with an error because we specified the location of Tomcat. Therefore, the build will copy shared jars, and deploy the base Pluto portal along with the Portlet Test Suite.

We are all set now to start Tomcat and access Pluto using the Browser. Enter the following address to access Pluto: <http://localhost:8080/pluto/portal>. As soon as the page shows up, select the hyperlink test to see if Pluto really can display portlets. The resulting page should look like Figure 7.20.

Test	<p><i>Test</i></p> <p>Test Portlet #1 edit help view max min nor</p> <p>This portlet is a portlet specification compatibility test portlet. It provides several tests of varying complexities which will assist in evaluating compliance with the portlet specification. It was originally developed for testing Apache Pluto, however, it does not utilize any proprietary APIs and should work with all compliant portlet containers.</p> <p>Please select one of the following tests:</p> <p>Simple Parameter Test Test</p> <p>Simple Attribute Test Test</p> <p>Complex Attribute Test Test</p> <p>External App Scoped Attribute Test Test</p> <p>Context Init Parameter Test Test</p> <p>Simple Preference Test Test</p> <p>Portlet Mode Test Test</p> <p>Window State Test Test</p> <p>Misc Test Test</p>	<p>Test Portlet #2 edit help view max min nor</p> <p>This portlet is a portlet specification compatibility test portlet. It provides several tests of varying complexities which will assist in evaluating compliance with the portlet specification. It was originally developed for testing Apache Pluto, however, it does not utilize any proprietary APIs and should work with all compliant portlet containers.</p> <p>Please select one of the following tests:</p> <p>Simple Parameter Test Test</p> <p>Simple Attribute Test Test</p> <p>Complex Attribute Test Test</p> <p>External App Scoped Attribute Test Test</p> <p>Context Init Parameter Test Test</p> <p>Simple Preference Test Test</p> <p>Portlet Mode Test Test</p> <p>Window State Test Test</p> <p>Misc Test Test</p>
----------------------	--	---

Figure 7.20: Browser's view of Pluto showing the Test page.

The manual steps involved to get Pluto installed in Tomcat are shown in the following list of tasks:

- Copy the jar files <PLUTO_HOME>/container/target/pluto-1.0.1.jar and <PLUTO_HOME>/api/target/portlet-api-1.0.jar to <TOMCAT_HOME>/shared/lib
- Unzip the web archive <PLUTO_HOME>/portal/target/pluto.war to <TOMCAT_HOME>/webapps
- Copy the xml file <PLUTO_HOME>/portal/pluto.xml to <TOMCAT_HOME>/webapps

After you have processed all the tasks, you can start Tomcat and access Pluto. However, you won't see any portlets, as we did not yet deploy portlets into Tomcat by hand. This is shown in the next section.

7.3.5 Deployment of Portlets

The deployment of portlets into the Pluto Runtime is not as easy as unzipping any web application into the webapps directory of tomcat. Pluto needs to prepare the portlet web application first to run it later in its runtime. This special processing modifies, amongst others, the deployment descriptors, or provides the portlet tag library. The deployment of portlets is handled by the Maven subproject "deploy" and is surfaced through the Maven build.

To deploy portlets through the Maven build, use the following command line:

```
maven deploy -Ddeploy=<PORTLET_WAR>
```

The portlet web application option must include the complete path, such as shown in Figure 7.21 at the example of the testsuite portlet.

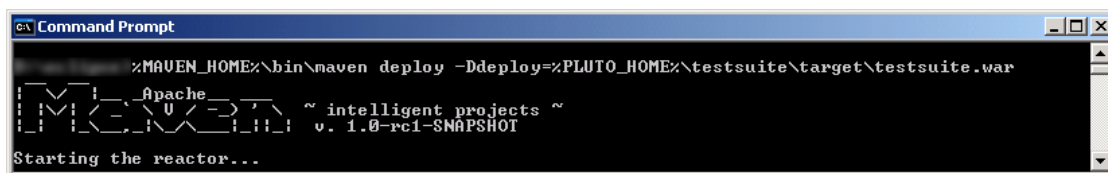


Figure 7.21: Deploying the testsuite portlet application.

If you are interested in more information, or more recent information about the installation of Pluto, you can look on the home page of Pluto at <http://portals.apache.org/pluto/install.html>.

7.4 *How use Pluto Container in your own Portal*

So far this chapter has given you an overview of Pluto, shown you the different parts and components of Pluto, as well as how to configure and install it. In this section, we demonstrate how you can put the knowledge gained into practice. First, we will quickly recap what we have already learned, and thereafter show you step-by-step what needs to be fulfilled to successfully integrate Pluto into your own portal.

You might wonder why there is a need to write your own portal at all. As a matter of fact, there are numerous reasons. One for sure is that there is only a test portal implementation within Pluto, which might not fulfill your needs. On the other hand there is Jetspeed and other Portal projects that already provide you with the required capability. Whether they have all the capabilities or not, in the end there is always one missing piece, which either leaves you writing your own portal or extending another project. In all these cases the following sections provide you with very useful information to start right into it.

7.4.1 *Preparation*

The following list provides a complete overview of the pieces needed to integrate the portlet container of Pluto. The services shown in the following list are implementations of the interfaces that are required by the portlet container to be available in order to work.

- Container Service - Information Provider (mandatory)
- The InformationProvider is used to get hold of information that can only be known by the portal, like URL generation, portlet mode and window state handling, etc.
- Container Service - Factory Manager (mandatory)
- The FactoryManager provides the portlet container with the ability to use the factory pattern. The portlet container asks for a factory that returns an implementation of the given interface.
- Container Service - Log (mandatory)
- This is an abstraction of the logging mechanism used in the portlet container, so that the portal can decide what logging facility to use.
- Portlet Object Model (mandatory)
- The object model provides the portlet container with the all required information of the portlets that it is supposed to execute. As the portal is responsible for deploying the portlets, it also has all information stored from the deployment descriptor. The portlet object model returns this information via well-defined interfaces.
- Container Service - Property Manager (optional)
- This interface allows you to associate properties with the portlet request, which can be read from portlets. It also allows retrieving properties that might have been set at the PortletResponse by the portlet.
- Container Service - Dynamic Title (optional)
- This Container Service allows a portal to support dynamic portlet titles as defined in the JSR 168.

After we have shown you what is necessary to use the portlet container in your portal, we next look at the question of how all this information and functionality provided through these interfaces is passed on to the portlet container. The answer is simple: By the only API provided by the portlet container, the Portlet Container Invoker API. There are two different points where the portal passes the information:

1. The Container Services are passed on to the portlet container when calling the init method. Therefore the portal must pass a context, manifested in an implementation of the interface `PortletContainerEnvironment`, which gives the portlet container access to the different container

services.

2. The Portlet Object Model is passed each time the portlet container is called during a request phase. Since all interfaces of the object model are linked together the portal only have to pass the `PortletWindow`. All other information that is required by the portlet container can be retrieved from this starting point.

The next two sections will demonstrate how to apply all that by showing examples and source code snippets that you can code in your portal to access the portlet container.

7.4.2 Container Services Initialization

Among others, the Portlet Container Invoker API consists of lifecycle methods `init()` and `destroy()`. The `init()` method takes several parameters to initialize the portlet container. One of them is a `PortletContainerEnvironment`, which gives the portlet container the ability to access container services. The following code snippet shows the interface definition. The `ContainerService` interface is only used to set the type.

```
public interface PortletContainerEnvironment
{
    ContainerService getContainerService(Class service);
}
```

Listing 7.1 shows how a simple implementation might look. In Pluto, there is a more complex implementation that automatically detects the implemented interfaces via reflection.

Listing 7.1: Simple sample implementation of `PortletContainerEnvironment`

```
public class PortletContainerEnvironmentImpl implements PortletContainerEnvironment
{
    private Map services;
    public PortletContainerEnvironment(Map aServices)
    {
        iServices = aServices;
    }
    public ContainerService getContainerService(Class service)
    {
        return (ContainerService)services.get(service);
    }
}
```

Map containing interface of type class as name
and reference to container service as value

Constructor stores
passed map in
instance variable

Returns
implementation of
requested container

To initialize the portlet container correctly, a portal needs to pass the following parameters.

- A unique portlet container name, since there might be more than one instance of the portlet container to serve for different purposes, e.g. WSRP.
3. The servlet configuration.
4. The portlet container environment containing all container services.
5. General properties.

Since the portlet container is a portal component that must always be available and must work efficiently, we recommend handling it as a singleton. Therefore, it is best to initialize it in the servlet's `init()` method and take it off service in `destroy()`.

Listing 7.x depicts a sample of this issue.

Listing 7.x: Sample demonstrating the initialization of a portlet container

```
public void init (ServletConfig config) throws ServletException
{
    super.init (config);

    Map services = new HashMap();2
    services.set
        (LogService.class, getLogService());
    services.set
        (FactoryManagerService.class, getFactoryManagerService());
    services.set
        (InformationProviderService.class, getInformationService());
    services.set
        (PropertyManagerService.class, getPropertyManagerService());
    services.set
        (DynamicTitleService.class, getDynamicTitleService());

    PortletContainerEnvironment env =
        new PortletContainerEnvironmentImpl(services);

    getPortletContainer().init("pluto", config, env, new Properties());
}
```

← Init method of a servlet; called only once

Creating map containing container services

Creating PortletContainerEnvironment

Initializing PortletContainer "pluto"

After we have shown how the portlet container can successfully be initialized, we next look at all required and optional container services that have been defined by the Portlet Container.

7.4.3 Log Service

Since there are a lot of different logging facilities available, we wouldn't want to bind the portlet container to a specific solution. Therefore Pluto defines a simple interface that allows you to plug-in any logging facility as implementation. Thus, the portlet container is independent and can be reused in any portal environment that is able to fulfill the Log Service.

Listing 7.4 shows a very simplistic sample implementation of the Log Service. In Pluto there is already another implementation available that is using Commons Logging.

Listing 7.4: Simplistic Sample implementation of Log Service

```
public class LogServiceImpl extends LogService
{
    public boolean isDebugEnabled (String aComp) { return true; }
    public boolean isInfoEnabled (String aComp) { return true; }
    public boolean isWarnEnabled (String aComp) { return true; }
    public boolean isErrorEnabled (String aComp) { return true; }

    public void debug (String aComp, String aMsg)
    {
        System.out.println ("DEBUG " + aComp + " " + aMsg);
    }
    public void debug (String aComp, String aMsg, Throwable aThrow)
    {
        System.out.println ("DEBUG " + aComp + " " + aMsg);
    }
    public void info (String aComp, String aMsg)
    {

```



```

        System.out.println ("INFO " + aComp + " " + aMsg);
    }
    public void warn (String aComp, String aMsg)
    {
        System.out.println ("WARN " + aComp + " " + aMsg);
    }
    public void error (String aComp, String aMsg, Throwable aThrow)
    {
        System.out.println ("ERROR " + aComp + " " + aMsg);
        if (aThrow != null)
            aThrow.printStackTrace (System.out);
    }
    public void error (String aComp, Throwable aThrow)
    {
        error (aComp, "An exception has been thrown:", aThrow);
    }
}

```

Note: At the time this book was written, the Pluto community discussed removing one abstraction layer with regard to logging in favor of Commons Logging, as it allows you to plug in your own logging facility as well. Therefore, it might be possible that this Log Service has already been removed from future builds. The only implication is that a portal needs to package Commons Logging with it in order to use the portlet container. Please refer to the documentation of Commons Logging to learn how to use their plug in mechanism.

7.4.4 Factory Manager Service

The factory pattern is a generic piece of software that can be used in all components of an application. With that idea in mind, Pluto leverages this as a standard feature of a portal since the container is only one single component within the whole application. Another reason was to allow the portal to change implementations if necessary and gain better flexibility. For instance, the WSRP4J implementation used this once to adapt the portlet container to their usage. However, it is not recommended to return different implementations than the ones provided from Pluto because you risk failing the Compliance Test Suite and not being in standard conformance anymore.

The Factory Manager Service defines an interface with only one method that takes an interface as parameter. The return value is expected to be a factory, which has the ability to return an object of the interface that has been passed to this method. Example: When passing `javax.portlet.RenderRequest` into the method, it must return an implementation of the interface `org.apache.pluto.factory.RenderRequestFactory`. This factory has the ability to return an implementation of `javax.portlet.RenderRequest`.

Listing 7.5 demonstrates an implementation of the Factory Manager Service.

Listing 7.5: Demo implementation of a Factory Manager

```

public class BookFactoryManagerServiceImpl extends FactoryManagerService
{
    private final static String CONFIG_FACTORY_PRE = "factory.";
    protected void init(ServletConfig config) throws Exception
    {
        ServletContext context = config.getServletContext();
    }
}

```

Prefix for all context parameters regarded by the algorithm

Called to initialize the FactoryManager

```

Enumeration configNames = context.getInitParameterNames();
while (configNames.hasMoreElements())
{
    String configName = (String) configNames.nextElement();
    if (configName.startsWith(CONFIG_FACTORY_PRE))
    {
        String factoryInterfaceName =
            configName.substring(CONFIG_FACTORY_PRE.length());
        String factoryImplName =
            context.getInitParameter(factoryInterfaceName);

        try {
            Class factoryInterface = Class.forName(factoryInterfaceName);
            Class factoryImpl = Class.forName(factoryImplName);

            Factory factory = (Factory) factoryImpl.newInstance();
            factory.init(config, new HashMap());

            factoryMap.put(factoryInterface, factory);
        }
        catch (Exception e) {
            // do logging
            throw e;
        }
    }
}

protected void destroy(ServletConfig config)
{
    Set factoryList = factoryMap.entrySet();
    for (Iterator iterator = factoryList.iterator(); iterator.hasNext();)
    {
        Factory factory = (Factory) iterator.next();
        try {
            factory.destroy();
        }
        catch (Exception exc) {
            // do some logging
        }
    }
    factoryMap.clear();
}

public Factory getFactory(Class theClass)
{
    return ((Factory) factoryMap.get(theClass));
}

private Map factoryMap = new HashMap();
}

```

← Iterate through all context init parameters

← Check for prefix, otherwise it is no factory definition

Get Class representations of String representations

Instantiate and initialize factory

← Store factory interface and implementation for later usage

Only this method is required by the FactoryManagerService

← Return factory for passed interface

Note: At the time this book was written, the Pluto community discussed rectifying the factory manager's implicit dependency between the requested interface and the returned implementation. The

Pluto Portlet Container uses the factory manager in a way that it passes an interface (e.g. `javax.portlet.RenderRequest`) and expects an implementation of the factory (e.g. `org.apache.pluto.factory.RenderRequestFactory`) that returns an implementation of the passed interface. The desired behavior would be to request a factory by passing an interface (e.g. `org.apache.pluto.factory.RenderRequestFactory`) and get an implementation of that interface.

7.4.5 Information Provider Service

The Information Provider is a callback mechanism for the portlet container into the portal, to get hold of necessary request-based information that can only be known by the portal. In contrast to all other container services, it is the largest one, and the one that is used the most. The Information Provider can be sub categorized in the following elements:

- Static: Information Provider
 - This provider deals with non-dynamic information that holds true for the lifetime of a portal.
- PortalContextProvider
 - This provider transfers the information required for the Portlet API object `PortalContext` from portal to portlet.
- Dynamic Information Provider
 - This provider deals with dynamic information that changes between every request. Besides the shown sub providers it deals with miscellaneous request dependent data, such as portlet modes and window states.
- PortletURLProvider
 - This provider is responsible for creating URLs that point to portlets, and at the same time, support all additional features of the Java Portlet Specification like render parameters.
- ResourceURLProvider
 - This provider is used to return URLs pointing to a resource when calling `encodeURL` of the Portlet API.
- PortletActionProvider
 - This provider is used to handle portlet action. For instance, the portlet container uses this provider to inform the portal of portlet mode or window state changes during the action phase.

Listing 7.6 demonstrates an implementation of the Information Provider Service that gives access to all the previously described providers. Since there are two different information providers with a defined scope, the implementation is able to optimize the handling. The static information provider is created only once, and in all subsequent calls, it is returned out of the servlet context. In contrast, the dynamic information provider is created anew for each request, and in all subsequent calls, during the same request is returned out of the servlet request.

Listing 7.6: Sample implementation of the Information Provider Service

```
public class InformationProviderServiceImpl
    implements InformationProviderService
{
    private ServletConfig servletConfig;

    public void init(ServletConfig config)
    {
        servletConfig = config;
    }
}
```

← Initialize the implementation during server startup, e.g. servlet's init method

```

public StaticInformationProvider getStaticProvider()
{
    ServletContext context = servletConfig.getServletContext();
    StaticInformationProvider provider =
        (StaticInformationProvider)context.getAttribute(
            "org.apache.pluto.portalImpl.StaticInfoProv");
    if (provider == null) {
        provider = new StaticInformationProviderImpl(servletConfig);
        context.setAttribute(
            "org.apache.pluto.portalImpl.StaticInfoProv", provider);
    }
    return provider;
}

public DynamicInformationProvider getDynamicProvider
    (HttpServletRequest request)
{
    DynamicInformationProvider provider =
        (DynamicInformationProvider)request.getAttribute(
            "org.apache.pluto.portalImpl.DynamicInfoProv");
    if (provider == null) {
        provider = new DynamicInformationProviderImpl(
            request, servletConfig);
        request.setAttribute(
            "org.apache.pluto.portalImpl.DynamicInfoProv", provider);
    }
    return provider;
}
}

```

Retrieve a previously
stored static
information provider

Create and
store it in the
servlet context

Retrieve a previously
stored
dynamic information

Create and store it in
the servlet request

So far we have shown you the beginnings of how to create your own Information Provider Service. The next required tasks include implementing the static and dynamic information provider as well as their sub interfaces. Since these are very extensive and large interfaces, we will not provide a sample implementation of each information provider along with a description, as this would take away a fair amount of pages from this book. Instead, we will provide you with a general description for the remaining interfaces in this section, and provide you with a sample implementation in the demo sports portal.

The `StaticInformationProvider` handles non-dynamic information that normally does not change during a server lifecycle. Therefore it can be implemented as a singleton. It explicitly deals with the `PortalContext` and `PortletDefinition`. **Note:** At the time this book was written, the Pluto community discussed to removing the method to receive `PortletDefinitions` from the portal. It is possible that a newer version of Pluto does not have this method at the interface anymore.

The `PortalContextProvider` is the counterpart to the `PortalContext` interface of the Portlet API. The portlet container cannot contribute to this information at all so that all information that a portlet tries to get from `PortalContext` is directly requested from the portal via this provider. A portlet can access properties, supported portlet modes and window states as well as a generic portal info string.

The `DynamicInformationProvider` handles information that normally changes very often. It represents all kind of information; a full overview is provided in Table 7.4

Table 7.4: Overview of all information required by the portlet container.

Item	Interface/Method(s)	Description
PortletMode	getPortletMode isPortletModeAllowed	Returns the portlet mode for a portlet window or whether it is allowed to switch to another mode.
WindowState	getWindowState isWindowStateAllowed	Returns the window state for a portlet window or whether it is allowed to switch to another state.
ContentTypes	ServletReq.getContentType	Returns the content type of the client request.
ResponseContentTypes	getResponseContentType getResponseContentTypes	Returns the allowed content types that a portlet might use during its response.
Locales	ServletReq.getLocale ServletReq.getLocales	Returns the locales that a portlet can use during its response.
RenderParameters	ServletReq.getParameter ServletReq.getParameterNames ServletReq.getParameterMap	Returns the parameter and render parameters for a portlet window.
PortletURL	getPortletURLProvider setPortletMode setWindowState setAction setSecure clearParameter setParameters toString	Takes care of the URL generation: Therefore the interface takes a portlet mode, window state and render parameters for a portlet window. Additionally one can define whether the new URL is an action URL and/or a secured URL.
ResourceURL	getResourceURLProvider setAbsoluteURL setFullPath toString	Takes care of the URL generation with regard to resources, like graphics.
ActionHandling	getPortletActionProvider setPortletMode setPortletWindowState setRenderParameters setRedirectLocation	Provides the portal with all information that a portlet might have set during its action method.

7.4.6 Property Manager Service

The Java Portlet Specification defines a mechanism to transfer data between portlet container and portlets. The data is represented by a name/value pair of strings. It is meant to allow portals to transport proprietary information to portlets.

The Property Manager Service is introduced to support this feature. It is an optional service of the portlet container that allows the portal to support the property mechanism – it is not necessary to provide an implementation of this service if the portal does not want to provide additional information to portlets.

The Java Portlet Specification defines access to content types. However, the problem with content type is that it can represent multiple markups. Therefore, it is useful for a portlet to also get access to the markup of the current request.

Listing 7.7 shows an implementation that provides only one property to all portlets named myportal.markup. The portal recognizes the correct markup by analyzing the parameters of the servlet request, stores the result in it, and passes it on to the portlet by using the property manager. This example does not use the other direction in which the portlet sets properties at the response and communicates some information back to the portal.

Listing 7.7: Sample implementation of the Property Manager Service

```
public class PropertyManagerImpl implements PropertyManagerService
{
    public void setResponseProperties
        (PortletWindow window, HttpServletRequest request,
         HttpServletResponse response, Map properties)
    {
        // remember properties for later usage
    }
}
```

```

    }
    public Map getRequestProperties
        (PortletWindow window, HttpServletRequest request)
    {
        return Collections.singletonMap(
            "myportal.markup",
            new String[] { (String)request.getAttribute("myportal.markup") });
    }
}

```

Return Map containing all properties

Receive markup value that has been calculated by the portal

7.4.7 Dynamic Title Service

Portlets have the ability to not only define a static title in the portlet deployment descriptor, but also programmatically generate a dynamic title and pass it on to the portlet container by using `renderResponse.setTitle()` of the Portlet API. The portlet container passes this information along to the portal by using the container service Dynamic Title. It is an optional service of the portlet container that allows the portal to support this feature – it is not necessary to provide an implementation of this service if the portal does not support it.

The following code snippet shows the interface definition of the Dynamic Title Service.

```

public interface DynamicTitleService extends ContainerService
{
    public void setDynamicTitle
        (PortletWindow window, HttpServletRequest request, String dynamicTitle);
}

```

The sample implementation shown in Listing 7.8 extends this interface with an additional method which allows a portal to retrieve the dynamic title for a given portlet window. The portlet container sets the dynamic title by calling the `setDynamicTitle` method; afterwards the portal retrieves the title by calling `getDynamicTitle`. This sample implementation uses the servlet request to store the title.

Listing 7.8: Sample implementation of the Dynamic Title Service

```

public class DynamicTitleServiceImpl implements DynamicTitleService
{
    public void setDynamicTitle (PortletWindow window,
        HttpServletRequest request, String dynamicTitle)
    {
        request.setAttribute
            ("org.apache.pluto.dynamic_title."+window.toString(),
            dynamicTitle);
    }
    public String getDynamicTitle
        (PortletWindow window, HttpServletRequest request)
    {
        return (String)request.getAttribute
            ("org.apache.pluto.dynamic_title."+window.toString());
    }
}

```

This method is called by the portlet container to set the dynamic title

Store the dynamic title in the servlet request with appropriate

Retrieve the dynamic title from the servlet request for the given portlet window

7.4.8 Portlet Object Model

The Portlet Object Model allows a portal to stay totally independent from the underlying implementation - it is transparent for the container if it is streamed into an XML file, a DB or even if there is a caching layer built in between. It represents the data of the portlet deployment descriptor that is processed and read by the portal during portlet deployment.

The portlet container defines a set of interfaces for the Portlet Object Model that has to be implemented by any portal using the portlet container. The package name is `org.apache.pluto.om` and has five sub packages dividing it into logical sub-packages. The base package contains some type marker interfaces to uniquely mark a model or controller interface as well as a static class to access the appropriate controller for a given model. The sub packages are:

org.apache.pluto.om.common—This package describes functionality that is common with regard to the other sub packages, and does not apply to one specific package, such as the `Language` and `LanguageSet`. The functionality covered in this package consists of the interfaces shown in Table 7.5. To not make it too complex, we left out all controller interfaces.

Table 7.5: Overview of the interfaces used in the common package.

Interfaces/Methods	Description
Description	A localizable description
DescriptionSet	A set of Description objects
DisplayName	A localizable display name
DisplayNameSet	A set of DisplayName objects
Language getLocale getTitle getShortTitle getKeywords getResourceBundle	A localized object containing all portlet related display information like title, short title, keywords and the resource bundle
LanguageSet	A set of Language objects
ObjectID	An identifier used instead of a string for performance reasons
Parameter	Object representing a parameter containing name, value and a localized description
ParameterSet	A set of Parameter objects
Preference	Object representing a portlet preference containing name, values and a read only flag
PreferenceSet	A set of Preference objects
SecurityRole	Storing security related information read from the deployment descriptor
SecurityRoleRef	Storing security related information read from the deployment descriptor

org.apache.pluto.om.window—This package describes functionality that relates to a portlet window. A portlet window is the window in which the markup fragment produced by a portlet displayed. A portlet window is created whenever a user attaches a portlet entity to a specific portal page. This decoupling of the portlet window from the portlet entity allows having different windows pointing to the same portlet entity. This enables the user to have the same news portlet or calendar portlet on different pages.

To get a better impression of the interfaces in the window package, see Figure 7.22.

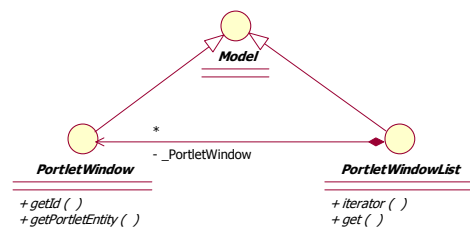


Figure 7.22: Overview of the relationship between the interfaces in the window package.

org.apache.pluto.om.entity—This package describes functionality that relates to a portlet entity. A portlet entity provides the user with the ability to customize the portlet. Portlet entities are portlet definitions with an additional user specific data set, called preferences in the case of JSR 168. Additionally, the notion of a portlet application entity is introduced with this package. The application is a container with the exactly the same scope as the portlet application as defined in the portlet deployment descriptor, except that it is on the entity level.

The functionality covered in this package consists of the interfaces shown in Table 7.6. To make it easy to look at we left out all controller and -Set interfaces.

Table 7.6: Overview of the interfaces used in the entity package.

Interfaces/Methods	Description
PortletEntity	The PortletEntity is an instantiation of a PortletDefinition that is bound to a portal resource
PortletApplicationEntity	The PortletApplicationEntity is an instantiation of a PortletApplicationDefinition that is bound to a portal resource. It contains a set (either all or a subset) of portlets that participate all in the same PortletApplicationDefinition

To get a better impression of the interfaces in the entity package, see Figure 7.23.

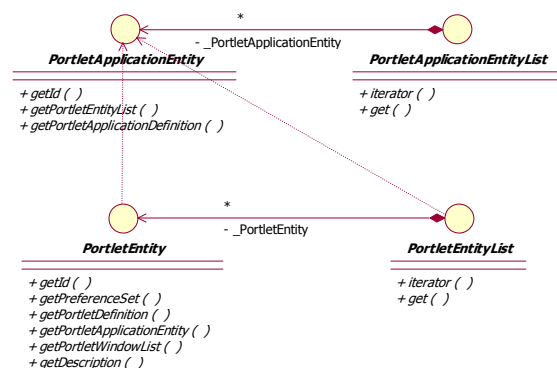


Figure 7.23: Overview of the relationship between the interfaces in the entity package.

org.apache.pluto.om.portlet—This package describes functionality that relates to a portlet definition. A portlet definition provides the administrator with the ability to customize portlet settings, like the stock quote server of a stocks portlet. Just like the entity package, the portlet package additionally has a portlet application definition. This portlet application definition matches exactly to the one defined in the portlet deployment descriptor, except that it might be modified during runtime of the portal, such as customizing portlet settings.

The functionality covered in this package consists of the interfaces shown in Table 7.7. To make it easy to look at, we left out all controller and -Set interfaces.

Table 7.7: Overview of the interfaces used in the portlet package.

Interfaces/Methods	Description
ContentType	An object containing all data related to the content type, such as the content type itself and the associated portlet modes
PortletDefinition	Describes a portlet (and its properties) that is not bound to any portal resource. A portal resource is any resource type available in a portal, such as a page
PortletApplicationDefinition	The PortletApplicationDefinition describes a set (either all or a subset) of portlets that participate all in the same WebApplicationDefinition

We do not show another interface diagram, since the portlet definition layer looks very much the same as the entity layer. Please see Figure 7.24 for an interface diagram containing the main interface of the object model.

org.apache.pluto.om.servlet—This package describes functionality that relates to a servlet definition. A servlet definition is a rather technical thing of Pluto and splits the portlet entries of the portlet deployment descriptor in a modifiable and fixed entry. The modifiable data is stored in the portlet definition, the fixed data is stored in the servlet definition. Just like the portlet package, the servlet package additionally has a web application definition. This definition matches exactly to the one defined in the web deployment descriptor.

The functionality covered in this package consists of the interfaces shown in Table 7.8. To make it easy to look at, we left out all controller and -Set interfaces.

Table 7.8: Overview of the interfaces used in the servlet package.

Interfaces/Methods	Description
ServletDefinition	Describes the fixed part of the portlet and its properties.
WebApplicationDefinition	The WebApplicationDefinition represents the ServletContext.

We do not show another interface diagram, since the servlet definition layer looks very much the same as the portlet definition layer. Please see Figure 7.24 for an interface diagram containing the main interfaces of the object model.

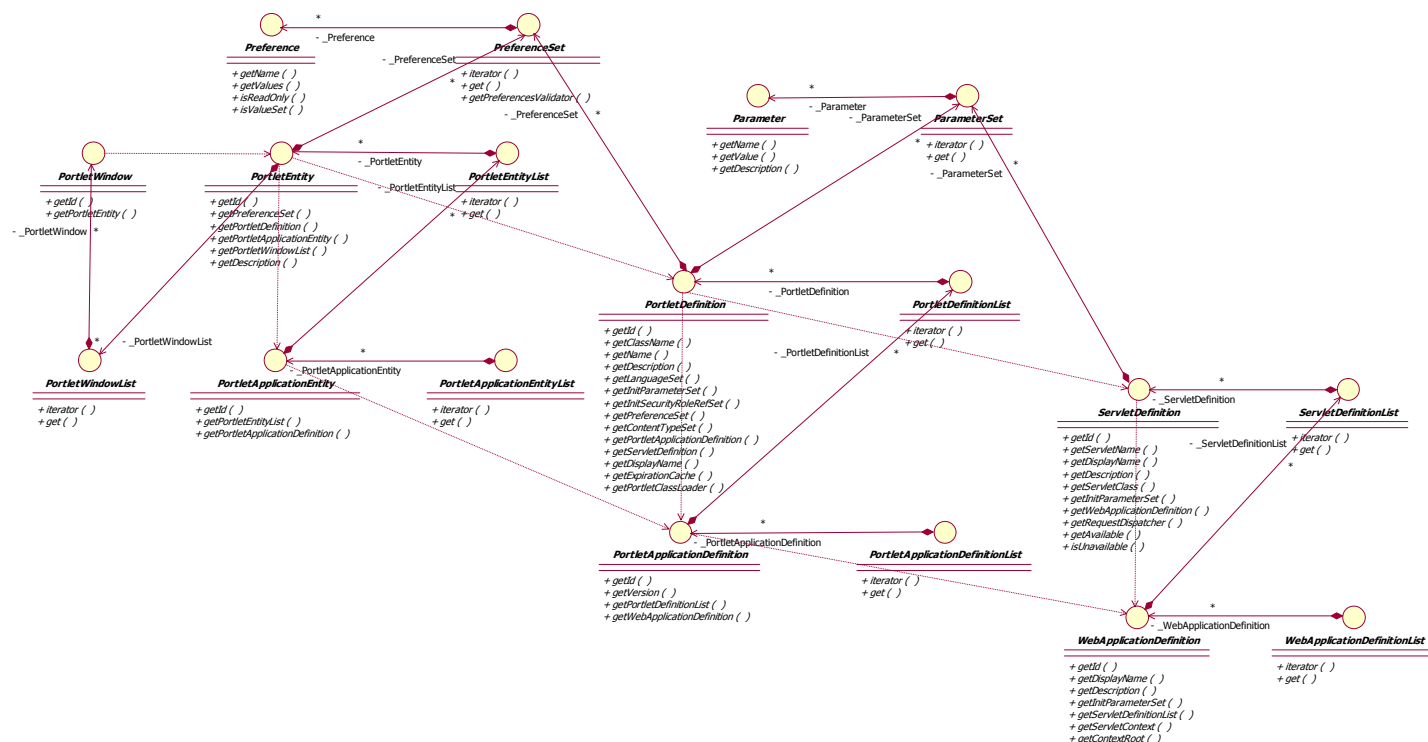


Figure 7.24: Overview of the main interfaces of the portlet object model.

As you have seen in the figure above, Pluto has a very flexible and extensive object model.

7.5 Summary

In this chapter, we introduced you to the Pluto Project by starting with an overview of the system, its goals and purpose. You have learned the meaning of the different parts, such as the Test Portal Implementation, its meaning and most important the reason why it is not integrated in Jetspeed nor why it would be against the Pluto community philosophy to create another full-featured portal.

Right after that, we started into the technical depths of Pluto and explained the techniques and patterns that are used in the Pluto Runtime. Important ones were the Core, responsible for joining the portlet and servlet world in order to leverage most of the servlet capabilities and thus saved a lot of duplicate code, and the factory pattern used to instantiate all objects within the runtime and thus giving a lot of freedom for performance improvements.

Another very important – if not the most important – part of this chapter describes the programming model of the portlet container which allows you to plug in the portlet container in your own portal. It defines the contract between portal and portlet container.

As described above, the Core unifies the portlet and servlet world. This is the groundwork for another important part of this chapter that explains in detail what part of J2EE the portlet container is leveraging, including all of the advantages.

After all the theoretical work, we switched to get our hands dirty and explained step-by-step how to download, install, and configure the Pluto environment. As a result, you have a latest version of Pluto running on your machine and ready to go.

As the last and final step in this chapter, we moved the bar one notch higher and explained how you integrate the portlet container in your own portal environment by using real source code examples. First, we showed you how to call the portlet container, and second, how to implement the Programming Model Interfaces accordingly. These two steps brought together enable you to use the portlet container in your own portal.

In the next chapter, we will explain Jetspeed in detail, starting with the architecture and components and digging deeper into it until we arrive at a point where we show how Pluto is integrated into Jetspeed and what was necessary to make this happen.

7.6 Resources

- [1] Apache Community, URL: <http://www.apache.org/>
- [2] Pluto, Reference Implementation of JSR 168, URL: <http://portals.apache.org/pluto/>
- [3] Jetspeed-2, URL: <http://portals.apache.org/jetspeed-2/>
- [4] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Addison-Wesley, 1995
- [5] Tomcat, Reference Implementation for Servlets and JSPs, URL: <http://jakarta.apache.org/tomcat/>
- [6] Eclipse, Software Developer Platform IDE, URL: <http://www.eclipse.org/>
- [7] The Java Portlet Specification, URL: <http://jcp.org/en/jsr/detail?id=168>
- [8] Tomcat, Homepage of Tomcat, URL: <http://jakarta.apache.org/tomcat/>

Chapter 8 Working with the Jetspeed Portal

In the previous chapter we saw how to run portlets in the Pluto portlet container and how to use the Pluto portal sample driver to see the portlets displayed in a browser. However, the Pluto portal driver is a very simple portal that is only useful for simple tests and not intended for real-world usage. This chapter will introduce you to the popular enterprise portal housed at Apache Portals: The Jetspeed Enterprise Portal that consists of all features you would expect from a real portal, like administration, configuration and security. The Apache Portals project, which currently hosts Jetspeed version 1 and Jetspeed version 2, is an open source community based on portal technology. It is a part of the Apache Software Foundation. Apache is the one of the most popular and most respected open source communities.

We will discuss both versions of the Jetspeed Portal in this chapter, because Jetspeed-1 remains in active use by a large community and will still be used by many people in the near future. If you are just getting started, however, we recommend jumping straight to Jetspeed-2 for its better standards compliance.

Along with an introduction to Jetspeed, a brief step-by-step guide to installing and running it is provided here. For more in-depth examples of using Jetspeed, see Part V of this book.

8.1 Introducing the Jetspeed Enterprise Portal

In Chapter 6 we explained in detail how J2EE portals are structured and architected and why they are designed that way. In this chapter we take a look at a concrete implementation of the principles discussed in Chapter 6. We'll start with a short introduction into Jetspeed and the differences between Jetspeed-1 and Jetspeed-2.

Jetspeed is an open source software project with full source code provided. Designed for easy personalization and customization, Jetspeed runs large enterprise portals or small business or organization portals out of the box.

The initial Jetspeed code base was developed between 1999-2002 under the Jakarta open source project at Apache. The initial Jetspeed is now known as Jetspeed-1. Jetspeed-1 was actually developed before portals were standardized, and so is not a standards-based portal implementation.. However it is still a very popular portal. The most current version of Jetspeed is Jetspeed Version 2, or simply Jetspeed-2. Jetspeed-2 has been under development since 2002, and is likely being released when this book is in your hands. Because it was developed at the same time as the relevant standards, it is based on standards, modern server software patterns and is designed for high performance enterprise portals.

Jetspeed was originally housed in the Jakarta project at Apache. The Jakarta project was an umbrella project for Java-related software projects. In January 2004, the Apache Portals project was created, and now houses several open source portal sub-projects, including Jetspeed-1 and Jetspeed-2.

8.1.1 Jetspeed-1

Jetspeed-1 has been around since 1999 and has a large user community. Jetspeed-1 uses XML extensively for display and back-end functionality including use of RSS feeds and XML data into portlets. The Portal Structure Markup Language (PSML) is used to store portal specific information including styles, personalization information and portlet registries.

Jetspeed makes network resources (applications, databases and so forth) available to end-users. The user can access the portal via a web browser, WAP-phone, pager or any other device. Jetspeed acts as the central hub where information from multiple sources is made available in an easy to use manner.

The data presented via Jetspeed is independent of content type. This means that content from for example XML, RSS or SMTP can be integrated with Jetspeed. The actual presentation of the data is handled via XSL and delivered to the user for example via the combination of Java Server Pages (JSPs) and HTML. Jetspeed provides support for templating and content publication frameworks such as Cocoon, WebMacro and Velocity. Jetspeed also supports WAP devices.

Jetspeed helps you build portal applications quickly. The goal is to make Jetspeed a tool for both portal developers as well as user interface designers. Currently the focus is on providing developers with a set of tools that facilitates building the base for the portal. With Jetspeed you can quickly build an XML portal and also syndicate your own content. You will see examples of these techniques in Chapters 12 – 13 of this book.

As mentioned before, Jetspeed-1 was created before any portlet standard was in place. Thus Jetspeed-1 has its own portlet API that is not compliant with other portal servers portlet APIs or the Java Portlet API.

8.1.2 Jetspeed-2

Jetspeed-2 is the next-generation enterprise portal at Apache. Jetspeed-2 offers several architectural enhancements and improvements over Jetspeed-1. First, Jetspeed-2 is conformant to the Java Portlet Standard and will provide a standard mechanism for the deployment of portlets. Second, Jetspeed-2 has matured to a more scalable architecture featuring multi-threaded functionality. Third, Jetspeed-2 is decoupled from several legacy open source projects. Fourth, Jetspeed-2 is based on component architecture that allows for easy enhancement of Jetspeed-2 with new features.

Figure 8.1 shows the basic architecture of Jetspeed-2. When you compare this picture with the basic architecture pictures of J2EE portals we showed in Figure 6.3 and 6.4 you can see how similar Jetspeed-2 is to the J2EE portal architecture of Chapter 6.

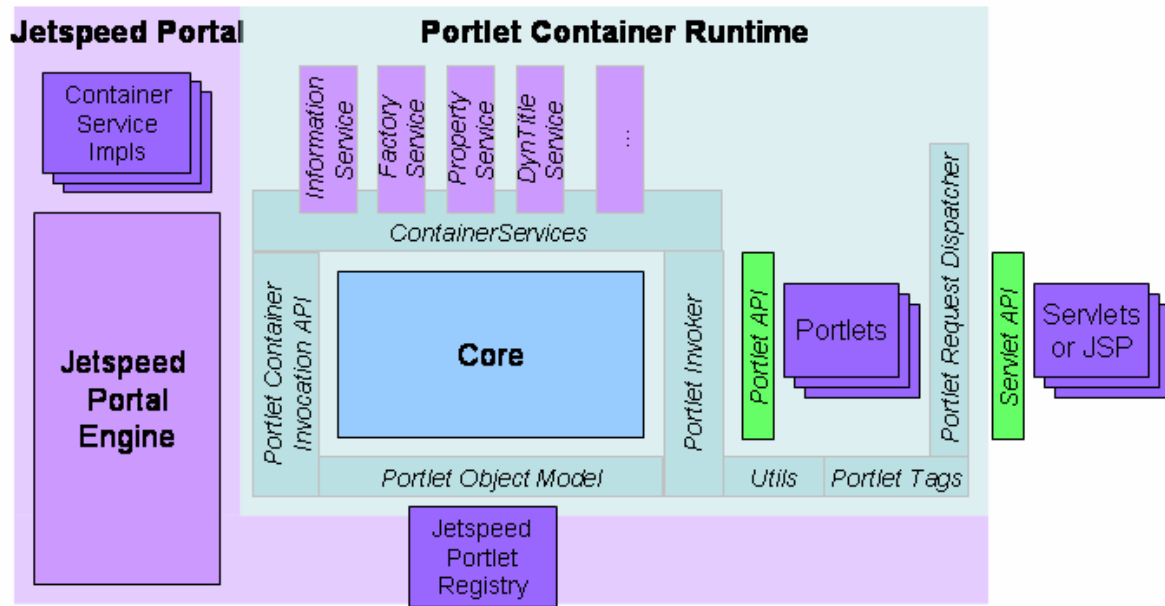


Figure 8.1: Jetspeed-2 architecture with the component concept of a portal engine, a portlet container and pluggable services.

The Jetspeed-2 architecture is modularized and not monolithic as the previous Jetspeed-1 architecture. Jetspeed-2 consists of a portal engine that is responsible for the markup aggregation and the portlet container who is responsible for running the portlets. If you've read Chapter 7 about Pluto before you may recognize that the portlet container pictures in Chapter 7 are very similar to the one shown here. This is due to the fact that Jetspeed-2 reuses the Pluto portlet container.

8.1.3 How does Jetspeed fulfill our portal requirements?

As you may remember we introduced in Section 6.2.1 of Chapter 6 a list of requirements that we think are the basic requirements that every portal used in production should fulfill. Let's see how Jetspeed-2 compares to our requirements list.

Table 8.1: Jetspeed-2 fulfilling the portal requirements

Requirement	Jetspeed-2
Display portlets	Yes, see samples
Deploy / remove portlets	Yes, see section 8.5
Customize portal layout	Yes, see section 12.3
Navigate	Yes, see section 12.4
National Language Support	Yes, full compliance with JSR-168 localization of resources and resource bundles (PLT 21.8 of the Portlet Spec)
Multiple markup support	Yes, see section 14.2 for examples of the RSS portlet supporting HTML and WML
Multiple device support	Yes, Jetspeed supports an array of client devices including hand-held devices and WAP phones
Skins and Themes	Yes, see section 6.4.2
Authentication	Yes, see section 6.3.2
User Management	Yes, see section 12.6
Personalization	Yes, see section 12.3
Authorization	Yes, using JAAS-based policies, see also section 12.6
Credential Store	Yes, Jetspeed provides an SSO (Single Sign-on Feature)
Datastore	Yes, using any JDBC compliant database
Remote Portlets	Yes, via WSRP4J, see Chapter 10
Administration GUI	Yes, see section 12.2
Logging and Tracing	Yes, using Log4J

Configuration Service

Yes, Jetspeed-2 provides a service extension to the portal container defined in a jetspeed-portlet.xml.descriptor file. Jetspeed-2 services run inside a Spring IOC container. See section 13.5.

Event Dispatcherq

Yes, via the Spring infrastructure

As you can see Jetspeed-2 is really a full-fledged portal that fulfills all our requirements. You still are not quite sure why use Jetspeed? We'll have some more arguments in favor for Jetspeed in the next section.

8.1.4 Why Jetspeed?

First: standards. Jetspeed-2 is conformant to the Java Portlet and WSRP standards providing a standard mechanism for the deployment and interoperability of portlets. Jetspeed also is based on J2EE standards for important framework building blocks such as security . With standards comes interoperability. You can take any Java portlet application and drop it into Jetspeed ready to use.

Second, Jetspeed is a high-quality open source project. Jetspeed has an established world-wide developer and user base. It is a mature open source project and is used by many major companies and organizations. With Jetspeed, you get the source code.

Third, Jetspeed is enterprise ready. Jetspeed-2, the second incarnation of Jetspeed, has matured to a more scalable architecture featuring Inversion of Control (IOC) components and a multi-threaded aggregation engine. Jetspeed's architecture follows the separation of concerns design pattern. It is foremost a portal server. It focuses on its job as a portal server, and it does not try to do anything else.

Fourth, Jetspeed-2 is founded on and leverages the functionality provided by many reputable open source projects such as Pluto, Spring, Lucene, MySQL, Velocity, and Xerces.

Finally, creating a portal with Jetspeed just makes sense. You have nothing to lose in trying it. It's open source and free of charge. Jetspeed has matured into a very easy to use portal. The administration of the portal is all done interactively using portlets and portlet applications. There is no need for complicated configuration properties. Just download, install and get started creating your portal immediately!

Now that we' ve convinced you that Jetspeed is of value for your portal project we' ll take a look at what you need to consider when developing portlets for Jetspeed.

8.2 Portlet Development with Jetspeed

As mentioned before, Jetspeed-1 pre-dates the Java Portlet Standard, and thus never supported the standard. This means that portlets programmed against the `org.apache.jetspeed.portlet.Portlet` class will only run on the Jetspeed-1 portal and on no other portal. In this book we will not cover the old Jetspeed-1 portlet API as there is a better solution for developing portlets on Jetspeed-1 available now.

In June 2004, the Jetspeed Fusion project was developed. Fusion enables Jetspeed-1 to run standard Java Portlet applications in Jetspeed-1. This was made possible by leveraging the component architecture in Jetspeed-2. The Jetspeed-2 portal engine is compliant with the Java Portlet Standard. Fusion embeds the Jetspeed-2 portal engine inside of the Jetspeed-1 portal. Inside Fusion and Jetspeed-2 the Pluto portlet container is leveraged in order to provide Fusion and Jetspeed-2 with a standard compliant portlet container. Pluto is covered in detail in Chapter 7.

The examples in this book will run inside either Jetspeed-2, a Fusion-enabled Jetspeed-1, or any other Java Portlet Standard compliant portal server. This gives you the option to try different portals and decide which one best fits your needs. While Jetspeed-1 is a more mature and stable project, Jetspeed-2 is the more active project, but also undergoing more development and change.

Jetspeed-2 has introduced the Bridges concept to provide portlet developers with additions to the basic portlet specification. Inside the Bridges sub-project there are many different Bridges that allow portlets to

send messages to other portlets (we’ ll cover this capability in more detail in Chapter 13.5) and integration with other web technologies, like Perl, PHP, JSF, Struts, and Velocity. We’ ll explain the Bridges framework in more detail in Chapter 11.

8.3 Installing the Jetspeed-1 Enterprise Portal

The Apache Portals Jetspeed-1 Enterprise portal is ready to use out of the box. It also provides a framework for developers to create portlet applications that are easy to build, extend, and maintain. Jetspeed-1 is already the portal of choice for both novice and experienced developers throughout the world.

In this section we’ ll show you how to install Jetspeed-1 and what you need in order to successfully run Jetspeed-1.

8.3.1 Prerequisites

Jetspeed runs on Linux, FreeBSD, Mac OS X, Windows NT or XP, or any other operating system supporting Java 1.4. There is one single prerequisite beyond that: a Java 1.4.2 (or higher) SDK. If you don’t have it on your computer, go to <http://java.sun.com/j2se/> to download the latest Java SDK. After installing the Java SDK, and ensuring that it’s running, continue on with the instructions below.

NOTE: All of the code provided in the examples is 100% Java.

8.3.2 Downloading the full enterprise portal in a box

Once you have the Java 1.4.2 SDK, you can download the full Jetspeed portal examples from the Manning site

To make development as easy as possible for you, we provide zip files available for download from <http://www.manning.com/portlets>. The zip files include:

- All the sample code from this chapter and all other chapters
- Ready to run example portals using Jetspeed-1 and Jetspeed-2

In order to build the examples, you will need to download Ant and Maven (assuming you have not yet installed Ant and Maven):

1. Install Ant (1.5 or greater) <http://ant.apache.org>
2. Install Maven (1.0 or greater) <http://maven.apache.org>

After installing Ant and Maven, you are now ready to build the examples. The examples also come ready to edit inside Eclipse.

If you want to you can also download Jetspeed-1 from <http://portals.apache.org/jetspeed-1/> and follow the install instructions mentioned there.

8.4 Using standard portlets with Jetspeed-1

As mentioned before Jetspeed-1 was created before there was a Java Portlet Standard. Thus Jetspeed-1 comes with a non-standard Portlet API out of the box. Therefore you are not able to run the portlet examples of this book on a plain Jetspeed-1 portal. Luckily there is some help available and you can plug the new, standards-

based Pluto portlet container that Jetspeed-2 uses into Jetspeed-1. We'll show you how to do this in the first part of this section.

Next we'll deploy some of our sample portlets in this upgraded Jetspeed-1 portal and add them to our portal page. First let's start by getting Jetspeed-1 up and running.

Starting up the Jetspeed-1 server is easy. From the root of the book's distribution directory, change to the "Examples" directory. From there, run either start-j1.bat or start-j1.sh that come with Jetspeed-1:

On Windows:

```
start-j1.bat
```

On Linux:

```
./start-j1.sh
```

Point your browser to

<http://localhost:8080/jetspeed/portal>

You will see the main Jetspeed page, as shown in figure 8.2.

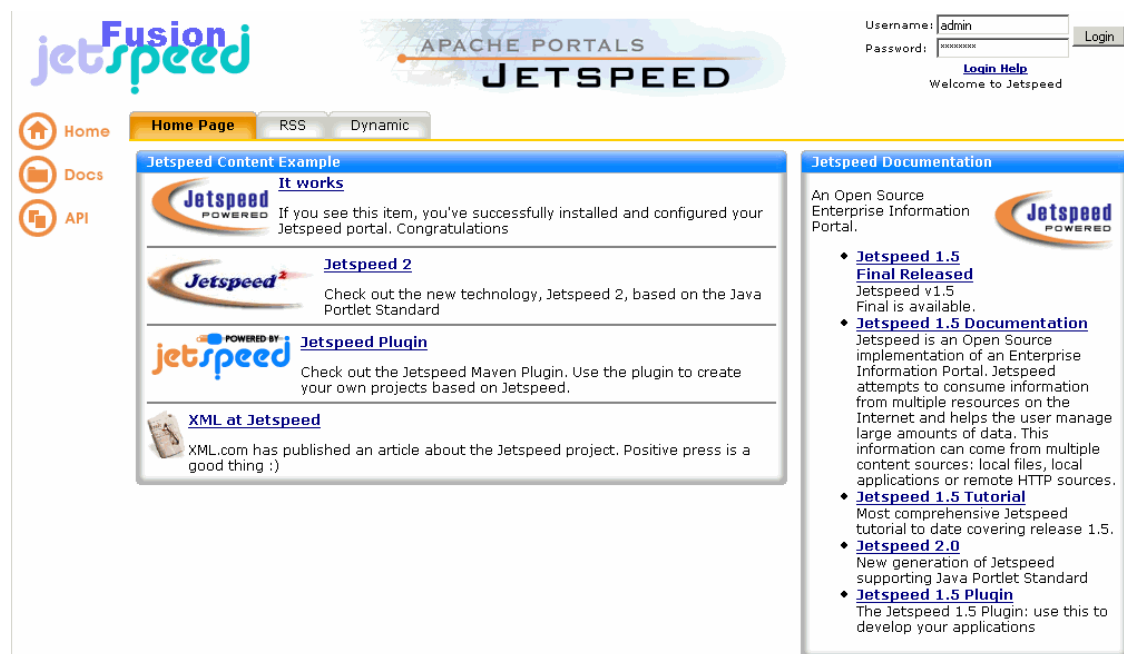


Figure 8.2 Welcome Page - Jakarta Jetspeed Portal Version 1.6 + Fusion

You are now ready to start developing and deploying portlets to Jetspeed-1!

8.4.1 Enabling Jetspeed-1 for JSR 168 portlets

After Jetspeed-2 started and introduced support for standard JSR 168 portlets, many Jetspeed-1 users also wanted to run JSR 168 portlets without needing to migrate to Jetspeed-2. With the need for JSR-168 support in mind for Jetspeed-1 users, the Fusion sub-project was founded.

As mentioned before, Jetspeed-1 was designed long before JSR 168 started to standardize the Java Portlet API. Thus Jetspeed-1 per default supports its own, Jetspeed specific portlet API. Fusion embeds the Jetspeed-2 engine and Pluto portlet container into Jetspeed-1. Any requests for JSR-168 portlets are routed to the embedded Jetspeed-2 engine. The final content of all portlets is still aggregated by the Jetspeed-1 engine.

You can even enable other non-Apache application servers, like JBoss, BEA WebLogic or IBM WebSphere Application server with Fusion to run JSR 168 portlets. More information on how to enable these application servers with Fusion can be found at <http://portals.apache.org/jetspeed-1/fusion.html> and <http://wiki.apache.org/portals/Jetspeed2/Fusion>.

If you used the Jetspeed bundle that comes pre-packaged with the book as described above, you can skip this section and move directly to section 8.4.2. You are all set for running JSR 168 portlets, as this distribution already includes the additional Fusion component needed to run JSR 168 portlets. If you want to setup your own Jetspeed-1 portal this section shows you how to enable Jetspeed-1 for JSR 168 portlets. Note that if you already use the new Jetspeed-2 you get JSR 168 support already built in.

Let's get back to enabling Jetspeed-1 with Fusion. Note that Fusion only works with the Jetspeed-1 1.6 version and not with earlier Jetspeed-1 versions. As Fusion is part of the Jetspeed-2 code stream you first need to download and build Jetspeed-2. Next step is to set

```
org.apache.jetspeed.fusion=true
```

in your \${HOME}/build.properties file. Now you can rebuild Jetspeed-1 with

```
maven -Dmaven.test.skip=true clean war
maven -o deploy
goto: http://localhost:8080/jetspeed/portal
```

As Fusion comes with its own database for storing information you need to start this database in addition to the Jetspeed-1 database. For this you need to go to the Fusion source directory and run

```
db.fusion.start
```

Now you are all set and can deploy JSR 168 portlet applications on your Jetspeed-1 portal. You can install JSR 168 portlet applications by simply dropping them into the \${TOMCAT_HOME}/webapps/jetspeed/portal/WEB-INF/deploy directory. The portlets in these portlet applications will automatically show up in the "Add Portlet" browser in the Jetspeed 1.6 customizer.

8.4.2 Deploying a portlet application to Jetspeed-1

Deploying a portlet application to Jetspeed is very easy. All you need to do is drop the portlet application archive (WAR) file into Jetspeed's deploy directory, and Jetspeed handles the rest automatically. This feature is called "Auto Deployment."

This distribution contains all the pre-built portlet applications ready to be deployed. All of the portlet applications for this book are already deployed to the Tomcat application server. When you are ready to deploy a new portlet application to the server, simply paste the WAR file into Jetspeed-1's Auto Deployment directory located under ./Deployment/jetspeed-1/webapps/jetspeed/WEB-INF/deploy/ as shown in Figure 8.3.

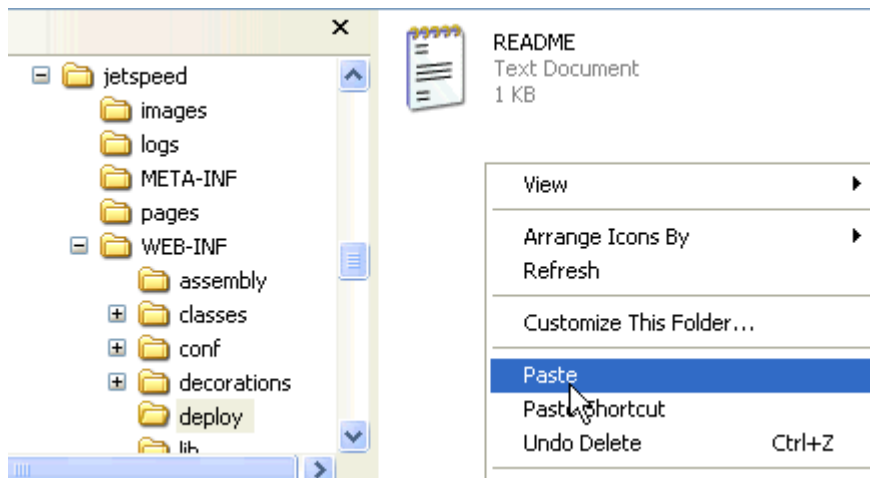


Figure 8.3 Copy EPBookPortletApp.war from the portlet-apps directory into the deploy directory

Jetspeed will automatically deploy the portlet application within 10 seconds or so and from then on you will be able to use the portlet application and can put the portlets in your portlet application on pages. We'll show you how to do this in the next section.

8.4.4 Adding portlets to your home page with the Jetspeed-1 page customizer

After deploying a new portlet application in the previous section you will want to see the portlets on one of your portal pages. In this section we will show you how you can leverage the Jetspeed-1 page customizer in order to add the new portlet on a portal page.

To be able to change the page content we need to authenticate ourselves to the portal. For this example, we login to Jetspeed with the credentials:

```
Username: turbine
Password: turbine
```

Click "Customize HTML" to bring up the Jetspeed Page Customizer:

1. To make it easier to view your new portlets, let's create a new tab for your home page. Click Add Pane, and then give that pane a name, for example: "Chapter8", and then press Apply:
2. Next, we will customize the Chapter8 pane. Click the Chapter8 link, and click Add Portlet:
3. Clicking the Add Portlet button brings us to the Portlet Browser (see Figure 8.5):

Category	All Portlets	Parent	All Parents
Add Title		Description	
<input type="checkbox"/> Apache News from Apache Week		The essential resource for anyone running an Apache server or any Apache based services	
<input type="checkbox"/> Apacheweek		Description not available	
<input type="checkbox"/> Barrapunto		Slashdot clon in Spanish	
<input type="checkbox"/> BASIC Authentication IFrame Portlet		Display secured URL within IFrame after automatic authentication	
<input type="checkbox"/> BBC Front Page News		Description not available	
<input type="checkbox"/> BBC Technology News		Description not available	
<input type="checkbox"/> BBC UK News		Description not available	
<input type="checkbox"/> BBC World News		Description not available	
<input type="checkbox"/> Bookmark Portlet		Bookmark Portlet	
<input type="checkbox"/> Change language portlet		Change language portlet	
<input type="checkbox"/> DatabaseBrowserTest		Simple Test Database Browser Portlet Example	
<input type="checkbox"/> DatabaseBrowserTest with parameters		Simple Test Database Browser Portlet Example	
<input type="checkbox"/> Delay-rendered Stock Portfolio		Delay-rendered Stock Portfolio Portlet	
<input type="checkbox"/> Different URLs example		Example of clipping from different URLs	
<input type="checkbox"/> Email		Send Email	

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [H](#) [I](#) [J](#) [M](#) [N](#) [O](#) [P](#) [R](#) [S](#) [T](#) [W](#) [X](#)

Figure 8.4 The Jetspeed Portlet Browser displaying all available portlets that can be put on a page

4. The Portlet Browser allows you to search for portlets to place on your home page. Let's filter the browser to only see portlets that are compliant with the JSR 168 Portlet Standard. Select the category of JSR 168 portlets in the upper left.
5. And now we only see JSR 168 compliant portlets. In this case, it's only the portlets provided from this chapter. Select one of those portlets:
6. Finally, press Apply, Apply again, then finally Save and Apply. You will be back to your home page:

You now have a new tab menu at the top of your home page. Click on it and you will see the content of the portlet that you selected in the Portlet Browser. That's it! You have now deployed a portlet application, and added a portlet to your home page!

In Figure 8.6 you will see the Bookmark Portlet from our Chapter 3 sample portlet application now running on the JSR 168 enabled Jetspeed-1 portal.

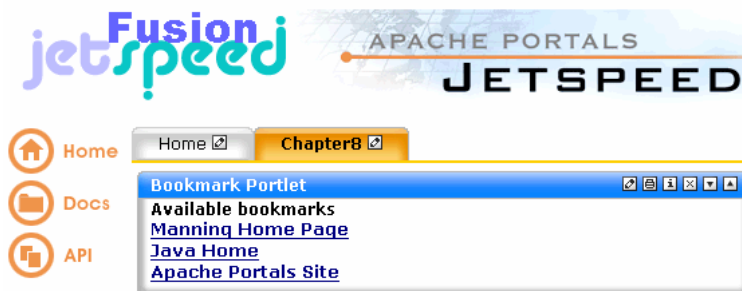


Figure 8.5 JSR 168 Bookmark Portlet in View Mode running on the Jetspeed-1 portal with the Fusion extension for JSR 168 portlets.

8.4.5 Portlet customization

In addition to customizing the page, Jetspeed supports customization of a specific portlet window. Customization of a portlet correlates with the Portlet API's Edit mode. To go into edit mode, press the Edit Portlet (customize) button (Figure 8.7):



Figure 8.6 The Edit Portlet (Customize) Button

From here you can manage your bookmarks, adding and deleting bookmarks to customize, or edit your portlet window.

This concludes a quick demo of deploying a portlet application to the Jetspeed Enterprise Portal. In chapter 12, we will further explore the features of the Jetspeed portal.

8.5 Getting started with the new Jetspeed-2

After we've covered how you can enable the Jetspeed-1 portal to run JSR 168 portlets let's take a look at the new Jetspeed-2 portal that has this capability already built in. If you start a new project, your choice should be Jetspeed-2 as this is the new, extensible and more standards based version of Jetspeed. In this section we'll explain how to run Jetspeed-2 and how to deploy a new portlet application and put the portlet on a page. Let's start with running the Jetspeed-2 portal.

To start the Jetspeed-2 server, from the root of the book's distribution directory, change to the "Examples" Directory. From there, run either start-j2.bat or start-j2.sh:

On Windows:

```
start-j2.bat
```

On Linux:

```
./start-j2.sh
```

Point your browser to

<http://localhost:8080/jetspeed/portal.>

You will see the main Jetspeed page as shown in Figure 8.7 with the welcome page.



Figure 8.7 Welcome Page - Jakarta Jetspeed Portal Version 2.0

You are now ready to start deploying portlets to Jetspeed-2!

8.5.1 Deploying a portlet application to Jetspeed-2

Deploying to Jetspeed-2 is as simple as copying your portlet application WAR files into a 'deploy' directory under Jetspeed-2. The default location for deploying your WAR files is the directory: `./Deployment/jetspeed-2/webapps/jetspeed/WEB-INF/deploy/`. Give the portlet application approximately 10 seconds to deploy to the portal.

For our examples, we have already pre-deployed the book examples to Jetspeed-2. So all you have to do is start up Jetspeed-2 using the provided scripts. Then click on the "Manning Portals Book" menu option on the left menu of Jetspeed-2, as in Figure 8.9.



Figure 8.8 Start the examples in this book by simply loading them from the Jetspeed menu.

You will then see all of the chapter example pages for this book. Select the “Manning Enterprise Portals Book Example, Chapter 8” tab at the top of page to see most of the sample portlets created in this book.

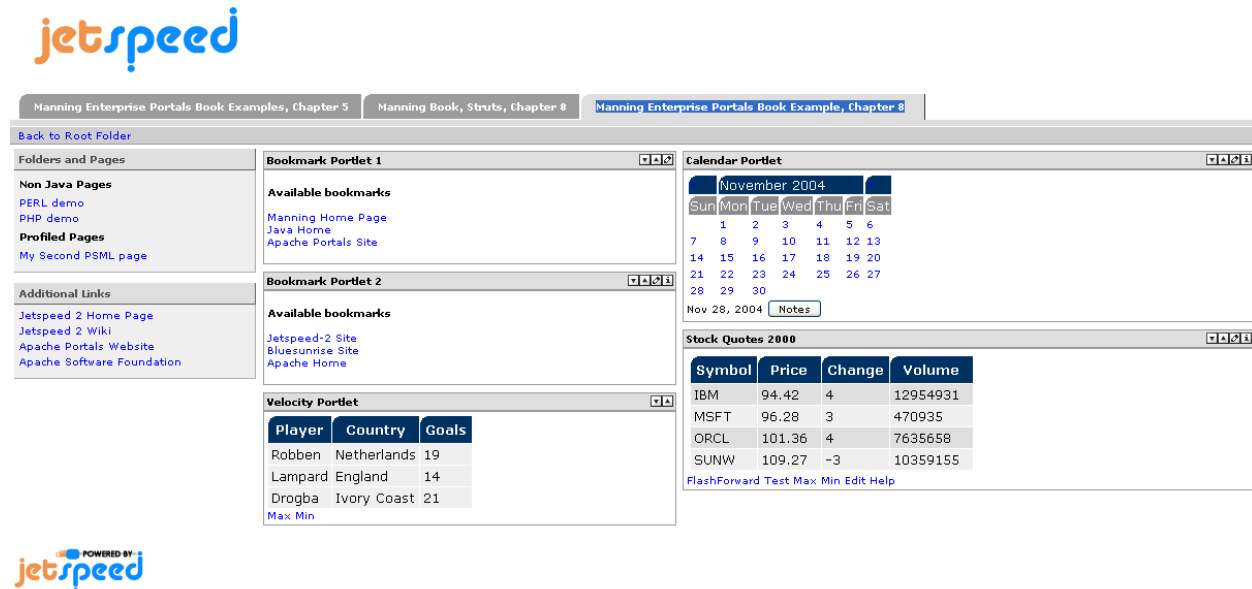


Figure 8.9 Some portlet examples from this book running on Jetspeed-2.

Until now we’ve provided for you the creation of the WAR files for the portlet applications. Now let’s see how you can build and develop your own portlet applications on Jetspeed-2. We’ll use the example portlet applications provided with this book to show you how to do this.

8.6 Building the book’s portlet samples with Maven

Until now we have just run pre-packaged portlets in Jetspeed, but how would you compile and package the samples of this book yourself? This will become important to you when you start to play around with the portlet code we provided in this book and change things in the portlets. You then need to be able to compile the changed samples and package the new code into a portlet application that you can deploy again on the Jetspeed portal.

For this book we’ve used the well known Maven tool, which is also an Apache project (<http://maven.apache.org/>) and greatly simplifies building projects. Maven is a Java project management and build tool. Maven is based on the concept of a project object model (POM). With Maven, you define all of your project dependencies, and Maven figures out how to build your Java project. Maven has an extensible model for adding features to the Maven engine. Maven comes with a large set of default plugins for common tasks such as compiling Java projects. Ant is built into Maven. Anything you can do in Ant can be done in Maven. In this section we are just touching upon the abilities of Maven and leverage only a small subset of Maven’s capabilities. Let’s start with using Maven for building our portlet projects.

After you’ve downloaded Jetspeed with all the examples in the beginning of this chapter from the Manning server you can find the code of all samples on the book’s distribution under the *Examples* directory. Building all sample portlet applications is easy:

```
cd Examples
maven war
```

This one command is all that is needed to build all the book's portlet applications. The WAR files of the portlet applications are written to the target sub-directory, to files named *chapter*.war*. Now you can go ahead and deploy the changed examples in the Jetspeed portal again and run your version of the samples.

8.7 Summary

This chapter provided an introduction to Jetspeed-1 and Jetspeed-2 and explained the differences between the two.

After learning that Jetspeed-2 is the future and the more advanced portal we spend some time for users of the more mature Jetspeed-1 to show you how to enable Jetspeed-1 to run JSR 168 portlets with the help of the Fusion extension. We then explained how to deploy portlets on Jetspeed-1 and how to put them on our portal page.

After covering Jetspeed-1 we switched to Jetspeed-2 and explained how you can deploy and run your portlet applications on Jetspeed-2.

Finally, we explained how you can modify the samples of this book, recompile them and package them again into portlet applications. You can then re-deploy the changed samples on Jetspeed again. We did this compiling and packaging by using the Apache Maven tool.

This chapter is the base for the more complex portal scenario we will create in Part IV and V of this book.

Chapter 9 Working with WSRP

In the previous chapters we have shown the basic building blocks of portal architectures and described portlets as locally pluggable components that provide the presentation layer of applications and information systems. Portlets typically render their content based on information retrieved from various sources like databases, transaction systems, content management systems, remote web sites or even web services. In addition to pure content presentation, portlets usually include interactive applications like email, calendar functions, booking services, etc.

Running portlets as local components on a portal puts a burden on portal vendors and organizations running portals for two reasons. Firstly, the presentation layer needs to be programmed for each service aggregated by the portal. Secondly, special adapters need to be written that enable communication between application or content providers and the various interfaces and protocols they may be using.

Imagine a weather service that provides weather forecasts and statistics for a given zip code. Portals that want to service their users with personalized weather information need to include portlets that provide a user interface specific to that weather service and additionally need to include code that communicates with that particular remote service and retrieves the desired data. As you can see, this approach is feasible for enabling base functionality of a portal with a limited set of applications and content. But when it comes to dynamic integration of business applications and information sources into portals it is not well suited.

This is where the Web Services for Remote Portlets (WSRP) standard comes into play. In contrast to data-oriented web services, WSRP moves the presentation layer to the service provider side. The WSRP standard defines user-facing, presentation-oriented and interactive web services offering a common set of interfaces. The protocol describes the generation of markup fragments suitable for aggregation into portals as well as the handling of user interactions with the markup.

In this chapter, we'll show you how WSRP enables web services to be easily pluggable into all WSRP compliant portals without requiring any service specific adapters. Your portals, therefore, can use a single, service-independent adapter to integrate and consume any WSRP conformant service. This approach greatly enlarges the marketplace for portals. Millions of interactive user-facing web services coming from thousands of application and content service providers can be integrated into portals with no more effort than just a few mouse clicks.

The WSRP standard is based on the web services architecture. It builds on the existing web service standards and, in fact, will leverage upcoming standards in the web services area like Web Services Security. Before we dig into the WSRP world, let's first take a look at the web services stack and the Service Oriented Architecture (SOA) to understand how it serves as the basis for remote portlets.

Understanding web services

Remote portlets are not running on the local portal server, as the name suggests. Instead they are running remote on another machine. Some protocol is needed to allow the portal to talk to this remote portlet, and the most interoperable and technology-independent solution is the web service technology. To better understand the WSRP remote portlet standard, we'll first take a look at its foundation, web services.

The term web services was coined during the Internet and Web boom. Every company wanted to have a presence on the Web and base their offerings on common Web technology. But web services are more than a marketing buzz word.

There is the technical aspect, too. When thinking of the Web we usually think of HTTP. Most web services will leverage HTTP as the transport mechanism. Is this because HTTP is, in a technical sense, the best transport option? No, it's far from it. But HTTP is well-known and widely accepted. Companies feel so comfortable using HTTP that they often leave firewalls open to HTTP communication. This allows web services to be delivered into every organization today without the need of reinventing and bringing up new IT infrastructures. Everything we need is already there, at least from the transport point of view.

What about services? Think of Application Service Providers (ASPs). Rather than offering products and solutions that are deployed directly within the customer's infrastructure, ASPs typically offer services. For example, imagine a credit card company that allows re-sellers to check the credit card information and limits of their customers. A flower shop's cash register application can talk directly to the credit card company's credit card checking application directly over the network.

Basically we are describing business-to-business communication (B2B) here. B2B in fact means application-to-application communication. Business-to-business has been around for years, so what's new about web services? Web services bring the service and transport pieces together. They enable application-to-application communication over well known Internet technologies.

That said, we can simply define a web service as follows:

A web service is any piece of code that communicates with other pieces of code over the Internet.

While this definition is enough to underscore the concept of web services, the truth is more complicated. To really enable application-to-application communication, we need to make sure that the technologies we use are platform-, language- and vendor-neutral. Programs written in Java, C++ or COBOL need to communicate with each other. No matter what platform an organization uses, be it J2EE, .NET or any other architecture you want to think of, we need interoperability.

Web services promise to solve the interoperability problem. Using web services, software can be truly modularized even across networks. Application developers can concentrate on solutions relative to their business rather than thinking about the technical details of how to communicate and exchange data with various business partners. But web services are not a single masterpiece. Indeed web services are an amalgamation of several standard technologies.

In the next section we will discuss the Service Oriented Architecture (SOA) that everyone is talking about these days and show how web services can help to implement that architecture. WSRP will play a very important role in the future deployments of SOA as at some point in time most services need to interact with an end user. WSRP provides the User Interface (UI) part of SOA.

Understanding Service Oriented Architecture

In the previous section we learned how to access remote services as web services. Now it is time to look at the other end of the spectrum and get a feeling why WSRP is such an important standard that will multiply the value of the portlets you may already have written. WSRP can play a leading role in providing an UI for your SOA-based applications. We'll learn in this section what SOA is all about and which problems it helps to solve.

Over decades, software complexity has been increasing. Integration of applications has always been a key requirement, but while complexity continues to increase, current systems and architectures seem to have reached their limits in terms of handling the integration challenge.

Problem 1: integration—When looking at current IT infrastructures and enterprise applications, we face a vast amount of different middleware, languages, operating systems, data stores and hardware that is integrated by countless connectivity products. In essence, we have many “isle solutions” that have been “bridged” with custom, proprietary software.

Problem 2: Reuse versus redevelopment of legacy applications—The Internet revolution brought new chances and challenges for businesses. New business partners and customers are available and ready for new business. But on the other hand, while the traditional needs remain, businesses need to respond to the new requirements even more quickly and cost effectively. Not only must applications between business partners be integrated, but a business must also provide its own legacy applications to its partners. Reuse of legacy systems is mandatory since re-development is a cost-killer.

Problem 3: Reuse of business logic across multiple applications—Another key point is reuse of software. See in your mind's eye a re-seller company which orders goods from manufacturers and sells them to customers. An internal ordering system might implement a `verifyZipCode` function when a new manufacturer is added to the system. Yet another, separately developed sales application running on a laptop might also implement the same function when entering new customer data. Now a new project is launched, allowing customers to order goods via a voice service. Guess which function will be implemented when the new application requires users to pass their address and zip code? This type of reuse often occurs when isolated projects are launched, perhaps, by different divisions without the existence of an infrastructure that allows such fine grained reuse of application logic.

Problem 4: Scaling distribution of applications—When looking at problems within distributed systems, perhaps the most pressing is the $n*(n-1)$ problem. Dealing with n applications that need to be integrated requires that $n*(n-1)$ connections or interfaces be developed. Adding just another application adds $2*n$ new connections (we leave it to the reader's exercise to prove that). Figure 9.1 shows an example for the case $n=4$ and that each application has 3 interfaces to other applications.

In addition to the development, test and deployment of these $2*n$ new interfaces, each existing application must be touched and modified in order to support these new interfaces. And that incurs consequences and costs.

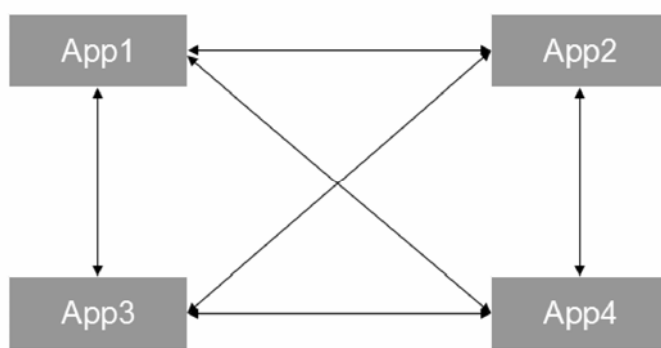


Figure 9.1 The $n*(n-1)$ scalability problem for remote applications for the case of $n=4$.

Given the problems described, we must develop systems where heterogeneous environments are a given. Most problems arise because we lack an architectural framework which allows us to quickly develop, integrate and reuse applications. Going one step further, we need a framework which allows us to assemble components and services for the quick and dynamic delivery of business solutions. Service Oriented Architecture (SOA) is being promoted in the industry as the next step in distributed software architecture which helps organizations and software developers to meet the rising challenges.

SOA describes an architecture where application logic is divided into individual, independent services. These services are described through well-defined interfaces. Any defined sequence of calls to the services can form a business process.

Basically a service can be a simple business function like `checkCreditCard`, or a business transaction like `debitAccountA-creditAccountB`. It can even be a system function like `getCurrentTime` or, you guessed it, `verifyZipCode`.

Services are treated as “black boxes.” This means that a service invoker does not know or even care about how a service accomplishes its task. The only things a user is interested in are the input parameters and the returned result.

For a user, it is also irrelevant whether a service is local or remote, which mechanisms, protocols and infrastructures are used to issue the call. All the user is interested in is the interface and the business value a service provides.

To support a Service Oriented Architecture we need some more ingredients. We need a means to publish services to a service directory where they can be easily found by service clients. But dumping services into a directory is not enough. Since there is no predefined community that knows about the semantics of the services, we need to capture both syntax (or better: interfaces) and semantics. Taxonomies are used to capture services. We also need to keep in mind that semantics must be understood by human beings as well. Using a service directory that contains detailed descriptions about services, like interfaces, protocols, and transport mechanisms, allows clients to dynamically bind to services and use them.

Suppose your portal wants to provide personalized weather information to users. Instead of maintaining its own weather application, the portal uses services to retrieve the information. Figure 9.2 illustrates how the portal does this by searching the registry (e.g. a UDDI directory) for services that implement the `getForecastForZipCode` interface described by taxonomy.

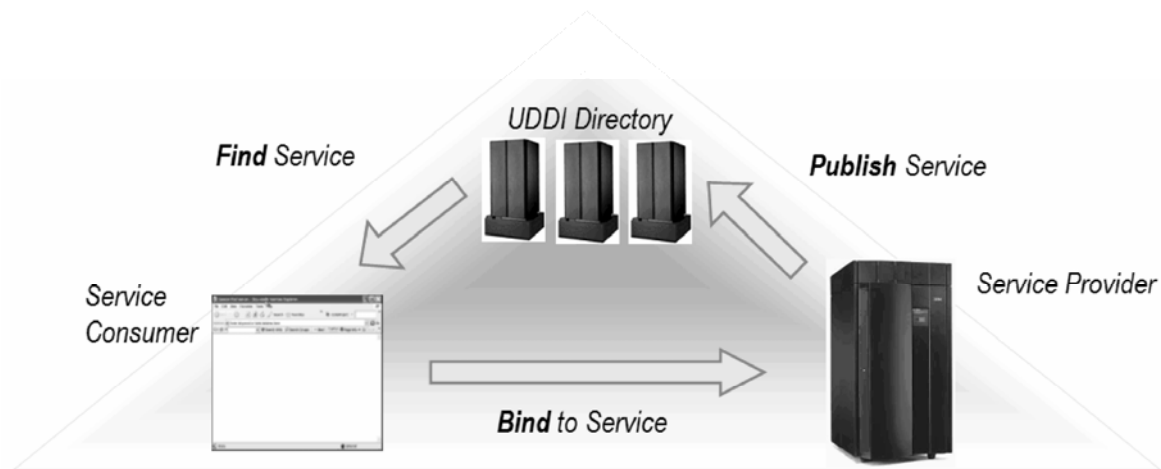


Figure 9.2 Publish/Find/Bind Paradigm where a service provider publishes a service the a directory, like UDDI, and a service consumer can find a specific service in this directory and bind to this service. After the binding the service consumer talks directly with the service provider.

When multiple services implement the `getForecastForZipCode` interface, the portal can arbitrarily choose a service, use the binding information from the registry, bind to the service, call the desired function and present the result to the end-user. As long as the requirements of the portal are met, it doesn't care where the service came from.

Now that we've learned what SOA is, let's try to relate it to what we've learned in the previous section about web services.

Web services = Service Oriented Architecture?

The short answer is no. Service Oriented Architecture is an architecture, that's it. SOA allows designing software systems that provide services to other applications. SOA does this through interfaces that get published in a directory and can be discovered by service consumers. The service consumers can then invoke these services dynamically over a network. Basically SOA is a way of thinking about building software out of small, reusable components, called services.

Web Services, as we discussed in section 9.1, are a set of technologies and standards like XML, SOAP, WSDL and UDDI which we explained briefly in chapter 2. Web services give us new options for building applications in heterogeneous environments with a more flexible and powerful programming model. Indeed current web service technologies prove that SOA is an architecture you can implement.

After this short dive into the foundation of remote services and the problems an architecture based on remote services can address, let's get back to our remote services: the remote portlets that can be used as UI components for a SOA-based system.

Introducing WSRP

After discussing remote services and the Service Oriented Architecture in the previous section we'll take a closer look at the Web Services for Remote Portlets standard in this section

The effort to standardize the remote portlet vision started in January 2002. A technical committee was founded at the OASIS consortium (Organization for the Advancement of Structured Information Standards). OASIS is a not-for-profit, international consortium that drives the development, convergence, and adoption of e-business standards. Founded in 1993, OASIS has more than 3,000 participants representing over 600 organizations and individual members in 100 countries. The consortium produces more Web services

standards than any other organization, along with standards for security and e-business. It also pushes standardization efforts in the public sector and for application-specific markets.

From the beginning, the Web Services for Remote Portlets (WSRP) standard had significant support, from the many vendors and interest groups that joined the WSRP technical committee. Nearly all major players in the portal industry participated and still participate in the standardization efforts. More information about the WSRP TC can be found in [1].

In September 2003 the WSRP vision was realized and the WSRP 1.0 standard was accepted by OASIS. The specification document can be found [in](#) [2]. Let's take a look at which problems WSRP aimed to solve and why this impacts you as portlet developer.

The WSRP Vision

WSRP extends the idea of Service Oriented Architecture and web services to the domain of portals and interactive user-facing applications. While web services introduce a means for integration of business logic via the Internet, the WSRP standard introduces the means for integration of presentation-oriented components. Using web services as the base, the presentation layer could be shared in a platform-, language- and vendor-neutral manner.

Ideally, WSRP would enable all content and application providers to offer their services in such a way that their services could easily be discovered and plugged into all standard compliant applications without any extra programming efforts on the client's side. (See Figure 9.3.) This vision doesn't limit the clients to just being portals, although it is expected that portals will become the main contributors here.

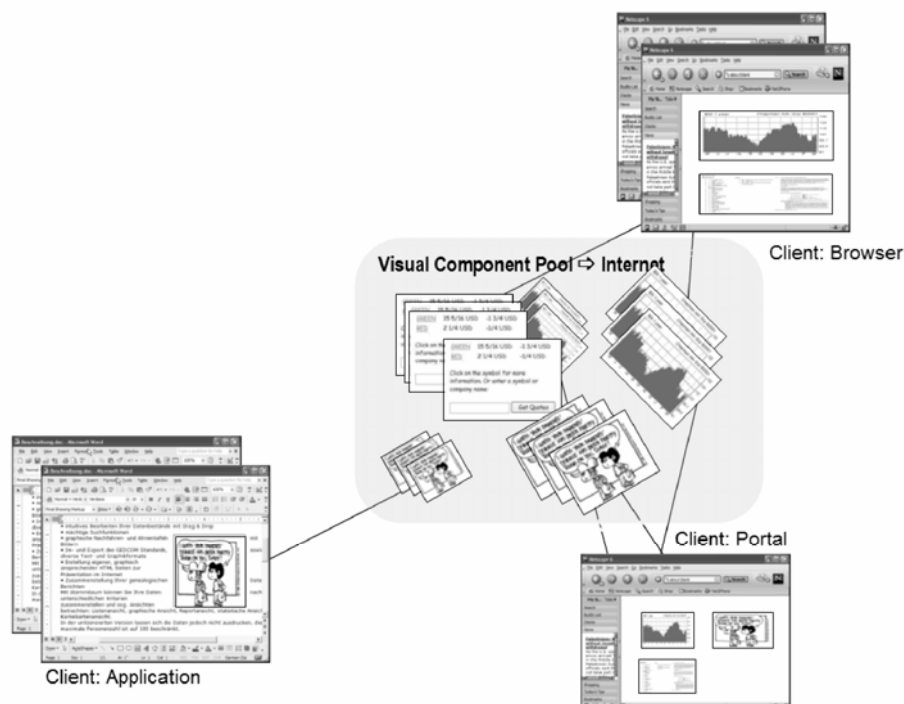


Figure 9.3 The WSRP Vision sees many types of services delivering information to the user without extra client-side programming

From a portal perspective, WSRP extends the model of local portlets being pluggable visual components to the World Wide Web. Instead of the need to deploy each application or portlet on each server that intends to use it, application sharing across network boundaries is advantageous and desired.

The advantages of such an approach are obvious. First, portal administration becomes easier and more effective. Administrators don't need to manage all the local deployments of pluggable components. They can find and integrate the WSRP services they need with just a few mouse clicks, typically by using their portal's administration user interface to browse a registry for available services, and selecting some for automatic integration into the portal. This eases the on-demand integration of content into portals. Users benefit by having more services made available to them in a timely manner.

Second, sharing of portlets and applications allows administrators to offload the servers and distribute resources and computing power across the network.

Third, it is often not desirable or even possible to have a local deployment and the infrastructure necessary for a particular application. One example here is a banking application accessing secured backend systems. Distributing just the presentation layer preserves the computing environment within the security domain of the application provider, while users can interact with the shared user interface.

This distributed approach has also advantages for content providers. Sharing the presentation layer instead of just plain application logic enables the content provider to remain in control of the way the content they intend to provide is presented to the end users.

In addition portals act as intermediaries between end users and WSRP services and aggregate services from many different content providers. They often offload significant traffic from content providers by caching content, enabling content providers to serve a huge number of users with a lighter IT infrastructure.

Having one unified protocol and methodology to share interactive applications also gives the content providers access to a larger set of customers. Once a common standard is established and accepted, content and application providers need to concentrate on only one technology, allowing them to be pluggable into various products without the need to develop specific adapters for each target platform.

By providing the types of services that WSRP defines, content and application providers can leverage portals as multiplying intermediaries to reach a number of end users that they never could have reached otherwise.

With WSRP, we enable sharing of portlets and content over the Internet using a common interface and protocol and thus allow cross-vendor publishing and consuming independent of underlying languages, technologies and platforms. For example, an interactive application using the .NET framework running on MS Windows can be shared with a portal based on J2EE running on Linux.

WSRP Usage Scenarios

Through its vendor-, platform- and language-neutrality, WSRP enables us to cover various usage scenarios which are not tied just to portal environments. In general, WSRP allows any application which adheres to the WSRP contract and protocol to expose its presentation layer over the network and allow any standard conformant applications to aggregate the content provided by WSRP services.

In figure 9.4, you see how a stand-alone application, like MS Word, could be enabled to integrate live content via the Internet and enrich the end-user experience with this content. One example might be the display of up-to-date stock quote charts within a document.

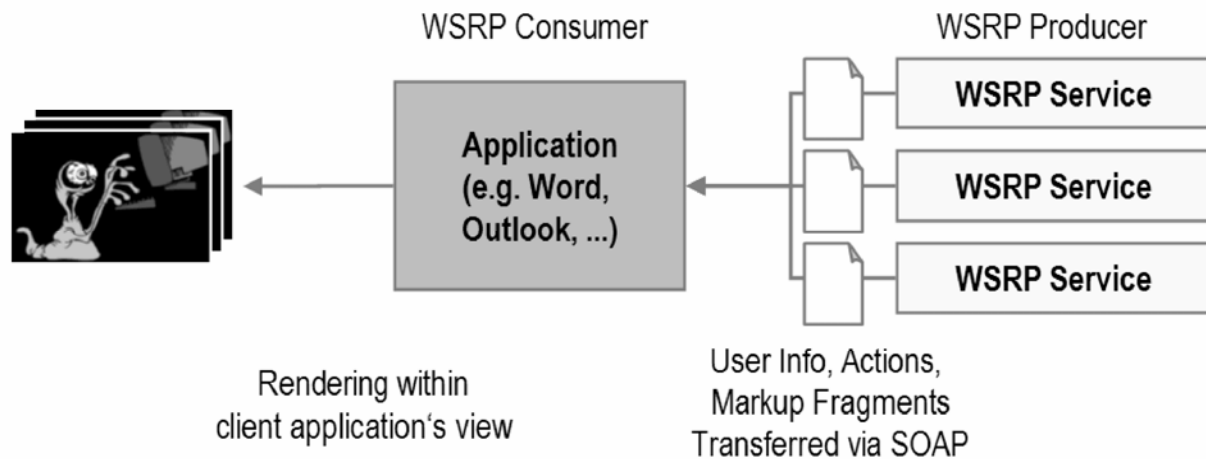


Figure 9.4 Applications consuming WSRP services

Of course, as the standard's name indicates, WSRP has a close relation to portals and portlets. Portals can benefit from WSRP in two ways.

Way 1: Consuming WSRP services

One major problem all portal environments deal with is the integration of content. The intent of portals is to provide a vast amount of information and data to the end-user aggregated from various sources. Typically special adapters need to be developed to access such remote services. In addition specialized user interfaces need to be developed for each integrated remote service. Imagine a portal integrating weather information from a weather service provider and stock quotes from an online broker service. To make these services accessible and usable by its portal users, the portal vendor needs to develop separate portlets providing the user interface for each service along with code that actually communicates to the remote services and retrieves the desired data to be displayed. While this sounds feasible for a small installation with a limited set of functionality, this approach is limiting for large portal environments, causing immense development costs and preventing the dynamic integration of content.

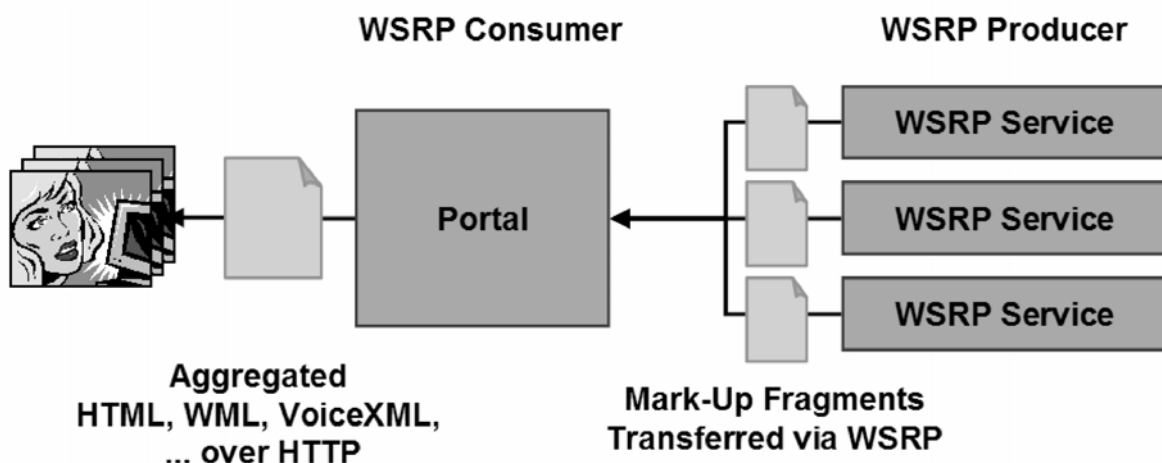


Figure 9.5 Portals consuming WSRP Services

With WSRP, remote services can provide the user interface along with the data and thus can be easily plugged into the portal environment with no extra coding efforts, as in Figure 9.5. WSRP optionally allows the services to be aware of the portal context. The consuming portal can provide information about the end user profile, the desired languages, markup types and the end user's device type. For example, the portal could pass

the end user's zip code and the fact that the user accesses the portal using a handheld device to the weather service. In that case the weather service could display current weather information relative to the zip code and a small weather map suitable for such handheld devices.

Way 2: Sharing portlets by being a WSRP producer

The approach of having only locally deployable portlets can also raise administration costs. Each portal that wants to provide certain portlets to its end-users needs to deploy them locally along with the administration overhead like access rights management, setup of the application's environment and maintenance costs.

Let us consider the following scenario. A company is running two portals. The employee portal provides information for all employees. The finance and controlling department portal allows only department employees to access applications related to them. The finance and controlling department decides to provide a service to all company employees estimating the variable pay for each employee based on the company's current level of success and the employee's ID. In a local-only environment this would require the employee portal administrators to locally install a portlet which accesses the remote finance and controlling service together with its underlying environment like database access, firewall setup etc. The portlet also needs to be maintained concurrently to the remote service. For example some database tables could be changed on the service side, which would require an update and redeployment of the local portlet. This is cumbersome and not cost effective.

It would be more convenient if the finance and controlling portal would share the portlet displaying the variable pay for each employee with the employee portal. This way the maintenance and setup remains in the domain of the finance and controlling portal. Updates and changes would be transparent to the employee portal since it only consumes the presentation logic provided by the remote portal.

WSRP enables such a usage scenario and allows portals to share their portlets (Figure 9.6). This approach significantly can reduce administration and development costs.

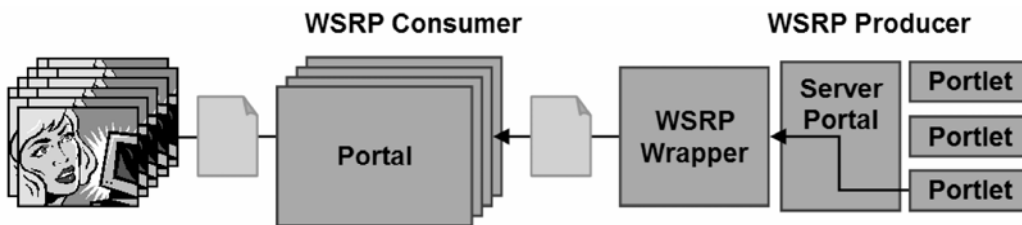


Figure 9.6 Portals sharing portlets

We will discuss how portals can be enabled to consume WSRP services and expose their locally deployed portlets as remote WSRP services in later sections.

After we have looked at the different usage scenarios of WSRP you may wonder why these scenarios cannot be implemented with today's web services. Why invent something new? We'll answer this question in the next section.

Data-Oriented vs. Presentation-Oriented Web Services

You might have asked yourself whether WSRP is reinventing web services from scratch and what the difference is to the well-known web services paradigm. The answer is simple. WSRP leverages the web services building blocks and indeed adds another layer to the web service stack. From a plain web services point of view, WSRP is an application layer protocol. The WSRP protocol uses web services as the base to enable applications to provide markup fragments and handle user interactions with that markup.

The main difference between WSRP and typical web services is how the presentation layer is handled. As you might know from the Client-Server-Paradigm a multi-tier application can be split in the presentation-, application- and data-layer. In today's typical web services scenario we speak about remote presentation. This means that the presentation layer is handled completely by the client side while the application logic and the data layer reside on the server side.

WSRP splits the presentation layer between the client and the server. Here we speak about distributed presentation. The distributed presentation approach is also well-known from the X-Window system. In a WSRP environment the service provides parts of the presentation logic which in turn is integrated in the client's presentation. Also interactions with the provided user interface fragments are forwarded to the service which handles these interactions.

Since WSRP enables this distributed presentation paradigm and leverages standard web services technologies to accomplish this, the term Presentation-oriented Web Services was found to distinguish WSRP services from common web services. Indeed WSRP can be seen as the presentation layer for web services.

Comparing approaches

Let's take a look at the differences between both approaches and see how interactive, user-facing applications can benefit from presentation-oriented web services. Imagine a service provider which provides data about stock quotes in the course of time to its customers. The customers may choose to obtain stock quotes data during the course of the day, month or year. Figure 9.7 shows the different approaches to this problem.

In the "traditional" web service usage scenario the web service might provide a web service operation called `getStockQuoteFor(Symbol, Period, Interval)` which takes the symbol of the stock, the period for which data is to be provided and the interval of the value snapshots as parameters. As a result the web service delivers an array of float values representing the quotes over the course of time. An interactive application which wants to provide a stock quote chart to their end-users needs to provide the complete presentation layer which enables the end-user to interact with the remote service. For example, it needs to render the chart based on the data obtained from the remote web service, needs to add navigational elements which allow the user to choose between the various periods and to choose stock symbols and also need handle failures and present them to the end-user accordingly.

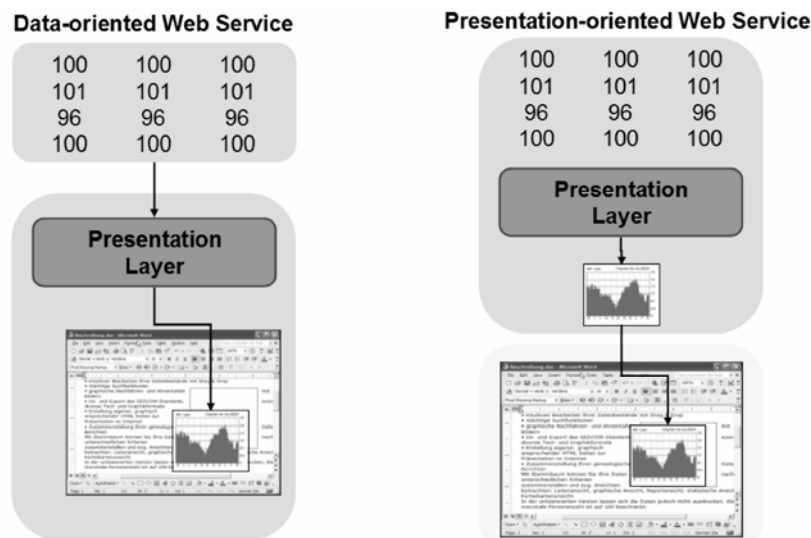


Figure 9.7 Data-oriented vs. Presentation-oriented Web Services

When using presentation-oriented web services a service provider would provide a WSRP service which offers a remote portlet called `StockQuoteChart`. This remote portlet doesn't just provide plain data which needs to

be interpreted and rendered by the client, but instead provides the presentation layer directly. This enables the client to integrate the remote service straightforwardly into its user interface without the need of developing any user interface parts specific to the remote service. Interaction with the user interface fragments of the web service are forwarded to the server side and handled there. Results of the interaction are then again provided as user interface fragments back to the client.

The advantages of presentation-oriented web services become even more evident when we take a closer look at the use of web services into the domain of portals.

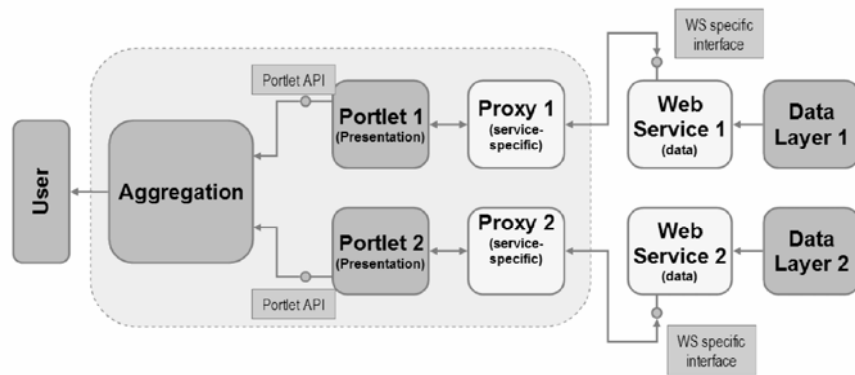


Figure 9.8 Usage of data-oriented Web Services in Portals

Figure 9.8 shows a simplified picture of a typical portal architecture. Usually the aggregation engine provides the portal user interface to the end user. The actual content is delivered by the portlets which are called by the aggregation engine using the local portlet API and deliver markup fragments.

Integrating a remote service into a portal

Let's revisit the stock quote example and take a look at the steps necessary to integrate a remote service into a typical portal. First a local portlet needs to be developed which generates the user interface specific to the stock quote service. For example, in rendering the stock quote chart, the portlet should have UI elements allowing the user to choose between the various time periods to display, set the intervals to check on the quote, and choose a stock symbol. Second, to be able to obtain the data from the remote service the local portlet needs to include code which invokes the remote service operation. Usually this piece of code is a service-specific proxy which offers a local API to the portlet and translates the local code to remote calls and handles communication between the service client and server. On the server the corresponding service stubs exist, which are called by the clients. These service stubs are the entry points into the service which finally executes the application logic and returns the results.

As you can easily imagine this approach is very expensive and prevents the easy integration of content and remote services into portals. Each single remote service that is going to be used requires service-specific code to be developed. Once things change on the remote side the local portlets need to be changed, too. For example, the stock quote service provider could add operations which allow service users to compare a stock to an index over the course of time. In this case the user interface provided by the special local portlet would need a re-design, code changes and re-deployment on the client portal side. Since the portal's main purpose and intent is to aggregate content and applications from various sources this additional effort and costs are multiplied.

The situation changes dramatically once presentation-oriented web services are used to integrate content and applications into portals. Because WSRP defines one protocol and contract common to all WSRP services, no service-specific code is required any more. WSRP defines web service operations which allow

clients to obtain markup fragments that can be aggregated into the portal's user interface and which allow passing information about the interactions with this markup back to the service.

Let's take a look at Figure 9.9 which shows the architecture of a portal consuming presentation-oriented web services. Here we can use one generic proxy portlet written against the local portlet API which can be used as a proxy to arbitrary remote services.

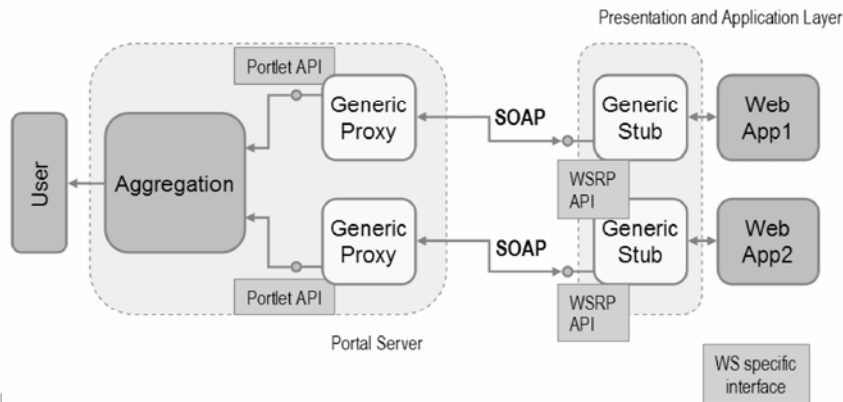


Figure 9.9 Usage of presentation-oriented Web Services in Portals

The code of the generic proxy needs to be developed and deployed once and is applicable for all remote WSRP services because of the unified protocol. Only the runtime configuration of each generic proxy portlet instance decides which actual remote web service is to be used. The generic proxy portlet translates the calls of the aggregation engine to remote WSRP calls and manages the communication with the remote service.

This way, portals can easily integrate remote services on the fly with no need of the deployment of service- specific code. Also changes to the application logic and feature enhancement are directly reflected in the aggregated portal user interface with no local changes.

Hopefully we've convinced you at this point the WSRP brings a lot of value to the portal landscape. In the next section we'll take a closer look under the hood of WSRP and the actors involved.

Working with the Actors in the WSRP World

We already used some terms to describe the parties playing a role in WSRP. As shown in figure 9.10, the WSRP specification basically defines a protocol between Producer and Consumers which interact on behalf of end-users.

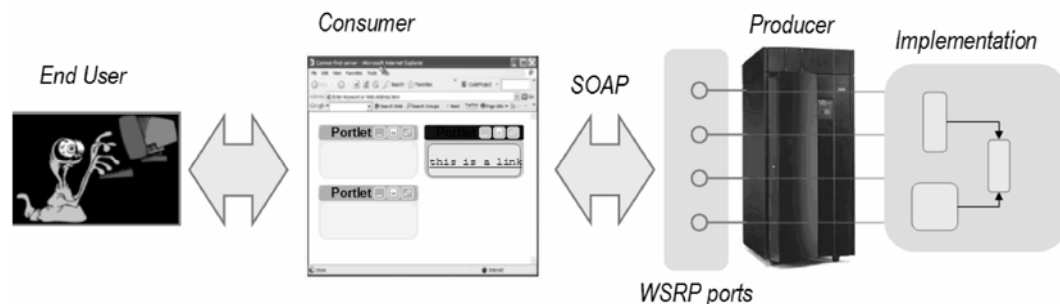


Figure 9.10: The WSRP specification defines how presentation-oriented web services and aggregators interact on behalf of end-users

The protocol mainly defines operations that allow the Consumer to retrieve markup from remote portlets, to communicate user interactions with that markup back to the portlets, and to manage the life cycle and configurations of remote portlets. Let's start with the role of the Producer who generates the markup.

Understanding Producers

Producers are presentation-oriented web services that provide entities (portlets) which are able to generate markup fragments and process user interaction requests. Producers are modeled as containers for portlets. They manage the persistent and transient lifecycles of portlets and sessions and handle the interactions triggered by the Consumer on the end-user's behalf.

Producers and Consumers communicate with each other using SOAP (Simple Object Access Protocol) providing a platform-, language- and vendor-neutral means for invocation of and interaction with remote services.

For the conversation between the Consumer and the Producer WSRP defines four web service portTypes (interfaces). Some of these interfaces are optional to enable different levels of sophistication of Producers and Consumers.

Table 9.1 WSRP portTypes

portType

Service Description Interface

Markup Interface

Registration Interface

Portlet Management Interface

Purpose

This interface is a mandatory interface. It allows the Consumer to obtain information about the capabilities and requirements of the Producer and about the portlets it hosts.

This interface is a mandatory interface. It is the most important interface and enables end user interaction with the Producer via the Consumer, while still staying embedded in the Consumer application.

This interface is optional. It allows the Consumer and Producer to establish a relationship and to scope all subsequent interactions to a specific registration.

This interface is optional. It allows the Consumer to access the life cycle of portlets (e.g. cloning and destroying portlets). In addition this interface contains the portlet property management which allows the Consumer to programmatically access and manipulate portions of the portlet's persistent state.

For each portType, a default HTTP binding is defined by the specification. Let's take a look at the actual Portlets now.

The role of the Portlet

Portlets in the WSRP sense are entities that render their state in markup and handle user interactions with that markup. Similar to JSR168, portlets include application logic as well as a particular configuration of any settings or properties the portlet provides.

Portlets can be viewed as resources within the Producer environment. To address a certain portlet a Portlet Handle is used. This handle is passed with each operation targeting a portlet at the Producer, e.g. operations from the Markup Interface and Portlet Management Interface.

The WSRP specification distinguishes between two notions of portlets. Producer Offered Portlets are portlets which are exposed by the Producer as available in the service description. These portlets are preconfigured and not modifiable by Consumers. For example a weather portlet might have a preconfigured setup of weather service providers from which to obtain data from and use it for display in the markup.

If the Portlet Management Interface is provided by the Producer, Consumers can clone the offered portlets and customize these cloned portlets according to their needs. Such a uniquely configured portlet is called a Consumer Configured Portlet. We'll explore Consumers in the next section.

Understanding Consumers

Consumers are intermediaries that communicate with Producers and integrate presentation-oriented web services into their applications. They aggregate the markup delivered by remote portlets and present the aggregated user interface to their end-users. Interactions with the markup flow via the Consumer back to the Producer and the targeted portlets.

Portals are typical intermediaries, because they aggregate content from various sources on their own pages and present them to their end-users. But a WSRP Consumer could be any kind of application that utilizes the WSRP protocol to obtain and interact with Producers and the exposed remote portlets.

As already mentioned the WSRP protocol specifies the interaction between the Consumer and the Producer. It remains quiet on the interaction between the end-user and the Consumer. In typical portal environments end-users use a web browser and the HTTP protocol to communicate with the Portal web application.

Interaction with the End-User

The end-user represents the person that comes to the Consumer's Web site to use the Producer's application in the Consumer's context. The end-user initiates interactions with the portlet's markup using the Consumer's environment.

In the case of a portal the end-user is the portal user that interacts with the portal page and the portlets on that page. In case one of these portlets is a WSRP portlet the portal user will become the WSRP end-user for this WSPR portlet.

Now that we have all involved roles and the overall architecture covered we'll take a look at some real WSRP interactions between a producer and a consumer in the next section.

Handling Basic WSRP Interactions

Without digging very deeply into the details of the WSRP protocol, let's step into the most important part of WSRP: generation of markup fragments and interaction with that markup. We skipped the preliminary administrative tasks like registering the Consumer with the Producer, obtaining information about the Producer and its portlets and adding a remote portlet on the Consumer's page.

Here we simply assume a user has put a remote portlet on his page on the Consumer portal and is ready for action! As briefly mentioned in section 9.3.3, one implementation choice could be for the Consumer portal to provide a generic proxy portlet which is configured to point to a remote portlet. This pre-configured proxy portlet is a local portlet which users can put on their pages and translates the local portlet API calls to remote WSRP calls.

In this section, we'll cover the different interactions an end user can perform and how they are reflected in WSRP. We'll start with displaying the initial portal page, then go to interactions of the end-user that changes the view or navigational state of the portlet and finally take a look at how end-user actions that are intended to change state at the producer are handled.

Initial Page Load

Let's start with the obvious first step. After adding the remote portlet to his page the user wants this page to be displayed. Typically, in the local case, the Consumer portal issues render requests on each portlet window to obtain the markup. Similarly, in the remote case, the Consumer issues a SOAP request to the WSRP Producer addressing the remote portlet. The operation used here is the `getMarkup()` operation defined in the WSRP Markup Interface.

The Producer receiving the `getMarkup()` SOAP request retrieves the portlet handle from the request addressing the target remote portlet and passes the request to that portlet. Note that the WSRP specification remains calm on how Producers and remote portlets are modeled. Any implementation is legal as long as it adheres to the WSRP protocol between the Consumer and Producer. If we considered the Producer being a Portal following the architectural principles described in Chapter 6, the request would flow through the (JSR168) portlet container to the target portlet, namely the portlet API `render()` method would be called on a portlet instance represented by the passed portlet handle.

Once the request is processed by the remote portlet, the Producer returns a SOAP response message called `getMarkupResponse` containing the generated markup fragments along with some metadata. This returned markup is then aggregated by the Consumer portal on the user's page. Figure 9.11 illustrates the message flow between the end-user, the Consumer and the Producer.

The returned markup fragments can contain URLs triggering further interactions with the portlet. Similar to JSR168, WSRP defines various URL types allowing to trigger further `getMarkup()` requests or actions. We will talk about URL generation in section 9.6. For now it is sufficient to understand that, depending on the URL type, a certain WSRP request is being issued.

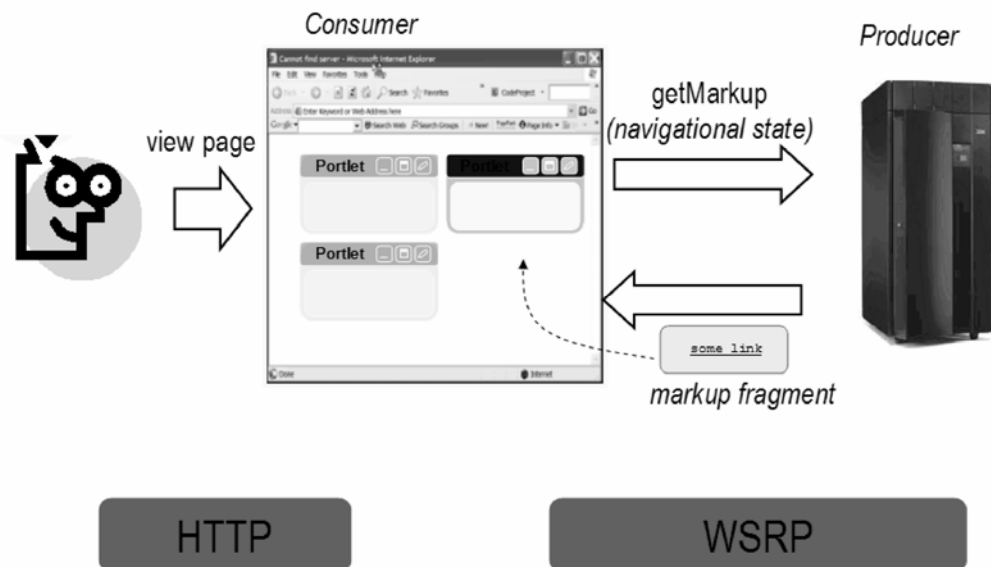


Figure 9.11: Initial Page Load

Let's dive down to the SOAP level and take a look at the SOAP messages flowing between the Consumer and Producer.

Requesting a remote portlet via SOAP

Let's start with the request:

9.1 Example `getMarkup` request

```
<?xml version="1.0" encoding="UTF-8"?>
  <soapenv:Envelope xmlns:soapenv=
    "http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Body>
      <getMarkup xmlns="urn:oasis:names:tc:wsrp:v1:types">
        <registrationContext>
```

getMarkup request top level element, contains all parameters of getMarkup request

registration context input parameter, represents relation between Consumer and Producer

```

    <registrationHandle>
      192.168.66.31_1088863464714_0
    </registrationHandle>
  </registrationContext>
  <portletContext>
    <portletHandle>
      0.192.168.66.31_1088863582013_1
    </portletHandle>
  </portletContext>
  <runtimeContext>
    <userAuthentication>wsrp:none</userAuthentication>
    <portletInstanceKey>
      ProxyTest_row1_col1_p1
    </portletInstanceKey>
    <namespacePrefix>
      Pluto_ProxyTest_row1_col1_p1_
    </namespacePrefix>
  </runtimeContext>
  <userContext>
    <userContextKey>
      dummyUserContextKey
    </userContextKey>
  </userContext>
  <markupParams>
    <secureClientCommunication>
      false
    </secureClientCommunication>
    <locales>en</locales>
    <locales>de</locales>
    <mimeType>text/html</mimeType>
    <mode>wsrp:view</mode>
    <windowState>wsrp:normal</windowState>
    <clientData>
      <userAgent>
        WSRP4J Proxy Portlet
      </userAgent>
    </clientData>
    <markupCharacterSets>
      UTF-8
    </markupCharacterSets>
    <validNewModes>
      wsrp:view
    </validNewModes>
    <validNewModes>
      wsrp:help
    </validNewModes>
    <validNewModes>
      wsrp:edit
    </validNewModes>
    <validNewWindowStates>
      wsrp:normal
    </validNewWindowStates>
    <validNewWindowStates>
      wsrp:maximized
    </validNewWindowStates>
  </markupParams>

```

← portlet context input parameter, addressing the portlet

← runtime context input parameter

← user context input parameter

← markup parameters input parameter

```

        <validNewWindowStates>
            wsrp:minimized
        </validNewWindowStates>
    </markupParams>
</getMarkup>
</soapenv:Body>
</soapenv:Envelope>

```

The SOAP Body contains one top level element; the getMarkup element. It holds further sub-elements representing the parameters. The first parameter is the registration context. The registration handle represents the relationship between the Consumer and the Producer which was set up during registration. The registration handle is assigned by the Producer identifying the Consumer. We will talk about registration in a later section.

The next important parameter is enclosed in the portlet context element. The mandatory sub-element is the portlet handle. Using this handle the Consumer is addressing the portlet targeted for the getMarkup() operation.

The runtime context provides transient information for one particular operation between the Consumer and Producer. Firstly the type of user authentication is passed. Here, the Consumer indicates to the Producer how the user on whose behalf the actual call is issued authenticated itself against the Producer. This information might be used by the portlet to influence the markup which is being generated. Secondly, the portlet instance key is an opaque value which allows the portlet to namespace itself in an optimal manner. A typical usage of the key is the multiple inclusion of the same portlet (identified by the portlet handle) on one Consumer's page. Thirdly, the namespace prefix is provided by the Consumer to allow the portlet to namespace elements in the markup accordingly. One example of such a namespacing is the usage of JavaScript functions in the markup. Since the function names need to be unique on the aggregated Consumer page, they need to be namespaced.

The user context element is used to provide information about the end-user. This structure typically contains user profile information which is aligned to the P3P user profile definition. Here the consumer can also assert user categories for the particular user which might influence the content being rendered in the markup.

The markup parameters contain information for the portlet needed to generate the markup. Here, the Consumer requests the desired locales, the markup character set as well as the mode and windows state for the target markup. Also the Consumer provides information about the valid new modes and windows states. This way the Consumer can restrict the portlet to generate only URLs within the markup which allow only transitions to particular modes and window states. For example, the Consumer may not want a certain user group to edit the portlet, and so not provide the edit mode. In this case the portlet must not generate markup which could contain URLs which switch to a forbidden mode. We will discuss modes and window states in a later section. In addition the Consumer also provides information about the end-user agent, i.e., the device the end-user is using to show the aggregated markup, and whether the communication channel between the Consumer and the end-user agent are secured.

Given this information the Portlet has all it needs to generate the markup fragment. Now, let's take a look at the response sent back to the Producer.

Responding to the Consumer

Listing 9.2 shows how the producer responds to getMarkup.

9.2 Example getMarkup response

```

<?xml version="1.0" encoding="utf-8"?>
  <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"

```

**getMarkup response top level element,
contains the message response** 35


```

xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <getMarkupResponse xmlns="urn:oasis:names:tc:wsrp:v1:types">
      <markupContext>
        <mimeType>text/html; charset=UTF-8</mimeType>
        <markupString>&lt;h2&gt;URL Types Test&lt;/h2&gt;Time:Sat Jul 03 [...</markupString>
        <locale>en</locale>
        <requiresUrlRewriting>true</requiresUrlRewriting>
      </markupContext>
    </getMarkupResponse>
  </soapenv:Body>
</soapenv:Envelope>

```

markup context, contains information related to generated markup

At the SOAP level the response is pretty similar to the request. The SOAP body contains just one top level element, the `getMarkupResponse`. The markup context contains all information related to the generated markup. It contains the mime type and the locale of the generated markup fragment as well as the fragment itself (contained in the `markupString` element; we've cut the markup for better readability here). Another important element is the `requiresUrlRewriting` flag. This flag indicates whether a post-processing is necessary on the Consumer side to rewrite the URLs within the markup fragment. This is necessary since the URLs need to point to the Consumer rather than to the portlet on the Producer – remember: the end-user is interacting with the Consumer not with the Producer. WSRP uses various URL types to trigger calls back to the Producer. You can straight forwardly compare this to render and action URLs in JSR168.

Let's take a look at an example portlet. Figure 9.12 shows the portlet containing the markup fragment delivered by the above `getMarkup()` call. Within the portlet decoration generated by the Consumer's aggregation engine we see the embedded markup fragment. It displays the date and time, the current mode and window state, as well as various links which trigger further interaction with the portlet. For example the "Next>>" link will trigger a new `getMarkup()` call, whereas the "Action" link will trigger a `performBlockingInteraction()` call. We'll come to that in the next sections.

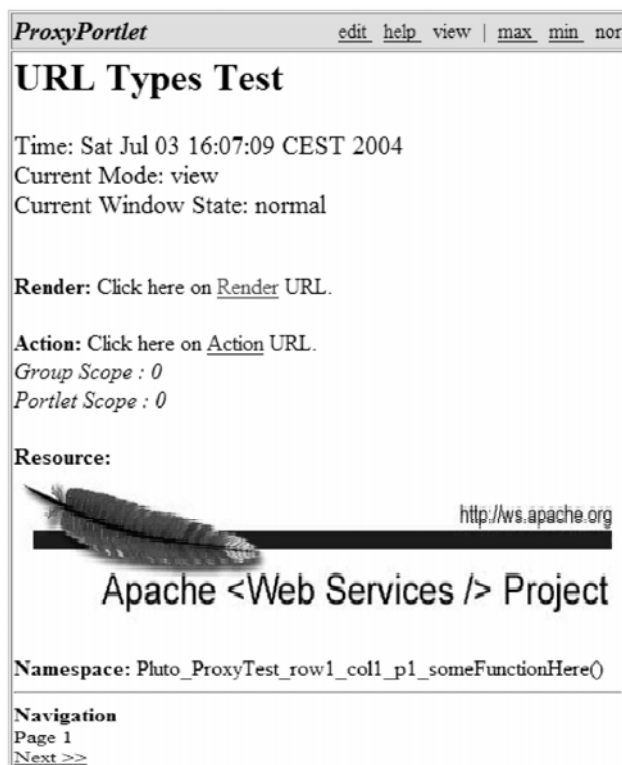


Figure 9.12: Initial view of an example portlet

Rendering a new navigational state

After we've received the initial markup fragment of the portlet we can interact with it. First, let's start with an interaction that does not manipulate the portlet's session state or persistent state. Rather, it manipulates the navigational state. The navigational state is used to render a portlet in different ways although the rendition usually is made according to the portlet's session or persistent state.

Imagine a portlet displaying weather information for a certain zip code. To enhance the information, the portlet could allow the end-user to switch between the current weather data, a three day forecast and a long term forecast. Typically the zip code would be stored persistently for the portlet (persistent state) and would apply to all three "views". In contrast the three "views" can be modeled as navigational states which only affect the rendering of the portlet; they don't actually change the state of the portlet, i.e., the markup fragments generated always render weather information for the given zip code.

Another example is the "next>>" link in our example portlet above. This link stores a navigational state that – when invoked – switches the view to another page.

So how does the navigational state flow from the Consumer to the Producer and the target portlet? As you will learn in section 9.6, the generated markup can contain various URL types. One of them is the render URL. Render URLs can contain the navigational along with other well-known URL parameters.

Let's take our example portlet in Figure 9.12. The "next>>" link contains "page2" as the new navigational state. Please note that the navigational state is opaque to the Consumer and may be encoded in a manner the portlet chooses, e.g. in base64 encoding. Once the user clicks on such a render URL the Consumer translates this interaction into a `getMarkup()` call and adds the navigational state from the invoked URL to the markup parameters. The portlet processing this request takes the new navigational state into account and renders the markup accordingly. In our example it renders page 2. Figure 9.13 illustrates that flow.

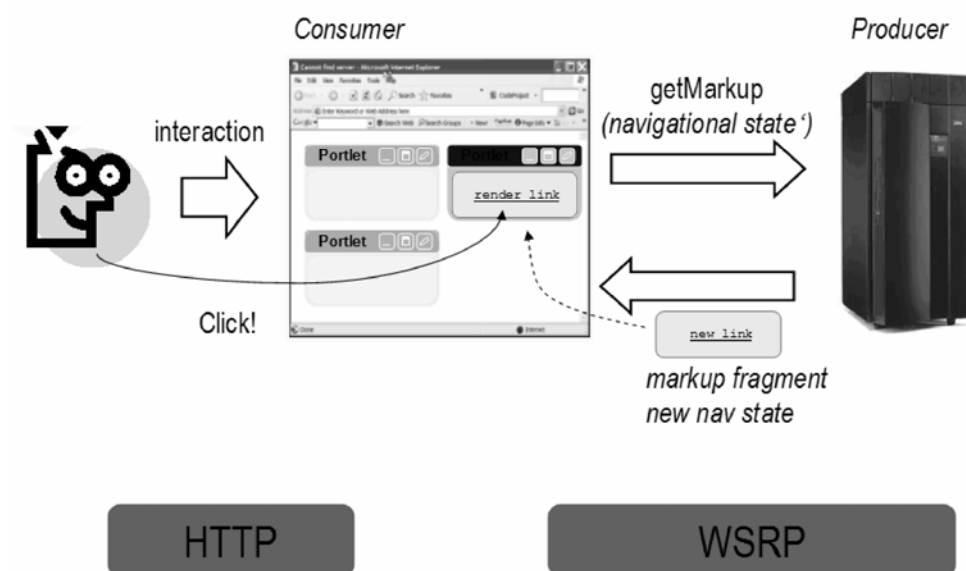


Figure 9.13 Render a new fragment after the user clicked on a link that changes the view or navigational state

The following listing shows fragments of the `getMarkup()` request on the SOAP level. Basically, request/response messages are the same as in the initial case. The only exception is the additional navigation state parameter passed from the Consumer to the Producer (line 11).

9.3 markupParams containing navigational state

```
<markupParams>                                     ← markup parameters, input to markup generation
  <secureClientCommunication>false</secureClientCommunication>
  <locales>en</locales>
  <locales>de</locales>
  <mimeType>text/html</mimeType>
  <mode>wsrp:view</mode>
  <windowState>wsrp:normal</windowState>
  <clientData>
    <userAgent>WSRP4J Proxy Portlet</userAgent>
  </clientData>
  <navigationalState>r00ABXNyABFq</navigationalState> ← navigational state
  <markupCharacterSets>UTF-8</markupCharacterSets>
  <validNewModes>wsrp:view</validNewModes>
  <validNewModes>wsrp:help</validNewModes>
  <validNewModes>wsrp:edit</validNewModes>
  <validNewWindowStates>wsrp:normal</validNewWindowStates>
  <validNewWindowStates>wsrp:maximized</validNewWindowStates>
  <validNewWindowStates>wsrp:minimized</validNewWindowStates>
</markupParams>
```

We omit the response from the Producer here. It simply is a new `getMarkupResponse` message we already saw in the last section.

Processing Interactions

Beyond simple content rendering based on the navigation state we want to be able to interact with the portlet and manipulate its state. In general portlets render their markup according to their state. This state can be either persistent state or some session state maintained by the portlet. In our weather service example above, a HTML form could be provided by the portlet to allow users to change the zip code for which weather information needs to be displayed.

How are interactions invoked? Similar to the `getMarkup()` request, interactions are invoked using a special URL type in the markup. WSRP defines the “blockingAction” URL type for this purpose. When a user clicks on such an URL, a `performBlockingInteraction()` call is issued to the target Producer/Portlet.

Invocations of blockingActions URLs initiate the two-step protocol of WSRP. The flow is displayed in Figure 9.14.

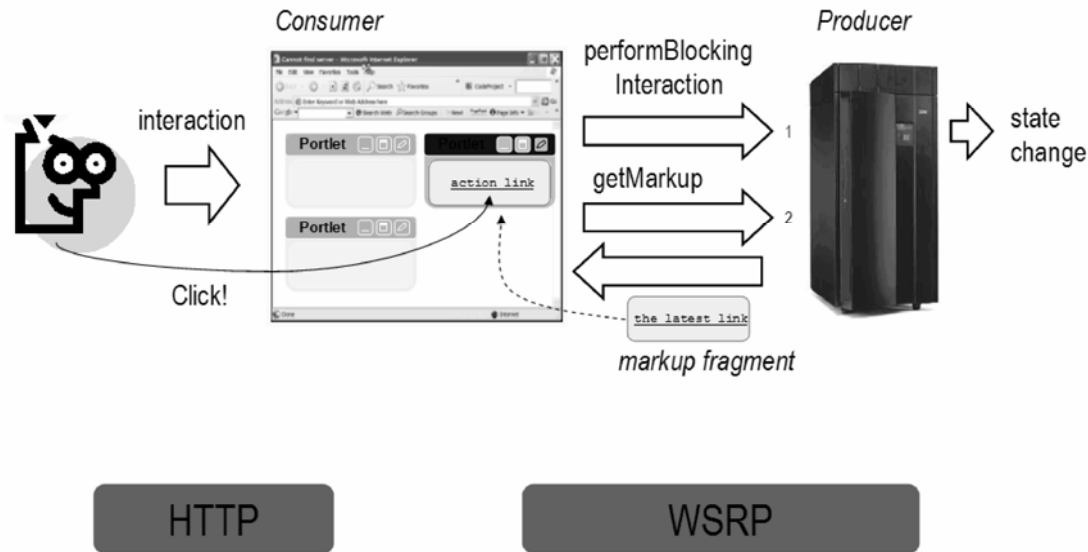


Figure 9.14 Interaction Handling after the user clicks on an action link that changes state in the producer portlet

First, the `performBlockingInteraction()` call passed to the Portlet allows it to modify its state. The Consumer needs to block until it returns the response for this call. Then – when the response message is received – it continues to retrieve the markup fragments representing the changed state. This is done by calling the `getMarkup()` operation we described in the previous section.

Why do we need the separate two phases here and why does the Consumer need to block?

Firstly, the Consumer is an aggregator which aggregates markup fragments from various portlets. Here, the Consumer could integrate multiple portlets from the same Producer on the end-user's page.

Secondly, portlets could share state. This can be done either by sharing some session data or by sharing some kind of persistent state, i.e., the same backend system. We refer to this as Producer mediated state.

Thirdly, the markup of a portlet usually represents the portlet's state, be it shared or private.

Since changes in one portlet's state could affect the state of other portlets, we need to separate the operations for manipulating the state and operations for rendering the state to enable a seamless end-user experience. Imagine the Consumer integrating portlets from a one particular producer on an end-user's page. There could be a user management portlet which allows modifying some end-user preferences like the zip code. A second weather portlet could display information based on the zip as well as the third news portlets displaying some local news flashes. If the user modifies the zip code in the management portlet it may affect rendition on the other portlets, too. If the Consumer would not utilize the two-step protocol and called the `getMarkup()` requests in parallel to the `performBlockingInteraction()` request, the weather portlet and news portlet would probably render markup on an outdated state depending on the processing order on the Producer side. Therefore separation of the state modification phase and the rendering phase is required.

Basically the `performBlockingInteraction()` call manipulates the interaction state of a portlet. This is similar to the manipulation of the navigational state on an `getMarkup()` call. So how does the interaction state flow from the Consumer to the Producer? Again URL types and their according parameters are key here. The “blockingAction” URL type can contain an URL parameter called interaction state. When the end-user invokes such an action URL the Consumer extracts the interaction state encoded in the URL and passes it as an parameter to the `performBlockingInteraction()` call. Do you see the similarity to the navigational state encoded in render URLs here?

Let's take our example portlet in Figure 9.12. The “Action” link contains “actionClicked” as the interaction state parameter. Please note that the interaction state is opaque to the Consumer and may be

encoded in a manner the portlet chooses, e.g. in base64 encoding. When the portlet receives this interaction state as a parameter of the `performBlockingInteraction()` call it increases a click counter, i.e. manipulates its state. The next `getMarkup()` call to this portlet renders the markup according to the new state. In our example it displays the new click count (see Figure 9.15).



Figure 9.15: Example Action Processing

To really understand how this all works, let's look at the low-level protocol exchange between the consumer and the producer.

Retrieving the interaction state

Let's take a look at the SOAP message level:

9.4 Example `performBlockingInteraction` request

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <performBlockingInteraction xmlns="urn:oasis:names:tc:wsrp:v1:types">
      <registrationContext>
        <registrationHandle>192.168.66.31_1088863464714_0</registrationHandle>
      </registrationContext>
      <portletContext>
        <portletHandle>0.192.168.66.31_1088863582013_1</portletHandle>
      </portletContext>
      <runtimeContext>
        <userAuthentication>wsrp:none</userAuthentication>
        <portletInstanceKey>ProxyTest_row1_col1_p1</portletInstanceKey>
        <namespacePrefix>Pluto_ProxyTest_row1_col1_p1</namespacePrefix>
      </runtimeContext>
      <userContext>
        <userContextKey>dummyUserContextKey</userContextKey>
      </userContext>
    </performBlockingInteraction>
  </soapenv:Body>
</soapenv:Envelope>
```

requests top level element, contains all parameters of the action request

registration context input parameter, represents relation between Consumer and Producer

portlet context input parameter, addressing the portlet

runtime context input parameter

user context input parameter

```

<markupParams>
  <secureClientCommunication>false</secureClientCommunication>
  <locales>en</locales>
  <locales>de</locales>
  <mimeTypes>text/html</mimeTypes>
  <mode>wsrp:view</mode>
  <windowState>wsrp:normal</windowState>
  <clientData>
    <userAgent>WSRP4J Proxy Portlet</userAgent>
  </clientData>
  <markupCharacterSets>UTF-8</markupCharacterSets>
  <validNewModes>wsrp:view</validNewModes>
  <validNewModes>wsrp:help</validNewModes>
  <validNewModes>wsrp:edit</validNewModes>
  <validNewWindowStates>wsrp:normal</validNewWindowStates>
  <validNewWindowStates>wsrp:maximized</validNewWindowStates>
  <validNewWindowStates>wsrp:minimized</validNewWindowStates>
</markupParams>
<interactionParams>
  <portletStateChange>readWrite</portletStateChange>
  <interactionState>r00ABXNyA...</interactionState>
</interactionParams>
</performBlockingInteraction>
</soapenv:Body>
</soapenv:Envelope>

```

markup parameters
input parameter

The `performBlockingInteraction()` call takes nearly the same parameters as `getMarkup()` which we already discussed in the previous section. The Consumer passes the `interactionParams` structure to the targeted portlet. One of the elements contained in the structure is the interaction state retrieved from the action URL. Another interaction parameter is the `portletStateChange` flag. This flag indicates to the portlet whether it is allowed to modify its persistent state or not. We will cover this flag in a later section. For now it is enough to understand that the interaction state is retrieved from an action URL and passed as interaction parameters back to the portlet.

You might have asked yourself why markup parameters are passed to the portlets when there is no markup generation. Well, we didn't tell you the complete truth, yet. WSRP defines an optimization to the two-step protocol. Portlets are able to return markup directly in the response message to the `performBlockingInteraction()` request. Since the portlet implementer knows whether the portlet's state will affect other portlets he can decide to directly return markup with the response if he knows it is safe to do so. Using this optimization the Consumer can omit the additional roundtrip to obtain the markup.

Responding with the new navigational state

Now, let's look at the response:

9.5 Example `performBlockingInteraction` response

```

<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <performBlockingInteractionResponse xmlns="urn:oasis:names:tc:wsrp:v1:types">
      <updateResponse>

```

performBlockingInteraction response top
level element, contains the message response

update response, contains the
new navi

```

    <navigationalState>r00ABXNyAB...</navigationalState>
  </updateResponse>
</performBlockingInteractionResponse>
</soapenv:Body>
</soapenv:Envelope>

```

In our example the response is quite simple. All the Consumer needs to know is the probably new navigational state for which the content needs to be generated. The Consumer uses the navigational state from this response as an input parameter to the following `getMarkup()` request. So basically we have two means for the Consumer to obtain the opaque navigational state needed for rendition: a) from the render URL and b) from the `updateResponse` element.

If the portlet wants to take advantage of the immediate markup return, it would return an additional `markupContext` structure contained in the update response. We saw the `markupContext` structure already when we discussed the `getMarkup()` operation and the according `getMarkupResponse()`.

Now that we've seen how the user interacts with the remote portlet and the messages that are exchanged between the consumer and producer as result of this user interactions we can take a look at another very basic functionality that is required to enable these user interactions. What we mean is generating the URLs that the user uses in order to interact with the remote portlet. This may sound simple, but as we'll see in the next section this is quite complex in the remote case. It is important to understand that you cannot treat URLs simply as strings anymore that you can manipulate as in the servlet world. As mentioned in previous chapters this is not true for the local Java portlet model and it is even more true for the remote world.

Generating URLs

In the last sections we briefly talked about various URL types triggering WSRP operations to the Producer and portlets. But how does the URL handling work? Figure 9.16 shows us a model.

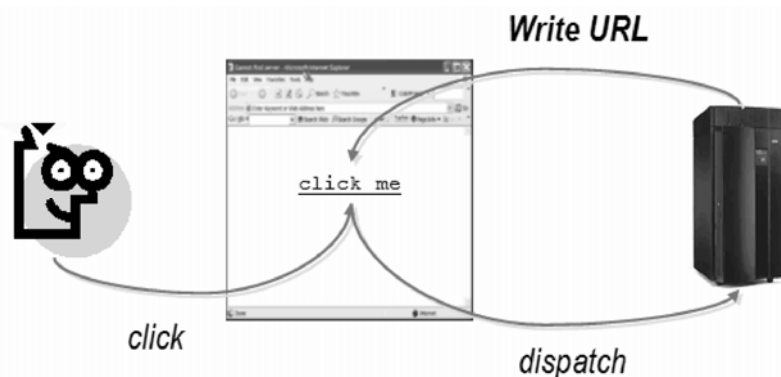
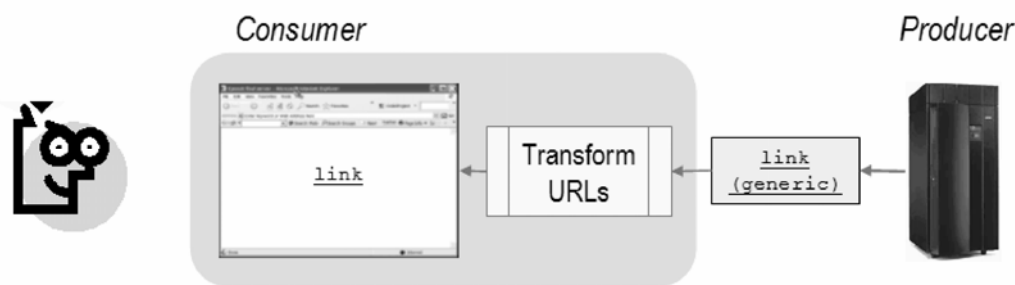


Figure 9.16: URL Generation : The producer generates a URL, the consumer renders the URL, the end-user clicks on this URL and the consumer needs to dispatch this end-user interaction to the producer.

URLs need to point to the Consumer since the end-user is actually interacting with the Consumer's aggregated pages. When looking at the base WSRP scenarios, it is only guaranteed that user agent (i.e., browser) can reach the Consumer, the Producer may be shielded from the clients by a firewall. Also the Consumer may want to have control over the end-user's interactions to, for example, enrich the request with context information or for bookkeeping purposes. Thus all interactions need to be directed to the Consumer and then appropriately re-routed to the target remote portlets. But on the other hand portlets generate the markup fragments including URLs which trigger interactions.



In addition, the same Portlet can be instantiated more than once in a single aggregated page on the Consumer side, therefore portlet URLs will have to allow the Consumer to track the Portlet to which the request is targeted. The problem is that the Producer requires Consumer-dependent information to write such a link. To finally generate a link in the markup fragments fulfilling all of our requirements, the Consumer and the remote portlets need to cooperate.

WSRP provides two means to solve the problem: Consumer URL rewriting and URL Templates. Both have their pros and cons and will be discussed in the following sections.

Understanding Consumer URL Rewriting

The idea of URL rewriting is simple. Portlets use a special syntax when generating URLs in the markup fragment. Consumers having knowledge of this syntax can extract the special URLs (which in fact, aren't URLs at this point in time) in the markup, modify them according to their requirements and finally place the modified URL back into the markup. This way portlets can generate markup fragments containing links independent of the Consumer environment. The same markup - including links - would be generated if the portlet served yet another Consumer. Figure 9.17 illustrates URL rewriting.

Figure 9.17: Consumer URL Rewriting generates markup independent of the Consumer environment

Let's take a look at the special URLs generated by remote portlets. A WSRP URL is demarked by a starting and an ending token. The starting token is "wsrp-rewrite?" while the ending token is expressed as "/wsrp-rewrite". In between the tokens the special URLs consist of name-value-pairs containing further information. The information stored in here are "well known" portlet URL parameters defined by the WSRP specification. This way the Consumer can easily parse the markup stream, obtain all necessary information and rewrite the special URLs to "real" URLs pointing back to the Consumer and the portlets on its page.

In general, WSRP portlet URLs look like this:

```
wsrp-rewrite?param1=value1&param2=value2&param3=value3&.../wsrp-rewrite
```

Let's explore the well-known URL parameters which may occur in such a portlet URL:

The first and mandatory parameter which must occur in the URL expression is the URL type. We talked briefly about URL types in the previous sections. An URL type (as the name indicates) classifies the type of an URL and denotes what kind of actions need to be taken when such an URL is invoked. WSRP defines three URL-types:

Table 9.2 WSRP URL-types

URL Type	Purpose
blockingAction	Invocation of this URL type will result in a performBlockingInteraction() operation
Render	Invocation of this URL type will result in a getMarkup() operation
Resource	Activation of this URL type will result in the Consumer acting as a gateway. Here the Consumer will fetch the passed resource, for example, a gif image, and return it to the user agent.

In addition to the URL type, further parameters are required in order to let the consumer generate a complete URL. These are

- **wsrp-navigationalState**: This parameter allows the portlet to add a new target navigational state to an URL. The Consumer is required to pass this navigational state as a parameter to the getMarkup()

request.

- `wsrp-interactionState`: This parameter allows the portlet to add a navigation state to an URL representing the interaction. The Consumer is required to pass this interaction state as an parameter to the `performBlockingInteraction()` operation.
- `wsrp-mode`: This parameter is used to change the portlet mode when invoking a portlet URL. The Consumer is required to pass the new mode as a markup parameter.
- `wsrp-windowState`: This parameter is used to change the window state when invoking a portlet URL. The Consumer passes the new window state as a markup parameter.
- `wsrp-secureURL`: The value of this parameter is a Boolean indicating whether the communication between the user agent and the Consumer needs to be secured (e.g. SSL) when this link is invoked.
- `wsrp-url`: The value of this parameter provides the actual resource URL. This parameter is used when the URL type “render” is used.
- `wsrp-requiresRewrite`: The value of this parameter is a Boolean indication whether a resource retrieved by the Consumer needs additional rewriting using the WSRP scheme. This parameter is user when the URL type “render” is used.

Let’s take a look at some examples.

Render URL

First, here is a render URL. It requests the Consumer to render the portlet’s “page2” in a new mode and window state.

```
wsrp_rewrite?wsrp-urlType=render&wsrp-navigationalState=page2&wsrp-  
mode=help&wsrp-windowState=maximized/wsrp_rewrite
```

Action URL

The following is an action URL requiring the Consumer to invoke a new action which increases the click count.

```
wsrp_rewrite?wsrp-urlType=blockingAction&wsrp-  
interactionState="newClick"/wsrp_rewrite
```

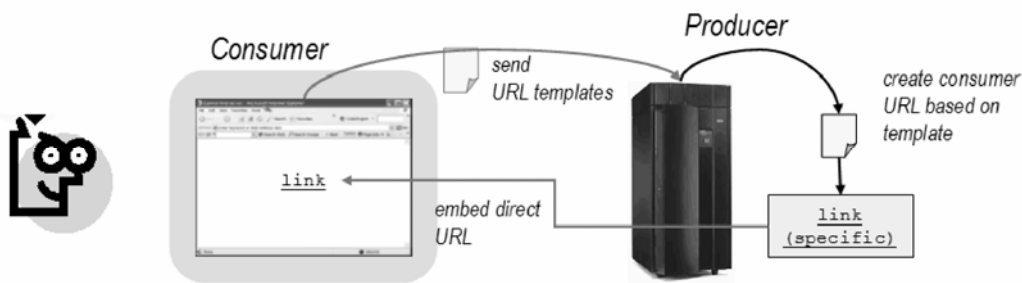
Render URL

Last is a render URL which requires the Consumer to fetch a resource and return it to the browser.

```
wsrp_rewrite?wsrp-urlType=render&wsrp-  
url=http%3A%2F%2Fyour.com%2Fimages%2Fpic1.jpg&wsrp-  
requiresRewrite=false/wsrp_rewrite
```

Now you have learned how the consumer can rewrite the URLs that the producer created in its markup to make them fit into the actual consumer environment. You may have noticed that this technique puts the entire burden on the consumer and is very easy and efficient for the producer.

In the next section we cover an alternative approach that WSRP defines for URL generation that puts the burden on the producer instead of the consumer.



Working with URL Templates

URL Templates take a different approach on URL generation. Rather than generating special URLs in the markup which need to be rewritten by the Consumer, Portlets using the templating mechanism write ready-to-use URLs directly into the markup. To enable this, Consumers send predefined URL templates to the portlet which contain placeholders for the well-known URL parameters we already discussed in the previous section. Portlets use these templates for their URLs in the markup and replace the replacement tokens with valid values.

Figure 9.18 illustrates the interaction between the Consumer and portlets.

Figure 9.18: URL Templates describe well-known URL parameters, allowing portlets to enter values for the replacement tokens.

Let's take a closer look at the templates. The formats of templates are completely up to the Consumer, since only the Consumers know how the final URLs should look like. So basically a template is an URL string with predefined replacement tokens. The replacement tokens are enclosed in curly braces. The value within the curly braces is the name of an URL parameter that should be replaced by the portlet.

The following is an example of a generic URL template:

```
http://consumer.your.com/appPath/{wsrp-URLtype}?mode={wsrp-
mode}&navigationalState={wsrp-navigationalState}&...
```

From the portlet's perspective all characters in the template are constants except for the replacement tokens. This way the Consumer can prepare a ready-to-use URL that need no additional rewriting. However, portlets must be aware that the URL stored in the template may still not be a valid URL but may be composed in such a way that Consumers use it for further additional processing. As a consequence portlets required to treat templates as they are and must not modify them.

Besides the well-known URL parameters in the previous section, there are four more URL parameters defined for template usage. These parameters need to be replaced by the portlet with values supplied by the Consumer in a request data field. This way Consumers can be enabled to handle URL templates being more generic, i.e., this additional data helps the Consumers to generate generic templates valid for all targeted portlets without the need of a per-portlet modification of templates sent over the wire:

Table 9.3 Additional URL template parameters

WSRP_specific URL Parameter	Purpose
wsrp-portletHandle	The portlet must replace this token with the value of the portletHandle field in the PortletContext structure. We've seen this structure when discussing the getMarkup() operation.
wsrp-userContextKey	The portlet must replace this token with the value of the userContextKey in the UserContext structure. The user context key is used to distinguish between different users on the Consumer interacting with portlets.
wsrp-portletInstanceKey	The portletInstanceKey is also sent in the RuntimeContext structure with each request. The portlet instance key is set by the Consumer to identify multiple instances of the same portlet (identified by the portlet handle). This can be useful when the same portlet is being put on the same page of one user twice. Here the instance key helps to distinguish between these two instances.
wsrp-sessionID	The sessionID is also sent in the runtime context.

The WSRP specification defines various templates to be used by the portlets depending on which links need to be generated.

- `blockingActionTemplate`: This template has to be used by the portlet whenever an action URL needs to be generated. As a result of invocation of this URL will result in a `performBlockingInteraction()` call back to the portlet.
- `renderTemplate`: Render Templates are obviously used to generate render URLs which result in the invocation of `getMarkup()`.
- `resourceTemplate`: This template is to be used for resources like images. This URLs make the Consumer fetch resources – probably in a cached manner – and act as a proxy for the end-user agents.
- `defaultTemplate`: This is a generic template that has to be used whenever a specialized template was not provided by the Consumer.

For all these templates there are their secure pendants. For example the `secureRenderTemplate` is a template that has to be used whenever secure communication channels are to be used, i.e., the difference here is that the URLs stored in these secure templates use SSL connections.

Templates are sent by the Consumer to the Producer/portlets in the runtime context with each operation. But templates can be considerably large and there are two mechanisms to generate URLs, so isn't that adding additional overhead to the protocol? Yes, it does add overhead, but there are optimizations.

Portlets can declare in their meta-data whether they want to use templates or not. Consumers obtain this meta-data from the service description. We briefly mentioned the service description interface in section 9.4. This interface provides the `getServiceDescription()` operation which allows the Consumer to obtain all meta-data about the Producer and the hosted portlets. If a portlet indicates that it supports template processing, the Consumer must send the templates, otherwise it can omit them.

Another optimization is that the portlet can declare that it stores the templates within its session. This way the Consumer needs to send the templates only once and resend them whenever the session has expired and a new session has been set up by the portlet.

Using both optimizations, the overhead of template usage gets minimized to a reasonable extent.

Now that we've learned how the end-user can interact through URLs with the WSRP producer it is time to look at other functions that you normally take for granted in the web world and see how they are implemented in WSRP. The next important function that we'll cover is the concept of a session to store and share data related to an end-user session.

Managing Sessions

When we discussed basic interaction handling, we saw two types of states involved in these interactions: navigational state and interaction state. The navigational state is used to generate the markup fragments correctly multiple times and is usually stored on URLs to enable bookmarkability of these fragments. The interaction state is a transient state which lives over the course of one request cycle. Interaction state can be considered as being input parameters to interactions.

But applications usually want to accomplish more than this. Portlets want to maintain state across a sequence of operations. A typical example for maintaining such state over the course of time is the famous shopping cart example. Users can navigate an eCommerce site, add items to their shopping cart and finally, once they are finished, order the items put into the cart. The items in the cart need to be maintained until the user submits his order or logs out.

This state is referred to as a session. WSRP provides session support and can be compared to HTTP sessions. A session is a transient state maintained by the Producer. Consumers refer to this session using an opaque handle called the session id. The following figure gives an overview of WSRP sessions.



Figure 9.19 WSRP Sessions permit a portlet to maintain state across a sequence of operations.

The portlet session is scoped to a portlet and is directly reflected in the WSRP protocol. The life cycle of a session is transient and is established by the Producer whenever a portlet needs to store some state in the session. The session is established by returning a `SessionContext` structure as a result of a `getMarkup()` or `performBlockingInteraction()` operation.

The session life cycle ends implicitly when the session expiration time is being reached or if the Producer returns a new `sessionID` as a result of an operation thus invalidating the old one. In addition Consumers can explicitly destroy sessions by using the `destroySessions()` operation of the markup interface.

In addition to portlet sessions Producers can implement “group sessions”, producer-mediated sharing of sessions. For example, the Producer might introduce a shared data area for this purpose. Group sessions are not directly reflected in the WSRP protocol. This session type is typically represented using transport mechanisms, i.e., HTTP cookies. Producers can use HTTP cookies to implement this type of session sharing.

Producers can indicate which portlets might share common data in sessions using a `groupID` as an attribute in the portlets’ descriptions (obtained by the Consumer through the service description).

The Consumer must assist the Producer to establish HTTP cookies correctly. This is done in the following manner. First, the Producer indicates how cookies are scoped. This is part of the meta-data Producers provide in their service description. Here the Producer can declare that it doesn’t use cookies at all, that it scopes the cookies per user on the Consumer or that they scope the cookies per `groupID` and per user on the Consumer.

To establish cookies the Consumer must call the `initCookies()` operation according to the requirements described in the meta-data. The `initCookie()` operation returns HTTP cookies on the transport level. These cookies must be provided by the Consumer with each markup interface operation invocation.

Why does WSRP define HTTP cookie handling while Web Services are transport-agnostic in general? This is definitely a true statement, but the WSRP technical committee couldn’t ignore the fact that most Producers and Consumers still are web applications which leverage today’s Web technologies and often depend on HTTP cookies. For example many J2EE load balancers rely on HTTP sessions to appropriately deliver requests to cluster nodes.

Providing Portlet Customization

So far we have talked about Consumers interacting with portlets offered by the Producers and modifying their transient state over the course of the interaction. But one key point of portals and portlets is personalization and customization of portlets. Users want to have their own instances of portlets with their own preferences set.

An example of such a customization is a mail portlet. The producer offers a mail portlet which connects to a mail server for the end-user. It wouldn’t make sense to connect each user of the portlet to the mail server using the same preconfigured account. Here the portlet has to provide customization methods to allow the user to enter his account information used for the connection. We could go even one step further and have a generic mail portlet allowing users to connect to arbitrary mail servers.

Of course we don’t want users to enter this connection information every time they log in to the portal and start using the remote mail portlet – this would be the case if we just used sessions. This leads us to two

requirements. Firstly, portlets must be able to handle persistent state. Secondly, Consumers must be able to generate separate instances of portlets on the Producer on behalf of their users.

In WSRP we distinguish between two “types” of portlets. As said, Producers expose portlets Consumers can use in their service description; each portlet is uniquely identified by its portlet handle. We refer to these portlets as “Producer Offered Portlet”. Producer offered portlets are preconfigured and not modifiable by Consumers, i.e., Consumers can not modify the persistent state of such portlets.

To allow Consumers to clone and customize portlets, Producer can expose the portlet management interface. Using this interface Consumers may request a unique configuration of a portlet. We refer to such cloned portlets as to “Consumer Configured Portlets”.

So how can a portlet’s persistent state be modified? A portlet has multiple ways to do this. The first way is implicit. Portlets can store their persistent state whenever user interactions with the portlet’s markup result in a persistent state change. One example is the portlet’s “edit” mode, where the portlet can display forms and input fields allowing the user to customize the portlet. In our mail example, the portlet would generate a form allowing the user to enter the target mail server and the account information. Once the user submits the form (resulting in a `performBlockingInteraction()` operation), the portlet stores the submitted data persistently on behalf of the end-user.

In addition portlets may expose properties in their portlet description. Consumers can use the `get/setProperties()` operations from the portlet management interface to directly modify the portlet’s customization. This can be useful when Consumers already have the necessary information required to customize a portlet accordingly. One example could be a Consumer-generated user interface allowing the end-user to enter the data. In our mail example, a Consumer could already have a preconfigured mail server with all of their end-users’ account information and explicitly configure the remote mail portlets on behalf of its end-users.

Let’s take a look at cloning. There are two ways to clone a portlet: explicit cloning initiated by the Consumer and implicit cloning.

Explicitly Cloning a portlet

Consumers can explicitly clone a portlet by invoking the `clonePortlet()` operation from the portlet management interface. Let’s take a closer look at this operation.

Here is the request sent by the Consumer:

9.6 Example clonePortlet request

```
?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <clonePortlet xmlns="urn:oasis:names:tc:wsrp:v1:types">
      <registrationContext>
        <registrationHandle>192.168.66.57_1096115097926_0</registrationHandle>
      </registrationContext>
      <portletContext>
        <portletHandle>0.1</portletHandle>
      </portletContext>
      <userContext>
        <userContextKey>Erebus</userContextKey>
        <profile>
          <name>
            <given>John</given>
            <family>Doe</family>
          </name>
        </profile>
      </userContext>
    </clonePortlet>
  </soapenv:Body>
</soapenv:Envelope>
```

clonePortlet request top level element, contains all parameters of the clone request (points to `<clonePortlet>`)

portlet context input parameter, addressing the portlet (points to `<portletHandle>`)

user context input parameter (points to `<userContextKey>`)

registration context input parameter, represents relation between Consumer and Producer (points to `<registrationHandle>`)

```

        <middle>Joe</middle>
        <nickname>Jo</nickname>
    </name>
    <gender>mmh</gender>
    <employerInfo/>
    <homeInfo/>
    <businessInfo/>
</profile>
</userContext>
</clonePortlet>
</soapenv:Body>
</soapenv:Envelope>

```

The clone portlet request takes three parameters: the registration context, the portlet context and the user context. The key parameter is the portlet context containing the portlet handle of the portlet which needs to be cloned.

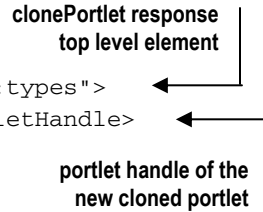
As a result the Producer returns the following response:

9.7 Example clonePortlet response

```

<?xml version="1.0" encoding="utf-8"?>
  <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Body>
      <clonePortletResponse xmlns="urn:oasis:names:tc:wsrp:v1:types">
        <portletHandle>0.192.168.66.57_1096115142350_1</portletHandle>
        <portletState/>
      </clonePortletResponse>
    </soapenv:Body>
  </soapenv:Envelope>

```



The response returns a new portlet context identifying the new portlet. From this point in time the Consumer can use this new portlet handle to address a unique configuration and customization of the portlet.

Implicitly Cloning a portlet

With the explicit clone Consumers have a means to provide individually customized portlets for their end-users. But keeping a separate instance of one portlet for each end-user may be inefficient and not always required. Assume the following scenario.

The consumer uses a remote flight schedules portlet. This portlet displays flight schedules of airports of a certain region. The region to display the airports might be one customization parameter the portlet offers – again either by providing a user interface which allows the end-user to modify the region or by exposing a public portlet property.

The Consumer has preconfigured the portlet to display east coast airports as the default because the end-users the Consumers serves are mainly from the east coast. In this case it wouldn't be necessary to have copies of the portlet for each end-user; rather it would be more efficient if all the users shared the same portlet. The sharing of the portlet can last as long as users don't customize the portlet for their own purposes. If a user decides to customize the portlet by, for example, changing the preferred region of airports to display, a new clone must be generated for this end-user.

To accomplish this Consumers and Producers/portlets must cooperate. Only the portlet can know when a persistent state change is required. In general such state change can principally occur on any user

interaction. On the other hand only the Consumer knows whether or not such a state change would be safe. In our example if users share the same portlet, modification by one user would affect other users, too.

WSRP introduces a solution to that by defining the clone before write behavior. Figure 9.20 illustrates how this is accomplished.

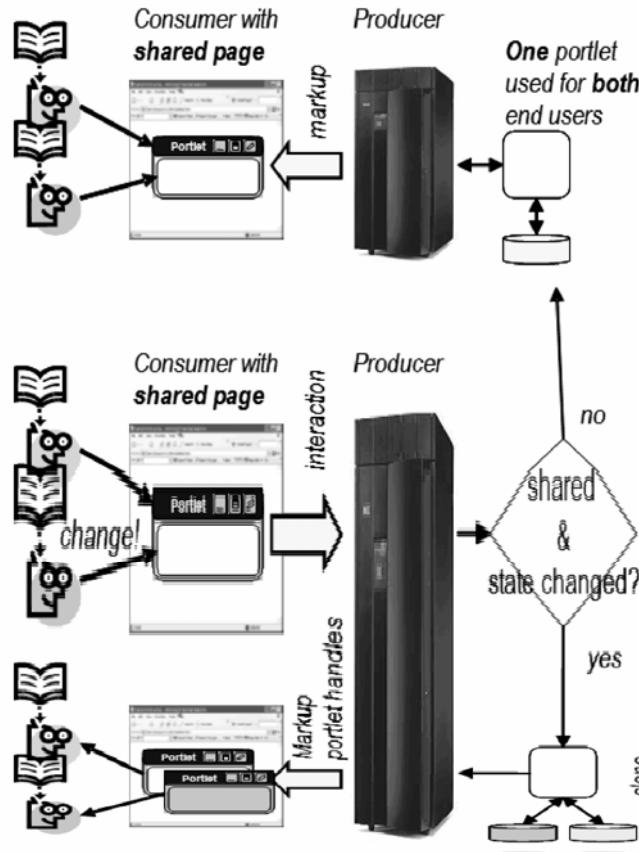


Figure 9.20 Clone Before Write

At the beginning, users share the same portlet and can interact with it. Since the Consumer cannot determine when a persistent state change of the portlet is required it passes a flag with each `performBlockingInteraction()` request to the portlet to indicate whether a state change is allowed or if appropriate actions need to be taken prior to invoking the action on the portlet.

The state change flag can have the following values:

Table 9.4 WSRP state change flag values

State Change Flag Value	Purpose
<code>readOnly</code>	the Consumer informs the portlet that a persistent state change is not allowed. If an interaction would result in a necessary state change, the portlet throws a fault. In this case the Consumer can explicitly clone the portlet for the end-user or display an error message.
<code>readWrite</code>	the portlet can proceed and change the persistent state with no additional effort
<code>cloneBeforeWrite</code>	instructs the Producer to clone the portlet implicitly prior to writing any persistent state change for the portlet instance. Once the portlet is cloned the portlet may write the changes. As a result of such an implicit clone on a <code>performBlockingInteraction()</code> the Producer returns the newly generated portlet handle back to the Consumer. From that point in time the Consumer can associate this new portlet clone with that particular user

As we've discovered in this section, WSRP covers more than just how to produce markup remotely. It also deals with management aspects of portlets and thus goes a step further than the Java Portlet Specification.

Important to note for you as Java Portlet developer is that this management complexity is hidden from you by the infrastructure of the producer.

Summary

In this chapter we introduced you to the distributed world of Web Services for Remote Portlets. We talked about the WSRP vision to enable the Internet being a pool of visual, user facing components ready to be plugged-in to portals with just a few mouse clicks. We discussed how Web services and SOA can be key technologies to enable this vision and let WSRP act as a bridge over various platforms and technologies and allow easy integration of applications across these boundaries.

We then stepped a little bit deeper into the protocol and showed you how Consumers can interact with remote portlets and what happens on the protocol level. Next we looked at two examples on how end-users will interact with the remote portlet and how these interactions map to different WSRP calls: the getMarkup call for navigational state changes and the performBlockingInteraction call for interactions that change state on the consumer.

As the interaction of the end-user with the remote portlet is based on URLs that the user clicks we explained the two mechanisms how a producer can create URLs in the markup: either by requesting the consumer to rewrite the URLs or by taking a template from the consumer and filling in the correct values. We then took a look at how sessions are realized in WSRP and enable remote portlets to share data in a user session scope like programmer would expect that are familiar with J2EE and the HTTP session concept.

Finally we got into the management capabilities that WSRP defines and how portlet customization can be implemented. As noted previously this complexity will be hidden from the Java Portlet programmer as the producer container will take care of the management and lifecycle operations.

This was a brief introduction and we definitely did not cover all the details and capabilities of the protocol. Interested readers are invited to read the WSRP specification and accompanying resources found on the WSRP home page [1].

In the next chapter, we'll discuss one of the first implementations of the WSRP protocol: the Apache WSRP4J project.

References

Web Services for Remote Portals Technical Committee: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrp.

WSRP 1.0 standard specification document: <http://www.oasis-open.org/committees/download.php/3343/oasis-200304-wsrp-specification-1.0.pdf>.

Jakarta Tomcat: <http://jakarta.apache.org/tomcat>

WSRP Primer 1.0: <http://www.oasis-open.org/committees/download.php/10539/wsrp-primer-1.0.html>

Chapter 10 WSRP in Action: WSRP4J

Now that you have a first impression of the concepts and ideas behind the Web Services for Remote Portlets (WSRP) standard, it's time to get real and take a look at WSRP in action. The Apache WSRP for Java (WSRP4J) project is the open source implementation of the WSRP standard. WSRP4J defines an open architecture that provides WSRP conformant services based on free open source software like Apache Tomcat, Apache Axis and Apache Pluto. It also provides a generic proxy portlet written against the JSR168 portlet API which can be plugged into JSR168 compliant portals and enables the integration of WSRP services into such portals.

WSRP4J provides a WSRP Producer and two Consumer implementations. In this chapter, we will give you a brief overview of the WSRP4J architecture and discuss how to install and run the WSRP Producer and the two Consumers.

Understanding the Architecture

The WSRP4J project provides two Consumer implementations and one Producer implementation.

The first Consumer is a sample Java Swing-based application. It was developed for simple demo and testing purposes and is not a full blown implementation. However it can be used to demonstrate the main capabilities of the WSRP protocol.

Yet another Consumer is the Pluto proxy portlet. The proxy portlet is a JSR168 portlet that can be plugged into the Pluto portal and act as a proxy to WSRP services. From the portal perspective, the proxy portlet behaves like any other JSR168 portlet. However, the proxy portlet points to a remote portlet and translates all local API calls to WSRP requests and returns the markup of the remote portlet. We have discussed the idea of generic proxies in chapter 9.3.3.

On the Producer side, WSRP4J introduces a pluggable provider concept. The main implementation provides the general protocol handling (WSRP Engine) and interfaces which various providers may use to allow the engine to interact with them. A provider is basically a component which actually manages and implements remote portlets, i.e., it provides the business logic for the WSRP services.

WSRP4J implements a provider based on the Apache Pluto portlet container. The portlet container manages the life cycle of portlets and handles interactions with them (Figure 10.1). Virtually any JSR168-based portlet can be exposed as a WSRP service using the WSRP4J Pluto provider.

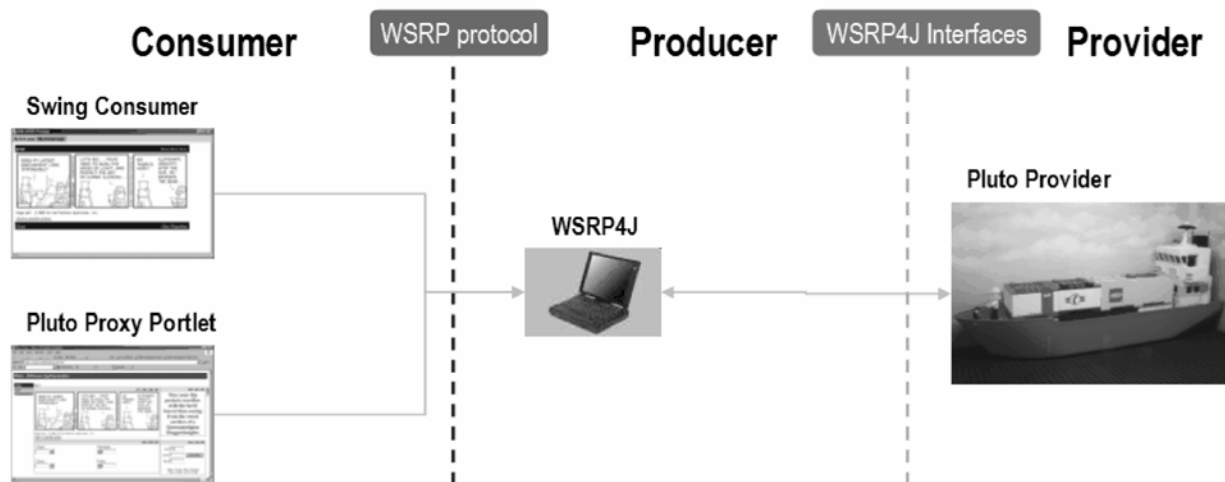


Figure 10.1 WSRP4J Implementations

We'll discuss each of these components in more detail later, but first, let's take a closer look at the integration of Portals and WSRP utilizing the WSRP4J architecture, shown in Figure 10.2:

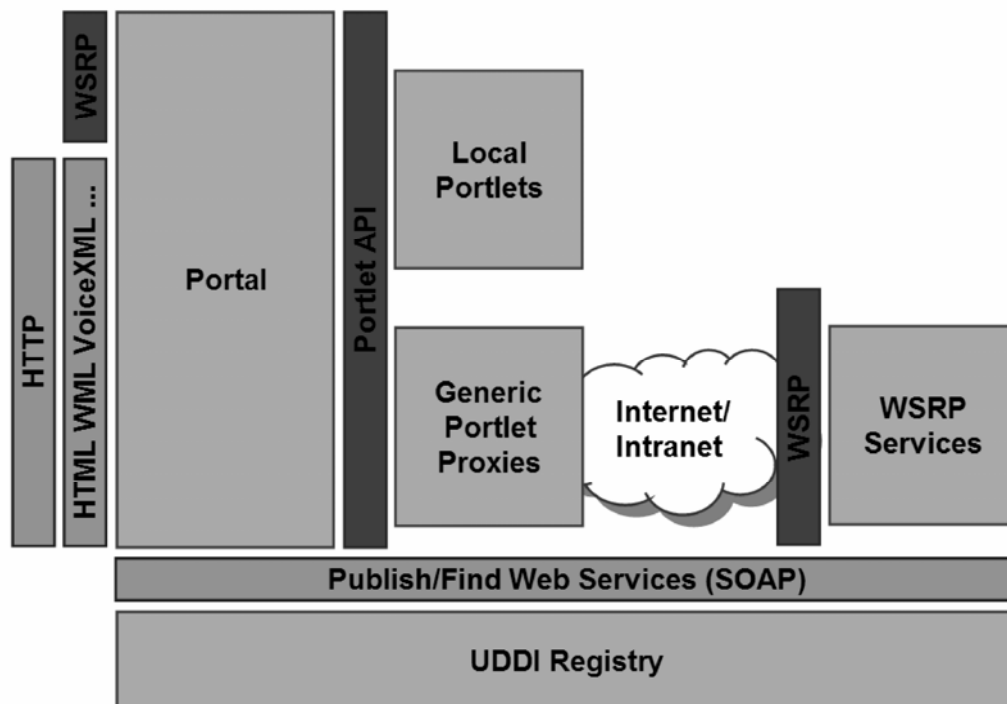


Figure 10.2 WSRP4J allows portals to be exposed as a WSRP service and to consume other WSRP services

By exposing the WSRP portTypes and enabling SOAP access to the portal, local portlets can be exposed as WSRP services. SOAP clients implementing the WSRP protocol (namely WSRP Consumers) can access the locally deployed portlets managed by the portlet container.

On the Consumer side, portals can utilize the generic proxy portlet concept to access WSRP services. As said, such generic proxy portlets behave just like locally deployed portlets but instead of implementing the business logic of a portlet, the generic proxy portlet points a remote portlet and translates the local invocations to WSRP remote calls and thus allows integrating any remote WSRP service into the portal.

Using this infrastructure, portals can share their portlets over the network. Thinking one step further, even the portal implementations get completely decoupled. The consuming portal could be based on J2EE while the producing portal could be based on the .NET framework.

Producer

The Producer architecture provides a very modular architecture enabling an easy exchange of each component's implementation. All components excel by interfaces hiding the provider environment's object model and thus gaining independence of changes in the provider implementation or design. Figure 10.3 gives an overview of the Producer architecture.

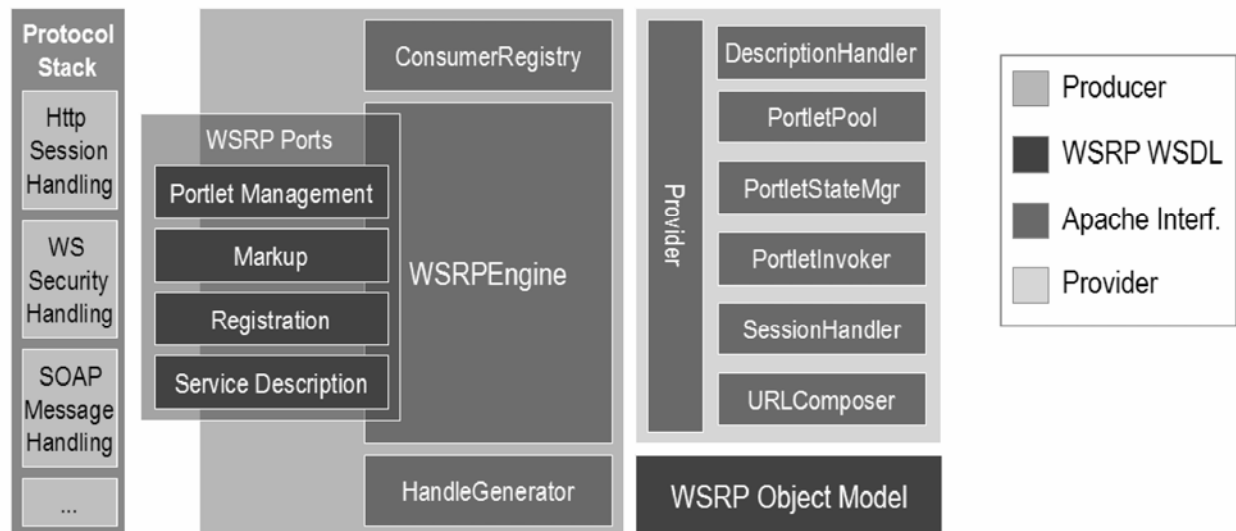


Figure 10.3 WSRP4J Producer Architecture describes a pluggable model allowing to implement various providers]]

The main entry point to the Producer is the WSRP engine component. The WSRP engine is responsible for the handling of the WSRP protocol and for interacting with the Provider component(s). The engine exposes the four WSRP portTypes (interfaces) we discussed in chapter 9. The engine is deployed as a Web Service on the application server – WSRP4J uses Apache Tomcat as the application server.

The ConsumerRegistry is responsible for managing the Consumer registrations to the Producer. The HandleGenerator generates all required handles/IDs used for interactions in the WSRP protocol.

All protocol interactions base on the WSRP Object Model. The object model is generated from the WSRP WSDL and schema files. It is used to generate the appropriate messages and types embedded in the WSRP requests and responses.

Providers need to implement the Provider interface. This interface is the access point for the WSRP engine to the provider implementation and hides the provider's implementation details. The engine uses this interface to obtain the other components to invoke and handle interactions with portlets.

The main provider component is the PortletInvoker which wraps the invocation mechanisms of the provider and provides the implementation with the required environment components.

The PortletPool manages the life cycle of portlets and allows the engine to clone or destroy portlet instances.

The DescriptionHandler is used to obtain descriptions of the exposed portlets. The WSRP engine uses this interface to generate the WSRP service description with the embedded portlet descriptions which is exposed to the Consumers.

The PortletStateManager allows the engine to obtain an opaque representation of the portlet's state and can use this blob to push it to the Consumer. The WSRP protocol allows the portlet's state to be pushed to the Consumer to enable light weight Producer environments.

The SessionHandler is responsible for managing portlet sessions. In WSRP4J the session management is completely based on HTTP sessions and cookies, i.e., the Pluto provider doesn't need to provide its own session management.

Another important component is the URLComposer. As we discussed in Chapter 9 special URLs need to be generated to finally point to the Consumer rather than the Producer. Therefore the Provider's URL generation mechanism might not be appropriate. WSRP4J provides an URLComposer which generates correct WSRP URLs and can utilize either Consumer URL rewriting or URL template processing.

By implementing the described provider interfaces virtually any external component can be wrapped as a provider and plugged into the Producer architecture. This enables these components to be exposed as WSRP services ready to be integrated by WSRP Consumers.

In the following we will describe the Consumer architecture to complete the picture.

Consumer

The Consumer architecture is also very modular and defines various interfaces which hide the implementation details of the underlying components. Figure 10.4 illustrates the Consumer architecture.

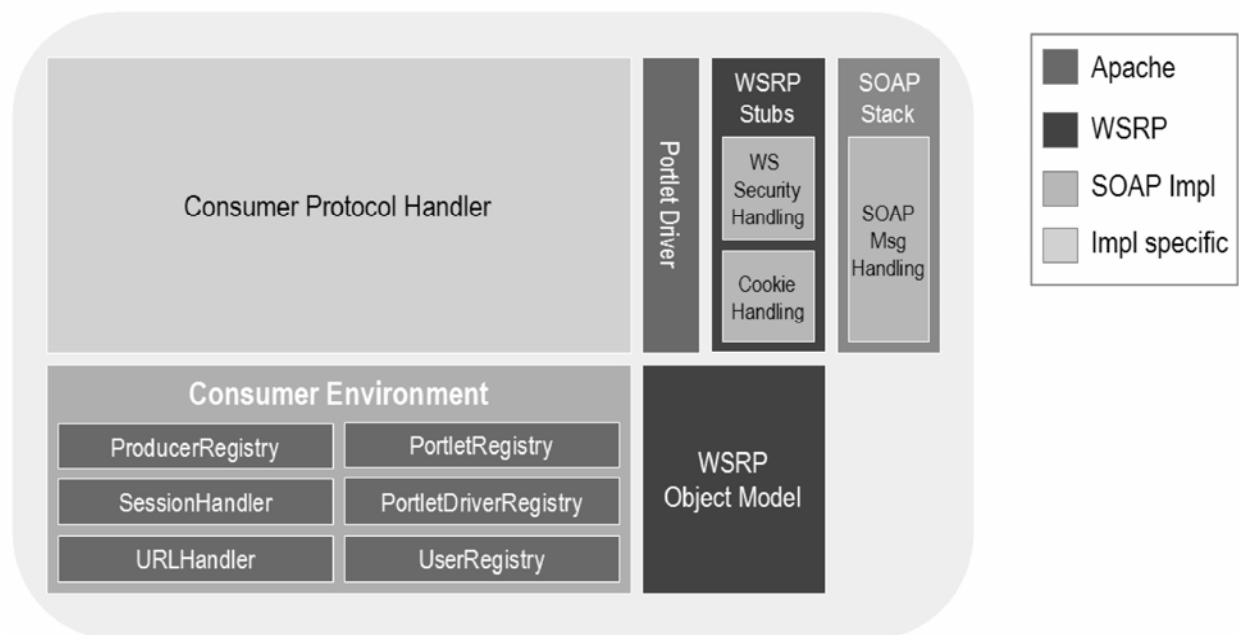


Figure 10.4 WSRP4J Consumer Architecture

The main component is the Consumer Protocol Handler. This handler is implementation specific and depends on the Consumer environment. In our case of the WSRP4J proxy portlet, the proxy portlet implements the Java Portlet API and handles calls from the portlet container to the portlet. Here the translation to WSRP calls takes place.

The protocol handler utilizes well defined components provided by the Consumer environment the protocol handler is running in. The access to these components is modeled via interfaces allowing the implementations to change.

The ProducerRegistry manages details about remote WSRP Producers from which portlets are being integrated.

The PortletRegistry provides an abstraction of remote portlets' meta-data and descriptions.

The UserRegistry provides access to the currently logged-on end-user on whose behalf the requests are issued.

The responsibility of the URLHandler is to manage URL generation on the Consumer side. The URL handler mainly rewrites the WSRP specific URLs to the Consumer environment's URLs so that users interacting with the generated markup are directed to the Consumer environment.

Like the SessionHandler on the Producer side, the Consumer's SessionHandler is responsible for managing the remote WSRP sessions. The proxy portlet WSRP4J implementation completely relies on HTTP session, so no special session handler is required.

To obtain a particular PortletDriver implementation the PortletDriverRegistry is being used. WSRP4J provides a default PortletDriver.

The PortletDriver is the main component for connecting the local portlet to the outside world. It provides a task oriented abstraction of the WSRP protocol handling. Using the PortletDriver, remote calls to the markup interface can be issued (like `getMarkup()` and `performBlockingInteraction()`). The PortletDriver uses the WSRP Object Model which again is generated from the WSRP WSDL and schema files. To issue a remote call the WSRP stubs are being used. These stubs are responsible for the SOAP protocol and HTTP session handling.

In this rest of this chapter, we guide you through the installation and configuration steps for each component:

- the WSRP4J Producer which utilizes the Pluto portlet container
- the simple Java Swing Consumer
- the WSRP4J proxy portlet which runs on the Apache Pluto portal.

First, though, you'll need to assemble a few tools to get going with WSRP4J.

Getting Started: WSRP4J Prerequisites

Now it's time to install the WSRP4J Producer and Consumers. First, you will need to download the code and setup the environment. We will then start building and configuring the Producer. Finally we complete the demonstration by setting up the two Consumers provided by WSRP4J.

CVS Client to Download the Code

To run the WSRP4J Producer and Consumers you first need to obtain and build the code from CVS. You can use the Eclipse CVS client if you work in Eclipse, but any CVS client will do.

As of writing this book, WSRP4J was in the Apache Incubator. This means that there are no downloadable packages at that time. Rather you must obtain the latest code from the CVS repository directly – which is more convenient for developers anyway.

Anyone can checkout the code directly from CVS directly using the anonymous access. Use the following command to obtain the complete source tree. If you have a CVS client you can configure it accordingly.

```
cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic login  
password: anoncvs
```

```
cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic checkout ws-wsrp4j
```

Alternatively, we provide you with a snapshot of the WSRP4J code tree. You can download the snapshot from <http://www.manning.com/portlets>.

JDK to Build the Code

We used JDK 1.4 to build the code, however the minimal requirement is JDK 1.3.

Tomcat to Run the Pluto Portal

Since the WSRP4J Producer and the proxy portlet running on the Pluto portal are deployed on a J2EE application server you need to install Apache Tomcat, too. You can download a binary distribution of Tomcat from the Jakarta Tomcat Web site [3]. Please refer to the installation instructions provided there. In the further sections we assume you have Tomcat appropriately installed.

Setting up the Producer

Installation of the WSRP4J Pluto provider is straightforward. We assume you have successfully downloaded the code to a directory we refer to as WSRP4J_ROOT.

The WSRP4J build environment is based on Apache Ant. For convenience WSRP4J comes with install and build scripts/batch files which you can start from a command line. Of course you can use the built in Ant support in Eclipse to run the Ant tasks. In our examples we will use the provided scripts.

Building and Installing the Code

First we need to build the code. To do this, open the command prompt and change to the WSRP4J_ROOT/build directory. In this directory you will find the build.xml file which defines the various Ant tasks we need for building the code and installing the parts.

To build the WSRP4J Pluto provider run the build.bat or build.sh script with the parameter “build”:

```
build.bat build
```

This command invokes Ant with the target task “build”. Once the command finishes you should see a “Build successful” message on the screen.

The next step is to install the WSRP4J Pluto provider on Apache Tomcat. Before doing so you need to edit a property file which defines the Tomcat home directory, the HTTP port Tomcat listens to and Tomcat’s major version number. WSRP4J comes with a sample file called build.properties.example. Copy the file to build.properties and edit the contents of the new file:

```
// the tomcat home dir where wsrp4j will be installed
TOMCAT_HOME=D:/Apache/Tomcat
// the port tomcat is listening on
TOMCAT_PORT=8080
// tomcat major version
tomcat.major.version=5
```

You need to change the variable value of TOMCAT_HOME to the directory you installed Apache Tomcat and to change the variable value of TOMCAT_PORT to the port Tomcat listens to. By default Tomcat is configured to listen to the HTTP port 8080.

Deploying the Pluto Provider

Now we're ready to deploy the WSRP4J Pluto provider. This is simply done by invoking the `install-provider-pluto.bat/sh` script. This script again invokes Ant with the target "install-provider-pluto":

```
Install-provider-pluto.bat
```

After the script has finished you should see a "build successful" message on your screen.

Note that this target also implicitly deploys the portlets we expose through the Pluto provider. Don't care about that yet. We will discuss the deployment of portlets in a separate section to give you a better understanding how things work. However, for the future you can keep in mind that you don't need to separately deploy your portlets.

Testing the Installation

After we've installed the Pluto provider, we should check if things really work. We do this by checking if the WSRP engine's Web service ports are deployed correctly and ready to receive requests. First start your Tomcat server. Once the server has started point your browser to one of the following URLs (we assume Tomcat is running on your local machine on the default TCP port here):

<http://localhost:8080/wsrp/wsrp4j/WSRPServiceDescriptionService>

<http://localhost:8080/wsrp/wsrp4j/WSRPBaseService>

<http://localhost:8080/wsrp/wsrp4j/WSRPPortletManagementService>

<http://localhost:8080/wsrp/wsrp4j/WSRPRegistrationService>

You should see something similar to Figure 10.5 in your browser window:



Figure 10.5 Testing the WSRP Ports

If the Apache Axis servlet returns the output shown above we know our Web service is deployed correctly. For now you can shutdown Tomcat.

By completing this step we have the Pluto provider up and running. Next, we need to provide the actual content that will be made available via WSRP.

Setting up the Test Portlet

Now that we have the Pluto provider up and running we're missing one thing: the portlets we want to expose. We already mentioned that the WSRP4J Pluto provider can expose any JSR168 portlet as a WSRP service. The WSRP4J implementation comes with one test portlet which we want to deploy in our Pluto provider and expose as a WSRP service. Before we start, let's take a look at the build environment and infrastructure for portlets.

In the WSRP4J_ROOT directory you will find the following directory structure:

```
.
••portlets
• ••build
• • ••wsrpctest.include
• ••lib
• ••wsrpctest
• • ••src
• • ••war
• • ••build.properties
• • ••...
••...
```

The root directory for all portlets to be built and deployed is the WSRP4J_ROOT/portlets directory. Each portlet has its own sources and property files in a subdirectory. If you intend to add further JSR168 portlets you need to create another subdirectory with the portlet application's name.

Each portlet needs to have a `build.properties` file. This file contains the Tomcat home directory and the portlet container's application name, into which the portlet will be deployed.

Preparing the Test Portlet

WSRP4J comes with a sample file called `build.properties.example`. Copy the file to `build.properties` and edit the contents of the new file:

```
// the name of the target jar file
jarfile=wsrpctest.jar
// the name of the target war file
warfile=wsrpctest.war
// the tomcat dir we want this portlet to be deployed to
TARGET_TOMCAT_HOME=D:/Apache/Tomcat
// the webapp containing the portlet container we use for deployment
TARGET_WEBAPP_NAME=wsrp
```

You need to change the variable value of `TARGET_TOMCAT_HOME` to the Apache Tomcat install directory. You can leave the other values as they are.

After you have prepared the portlet, it needs to be included in the build and deployment process. This is simply done by adding an include file in the `portlets/build` directory. By default the `wsrpctest` portlet is already included in the build process.

Building and Deploying the Test Portlet

To build portlets, change back to the WSRP4J_ROOT/build directory and run the build script with the target "build_portlets".

```
build.bat build_portlets
```


After the script has finished you should again see a “build successful” message on your screen.

Portlet deployment is as easy as the other tasks. Simply run the build script with the “deploy_portlets” parameter.

```
build.bat deploy_portlets
```

Again a “build successful” message should indicate to you the proper deployment.

Configuring the Producer

There is one final step we need to accomplish. What we have done so far is deploy the WSRP Producer/Pluto provider and deploy the portlets into the container. There is one additional configuration file – the `portletentityregistry.xml` - which defines which portlets deployed on the container are actually exposed as WSRP services and therefore accessible for Consumers.

You can find this file on the deployed WSRP4J Pluto provider in the directory `TOMCAT_HOME/webapps/wsrp/WEB-INF/data`. You can change the file for your deployed Producer, but note that changes are only applied after you restart Tomcat.

The source of WSRP4J contains the file in the `WSRP4J_ROOT/provider/pluto/war/WEB-INF/data` directory. This file is copied when the Pluto provider is being deployed.

Let’s take a look at the sample `portletentityregistry.xml` delivered with WSRP4J:

10.1 Example Producer Portlet Entity Registry

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-entity-registry>

  <application id="0">                                     | #1
    <definition-id>wsrpctest</definition-id>
    <portlet id="1">                                       | #2
      <definition-id>wsrpctest.WSRPTestPortlet</definition-id>
    </portlet>
    <portlet id="2">                                       | #3
      <definition-id>wsrpctest.WSRPTestPortlet</definition-id>
    </portlet>
  </application>
  <application id="99">                                     | #4
    <definition-id>wsrpctest</definition-id>
    <portlet id="1">                                       | #5
      <definition-id>wsrpctest.WSRPTestPortlet</definition-id>
    </portlet>
    <portlet id="2">                                       | #6
      <definition-id>wsrpctest.WSRPTestPortlet</definition-id>
    </portlet>
  </application>

</portlet-entity-registry>
(annotation)<#1 application “0” definition, consists of two portlets
(annotation)<#2 portlet “1” definition
(annotation)<#3 portlet “2” definition
(annotation)<#4 application “99” definition, consists of two portlets
(annotation)<#5 portlet “1” definition
(annotation)<#6 portlet “2” definition
```

You have already seen this file, do you remember? Right, we've seen this file when we were discussing deployment of applications in the Pluto portal in chapter 1. So, this one is similar. All we do is to define application entities which in turn contain portlet definitions.

In our example we define two applications, one with the id "0" and one with the id "99". As you will see later, when we will be discussing the setup of Consumers, the application id is directly reflected in the generated portlet handles used to address a portlet on our Producer.

So what were those definition ids for? The definition-id of the application corresponds to the web application name in which our portlet resides. In our case this was "wsrpctest". The definition id of a portlet is the concatenation of the web application name and the portlet name defined in the `portlet.xml`. If we looked in this file we would see that our test portlet is named "WSRPTestPortlet".

So why do we have the same portlet exposed four times as a WSRP service? The answer to this is simple: for demonstration and test purposes. If you remember, we mentioned session sharing between portlets within one application. This is exactly what we showcase with here. Each of the applications ids is exposed as a groupID in the WSRP protocol. All portlets with the same group id share the session in the WSRP4J implementation. So as a result we should see that portlet "1" and "2" of application "0" will share the session. The same applies to the portlets in application "99".

After we've configured our Producer in this way, we're ready to go. Our Producer is ready for interaction. You can now start the Tomcat server.

Setting up the Swing Consumer

The Swing Consumer is a sample application that demonstrates the WSRP protocol and is useful for test purposes. It is a WSRP Consumer and a simple HTML Browser in one application.

Building and Installing the Swing Consumer

The build and installation of the Java Swing Consumer is straightforward. All you need to do is to run the `install-swing-consumer.bat/sh` script or directly invoke the Ant task "install-swing-consumer" by passing it as a parameter to the build script.

The build and installation results in a new directory called `WSRP4J_ROOT/driver/SwingConsumer` where you will find start scripts, configuration files and the JAR files.

Configuring the Swing Consumer

Prior to starting the Swing Consumer we need to configure it to display the remote portlets on its pages. The WSRP4J source code comes with a pre-configured installation which integrates the four portlet instances we provided with the WSRP4J Pluto provider in the last section. In this section, we will discuss the Swing Consumer's configuration files to give you a clue how things work.

The Swing Consumer's configuration can be found in the persistence subdirectory. You can find this directory in the Swing Consumer source tree at `WSRP4J_ROOT/SwingConsumer/persistence` or once in the SwingConsumer's installation root `WSRP4J_ROOT/driver/SwingConsumer/persistence`. The configuration is kept in three subdirectories producers, portlets and pages.

```
••SwingConsumer
• ••classes
• ••persistence
•   ••producers
•   ••portlets
```

- ••pages

Producers

The `producers` directory contains definitions and properties of WSRP Producers from which portlets shall be integrated. There is one file for each remote Producer. The files need to follow a specific naming convention which is used by the persistence layer.

Each file must start with the prefix

`"org.apache.wsrp4j.consumer.driver.ProducerImpl@"`

and must end with the suffix `".xml"`. Any other valid file name characters may appear in between these two. Typically you would assign a Producer name here.

The structure of a Producer definition file is as follows:

10.2 Example Producer Definition

```
<?xml version="1.0"?>
<Producer id="1" registrationRequired="false" > | #1

    <markup-interface-url>http://localhost:8081/wsrp/wsrp4j/WSRPBaseService</markup-
interface-url> | #2
    <service-description-interface-
url>http://localhost:8081/wsrp/wsrp4j/WSRPServiceDescriptionService</service-
description-interface-url> | #3
    <registration-interface-
url>http://localhost:8081/wsrp/wsrp4j/WSRPRegistrationService</registration-interface-
url> | #4
    <portlet-management-interface-
url>http://localhost:8081/wsrp/wsrp4j/WSRPportletManagementService</portlet-
management-interface-url> | #5

    <registration-data> | #6
        <consumer-name>WSRP4J Swing Consumer</consumer-name>
        <consumer-agent>WSRP4J Swing Consumer V. 0.1</consumer-agent>
    </registration-data>

</Producer>
(annotation)<#1 producer "1" definition
(annotation)<#2 URL of markup interface
(annotation)<#3 URL of service description interface
(annotation)<#4 URL of registration interface
(annotation)<#5 URL of portlet management interface
(annotation)<#6 registration data to be sent to that particular producer
```

In general this file contains the end-point URLs of each Port the Producer exposes (lines 4-7). In our example the URLs point to our previously deployed Pluto provider. You may have noticed that the port is set to 8081 rather than 8080. The WSRP4J default to a SOAP monitor port which listens on port 8081 and forwards the HTTP traffic to port 8080. This can be used to monitor and debug the SOAP messages exchanged between the Consumer and Producer. You can safely change the port to your Tomcat port if you do not intend to monitor the request and response messages.

Each Producer needs to be uniquely identified within the Swing Consumer environment. This is done by assigning a unique id for each Producer (line 2).

The properties for the registration data are properties the Swing Consumers uses to announce itself to the Producer.

Portlets

The `portlets` subdirectory contains portlet definitions of remote portlets which may be put on the Swing Consumer's pages. In general one file per remote portlet is kept. Similar to the Producer definitions, each file needs to follow a naming convention used by the persistence subsystem. The files must start with the "[org.apache.wsrp4j.consumer.driver.WSRPPortletImpl](#)@`" prefix and end with the ".xml" suffix. Any valid file name character is valid in between these two.`

Each portlet definition file has the following structure:

10.3 Example Portlet Definition

```
<?xml version="1.0"?>
<Portlet>
  <portlet-key>                                | #1
    <portlet-handle>0.1</portlet-handle>        | #2
    <producer-id>1</producer-id>                | #3
  </portlet-key>
  <parent-handle>0.1</parent-handle>           | #4
</Portlet>
(annotation)<#1 portlet key, consisting of portlet handle and the prducer id hosting the portlet
(annotation)<#2 portlet handle
(annotation)<#3 producer id, providing the portlet
(annotation)<#4 the producer offered handle of the portlet
```

Basically this file defines a portlet key. The key consists of the Producer id of the Producer hosting the remote portlet and the portlet handle. Remember, we defined the Producer with the id "1" in the previous file.

What about the portlet handles? In general you need to obtain the portlet handles from the Producer offering remote portlets. The Pluto provider generates portlet handles for its "Producer offered Portlets" according to the following rule: The portlet handle is a concatenation (using "." as the separator) of the application id and the portlet id we defined in the `portletentityregistry.xml`.

The parent-handle should be the same as the portlet-handle when you integrate Producer offered Portlets. This parent handle is used by the Swing Consumer implementation once you clone (either explicitly or implicitly) portlets.

WSRP4J ships with four portlet definitions, each pointing to a remote portlet from the Pluto provider.

Pages

Finally we have the page definitions. The Swing Consumer can display various pages as tabbed panes. Each page is defined in a separate file in the `pages` subdirectory.

Similar to the other property files page definition files follow a naming convention. The files must start with the "[org.apache.wsrp4j.consumer.app.driver.PageImpl](#)@`" prefix and end with the ".xml" suffix. Any valid file name character is valid in between these two.`

Let's take a look at the page definition:

10.4 Example Swing Consumer Page Definition

```
<?xml version="1.0"?>
<page pageID="1" title="WSRP4J Test" >                                | #1
  <portlet-key xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="java:org.apache.wsrp4j.consumer.driver.PortletKeyImpl">    | #2
    <portlet-handle>0.1</portlet-handle>
    <producer-id>1</producer-id>
  </portlet-key>
```

```

    <portlet-key xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="java:org.apache.wsrp4j.consumer.driver.PortletKeyImpl">      | #3
    <portlet-handle>0.2</portlet-handle>
    <producer-id>1</producer-id>
</portlet-key>
    <portlet-key xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="java:org.apache.wsrp4j.consumer.driver.PortletKeyImpl">      | #4
    <portlet-handle>99.1</portlet-handle>
    <producer-id>1</producer-id>
</portlet-key>
    <portlet-key xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="java:org.apache.wsrp4j.consumer.driver.PortletKeyImpl">      | #5
    <portlet-handle>99.2</portlet-handle>
    <producer-id>1</producer-id>
</portlet-key>
</page>
(annotation)<#1 page definition
(annotation)<#2 key of portlet 1
(annotation)<#3 key of portlet 2
(annotation)<#4 key of portlet 3
(annotation)<#5 key of portlet 4

```

Each page is uniquely identified by a pageID and contains a title which is displayed on the tabbed pane. The page consists of portlet-key elements each referring to a portlet definition. Again the portlet key is an agglomeration of the portlet handle and the Producer id of the hosting Producer.

In the example above we define a page with four portlet windows on it, each pointing to one of the remote portlets we provided with the Pluto provide.

Running the Swing Consumer

Prior to starting the Swing Consumer, we start the TCP monitor which redirects the traffic through the monitor to the target host and port. You do not need to run the monitor in case you changed the Producer definition to point directly to the Producer instead of routing the traffic through the monitor.

WSRP4J ships the Apache Axis TCP monitor. You can start the monitor by invoking the `tunnel.bat/sh` script in the `WSRP4J_ROOT/tools` directory.

Now you can safely start the Swing Consumer by executing the `run.bat/sh` script in the Swing Consumer root directory.

When starting, the Swing Consumer connects to the Producer, registers itself with the Producer, sets up the HTTP cookies and retrieves markup from the remote portlets.

As a result you should see Figure 10.6 displayed.



Figure 10.6 Swing Consumer displays page containing 4 remote portlets serviced by the Pluto provider

The Swing Consumer displays one tabbed pane with the name “WSRP4J Test”. On this page you see four portlet windows integrating the remote portlets from our Producer. The decoration shows the remote portlet’s title and its handle.

Please note that at the time of writing this book the Swing Consumer didn’t support HTML forms. However, you can now play around and interact with the portlets and the window decorations to discover the world of WSRP.

Once you finished exploring the Swing Consumer let’s take the next step. In the next section we will set up the Proxy Portlet. We will deploy it into the Apache Pluto portal and thus enable the portal to consume WSRP services.

Setting up the Proxy Portlet

The build and installation process of the proxy portlet is similar to other portlets we deploy in the Producer environment. Remember that in contrast to the other portlets the proxy portlet is installed as a Consumer component running in the Apache Pluto portal. We therefore assume that you have installed the Pluto portal accordingly. You can refer to the Pluto installation instructions in chapter 7.

Building and Installing the Proxy Portlet

To include the proxy portlet into the WSRP4J build and deployment process we first need to add a include file into the WSRP4J_ROOT/portlets/build directory. You can do so by simply creating a new file named `proxyportlet.include`.

The second step is to modify the `build.properties` of the proxy portlet. Similar to the other portlets this file provides properties holding the target Tomcat directory and the web application name running a portlet container to which we deploy the portlet.

WSRP4J comes with a sample file called `build.properties.example`. Copy the file to `build.properties` and edit the contents of the new file in the WSRP4J_ROOT/portlets/proxyportlet directory:

```
// the name of the target jar file
jarfile=proxyportlet.jar
// the name of the target war file
warfile=proxyportlet.war
// the tomcat dir we want this portlet to be deployed to
TARGET_TOMCAT_HOME=D:/Apache/Tomcat
// the webapp containing the portlet container we use for deployment
// for proxyportlet it's the pluto installation
TARGET_WEBAPP_NAME=pluto
```

You need to change the variable value of `TOMCAT_HOME` to the directory you installed the Apache Tomcat instance running your Pluto portal. Note that we deploy the proxy portlet into the Pluto portlet container not the WSRP4J Pluto provider container. Therefore you must set the `TARGET_WEBAPP_NAME` properly. By default the Pluto portal is installed in the web application “pluto”. You can leave the other values as they are.

To build the proxy portlet, change back to the WSRP4J_ROOT/build directory and run the build script with the target “`build_portlets`”.

```
build.bat build_portlets
```

After the script has finished you should again see a “build successful” message on your screen.

Portlet deployment is as easy as the build task. Simply run the build script with the “`deploy_portlets`” parameter.

```
build.bat deploy_portlets
```

Again a “build successful” message should indicate to you the proper deployment. Now the proxy portlet should be deployed and ready for use in the Pluto portal.

Configuring the Proxy Portlet

Before we can use the proxy portlet, we must configure it accordingly. Basically all we need to do is to define a portlet entity for each remote portlet we want to use in Pluto’s `portletentityregistry.xml`. We’ve talked about this file in detail already in chapter 1 when we developed our own portlet.

Portlet Entity

WSRP4J provides a sample portlet entity registry file in the WSRP4J_ROOT/portlets/proxyportlet/war/examples directory. You can either copy the file into Pluto's runtime configuration (and thus delete other portlet entity definitions) or add the contents of the sample file to Pluto's portlet entity registry.

The example file contains the following portlet definitions:

10.5 Sample Portlet Entity Registry

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-entity-registry>
  <!-- wsrp4j wsrp proxy portlet -->
  <application id="99"> | #1
    <definition-id>proxyportlet</definition-id>
    <portlet id="1"> | #2
      <definition-id>proxyportlet.ProxyPortlet</definition-id>
      <!-- portlet -->
      <preferences>
        <pref-name>wsrp_portlet_handle</pref-name>
        <pref-value>0.1</pref-value>
        <read-only>false</read-only>
      </preferences>
      <!-- producer -->
      <preferences>
        <pref-name>wsrp_producer_id</pref-name>
        <pref-value>1</pref-value>
        <read-only>false</read-only>
      </preferences>
    </portlet>
  <portlet id="2"> | #3
    <definition-id>proxyportlet.ProxyPortlet</definition-id>
    <!-- portlet -->
    <preferences>
      <pref-name>wsrp_portlet_handle</pref-name>
      <pref-value>0.2</pref-value>
      <read-only>false</read-only>
    </preferences>
    <!-- producer -->
    <preferences>
      <pref-name>wsrp_producer_id</pref-name>
      <pref-value>1</pref-value>
      <read-only>false</read-only>
    </preferences>
  </portlet>
</application> | #4
<application id="98">
  <definition-id>proxyportlet</definition-id>
  <portlet id="1"> | #5
    <definition-id>proxyportlet.ProxyPortlet</definition-id>
    <!-- portlet -->
    <preferences>
      <pref-name>wsrp_portlet_handle</pref-name>
      <pref-value>99.1</pref-value>
      <read-only>false</read-only>
```



```

    </preferences>
    <!-- producer -->
    <preferences>
        <pref-name>wsrp_producer_id</pref-name>
        <pref-value>1</pref-value>
        <read-only>>false</read-only>
    </preferences>
</portlet>
<portlet id="2"> | #6
    <definition-id>proxypportlet.ProxyPortlet</definition-id>
    <!-- portlet -->
    <preferences>
        <pref-name>wsrp_portlet_handle</pref-name>
        <pref-value>99.2</pref-value>
        <read-only>>false</read-only>
    </preferences>
    <!-- producer -->
    <preferences>
        <pref-name>wsrp_producer_id</pref-name>
        <pref-value>1</pref-value>
        <read-only>>false</read-only>
    </preferences>
</portlet>
</application>
</portlet-entity-registry>
(annotation)<#1 application "99" definition
(annotation)<#2 portlet "1" definition, proxy portlet pointing to remote WSRP portlet
(annotation)<#3 portlet "2" definition, proxy portlet pointing to remote WSRP portlet
(annotation)<#4 application "99" definition
(annotation)<#5 portlet "1" definition, proxy portlet pointing to remote WSRP portlet
(annotation)<#6 portlet "2" definition, proxy portlet pointing to remote WSRP portlet

```

You should already be familiar with the structure of this file. Proxy portlet entity definitions are similar to any other portlet. But there is one addition we use for proxy portlet: portlet preferences. As you know, the proxy portlet acts as a placeholder for remote portlets in a local environment. What we need to do is to add a pointer to the remote portlet in the proxy portlet's configuration.

Portlet Preferences

WSRP4J utilizes portlet preferences for this purpose. We define two preferences for each portlet. First, the preference named "wsrp_portlet_handle" holds the remote portlet's handle the proxy portlet dispatches to. Second, the preference "wsrp_producer_id" holds the id of the Producer hosting the portlet – we will come in a few seconds to where this id is defined.

Let's take a look at the sample file above. We define an application with the id "99" containing two proxy portlets with the ids "1" and "2". Both proxy portlets connect to the Producer with the id "1". Proxy portlet "1" interacts with the remote portlet identified by the portlet handle "0.1" while proxy portlet "2" communicates to the remote portlet represented by the portlet handle "0.2".

Note that the portlet preferences are not flagged as "read-only". This is needed because the portlet handle might change when we interact with the remote portlets. The proxy portlet utilizes WSRP's clone before write behavior and starts to use Producer offered Portlets first. Once a persistent state change is needed on the remote portlet the Producer will clone the portlet and return a new handle. This new handle is stored in the portlet preferences persistently.

Connecting to the Producer

The next step is to define the Producer we want to connect to. This setup is similar to the Producer definitions we saw for the Swing Consumer. You can find the configuration directory in `TOMCAT_HOME/webapps/proxyportlet/WEB-INF/persistence`. There is only one subdirectory “producers” storing the producer configurations.

The same producer configuration file we saw for the Swing Consumer can be found here. It points to our deployed WSRP4J Pluto provider. Note that we use the HTTP port 8081 in the URLs to monitor the traffic in the SOAP monitor. You can safely change the port to 8080 (default Tomcat port) or to whatever port you configured on your Tomcat installation.

Adding Proxy Portlet Instances to Page

The final step is to add our proxy portlet instances on a Pluto portal page. We already described the structure of the page registry in chapter 1. The following Listing 9.13 shows the example page definition shipped with WSRP4J. Note that this listing contains the full definition of a page. Of course you can just add the page fragment definition to your existing Pluto portal pages.

10.6 Example Page Registry

```
<?xml version="1.0" encoding="UTF-8"?>
<portal>
  <fragment name="navigation"
class="org.apache.pluto.portalImpl.aggregation.navigation.TabNavigation">
    </fragment>

    <fragment name="ProxyTest" type="page"> | #1
      <navigation>
        <title>ProxyTest</title>
        <description>This page contains wsrp4j proxyportlets</description>
      </navigation>

      <fragment name="row1" type="row"> | #2
        <fragment name="col1" type="column">
          <fragment name="p1" type="portlet"> | #3
            <property name="portlet" value="99.1"/>
          </fragment>
          <fragment name="p2" type="portlet"> | #4
            <property name="portlet" value="99.2"/>
          </fragment>
        </fragment>
      </fragment>

      <fragment name="row2" type="row"> | #5
        <fragment name="col1" type="column">
          <fragment name="p1" type="portlet"> | #6
            <property name="portlet" value="98.1"/>
          </fragment>
          <fragment name="p2" type="portlet"> | #7
            <property name="portlet" value="98.2"/>
          </fragment>
        </fragment>
      </fragment>
    </fragment>
  </portal>
(annotation)<#1 page definition
```

(annotation)<#2 row definition
 (annotation)<#3 column definition holding portlet “99.1”
 (annotation)<#4 column definition holding portlet “99.2”
 (annotation)<#5 row definition
 (annotation)<#6 column definition holding portlet “98.1”
 (annotation)<#7 column definition holding portlet “98.2”

The setup is straightforward. We define a page with the id and title “ProxyTest” and add two rows with two columns each on the page. Each cell contains one of the proxy portlet instances we defined in our portlet entity registry. You can either completely replace Pluto’s page registry with this file or add the page definition to it.

Running the Proxy Portlet

Now we are ready to try out the proxy portlet. Remember to start the TCP monitor if you didn’t change the HTTP port in the example files provided with WSRP4J. Simply start the Apache Tomcat instance where you installed Pluto and point your browser to the Pluto portal. Typically the URL is “<http://localhost:8080/pluto/portal/>”. You should see the Pluto initial page with a navigational link to the “ProxyTest” page. Click on the link and the page containing the four portlet windows should be displayed (Figure 10.7):

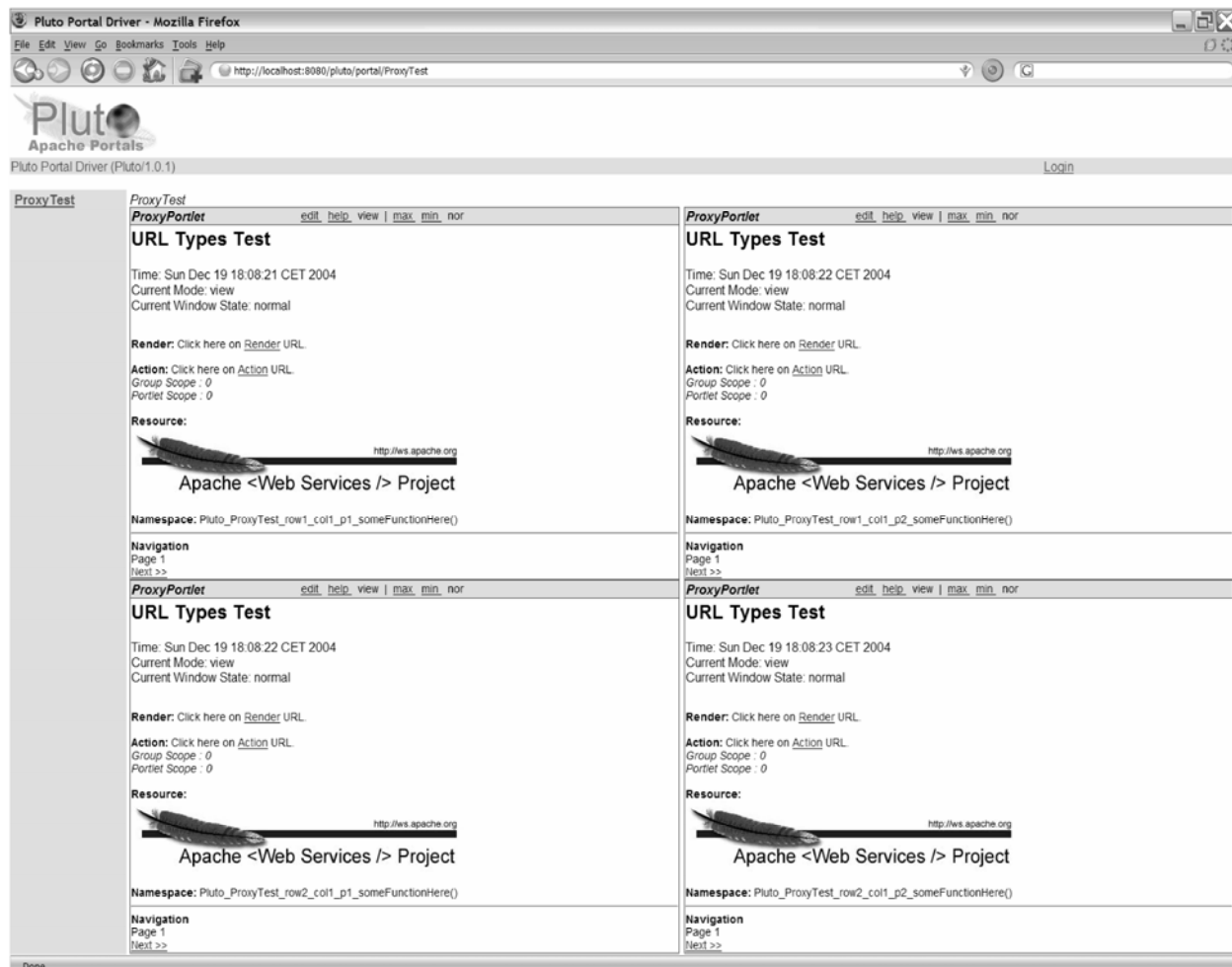


Figure 10.7 Proxy Portlet on Pluto Portal

You are now ready to interact with the remote portlets.

Troubleshooting

The WSRP4J community is growing and things evolve quite fast. If you intend to use the latest WSRP4J code instead of our packaged snapshot and encounter problems, the best thing to do is to request the community's help. You can do this by subscribing to the WSRP4J mailing lists.

Currently there are two mailing lists:

wsrp-user@ws.apache.org: This list is for developers and users that are using WSRP4J in their own projects to ask questions, share knowledge, and discuss issues related to using wsrp4j. You can easily subscribe by simply sending an email to wsrp4j-user-subscribe@ws.apache.org

wsrp-dev@ws.apache.org: This is the list where participating developers of the wsrp4j project meet and discuss issues, code changes/additions, etc. You can easily subscribe by simply sending an email to wsrp4j-dev-subscribe@ws.apache.org

Summary

In this chapter, we gave you a guided tour of one of the first implementations of the WSRP protocol: the Apache WSRP4J project. We gave you an overview of the architecture and showed you how to build and install the various components of the project. With the Producer and pluggable provider concept we showed you how in general UI components can be exposed as WSRP services. We discussed the Pluto provider as a means to provide JSR168 portlets as remote portlets. The Swing Consumer acts as a sample test client to demonstrate the protocol and test the underlying implementation. The Pluto proxy portlet however is more sophisticated and shows how JSR168 conformant portals can be extended to be able to integrate remote portlets into a portal installation.

The intent here was to give you a starting point into WSRP, provide you with a first impression of what is possible, and, finally, stimulate your imagination and interest for WSRP.

Now that we have covered the various portlet related projects at Apache and gave you an overview of the portal world, we will broaden our view in the next chapter. We will now discuss how to bridge between the existing web technologies like JSPs, Java Server Faces (JSF), Struts and the portal technology.

References

- Web Services for Remote Portals Technical Committee: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrp.
7. WSRP 1.0 standard specification document: <http://www.oasis-open.org/committees/download.php/3343/oasis-200304-wsrp-specification-1.0.pdf>.
 8. Jakarta Tomcat: <http://jakarta.apache.org/tomcat>
 9. WSRP Primer 1.0: <http://www.oasis-open.org/committees/download.php/10539/wsrp-primer-1.0.html>
 10. Apache WSRP4J project homepage: <http://ws.apache.org/wsrp4j/>
 11. Apache Pluto project homepage: <http://portals.apache.org/pluto/>

Chapter 11 Developing Portlets Using Existing Web Frameworks

Until now, we have covered the different portal projects hosted at Apache Portals in Part 3 of this book. Now we come to a subproject of Apache Portals called *Bridges*. Portals Bridges are independent Java components and portlets that bridge existing web frameworks with the new world of Java portlets. Because these bridges are designed to run on any JSR 168 compliant portal, you can use them on your specific portal. How to use bridges on different portals will be covered in the first section of this chapter.

Next we will cover several prominent web technologies and frameworks used in the servlet world, such as JSPs, JavaServer Faces (JSF), Struts, and Velocity.

Next we will cover Struts in detail and take an existing Struts application and convert the application into a portlet Struts application step-by-step. Finally, we will cover the Spring framework and also walk through an example of using Spring for portlet development. before we start looking at further web technologies.

Bridging two worlds: the Web and Java portlets

Portals Bridges are portlet application frameworks for bridging from portlet applications to specific web application technologies. The Bridges project supports a growing number of portal bridges, including JSP, JSF, Struts, PHP, and Perl. Each of these bridges provides a foundation for developing portlet applications with a web application technology. These portlet applications can in turn have their own Model-View-Controller (MVC) frameworks, such as Struts portlet applications, JSF portlet applications, or Turbine portlet application frameworks. There is a clean separation of concerns. The Jetspeed portal (or any other portal) does not contain any bridge-specific code. In this section, we will demonstrate:

- Developing a Portlet with the JSP Bridge
- Developing a Portlet with the JSF Bridge
- Developing a Portlet with the Struts Bridge
- Developing a Portlet with the Velocity Bridge
- Developing a Portlet with Spring and Portal Bridges

Developing portlets using the JSP Bridge

To make servlet and JSP development easier, the Apache Portals Bridges project comes with a little helper class called the Generic Servlet Portlet. This helper class, shown below, extends the *GenericPortlet* class from the Portlet API.

```
public class GenericServletPortlet extends GenericPortlet
```

Perhaps the most important way that *GenericServletPortlet* helps you with developing JSP and servlet-based portlets, is that you don't always have to write your boilerplate *doView*, *doEdit* and *doHelp* methods. Within portlets these methods are required for every basic portlet to work properly. The *GenericServletPortlet* base class handles this for you automatically and therefore eliminates the need to manually write these methods over and over again within a servlet.

To build and compile the JSP Bridge, a portal must have the common jar of Apache Portals Bridges. Certain servers (including Jetspeed) already contain the common jar, and therefore it would not be required. The Maven configuration file, *project.xml*, has a dependencies section. Here we have declared a dependency on the Portals Bridges common jar. Depending on which server this application should run, we need to include the jar file. By setting the *war.bundle* to false, we are not deploying the jar file into this portlet application. With Jetspeed, this works fine since Jetspeed comes with this required jar.

```
<dependency>
<id>portals-bridges-common</id>
<groupId>portals-bridges</groupId>
<version>0.1</version>
<properties>
<war.bundle>>false</war.bundle>    |#1
</properties>
</dependency>
```

(Annotation) <#1 This setting defines whether the jar file should be packaged with the application

In the deployment descriptor, you can specify the names of the JSP:

Listing 11.1: Portlet Init Parameters

```
<init-param>
  <description>This parameter sets the JSP
    used in view mode.</description>
  <name>ViewPage</name>
  <value>/WEB-INF/view/edit.jsp</value>
</init-param>
<init-param>
  <description>This parameter sets the JSP
    used in edit mode.</description>
  <name>EditPage</name>
  <value>/WEB-INF/view/view.jsp</value>
</init-param>
<init-param>
  <description>This parameter sets the JSP
    used in help mode.</description>
  <name>HelpPage</name>
  <value>/WEB-INF/view/help.jsp</value>
</init-param>
```

The benefit is that your code no longer has to dispatch to the JSP. In fact, in this case, we have no need to write the `doView`, `doEdit` and `doHelp` methods. *GenericServletPortlet* handles the dispatching for you. The following listing demonstrates how little code we need to create a full-featured portlet using the JSP Bridge. By extending, we actually added functionality (help), yet the code shrank substantially. In this example we created the `processAction` method to handle adding and deleting bookmark entries.

Listing 11.2: Bookmark Portlet 2 with Help

```
public class BookmarkPortlet2 extends GenericServletPortlet
{
    public void processAction (ActionRequest request,
        ActionResponse actionResponse)
        throws PortletException, java.io.IOException
    {
        String removeName = request.getParameter("remove");
        if (removeName!=null)
        { // remove
            PortletPreferences prefs = request.getPreferences();
            prefs.reset(removeName);
            prefs.store();
        }
        String add = request.getParameter("add");
        if (add!=null)
        { // add
            PortletPreferences prefs = request.getPreferences();
            prefs.setValue(request.getParameter("name"),
                request.getParameter("value"));
            prefs.store();
        }
    }
}
```

Figure 11.1 shows the results of this code.

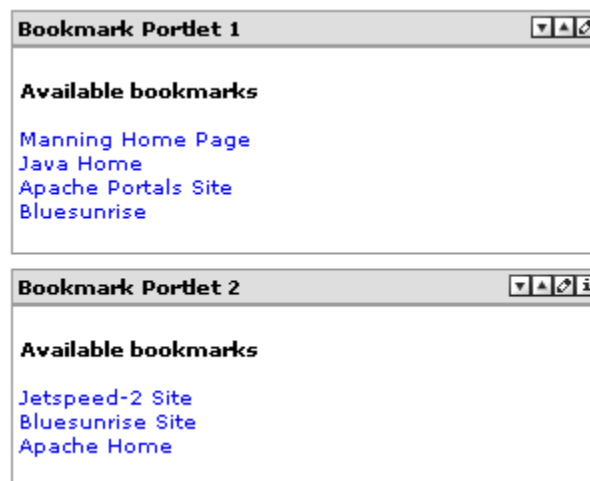


Figure 11.1 This portlet includes the help button

In this section we learned how to use the JSP Bridge and how little code is required to create full-featured portlets. Now let's look at how to develop with Apache Bridges and JSF.

Developing portlets using the JSF Bridge

JavaServer Faces (JSF) is a Java standard and *user interface* (UI) framework for Java web applications. It is designed to significantly ease the burden of writing and maintaining web applications. The JSF Bridge comes along with Jetspeed and can leverage nearly every JSF implementation available. For more background information about JSF and JSF portlets please see Chapter 5 in this book.

Keeping with the theme of Apache projects, we make use of the Apache My Faces JSF project in this chapter to create a simple but yet powerful calendar portlet. My Faces is a compliant JSF implementation developed at Apache, providing a rich set of components such as a tree view, a calendar and advanced data view components. In our introductory example here, we'll develop the calendar portlet displayed in Figure 11.2..

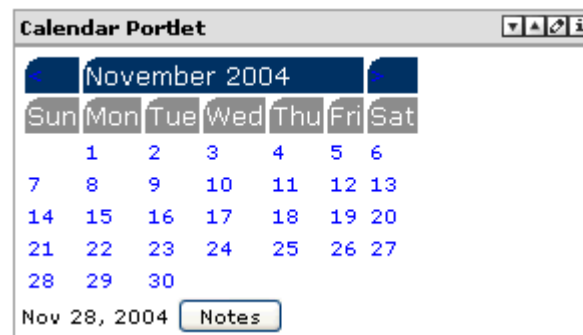


Figure 11.2 The Calendar Portlet written with the JSF My Faces Bridge

With this sample portlet, you can view a simple calendar, and add notes for a given day. Since our portlet is just a sample to introduce you to using the JSF Bridge, we do not store the notes to permanent storage. We'll leave that as an exercise for the reader.

One thing you might notice about the code used to write the Calendar portlet: most of the code is not written to the Portlet API! It is all standard JSF code. The bridge handles translating portlet messages and actions to the equivalent JSF actions. The JSF Bridge is a great way for JSF programmers to become immediately productive writing portlet applications. The JSF Bridge does not stop you from writing to the Portlet API. You are free to write as much Portlet API code by extending the base bridge class. However, it is not required. Existing JSF applications can be run inside a Portlet API compliant portal by including the bridge inside your portlet application. Let's see what it takes to get started. First, look at how to bring in the dependencies required by My Faces. As shown below, we simply specify an external XML entity (for My Faces) in the Maven *project.xml* dependencies section, and bring in all the dependencies from another XML file provided in the distribution. In this book, we often use this approach to nicely encapsulate all dependencies for a particular framework:

```
<!ENTITY % locator-entities SYSTEM "file:locator.ent"> %locator-entities;  
...  
&myfaces;
```


In the deployment descriptor, you can specify the JSF Bridge base portlet class, and the names of the JSP templates. The portlet.xml file, shown below, references the JSF Bridge base portlet class to define standard portlet views

```
<portlet-class>
  org.apache.portals.bridges.myfaces.FacesPortlet</portlet-class>
<init-param>
  <name>ViewPage</name>
  <value>/WEB-INF/view/chapter8/calendar-view.jsp</value>
</init-param>
<init-param>
  <name>HelpPage</name>
  <value>/WEB-INF/view/chapter8/calendar-help.html</value>
</init-param>
<init-param>
  <name>EditPage</name>
  <value>/WEB-INF/view/chapter8/calendar-edit.jsp</value>
</init-param>
```

The *web.xml* file has a number of JSF-specific configuration parameters that we will not discuss in great detail here. One important parameter from the *web.xml*, however, is the name and location (?) of the JSF configuration file.

```
<context-param>
  <param-name>javax.faces.application.CONFIG_FILES</param-name>
  <param-value>/WEB-INF/myfaces/faces-config.xml</param-value>
</context-param>
```

The JSF configuration file contains our settings for JSF Application Beans and JSF Navigations. Shown below is a definition of a navigation rule from the Calendar view to the Notes view:

```
<navigation-rule>
  <from-view-id>/WEB-INF/view/chapter8/calendar-view.jsp</from-view-id>
  <navigation-case>
    <from-outcome>editNotes</from-outcome>
    <to-view-id>/WEB-INF/view/chapter8/calendar-notes.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Navigation rules define navigations paths based on action outcomes to navigate inside of your portlet. JSF has the concept of views, which are really just JSF pages. In the JSF configuration seen above, we define a navigation from the main Calendar view to the Notes view. This navigation occurs whenever someone presses the Notes button.

Encapsulation of application navigation into a centralized configuration greatly reduces the maintenance and complexity of your JSF application. An action is attached to the Notes button. When that action returns, it returns the name of the abstract view. In the case above, the action target is called: *editNotes*. Actions can also go to different navigations based on runtime parameters (not shown in our examples). Here is the Notes view of the same Calendar portlet:

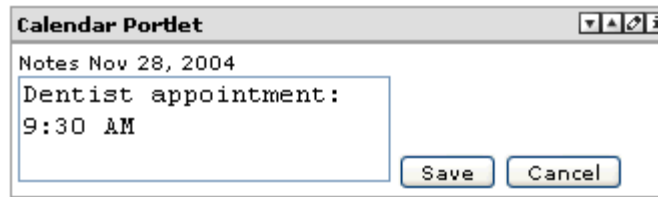


Figure 11.3 The Calendar Portlet, Notes view

The notes screen similarly uses actions and navigation rules to cancel or save and then return to the main calendar page. From the code above, you simply reverse the `<from-view-id>` and `<to-view-id>` line references, and change the `<from-outcome>` line to `returnFromNotes`

Creating our calendar portlet

Now let's take a look at the actual JSF components and beans used to create our portlet. If you are not familiar with JSF please review Chapter 5 first.

Calendar view

Here is the first JSF page that you see in the portlet's view mode, the calendar view:

```
<f:view>
  <h:form id="calendarForm">
    <x:inputCalendar value="#{calendar.date}"
      monthYearRowClass="portlet-section-header"
      weekRowClass="portlet-section-subheader"
      dayCellClass="portlet-menu-item"
      currentDayCellClass="portlet-menu-item-selected"
    />
    <h:outputText value="#{calendar.date}" />
    <h:commandButton id="edit"
      value="#{MESSAGE['calendar.notes']}"
      action="#{calendar.selectDate}" />
  </h:form>
</f:view>
```

The above code contains JSF tags and expressions. Note that JSF expressions are evaluated with the `#{}` syntax. Thus the expression `#{calendar.date}` evaluates to the calendar session bean defined in this JSF configuration file:

```
<managed-bean>
  <managed-bean-name>calendar</managed-bean-name>
  <managed-bean-class>
    com.manning.enterpriseportals.portlets.chapter8.CalendarBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

The tag `<x:inputCalendar` defines the My Faces component. The component handles rendering itself. The state of the component is stored in our managed session bean. Also take note that we use standard Portal cascading styles sheets as defined in the Portlet specification:

```

<x:inputCalendar value="#{calendar.date}"
  monthYearRowClass="portlet-section-header"
  weekRowClass="portlet-section-subheader"
  dayCellClass="portlet-menu-item"
  currentDayCellClass="portlet-menu-item-selected"
/>

```

The four CSS classes that define the styles for different parts of the rendered calendar are styles defined in the portal. Thus the calendar will automatically pick the style (skin) of the current portal page where it is placed.

Take a look at the Notes command button. When the Notes button is pressed, an action on the Calendar application bean is invoked from:

```
action="#{calendar.selectDate}
```

Calling the action as a Java method on the Calendar bean, the *selectDate* method:

Listing 11.3: The *selectDate* method

```

public String selectDate()
{
    if (this.date == null)
    {
        return "editNotes";
    }
    String selectedDate = getDateKey(this.date);
    PortletRequest request =
        (PortletRequest)FacesContext.getCurrentInstance().
        getExternalContext().getRequest();
    notes = request.getPreferences().getValue(selectedDate, "");
    if (notes == null)
    {
        notes = "";
    }
    return "editNotes"; // goto the navigation rule named "editNotes"
}

```

The *selectDate* method gets the portlet preferences, as shown on line 11 above, keyed on the selected date. Each date can hold a notes string in the portlet preferences database. The portlet preferences are stored per user / per portlet window. The portlet preferences easily facilitate the requirement for one calendar per user. On line 16, we return a string “*editNotes*”, which is a JSF navigation telling JSF to navigate to a logical view called “*editNotes*”, which is where we can edit the contents of the Note for a given date.

Here is the *getDateKey* method which formats the date key in a simple year/month/day format:

```

public String getDateKey(Date date)
{
    SimpleDateFormat formatter =
        new SimpleDateFormat ("yyyy-MM-dd", Locale.getDefault());
    return formatter.format(date);
}

```

A JSF application bean contains all the logic and state for our portlet. The bean is a plain old Java object, (POJO), a Calendar class with getters and setters that JSF uses automatically when processing web forms:

Listing 11.4: The Calendar managed bean

```
public class CalendarBean
{
    private Date date = new Date();
    private String notes = "";

    public Date getDate()
    {
        return date;
    }
    public void setDate(Date date)
    {
        if (date != null)
        {
            this.date = date;
        }
    }
    public String getNotes()
    {
        return notes;
    }
    public void setNotes(String notes)
    {
        this.notes = notes;
    }
}
```

This finishes the first view of our JSF Portlet including all configurations. The next section describes how we can add additional sections very easily.

Notes view

The Notes View is implemented as another JSF page and is based on a standard JSF multi-line text input:

```
<h:form id="calendarForm">
    <h:inputTextarea value="#{calendar.notes}" />
    <h:commandButton id="save" value="#{MESSAGE['add']}"
        action="#{calendar.save}" />
    <h:commandButton id="cancel" value="#{MESSAGE['cancel']}"
        action="returnFromNotes" immediate='true'>
    </h:commandButton>
</h:form>
```

Again, command buttons are hooked into actions, and the form is processed and saved:

Listing 11.5: The save method on the Calendar bean

```

public String save()
{
    If (this.date != null)
    {
        PortletRequest request =
            PortletRequestFacesContext.getInstance().
            getExternalContext().getRequest();
        PortletPreferences prefs = request.getPreferences();
        try
        {
            prefs.setValue(getDateKey(this.date), this.notes);
            prefs.store();
        }
        catch (Exception e)
        {
            System.err.println("error storing prefs " + e);
        }
    }
    return "returnFromNotes";
}

```

The save method stores the notes for a particular date in the Portlet Preferences for the current user. The preferences database is a standardized storage area for portlets to generically store information associated with a user and portlet instance. Here the data is stored unique to this user, the unique portlet instance, and the specified date.

Best practices for the JSF Bridge

As you can see, JSF makes web application development very easy. You don't have to worry about mapping HTTP parameters to Java objects. The JSF Bridge makes JSF portlet development seem the same as developing JSF servlets. To the application programmer, it should behave no differently.

When porting a JSF web application to portlet application, remember to follow these rules:

12. Add the JSF Bridge portlet to all of your portlet definitions in the *portlet.xml* descriptor (or a derivative class of the JSF Bridge class (*FacesPortlet*))
13. Define your default portlet view classes (you can have many more) for the portlet's supported modes: (*ViewPage*, *EditPage*, *HelpPage*) in your portlet descriptor.
14. Define any other relevant information in your portlet descriptor such as preferences, init parameters, supported languages, resources, mime types, and application user attribute
15. Modify your JSF source to adhere to PLT.B of the Portlet Specification (Markup Fragments): remove all base, body, *iframe*, frame, frameset, head, html and title tags.
16. Make use of the portlet specifications PLT.C Style Definitions to make your application better fit the style of the portal.

Next let's take a look at another bridge that allows applications developers to write portlet applications using an existing web framework: the Struts Portal Bridge.

Developing portlets using the Struts Bridge

Struts is a popular open source framework for building Java web applications. Struts promotes structured development and best practices of Java servlet applications. It is probably the best known example of the classic MVC application architecture (design pattern). Struts provides the controller and integrates with other technologies to provide the Model and the View. Typically the view is JavaServer Pages (JSP). The model can be *POJOs*, JDBC result sets, Enterprise Java Beans (EJB), or various object-relational technologies.

The Struts Portal Bridge tries to be as non-intrusive as possible and let programmers develop Struts portlet applications as if they were writing ‘normal’ Struts programs, which were traditionally servlets. However there are differences between servlets and portlets; thus minimal porting is required to make your Struts application run as a portlet application. For example, you will need to strip out all `<HTML>`, `<HEAD>`, and `<BODY>` tags from your *JSPs* to be compatible with the portlet API. Additional guidelines for porting servlet-based applications to portlets are provided in the appendix of this book.

Since there are many existing Struts applications, we are going to demonstrate how to take an existing Struts application and convert it to a portlet. Not every Struts application will run out-of-the-box. By following the steps that we’ll walk you through in this section, you should have a roadmap for getting Struts applications up in running in no time at all.

Converting a Struts Servlet Application to a Struts Portlet Application

Let’s begin by taking an existing Struts application and port it to a portlet. This application, the Struts Validation example, is provided by the Struts team as an example of forms and validation processing with Struts. We will walk through, step-by-step, the porting of this application to a portlet.

The Portlet Specification expects Portlets to use a true MVC architecture: changing the state of a Portlet is expected to be done during the action phase only. The render phase, which may occur many times, is expected to only render the current state of a portlet.

For a proper usage of the Struts Bridge and to conform to the Portlet Specification requirements, the Struts application must be configured to use separate Action processing and View rendering Actions.

Here are the general rules and restrictions for using the Struts Portlet Bridge:

Struts Portlet Bridge Rules and Restrictions

1. **Actions** – All user interaction must go through actions. Direct JSP access by the user will break the framework. This best practice is already recommended for any Struts application.
2. **Secure JSP Files** – Place all JSP files under the secured WEB-INF directory. This will guarantee that no JSP files can be accessed directly and will help in enforcing rule #1 above.
3. **No Rendering from an Action** – No direct output rendering from an action. All output should be rendered from an Action Forward after action processing. The Portlet API is a two phase API separating the action from render phase. To be compatible with the Portlet API, rendering in the action phase is forbidden. Again, this is a good Struts practice.
4. **Strip out the following HTML Tags:** `<base>`, `<body>`, `<iframe>`, `<frame>`, `<frameset>`, `<head>`, `<html>`, `<title>`
5. **Use the Enhanced Versions of Struts Bridges Tags** – There are a few tags that must be replaced in order to properly write back URLs to the portal. This is accomplished easily by with replacement TLD files that you include in your portlet application, requiring little or no change to the actual JSP code: `<html:form>`, `<html:link>`, `<html:rewrite>`, `<html:image>`, `<html:img>`

The remainder of this section will walk you through converting Struts applications to a Struts Portlet Application following the rules and regulations above, starting with the servlet deployment descriptor, the *web.xml*.

Converting the Deployment Descriptor (web.xml)

First we need to change the default Struts action servlet and replace it with the Bridges action servlet:

Table 11.1 shows the servlet definition before, using the default Struts action servlet, and after, where we replace the default servlet with the Struts Bridge action servlet.

Table 11.1 Changing Servlet Definitions with the Struts Bridge

The servlet definition before...

```
<!-- Action Servlet Configuration -->
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-
class>org.apache.struts.action.ActionServlet</servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  ...
</servlet>
```

the servlet definition after, using the Struts Bridge

```
<!-- Struts Bridge -->
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
    org.apache.portals.bridges.struts.PortletServlet
  </servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>
      /WEB-INF/struts-config.xml,
      /WEB-INF/struts-config-registration.xml
    </param-value>
  </init-param>
  ...
</servlet>
```

The Struts ActionServlet acts as the controller in the MVC architecture. It's the servlet entry point, binding together all other Struts components. The Struts action servlet is extensible. The Bridges PortletServlet extends the default Struts ActionServlet, overriding request, response and action processing to make Struts applications work properly in a portlet container. The Bridge also must 'bridge' the gap between typical servlets (such as Struts), having only a single request phase, and portlets, having a request phase and an action phase. Anyway, as a Struts developer you won't have to deal with those implementation details.

We've removed all of the tag library (TLD) requirements from the *web.xml*, along with all TLD files in the WEB-INF directory. JSP now supports putting the TLD files directly in the JAR file. Next we will look at how to access the TLD definitions without requiring entries in the *web.xml* or having copies of the TLD files in your WEB-INF directory (although placing the TLDs in the WEB-INF is still supported by the Struts Bridge).

Configuring the Portlet in the Portlet.xml

The portlet.xml file defines the portlet initialization parameters, default preferences, supported modes and MIME types, implementing class and display information. The parameters relevant for a Struts portlet are highlighted in the listing below:

Listing 11.6: portlet.xml, the portlet descriptor

```
<portlet id="StrutsValidatorPortlet">
  <init-param>
    <name>ServletContextProvider</name>
    <value>org.apache.jetspeed.portlet.ServletContextProviderImpl</value>
  </init-param>
```

```

<init-param>
  <name>ViewPage</name>                                |#1
  <value>/welcome.do</value>                             |#2
</init-param>
<portlet-name>StrutsValidatorPortlet</portlet-name>
<display-name>Struts Validator Portlet</display-name>
<description>This is the Struts Validator example servlet
  ported to a portlet</description>
<portlet-class>org.apache.portals.bridges.struts.StrutsPortlet</portlet-class>
|#3
<expiration-cache>-1</expiration-cache>
<supports>
  <mime-type>text/html</mime-type>
  <portlet-mode>VIEW</portlet-mode>
</supports>
<portlet-info>
  <title>Struts Validator Portlet</title>
  <short-title>Struts</short-title>
  <keywords>Struts, validator</keywords>
</portlet-info>
</portlet>

```

(Annotation) <#1 ViewPage is one of the keywords known by the Struts Bridge

(Annotation) <#2 /welcome.de is executed when the ViewMode is being rendered

(Annotation) <#3 Struts Portlet Bridge class name

First you need to set your portlet-class as *org.apache.portals.bridges.struts.StrutsPortlet*, or as a derivative of this class (1). The *StrutsPortlet* base class provides the Struts bridge support foundation for your portlet.

Next you need to define a few initialization parameters. As the Struts Portlet Bridge extends the *GenericServletPortlet* (defined in the preceding section in this chapter ‘Developing with the JSP Bridge’), the same three initialization parameters are supported here: *ViewPage*, *EditPage*, and *HelpPage*.

These three initialization parameters define the first view navigated to for each particular mode. From your respective view (View, Edit, Help), you can then navigate to other Struts pages (views) in your Struts portlet. These are just the initial entry points for when the user visits the portlet the first time during a session. In the case of our example, this portlet only supports View mode (1), not Help or Edit modes. The name of the default view for View mode is defined as */welcome.do* (2). This is a typical name for an action forward in Struts. Without getting into too much detail, action forwards are logical names, or views, to actual JSP pages. In this case it maps to *the index.jsp* template:

```

<action-mappings>
  <action path="/welcome" include="/WEB-INF/view/chapter8/struts/index.jsp" />

```

The *ServletContextProvider* initialization parameter defines a pluggable interface for interacting with portal implementations. This is a small piece of code required to make the Struts Bridge operate in different portals. Out of the box, the bridge works with the Jetspeed-2 portal.

Integration with other portals requires you to provide a replacement class that implements the interface *ServletContextProvider* and replace the entry in your portlet’s init parameter as shown above.

Replace Direct Usage of JSP templates with Actions to Logical Views or Forwards

This step fulfills the first requirement in the *Struts Portal Bridge Rules and Restrictions* section above. All user interaction must go through actions. Direct JSP access by the user breaks the framework. This best practice is already recommended for any Struts application today.

Let's take a look at the *index.jsp* file, where there is a direct link to a JSP page:

```
<html:link page="/registration.jsp">
```

We replace the above direct JSP tag with a Struts action equivalent:

```
<html:link action='/registration'>
```

Note that the *page* attribute has been dropped, as has the specific JSP filename *"/registration.jsp"*, replaced by a more accurately named *action* attribute, since it is now an action and not a page that we are referencing. The action value *"/registration"*, which we are creating a link to, is a logical view or action forward, not the actual name of the template. Abstracting JSP templates and replacing them with action forwards is now considered a best practice for all Struts applications. This leads into a more generalized requirement for porting to a Struts Portlet: replacing the usage of a few Struts Tags with the Struts Bridges equivalent tags. The next section covers doing just that. While we are replacing, we can also fix any links that do not use actions at the same time.

Use the Enhanced Struts Bridge JSP Tags

This step fulfills the fifth requirement in the *Struts Portal Bridge Rules and Restrictions* section above. Here we introduce a new tag library, the *Struts Portlet Tag Library*.

In your JSP pages, you may need to enhance a few Struts JSP tags with their equivalent Struts Bridges tags. This is sometimes necessary for the Struts application to navigate within a portal environment. All forms and links in your Struts application normally write back to the Struts servlet. The Struts Portal Bridge tags ensure that all links are written back to the portal, and all forms are posted back to the portal, and not the Struts application itself.

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<%@ taglib uri="http://portals.apache.org/bridges/struts/tags-portlet-html" prefix="html" %>
```

--- OR ---

```
<%@ taglib uri="http://portals.apache.org/bridges/struts/tags-portlet-html-el" prefix="html-el"
%>
```

Note that the last tag library definition replaces the original Struts HTML tag library completely. The new tag library is the Struts Bridge Tag Library and is required for the following Struts tags:

Table 11.2 Struts Bridge Tag Library

Tag	Description
<html:link>	Renders <a> navigation links inside your Struts application
<html:form >	Renders HTML Forms that post back to the Struts application
<html:rewrite >	Same as <html:link> but does not generate a <a> hyperlink
<html:script>	Renders Java Script validation code directly into the output
<html:image>	Renders Images back to the web application
<html:img>	Renders Images back to the web application

Note that the Struts Bridge library contains all the TLD definition files in the distributed JAR file. So you don't need to add the TLD files yourself under WEB-INF/.

Struts EL Tags

Note that you have the choice of the plain HTML tags or the HTML-EL variant. The HTML-EL variant provides additional Expression Language (EL) support. Each JSP custom tag in this library is a subclass of an associated tag in the Struts tag library. One difference is that this tag library does not use "*rtexprvalues*",. Instead, it uses the expression evaluation engine in the Jakarta Taglibs implementation of the JSP Standard Tag Library (version 1.0) to evaluate attribute values.

The base Struts tag library contains tags which rely on the evaluation of "*rtexprvalue*"s (Runtime Scriptlet Expressions) to evaluate dynamic attribute values. For instance, to print a message from a properties file based on a resource key, you would use the *bean:write* tag, perhaps like this:

```
<bean:message key='<%= stringvar %>'/>
```

This assumes that *stringvar* exists as a JSP scripting variable. If you're using the **Struts-EL** library, the reference looks very similar, but slightly different, like this:

```
<bean-el:message key="${stringvar}"/>
```

The Link Tag

The Link Tag renders an HTML `<a>` element as an anchor definition (if "*linkName*" is specified) or as a hyperlink to the specified URL. URL rewriting will be applied automatically, to maintain session state in the absence of cookies. The content displayed for this hyperlink will be taken from the body of this tag. We have to convert link tags so that all links point back to the portal, not back to the Struts application.

Here is the code in the original Struts Validator example servlet:

```
<html:link page="/registration.jsp">...
```

After portlet conversion, it becomes:

```
<html:link renderURL='true' action='/registration'>
```

The changes made are:

1. Replace the *page* attribute with the specific JSP filename "*/registration.jsp*," with a more accurate *action* attribute, since it is now an action and not a page that we are referencing. The action value "*/registration*", which we are linking to, is a logical view or action forward, not the actual name of the template.
2. Add the Boolean attribute *renderURL* to specify that the link is a render URL. Note that Render URLs are the default, thus the attribute is not needed. This attribute denotes portlet specific behavior and is not relevant in a servlet. Other possible Boolean URL attributes are: *actionURL* and *resourceURL*.

All of the links you see on the page shown in Figure 11.4 are created with the Struts Portlet Bridge HTML tags:

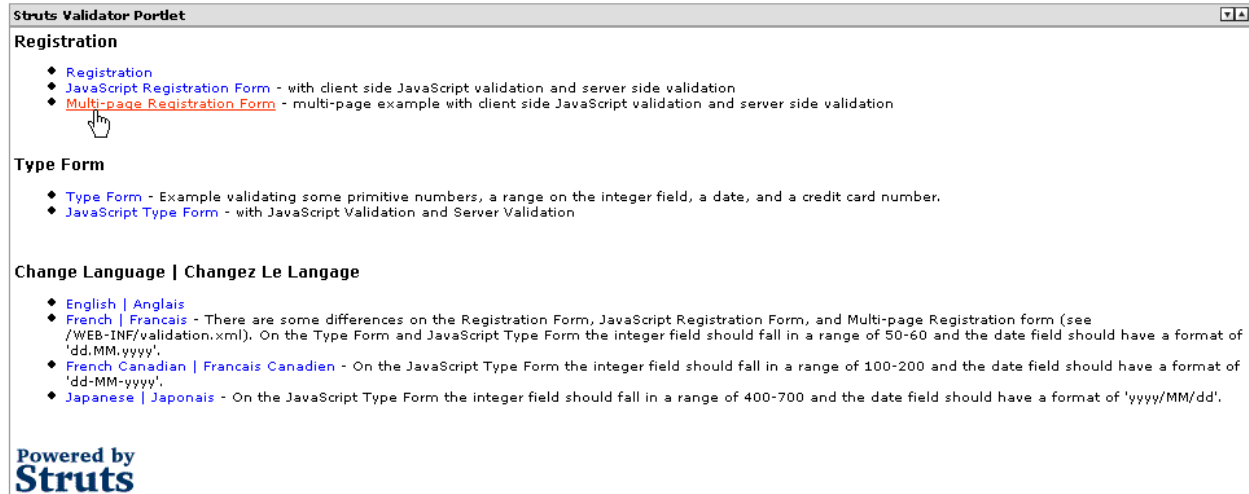


Figure 11.4 Struts Portlet Links

Hover the mouse over one of the links, you will see that the links point back to the portal, as in Figure 11.5. (Note that Jetspeed-2 encodes all portlet parameters.)

`http://localhost:8080/jetspeed/portal/_ns:YW1hbm5pbmctOC4xc3xjMHxkMHxkX3NwYWdlPTE9L211bHRpUmVnaXN0cmF0aW9uLmRv/manning/chapter-8-struts.psmi`

Figure 11.5 The Struts Portlet Link URL

The Form Tag

The Struts `<html:form>` tag outputs a standard HTML form tag, and also links the input form with a Java Bean subclassed from the Struts Action Form object. The Bridges version of this tag ensures a Portlet compliant Action URL is generated for its action attribute. This is required so that all form tags post back to the portal as a portlet action during the portlet action phase, not back to the Struts application.

Here is the code in the original Struts Validator example servlet:

```
<html:form action="registration" onsubmit="return validateRegistrationForm(this);">
```

After portlet conversion, it becomes:

```
<html:form action="registration-submit" onsubmit="return validateRegistrationForm(this);">
```

No changes were made.

The form shown in Figure 11.6 looks just like the Struts servlet form, except it is housed in a portlet window:

The screenshot shows a web portlet titled "Struts Validator Portlet". It contains a form with the following fields and labels:

- First Name: A single-line text input field.
- Last Name: A single-line text input field.
- Address: A multi-line text area.
- City: A single-line text input field.
- State: A single-line text input field.
- Zip: A single-line text input field.
- Phone: A single-line text input field.
- E-mail: A single-line text input field.

At the bottom of the form are three buttons: "Save", "Reset", and "Cancel".

Figure 11.6 A Struts Form in a Portlet

The Rewrite Tag

The Rewrite Tag renders a request URI, but without creating the `<a>` hyperlink. This tag is useful when you want to generate a string constant for use by a JavaScript procedure. The Bridges version ensures a proper Portlet URL is generated. It also supports the Boolean *renderURL*, *actionURL*, or *resourceURL* attributes as the link tag does. You can use one of these Boolean attributes to explicitly specify the type of URL to be rendered (the default is an action URL, but that can be changed using the optional Struts Bridge configuration file as described below). These attributes denotes portlet specific behavior and is not relevant in a servlet.

The Image Tag

The enhanced `<html:img>` and `<html:image>` tags (as well as the `<html:link>` and `<html:rewrite>` tags with the *resourceURL = 'true'* attribute), can be used to generate resource URLs using relative (not prefixed with a '/') SRC references. The Tags determines the correct URL to generate using the current Struts Page URL.

Here is the code in the original *Struts Validator* example servlet:

```
<img SRC='../images/companyLogo.gif'>
```

After portlet conversion, it becomes:

```
<html:image SRC='../images/companyLogo.gif' >
```

The only change made is that we replaced the `` tag with the `<html:image>` tag.

The Script Tag

Render JavaScript validation based on the validation rules loaded by the Struts Validator Plug-in. The Bridges version ensures a proper Portlet URL pointing at the script source is generated. To allow proper functioning within a Portal context, make sure setting the *cdata* attribute to "false".

Struts Bridge Configuration File

The Struts Bridge supports an optional Struts Bridge Configuration file: *struts-portlet-config.xml*. This configuration is a non-intrusive way to tweak your Struts application without changing the code:

```
<config>
  <render-context>
    <attribute name="errors" />
    <attribute name="message" />
  </render-context>
  <portlet-url-type>
    <default="render | action"/>
    <action path="/shop/add" />
    <action path="/shop/switch" />
    <action path="/shop/remove" />
    <action path="/shop/signoff" />
    <action path="/shop/viewCategory" />
    <action path="/shop/viewItem" />
    <action path="/shop/viewProduct" />
    <action path="/shop/viewCart" />
    <action path="/shop/newOrder" />
    <render path="/shop/newOrderForm" />
    <action path="/shop/listOrders" />
    <resource path="/images/" />
  </portlet-url-type>
</config>
```

The bridge can generate three kinds of URL tags:

- Action URL
- Render URL
- Resource URL

The default value is to generate Render URLs. If you don't want to modify your JSP files, you can instruct the bridge using the configuration. The *<portlet-url-type>* element holds a collection of URL directives. The directive below instructs the bridge to create an Action URL for the path *"/shop/add"*:

```
<action path="/shop/add"/>
```

This directive instructs the bridge to create a Resource URL for all paths starting with *"/images/"*:

```
<resource path="/images/" />
```

If an attribute is not specified in the Struts tag, or if an entry is not provided in the configuration, the URL will by default generate a Render URL (except for *<html:form>* which always defaults to actions) unless a default override is provided with the default directive:

```
<portlet-url-type>
  <default="render | action"/>
..
```

Render Context Attributes

The `<render-context>` element of the Struts Configuration file informs the bridge of which request attributes must be passed from the action phase to the render phase. Since the portlet specification requires two phases (action, render), yet the servlet specification only one phase, request attributes cannot be passed from Struts actions to the rendering JSP. The Struts Bridge provides a render context for the passing of request parameters set in Struts Actions to the rendering JSP. The render context is configured in the Struts Bridge configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
  <config>
    <render-context>
      <attribute name="errors"/>
      <attribute name="message" keep="true"/>
    </render-context>
  </config>
```

With the above example configuration, request attributes set in your Struts code named *errors* and *message* are saved after the Action phase. The errors object is restored only once, the message object, with the *keep* attribute, will be restored after all subsequent *RenderRequests* until the next *ActionRequest* or when the Struts Page URL is changed.

Render Context Attributes allow for a Struts Action to forward (without redirecting) to a JSP in a Portlet while passing its required data through the request attributes. Request scoped *ActionForms* can also safely be used as the Struts Bridge will automatically removes them from the session again when they go out of scope!

Using the single use only configuration (without the *keep="true"* attribute) allows one to pass on rather large objects to the *RenderRequest* because, even if they are stored in the Session, this will only be for a very short period (milliseconds). Of course, with a subsequent *RenderRequest* these objects won't be available anymore and the specific *View Renderer* must be defined to handle that situation properly. Error objects or error messages (other than *ActionMessages* and *ActionErrors* which are already automatically saved by the Struts Bridge in a *StrutsRenderContext*) are also reasonable candidates to use like this.

The Portlet Request Processor and Portlet Tiles Request Processor

The Struts Bridge requires a different *RequestProcessor* to be used for the Struts Application: a *PortletRequestProcessor*. If a Struts configuration doesn't define one, or defines one which isn't based on the *PortletRequestProcessor*, it will be replaced automatically by the Struts Bridge. The Struts Bridge also supports Tiles to be used through its *PortletTilesRequestProcessor*. If the *TilesPlugin* is defined (as required to be able to use Tiles in Struts) it will be recognized by the Struts Bridge and then the *PortletTilesRequestProcessor* will be used instead (if not configured already).

Tiles can be used without problem within a Portlet context, even for Action Mapping or Action Forward paths, although the same considerations and restrictions must be taken into account as described above.

Strip Out all HTML Tags Not Allowed by the Portlet API

Because portlets generate markup fragments of content that are aggregated in a portal page document, tags that apply to pages or frames or titles are not applicable and should be stripped out. Portlets must conform to the following rules in order to function properly in a portal.

These tags are not allowed in a portlet generating HTML, XHTML, or XHTML-basic fragments per the Portlet API Specification, appendix PLT.B – Markup Fragments: `<base>`, `<body>`, `<iframe>`, `<head>`, `<html>`, `<title>`. The `<frame>` and `<frameset>` tags are also prohibited in plain HTML fragments as well.

Thus in our example, the *index.jsp* page starts out as a servlet page:

```
<html:html locale="true">
<head>
<title><bean:message key="index.title"/></title>
<html:base/>
</head>
<body bgcolor="white">

<logic:notPresent name="org.apache.struts.action.MESSAGE" scope="application">
...
</body>
</html:html>
```

The HTML tags are stripped out according to the Portlet Specification PLT.B, and the resulting page no longer contains the illegal tags. In the example above, all highlighted code is removed.

Optional Reorganization Best Practice

In the Struts Validator servlet example, all the JSP pages are placed in the root of the web application. As an optional step to keep all of the chapter content in some kind of order and make it easier for the reader to find source code, we moved the JSP files into the /WEB-INF/chapter8/struts directory. This step is in no way required and is considered developer's preference and best practice. This step also meets the 2nd rule and regulation for converting to the Struts Portlet Bridge: securing all JSP files by placing them under the WEB-INF directory. If you choose to do so, remember to update the *strut-config.xml* file for your action mappings:

```
<action-mappings>
  <action path="/welcome" include="/index.jsp" />
```

Becomes:

```
<action-mappings>
  <action path="/welcome" include="/WEB-INF/view/chapter8/struts/index.jsp" />
```

As an optional other best practice, for the Struts Validator example, we have also changed the TLD URI in the JSP files to use a fully qualified URI. The original code used relative URLs in the web application, requiring the TLD to be duplicated there (it is already in the Struts JAR file). Instead, we simply use the TLD URI and let the TLD resource be found in the META-INF directory of the Struts JAR file:

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<%@ taglib uri="http://portals.apache.org/bridges/struts/tags-portlet" prefix="html" %>
```

Finally we moved the Validator and Struts configuration files under /WEB-INF/struts to help organize our chapter 8 examples, which now has quite a diverse selection of portlet bridges!

Now we're ready to run the converted portlet.

Running the Struts Bridge portlet

After following the steps in this section, our Struts Validator application will now run in a portlet exactly the same as in a servlet. We can now see that the form processing, navigations and validations work as expected. If you run the example, go to the Registration page, and leave the form blank, and then press *Save*. Instead of saving your form, the Struts Validator rules kick in, and a validation error is displayed.

The validation rules were defined in the `/WEB-INF/struts/validator-rules.xml` file. For example, the last name field is a required field and must also match a regular expression mask:

```
<field property="lastName" depends="required,mask" page="1">
  <msg name="mask" key="registrationForm.lastname.maskmsg" />
  <arg key="registrationForm.lastname.displayname" />
  <var>
    <var-name>mask</var-name>
    <var-value>^[a-zA-Z]*$</var-value>
  </var>
</field>
```

Now that you learned a lot about how to optimize your portlet programming style by implementing a Model-View-Controller pattern based on Struts technology, we can take a look at how you can achieve the same separation using Velocity and Spring. If you still want more information about programming in Struts, check out the Manning book “Struts in Action”.

Introducing the Bridges Framework

The Bridges Framework, part of the Portals Bridges project at Apache is an application framework specific for developing portlet applications. While other web application frameworks such as JavaServer Faces and Struts were written for developing servlets, the Bridges Framework is specifically designed for developing portlets with front-end technology like Velocity or JSP. There will be several Bridges Framework examples discussed in this chapter that even go beyond the use of Velocity and JSP.

The Bridges Framework provides developers with functionality in addition to what is provided by the Portlet API and Velocity Framework. It is generally not compatible with the JSF or Struts frameworks, since there is an overlap of functionality between the three frameworks. To summarize, Bridges Framework provides:

Table 11.3 Technologies supported by the Bridges Framework

Bridges Framework Component	Purpose
Spring integration	
Model Beans	
Views	
Forwards	
Extended Action Support	
Validation	

In the following sections, we'll take a closer look at each of these technologies and develop at least simple examples that show how to use them.

Developing Portlets with Spring and Portals Bridges

Spring is an open source project; however it is not housed at Apache. *Spring* is a unique Java application framework meant to simplify the development of applications. Where as Enterprise Java Beans are very complex to use, the *Spring* Framework is an easy to use and understand framework for business applications. *Spring* focuses on:

- Providing a **simple way** to manage business objects lifetime and relationships
- A **layered architecture**. *Spring* is comprehensive yet modular. You only use what you need. Thus you may only make use of the JDBC support without sucking in an entire framework
- **Promotes best software development practices**. For example, *Spring* is designed to always promote test-driven development and eliminate the need for factories and singletons.
- **Inversion of Control**. *Spring* is an application container supporting interceptions and declarative aspect oriented programming.

MVC insists that you separate your application model from web interface view. This design pattern has now become a standard practice for all web applications. Within the Bridges Framework, instead of re-inventing the model wheel, we built our model upon *Spring*. With the Bridges Framework, you can take existing *Spring* beans and integrate them into your portlet application, or create new *Spring* beans. Although you can use *Spring* beans as you wish in your application, with bridges the *Spring* beans are used as the model objects backing portlet form processing as you will see in the examples in this chapter.

Model Beans

In the Bridges Framework, we call the objects that represent our model in the MVC paradigm, *Model Beans*. Model Beans represent the actual business objects that you are usually displaying in the view, and then validating storing back to some persistent store during actions. Model Beans can be implemented as *Spring* beans (which can also be Plain Old Java Objects), *Dynabeans* or as Portlet Preferences.

Portlet Preferences are the per-portlet entity settings defined in the Portlet API. In this case, the framework provides sample views for editing preferences, and by calling a framework method, the preferences are automatically stored for you.

Spring beans are defined in the standard *Spring* configuration file. You define your model beans just like any other *Spring* bean definition. For example here is a bean used in the example in this book:

```
<bean id="clientInfo" singleton='false'
      class="com.manning.portlets.chapter8.finance.ClientInfoBean">
  <description>Client Info bean used in help mode.</description>
</bean>
```

Model Beans give you three distinct advantages. The first is working with *Spring* managed objects. This means that you can integrate your portlet model with any existing applications using *Spring* to manage their objects. You don't have to bother with object lifecycle, *Spring* handles this. Your model objects gain all the benefits of running inside an IOC (Inversion of Control) container. Model objects can be assembled using constructor injected dependencies, and the implementation is pluggable via the *Spring* configuration file.

Secondly, your model can leverage the excellent database and JDBC support built into *Spring*. If your application already makes use of *Spring* for its database pooling and connections, then your portlet can too take advantage of the *Spring's* connection pooling and data source configuration.

Third, if you want to simply write custom views over the Portlet Preferences, the framework makes it easy to retrieve and store preferences with minimal effort integrated into the model beans approach.

Views

Views are logical, abstract names for the visible part of your application. Views are the ‘V’ part of the MVC pattern. In the case of the Bridges Framework, views are either JSP or Velocity templates in your portlet application and will be covered in more detail later in this chapter. The Bridges Framework encourages:

17. Abstracting your JSP or Velocity templates behind logical view names
18. Workflow capabilities within a portlet by supporting two or more views per portlet mode (View/Edit/Help)
19. Controlling the navigation and flow of your application by using views as the targets of abstract forward definitions
20. Java Developers to abstract all navigation and application flow behind views and forwards into a centralized repository of application views and forwards.
21. Template Designers to abstract all links and application flow behind views and forwards into a centralized repository of application views and forwards

Often you will see views used in story board or white board specifications. For example, here is a story board example of a very simple login case:

Table 11.4 Views Storyboard

Story	View	Implementation Target
User visits welcome page	Welcome	/views/public/welcome.jsp
User navigates to login page	Login	/views/public/login.jsp
User enters credentials	Login	/views/public/login.jsp
User is authenticated, home page is displayed	Home	/views/secure/home.jsp

The logic in your Java or template code should always use the view name, not the implementation (the template file name). By abstracting your application code from implementation details, you do not bind your application logic to implementation details. This makes your application navigation flow and logic easily extensible and configurable without the need to recompile your Java code. Simply change the view definition, and your application navigates to a new view without any need to recompile.

Forwards

Forwards are an extension to the View concept. Like views, they are abstract names for navigational entities in your application (usually JSP or Velocity templates). However, forwards have one added dimension: forwards can have an action associated with them. For example, a view can have a Success and Failure action associated with it. The action names, like the views are arbitrary. So now we have a more useful model for handling application logic. In your application, you could have:

```
if ( formIsValid )
{
    return "user-profile-form:success";
}
else
{
    return "user-profile-form:failure";
}
```

Additionally, Forwards are useful for changing portlet modes and window states using a simple logical name. The target of a forward can also be attributed with an optional portlet mode and windows state. For example, here is the same story board example from above, but with more complex stories for success and failure logic; and secondly filling out a loan form in the portlet's maximized mode or editing the settings for that form in the portlet's edit mode:

Story	Forward	View	Implementation Target
User visits welcome page		Welcome	/views/public/welcome.jsp
User navigates to login page		Login	/views/public/login.jsp
User enters credentials, login action succeeds	Home:Success	Home	/views/public/login.jsp
User enter credentials, login action fails	Home:Failure	ReLogin	/views/public/relogin.jsp
User choose to fill out a Home Loan Application (a very large form)	LoanEntry	LoanForm, state:maximized	/views/secure/loan-form.vm
User chooses to edit the Home Loan Application settings for localized settings.	LoanEntrySettings	LoanForm, state:maximized, mode:edit	/views/secure/loan-edit.vm

Table 11.5 Views and Forwards Storyboard

Just like with Views, the logic in your Java or template code should always use the view name, not the implementation (the template file name). By abstracting your application code from implementation details, you do not bind your application logic to implementation details. Note that forwards will only navigate within the current portlet. Since the Portlet API specification does not address inter-portlet communication in its first version, forwards are currently limited to navigations within your portlet.

Extended Action Support

Per the Portlet API specification, there is exactly one action method on your portlet: the *processAction* method of the *Portlet* interface. This works fine in many cases, but in more complex applications it is often necessary to clearly define different actions and associate them with specific methods on your portlet class. If you need to have a method per action solution, the Bridges Framework provides an easy to use solution: the name of the action method is specified in a hidden input on your HTML form as follows:

```
<form action="$renderResponse.createActionURL()" method="post">
<input type='hidden' name='portlet.action' value='processHelpAction' />
```

The *processHelpAction* is a method on your portlet. It is called automatically by the framework with the exact same signature as the Portlet API's *processAction* method.

```
public String processHelpAction(ActionRequest request, ActionResponse response, Object
bean)
    throws PortletException, IOException
{
    // do application specific code here

    // return a portlet forward
    return "stocks-help:success";
}
```

Also notice that the extended action classes can return a logical forward name or a logical view name. In the example above, *stocks-help:success* is returned by the method. This forwards to where the portlet will navigate to next within its workflow for the render phase of the portlet. To summarize, the extended action support provides:

1. Targeting of specific actions methods on your portlet class, so that your portlet action class doesn't have to check for a string and figure out which actual business logic method to call.
2. Forwarding to portlet views directly from your action classes

Validation

Web applications require user interaction with web forms. The data entered on a form must often be validated to enforce data integrity up front as a best practice. Typical input comes from HTML widgets such as text input fields, radio button, check boxes, and menu lists. Validation is the process of ensuring that the data entered is correct. Typical validations checks are for valid phone number formats, valid URLs, valid email addresses. The Jakarta Commons Validator provides a small toolkit for validating user input with regular expressions. Input can be validated on the client with Java Script, or back on the server. The Bridges Framework integrates Commons Validator into its portlet development request processing pipeline providing powerful validation processing in your portlet applications.

The Commons Validator works off of a validation configuration file in defining validations rules outside of your program. The configuration file defines the general validation rules. The file defines general validation rules that are used by your forms. The rules that come by default are:

Table 11.6: Validation Rules

Rule	Description
Required	Indicates whether this value is required.
Range (for various Java data types)	The range in which the value have to be.
Minimum Length, Maximum Length	Length restrictions for textual values.
Mask	The value must fulfill the format of the mask.
Data Type validations	The type the value has to represent.
Date	The value must be a valid Date.
Credit Card	The value must be valid Credit Card information.
Email	The value must be a valid Email address.
URL	The value must be a valid URL.

For your application, you then create forms in the Validator configuration file. Each form is associated with a HTML form, and each form can have one or more fields. These fields have one or more validation rules which they are dependent on as shown below:

```
<formset>
  <form name="registrationForm">

    <field property="lastName" depends="required,mask" page="1">
      <msg name="mask" key="registrationForm.lastname.maskmsg" />
      <arg key="registrationForm.lastname.displayname" />
      <var>
        <var-name>mask</var-name>
        <var-value>^[a-zA-Z]*$</var-value>
      </var>
    </field>
```

With the basic theoretical concepts known we will be more concrete in the next section and create a portlet example that applies the newly learned.

Bridges Framework in action: A Portlet Example

In this section we introduce the Bridges Framework with a simple example of a stock quote portlet. Note that the stock quotes provided with the example are fictitious and actually created by a random number generator.

This example portlet demonstrates:

1. Defining your Application (portlet, web) Descriptors
2. Using a *Spring* configuration file to configure your Bridges framework application:
 - a. Model Beans
 - b. Views
 - c. Forwards
 - d. View-to-Bean Map
 - e. View-to-Validator Map
3. Configuring your Validation Rules
4. Using a Model Bean with a Form
5. Using Preferences with a Form
6. Forwarding and Views
7. Extended Action Support

Let's get started with the deployment descriptors modifications for this example.

Defining your Application Deployment Descriptors

The portlet.xml contains the definition for our Bridges Framework-based portlet:

Listing 11.7 The portlet deployment descriptor for the fictitious Stocks Portlet

```
<portlet>
  <description>Chapter 8 Circa 2000 Stock Quote Portlet</description>
  <portlet-name>StockQuote2000</portlet-name>
  <display-name>Stock Quotes 2000</display-name>
  <portlet-class>com.manning.portlets.chapter8.StockQuote2000</portlet-class>
| #1

  <init-param>                                | #2
    <name>spring-configuration</name>         | #2
    <value>/WEB-INF/velocity/spring-portlet-configuration.xml</value>    | #2
  </init-param>                                | #2
  <init-param>
    <name>validator-configuration</name>
    <value>/WEB-INF/velocity/validator-configuration.xml</value>
  </init-param>

  <init-param>
    <name>ViewPage</name>
    <value>stocks-view</value>
  </init-param>
  <init-param>
    <name>HelpPage</name>
    <value>stocks-help</value>
  </init-param>
  <init-param>
```

```

        <name>EditPage</name>
        <value>stocks-edit</value>
    </init-param>

    <supports>
<mime-type>text/html</mime-type>
        <portlet-mode>VIEW</portlet-mode>
        <portlet-mode>EDIT</portlet-mode>
        <portlet-mode>HELP</portlet-mode>
    </supports>

    <supported-locale>en</supported-locale>
    <supported-locale>fr</supported-locale>

    <resource-bundle>com.manning.portlets.chapter8.resources.StockQuote</resource-
bundle>

    <portlet-info>
        <title>Stock Quote 2000</title>
        <short-title>Stocks</short-title>
        <keywords>stock, quote, finance</keywords>
    </portlet-info>

    <portlet-preferences>
        <preference>
            <name>symbols</name>
            <value>IBM,MSFT,ORCL,SUNW</value>
        </preference>
    </portlet-preferences>
</portlet>

```

#1) Starting with the class name, the implementing class for this portlet is:

```
<portlet-class>com.manning.portlets.chapter8.StockQuote2000</portlet-class>
```

Note that the `StockQuote2000` portlet extends the *VelocityFrameworkPortlet*, as it is a Velocity-based framework portlet. We could have also extended *GenericFrameworkPortlet* as this is the base class for the Bridges Framework that has the most substance.

#2) The location of the *Spring* configuration file is defined here as a portlet init parameter:

```

    <init-param>
        <name>spring-configuration</name>
        <value>/WEB-INF/velocity/spring-portlet-configuration.xml</value>
    </init-param>

```

#3) And likewise for the Commons Validator configuration file:

```

    <init-param>
        <name>validator-configuration</name>
        <value>/WEB-INF/velocity/validator-configuration.xml</value>
    </init-param>

```

#4) Three portlet init parameters define the default logical views for View mode, Help mode, and Edit mode respectively:

```

<init-param>
  <name>ViewPage</name>
  <value>stocks-view</value>
</init-param>
<init-param>
  <name>HelpPage</name>
  <value>stocks-help</value>
</init-param>
<init-param>
  <name>EditPage</name>
  <value>stocks-edit</value>
</init-param>

```

#5) The supported portlet modes for HTML are View, Edit, and Help:

```

<supports>
<mime-type>text/html</mime-type>
  <portlet-mode>VIEW</portlet-mode>
  <portlet-mode>EDIT</portlet-mode>
  <portlet-mode>HELP</portlet-mode>
</supports>

```

#6 The supported languages are English and French. The localized resource bundles are packaged in with the Java classes:

```

<supported-locale>en</supported-locale>
<supported-locale>fr</supported-locale>

<resource-bundle>com.manning.portlets.chapter8.resources.StockQuote</resource-
bundle>

```

#7 Of course you will have to use the supported locale extension to find properly name the resource files as:

```

StockQuote_fr.properties
StockQuote_en.properties

```

#8 We define the portlet title and short title (although this can be overridden in the resource bundle). The keywords are used by the portal for searching and locating portlets:

```

<portlet-info>
  <title>Stock Quote 2000</title>
  <short-title>Stocks</short-title>
  <keywords>stock, quote, finance</keywords>
</portlet-info>

```

#9 Finally, the portlet preferences are defined. For this portlet, we require one preference named 'symbols'. The default values are also provided here, although the users are encouraged to override these comma-separated values and supply their own stocks.

```

<portlet-preferences>
  <preference>
    <name>symbols</name>

```

```

        <value>IBM,MSFT,ORCL,SUNW</value>
    </preference>
</portlet-preferences>

```

With the deployment descriptor and therefore with the configuration for the portlet itself in place we can now take a look at the Spring related configuration used by the portlet.

The Spring Configuration File

The Spring Configuration file defines the beans, views, validations and forwards used in your portlet framework application. The file is found under *src/webapp/WEB-INF/velocity/spring-portlet-configuration.xml*. First we define the Spring bean definitions. In this sample application, there is just one bean defined.

Spring Beans

```

<!-- Model -->
<bean id="clientInfo" singleton='false'
      class="com.manning.portlets.chapter8.finance.ClientInfoBean">
    <description>Client Info bean used in help mode.</description>
</bean>

```

This bean holds the client information from the Help Mode form, where the user can enter information about themselves. This bean is backed by a POJO:

```

public class ClientInfoBean implements Serializable
{
    private String name;
    private String phone;
    private String email;
    private String stocks;
    private double income;
    ...
}

```

The View

Logical views define the abstract names for Velocity or JSP pages. This practice decouples your application templates and navigational structure from your application code. In our chapter, we have four views defined:

1. *stocks-view* – the default portlet View mode where the stock quotes are displayed in a table with quotes simulating the year 2000.
2. *stocks-view2* – secondary portlet View mode where the stock quotes are displayed in a table with quotes simulating more modern and sober times
3. *stocks-edit* – the default portlet Edit mode where the stock quotes preferences can be edited
4. *stocks-help* – the default portlet Help mode, displaying a form where the user can request additional information

```

<!-- Views -->
<bean id="portlet-views" class="java.util.HashMap">
    <description>Logical View name to actual view</description>

```



```

    <constructor-arg>
      <map>
        <entry key="stocks-view">
          <value>/WEB-INF/view/chapter8/stocks-view.vm</value>
        </entry>
        <entry key="stocks-view2">
          <value>/WEB-INF/view/chapter8/stocks-view2.jsp</value>
        </entry>
        <entry key="stocks-edit">
          <value>/WEB-INF/view/chapter8/stocks-edit.vm</value>
        </entry>
        <entry key="stocks-help">
          <value>/WEB-INF/view/chapter8/stocks-help.vm</value>
        </entry>
      </map>
    </constructor-arg>
  </bean>

```

Note that the portlet views are defined as a Java Map, and that the keys are the logical view names, whereas the values are the actual Velocity templates. Velocity is covered in detail later in this chapter.

Forwards

Forwards are an extension to the View concept. Like views, they are abstract names for navigational entities in your application (usually JSP or Velocity templates). However, forwards have one added dimension: forwards can have an action associated with them. For example, a view can have a Success and Failure action associated with it. The action names, like the views are arbitrary.

Forwards are defined in the *Spring* configuration as a Java Map:

```

<bean id="portlet-action-forward-map" singleton="true" class="java.util.HashMap">
  <description>Maps logical forward names to views</description>
  <constructor-arg>
    <map>
      <entry key="stocks-edit:success">
        <value>stocks-view,mode:view</value>
      </entry>
      <entry key="stocks-help:success">
        <value>mode:view,state:maximized</value>
      </entry>
    </map>
  </constructor-arg>
</bean>

```

Here we have a map from a forward name and action to a target value, which can be one of the following: a view, a portlet mode, a window state, a view and a portlet mode, or a view and a window state. The forward name and action are separated by a colon. Here the forward name is *stocks-view*, and the action is *success*:

```
stocks-view:success
```

The value holds the target of the forward, in this case it is a view named *stocks-view*, and putting that portlet into View mode:

stocks-view, mode:view

The next example stays on the same view, but changes both the portlet mode and window state:

mode:view, state:maximized

Table 11.7 Valid values for portlet mode forward targets

Forward Value	Portlet Mode
<i>mode:view</i>	View
<i>mode:help</i>	Help
<i>mode:edit</i>	Edit

Table 11.8 Valid values for window state forward targets

Forward Value	Portlet Window State
<i>state:normal</i>	Normal
<i>state:maximized</i>	Maximized
<i>state:minimized</i>	Minimized

View-to-Bean Map

Beans are associated with views by association again using a Java Map:

```
<bean id="portlet-view-bean-map" singleton="true" class="java.util.HashMap">
  <description>Maps views to model beans</description>
  <constructor-arg>
    <map>
      <entry key="stocks-help">
        <value>clientInfo</value>
      </entry>
    </map>
  </constructor-arg>
</bean>
```

In the above example, the stocks-help view is associated with the *clientInfo* Spring bean. This association implies that the view will attempt to populate the bean automatically when the form is processed.

View-to-Validator Map

In your bridges configuration, you associate validation rules with model bean:

```
<bean id="portlet-view-validator-map" singleton="true" class="java.util.HashMap">
  <description>Maps views to validators</description>
  <constructor-arg>
    <map>
      <entry key="stocks-help">
        <value>clientInfo</value>
      </entry>
      <entry key="stocks-edit">
        <value>symbols</value>
      </entry>
    </map>
  </constructor-arg>
</bean>
```

```

        </constructor-arg>
    </bean>

```

In the example above, the *stocks-help* view is associated with the *clientInfo* validation form. The second view, *stocks-edit*, is associated with the *symbols* validation form.

Configuring Your Validation Rules

The Commons Validator Configuration file defines the validation rules for forms used in your portlet framework application. The validation file is found under *src/webapp/WEB-INF/velocity/validator-configuration.xml*. The validation rules defined here are standard Commons Validator rules tailored to the portlet framework environment. You can take the basic rules defined here and apply them to the fields on your model (*Spring*) beans.

In our example, we define two validation forms: *clientInfo* is associated with help mode, and *symbols* is associated with edit mode. Only fields on the form that require validation should be entered into the validation configuration file. It is important to match the name of the field with the name of the public bean data member using standard Java Bean getter / setter naming. The Client Information form has a field named “phone”. Thus the associated (via the common view between the bean and Validator) bean must have a getter / setter named *getPhone* and *setPhone* respectively.

By default, if validation fails, the portlet will on the same page and ignore all navigation rules. Your form template can add fields for displaying error messages as we do in the help mode example:

Listing 11.x The *stocks-help.vm* file with examples of validation support

```

#set ($MESSAGES = $portletConfig.getResourceBundle($renderRequest.Locale))

<h3>$MESSAGES.getString("request.info")</h3>
<p>
$MESSAGES.getString("request.details")

</p>
<form action="$renderResponse.createActionURL()" method="post">
<input type='hidden' name='portlet.action' value='processHelpAction' />
<table>
    #formField('Name' "$!clientInfo.Name" "40" 'name' $MESSAGES $ERRORS)
    #formField('Phone' "$!clientInfo.Phone" "20" 'phone' $MESSAGES $ERRORS)
    #formField('Email' "$!clientInfo.Email" "40" 'email' $MESSAGES $ERRORS)
    #formField('Stocks' "$!clientInfo.Stocks" "40" 'stocks' $MESSAGES $ERRORS)
    #formField('Income' "$!clientInfo.Income" "12" 'income' $MESSAGES $ERRORS)
</tr>
</table>
<input type="submit" name="Save" value="$MESSAGES.getString("save")" />
</form>

#ErrorMessages($ERRORS)

#parse("/WEB-INF/view/chapter8/stocks-bottom.vm")

```

The *\$ERRORS* variable is automatically populated by the framework. If there are no errors, this object is empty. The *#ErrorMessages* executes a Velocity macro. This macro displays a list of one or more errors found by the Validator. The *#formField* macros are also capable of flagging fields that failed validation as shown in this example:

Stock Quotes 2000

Request More Information

If you would to find out more about our retro stock quote service, please fill out this form. And remember: Hold on to those stocks!

Name:

Phone:

Email:

Stocks:

Income:

Field Email is a required field.
Field Stocks is a required field.

[Normal](#) [Min](#) [View](#) [Edit](#)

Figure 11.7 Flagging invalid fields with the Validator

Using a Model Bean with a Form

Model Beans work together with a view and a form in processing the input of a typical data entry form. The Client Information example on the Help Mode view provides an example of using a plain old java bean as the model bean in the framework.

Listing 11.x A Plain Old Java Bean

```
package com.manning.portlets.chapter8.finance;

import java.io.Serializable;

/**
 * ClientInfoBean
 *
 * @author <a href="mailto:taylor@apache.org">David Sean Taylor</a>
 * @version $Id: ClientInfoBean.java,v 1.1 2004/11/15 17:16:37 david Exp $
 */
public class ClientInfoBean implements Serializable
{
    private String name;
    private String phone;
    private String email;
    private String stocks;
    private double income;

    public ClientInfoBean()
    {
    }

    public ClientInfoBean(String name, String phone, String email, String stocks)
    {
        this.name = name;
        this.phone = phone;
        this.email = email;
        this.stocks = stocks;
    }

    /**
     * @return Returns the email.
     */
}
```

```

    */
public String getEmail()
{
    return email;
}
/**
 * @param email The email to set.
 */
public void setEmail(String email)
{
    this.email = email;
}
/**
 * @return Returns the name.
 */
public String getName()
{
    return name;
}
/**
 * @param name The name to set.
 */
public void setName(String name)
{
    this.name = name;
}
/**
 * @return Returns the phone.
 */
public String getPhone()
{
    return phone;
}
/**
 * @param phone The phone to set.
 */
public void setPhone(String phone)
{
    this.phone = phone;
}
/**
 * @return Returns the stocks.
 */
public String getStocks()
{
    return stocks;
}
/**
 * @param stocks The stocks to set.
 */
public void setStocks(String stocks)
{
    this.stocks = stocks;
}
/**

```

```

    * @return Returns the income.
    */
    public double getIncome()
    {
        return income;
    }
    /**
    * @param income The income to set.
    */
    public void setIncome(double income)
    {
        this.income = income;
    }

    public boolean lookup(String key)
    {
        // dummy lookup method
        this.setEmail("joe@programma.com");
        this.setIncome(1.0);
        this.setName("Joe Programmer");
        this.setPhone("999 333 4444");
        this.setStocks("ABC,DEF");
        return true;
    }
}

```

As you can see there are mostly Java Bean style getters and setters. The *lookup* method provides a way to populate the object using a String key. This method is not used in this simple example. If you look at the Stock Quote 2000 portlet, you will see that there is not a specific *doHelp* method implemented on the portlet. This is because the base class handles the *doHelp* processing. And, it automatically populates the form bean into the request so that the view can access it.

Listing 11.x The *stocks-help.vm* file with accessing the client information bean (*\$clientInfo*)

```

#formField('Name' "$!clientInfo.Name" "40" 'name' $MESSAGES $ERRORS)
#formField('Phone' "$!clientInfo.Phone" "20" 'phone' $MESSAGES $ERRORS)
#formField('Email' "$!clientInfo.Email" "40" 'email' $MESSAGES $ERRORS)
#formField('Stocks' "$!clientInfo.Stocks" "40" 'stocks' $MESSAGES $ERRORS)
#formField('Income' "$!clientInfo.Income" "12" 'income' $MESSAGES $ERRORS)

```

The bean is accessed by the bean name defined in the *Spring* configuration file: *\$clientInfo*. (The *\$* prefix is standard Velocity syntax for request variables)

When saving the Client Information form, the bean is automatically populated by the framework. So there is no need to override the *processAction* Portlet API method in this case. In fact, a *processHelpAction* is implemented, which we will look at in more detail in the last section of this chapter.

Using a Preference with a Form

The Edit Mode page works with a different kind of bean, a Portlet Preference.

Stock Quotes 2000

Enter your symbols separated by commas::

Save

[Normal](#)
[Min](#)
[View](#)
[Help](#)

Figure 11.8 Edit Mode: using preferences in a custom form

The edit mode form is pretty minimal:

Listing 11.x Putting up the preferences

```
#set ($MESSAGES = $portletConfig.getResourceBundle($renderRequest.Locale))
#set ($symbols = $prefs.get('symbols'))

<form action="$renderResponse.createActionURL()" method="post">
<table>
#formField('enter.symbols' "$symbols" "40" "symbols" $MESSAGES $ERRORS)
</table>
<input type="submit" name="Save" value="$MESSAGES.getString("save")" />
</form>

#ErrorMessages($ERRORS)

#parse("/WEB-INF/view/chapter8/stocks-bottom.vm")
```

The framework automatically populates the Portlet Preferences into the request variable named *\$prefs*. From there its pretty straightforward. The action URL posts back to our portlet, and the framework automatically saves the preferences. After saving, the forward is automatically picked up off the application model, forwarding on success from stocks-edit to stocks-view.

Forwards and Views

Forwards are automatically handled by the framework upon success of a portlet action, or upon failure of a portlet action. The two reserved suffixes to a forward: success and failure are supported. Thus when a form is posted, an action is executed on a framework portlet. If the action succeeds, the success action forward is executed. When successfully saving the preferences from edit mode, the forward rule is applied:

```
<entry key="stocks-edit:success">
  <value>stocks-view,mode:view</value>
</entry>
```

And you are navigated back to view mode:

Stock Quotes 2000			
Symbol	Price	Change	Volume
IBM	73.97	3	10044467
MSFT	86.86	-0	348217
ORCL	85.36	-0	5088057
SUNW	103.02	3	6149390

[FlashForward](#)
[Test](#)
[Normal](#)
[Min](#)
[Edit](#)
[Help](#)

Figure 11.9 The final Stock Quote view

Forwards can also be referenced in extended actions as the next section covers.

Extended Action Support

Per the Portlet API specification, there is exactly one action method on your portlet: the *processAction* method of the *Portlet* interface. This works fine in many cases, but in more complex applications it is often necessary to clearly define different actions and associate them with specific methods on your portlet class. If you need to have a method per action solution, the Bridges Framework provides an easy to use solution: the name of the action method is specified in a hidden input on your HTML form as follows:

```
<form action="$renderResponse.createActionURL()" method="post">
<input type='hidden' name='portlet.action' value='processHelpAction' />
```

The *processHelpAction* is a method on the Stock Quote portlet. It is called automatically by the framework with the exact same signature as the Portlet API's *processAction* method.

```
public String processHelpAction(ActionRequest request, ActionResponse response, Object
bean)
    throws PortletException, IOException
{
    // do application specific code here

    // return a portlet forward
    return "stocks-help:success";
}
```

Also notice that the extended action classes can return a logical forward name or a logical view name. In the example above, *stocks-help:success* is returned by the method. This forwards to where the portlet will navigate to next within its workflow for the render phase of the portlet.

To summarize, the extended action support provides:

1. Targeting of specific actions methods on your portlet class, so that your portlet action class doesn't have to check for a string and figure out which actual business logic method to call.
2. Forwarding to portlet views directly from your action classes

With help of this sample you have learned the basic concepts of the Bridges framework and how the Spring project can be leveraged to help to comply with the MVC pattern especially in regards of the model and only mentioning the view part. Now we will dive into the front-end and discuss several options for technologies that can be used.

Developing portlets using the Velocity Bridge

Velocity is a Java-based template engine. It permits anyone to use the simple yet powerful template language to reference objects defined in Java code. Velocity is very popular with Java developers working on web applications. Velocity is open source and housed at the Apache Jakarta project.

When Velocity is used for web development, Web designers can work in parallel with Java programmers to develop web sites according to the MVC model, meaning that web page designers can focus solely on

creating a site that looks good, and programmers can focus solely on writing top-notch code. Velocity separates Java code from the web pages, making the web site more maintainable over the long run and providing a viable alternative to JSP or PHP.

In this section, we will develop a very simple Velocity portlet with the Velocity Bridge.

Developing A Very Simple Velocity Portlet

The *SimpleVelocityPortlet* will introduce you to the basic philosophy behind developing with Velocity: the separation of concerns pattern. We will see how the Java code puts objects in the Velocity context, and then the Velocity templates pull these Java objects (tools) out of the context.

The portlet descriptor is found in our portlet.xml. We've highlighted the most important elements of the deployment below.

Listing X: Velocity Portlet Deployment Descriptor

```
<portlet>
  <description>Chapter 8 Simple Velocity Portlet</description>
  <portlet-name>SimpleVelocityPortlet</portlet-name>
  <display-name>Simple Velocity Portlet</display-name>
  <portlet-class>
    com.manning.portlets.chapter8.SimpleVelocityPortlet 1
  </portlet-class>
  <init-param>
    <name>ViewPage</name>
    <value>/WEB-INF/view/chapter8/simple-velocity.vm</value> 2
  </init-param>
  <supports>
    <mime-type>text/html</mime-type>
    <portlet-mode>VIEW</portlet-mode>
  </supports>
  <supported-locale>en</supported-locale>
  <supported-locale>fr</supported-locale>
  <supported-locale>de</supported-locale>
  <resource-bundle>
    com.manning.portlets.chapter8.resources.SimpleVelocity 3
  </resource-bundle>
</portlet>
```

The portlet class ¹ tells the portal the name of the implementing portlet.

Listing X: Velocity Portlet

```
package com.manning.portlets.chapter8;

import java.io.IOException;
import java.util.LinkedList;
import java.util.List;

import javax.portlet.PortletException;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;
```

```

import org.apache.portals.bridges.velocity.GenericVelocityPortlet;
import org.apache.velocity.context.Context;

/**
 * Simple Velocity Portlet
 *
 * @author <a href="mailto:taylor@apache.org">David Sean Taylor</a>
 * @version $Id: SimpleVelocityPortlet.java,v 1.1 2004/11/20 21:55:07 david Exp $
 */
public class SimpleVelocityPortlet extends GenericVelocityPortlet
{
    public SimpleVelocityPortlet()
    {
    }

    public class Player
    {
        private int goals;
        private String name;
        private String country;

        public Player(String name, String country, int goals)
        {
            this.goals = goals;
            this.name = name;
            this.country = country;
        }

        /**
         * @return Returns the goals.
         */
        public int getGoals()
        {
            return goals;
        }

        /**
         * @return Returns the country.
         */
        public String getCountry()
        {
            return country;
        }

        /**
         * @return Returns the name.
         */
        public String getName()
        {
            return name;
        }
    }

    public void doView(RenderRequest request, RenderResponse response)
    throws PortletException, IOException

```

```

{
    Player robben = new Player("Robben", "Netherlands", 19);
    Player lampard = new Player("Lampard", "England", 14);
    Player drogba = new Player("Drogba", "Ivory Coast", 21);
    List chelsea = new LinkedList();
    chelsea.add(robben);
    chelsea.add(lampard);
    chelsea.add(drogba);

    Context context = super.getContext(request);
    context.put("team", chelsea); 1

    super.doView(request, response);
}
}

```

The Simple Velocity Portlet demonstrates the central concept of the ‘context’ used extensively in Velocity programs. Java objects are simply put into the Velocity context as shown in the example highlighted above ¹. By extending the Velocity Bridge, we can nicely integrate our *doView* method of the portlet with Velocity. The example above puts a Java list into the Velocity context.

Accessing the Java list

This listing demonstrates simple Velocity template. Velocity is designed for rapid development of Java and markup templates. In our case here, we are working with an HTML template. Any Java object put into the Velocity context is referenced in your HTML template with the \$ prefix. Next, let’s see how to access the list from a Velocity template:

Listing X: The Velocity Portlet Template

```

#set ($MESSAGES = $portletConfig.getResourceBundle($renderRequest.Locale))

<table border="1" cellspacing="1" cellpadding="3">
<tr>
    #headerCell($MESSAGES.getString("player.name"))
    #headerCell($MESSAGES.getString("player.country"))
    #headerCell($MESSAGES.getString("player.goals"))
</tr>
#foreach ($player in $team)
<tr>
    <td class='portlet-section-body'>
        $player.Name
    </td>
    <td class='portlet-section-body'>
        $player.Country
    </td>
    <td class='portlet-section-body'>
        $player.Goals
    </td>
</tr>
#end
</table>

#parse("/WEB-INF/view/chapter8/stocks-bottom.vm")

```

Thus, when we put in the Java List object named “team” into the context:

```
context.put("team", chelsea);
```

We then pull it out of the context in the HTML template:

```
#foreach ($player in $team)
```

The **\$team** variable is a Java List. Velocity automatically knows how to work with any object in the Java 2 Collection Framework (Collection, List, Set, Vector ...) or Array. Let’s step back to briefly learn how to write Velocity templates. Velocity has a very powerful set of constructs, yet Velocity is so easy to understand it shouldn’t take more than a few minutes to cover the entire syntax!

Velocity 101

Using \$ syntax

The Context - Any object put in the context by your Java program can be used in your template using the \$ syntax.

```
context.put("person", joe);
```

Is accessed in the templates as a Velocity variable:

```
$person
```

Using JavaBean syntax

Java Bean syntax is supported. Any public getter or setter on an object is easily accessed with simple Java Bean syntax:

```
$person.getName()
```

Or more simply as:

```
$person.Name
```

Using public methods and variables

Public methods are used much like in Java, and they do accept Velocity variables as parameters:

```
$person.formatName($locale, $date)
```

Variables (references) – The #set directive creates new variables or sets variables to new values:

```
#set ($count = $count + 1)
#set ($MESSAGES = $portletConfig.getResourceBundle($renderRequest.Locale))
```

Using loops

The **#foreach** directive is used to iterate over Java collections and iterators. A variable is assigned to the current value of the iterated collection:

```
#foreach ($player in $team)
<tr>
  <td>
    $player.Name
  </td>
#end
```

Using conditional logic

The **#if** and **#else** directives control logic:

```
#if ($player.Position == "Striker")
  <b>Striker</b>
#elseif ($player.Position == "Defender")
  <b>Defender</b>
#else
  <b>Keeper</b>
#end
```

Parsing

The **#parse** directive allows you to reuse snippets of templates and copy them into your template:

```
#parse("/WEB-INF/view/chapter8/stocks-bottom.vm")
```

Using macros

Macros (*Velocimacros*) are parameterized snippets of templates:

Here is a sample macro:

```
#macro (headerCell $body)
  <th class="portlet-section-header">
    $body
  </th>
#end
```

Then to use it in your template:

```
#headerCell($MESSAGES.getString("player.name"))
```

And that about covers it. You are now ready to start cranking out Velocity portlets! For more detailed information on programming with Velocity check out the website at Apache Jakarta.

Our Portlet Example

Getting back to our Portlet example, the Java is very straightforward. Every time we render the portlet, we put a Java collection object in the context. Then in the Velocity template we pull out the object to be merged with our HTML content.

```
#foreach ($player in $team)
<tr>
  <td class='portlet-section-body'>
    $player.Name
  </td>
  <td class='portlet-section-body'>
    $player.Country
  </td>
  <td class='portlet-section-body'>
    $player.Goals
  </td>
</tr>
```

The resulting portlet displays the contents of the list:



The screenshot shows a web browser window titled "Velocity Portlet". Inside the window is a table with three columns: "Player", "Country", and "Goals". The table contains three rows of data: Robben (Netherlands, 19), Lampard (England, 14), and Drogba (Ivory Coast, 21). At the bottom left of the table, there are links for "Max" and "Min".

Player	Country	Goals
Robben	Netherlands	19
Lampard	England	14
Drogba	Ivory Coast	21

Figure 11.10 The Velocity Portlet, View mode

All the essentials of a Velocity template are there:

- a loop directive to iterate over a Java Collection
- Java Bean syntax to access public attributes
- Assignment of a Java variable with the set directive
- A macro to help us with the header cells
- Parsing in another template for the common bottom links

We've now learned how to build a simple portlet with a Velocity template. Now let's make sure it is integrated with the Portlet API.

Portlet API Integration

The Velocity Framework will always add several variables to the context for you to access anytime. These variables emulate the variables that the Portlet API makes available to all JSP templates running inside a portlet: The Render Request object (`$renderRequest`), the Render Response object – The Render Request object from the Portlet API (`$renderResponse`), and the Portlet Config object – The Render Response object from the Portlet API(`$portletConfig` – The Portlet Config object from the Portlet API).

In addition several constants are added to the request context:

- `$MODE_EDIT` – same as Java constant *PortletMode.EDIT*
- `$MODE_VIEW` – same as Java constant *PortletMode.VIEW*
- `$MODE_HELP` – same as Java constant *PortletMode.HELP*
- `$STATE_NORMAL` – same as Java constant *PortletState.NORMAL*
- `$STATE_MIN` – same as Java constant *PortletState.MINIMIZED*
- `$STATE_MAX` – same as Java constant *PortletState.MAXIMIZED*

The Portlet API provides several JSP tags for URL creation and name spacing. In Velocity templates, you have to use the programmatic equivalents of these tags. For example:

```
#set($edit = $renderResponse.createRenderURL())
$edit.setPortletMode($MODE_EDIT)
<a href="$edit">Edit</a>
```

Finally, the Velocity Bridge provides helper functions for your portlet code. Your portlet often has to edit its preferences from Edit Mode. The Velocity Bridge provides you with a built in Preferences Editor requiring minimal coding on your part.

```
public void doPreferencesEdit(RenderRequest request, RenderResponse response)
throws PortletException, IOException
{
    Context context = getContext(request);
    PortletPreferences prefs = request.getPreferences();
    Iterator it = prefs.getMap().entrySet().iterator();
    context.put("prefs", it);
    super.doEdit(request, response);
}
```

And then your Velocity template can simply access the preferences as any other collection:

```
<form action="$renderResponse.createActionURL()" method="post">
<table>
#foreach ($pref in $prefs)
#prefField($pref.Key $pref.Value "40")
#end
</table>
<input type="submit" name="Save" value="Save" />
</form>
```

Above we are using a *Velocimacro* provided with the Velocity Framework called “#prefField”, taking the preference key, value, and field size as parameters.

This concludes the bridges chapter of this book. But Jetspeed Portal also has a lot of useful portlets based on the bridges technology. Actually, most of the portlets in Jetspeed-2 are at least minimally extending the JSP or Velocity portlet. Many of the administrative portlets extend the My Faces Bridge. There are also examples of SSO integration, RSS, Web Services and more.

Summary

This chapter provided an introduction to the bridges available to web application technologies at Apache Portals. We have created portlets using the Portals Bridges Framework, which allows those with experience in building web applications in a variety of methods to easily build JSR 168-compliant portals. We've build portals as JavaServer Pages, and JavaServer Faces. We converted a Struts application to a portlet. Finally, we built a portlet with a Velocity template.

This concludes Part 4; in the next part of the book you will have a chance to see many of the technologies and best practices presented in the book so far put into action.

Chapter 12 Customizing the Jetspeed Portal

This chapter demonstrates how to create your own custom portal based on the out-of-the-box Jetspeed-2 portal introduced in Chapter 8. The Jetspeed-2 portal provides easy integration points for creating a customized solution.

In section 12.1, we first introduce the Jetspeed-2 sample portal and portlets that are available out of the box. In section 12.2, we introduce our customized portal based on a fictitious company. From section 12.3 onwards we provide a step-by-step guide for going from the basic Jetspeed portal (in section 12.1) to creating a customized portal solution, shown in section 12.2. Starting in section 12.3, we will define the site layout using a CMS hierarchy of portal resources such as folders and pages.

Using security policies in section 12.5, we will demonstrate how to secure these resources. More than likely, your portal will need to integrate with a backend authentication or authorization system. Jetspeed comes with its own complete security system out of the box, complete with a user database and Java Authentication and Authorization Service (JAAS) security policy stored in the database.

Jetspeed is built with *Spring* components. In section 12.7, we will show you how to plug in your own portal services into the Jetspeed component architecture.

Getting to know the default Jetspeed-2 portal

In this section we will take a look at the out-of-the-box Jetspeed-2 portal. The distribution that comes with this book contains a ready-to-use portal: Jetspeed-2. The default demo-version of Jetspeed will look something like figure 12.1 (depending on your version of Jetspeed-2, here is version 2.0 M3 as of the writing of this book):



Figure 12.1 Jetspeed's out of the box guest page lets a user login, and presents a few other options.

Figure 12.1 shows the guest page. It is the page displayed to visitors of the site before they have authenticated.

(callout) <#1 Top Menu, contain links to other pages in the default folder>

(callout) <#2 Folders and Pages, contains links to other folders and pages not found in this folder>

(callout) <#3 Additional Links, contains external links outside of the portal defined for this folder>

(callout) <#4 Locale Selector Portlet >

(callout) <#5 Pick a Number JSF Example Portlet>

(callout) <#6 IFrame Example Portlet >

(callout) <#7 Login Portlet>

(callout) <#8 Other Example Portlets>

Note: If you need help getting Jetspeed-2 installed or running, see Chapter 8.

Navigating the portal

The left side of the portal contains menu items that allow the user to navigate around the portal site. The tabbed-menus across the top provide for additional portal navigation links. There are a number of ways that you can configure and display your portal menus. It all comes down to two kinds of menu navigations:

22. Built-in Menus

23. Declarative Menu definitions

Built-in Menu definitions are the default with Jetspeed. Built-in menus require no declarative menu definitions. If you are using one of the default portal layouts that Jetspeed provides, they will simply and automatically appear on the page. There are three different kinds of built-in menu elements:

Pages – the menu options displayed are the pages in the current folder

24. Navigational Menu – the menu options displayed are folders and top level (external) links

25. Breadcrumbs – a path back out of the current folder all the way up the folder hierarchy

In the default Jetspeed out of the box, pages and breadcrumbs are displayed across the top of the page, and navigations are displayed down the left-hand side.

Page Menus

The tabbed-menus across the top contain selections for each page found in the current folder. In our example (Figure 12.2), there are three pages found in the root folder of the portal for the guest user:



Figure 12.2 Page Menus displayed in a tabbed-menu style across top of page.

Breadcrumbs

The area immediately below the tabbed-menus contains the breadcrumb trail. From the breadcrumb trail (Figure 12.3), you can navigate back out of the folder hierarchy up to previous destinations.



Figure 12.3 The Breadcrumb trail, here at one folder-level down, we display a breadcrumb trail back to the root folder

Navigational Menus

The left menu area of the portal contains navigational menus. With the built-in menus, Jetspeed provides two kinds of sub-menus: Folders and Additional Links (Figure 12.4):



Figure 12.4 Navigational Menu holding both Folders and external Links

Declarative Menus

Menus can be declared directly into a folder's metadata resource. Declarative menus are useful for explicitly defining menu options that do not follow the layout of pages and folders.

```
<menu name="page-navigations">
  <separator>
    <text>Top Pages</text>
    <metadata name="text" xml:lang="fr">Page haut</metadata>
  </separator>
  <options>/Administrative</options>
  <separator>Profiled Pages</separator>
  <options regexp="true">/p[0-9][0-9][0-9].psml</options>
  <separator>Non Java Pages</separator>
  <options>/non-java</options>
```

</menu>

Jetspeed provides a simple and well known navigational paradigm for building menus from portal resources: folders, pages and links. Both built-in menus and declarative menus are driven by an underlying Folder/Page/Link structure. The Jetspeed portal site is defined by a navigational space of folders, pages, and links. Only pages can hold portlets. This navigational space is also known as the portal site. The portal site is very much like a typical file system or CMS (Content Management System) hierarchy. There is a root folder. That root folder can contain other folders, pages, and links. This organizational structure continues for all folders building a tree hierarchy of site content. We will look at how to customize menus in more details in section 12.4.

Next we will have a look at a few of the available portlets that come with the default Jetspeed distribution. Then we can move on to a hands-on example of customizing the guest page and authenticated user pages to create our own fictitious portal site: the Waxwing Benefits Portal.

A Quick Tour of the Jetspeed-2 portlets

As we mentioned, the guest page (Figure 12.1) displays when an unauthenticated user arrives at the portal. First, let's look at the portlets shown on the Welcome to Jetspeed-2 tab. There are several useful portlets here that provide great examples of how to get started with portlet programming. You can also extend these portlets for your own needs as we'll show you in chapters 13 and 14. These portlets come with the default deployment of Jetspeed. Starting from the top left corner:

Table 12.1 Portlets Included in the Default Jetspeed Portal

Portlet Name	Purpose	Portable?
Language Selector	Enables user to change the portal's language for the current user session. By default, most portals (Jetspeed included) will look at the capabilities of the browser and automatically provide the correct language to the user. The Locale selector allows you to view the portal in a language other than what your browser expects	N
Pick a Number Game	Although a very simple game, a good example of how to use portlet actions, preferences, edit mode, and help mode. This portlet is portable to other portal servers	Y
IFrame Prototype	The IFrame Portlet displays the contents of another website inside of a portlet. The user can interact with that website independent of interactions with the portal. By including external (or internal) content via an IFrame Portlet, you can secure access to protected resources using the security model of the portal. It is sometimes best to maximize the portlet window when using the IFrame Portlet. The portlet is portable to other portal servers.	Y
Login	Jetspeed-2 specific login using JAAS and active authentication to authenticate users. Even if you swap out your login module or Jetspeed-2 security components, since this portlet uses active authentication and the underlying servlet container's standardized authentication, you can still make use of this portlet to authenticate users. Portal sites using federated authentication services will want to remove the Login Portlet from their site, and delegate authentication to the authentication provider, which can then redirect to the portal upon successful authentication. This portlet is not portable to other portal services	N
Role Security Test	An example portlet demonstrating how role-based portlet security works with the <code>isUserInRole</code> Portlet API to determine if the current user has a set of roles	Y
User Info Test	An example portlet demonstrating how to retrieve user attributes with the Portlet API	Y
Bookmark	A portlet that tracks bookmarks to external websites. The portlet demonstrates programming both view and edit modes of portlet development.	Y

We'll discuss the Login portlet and security in general in more detail in section 12.6. Next let's navigate over to the JPetStore link in the top page menu options.

The JPetStore portlet

JPetStore is a well-known example Struts application in the servlet programming world. Here we can see a complete Struts application running inside Jetspeed-2 (Figure 12.5):

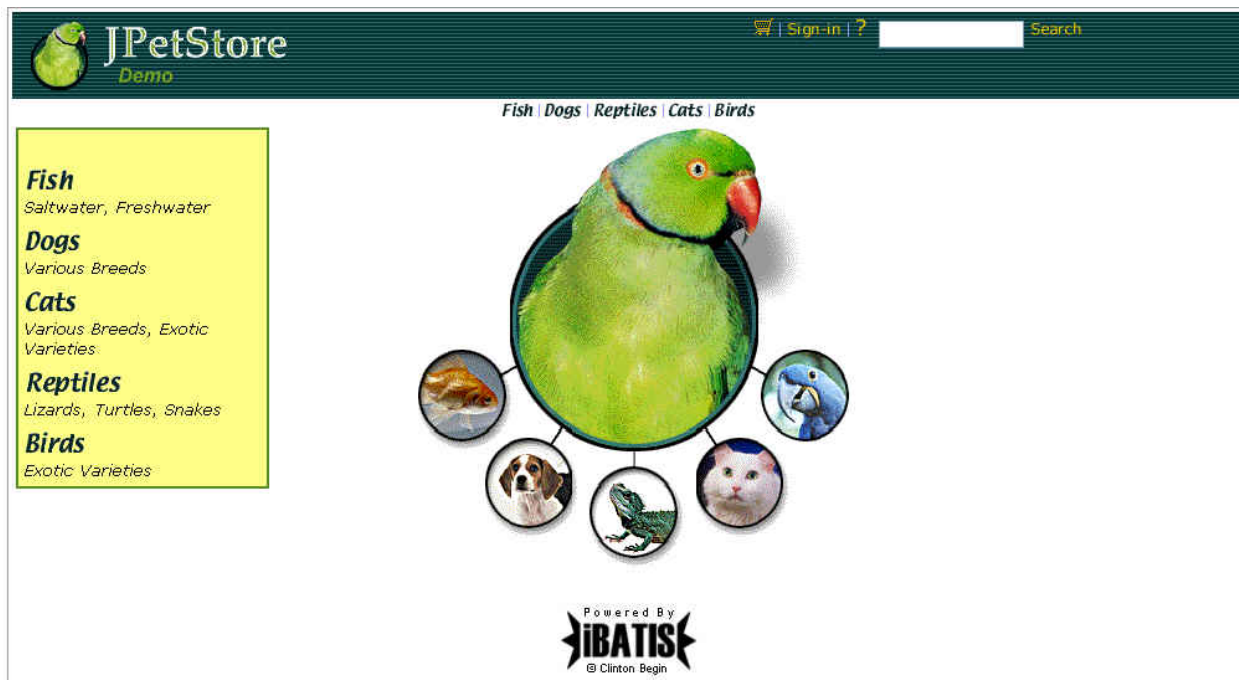


Figure 12.5 The JPetStore Struts application running inside of a portlet via the Apache Portals Struts Bridge
The source code to JPetStore is provided with the book's distribution. It can be found under the Jetspeed-2 source distribution, under the applications/jpetstore subdirectory. A detailed programming example with the Struts Bridge is covered in more detail in chapter 11.

RSS portlets

By clicking on the RSS Demo tab, we see, in Figure 12.6, two **RSS Portlets**, one based on the Extensible Stylesheet Language Transformations (XSLT), the other using the Rome RSS framework. Rome [1] is a set of Java classes for working with standard syndication feed formats such as Atom and RSS from within a Java program:

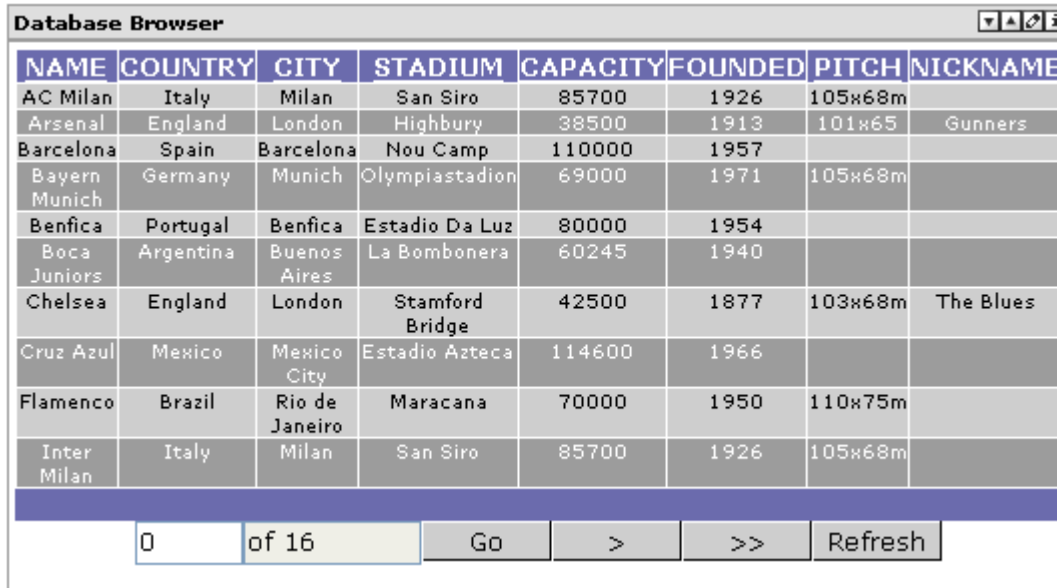


Figure 12.6 RSS Portlets, left side using XSLT, right side using the Rome API.

A detailed programming example with the RSS Portlet is covered in more detail in chapter 14.

The Database portlet

We'll conclude our quick tour of the Jetspeed portal by looking at the Database portlet. By clicking on the Public Folders link on the left-side menu, we navigate to an example **Database Portlet** (Figure 12.7):



NAME	COUNTRY	CITY	STADIUM	CAPACITY	FOUNDED	PITCH	NICKNAME
AC Milan	Italy	Milan	San Siro	85700	1926	105x68m	
Arsenal	England	London	Highbury	38500	1913	101x65	Gunners
Barcelona	Spain	Barcelona	Nou Camp	110000	1957		
Bayern Munich	Germany	Munich	Olympiastadion	69000	1971	105x68m	
Benfica	Portugal	Benfica	Estadio Da Luz	80000	1954		
Boca Juniors	Argentina	Buenos Aires	La Bombonera	60245	1940		
Chelsea	England	London	Stamford Bridge	42500	1877	103x68m	The Blues
Cruz Azul	Mexico	Mexico City	Estadio Azteca	114600	1966		
Flamengo	Brazil	Rio de Janeiro	Maracana	70000	1950	110x75m	
Inter Milan	Italy	Milan	San Siro	85700	1926	105x68m	

0 of 16 Go > >> Refresh

Figure 12.7 Jetspeed's Database Browser Portlet in view mode

The Database Portlet has a lot of configuration options from Edit Mode, seen in Figure 12.8. From this Edit page, you can configure the database connection, choosing from three different connection types:

1. **JNDI Data Source** – the data source is located by a simple JNDI name that is configured in your application server. This is the preferred solution since you can configure all JDBC drivers and connection pooling using the centralized management of your application server.
2. **JDBC/DBCP** – use standard JDBC connection parameters for your JDBC driver. Requires that you put the JDBC driver in your application server class path. Jetspeed makes use of DBCP (Jakarta Database Connection Pool) to pool database connections.
3. **SSO** –uses Jetspeed Single Sign-on to store credentials. The connection properties work the same as JDBC/DBCP. This allows you to store and share credentials in a secure SSO data store.

The SQL parameter in the General Setting section of the page allows you to set the SQL query string. Note that you will not want to give Edit access to most users, since these properties are meant for administrative personnel. A detailed programming example with the Database Portlet is covered in more detail in chapter 13.

DbBrowserExample

Database Browser Preferences

Data Source Type

☐ JNDI Data Source
☒ DBCP Data Source
☐ SSO Credential Store

Please Select a Data Source Type

JNDI Settings

JNDI Data Source

JDBC/DBCP Settings

JDBC Driver

com.mysql.jdbc.Driver

JDBC Connection

jdbc:mysql://j2-server/j2

JDBC Username

j2

JDBC Password

J2 SSO Settings

JDBC Driver

com.mysql.jdbc.Driver

JDBC Connection

jdbc:mysql://127.0.0.1/j2

SSO Site

General Settings

Window Size

10

SQL

select * from CLUBS

Save

Test

Figure 12.8 Configuring data source, connection parameters, and a SQL query in edit mode with the Database Browser Portlet

All of the portlets demonstrated here (JPetStore, RSS, and Database Browser) are portable to other Portlet API-compliant application servers. Jetspeed-2 portlets are always based on the

standard and most portlets are naturally portable. Only administrative or security type portlets are not portable, since they are specific to the portal implementation and necessary to manage the unique implementation.

Now that we've seen the default portal site and some of the available portlets, let's see what we can do to customize the portal .

Introducing the Waxwing Benefits Custom Portal

The remainder of this chapter demonstrates how we customized the Jetspeed portal for our fictitious client, Waxwing Benefits. Waxwing Benefits is a fictitious employee benefits company. The portal provides live, up-to-date access for employees to their employee benefits, accounts and status of ongoing claims. The company also provides access to a collection of company documents via a set of content management portlets. Additionally, we provide useful portlets to its employees such as RSS news feeds and a web-service-based stock quote portlet. Let's start with a look at what the welcome (guest) page will look like when visitors come to the portal:



Figure 12.9 Waxwing Benefits Portal, guest page

(callout) <#1 Welcome Portlet>

(callout) <#2 Login Portlet>

(callout) <#3 a set of CMS Document Viewer portlets available on other guest pages>

The guest page is the entry point into the portal. From here, guests can find out general information about the company. Authenticated users can also login from this portal entry point.

There are several portlets available to the guest:

- Welcome Portlet – a simple portlet displaying the welcome HTML content. This portlet demonstrates displaying the contents of a file in a portlet.
- CMS Document Viewer Portlet – a portlet that displays the content from a CMS document. Here we display the content of four CMS documents: Products, Services, Company Profile, and Contact Us. You can navigate to these portlets from the left navigation menu, as the portlets are located on another page. We will cover programming CMS portlets with Apache Portals Graffito in chapter 14.
- Login Portlet – The standard Jetspeed login portlet for authenticating users to the portal.

The remainder of the chapter will demonstrate how we went from the default Jetspeed-2 portal (displayed in figure 12.1) to the customized portal in figure 12.9. This customization process involves:

- Setting up the portal folder and page structure (section 12.3)
- Formatting the page with Jetspeed layouts and decorators (section 12.4)
- Securing your Portal (section 12.5)
- Defining Portal Integration Points (section 12.6)

Chapters 13 and 14 will then dive into the details of how to develop all the portlets found in the Waxwing Benefits portal. First, let's get started at the beginning: setting up the portal site structure.

Setting up the Portal Site structure: Folders and Pages

A portal site's content is often organized in a file system-like hierarchy. Some portals store all content in a CMS repository, which can also have a similar hierarchical nature to file systems. The Jetspeed portal has the concept of a site being made up of folders and pages. A folder can hold one or more folders and pages, and each folder likewise can hold the same one or more folders and pages. A page holds one or more portlets along with layout and decoration metadata. The Portlet API does not address pages and folders, thus it is entirely up to the portal vendor (or open source project) to implement pages as they wish. In this section we introduce Jetspeed-2's page and folder implementation. You will find that the portal site map is similar to other commercial portal implementations.

In this section, we'll discuss the Waxwing Benefits portal file structure, how to manage user roles, how to secure access to the portal, how to write a simple welcome portlet, and finally how to define integration points into portal.

Organizing the portal folders

Figure 12.9 shows the folder structure of our Waxwing Benefits example portal. By default, Jetspeed contains a root folder. One or more sub-folders can be located under that folder. There are two reserved folders in Jetspeed that must always exist: `_role`, `_user`, and (optionally) `_group`. Reserved folders start with an underscore (`_`).

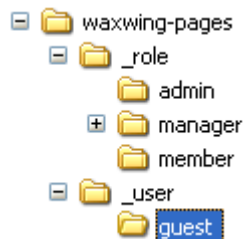


Figure 12.9 Waxwing Benefits portal folders, including the two reserved folders, `_role` and `_user`

Note that we have created our own portal directory structure named “waxwing-pages”. Sometimes this directory is also called “pages”. The content of this directory is completely different from the content in the default Jetspeed distribution. This is one area where we customized the portal.

Exploring the portal pages

The Jetspeed-2 portal stores the site on the file system. Future implementations may store the site in a CMS repository or a relational database. Each folder holds one or more pages. Let's take a closer look at those pages. Figure 12.10 shows the contents of the `_user/guest` folder:

Name	Size	Type
contactus	1 KB	PSML File
default-page	1 KB	PSML File
folder.metadata	1 KB	METADATA File
products	1 KB	PSML File
profile	1 KB	PSML File
services	1 KB	PSML File

Figure 12.10 Content of the guest folder: PSML pages and folder metadata

There are five Portal Structure Markup Language (PSML) pages that correspond to the top level menus (Figure 12.11) available to the guest (un-authenticated) user:

Resources
Waxwing Benefits: Welcome
Products
Services
Company Profile
Contact Us

Figure 12.11 the menu automatically generated from the guest folder pages

Don't worry if you are not too familiar with PSML yet; we'll explain PSML in section 12.4.3... As you can see, it makes it very simple and intuitive to build the site map and navigational menus from folders and pages. We won't get into the Jetspeed-specific details of how to create menus and submenus. Jetspeed provides a rich template API for building navigational menus from folders, pages, and external links. The source code included with this book provides several good examples.

The *folder.metadata* holds additional information such as folder security constraints, sort order, localized titles, and the default page for the folder:

Listing 12.7 Ordering the display of documents

```
<?xml version="1.0" encoding="UTF-8"?>
<folder>
  <title>Waxwing Benefits</title>
  <metadata name='title' xml:lang='fr'>Benefices Waxwing</metadata>

  <default-page>login.psml</default-page>
  <document-order>default-page.psml</document-order>
  <document-order>products.psml</document-order>
  <document-order>services.psml</document-order>
  <document-order>profile.psml</document-order>
  <document-order>contactus.psml</document-order>
</folder>
```

We now have a basic structure for our portal. We now need to consider the roles of different users, and how to organize content to best serve each user role.

Organizing Content for Each User's Role

The Waxwing Benefits Portal uses an authenticated user's *role(s)* to locate the pages to be displayed. Although you can also store a user's pages under the `_user/{username}` directory, role-based pages are a common practice in portals.

The 'member' role represents a group member of an employee benefits plan accessing the portal.

The page provides a customized view for members.

There are two other roles in the Waxwing Benefits portal: 'manager' and 'admin'. When a user with the member role logs on to the system, they are presented with Figure 12.12:

Waxwing Benefits
Always Excellent Benefits and Services

Member Home Forms Search Claims

Claim Status

Member Name: Sir Robin
Member Company: Eruditorum
Benefits Group: 1700-3
Benefits Account: 176-945

Claims In Progress: 2
Claims Processed: 6
Awaiting Action: 1

Deductible Status

	Deductible Caps	Accumulated
Lifetime Maximum	\$10,000,000.00	\$285,000.00
Individual Deductible	\$200.00	\$200.00
Individual Deductible	\$500.00	\$425.00
Individual Out-of-Pocket	\$1,000.00	\$350.00
Family Out-of-Pocket	\$2,500.00	\$950.00

Stock Quote

Symbol	Price	High	Low
"DST"	46.45	46.74	46.4
"IBM"	92.10	92.44	91.5
"ORCL"	12.95	12.99	12.5
"ADSK"	28.58	30.70	28.0

RSS Portlet

NPR News: Top Stories

- ◆ [Bush, Schroeder Emp](#)
President Bush and C
Wednesday and pled
the German city of M
- ◆ [Allawi Announces Bid](#)
Iraq's interim Prime
government. The sec
he faces tough comp
- ◆ [Health Officials Soun](#)
International health
bird flu pandemic in
- ◆ [Calif. Sees State-Fun](#)
A new report says hu
into state-funded pre
that access to presch
- ◆ [Bush Criticizes EU Pla](#)
One of the issues th
that Europe wants to
pro-democracy activi
between China and T

Figure 12.12 The portal as viewed by benefits members

This page is located under the `_role/member` directory. There are four pages in the `_role/member` directory (Figure 12.13):

Name	Size	Type
claims	1 KB	PSML File
folder.metadata	1 KB	METADATA File
forms	1 KB	PSML File
member-home	2 KB	PSML File
search	1 KB	PSML File

Figure 12.13 Content of the member folder: PSML pages and folder metadata

These pages (Figure 12.14) correspond to the four tabbed menu options available to users with the *member* role:



Figure 12.14 The menu automatically generated from the member folder pages

Role-based location of a user's pages is useful for portals where many users with the same role require the same page content. The page to be displayed to a user is found by searching through all the roles for a user. This algorithm does not look in the user's home PSML folder. Users can then start with the same template based on one or more roles. Later on, then they can customize the page to their own user interface. A user's PSML page is found based on an algorithm called a *profiling rule*. Profiling rules are configurable on a per user basis and a global basis for users who do not have a profiling rule. Use the Jetspeed-2 Profiler Admin portlet to configure profiling rules. Typical profiling algorithms include role-based, group-based, role-group-based, and a media-type/language/country-code based algorithm.

Now let's move on to seeing how all the portlet content is brought together under one page by aggregating a page's content with layouts and decorators.

Aggregating the Page Content with Jetspeed Layouts and Decorators

In Jetspeed, a portlet page is the combination of a layout, a layout decorator, portlet decorators, collections of portlets called fragments, and finally portlets.

Rendering one or more portlets into a portlet page is largely made up of the process of aggregating portlets as dynamic content with templates. Jetspeed-2 makes use of Velocity templates in creating a page layout. Although the Jetspeed-2 architecture fully supports the use of JSP templates for decorators and layouts, the developers thus far have chosen Velocity as the tool of choice for writing templates. There are two types of templates in Jetspeed: layouts and decorators. The process of rendering a page is the combined aggregation of a layout template, a page decorator template, a PSML definition, and one or more portlet decorator templates. As of release 2.0, templates can be stored on the file system. In future releases, templates will also be stored in a CMS backend for versioning.

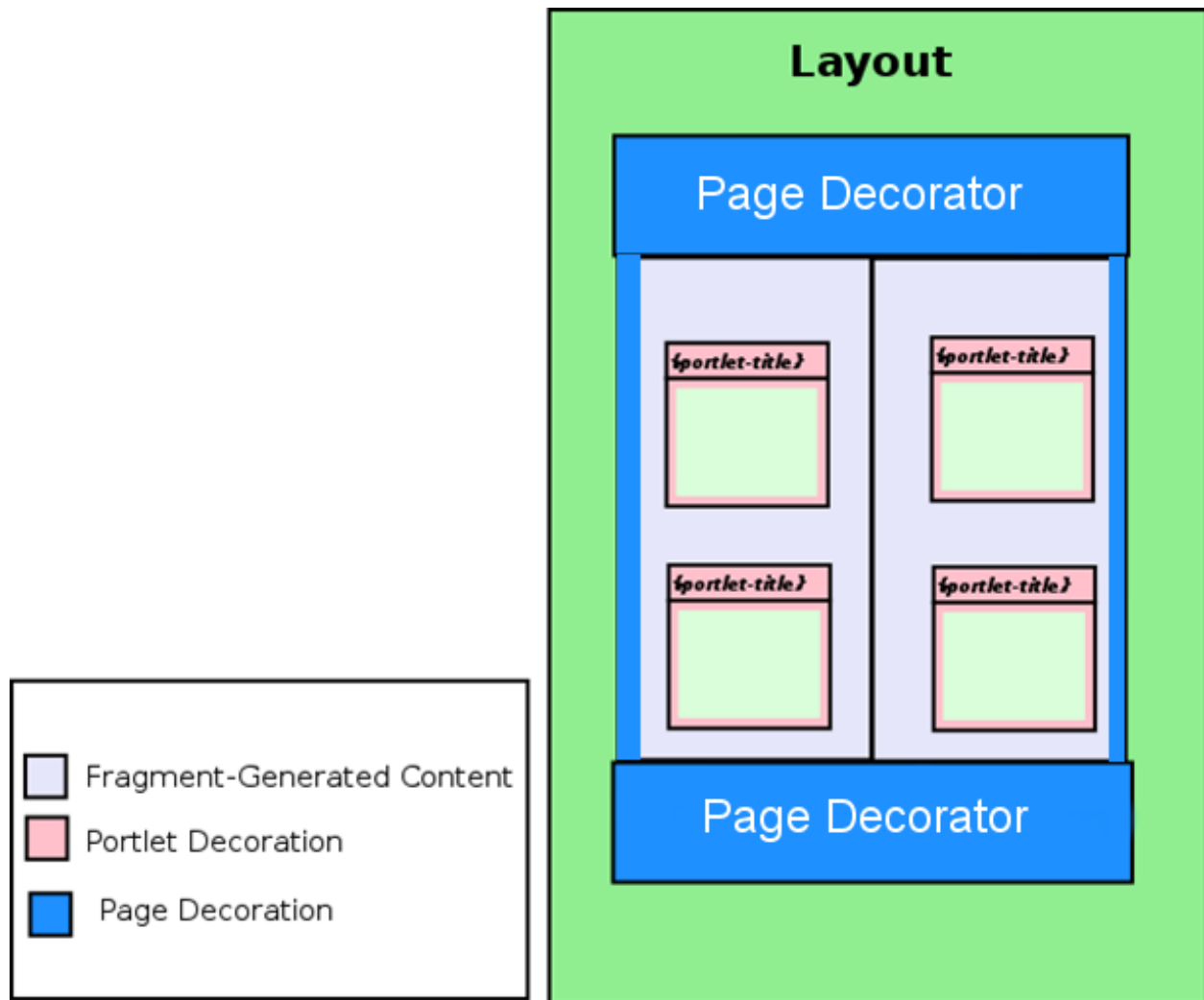


Figure 12.15 A portlet page in Jetspeed is made up of layouts, decorators, fragments and portlets. In this section, we'll cover creating layouts and decorators.

Layouts

Layouts are templates written in JSP or Velocity that control the overall aggregation of a portal page. Layout templates are combined with portlets, providing a component model for aggregation of portlet content.

A layout is made up of:

- One or more templates
- 26. A template descriptor
- 27. Images
- 28. A Stylesheet (CSS)
- 29. Macros

Layouts define how a single portal page is aggregated. A layout defines the fashion in which grouping of fragments (collections of one or more portlets) will be organized relative to the final, aggregated content of a request to the portal. Available layouts with Jetspeed-2 include:

Table 12.2 Jetspeed-2 Layout Choices

Layout	Description
One Column	A single column display of one or more fragments taking up 100% of the portlet display area.
Two Columns	A two column display of one or more fragments where each column is allocated to 50% respectively of the portlet display area. Fragments may be placed in either column using PSML fragment definitions. Percentages can be adjusted to other allocations such as 25%/75%.
Three Columns	A three column display of one or more fragments where each column is allocated to 33% (by default) of the portlet display area respectively. Fragments may be placed in either column using PSML fragment definitions.

The layout used in the Waxwing Benefits portal is a two column layout. The layout is always defined in the page definition (PSML) file. Since layouts are portlets, the definition of a layout is always defined like any other fragment definition: “jetspeed::VelocityTwoColumns”. We will be looking at PSML and fragments in more detail in section 3.4.4.

Listing, Page and Portlet Decorators

Decorators are UI components that are used to either decorate a portlet or a page. A Listing decorator can appear on a portlet or a page. A portlet decorator is the window that goes around the portlet-generated content. A page decorator decorates the entire page. Often, portlet decorators like those in Figure 12.15 display the title and actions used to change the Portlet API standardized modes and states:

Portlet Mode (view, edit, help)

30. Window State (minimize, maximize)

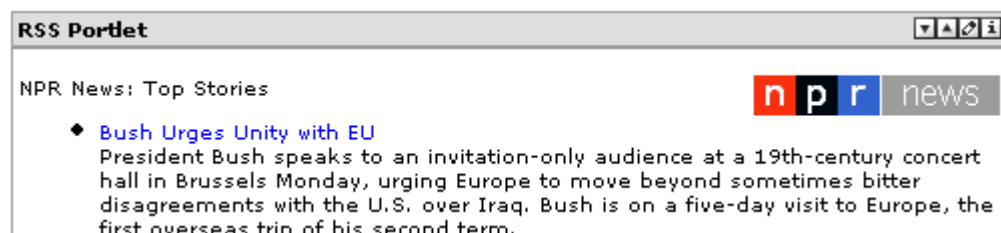


Figure 12.16 Portlet decorators appear in the upper right corner: minimize, maximize, edit, help

The default portlet decorator used on this page is a decorator known as *tigris*. Tigris is an open source community. One project there is called Tigris Style (<http://style.tigris.org/>). The CSS styles defined at Tigris are very popular with Java web applications.

PSML and Fragments

PSML is an acronym for Portal Structure Markup Language. It was created to allow content structure and abstraction within Jetspeed. PSML defines how portlets are aggregated, grouped, and decorated on a portal page. Note that page layout is not a part of the Java Portlet Standard API. Thus PSML is a Jetspeed-specific implementation. Also note that PSML in Jetspeed-2 is different from PSML in Jetspeed-1.

Jetspeed-2 uses a recursive fragment model to define collections of portlets on a page. These collections of portlets are known as fragments. In Jetspeed-2, all fragments are implemented as portlets. A special type of portlet is the layout portlet, which can contain other fragments by nesting the XML fragment definitions as shown in listing 12.9:

Listing 12.9 a PSML fragment

```
<fragment id="wwmh-001" type="layout" name="jetspeed::VelocityTwoColumns">
  <fragment id="wwmh-0.0" type="portlet" name="waxwing::ClaimStatus">
    <property layout="TwoColumns" name="row" value="0" />
    <property layout="TwoColumns" name="column" value="0" />
  </fragment>
</fragment>
```

Layout fragments can contain one or more portlet fragments. Layout fragments may also nest other layout fragments. Nesting of fragments provides a rich component-based layout model for designing page layout. Note that the notation used to reference portlets on a page is

waxwing::ClaimStatus

where *waxwing* is the portlet application name, and *ClaimStatus* is the portlet name defined in the *portlet.xml*.

Editing XML pages is something that is best left to the portal. The software UI component page that edits pages is often called the *Customizer*, since it customizes a page for a particular user.

Using a *customizer*, the end user can select which portlets they want to display on the page, where they want the portlets located, and other preferences customized to the end user. The Page Customizer is accessed by clicking on the pencil icon in the top right corner of the page, as in Figure 12.17.

The screenshot displays a web portal interface with three portlets arranged in a two-column layout. Each portlet has a 'Position' icon in its top right corner, indicating it can be customized. The portlets are:

- StockQuote**: A portlet showing stock market data for IBM, DST, and ORCL. It includes a table with columns: Symbol, Price, High, Low, Change, and Volume.
- Deductible Status**: A portlet showing deductible information. It includes a table with columns: Deductible Caps and Accumulated.
- RSS Portlet**: A portlet displaying NPR News.

To the right of the portlets is a **Claim Status** sidebar with the following information:

- Member Name: Sir Robin
- Member Company: Eruditorum
- Benefits Group: 1700-3
- Benefits Account: 176-945
- Claims In Progress: 2
- Claims Processed: 6
- Awaiting Action: 1

Figure 12.17 Customizing the position of portlet fragments inside of a two column layout

Let's take a look at the default page, "Member Home." This is where we defined the four portlets shown in figure 12.12.

Listing 12.10 Jetspeed's page descriptor language (PSML)

```
<page id="member-home">
  <defaults
    layout-decorator="waxwing-page"
    portlet-decorator="tigris"
  </defaults>
</page>
```

```

/>
<title>Member Home</title>
<fragment id="wmmh-001" type="layout" name="jetspeed::VelocityTwoColumns">
  <fragment id="wmmh-0.0" type="portlet" name="waxwing::ClaimStatus">
    <property layout="TwoColumns" name="row" value="0" />
    <property layout="TwoColumns" name="column" value="0" />
  </fragment>
  <fragment id="wmmh-0.1" type="portlet" name="waxwing::StockQuote">
    <property layout="TwoColumns" name="row" value="0" />
    <property layout="TwoColumns" name="column" value="1" />
  </fragment>
  <fragment id="wmmh-1.0" type="portlet" name="waxwing::DeductibleStatus">
    <property layout="TwoColumns" name="row" value="1" />
    <property layout="TwoColumns" name="column" value="0" />
  </fragment>
  <fragment id="wmmh-1.1" type="portlet" name="rss::RSS">
    <property layout="TwoColumns" name="row" value="1" />
    <property layout="TwoColumns" name="column" value="1" />
  </fragment>
</fragment>

<security-constraints>
  <security-constraint>
    <roles>member</roles>
    <permissions>view</permissions>
  </security-constraint>
  <security-constraint>
    <roles>manager</roles>
    <permissions>edit</permissions>
  </security-constraint>
</security-constraints>
</page>

```

Fragments contain properties for the position of the fragment on the page. In a two column layout, the row and column can be set using the Jetspeed-2 Page Customizer. Next we will look at how to secure portlet pages.

Securing your portal

Arguably one of the most important functions of a portal is its ability to secure portal resources such as portlets, folders and pages. Portals integrate applications and content from diverse areas of an organization. Without careful attention, all users potentially have access to delicate corporate data. Often, these users are not only organizational members, but also partners, customers, guests, or the general public. The portal must be able to grant or deny access to areas of the portal based on the authenticated user and the user's security profile. Portals need a fine grained security policy to handle this level of security.

In this section, we'll discuss security constraints and authentication in the waxwing benefits examples, then we'll show you the Jetspeed Login portlet.

Implementing security constraints

Jetspeed-2 uses two kinds of security:

- JAAS policies

31. declarative security constraints.

Jetspeed-2 comes out of the box with a JAAS security policy implementation. This JAAS policy is stored in a relational database backend.

To stay within the scope of this book, we will have a quick look at the simpler declarative security model, known as Jetspeed Bronco security constraints. A security constraint is an XML element found in a PSML file or in a folder metadata file. Constraints control access to a resource, such as a PSML page. A security constraint can either deny or grant access. The default is to grant access. Access is granted or denied to principals: users, groups or roles. Access is granted or denied based on portal permissions which map to the Portlet API's standard portlet modes: edit, view, and help.

Portal security constraints can be applied to different kinds of principals (user, role, group), pages, and portlets. In the Waxwing Benefits sample, we applied the following security constraint to the default page for the *member* role:

Listing 12.11 PSML: page level security constraints

```
<security-constraints>
  <security-constraint>
    <roles>member</roles>
    <permissions>view</permissions>
  </security-constraint>
  <security-constraint>
    <roles>manager</roles>
    <permissions>edit</permissions>
  </security-constraint>
</security-constraints>
```

Constraints work with principals (role, user, and group) and permissions. Permissions are actions provided by the portal implementation. Jetspeed-2 follows the portlet specification, providing permissions to mirror the default portlet modes: *view*, *edit*, and *help*. The constraint shown in Listing 12.10 constrains access to the default page for members by granting the *view* permission to users with the role member, and granting the *edit* permission to users with the role manager. Similarly, constraints can also be applied to pages and portlets.

Implementing Authentication

Authentication is the process of determining and validating a user's identity. Authentication is usually based on something that the user knows or has a secured copy of, such as a digital certificate. For the Waxwing Benefits example, we will make use of a simple password and a variation on form-based authentication called *active authentication*. Although our example does not use a secure protocol such as HTTPS to communicate over, we strongly recommend using a secure protocol for transferring critical data such as passwords over the internet. Portals often use federated authentication providers to handle the authentication of users.

Security in action: the Jetspeed-2 login portlet

Jetspeed-2 comes with a login portlet that you can use to get started. In our portal example, we make use of the Jetspeed-2 login portlet. The portlet is a part of the Jetspeed-specific security portlet application. The security portlet application is provided with the book's distribution and it includes several security and administrative portlets:

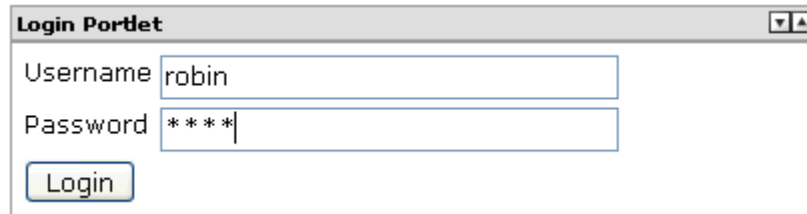
- User Management
- User Attribute Management
- Role and Group Management

Single Sign-on Administration Profiler Maintenance

The security portlet is implemented with this line:

```
<fragment id="wwb-03" type="portlet" name="security::LoginPortlet">
```

Which, in turn, generates this login portlet (Figure 12.18) on the page:



The screenshot shows a web portlet titled "Login Portlet". It contains two input fields: "Username" with the text "robin" and "Password" with four asterisks "****". Below the password field is a "Login" button.

Figure 12.18 The Jetspeed Login Portlet

Note: Jetspeed-2 also comes with a standard Java Login Module (Figure 11.16) so that you can authenticate portal principals in any Java standard application server. Jetspeed-2 authenticates, by default, against a relational database authentication store. A generic LDAP authentication store is also available.

The Waxwing Benefits example portal has four users:

Table 12.3 Waxwing Portal User Database

User Name	Role
admin	admin
manager	manager
robin	member
sparrow	member

To make it easy to use the example portal, all users have the same credential (password): **bird**. Of course in a production site, short passwords are not recommended.

The user is now logged in, and able to customize their page. How does that happen? Let's look.

Creating Content for the Portal Pages

Figure 12.19 shows what the guest welcome page looks like after we have customized it:

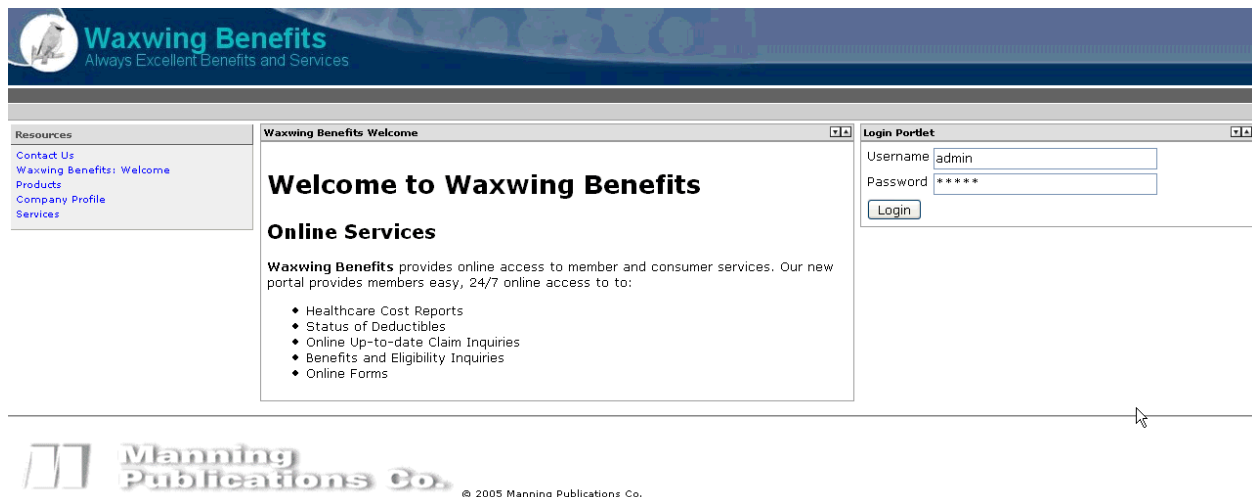


Figure 12.19 Unauthenticated users get the Waxwing Benefits portal welcome (guest) page when they visit.

The guest page is the entry point into the portal. From here, guests can find out general information about the company. Authenticated users can also login from this portal entry point.

So, what did we customize and how did we do it? There are several portlets available to the guest:

- The Welcome Portlet is the center of attention in the middle of the screen.
- The CMS Document Viewer Portlet on the left gives unauthenticated users access to a limited set of documents about Waxwing Benefits.
- The Login Portlet is on the right, and opens the door to further customizations.

In this section, we'll tell you more about each portlet and how they work.

Accessing content from a file: the Welcome Portlet

The Welcome Portlet displays HTML text from a resource file located in the portlet application distribution. The portlet descriptor defines [

Listing 12.13 portlet.xml deployment descriptor for the FilePortlet

```
<portlet>
  <description>Welcome to Waxwing Benefits</description>
  <portlet-name>Welcome</portlet-name>
  <display-name>Welcome to Waxwing Benefits</display-name>
  <portlet-class>
    com.waxwingbenefits.portal.portlets.FilePortlet
  </portlet-class>
  <supports>
    <mime-type>text/html</mime-type>
    <portlet-mode>VIEW</portlet-mode>
  </supports>
  <supported-locale>en</supported-locale>
  <portlet-info>
    <title>Waxwing Benefits Welcome</title>
    <short-title>Waxwing</short-title>
    <keywords>benefits, waxwing, html</keywords>
  </portlet-info>
  <portlet-preferences>
    <preference>
      <name>file</name>
      <value>idtb_home.htm</value>
    </preference>
  </portlet-preferences>
</portlet>
```

```

        </preference>
    </portlet-preferences>
</portlet>

```

The amount of code required for the File Portlet is minimal.

Listing 12.14 the FilePortlet class listing

```

public class FilePortlet extends GenericServletPortlet
{

    public void doView(RenderRequest request, RenderResponse response)
    throws PortletException, IOException
    {
        response.setContentType("text/html");
        PortletPreferences prefs = request.getPreferences();           |#1
        String fileName = prefs.getValue("file", null);
        if (fileName != null)
        {
            InputStream is = this.getPortletContext().getResourceAsStream(fileName);
            drain(is, response.getPortletOutputStream());               |#2
            is.close();
        }
        else
        {
            response.getWriter().println("Could not find file preference ");
        }
    }

    static final int BLOCK_SIZE=4096;

    public static void drain(InputStream r,OutputStream w) throws IOException
    {
        byte[] bytes=new byte[BLOCK_SIZE];
        try
        {
            int length=r.read(bytes);
            while(length!=-1)
            {
                if(length!=0)
                {
                    w.write(bytes,0,length);
                }
                length=r.read(bytes);
            }
        }
        finally
        {
            bytes=null;
        }
    }
}

```

(Annotation) <#1 Annotation here>

(Annotation) <#1 Annotation here>

The name of the file is stored as a Portlet Preference named file. The value in the *portlet.xml* is a default value. Preferences can be optionally overridden on a per user basis. The preference name is “file.”

Using the portlet context, the file is located as a resource relative to the web application. We then get a stream to the HTML file, and use a helper function to drain the stream out the portlet response stream:

Listing 12.15 retrieving a portlet’s preferences to lookup the filename

```
PortletPreferences prefs = request.getPreferences();
String fileName = prefs.getValue("file", null);
if (fileName != null)
{
    InputStream is = this.getPortletContext().getResourceAsStream(fileName);
    drain(is, response.getPortletOutputStream());
    is.close();
}
```

The resulting streamed output is displayed in a portlet window (Figure 12.7):

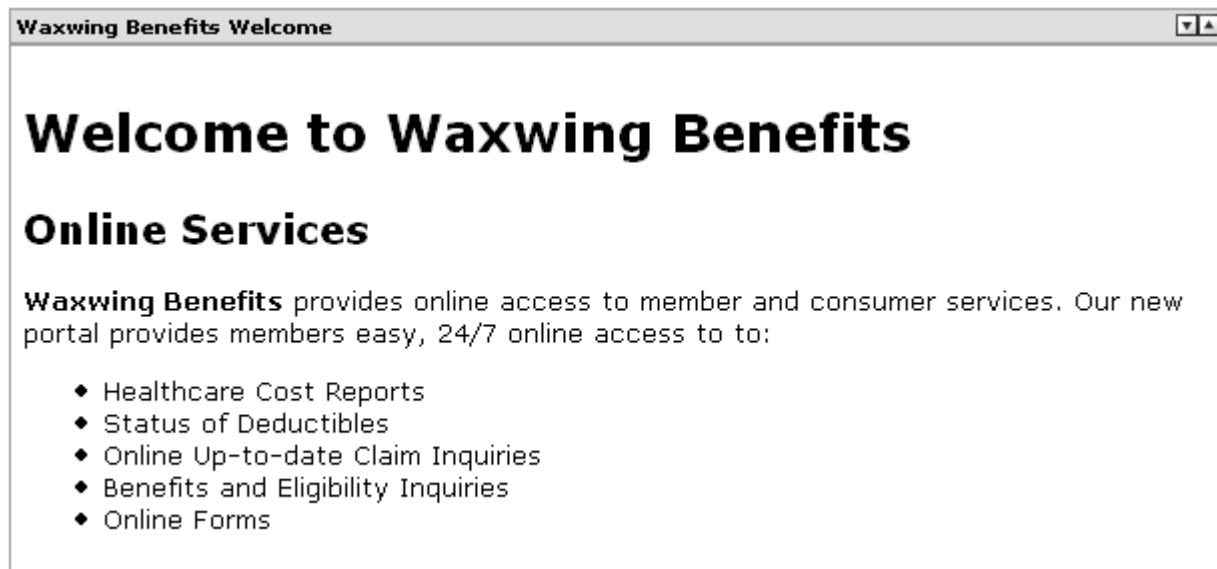


Figure 12.20 The Waxwing Benefits Welcome Portlet describes the portal to the unauthenticated user. [[say more]]

Although this file portlet is useful, it is often not very practical to store content inside the web application. Let’s now look at a more practical way of managing portlet content.

Accessing content from a CMS: the CMS Document Viewer Portlet

Content Management Systems (CMS) provide a standard way of managing portal content. A CMS provides a repository for storing and sharing documents in a community or enterprise. With a CMS, you can check out and version documents, secure access to the document, and provide workflows and transforms to the content. The Apache Portals Graffito project is about integrating CMS servers with portals. In this section we will use Graffito’s CMS API to access content in a CMS.

Graffito acts as a Virtual Content Management system capable of supporting WebDAV, the Java Content Repository Standard API (JCR), or any other proprietary content system. It contains flexible services that can transparently support different repositories. Your application can access

heterogeneous content stores. Graffito accesses all CMS content using a normalized URI access path. Thus content stores from different CMS repositories all work together in a normalized URI space. As of this writing, Graffito works with Jakarta Slide, and any CMS backend supporting the WebDAV protocol. JCR API support is in the works.

In our Waxwing portal example, we store our CMS in a hybrid CMS backend repository, with the documents stored on the file system and the metadata about the documents stored in a database. You will learn much more about Graffito CMS in Chapter 14.

Here is the portlet.xml for a single portlet entry for the Company Profile portlet:

Listing 12.16 portlet.xml deployment descriptor for the Company Profile portlet

```
<portlet>
  <description>Company Profile</description>
  <portlet-name>CompanyProfile</portlet-name>
  <display-name>Company Profile</display-name>
  <portlet-
class>com.waxwingbenefits.portal.portlets.CMSDocumentPortlet</portlet-class>
  <init-param>
    <name>EditPage</name>
    <value>/WEB-INF/views/edit-document.vm</value>
  </init-param>

  <supports>
    <mime-type>text/html</mime-type>
    <portlet-mode>VIEW</portlet-mode>
    <portlet-mode>EDIT</portlet-mode>
  </supports>
  <supported-locale>en</supported-locale>
  <portlet-info>
    <title>Company Profile</title>
    <short-title>Profile</short-title>
    <keywords>CMS, company, profile</keywords>
  </portlet-info>
  <portlet-preferences>
    <preference>
      <name>document-uri</name>
      <value>/waxwing/company/profile.html</value>
    </preference>
  </portlet-preferences>
</portlet>
```

Note that the preference: “document-uri” is a *Graffito*-normalized URI. Our repository is mounted on an access point named “waxwing.” Under the access point, we simply follow the file system paths. Graffito is configured as a *Spring* component in a *Spring* configuration file which is loaded at portlet application start-up. Here is the component definition for the Graffito Content Server interface:

Listing 12.17 Spring configuration for a Graffito CMS Service

```
<bean id="cmsServerImpl"
  class="com.waxwingbenefits.cms.impl.WaxwingContentServerServiceImpl" >
  <constructor-arg index="0">
    <value>/bluesunrise/manning/cms/stage/</value>
  </constructor-arg>
</bean>
```

A file system path is used to identify the path to the CMS root directory. Note that we called this directory “stage.” Often CMS repositories are too slow to support the heavy load of a portal site. The CMS just can’t keep up with the portal site hits. One solution is to put the CMS document in a staging area, or file system cache, where the portal can access these documents more quickly. Many CMS/Portal solutions use replication engines to synchronize the content between the CMS and the file system staging area. Graffito can be used to access both the CMS and the staged content. In our examples, our Graffito implementation hits the staged file system. Let’s have a look at the *doView* method of the *CMSDocumentPortlet*:

Listing 12.18 CMSDocumentPortlet’s doView method

```
public void doView(RenderRequest request, RenderResponse response)
throws PortletException, IOException
{
    response.setContentType("text/html");

    String currentFile = (String) PortletMessaging.receive(request,
"FileDisplayer",
"current_file");

    String fileName = null;
    PortletPreferences prefs = request.getPreferences();
    String uri = prefs.getValue("document-uri", null);
    if (uri != null && uri.length() > 0)
    {
        currentFile = uri;
    }
    if (currentFile != null)
    {
        ContentModelService cms =
            (ContentModelService) SpringAccessor.getBean("cmsModelImpl");

        try
        {
            Document doc = cms.getDocument(currentFile);
            if (doc.getContentType().indexOf("html") != -1)
            {
                String currentDocument = "<div>Current Document: <b>" +
doc.getProperty("title") + "</b> - " + doc.getUri() + "</div>";
                String anchors = "<hr/>"; //createAnchors(doc);

                response.getPortletOutputStream().write(currentDocument.getBytes());

                response.getPortletOutputStream().write(anchors.getBytes());

                response.getPortletOutputStream().write(DIV_START.getBytes());
                drain(doc.getContent().getContentStream(),
                response.getPortletOutputStream());

                response.getPortletOutputStream().write(DIV_END.getBytes());

                int hash = currentFile.indexOf('#');
                if(hash != -1)
```

```

        {
            String anchor = currentFile.substring(hash+1);

response.getPortletOutputStream().write(JAVASCRIPT.getBytes());

            response.getPortletOutputStream().write("<script>setAnchor('."getBytes());

            response.getPortletOutputStream().write(anchor.getBytes());

            response.getPortletOutputStream().write("');</script>."getBytes());
        }
    }
    else if (doc.getContentType().indexOf("pdf") != -1)
    {
        renderPDF(request, response, currentFile, doc);
    }
}
catch (ContentManagementException e)
{
    // TODO Auto-generated catch block
    e.printStackTrace();
}
}
}

```

The CMSDocumentPortlet pulls content out of the CMS using the Graffito API, and renders the document content into a scrollable document viewer. Walking through the doView method, lets see how it works:

Set the response type:

```
response.setContentType("text/html");
```

The viewer can be used with a Tree View portlet that navigates over the CMS content. These two portlets communicate using the PortletMessaging class. Since version 1.0 of the Portlet API does not support inter-portlet messaging, Jetspeed provides a simple portable implementation. Here, we receive a message name “current_file” under the topic “FileDisplay.” This is how the Tree View portlet informs the Document Portlet of the selected file.

```

String currentFile = (String) PortletMessaging.receive(request,
"FileDisplay",
"current_file");

```

Note that we receive the message. This leaves the message in the message queue. If we wanted to receive the message, and then remove it, we could have called consume:

```

String currentFile = (String) PortletMessaging.consume( request,
"FileDisplay",
"current_file");

```


This implementation of the Document Viewer allows for the document to be configured. A configured document overrides the messaging behavior. The configuration is done with a portlet preference, the “document-uri” preference. This is the case for retrieving our documents displayed to the Guest user:

Listing 12.19 DocumentViewerPortlet: overriding the selected document with the configured document

```
String fileName = null;
PortletPreferences prefs = request.getPreferences();
String uri = prefs.getValue("document-uri", null);
if (uri != null && uri.length() > 0)
{
    currentFile = uri;
}
```

Now we can get the Content Model Service. The Content Model Service is a Graffito API. It concerns itself with retrieving and storing CMS content. The Graffito component is retrieved from the *Spring* IOC container:

```
ContentModelService cms =
    (ContentModelService)SpringAccessor.getBean("cmsModelImpl");
```

And then the Graffito `getDocument()` API is called on the normalized CMS URI:

```
Document doc = cms.getDocument(currentFile);
```

We then check the content type of the CMS document. Our example portlet handles either HTML or PDF documents:

```
if (doc.getContentType().indexOf("html") != -1)
{
```

From here we do a few fancy things like putting the document in an HTML scrollable DIV area, but basically the essence is:

```
drain(doc.getContent().getContentStream(),
response.getPortletOutputStream());
```

All Graffito documents return a content stream. This stream contains the bytes that make up the document. In this case it’s an HTML document. Once again we use our helper function to drain the contents from the CMS document into the portlet output stream.

The end result is a Document Viewer for CMS content (Figure 12.21). In Chapter 14.1.4, we will see how to view PDF files.

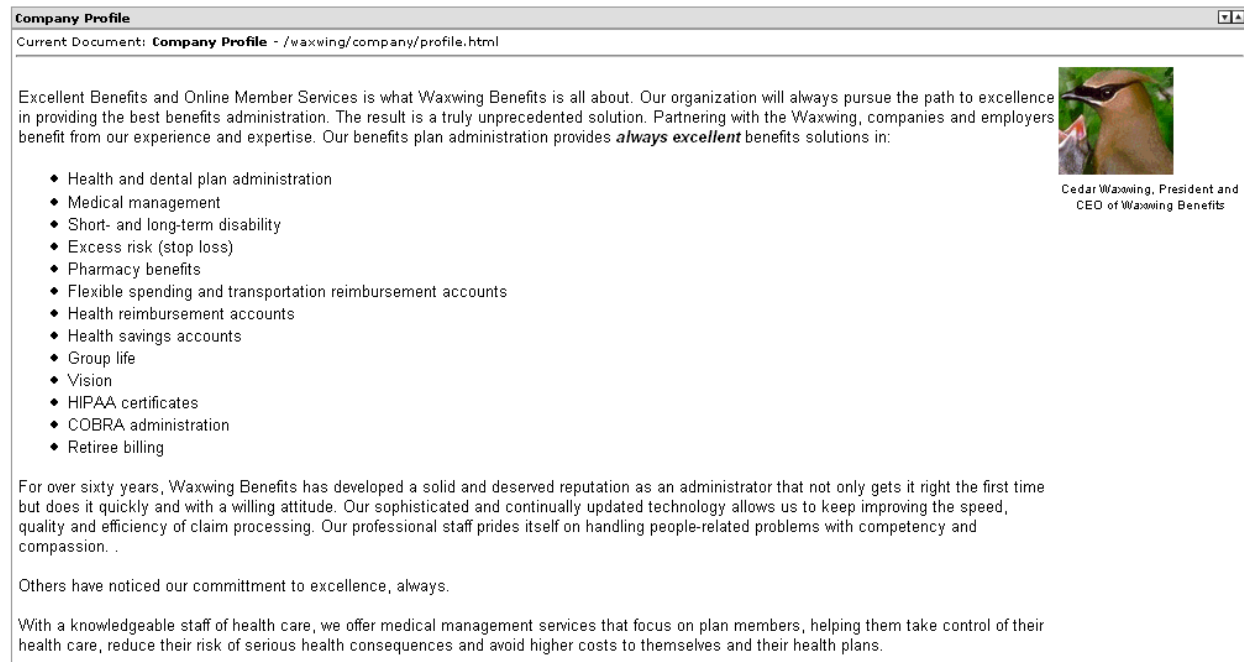


Figure 12.21 The Waxwing Benefits Company Profile document in the CMS Document Viewer portlet

We will visit the Login portlet in the next chapter, when we discuss securing your portal. Next let's look at how we layout the portal site with folders and pages.

Planning Your Portal Integration Points

Integration points are clearly defined API interfaces for accessing information outside of the portal. Whenever possible, you want to use Java standards-based interfaces as your integration points to your portlets. Typical integration points are databases, CMS, web services, XML content, EJBs, Spring components, and company programming APIs. Identifying your integration points to your portlets is an important step in designing your portlet applications. We strongly recommend that you do not move lots of business logic into your portlets. Portlets should be concerned with user interaction foremost. Most business logic should be encapsulated in reusable business objects, not in the portal.

With Waxwing Benefits, we have designed a basic application that would typically need to integrate with the following components:

Table 12.4 Waxwing Benefits Integration Points

Integration Point	Purpose	Implementation
Backend database	To access Claim and Deductible Information	In our implementation integrates with the Java JDBC API to access the claims and benefits database tables.
User information attributes	To access common, user information provided by the portal such as user name and contact information. User information attributes are common to all portlet applications and accessible via a standard Java API.	Our implementation integrates with the Portlet API standard to retrieve user information attributes
CMS	To provide company documents and forms to members	Our implementation integrates with the Apache Portals Graffito API to retrieve CMS content in a normalized manner.
Search engine	To provide free-text search of company documents and forms to members	Our implementation integrates with the Portals Graffito API to search CMS

		content in a normalized manner. Graffito in turn searches and indexes the content with Apache Lucene.
Web service	to provide important financial information to clients (in this simple example, its just stock quotes)	Our implementation integrates with the Apache Axis web services APIs to handle all web service implementation details.
RSS news feeds	to provide members with news and other relevant industry information	Our implementation integrates with the Rome RSS project to handle manipulation of RSS content.

You can see the model for this portal in Figure 12.22:

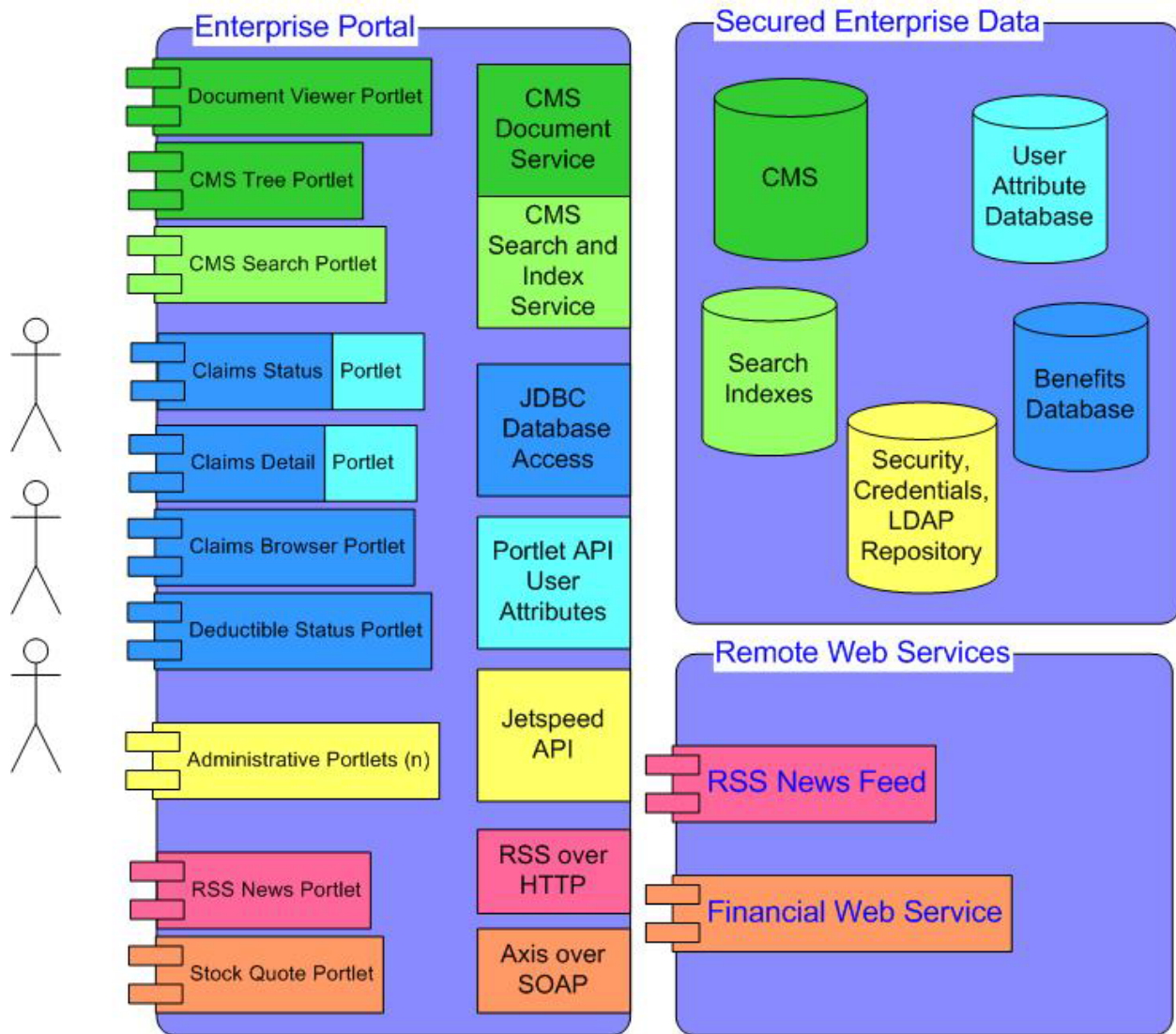


Figure 12.22 Integration Point View of the portal with portlets, service components, and data sources

All of the integration points defined here are covered in more detail in chapters 13 and 14. Before starting on the coding of your portlets, it's always good to clearly define the integration points into your portlets.

Summary

This chapter demonstrated how to create a working portal with standard portlets and open source portal technologies at Apache Portals. The sample portal, Waxwing Benefits, we created in this chapter is based on the Jetspeed-2 open source portal. However, all of the portlet examples are JSR-168 compliant and thus portable to portals from other vendors.

We familiarized you with the default Jetspeed-2 portal included with the book. Then we created a directory structure that helped create role-based custom content for each user role. Next, we integrated the Jetspeed security portal to provide a login and security structure, and showed you how to create the user database. We reviewed creating a Welcome portal, and introduced the Graffito content management system (about which there will be more in Chapter 14). Finally, we discussed planning the integration points for the portal page—what portlets you need and where you want to display them.

The portal that we designed in this chapter can be used as a foundation for you to create your own working portal. In the next chapter, we'll add some additional portlets to the Waxwing Benefits portal so that we can access a database and present the result to the portal user.

References

1. The Rome Project: <https://rome.dev.java.net/>
2. Apache Lucene: <http://jakarta.apache.org/lucene/docs/>
3. Apache Axis: <http://ws.apache.org/axis>

Chapter 13 Implementing Database Access

After setting up our own Waxwing Benefit portal in the previous chapter, we're ready to start developing more advanced portlets in this chapter. This chapter is about accessing data in backend systems and integrating that data in portlets using standard methods of data access. We will show you how to integrate portlets with data sources via the integration points outlined in table 13.1:

Data Source	Integration Point	Section
Relational Database	JDBC – Java Database Connectivity API	13.2, 13.5
LDAP or User Information Database	Portlet API User Attribute API	13.3
Content Management Systems	Apache Portals Graffito API	Chapter 14
Security Databases	Jetspeed Service API	13.6

Table 13.1 Data Sources and Integration Points demonstrated in this chapter

In section 13.1, we introduce the portlets covered in this chapter. In 13.2, we'll leverage JDBC to access the Waxwing Benefits corporate relational database to generate claims status and benefits reports within a portlet. In section 13.3, we demonstrate how to retrieve generic user information with the Portlet API's User Attributes API. Section 13.4 demonstrates using Velocity to display the data without coupling the data source (model) to the (view). Section 13.5 demonstrates extending the Jetspeed Database Browser portlet to create our own claims browser portlet. In section 13.6, we show how two portlets, a browser and a data entry form, can communicate using portlet messages.

About the Waxwing Benefits member portlets

Before we get into the details of database access and portlets, let's put our custom portal in context. Waxwing Benefits manages benefit packages for companies. Waxwing *members* are employees of companies who partner with Waxwing Benefits, and who often visit the benefits portal. These employees are usually members of a benefit group, such as a medical benefits group.

In this chapter, we are going to look at the database-related portlets provided to users who are assigned the role *member*. As we discussed in chapter 12, the Waxwing Benefits portal implements three roles: Member, Manager, and Admin. When members authenticate (login), they are provided with access to a specific set of portlets. These portlets are designed to meet the needs of the benefits members and are listed in table 13.2:

Member-specific Portlet	Purpose	Data Source	Page
Claims Status	Displays a report of a member's claim activity via JDBC from a corporate relational database; and user account information	JDBC, User Attributes	Member Home

	via Portlet API User Attributes from portal' s user attribute database.		
Deductible Status	Displays a report of a member' s deductible account details and accumulated benefits paid via JDBC from corporate relational database	JDBC,	Member Home
Claims Browser	Displays a report of all claims filed by a member via JDBC and the database browser from a corporate relational database	JDBC	Claims
Claims Detail	Displays details of a member' s single claim via JDBC from a corporate relational database; and user account information via Portlet API User Attributes from portal' s user attribute database.	JDBC, User Attributes	Claims

Table 13.2 Chapter 13 Portlets

In the sections that follow, we'll show you how to add these portlets to the Waxwing portal. All of the examples in this chapter follow the MVC (Model View Controller) design pattern of portlet development. The data sources are wrapped as the 'model' in beans or as standard attributes. The 'view' is handled in this chapter using Velocity templates. The portlet is the 'controller'. In section 13.2 we demonstrate using beans retrieved from a JDBC data source as the model, the portlet as the controller, and Velocity templates as the view.

Database Access using JDBC: the Claim Status Portlet

JDBC is the standard method for accessing databases in Java. Although most projects layer an object-relational mapping tool on top of JDBC, in this chapter we keep things standard and simple, going straight to the JDBC standard.

The Claim Status Portlet displays information about the member's account in the top half of the portlet view. In the bottom half, the member's benefit claims summary is displayed. The top half of the Claim Status Portlet is retrieved using User Attributes and the Portlet API. The bottom half is retrieved using JDBC from the Waxwing Benefits corporate database. This section concentrates on the JDBC solution.

The screenshot shows a portlet titled "Claim Status". It contains two main sections. The top section displays member account information in a table-like format: Member Name: Chip Sparrow, Member Company: Eruditorum, Benefits Group: 1700-3, and Benefits Account: 176-946. The bottom section displays a claim summary with three rows: Claims In Progress: 2, Claims Processed: 6, and Awaiting Action: 1.

Claim Status	
Member Name:	Chip Sparrow
Member Company:	Eruditorum
Benefits Group:	1700-3
Benefits Account:	176-946
Claims In Progress:	2
Claims Processed:	6
Awaiting Action:	1

Figure 13.1 The Claim Status Portlet

Let's have a look at the Claim Status Portlet in Figure 13.1. The *doView* method handles looking up the member account information and claim summary information.

Listing 13.1 ClaimStatus portlet's doView method

```
public void doView(RenderRequest request, RenderResponse response)
    throws PortletException, IOException
{
    Map userInfo =
        (Map) request.getAttribute(PortletRequest.USER_INFO); | #1
```

```

Context context = super.getContext(request);
String firstName = (String)userInfo.get("user.name.given");
context.put("firstName", userInfo.get("user.name.given"));
context.put("lastName", userInfo.get("user.name.family"));
context.put("company", userInfo.get("user.employer"));
context.put("group", userInfo.get("benefits.group"));
context.put("account", userInfo.get("benefits.account"));

ClaimStatus status = (ClaimStatus)
    request.getPortletSession().getAttribute("claimStatus");
if (status == null)
{
    // produce report
    Connection con = null;
    try
    {
        con = cms.getConnection();                | #2
        Integer processed =                       | #3
            countClaimStatus(con,
                request.getUserPrincipal().toString(), "PROCESSED");
        Integer inProcess =
            countClaimStatus(con,
                request.getUserPrincipal().toString(), "*** OPEN ***");
        Integer awaitingAction =
            countClaimStatus(con,
                request.getUserPrincipal().toString(), "*** HOLD ***");
        status = new ClaimStatus(processed, inProcess,
            awaitingAction);
        request.getPortletSession().setAttribute("claimStatus",
            status);
    }
    catch (SQLException e)
    {
        System.err.println("Exception retrieving claim status: " + e);
    }
    finally
    {
        releaseConnection(con); | #4
    }
}

context.put("claimStatus", status);                | #5
super.doView(request, response);                    | #6
}

```

(annotation) <#1 retrieving user attributes, see section 13.3 >

(annotation) <#2 get a JDBC connection>

(annotation) <#3 calculate claim status report>

(annotation) <#4 release the JDBC connection>

(annotation) <#5 the claims model bean is put in the velocity view>

(annotation) <#6 let the velocity portlet bridge handle processing the view>

We're going to skip over the first eight lines of code in the doView method, as that is covered in section 13.3 on User Attributes. The section concentrates on retrieving the claims status data from a relational database via JDBC, starting with getting a connection.

Accessing the Benefits Database in the Claims Status Portlet via JDBC

Members login to the portal to check their claims status. The claims status portlet accesses the corporate benefits database to produce a claims status report for the current user. The status report provides up-to-date claim information on a per-user basis. The current user is determined using the Portlet API. Specifically, the *PortletRequest* class provides a *getUserPrincipal* method which is then passed into the report generation code:

```
Integer processed =
    countClaimStatus(con,
        request.getUserPrincipal().toString(), "PROCESSED");
```

The portal's user principal is also stored as a column in the claims table of the claims database. When calculating our report, we only count claims that are for the current user principal. Since retrieving from a database can sometimes be an expensive operation, we demonstrate how to cache the claims status report in the portlet session. By caching the calculated report data in the portlet session, the report only calculates once per user, when the user starts a new session. By logging out, the user terminates the portlet session. The next time the user logs on; the report result is cleared from the portlet session and thus must be regenerated. Here is how to get an attribute out of the portlet session using the Portlet API:

```
ClaimStatus status = (ClaimStatus)
    request.getPortletSession().getAttribute("claimStatus");
if (status == null)
{
```

We are looking for an attribute named *claimStatus*. Since we put the attribute in the session, we know it is of the type *ClaimStatus*. When a user hits this page for the first time after logging on, the attribute is not found and we generate the report:

Listing 13.2 Generating a claim status report

```
// produce report
Connection con = null;
try
{
    con = cms.getConnection();
    Integer processed =
        countClaimStatus(con,
            request.getUserPrincipal().toString(), "PROCESSED");    |#1
    Integer inProcess =
        countClaimStatus(con, request.getUserPrincipal().toString(),
            "*** OPEN ***");    |#2
    Integer awaitingAction =
        countClaimStatus(con, request.getUserPrincipal().toString(),
            "*** HOLD ***")    |#3;
    status = new ClaimStatus(processed, inProcess, awaitingAction);
    request.getPortletSession().setAttribute("claimStatus", status);
}
catch (SQLException e)
{
    System.err.println("Exception retrieving claim status: " + e);
}
finally
{
    releaseConnection(con);
```



```

    }
(annotation) <#1 count claims for PROCESSED status>
(annotation) <#2 count claims for OPEN status>
(annotation) <#3 count claims for HOLD status>

```

Note that we grab a database connection once, and share it across three method calls to *countClaimStatus*. The *countClaimStatus* method counts the number of claims for the given status:

Listing 13.3 JDBC code to calculate a claim status count

```

public Integer countClaimStatus(Connection con, String principal, String status)
    throws SQLException
{
    String select =
        "select COUNT(*) from WWCLAIMS where USER_ID = ? and status = ?";
    PreparedStatement stm = con.prepareStatement(select);
    stm.setString(1, principal);
    stm.setString(2, status);
    ResultSet rs = stm.executeQuery();
    rs.next();
    int x = rs.getInt(1);
    rs.close();
    stm.close();

    return new Integer(x);
}

```

Although we have used JDBC directly here, in modern Java applications it is better to use object-relational mapping tools. Two popular, open source object-relational tools are the Hibernate or Apache OJB projects. For simplicity and a common ground of understanding, we use the JDBC standard.

A new *ClaimStatus* object is created:

```
status = new ClaimStatus(processed, inProcess, awaitingAction);
```

The claim status is then put in the session and cached:

```
request.getPortletSession().setAttribute("claimStatus", status);
```

Finally, the *ClaimStatus* object is put in the Velocity context:

```
context.put("claimStatus", status);
```

From there, our template can access the *ClaimStatus* object to render the report output:

Listing 13.4 Velocity template displayed calculated claim status in a portlet's view

```

<hr/>
<table>
  <tr>
    <td class='portlet-form-label'>Claims In Progress:</td>
    <td class='portlet-section-selected'>
      $claimStatus.InProgress    | #1
    </td>
  </tr>
</table>

```

```

</tr>
<tr>
    <td class='portlet-form-label'>Claims Processed:</td>
    <td class='portlet-section-selected'>
        $claimStatus.Processed      | #2
    </td>
</tr>
<tr>
    <td class='portlet-form-label'>
        Awaiting Action:
    </td>
    <td class='portlet-section-selected'>
        $claimStatus.AwaitingAction    | #3
    </td>
</tr>
</table>
(annotation) <#1 display the count of IN PROGRESS claims>
(annotation) <#2 display the count of PROCESSED claims>
(annotation) <#3 display the count of IN AWAITING ACTIONclaims>

```

This gives us the claim status report (Figure 13.2) displayed in the bottom half of the portlet output:

Claims In Progress:	2
Claims Processed:	6
Awaiting Action:	1

Figure 13.2 The rendered output of the Claim Status portlet

The *ClaimStatus* is an inner class stored in our portlet:

Listing 13.5 The ClaimStatus bean

```

public class ClaimStatus
{
    private Integer processed;
    private Integer inProgress;
    private Integer awaitingAction;

    public ClaimStatus(Integer processed, Integer inProgress,
        Integer awaitingAction)
    {
        this.processed = processed;
        this.inProgress = inProgress;
        this.awaitingAction = awaitingAction;
    }

    /**
     * @return Returns the awaitingAction.
     */
    public Integer getAwaitingAction()
    {
        return awaitingAction;
    }
}

```

```

        * @return Returns the inProgress.
        */
    public Integer getInProgress()
    {
        return inProgress;
    }
    /**
     * @return Returns the processed.
     */
    public Integer getProcessed()
    {
        return processed;
    }
}
}

```

Next we look into how the top half of the claims status report is generated using User Attributes from the Portlet API.

Accesssing user attributes

The member account details are gathered using a standard Portlet API feature called *User Attributes*. User Attributes are a set of shared attributes about the current user made available by the portal to every portlet that asks for them. The Portlet API requires portals to provide a set of user attributes available at runtime. This set of user attributes are requested by portlets by calling the *getAttribute* method on the *PortletRequest* class (and thus both the *RenderRequest* and *ActionRequest*):

```
Map userInfo = (Map) request.getAttribute(PortletRequest.USER_INFO);
```

The map provided is a subset of all available user attributes to this portlet application. The deployment descriptor defines the user attributes required by the portlet application. The portal only populates the User Info map with user attributes specified in the portlet application's deployment descriptor, shown in Listing 13.6:

Listing 13.6 User Attributes in the portlet.xml deployment descriptor

```

<user-attribute>
  <description>User Given Name</description>
  <name>user.name.given</name>
</user-attribute>
<user-attribute>
  <description>User Last Name</description>
  <name>user.name.family</name>
</user-attribute>
<user-attribute>
  <description>Employer</description>
  <name>user.employer</name>
</user-attribute>
<user-attribute>
  <description>Benefits Group</description>
  <name>benefits.group</name>
</user-attribute>
<user-attribute>

```

```

<description>Benefits Account</description>
<name>benefits.account</name>
</user-attribute>

```

The Portlet API does not specify how user attributes are configured or maintained in a portal. Jetspeed has the ability to retrieve user attributes from one or more user attribute services. This scenario can be very useful in a real world portal. For example, in our benefits company, we could provide some user attributes from the Benefits database, while other user attributes are retrieved from a centralized LDAP repository. By logging on as the *admin* user, you can manage all user attributes in the User Management portlet (Figure 13.3):

User Detail Information		
Principal : TOMCAT		
Attributes		Password
		Role
		Group
		Profile
	Name	Value
<input type="checkbox"/>	user.name.given	Tom
<input type="checkbox"/>	user.name.family	Cat
<input type="checkbox"/>	user.employer	Apache
<input type="checkbox"/>	benefits.group	3939-23
<input type="checkbox"/>	benefits.account	99533-1

Update Remove

Figure 13.3 Jetspeed User Management portlet: editing user attributes

After retrieving the user attributes, we need to include them in the rendered content. For this example, we use the Velocity Bridge and Velocity template engine. Velocity makes use of a simple context object to hold Java objects that can be accessed in a template. The objects are put into the context in the Java portlet code:

Listing 13.7 Putting user attributes in the Velocity context

```

Context context = super.getContext(request);
String firstName = (String)userInfo.get("user.name.given");
context.put("firstName", userInfo.get("user.name.given"));
context.put("lastName", userInfo.get("user.name.family"));
context.put("company", userInfo.get("user.employer"));
context.put("group", userInfo.get("benefits.group"));
context.put("account", userInfo.get("benefits.account"));

```

The Java objects are then pulled out of the context in the Velocity template using \$ notation. See Chapter 11 for more details on using the Velocity Bridge:

Listing 13.8 A Velocity template takes the user attributes out of the Velocity context

```

<table border="1" cellspacing="1" cellpadding="3">
  <tr>
    <td class='portlet-form-label'>Member Name:</td>
    <td class='portlet-section-alternate'>${firstName} ${lastName}</td>      | #1
  </tr>
</table>

```

```

</tr>
<tr>
    <td class='portlet-form-label'>Member Company:</td>
    <td class='portlet-section-alternate'>$company</td>> | #2
</tr>
<tr>
    <td class='portlet-form-label'>Benefits Group:</td>
    <td class='portlet-section-alternate'>$group</td>> | #3
</tr>
<tr>
    <td class='portlet-form-label'>Benefits Account:</td>
    <td class='portlet-section-alternate'>$account</td>> | #4
</tr>
</table>
(annotation) <#1 display first and last name>
(annotation) <#2 display company>
(annotation) <#3 display group>
(annotation) <#4 display account>

```

All user attributes are stored and accessed as strings. (For simplicity, we have not localized the labels used in this example.) This gives us the member account information displayed in the top half of the portlet output (Figure 13.4):

Claim Status	
Member Name:	Sir Robin
Member Company:	Eruditorum
Benefits Group:	1700-3
Benefits Account:	176-945

Figure 13.4 User Attributes rendered in the Claim Status portlet

This completes our coverage of user attributes in this chapter. We have learned how to access a Waxwing Benefits user account information from the portal using the standardized User Attributes API. The Portlet API shielded us from the actual implementation of the user attributes store; the attributes could have been stored in a LDAP or relational database. The implementation details are hidden in the portal and the portlet only had to know about the standardized portlet interfaces. Next, we will briefly cover how we initialize a portlets connection to a CMS repository during a portlet's initialization phase.

12.2 Implementing MVC with the Deductible Status Portlet

The MVC pattern is very common in web applications. Portlet applications should follow the same design patterns. In this section we explore how to implement the view portion of MVC, as well as the controlling portlet. Although we write JDBC code in our portlets, the abstraction is clear between view and model, as we only use Deductible model beans in our view. We'll start with looking at the controller implementation: the `doView` method of the Deductible Status portlet, followed by the model which uses JDBC to create a Deductible model bean. Finally, we display the Deductible content in a portlet using a Velocity view.

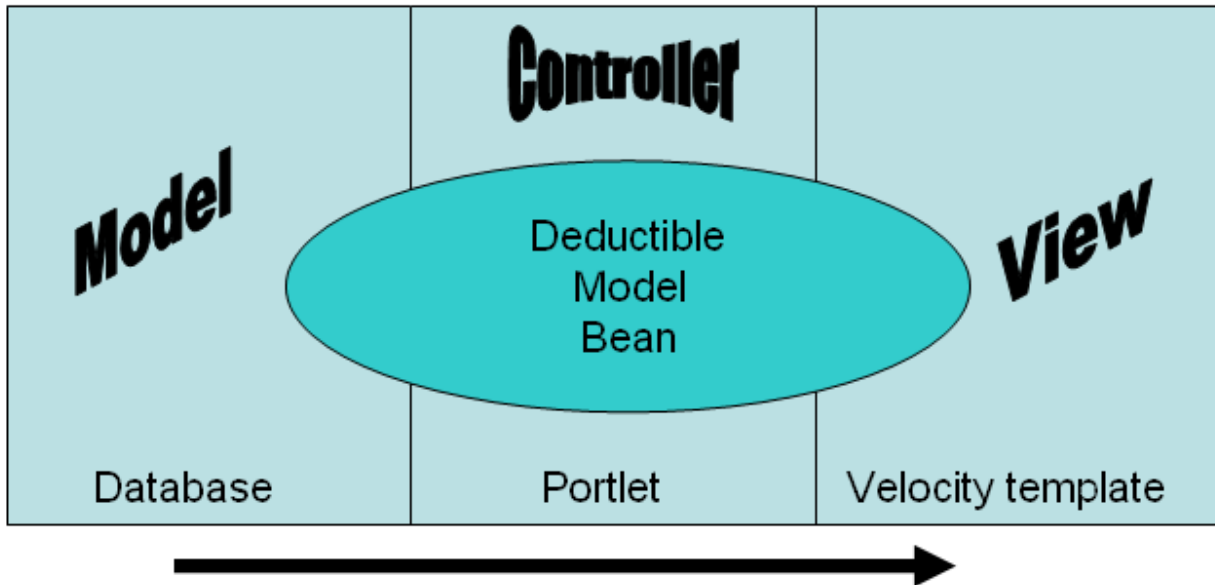


Figure 13.5 The deductible model bean originates in the database, and is rendered by the portlet into the velocity view

The doView method

On our Waxwing Benefits example's Home Page, the Deductible Status Portlet (Figure 13.6) displays information about the member's deductible premiums and accumulated deductibles over time:

Deductible Status		
	Deductible Caps	Accumulated
Lifetime Maximum	\$12,000,000.00	\$385,000.00
Individual Deductible	\$250.00	\$250.00
Individual Deductible	\$650.00	\$515.00
Individual Out-of-Pocket	\$1,500.00	\$550.00
Family Out-of-Pocket	\$3,000.00	\$1,250.00

Figure 13.6 The rendered output of the Deductible Status portlet

Let's have a look at the Deductible Status Portlet source code. The *doView* method handles looking up the deductible status:

Listing 13.9 doView method of the Deductible Status portlet

```
public void doView(RenderRequest request, RenderResponse response)
    throws PortletException, IOException
{
    Context context = super.getContext(request);
    Deductible deductible = (Deductible)
        request.getPortletSession().getAttribute("deductible");
    if (deductible == null)
    {
        deductible =
            lookupDeductible(request.getUserPrincipal().toString());
        request.getPortletSession().setAttribute("deductible", deductible);
    }
}
```

```

        context.put("deductible", deductible);
        super.doView(request, response);
    }

```

The portlet accesses the corporate benefits database to produce a deductibles report for the current user. This status report provides up-to-date deductible information on a per-user basis.

Identifying the current user

The current user is determined using the Portlet API. Specifically, the *PortletRequest* class provides a *getUserPrincipal* method which is then passed into the report generation code:

```
deductible = lookupDeductible(request.getUserPrincipal().toString());
```

The portal's user principal is also stored as a column in the deductible table of the claims database. When calculating our report, we only count deductible records that are for the current user principal.

Caching the report data

Since retrieving from a database can sometimes be an expensive operation, we demonstrate how to cache the deductible status report in the portlet session. By caching the calculated report data in the portlet session, the report only calculates once per user, when the user starts a new session. By logging out, the user terminates the portlet session. The next time the user logs on; the report result is cleared from the portlet session and thus must be regenerated.

Here is how to get an attribute out of the portlet session using the Portlet API:

```

        Deductible deductible = (Deductible)
request.getPortletSession().getAttribute("deductible");
        if (deductible == null)
        {

```

We are looking for an attribute named *deductible*. Since we put the attribute in the session, we know it is of the type *Deductible*. When a user hits this page for the first time after logging on, the attribute is not found and we generate the report.

Listing 13.10 lookup a deductible record for the given user (principal)

```

public Deductible lookupDeductible(String principal)
{
    Connection con = null;
    try
    {
        con = cms.getConnection();

        String select =
            "select LIFETIME_MAX, IND_DEDUCT, FAM_DEDUCT, IND_OOP, FAM_OOP, "+
            " ACC_LIFETIME, ACC_IND_DEDUCT, ACC_FAM_DEDUCT, ACC_IND_OOP, " +
            "ACC_FAM_OOP" +
            " from WWDEDUCTIBLE where USER_ID = ?";

        PreparedStatement stm = con.prepareStatement(select);
        stm.setString(1, principal);

```

```

        ResultSet rs = stm.executeQuery();           | #2
        rs.next();
        double lifetimeMax = rs.getDouble(1);
        double individualDeductible = rs.getDouble(2);
        double familyDeductible = rs.getDouble(3);
        double individualOutOfPocket = rs.getDouble(4);
        double familyOutOfPocket = rs.getDouble(5);
        double accLifetimeMax = rs.getDouble(6);
        double accIndividualDeductible = rs.getDouble(7);
        double accFamilyDeductible = rs.getDouble(8);
        double accIndividualOutOfPocket = rs.getDouble(9);
        double accFamilyOutOfPocket = rs.getDouble(10);

        rs.close();
        stm.close();

        return new Deductible(lifetimeMax,           | #3
                               individualDeductible,
                               familyDeductible,
                               individualOutOfPocket,
                               familyOutOfPocket,
                               accLifetimeMax,
                               accIndividualDeductible,
                               accFamilyDeductible,
                               accIndividualOutOfPocket,
                               accFamilyOutOfPocket);
    }
    catch (SQLException e)
    {
        System.err.println("Exception retrieving deductible status: " + e);
    }
    finally
    {
        releaseConnection(con);
    }

    return new Deductible(0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
}
(annotation) <#1 create the SQL statement>
(annotation) <#2 execute the SQL statement>
(annotation) <#1 create a new Deductible object>

```

In listing 13.10, the complete `lookupDeductible` method is shown. Here we get a JDBC connection, retrieve the deductible record from the database using a JDBC prepared statement, and return a new `Deductible` bean.

Back to the controller code in the portlet's `doView` method: the bean is passed on to the Velocity view by putting the *Deductible* object in the Velocity context:

```

deductible = lookupDeductible(request.getUserPrincipal().toString());
... context.put("deductible", deductible);

```

Rendering the report

From there, our template can access the *Deductible* object to render the report output:

Listing 13.11 Velocity template to render a deductible record

```
<table border="1" cellspacing="1" cellpadding="3">
  <tr>
    <th></th>
    <th class='portlet-form-label'>Deductible Caps</td>
    <th class='portlet-form-label'>Accumulated</td>
  </tr>
  <tr>
    <td class='portlet-section-alternate'>Lifetime Maximum</td>
    <td class='portlet-section-alternate'>${deductible.LifetimeMax}</td>
    <td class='portlet-section-alternate'>${deductible.AccLifetimeMax}</td>
  </tr>
  <tr>
    <td class='portlet-section-alternate'>Individual Deductible</td>
    <td class='portlet-section-alternate'>
      ${deductible.IndividualDeductible}
    </td>
    <td class='portlet-section-alternate'>
      ${deductible.AccIndividualDeductible}
    </td>
  </tr>
  <tr>
    <td class='portlet-section-alternate'>Individual Deductible</td>
    <td class='portlet-section-alternate'>
      ${deductible.FamilyDeductible}
    </td>
    <td class='portlet-section-alternate'>
      ${deductible.AccFamilyDeductible}</td>
  </tr>
  <tr>
    <td class='portlet-section-alternate'>Individual Out-of-Pocket</td>
    <td class='portlet-section-alternate'>
      ${deductible.IndividualOutOfPocket}
    </td>
    <td class='portlet-section-alternate'>
      ${deductible.AccIndividualOutOfPocket}
    </td>
  </tr>
  <tr>
    <td class='portlet-section-alternate'>Family Out-of-Pocket</td>
    <td class='portlet-section-alternate'>
      ${deductible.FamilyOutOfPocket}</td>
    <td class='portlet-section-alternate'>
      ${deductible.AccFamilyOutOfPocket}</td>
  </tr>
</table>
```

Velocity templates are perhaps the easiest to understand of all Java-based templates. (For a quick review of Velocity, see chapter 11 on the Velocity Bridge.) The deductible model bean is put into the Velocity context, where it can then be pulled out by the template using the `$deductible` variable. For getters on beans, Velocity allows for a short-hand syntax where the “get” prefix is not required and you can instead simply enter `$deductible.LifetimeMax` for the getter method

getLifetimeMax(). This is how we pull the dynamic content from the model into the view. Let's next have a closer look at the Deductible model bean.

Storing the Deductible inner class

The Deductible bean is an inner class stored in our portlet:

Listing 13.12 Deductible bean

```
public class Deductible
{
    private double lifetimeMax;
    private double individualDeductible;
    private double familyDeductible;
    private double individualOutOfPocket;
    private double familyOutOfPocket;
    private double accLifetimeMax;
    private double accIndividualDeductible;
    private double accFamilyDeductible;
    private double accIndividualOutOfPocket;
    private double accFamilyOutOfPocket;

    public Deductible(double lifetimeMax,
                      double individualDeductible,
                      double familyDeductible,
                      double individualOutOfPocket,
                      double familyOutOfPocket,
                      double accLifetimeMax,
                      double accIndividualDeductible,
                      double accFamilyDeductible,
                      double accIndividualOutOfPocket,
                      double accFamilyOutOfPocket)
    {
        this.lifetimeMax = lifetimeMax;
        this.individualDeductible = individualDeductible;
        this.familyDeductible = familyDeductible;
        this.individualOutOfPocket = individualOutOfPocket;
        this.familyOutOfPocket = familyOutOfPocket;
        this.accLifetimeMax = accLifetimeMax;
        this.accIndividualDeductible = accIndividualDeductible;
        this.accFamilyDeductible = accFamilyDeductible;
        this.accIndividualOutOfPocket = accIndividualOutOfPocket;
        this.accFamilyOutOfPocket = accFamilyOutOfPocket;
    }

    private final NumberFormat formatter =
        new DecimalFormat("$###,###.00");

    /**
     * @return Returns the familyDeductible.
     */
    public String getFamilyDeductible()
    {
        return formatter.format(familyDeductible);
    }
}
```

```

}
/**
 * @return Returns the familyOutOfPocket.
 */
public String getFamilyOutOfPocket()
{
    return formatter.format(familyOutOfPocket);
}
/**
 * @return Returns the individualDeductible.
 */
public String getIndividualDeductible()
{
    return formatter.format(individualDeductible);
}
/**
 * @return Returns the individualOutOfPocket.
 */
public String getIndividualOutOfPocket()
{
    return formatter.format(individualOutOfPocket);
}
/**
 * @return Returns the lifetimeMax.
 */
public String getLifetimeMax()
{
    return formatter.format(lifetimeMax);
}
/**
 * @return Returns the accFamilyDeductible.
 */
public String getAccFamilyDeductible()
{
    return formatter.format(accFamilyDeductible);
}
/**
 * @return Returns the accFamilyOutOfPocket.
 */
public String getAccFamilyOutOfPocket()
{
    return formatter.format(accFamilyOutOfPocket);
}
/**
 * @return Returns the accIndividualDeductible.
 */
public String getAccIndividualDeductible()
{
    return formatter.format(accIndividualDeductible);
}
/**
 * @return Returns the accIndividualOutOfPocket.
 */
public String getAccIndividualOutOfPocket()
{

```

```

        return formatter.format(accIndividualOutOfPocket);
    }
    /**
     * @return Returns the accLifetimeMax.
     */
    public String getAccLifetimeMax()
    {
        return formatter.format(accLifetimeMax);
    }
}
}

```

As you can see, following the MVC pattern leads to other good practices such as abstraction, encapsulation and separation of concerns when developing portlets. Although we used JDBC directly in our portlet, you could easily swap out the one method in a real world application and replace it with a popular object-relational mapping framework. But the *lookupDeductible* method should still return a model bean, which would then require no changes to your view. Likewise, you could replace the Velocity templates with JSP, but the model code would not care, since it only works with a Deductible bean and not the details of the template.

Next in section 13.5, we will further examine developing portlets with databases by exploring the Jetspeed Database browser portlet.

Browsing databases: the claims browser and claims detail portlets

The Claims Browser and Claims Detail Portlets are two portlets that work together to provide a combined user interface experience. The browser displays a list of all claims for the current user. When the user clicks on one of these rows, the browser communicates with the Claims Detail portlet, telling it which row was selected. The detail portlet then displays the detailed information for the selected claim in its portlet window.

When creating your portal page, make sure to put the Claims Detail portlet next to the Claims Browser portlet, as is done in Figure 13.7.

The screenshot shows two portlets side-by-side. The left portlet, titled 'Database Browser', contains a table with the following data:

Service	Group	Status
Diagnostic Radiology	MedPlan 1000	PROCESSED
Emergency Op Room Charges	MedPlan 1000	PROCESSED
Other Medical Services	CompPlan 2000	** OPEN **
Prescriptions	MedPlan 1000	PROCESSED
Hospital Misc	MedPlan 1000	PROCESSED
Lab	CompPlan 2000	** HOLD **
Surgery	CompPlan 2000	PROCESSED
Medical Visit	MedPlan 1000	PROCESSED
Room and Board	MedPlan 1000	** OPEN **

Below the table is a pagination bar showing '0 of 9' and buttons for 'Go' and 'Refresh'.

The right portlet, titled 'Claims Detail', shows the following information:

- Member Name: Chip Sparrow
- Member Company: Eruditorum
- Benefits Group: 1700-3
- Benefits Account: 176-946
- Service: Diagnostic Radiology
- Group: MedPlan 1000
- Paid by Plan: \$111.00
- Paid by Member: \$88.00
- Status: PROCESSED
- Comments: I need to get a more detailed lab report from the provider
- Buttons: Make Request

Figure 13.7 The Claims Browser and Claims Detail portlets side by side

The Claims Detail portlet is somewhat dependent on the Claims Browser portlet. Since the two portlets are loosely coupled, the Claims Detail portlet could just as easily get a primary key from another portlet to look up the current claim record. The Claims Browser portlet, though, is not dependent on the Claims Detail portlet. The browser can be used by itself.

In the following section, we will discuss how the Waxwing example allows users to browse claims by extending the Jetspeed database browser portlet.

Customizing The Jetspeed database browser portlet

The database browser portlet comes with the Jetspeed-2 distribution. It is based on the Velocity Bridge and renders all content with the Velocity template engine. The Database Browser portlet is a standardized portlet that will port to any portal (portlet container). It does require that jars from the Apache Portals Bridges project are included.

The Waxwing Claims Browser portlet is an example of extending an existing portlet and customizing it to read over our claims table in the Waxwing corporate database.. Let's take a look at the amount of Java code that we actually wrote to make this work:

Listing 13.13 ClaimsBrowserPortlet class

```
public class ClaimsBrowserPortlet extends DatabaseBrowserPortlet
{
    public ClaimsBrowserPortlet()
    {
    }

    protected void readSqlParameters(RenderRequest request)
    {
        List sqlParameters = new LinkedList();
        sqlParameters.add(request.getUserPrincipal().toString());
        setSQLParameters(sqlParameters);
    }

    public void processAction(ActionRequest actionRequest,
                             ActionResponse actionResponse)
        throws PortletException, IOException
    {
        String claim = actionRequest.getParameter("claim");

        // publish change to ClaimsDetail Portlet
        if (claim != null)
            PortletMessaging.publish(actionRequest,
                                     "claims", "selected", claim);

        super.processAction(actionRequest, actionResponse);
    }
}
```

As you can see, there isn't a whole lot there. This is usually the sign of a useful base portlet class. Before we explain the code, let's step back and look at how to configure a Database Portlet.

Configuring a database portlet

In order to connect to and query a relational table, each instance of a database portlet must be configured. Minimally, you will need an SQL statement and JDBC connection parameters. The *portlet.xml* contains the configuration:

Listing 13.14 portlet.xml deployment descriptor for the ClaimsBrowserPortlet

```
<portlet>
```

```

<description>Claims Browser</description>
<portlet-name>ClaimsBrowser</portlet-name>
<display-name>Claims Browser</display-name>
<portletclass>
    com.waxwingbenefits.portal.portlets.ClaimsBrowserPortlet</portlet-class>
<init-param>
    <name>ViewPage</name> | #1
<value>/WEB-INF/views/database-view.vm</value>
</init-param>
<init-param>
    <name>EditPage</name> | #2
    <value>/WEB-INF/views/database-edit.vm</value>
</init-param>
<init-param>
    <name>HelpPage</name> | #3
    <value>/WEB-INF/views/database-help.vm</value>
</init-param>
<expiration-cache>0</expiration-cache>
<supports>
    <mime-type>text/html</mime-type>
    <portlet-mode>VIEW</portlet-mode>
    <portlet-mode>EDIT</portlet-mode>
</supports>
<supported-locale>en</supported-locale>
<supported-locale>de</supported-locale>
<resource-bundle>
    org.apache.portals.gems.browser.resources.Browser</resource-bundle>

```

...
(annotation) <#1 the ViewPage init parameter>
(annotation) <#2 the EditPage init parameter>
(annotation) <#3 the HelpPage init parameter>

There are three initialization parameters required:

1. View Page – the View Mode template
2. Edit Page – the Edit Mode template
3. Help Page – the Help Mode template

Each one of these init parameters defines the template that is used to display the corresponding portlet mode. All three of these templates have been borrowed from the Jetspeed portlets. We will be discussing the view and edit mode templates in more detail throughout this section. First, let's look at the available connection configuration parameters.

Configuring database connection parameters

The database connection parameters are configured directly in the portlet preferences from edit mode. Figure 13.8 shows how to configure a JDBC connection:

Database Browser

Database Browser Preferences

Data Source Type

☐ JNDI Data Source
☒ DBCP Data Source
☐ SSO Credential Store

Please Select a Data Source Type

JNDI Settings

JNDI Data Source

JDBC/DBCP Settings

JDBC Driver
JDBC Connection
JDBC Username
JDBC Password

J2 SSO Settings

JDBC Driver
JDBC Connection
SSO Site

General Settings

Window Size

SQL

Figure 13.8 Configuring database connection parameters in the preferences editor

From edit mode, you will see the database preferences editor. This editor is an updatable view over the portlet preferences defined in your portlet.xml descriptor:

Listing 13.15 Portlet Preferences for Database Browser Portlets

```
<portlet-preferences>
  <preference>
```

```

    <name>DatasourceType</name>
    <value>dbcp</value>
</preference>
<preference>
    <name>JdbcDriver</name>
    <value>com.mysql.jdbc.Driver</value>
</preference>
<preference>
    <name>JdbcConnection</name>
    <value>jdbc:mysql://j2-server/waxwing</value>
</preference>
<preference>
    <name>JdbcUsername</name>
    <value>john</value>
</preference>
<preference>
    <name>JdbcPassword</name>
    <value>secret </value>
</preference>

```

Preference	Value (example)	Description
DatasourceType	1. dbcp 2. jndi 3. sso	1. a managed pool of JDBC connection 2. a JNDI database connection managed by the app server 3. SSO – Jetspeed SSO (Single Signon) – only applicable with Jetspeed-2 portal
JdbcDriver	com.mysql.jdbc.Driver	The class name of the JDBC driver
JdbcConnection	:mysql://j2-server/waxwing	The database-specific JDBC connection string
JdbcUsername	John	The username on the database
JdbcPassword	Secret	The password on the database

Table 13.3 JDBC connection parameters

Although putting the JDBC connection properties is useful for this book example, it is usually best to use an application server managed-JNDI connection.

Using a JNDI and Jetspeed-2 SSO connection

The Java Naming and Directory Interface (JNDI) is part of the Java platform, providing applications based on Java technology with a unified interface to multiple naming and directory services. You can build powerful and portable directory-enabled applications using this industry standard. Most application servers, including Tomcat, support JNDI lookups for application server resources such as data sources and database connection pools. Once the JNDI resource is configured in the application server, the database browser portlet simply has to reference it by name to get connections to the application server-managed database.

Listing 13.16 JNDI connection parameters in portlet preferences

```

<portlet-preferences>
  <preference>
    <name>DatasourceType</name>
    <value>jndi</value>
  </preference>
  <preference>
    <name>MySource</name>
    <value></value>
  </preference>
</portlet-preferences>

```


Another useful option is using the Jetspeed-2 SSO feature, although it is only available when the portlet is deployed to the Jetspeed-2 portal. Jetspeed-2 has a Single Sign-on (SSO) repository. Besides being used to store common database credentials, the SSO repository can also store credentials used to logon to other applications that the portal requires access to.

WindowSize preference

The *WindowSize* preference configures how many rows are displayed in the portlet window.

```
<preference>
  <name>WindowSize</name>
  <value>10</value>
</preference>
```

SQL preference

The *sql* preference configures the SQL select statement string.

```
<preference>
  <name>sql</name>
  <value>select CLAIM_ID, SERVICE as "Service", GROUPE as "Group", STATUS as
"Status" from WWCLAIMS where USER_ID = ?</value>
</preference>
```

We select from the WWCLAIMS table to retrieve all claims for the current user. Note that the question mark (?) is a placeholder used by JDBC when we prepare the SQL statement and bind a user name to the prepared statement. We select four columns, three of which will be visible in the portlet view. We also make use of giving the columns alternate names using the AS clause of the SELECT statement. Here we simply format the column name to be capitalized instead of all uppercase:

```
SERVICE as "Service"
```

In your customized database browser portlets, you probably do not want end users editing the SQL preference for obvious reasons. This concludes our discussion of connection parameters. Next, let's start looking at what you can do programmatically with the database browser portlet.

Using SQL query parameters

Having a static SQL string defined in your portlet preferences is very useful. But sometimes you will need to dynamically bind runtime criteria to an SQL query. If you need to query based on runtime parameters, you will need to override the *readSqlParameters* method. For example, here is how to filter the SQL query::

```
protected void readSqlParameters(RenderRequest request)
{
    List sqlParameters = new LinkedList();
    sqlParameters.add(request.getUserPrincipal().toString());
    setSQLParameters(sqlParameters);
}
```

The *readSqlParameters* method of the *DatabaseBrowserPortlet* can be overridden to provide a list of SQL objects that are passed in the browser's internal prepared statement. These objects must be in an ordered list, provided in the order necessary for substituting into the prepared

statement. In the example above, we get the user principal from the Portlet API, and provide it to the browser via the callback message. Thus, if the current logged on user is named *robin*, the resulting dynamic SQL statement becomes:

```
select CLAIM_ID, SERVICE as "Service", GROUPE as "Group", STATUS as "Status" from
WWCLAIMS where USER_ID = 'robin'
```

The query executes and generates a model that can be traversed by a Velocity portlet.

The Database Browser Velocity Template

We configure the Waxwing database browser's Velocity template in the *portlet.xml* configuration:

```
<init-param>
  <name>ViewPage</name>
  <value>/WEB-INF/views/database-view.vm</value>
</init-param>
```

First, look at the section of the template that creates the headings. The *DatabaseBrowserPortlet* provides a velocity variable named \$title, holding an array of column titles for each column in the result set. All of the column titles are generated from the result set metadata.

Listing 13.17 Rendering the Column Headers

```
<table cellpadding=0 cellspacing=1 border=0 width="100%">
  <tbody>
    <tr>
      #foreach ($column in $title)      | #1
        #if ($velocityCount == 1)
        #else
        #set ($columnLink = $renderResponse.createRenderURL())      | #2
        $columnLink.setParameter("js_dbcolumn",$column)
        <td align=CENTER class="jetdbHead" width="43"
          nowrap onClick="window.location.href='$columnLink'">
          <div align="center">$column</div>
        </td>
        #end
      #end
      #if ($rowLinks)
        #if ($rowLinks.size() > 0)
          <td align=CENTER width="43" class="jetdbHead"></td>
        #end
      #end
    </tr>
  </tbody>
</table>
```

(annotation) <#1 create column headers from velocity \$title variable>
(annotation) <#2 create a render URL link for each column header>

Note that we create links in the column headers. These links are created using the *renderResponse* object (provided by the *DatabaseBrowserPortlet*'s base class, *GenericVelocityPortlet*). The *GenericVelocityPortlet* provides the Velocity variables *\$renderRequest* and *\$renderResponse*, just like the Portlet API equivalents provided by the Portlet API tag library for JSP templates.

```
#set ($columnLink = $renderResponse.createRenderURL())
$columnLink.setParameter("js_dbcolumn",$column)
```

The render parameter *js_dbcolumn* is passed to the *DatabaseBrowserPortlet*. When it sees this render parameter, the browser knows that a request is being made to sort its content on the given column. Each row from the table is rendered inside of a doubly-nested for loop. The outer loop walks through the result set's rows using the velocity variable named *\$table*.

Listing 13.18 Iterating over the rows in the table

```
#foreach ( $row in $table )
<tr>
  #if ($velocityCount % 2 == 0)
    #set ($rowstyle = "jetdbEven")
  #else
    #set ($rowstyle = "jetdbOdd")
  #end
```

The inner for loop walks through each column for the current row. This row/column model can be easily traversed using a generic algorithm that is easily customizable. Each iteration of this loop generates a column for the current row:

Listing 13.19 Rendering the database browser columns for the current row

```
#foreach ( $entry in $row )      |#1
  ## specialized for this CLAIMS query ONLY, skip over column 1
  #if ($velocityCount == 1)
    #set ($rowid = $entry)
  #elseif ($velocityCount == 2)
    #set ($claimLink = $renderResponse.createActionURL())      |#2
    $claimLink.setParameter("claim",$rowid.toString())
    <td nowrap class="$rowstyle" width="23">
      <div class="$rowstyle" align="center">
        <a href='$claimLink'>$entry</a></div>      |#3
      </td>
    #else
      <td nowrap class="$rowstyle" width="23">
        <div class="$rowstyle" align="center">$entry</div>
      </td>
    #end
  #end
#end
(annotation) <#1 for each row, iterator over the columns>
(annotation) <#2 create an action URL for the Service column>
(annotation) <#3 display the content of the column>
```

The end result is a database browser over the CLAIMS table as shown in Figure 13.9:

Database Browser		
Service	Group	Status
Diagnostic Radiology	MedPlan 1000	PROCESSED
Emergency Op Room Charges	MedPlan 1000	PROCESSED
Other Medical Services	CompPlan 2000	** OPEN **
Prescriptions	MedPlan 1000	PROCESSED
Hospital Misc	MedPlan 1000	PROCESSED
Lab	CompPlan 2000	** HOLD **
Surgery	CompPlan 2000	PROCESSED
Medical Visit	MedPlan 1000	PROCESSED
Room and Board	MedPlan 1000	** OPEN **
0 of 9 Go Refresh		

Figure 13.9 The Claims Browser portlet in view mode

In figure 13.9, we show only the default navigational controls:

1. Go to Record
2. Refresh

Next, let's add some additional navigational controls to your database browser.

Adding navigation controls

The database browser supports the following controls:

- Refresh
- Go to Beginning of Result Set (>>)
- Go to End of Result Set (<<)
- Previous Page (<)
- Next Page (>)
- Go to Record #

<<	<	8	of 16	Go	>	>>	Refresh
----	---	---	-------	----	---	----	---------

Figure 13.10 Database Browser controls

Here is the Velocity template snippet required to do add the controls:

Listing 13.20 Rendering the Database Browser Controls

```
<table width="200" border="0" cellspacing="0" cellpadding="0" align="center">
  <tr>
    <td align="center">
      <div align="center">
        <form action="$renderResponse.createActionURL()" method="post"> | #1
          <input type='hidden' name='db.browser.action' value='first' />
          <input class="jetdbButton" type="submit" value="<<" />
          <input type="hidden" name="start" value="0" />
        </form>
      </div>
    </td>
  </tr>
</table>
```

```

</td>
<td valign="middle" height="30">
  <div align="center">
    <form action="$renderResponse.createActionURL()" method="post"> | #2
      <input type='hidden' name='db.browser.action' value='prev' />
      <input class="jetdbButton" type="submit" value="<">
      <input type="hidden" name="start" value="$prev">
    </form>
  </div>
</td>
#end
#if ($tableSize > 0)
  <form action="$renderResponse.createActionURL()" method="post"> | #3
    <td valign="middle" height="30">
      <div align="center">
        <input type='hidden' name='db.browser.action' value='change' />
        <input type="input" name='start' size='5' value="$start">
      </div>
    </td>
    <td valign="middle" height="30">
      <div align="center">
        <input type="input" readonly size='10' value="of $tableSize">
      </div>
    </td>
    <td valign="middle" height="30">
      <div align="center">
        <input class="jetdbButton" type="submit" value="Go">
      </div>
    </td>

  </form>
#end
#if ($next)
  <td valign="middle">
    <div align="center">
      <form action="$renderResponse.createActionURL()" method="post"> | #4
        <input type='hidden' name='db.browser.action' value='next' />
        <input class="jetdbButton" type="submit" value=">">
        <input type="hidden" name="start" value="$next">
      </form>
    </div>
  </td>
  <td valign="middle" height="30">
    <div align="center">
      <form action="$renderResponse.createActionURL()" method="post"> | #5
        <input type='hidden' name='db.browser.action' value='last' />
        <input class="jetdbButton" type="submit" value=">>">
        <input type="hidden" name="start" value="$tableSize">
      </form>
    </div>
  </td>
#end
#if ($tableSize > 0)
  <td valign="middle">
    <div align="center">

```

```

        <form action="$renderResponse.createActionURL()" method="post"> |#6
            <input type='hidden' name='db.browser.action' value='refresh' />
            <input class="jetddbButton" type="submit" name="eventSubmit_doRefresh"
value="$MESSAGES.getString('dbrefresh')" />
        </form>
    </div>
</td>
#end
</tr>
</table>
(Annotation) <#1 go to First record>
(Annotation) <#2 go to Prev record>
(Annotation) <#3 go to Nth record>
(Annotation) <#4 go to Next record>
(Annotation) <#5 go to Last record>
(Annotation) <#6 Refresh>

```

#1 An action URL is created for each database navigation action. It takes no additional code in your portlet to handle these built in actions. However if you override the *processAction* method of your portlet, make sure to pass control to the *DatabaseBrowserPortlet* after handling your action processing:

```
super.processAction(actionRequest, actionResponse);
```

Browsing other data

The base class of the *DatabaseBrowserPortlet* is the *BrowserPortlet* class. The *BrowserPortlet* uses the same concepts as the *DatabaseBrowserPortlet*, however it is not limited to relational databases. By overriding the *getRows* method, derived implementations can easily browse over other kinds of data.

For example, the User popups as in Figure 13.11 in the Jetspeed-2 security management portlets use a *BrowserPortlet* combined with a *UserManager* API:



Figure 13.11 The User Chooser portlet is a Browser portlet

And here is the entire source to the UserChooserPortlet:

Listing 13.21 The UserChooserPortlet implementation

```

public class UserChooserPortlet extends BrowserPortlet
{
    private UserManager userManager;

    public void init(PortletConfig config)

```

```

throws PortletException
{
    super.init(config);
    userManager = (UserManager)    |#1
        getPortletContext().getAttribute(
            SecurityResources.CPS_USER_MANAGER_COMPONENT);
    if (null == userManager)
    {
        throw new PortletException(
            "Failed to find the User Manager on portlet initialization");
    }
}

public void getRows(RenderRequest request, String sql, int windowSize)
throws Exception
{
    List resultSetTitleList = new ArrayList();
    List resultSetTypeList = new ArrayList();
    try
    {
        Iterator users = userManager getUsers("");    |#2

        resultSetTypeList.add(String.valueOf(Types.VARCHAR));
        resultSetTitleList.add("User");

        List list = new ArrayList();
        while (users.hasNext())
        {
            User user = (User)users.next();
            Principal principal = getPrincipal(user.getSubject(),
                UserPrincipal.class);    |#3
            list.add(principal.getName());
        }
        BrowserIterator iterator = new DatabaseBrowserIterator(
            list, resultSetTitleList, resultSetTypeList,
            windowSize);
        setBrowserIterator(request, iterator);
        iterator.sort("User");    |#4
    }
    catch (Exception e)
    {
        //log.error("Exception in CMSBrowserAction.getRows: ", e);
        e.printStackTrace();
        throw e;
    }
}

```

(Annotation) <#1 acquire user manager service on portlet init>

(Annotation) <#2 getUsers API retrieve all users in portal>

(Annotation) <#3 build the list of users as JAAS principals>

(Annotation) <#4 sort the final user list>

As you can see in figure 13.11, we have quickly created a portlet for browsing over the users in the portal. Now that we have seen how to program a database browser portlet, let's look at how you can make the database browser portlet communicate with another portlet: the Claims Detail portlet.

Using inter-portlet communication: the claims browser and claims detail portlets

The Claims Browser portlet is a great way for an employee to review their claims. However, what if the employee needed to see more claims details? Or if the employee needed to update or interact with the claims record? In this section, we will show you how to solve these problems. By combining two portlets on one portal page, we give the employee the ability to both browse their claims, and update their claims detail status.

This is achieved with a basic inter-portlet communication solution based on action URLs and some helper classes from Portals Bridges. Note that this entire solution is portable to any Portlet API-compliant portal server.

First, let's look at how we send the message. This occurs when someone clicks on the service column in the Claims browser:

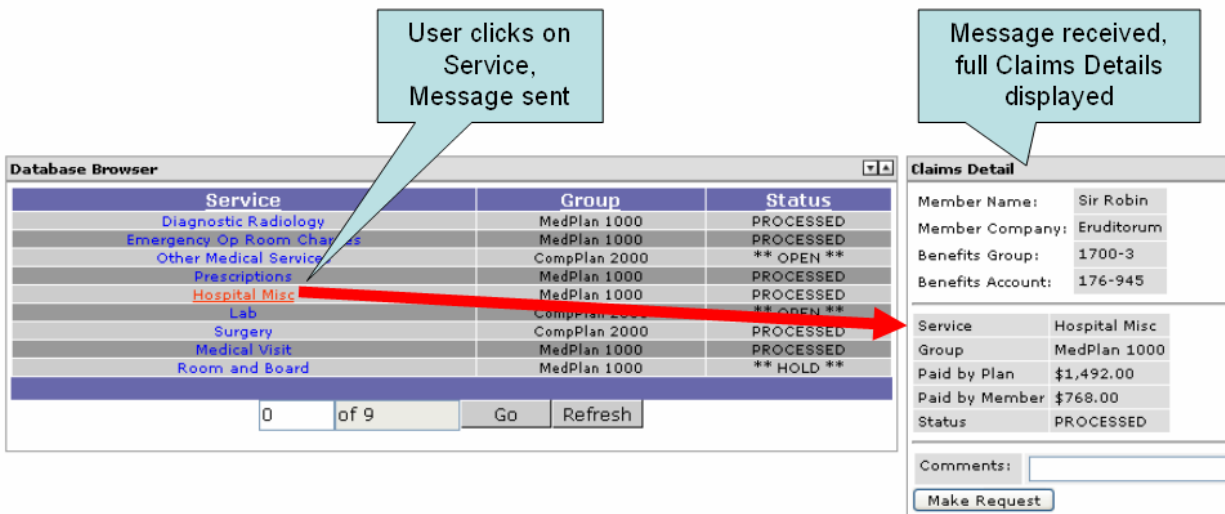


Figure 13.12 Inter-portlet communication: the claims browser sends a message to the claims detail

The service name is sent in a message to the Claims Detail portlet during the Claims Browser's action phase. The Claims Detail portlet receives the message, and knows to display the details for the selected claim.

Version 1.0 of the Portlet API does not address inter-portlet communication. For this book, we have made use of a fairly abstract implementation based on using the portlet application's session state. Since application scoped session state can be easily shared amongst portlets, we have provided a simple abstraction over the session API. The remainder of this section will demonstrate how we implemented our inter-portlet communication.

Sending a message

For the Service column of the database browser, we always create a URL around the column label "Service". The link is created using the *actionResponse* object (provided by the *DatabaseBrowserPortlet*'s base class, *GenericVelocityPortlet*). This URL is an action URL. The action URL has a parameter named *claim*.

```
#elseif ($velocityCount == 2)
#set ($claimLink = $renderResponse.createActionURL())
$claimLink.setParameter("claim", $rowid.toString())
```



```
<td nowrap class="$rowstyle" width="23">
  <div class="$rowstyle" align="center">
    <a href='$claimLink'$entry</a></div>
  </td>
```

The unique claim ID comes from the first column of the query:

```
#if ($velocityCount == 1)
  #set ($rowid = $entry)
```

The claim ID then passed as a parameter with the action URL:

```
$claimLink.setParameter("claim", $rowid.toString())
```

This parameter is picked up in our *ClaimBrowserPortlet*'s *processAction* method:

```
String claim = actionRequest.getParameter("claim");

// publish change to ClaimsDetail Portlet
if (claim != null)
  PortletMessaging.publish(actionRequest,
    "claims", "selected", claim);
```

We then make use of a helper class provided with Portals Bridges. The *PortletMessaging* class facilitates the passing of messaging between portlets within the same portlet application (Note that the default implementation will not support cross web-context messaging). The other message parameters include:

- The second parameter, “claims”, is a general message topic.
- The third parameter, “selected”, is the message name.
- The fourth parameter is the value of the message.

The next version of the Portlet API will support portlet messaging. The code that we have shown you in this section works as an abstraction towards the anticipated API change, minimizing the changes required to our application when the API is public.

Next, we will see how the *ClaimsDetailPortlet* picks up this message.

Receiving the message

The *ClaimsDetailPortlet* displays the full details of a Claims record. It also has the ability to update the comments field on the Claims record.

The Claims Detail portlet expects a primary key to be made available by another portlet. Looking at the *doView* method in Listing 13.22, we see that the portlet first retrieves user attributes(1). We discussed User Attributes in section 13.3. .

Listing 13.22 The doView method of the ClaimsDetailPortlet

```
public void doView(RenderRequest request, RenderResponse response)
  throws PortletException, IOException
{
  Map userInfo =      |#1
    (Map) request.getAttribute(PortletRequest.USER_INFO);

  Context context = super.getContext(request);
  String firstName = (String)userInfo.get("user.name.given");
```

```

context.put("firstName",      |#2
            userInfo.get("user.name.given"));
context.put("lastName", userInfo.get("user.name.family"));
context.put("company", userInfo.get("user.employer"));
context.put("group", userInfo.get("benefits.group"));
context.put("account", userInfo.get("benefits.account"));

String claimId =      |#3
    (String)PortletMessaging.consume(request, "claims", "selected");
Claim claim =      |#4
    (Claim)request.getPortletSession().getAttribute("claim");
if (claimId != null)
{
    claim = lookupClaimStatus(claimId);      |#5
    request.getPortletSession().setAttribute("claim", claim);
}

context.put("claim", claim);      |#6
super.doView(request, response);
}

```

(Annotation) <#1 get USER_INFO from Portlet API>

(Annotation) <#2 get required user attributes, then put them in Velocity context>

(Annotation) <#3 receive a message from Claims browser upon selection>

(Annotation) <#4 get the claim by ID from the portlet session>

(Annotation) <#5 if a new claim is selected, retrieve from database>

(Annotation) <#6 put the claim into the view for rendering in velocity template

Getting the claim selected message

Next, the portlet receives a message from the database browser portlet, informing the detail portlet that a row was clicked on in the browser:

```

String claimId = (String)PortletMessaging.consume(request, "claims",
"selected");
Claim claim = (Claim)request.getPortletSession().getAttribute("claim");
if (claimId != null)
{
    claim = lookupClaimStatus(claimId);
    request.getPortletSession().setAttribute("claim", claim);
}

```

Note the message is consumed, meaning it is removed from the message queue since the lifetime of a “claim selected” message spans only one portlet render cycle.

Looking up the correct claim status record

When a claim message is received, the portlet looks up a Claim Status record, using the primary key, *claimId*, passed in the message. The claim record is put into the Velocity context:

```
context.put("claim", claim);
```

The claim detail is rendered in the Velocity template:

Listing 13.23 The velocity template for the claims detail

```
<table border="1" cellspacing="1" cellpadding="3">
  <tr>
    <td class='portlet-section-alternate'>Service</td>
    <td class='portlet-section-alternate'>${!claim.Service}</td>
  </tr>
  <tr>
    <td class='portlet-section-alternate'>Group</td>
    <td class='portlet-section-alternate'>${!claim.Group}</td>
  </tr>
  <tr>
    <td class='portlet-section-alternate'>Paid by Plan</td>
    <td class='portlet-section-alternate'>${!claim.PlanPaid}</td>
  </tr>
  <tr>
    <td class='portlet-section-alternate'>Paid by Member</td>
    <td class='portlet-section-alternate'>
      ${!claim.MemberPaid}
    </td>
  </tr>
  <tr>
    <td class='portlet-section-alternate'>Status</td>
    <td class='portlet-section-alternate'>${!claim.Status}</td>
  </tr>
</table>
```

The final result is the rendered Claim Detail portlet (Figure 13.13):

Claims Detail	
Member Name:	Chip Sparrow
Member Company:	Eruditorum
Benefits Group:	1700-3
Benefits Account:	176-946
<hr/>	
Service	Diagnostic Radiology
Group	MedPlan 1000
Paid by Plan	\$111.00
Paid by Member	\$88.00
Status	PROCESSED
<hr/>	
Comments:	<input type="text" value="I need to get a more detailed lab report from the provider"/>
<input type="button" value="Make Request"/>	

Figure 13.13 The Claims Detail portlet in view mode

This concludes our coverage of inter-portlet communication. Next, let's look at how we can use the Claims Detail portlet to update a database record.

Updating the Comments field

The portlet also provides an example of updating the Comments field. An action URL is created on the HTML form's action element. The action URL directs the posted data back to our portlets *processAction* method.

Listing 13.24 Creating an action URL to post back to the portlet

```
<form name='documentForm' action="$renderResponse.createActionURL()" method="post">
<table>
  <tr colspan="2" align="right">
    <td nowrap class="portlet-section-alternate" align="right">Comments:&nbsp;  </td>
    <td class="portlet-section-body" align="left">
      <input type="text" name="comments" size="80" value="$!claim.Comments"
class="portlet-form-field-label">
    </td>
  </tr>
</table>
#if ($claim)
<input name='claimId' type='hidden' value='$!claim.ClaimId' />
<input name='save' type="submit" value="Make Request" class="portlet-form-button" />
#end
</form>
```

The *processAction* method looks for a hidden *claimId* parameter.

Listing 13.25 The processAction method for the ClaimsDetail portlet

```
public void processAction(ActionRequest actionRequest,
    ActionResponse actionResponse) throws PortletException, IOException
{
    String claimId = actionRequest.getParameter("claimId");
    String comments = actionRequest.getParameter("comments");
    if(claimId != null)
    {
        Claim claim = updateClaimComment(claimId, comments);
        actionRequest.getPortletSession().setAttribute("claim", claim);
        PortletMessaging.publish(actionRequest, "claims", "selected", claimId);
    }
}
```

When the hidden parameter is present, we attempt to update the database in the *updateClaimComment* method:

Listing 13.26 The updateClaimComment method

```
private Claim updateClaimComment(String claimId, String comments)
{
    Connection con = null;
    try
    {
        con = cms.getConnection();
        String sql = "UPDATE WWCLAIMS SET COMMENTS = ? WHERE CLAIM_ID = ?";
        PreparedStatement stm = con.prepareStatement(sql);
        stm.setString(1, comments);
        int id = Integer.parseInt(claimId);
        stm.setInt(2, id);
        int count = stm.executeUpdate();
        return lookupClaimStatus(claimId);
    }
    catch (SQLException e)
    {
        System.err.println("Exception updating claim : " + e);
    }
}
```

```

    }
    finally
    {
        releaseConnection(con);
    }
    return new Claim(0, "", "", 0, 0, "", "");
}

```

Claim is an inner class stored in our portlet:

Listing 13.27 The Claim inner class

```

    private final NumberFormat formatter = new DecimalFormat("$###,###.00");

    public Claim(int claimId, String service, String group, double planPaid,
double memberPaid,
                    String status, String comments)
    {
        this.claimId = claimId;
        this.service = service;
        this.group = group;
        this.planPaid = planPaid;
        this.memberPaid = memberPaid;
        this.status = status;
        this.comments = comments;
    }

    public String getPlanPaid()
    {
        return formatter.format(planPaid);
    }

    public String getMemberPaid()
    {
        return formatter.format(memberPaid);
    }

    /**
     * @return Returns the comments.
     */
    public String getComments()
    {
        return comments;
    }

    /**
     * @return Returns the group.
     */
    public String getGroup()
    {
        return group;
    }

    /**
     * @return Returns the service.
     */
    public String getService()

```

```

    {
        return service;
    }
    /**
     * @return Returns the status.
     */
    public String getStatus()
    {
        return status;
    }
    /**
     * @return Returns the claimId.
     */
    public int getClaimId()
    {
        return claimId;
    }
}

```

Portlet CSS style definitions

You may have noticed that some CSS style classes have names like *portlet-section-alternate*. The Portlet API recommends a common CSS style sheet naming convention for links, fonts, messages, sections and forms. See the Portlet API specification appendix, PLT.C for a complete list of recommended styles. If you would like your portlets to have the same overall style as other portlets in the portal, we recommend using these styles whenever possible in your portlet markup.

```
<td nowrap class="portlet-section-alternate" align="right">Comments:&nbsp;  </td>
```

Summary

In this chapter we learned about accessing data in backend systems and integrating that data in portlets using standard methods of data access. We showed you how to integrate portlets with relational databases, Portlet API user preferences, and the Jetspeed database browser portlet. We demonstrated how to use the Model View Controller design pattern in portlets. Additionally, we learned how to develop two portlets that work together, using a simple but effective and portable inter-portlet communication method.

In the process, we developed four portlets for the Waxwing Benefits portal:

1. Client Status portlet
2. Deductible Status portlet
3. Claims Browser portlet
4. Claims Detail portlet

Now that we have added these portlets to our Waxwing Benefit portal let's take a look at some advanced portlet features in the next chapter. We will integrate the Apache Graffito content management system, RSS portlets, the Apache Lucene search engine and access data via a web service.

Chapter 13 Implementing Database Access

After setting up our own Waxwing Benefit portal in the previous chapter, we're ready to start developing more advanced portlets in this chapter. This chapter is about accessing data in backend systems and integrating that data in portlets using standard methods of data access. We will show you how to integrate portlets with data sources via the integration points outlined in table 13.1:

Data Source	Integration Point	Section
Relational Database	JDBC – Java Database Connectivity API	13.2, 13.5
LDAP or User Information Database	Portlet API User Attribute API	13.3
Content Management Systems	Apache Portals Graffito API	Chapter 14
Security Databases	Jetspeed Service API	13.6

Table 13.1 Data Sources and Integration Points demonstrated in this chapter

In section 13.1, we introduce the portlets covered in this chapter. In 13.2, we'll leverage JDBC to access the Waxwing Benefits corporate relational database to generate claims status and benefits reports within a portlet. In section 13.3, we demonstrate how to retrieve generic user information with the Portlet API's User Attributes API. Section 13.4 demonstrates using Velocity to display the data without coupling the data source (model) to the (view). Section 13.5 demonstrates extending the Jetspeed Database Browser portlet to create our own claims browser portlet. In section 13.6, we show how two portlets, a browser and a data entry form, can communicate using portlet messages.

About the Waxwing Benefits member portlets

Before we get into the details of database access and portlets, let's put our custom portal in context. Waxwing Benefits manages benefit packages for companies. Waxwing *members* are employees of companies who partner with Waxwing Benefits, and who often visit the benefits portal. These employees are usually members of a benefit group, such as a medical benefits group.

In this chapter, we are going to look at the database-related portlets provided to users who are assigned the role *member*. As we discussed in chapter 12, the Waxwing Benefits portal implements three roles: Member, Manager, and Admin. When members authenticate (login), they are provided with access to a specific set of portlets. These portlets are designed to meet the needs of the benefits members and are listed in table 13.2:

Member-specific Portlet	Purpose	Data Source	Page
-------------------------	---------	-------------	------

Claims Status	Displays a report of a member' s claim activity via JDBC from a corporate relational database; and user account information via Portlet API User Attributes from portal' s user attribute database.	JDBC, User Attributes	Member Home
Deductible Status	Displays a report of a member' s deductible account details and accumulated benefits paid via JDBC from corporate relational database	JDBC,	Member Home
Claims Browser	Displays a report of all claims filed by a member via JDBC and the database browser from a corporate relational database	JDBC	Claims
Claims Detail	Displays details of a member' s single claim via JDBC from a corporate relational database; and user account information via Portlet API User Attributes from portal' s user attribute database.	JDBC, User Attributes	Claims

Table 13.2 Chapter 13 Portlets

In the sections that follow, we'll show you how to add these portlets to the Waxwing portal. All of the examples in this chapter follow the MVC (Model View Controller) design pattern of portlet development. The data sources are wrapped as the 'model' in beans or as standard attributes. The 'view' is handled in this chapter using Velocity templates. The portlet is the 'controller'. In section 13.2 we demonstrate using beans retrieved from a JDBC data source as the model, the portlet as the controller, and Velocity templates as the view.

Database Access using JDBC: the Claim Status Portlet

JDBC is the standard method for accessing databases in Java. Although most projects layer an object-relational mapping tool on top of JDBC, in this chapter we keep things standard and simple, going straight to the JDBC standard.

The Claim Status Portlet displays information about the member's account in the top half of the portlet view. In the bottom half, the member's benefit claims summary is displayed. The top half of the Claim Status Portlet is retrieved using User Attributes and the Portlet API. The bottom half is retrieved using JDBC from the Waxwing Benefits corporate database. This section concentrates on the JDBC solution.

The screenshot shows a web portlet titled "Claim Status". It contains two main sections. The top section displays member account information in a table-like format: Member Name: Chip Sparrow, Member Company: Eruditorum, Benefits Group: 1700-3, and Benefits Account: 176-946. The bottom section displays a claim summary with three rows: Claims In Progress: 2, Claims Processed: 6, and Awaiting Action: 1. The numbers 2, 6, and 1 are highlighted in yellow.

Figure 13.1 The Claim Status Portlet

Let's have a look at the Claim Status Portlet in Figure 13.1. The *doView* method handles looking up the member account information and claim summary information.

Listing 13.1 ClaimStatus portlet's doView method

```
public void doView(RenderRequest request, RenderResponse response)
    throws PortletException, IOException
{
```



```

Map userInfo =
    (Map) request.getAttribute(PortletRequest.USER_INFO); | #1
Context context = super.getContext(request);
String firstName = (String)userInfo.get("user.name.given");
context.put("firstName", userInfo.get("user.name.given"));
context.put("lastName", userInfo.get("user.name.family"));
context.put("company", userInfo.get("user.employer"));
context.put("group", userInfo.get("benefits.group"));
context.put("account", userInfo.get("benefits.account"));

ClaimStatus status = (ClaimStatus)
    request.getPortletSession().getAttribute("claimStatus");
if (status == null)
{
    // produce report
    Connection con = null;
    try
    {
        con = cms.getConnection(); | #2
        Integer processed = | #3
            countClaimStatus(con,
                request.getUserPrincipal().toString(), "PROCESSED");
        Integer inProcess =
            countClaimStatus(con,
                request.getUserPrincipal().toString(), "*** OPEN ***");
        Integer awaitingAction =
            countClaimStatus(con,
                request.getUserPrincipal().toString(), "*** HOLD ***");
        status = new ClaimStatus(processed, inProcess,
            awaitingAction);
        request.getPortletSession().setAttribute("claimStatus",
            status);
    }
    catch (SQLException e)
    {
        System.err.println("Exception retrieving claim status: " + e);
    }
    finally
    {
        releaseConnection(con); | #4
    }
}

context.put("claimStatus", status); | #5
super.doView(request, response); | #6
}

```

(annotation) <#1 retrieving user attributes, see section 13.3 >

(annotation) <#2 get a JDBC connection>

(annotation) <#3 calculate claim status report>

(annotation) <#4 release the JDBC connection>

(annotation) <#5 the claims model bean is put in the velocity view>

(annotation) <#6 let the velocity portlet bridge handle processing the view>

We're going to skip over the first eight lines of code in the `doView` method, as that is covered in section 13.3 on User Attributes. The section concentrates on retrieving the claims status data from a relational database via JDBC, starting with getting a connection.

Accessing the Benefits Database in the Claims Status Portlet via JDBC

Members login to the portal to check their claims status. The claims status portlet accesses the corporate benefits database to produce a claims status report for the current user. The status report provides up-to-date claim information on a per-user basis. The current user is determined using the Portlet API. Specifically, the *PortletRequest* class provides a *getUserPrincipal* method which is then passed into the report generation code:

```
Integer processed =
    countClaimStatus(con,
        request.getUserPrincipal().toString(), "PROCESSED");
```

The portal's user principal is also stored as a column in the claims table of the claims database. When calculating our report, we only count claims that are for the current user principal. Since retrieving from a database can sometimes be an expensive operation, we demonstrate how to cache the claims status report in the portlet session. By caching the calculated report data in the portlet session, the report only calculates once per user, when the user starts a new session. By logging out, the user terminates the portlet session. The next time the user logs on; the report result is cleared from the portlet session and thus must be regenerated. Here is how to get an attribute out of the portlet session using the Portlet API:

```
ClaimStatus status = (ClaimStatus)
    request.getPortletSession().getAttribute("claimStatus");
if (status == null)
{
```

We are looking for an attribute named *claimStatus*. Since we put the attribute in the session, we know it is of the type *ClaimStatus*. When a user hits this page for the first time after logging on, the attribute is not found and we generate the report:

Listing 13.2 Generating a claim status report

```
// produce report
Connection con = null;
try
{
    con = cms.getConnection();
    Integer processed =
        countClaimStatus(con,
            request.getUserPrincipal().toString(), "PROCESSED");    |#1
    Integer inProcess =
        countClaimStatus(con, request.getUserPrincipal().toString(),
            "*** OPEN ***");    |#2
    Integer awaitingAction =
        countClaimStatus(con, request.getUserPrincipal().toString(),
            "*** HOLD ***")    |#3;
    status = new ClaimStatus(processed, inProcess, awaitingAction);
    request.getPortletSession().setAttribute("claimStatus", status);
}
catch (SQLException e)
{
    System.err.println("Exception retrieving claim status: " + e);
```

```

    }
    finally
    {
        releaseConnection(con);
    }
}
(annotation) <#1 count claims for PROCESSED status>
(annotation) <#2 count claims for OPEN status>
(annotation) <#3 count claims for HOLD status>

```

Note that we grab a database connection once, and share it across three method calls to *countClaimStatus*. The *countClaimStatus* method counts the number of claims for the given status:

Listing 13.3 JDBC code to calculate a claim status count

```

public Integer countClaimStatus(Connection con, String principal, String status)
throws SQLException
{
    String select =
        "select COUNT(*) from WWCLAIMS where USER_ID = ? and status = ?";
    PreparedStatement stm = con.prepareStatement(select);
    stm.setString(1, principal);
    stm.setString(2, status);
    ResultSet rs = stm.executeQuery();
    rs.next();
    int x = rs.getInt(1);
    rs.close();
    stm.close();

    return new Integer(x);
}

```

Although we have used JDBC directly here, in modern Java applications it is better to use object-relational mapping tools. Two popular, open source object-relational tools are the Hibernate or Apache OJB projects. For simplicity and a common ground of understanding, we use the JDBC standard.

A new *ClaimStatus* object is created:

```
status = new ClaimStatus(processed, inProcess, awaitingAction);
```

The claim status is then put in the session and cached:

```
request.getPortletSession().setAttribute("claimStatus", status);
```

Finally, the *ClaimStatus* object is put in the Velocity context:

```
context.put("claimStatus", status);
```

From there, our template can access the *ClaimStatus* object to render the report output:

Listing 13.4 Velocity template displayed calculated claim status in a portlet's view

```

<hr/>
<table>
    <tr>

```

```

        <td class='portlet-form-label'>Claims In Progress:</td>
        <td class='portlet-section-selected'>
            $claimStatus.InProgress      | #1
        </td>
    </tr>
    <tr>
        <td class='portlet-form-label'>Claims Processed:</td>
        <td class='portlet-section-selected'>
            $claimStatus.Processed      | #2
        </td>
    </tr>
    <tr>
        <td class='portlet-form-label'>
            Awaiting Action:
        </td>
        <td class='portlet-section-selected'>
            $claimStatus.AwaitingAction | #3
        </td>
    </tr>
</table>

```

(annotation) <#1 display the count of IN PROGRESS claims>

(annotation) <#2 display the count of PROCESSED claims>

(annotation) <#3 display the count of IN AWAITING ACTIONclaims>

This gives us the claim status report (Figure 13.2) displayed in the bottom half of the portlet output:

Claims In Progress:	2
Claims Processed:	6
Awaiting Action:	1

Figure 13.2 The rendered output of the Claim Status portlet

The *ClaimStatus* is an inner class stored in our portlet:

Listing 13.5 The ClaimStatus bean

```

public class ClaimStatus
{
    private Integer processed;
    private Integer inProgress;
    private Integer awaitingAction;

    public ClaimStatus(Integer processed, Integer inProgress,
        Integer awaitingAction)
    {
        this.processed = processed;
        this.inProgress = inProgress;
        this.awaitingAction = awaitingAction;
    }

    /**
     * @return Returns the awaitingAction.
     */
    public Integer getAwaitingAction()

```

```

    {
        return awaitingAction;
    }
    /**
     * @return Returns the inProgress.
     */
    public Integer getInProgress()
    {
        return inProgress;
    }
    /**
     * @return Returns the processed.
     */
    public Integer getProcessed()
    {
        return processed;
    }
}
}

```

Next we look into how the top half of the claims status report is generated using User Attributes from the Portlet API.

Accesssing user attributes

The member account details are gathered using a standard Portlet API feature called *User Attributes*. User Attributes are a set of shared attributes about the current user made available by the portal to every portlet that asks for them. The Portlet API requires portals to provide a set of user attributes available at runtime. This set of user attributes are requested by portlets by calling the *getAttribute* method on the *PortletRequest* class (and thus both the *RenderRequest* and *ActionRequest*):

```
Map userInfo = (Map) request.getAttribute(PortletRequest.USER_INFO);
```

The map provided is a subset of all available user attributes to this portlet application. The deployment descriptor defines the user attributes required by the portlet application. The portal only populates the User Info map with user attributes specified in the portlet application's deployment descriptor, shown in Listing 13.6:

Listing 13.6 User Attributes in the portlet.xml deployment descriptor

```

<user-attribute>
  <description>User Given Name</description>
  <name>user.name.given</name>
</user-attribute>
<user-attribute>
  <description>User Last Name</description>
  <name>user.name.family</name>
</user-attribute>
<user-attribute>
  <description>Employer</description>
  <name>user.employer</name>
</user-attribute>
<user-attribute>

```

```

    <description>Benefits Group</description>
    <name>benefits.group</name>
  </user-attribute>
  <user-attribute>
    <description>Benefits Account</description>
    <name>benefits.account</name>
  </user-attribute>

```

The Portlet API does not specify how user attributes are configured or maintained in a portal. Jetspeed has the ability to retrieve user attributes from one or more user attribute services. This scenario can be very useful in a real world portal. For example, in our benefits company, we could provide some user attributes from the Benefits database, while other user attributes are retrieved from a centralized LDAP repository. By logging on as the *admin* user, you can manage all user attributes in the User Management portlet (Figure 13.3):

The screenshot shows a web interface titled "User Detail Information". At the top, it says "Principal : TOMCAT". Below this are five tabs: "Attributes", "Password", "Role", "Group", and "Profile". The "Attributes" tab is selected, displaying a table with two columns: "Name" and "Value". The table contains five rows of user attributes, each with a checkbox in the first column:

	Name	Value
<input type="checkbox"/>	user.name.given	Tom
<input type="checkbox"/>	user.name.family	Cat
<input type="checkbox"/>	user.employer	Apache
<input type="checkbox"/>	benefits.group	3939-23
<input type="checkbox"/>	benefits.account	99533-1

At the bottom of the table are two buttons: "Update" and "Remove".

Figure 13.3 Jetspeed User Management portlet: editing user attributes

After retrieving the user attributes, we need to include them in the rendered content. For this example, we use the Velocity Bridge and Velocity template engine. Velocity makes use of a simple context object to hold Java objects that can be accessed in a template. The objects are put into the context in the Java portlet code:

Listing 13.7 Putting user attributes in the Velocity context

```

Context context = super.getContext(request);
String firstName = (String)userInfo.get("user.name.given");
context.put("firstName", userInfo.get("user.name.given"));
context.put("lastName", userInfo.get("user.name.family"));
context.put("company", userInfo.get("user.employer"));
context.put("group", userInfo.get("benefits.group"));
context.put("account", userInfo.get("benefits.account"));

```

The Java objects are then pulled out of the context in the Velocity template using \$ notation. See Chapter 11 for more details on using the Velocity Bridge:

Listing 13.8 A Velocity template takes the user attributes out of the Velocity context

```

<table border="1" cellspacing="1" cellpadding="3">
  <tr>
    <td class='portlet-form-label'>Member Name:</td>
    <td class='portlet-section-alternate'>${firstName} ${lastName}</td>      | #1
  </tr>
  <tr>
    <td class='portlet-form-label'>Member Company:</td>
    <td class='portlet-section-alternate'>${company}</td>>                | #2
  </tr>
  <tr>
    <td class='portlet-form-label'>Benefits Group:</td>
    <td class='portlet-section-alternate'>${group}</td>>                  | #3
  </tr>
  <tr>
    <td class='portlet-form-label'>Benefits Account:</td>
    <td class='portlet-section-alternate'>${account}</td>>                | #4
  </tr>
</table>
(annotation) <#1 display first and last name>
(annotation) <#2 display company>
(annotation) <#3 display group>
(annotation) <#4 display account>

```

All user attributes are stored and accessed as strings. (For simplicity, we have not localized the labels used in this example.) This gives us the member account information displayed in the top half of the portlet output (Figure 13.4):

Claim Status	
Member Name:	Sir Robin
Member Company:	Eruditorum
Benefits Group:	1700-3
Benefits Account:	176-945

Figure 13.4 User Attributes rendered in the Claim Status portlet

This completes our coverage of user attributes in this chapter. We have learned how to access a Waxwing Benefits user account information from the portal using the standardized User Attributes API. The Portlet API shielded us from the actual implementation of the user attributes store; the attributes could have been stored in a LDAP or relational database. The implementation details are hidden in the portal and the portlet only had to know about the standardized portlet interfaces. Next, we will briefly cover how we initialize a portlets connection to a CMS repository during a portlet's initialization phase.

12.2 Implementing MVC with the Deductible Status Portlet

The MVC) pattern is very common in web applications. Portlet applications should follow the same design patterns. In this section we explore how to implement the view portion of MVC, as well as the controlling portlet. Although we write JDBC code in our portlets, the abstraction is clear between view and model, as we only use Deductible model beans in our view. We'll start with looking at the controller implementation: the

doView method of the Deductible Status portlet, followed by the model which uses JDBC to create a Deductible model bean. Finally, we display the Deductible content in a portlet using a Velocity view.

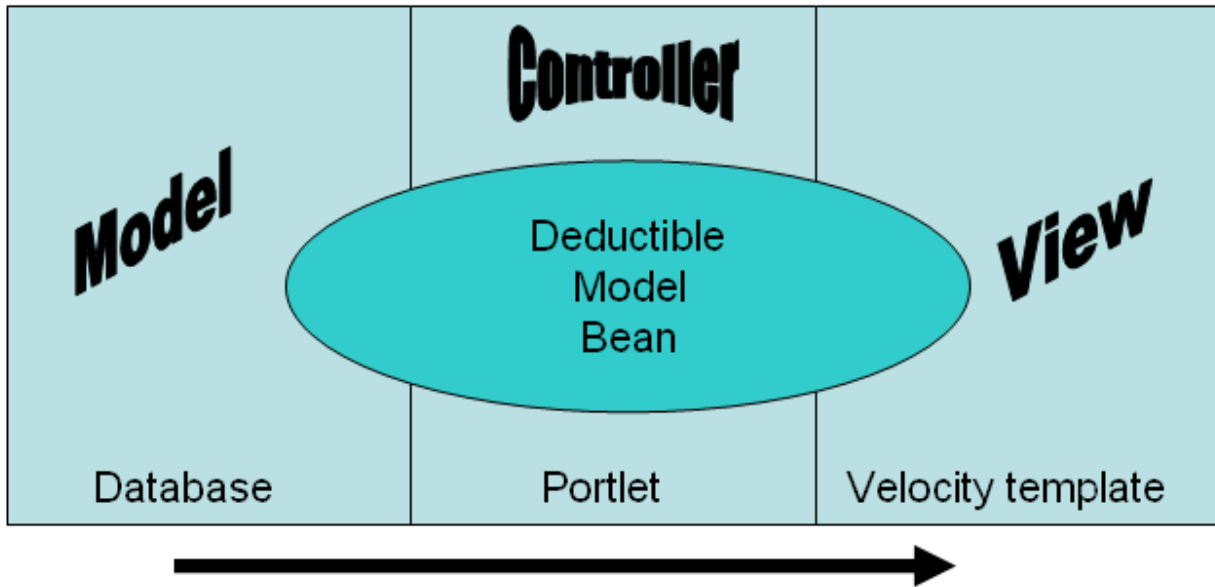


Figure 13.5 The deductible model bean originates in the database, and is rendered by the portlet into the velocity view

The doView method

On our Waxwing Benefits example's Home Page, the Deductible Status Portlet (Figure 13.6) displays information about the member's deductible premiums and accumulated deductibles over time:

Deductible Status		
	Deductible Caps	Accumulated
Lifetime Maximum	\$12,000,000.00	\$385,000.00
Individual Deductible	\$250.00	\$250.00
Individual Deductible	\$650.00	\$515.00
Individual Out-of-Pocket	\$1,500.00	\$550.00
Family Out-of-Pocket	\$3,000.00	\$1,250.00

Figure 13.6 The rendered output of the Deductible Status portlet

Let's have a look at the Deductible Status Portlet source code. The *doView* method handles looking up the deductible status:

Listing 13.9 doView method of the Deductible Status portlet

```
public void doView(RenderRequest request, RenderResponse response)
    throws PortletException, IOException
{
    Context context = super.getContext(request);
    Deductible deductible = (Deductible)
        request.getPortletSession().getAttribute("deductible");
    if (deductible == null)
    {
        deductible =
```



```

        lookupDeductible(request.getUserPrincipal().toString());
        request.getPortletSession().setAttribute("deductible", deductible);
    }

    context.put("deductible", deductible);
    super.doView(request, response);
}

```

The portlet accesses the corporate benefits database to produce a deductibles report for the current user. This status report provides up-to-date deductible information on a per-user basis.

Identifying the current user

The current user is determined using the Portlet API. Specifically, the *PortletRequest* class provides a *getUserPrincipal* method which is then passed into the report generation code:

```
deductible = lookupDeductible(request.getUserPrincipal().toString());
```

The portal's user principal is also stored as a column in the deductible table of the claims database. When calculating our report, we only count deductible records that are for the current user principal.

Caching the report data

Since retrieving from a database can sometimes be an expensive operation, we demonstrate how to cache the deductible status report in the portlet session. By caching the calculated report data in the portlet session, the report only calculates once per user, when the user starts a new session. By logging out, the user terminates the portlet session. The next time the user logs on; the report result is cleared from the portlet session and thus must be regenerated.

Here is how to get an attribute out of the portlet session using the Portlet API:

```

Deductible deductible = (Deductible)
request.getPortletSession().getAttribute("deductible");
if (deductible == null)
{

```

We are looking for an attribute named *deductible*. Since we put the attribute in the session, we know it is of the type *Deductible*. When a user hits this page for the first time after logging on, the attribute is not found and we generate the report.

Listing 13.10 lookup a deductible record for the given user (principal)

```

public Deductible lookupDeductible(String principal)
{
    Connection con = null;
    try
    {
        con = cms.getConnection();

        String select =
            "select LIFETIME_MAX, IND_DEDUCT, FAM_DEDUCT, IND_OOP, FAM_OOP, "+
            " ACC_LIFETIME, ACC_IND_DEDUCT, ACC_FAM_DEDUCT, ACC_IND_OOP, " +
            "ACC_FAM_OOP" +
            " from WWDEDUCTIBLE where USER_ID = ?";

```

```

        PreparedStatement stm = con.prepareStatement(select);
        stm.setString(1, principal);
        ResultSet rs = stm.executeQuery();           | #2
        rs.next();
        double lifetimeMax = rs.getDouble(1);
        double individualDeductible = rs.getDouble(2);
        double familyDeductible = rs.getDouble(3);
        double individualOutOfPocket = rs.getDouble(4);
        double familyOutOfPocket = rs.getDouble(5);
        double accLifetimeMax = rs.getDouble(6);
        double accIndividualDeductible = rs.getDouble(7);
        double accFamilyDeductible = rs.getDouble(8);
        double accIndividualOutOfPocket = rs.getDouble(9);
        double accFamilyOutOfPocket = rs.getDouble(10);

        rs.close();
        stm.close();

        return new Deductible(lifetimeMax,           | #3
                               individualDeductible,
                               familyDeductible,
                               individualOutOfPocket,
                               familyOutOfPocket,
                               accLifetimeMax,
                               accIndividualDeductible,
                               accFamilyDeductible,
                               accIndividualOutOfPocket,
                               accFamilyOutOfPocket);
    }
    catch (SQLException e)
    {
        System.err.println("Exception retrieving deductible status: " + e);
    }
    finally
    {
        releaseConnection(con);
    }

    return new Deductible(0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
}
(annotation) <#1 create the SQL statement>
(annotation) <#2 execute the SQL statement>
(annotation) <#1 create a new Deductible object>

```

In listing 13.10, the complete `lookupDeductible` method is shown. Here we get a JDBC connection, retrieve the deductible record from the database using a JDBC prepared statement, and return a new `Deductible` bean.

Back to the controller code in the portlet's `doView` method: the bean is passed on to the Velocity view by putting the *Deductible* object in the Velocity context:

```

deductible = lookupDeductible(request.getUserPrincipal().toString());
... context.put("deductible", deductible);

```

Rendering the report

From there, our template can access the *Deductible* object to render the report output:

Listing 13.11 Velocity template to render a deductible record

```
<table border="1" cellspacing="1" cellpadding="3">
  <tr>
    <th></th>
    <th class='portlet-form-label'>Deductible Caps</th>
    <th class='portlet-form-label'>Accumulated</th>
  </tr>
  <tr>
    <td class='portlet-section-alternate'>Lifetime Maximum</td>
    <td class='portlet-section-alternate'>${deductible.LifetimeMax}</td>
    <td class='portlet-section-alternate'>${deductible.AccLifetimeMax}</td>
  </tr>
  <tr>
    <td class='portlet-section-alternate'>Individual Deductible</td>
    <td class='portlet-section-alternate'>
      ${deductible.IndividualDeductible}
    </td>
    <td class='portlet-section-alternate'>
      ${deductible.AccIndividualDeductible}
    </td>
  </tr>
  <tr>
    <td class='portlet-section-alternate'>Individual Deductible</td>
    <td class='portlet-section-alternate'>
      ${deductible.FamilyDeductible}
    </td>
    <td class='portlet-section-alternate'>
      ${deductible.AccFamilyDeductible}</td>
  </tr>
  <tr>
    <td class='portlet-section-alternate'>Individual Out-of-Pocket</td>
    <td class='portlet-section-alternate'>
      ${deductible.IndividualOutOfPocket}
    </td>
    <td class='portlet-section-alternate'>
      ${deductible.AccIndividualOutOfPocket}
    </td>
  </tr>
  <tr>
    <td class='portlet-section-alternate'>Family Out-of-Pocket</td>
    <td class='portlet-section-alternate'>
      ${deductible.FamilyOutOfPocket}</td>
    <td class='portlet-section-alternate'>
      ${deductible.AccFamilyOutOfPocket}</td>
  </tr>
</table>
```

Velocity templates are perhaps the easiest to understand of all Java-based templates. (For a quick review of Velocity, see chapter 11 on the Velocity Bridge.) The deductible model bean is put

into the Velocity context, where it can then be pulled out by the template using the `$deductible` variable. For getters on beans, Velocity allows for a short-hand syntax where the “get” prefix is not required and you can instead simply enter `$deductible.LifetimeMax` for the getter method `getLifetimeMax()`. This is how we pull the dynamic content from the model into the view. Let's next have a closer look at the Deductible model bean.

Storing the Deductible inner class

The *Deductible* bean is an inner class stored in our portlet:

Listing 13.12 Deductible bean

```
public class Deductible
{
    private double lifetimeMax;
    private double individualDeductible;
    private double familyDeductible;
    private double individualOutOfPocket;
    private double familyOutOfPocket;
    private double accLifetimeMax;
    private double accIndividualDeductible;
    private double accFamilyDeductible;
    private double accIndividualOutOfPocket;
    private double accFamilyOutOfPocket;

    public Deductible(double lifetimeMax,
                     double individualDeductible,
                     double familyDeductible,
                     double individualOutOfPocket,
                     double familyOutOfPocket,
                     double accLifetimeMax,
                     double accIndividualDeductible,
                     double accFamilyDeductible,
                     double accIndividualOutOfPocket,
                     double accFamilyOutOfPocket)

    {
        this.lifetimeMax = lifetimeMax;
        this.individualDeductible = individualDeductible;
        this.familyDeductible = familyDeductible;
        this.individualOutOfPocket = individualOutOfPocket;
        this.familyOutOfPocket = familyOutOfPocket;
        this.accLifetimeMax = accLifetimeMax;
        this.accIndividualDeductible = accIndividualDeductible;
        this.accFamilyDeductible = accFamilyDeductible;
        this.accIndividualOutOfPocket = accIndividualOutOfPocket;
        this.accFamilyOutOfPocket = accFamilyOutOfPocket;
    }

    private final NumberFormat formatter =
        new DecimalFormat("$###,###.00");

    /**
     * @return Returns the familyDeductible.
     */
}
```

```

public String getFamilyDeductible()
{
    return formatter.format(familyDeductible);
}
/**
 * @return Returns the familyOutOfPocket.
 */
public String getFamilyOutOfPocket()
{
    return formatter.format(familyOutOfPocket);
}
/**
 * @return Returns the individualDeductible.
 */
public String getIndividualDeductible()
{
    return formatter.format(individualDeductible);
}
/**
 * @return Returns the individualOutOfPocket.
 */
public String getIndividualOutOfPocket()
{
    return formatter.format(individualOutOfPocket);
}
/**
 * @return Returns the lifetimeMax.
 */
public String getLifetimeMax()
{
    return formatter.format(lifetimeMax);
}
/**
 * @return Returns the accFamilyDeductible.
 */
public String getAccFamilyDeductible()
{
    return formatter.format(accFamilyDeductible);
}
/**
 * @return Returns the accFamilyOutOfPocket.
 */
public String getAccFamilyOutOfPocket()
{
    return formatter.format(accFamilyOutOfPocket);
}
/**
 * @return Returns the accIndividualDeductible.
 */
public String getAccIndividualDeductible()
{
    return formatter.format(accIndividualDeductible);
}
/**
 * @return Returns the accIndividualOutOfPocket.

```

```

    */
    public String getAccIndividualOutOfPocket()
    {
        return formatter.format(accIndividualOutOfPocket);
    }
    /**
     * @return Returns the accLifetimeMax.
     */
    public String getAccLifetimeMax()
    {
        return formatter.format(accLifetimeMax);
    }
}
}

```

As you can see, following the MVC pattern leads to other good practices such as abstraction, encapsulation and separation of concerns when developing portlets. Although we used JDBC directly in our portlet, you could easily swap out the one method in a real world application and replace it with a popular object-relational mapping framework. But the *lookupDeductible* method should still return a model bean, which would then require no changes to your view. Likewise, you could replace the Velocity templates with JSP, but the model code would not care, since it only works with a Deductible bean and not the details of the template.

Next in section 13.5, we will further examine developing portlets with databases by exploring the Jetspeed Database browser portlet.

Browsing databases: the claims browser and claims detail portlets

The Claims Browser and Claims Detail Portlets are two portlets that work together to provide a combined user interface experience. The browser displays a list of all claims for the current user. When the user clicks on one of these rows, the browser communicates with the Claims Detail portlet, telling it which row was selected. The detail portlet then displays the detailed information for the selected claim in its portlet window.

When creating your portal page, make sure to put the Claims Detail portlet next to the Claims Browser portlet, as is done in Figure 13.7.

The screenshot shows two portlets side-by-side. The left portlet, titled 'Database Browser', contains a table with the following data:

Service	Group	Status
Diagnostic Radiology	MedPlan 1000	PROCESSED
Emergency Op Room Charges	MedPlan 1000	PROCESSED
Other Medical Services	CompPlan 2000	** OPEN **
Prescriptions	MedPlan 1000	PROCESSED
Hospital Misc	MedPlan 1000	PROCESSED
Lab	CompPlan 2000	** HOLD **
Surgery	CompPlan 2000	PROCESSED
Medical Visit	MedPlan 1000	PROCESSED
Room and Board	MedPlan 1000	** OPEN **

Below the table is a pagination bar showing '0 of 9' and buttons for 'Go' and 'Refresh'.

The right portlet, titled 'Claims Detail', displays the following information:

- Member Name: Chip Sparrow
- Member Company: Eruditorum
- Benefits Group: 1700-3
- Benefits Account: 176-946

Below this is a table with the following data:

Service	Diagnostic Radiology
Group	MedPlan 1000
Paid by Plan	\$111.00
Paid by Member	\$88.00
Status	PROCESSED

At the bottom of the Claims Detail portlet is a 'Comments' section with a text input field containing 'I need to get a more detailed lab report from the provider' and a 'Make Request' button.

Figure 13.7 The Claims Browser and Claims Detail portlets side by side

The Claims Detail portlet is somewhat dependent on the Claims Browser portlet. Since the two portlets are loosely coupled, the Claims Detail portlet could just as easily get a primary key from

another portlet to look up the current claim record. The Claims Browser portlet, though, is not dependent on the Claims Detail portlet. The browser can be used by itself.

In the following section, we will discuss how the Waxwing example allows users to browse claims by extending the Jetspeed database browser portlet.

Customizing The Jetspeed database browser portlet

The database browser portlet comes with the Jetspeed-2 distribution. It is based on the Velocity Bridge and renders all content with the Velocity template engine. The Database Browser portlet is a standardized portlet that will port to any portal (portlet container). It does require that jars from the Apache Portals Bridges project are included.

The Waxwing Claims Browser portlet is an example of extending an existing portlet and customizing it to read over our claims table in the Waxwing corporate database.. Let's take a look at the amount of Java code that we actually wrote to make this work:

Listing 13.13 ClaimsBrowserPortlet class

```
public class ClaimsBrowserPortlet extends DatabaseBrowserPortlet
{
    public ClaimsBrowserPortlet()
    {
    }

    protected void readSqlParameters(RenderRequest request)
    {
        List sqlParameters = new LinkedList();
        sqlParameters.add(request.getUserPrincipal().toString());
        setSQLParameters(sqlParameters);
    }

    public void processAction(ActionRequest actionRequest,
                             ActionResponse actionResponse)
        throws PortletException, IOException
    {
        String claim = actionRequest.getParameter("claim");

        // publish change to ClaimsDetail Portlet
        if (claim != null)
            PortletMessaging.publish(actionRequest,
                                     "claims", "selected", claim);

        super.processAction(actionRequest, actionResponse);
    }
}
```

As you can see, there isn't a whole lot there. This is usually the sign of a useful base portlet class. Before we explain the code, let's step back and look at how to configure a Database Portlet.

Configuring a database portlet

In order to connect to and query a relational table, each instance of a database portlet must be configured. Minimally, you will need an SQL statement and JDBC connection parameters. The *portlet.xml* contains the configuration:

Listing 13.14 portlet.xml deployment descriptor for the ClaimsBrowserPortlet

```
<portlet>
  <description>Claims Browser</description>
  <portlet-name>ClaimsBrowser</portlet-name>
  <display-name>Claims Browser</display-name>
  <portletclass>
    com.waxwingbenefits.portal.portlets.ClaimsBrowserPortlet</portlet-class>
  <init-param>
    <name>ViewPage</name>      | #1
    <value>/WEB-INF/views/database-view.vm</value>
  </init-param>
  <init-param>
    <name>EditPage</name>      | #2
    <value>/WEB-INF/views/database-edit.vm</value>
  </init-param>
  <init-param>
    <name>HelpPage</name>      | #3
    <value>/WEB-INF/views/database-help.vm</value>
  </init-param>
  <expiration-cache>0</expiration-cache>
  <supports>
    <mime-type>text/html</mime-type>
    <portlet-mode>VIEW</portlet-mode>
    <portlet-mode>EDIT</portlet-mode>
  </supports>
  <supported-locale>en</supported-locale>
  <supported-locale>de</supported-locale>
  <resource-bundle>
    org.apache.portals.gems.browser.resources.Browser</resource-bundle>
  ...

```

(annotation) <#1 the ViewPage init parameter>

(annotation) <#2 the EditPage init paramter>

(annotation) <#3 the HelpPage Init parameter>

There are three initialization parameters required:

4. View Page – the View Mode template
5. Edit Page – the Edit Mode template
6. Help Page – the Help Mode template

Each one of these init parameters defines the template that is used to display the corresponding portlet mode. All three of these templates have been borrowed from the Jetspeed portlets. We will be discussing the view and ed it mode templates in more detail throughout this section. First, lets look at the available connection configuration parameters.

Configuring database connection parameters

The database connection parameters are configured directly in the portlet preferences from edit mode. Figure 13.8 shows how to configure a JDBC connection:

The screenshot shows a web-based preferences editor titled "Database Browser". It contains several sections for configuring database connections:

- Database Browser Preferences**: A header section.
- Data Source Type**: Three radio buttons are present: "JNDI Data Source", "DBCP Data Source" (which is selected), and "SSO Credential Store". Below them is the text "Please Select a Data Source Type".
- JNDI Settings**: A section with a "JNDI Data Source" text input field.
- JDBC/DBCP Settings**: A section with four text input fields: "JDBC Driver" (containing "org.hsqldb.jdbcDriver"), "JDBC Connection" (containing "jdbc:hsqldb:hsq://127.0.0.1:9001"), "JDBC Username" (containing "sa"), and "JDBC Password" (empty).
- J2 SSO Settings**: A section with three text input fields: "JDBC Driver" (containing "org.hsqldb.jdbcDriver"), "JDBC Connection" (containing "jdbc:hsqldb:hsq://127.0.0.1:9001"), and "SSO Site" (empty).
- General Settings**: A section with a "Window Size" text input field (containing "10") and an "SQL" text area (containing "select * from CLUBS").

At the bottom of the form are two buttons: "Save" and "Test".

Figure 13.8 Configuring database connection parameters in the preferences editor

From edit mode, you will see the database preferences editor. This editor is an updatable view over the portlet preferences defined in your portlet.xml descriptor:

Listing 13.15 Portlet Preferences for Database Browser Portlets

```
<portlet-preferences>
  <preference>
    <name>DatasourceType</name>
    <value>dbcp</value>
  </preference>
  <preference>
    <name>JdbcDriver</name>
    <value>com.mysql.jdbc.Driver</value>
  </preference>
  <preference>
    <name>JdbcConnection</name>
    <value>jdbc:mysql://j2-server/waxwing</value>
  </preference>
  <preference>
    <name>JdbcUsername</name>
    <value>john</value>
  </preference>
  <preference>
    <name>JdbcPassword</name>
    <value>secret </value>
  </preference>
</portlet-preferences>
```

Preference	Value (example)	Description
DatasourceType	1. dbcp 2. jndi 3. sso	1. a managed pool of JDBC connection 2. a JNDI database connection managed by the app server 3. SSO – Jetspeed SSO (Single Signon) – only applicable with Jetspeed-2 portal
JdbcDriver	com.mysql.jdbc.Driver	The class name of the JDBC driver
JdbcConnection	:mysql://j2-server/waxwing	The database-specific JDBC connection string
JdbcUsername	John	The username on the database
JdbcPassword	Secret	The password on the database

Table 13.3 JDBC connection parameters

Although putting the JDBC connection properties is useful for this book example, it is usually best to use an application server managed-JNDI connection.

Using a JNDI and Jetspeed-2 SSO connection

The Java Naming and Directory Interface (JNDI) is part of the Java platform, providing applications based on Java technology with a unified interface to multiple naming and directory services. You can build powerful and portable directory-enabled applications using this industry standard. Most application servers, including Tomcat, support JNDI lookups for application server resources such as data sources and database connection pools. Once the JNDI resource is configured in the application server, the database browser portlet simply has to reference it by name to get connections to the application server-managed database.

Listing 13.16 JNDI connection parameters in portlet preferences

```
<portlet-preferences>
  <preference>
    <name>DatasourceType</name>
    <value>jndi</value>
  </preference>
</portlet-preferences>
```

```

<preference>
  <name>MySource</name>
  <value></value>
</preference>

```

Another useful option is using the Jetspeed-2 SSO feature, although it is only available when the portlet is deployed to the Jetspeed-2 portal. Jetspeed-2 has a Single Sign-on (SSO) repository. Besides being used to store common database credentials, the SSO repository can also store credentials used to logon to other applications that the portal requires access to.

WindowSize preference

The *WindowSize* preference configures how many rows are displayed in the portlet window.

```

<preference>
  <name>WindowSize</name>
  <value>10</value>
</preference>

```

SQL preference

The *sql* preference configures the SQL select statement string.

```

<preference>
  <name>sql</name>
  <value>select CLAIM_ID, SERVICE as "Service", GROUPE as "Group", STATUS as
"Status" from WWCLAIMS where USER_ID = ?</value>
</preference>

```

We select from the WWCLAIMS table to retrieve all claims for the current user. Note that the question mark (?) is a placeholder used by JDBC when we prepare the SQL statement and bind a user name to the prepared statement. We select four columns, three of which will be visible in the portlet view. We also make use of giving the columns alternate names using the AS clause of the SELECT statement. Here we simply format the column name to be capitalized instead of all uppercase:

```
SERVICE as "Service"
```

In your customized database browser portlets, you probably do not want end users editing the SQL preference for obvious reasons. This concludes our discussion of connection parameters. Next, let's start looking at what you can do programmatically with the database browser portlet.

Using SQL query parameters

Having a static SQL string defined in your portlet preferences is very useful. But sometimes you will need to dynamically bind runtime criteria to an SQL query. If you need to query based on runtime parameters, you will need to override the *readSqlParameters* method. For example, here is how to filter the SQL query::

```

protected void readSqlParameters(RenderRequest request)
{
    List sqlParameters = new LinkedList();
    sqlParameters.add(request.getUserPrincipal().toString());
    setSQLParameters(sqlParameters);
}

```

```
}
```

The *readSqlParameters* method of the *DatabaseBrowserPortlet* can be overridden to provide a list of SQL objects that are passed in the browser's internal prepared statement. These objects must be in an ordered list, provided in the order necessary for substituting into the prepared statement. In the example above, we get the user principal from the Portlet API, and provide it to the browser via the callback message. Thus, if the current logged on user is named *robin*, the resulting dynamic SQL statement becomes:

```
select CLAIM_ID, SERVICE as "Service", GROUPE as "Group", STATUS as "Status" from
WWCLAIMS where USER_ID = 'robin'
```

The query executes and generates a model that can be traversed by a Velocity portlet.

The Database Browser Velocity Template

We configure the Waxwing database browser's Velocity template in the *portlet.xml* configuration:

```
<init-param>
  <name>ViewPage</name>
  <value>/WEB-INF/views/database-view.vm</value>
</init-param>
```

First, look at the section of the template that creates the headings. The *DatabaseBrowserPortlet* provides a velocity variable named *\$title*, holding an array of column titles for each column in the result set. All of the column titles are generated from the result set metadata.

Listing 13.17 Rendering the Column Headers

```
<table cellpadding=0 cellspacing=1 border=0 width="100%">
  <tbody>
    <tr>
      #foreach ($column in $title)      |#1
        #if ($velocityCount == 1)
        #else
        #set ($columnLink = $renderResponse.createRenderURL())      |#2
        $columnLink.setParameter("js_dbcolumn",$column)
        <td align=CENTER class="jetdbHead" width="43"
          nowrap onClick="window.location.href='$columnLink'">
          <div align="center">$column</div>
        </td>
        #end
      #end
      #if ($rowLinks)
        #if ($rowLinks.size() > 0)
          <td align=CENTER width="43" class="jetdbHead"></td>
        #end
      #end
    </tr>
```

(annotation) <#1 create column headers from velocity \$title variable>

(annotation) <#2 create a render URL link for each column header>

Note that we create links in the column headers. These links are created using the *renderResponse* object (provided by the *DatabaseBrowserPortlet*'s base class, *GenericVelocityPortlet*). The *GenericVelocityPortlet* provides the Velocity variables *\$renderRequest* and *\$renderResponse*, just like the Portlet API equivalents provided by the Portlet API tag library for JSP templates.

```
#set ($columnLink = $renderResponse.createRenderURL())
$columnLink.setParameter("js_dbcolumn",$column)
```

The render parameter *js_dbcolumn* is passed to the *DatabaseBrowserPortlet*. When it sees this render parameter, the browser knows that a request is being made to sort its content on the given column. Each row from the table is rendered inside of a doubly-nested for loop. The outer loop walks through the result set's rows using the velocity variable named *\$table*.

Listing 13.18 Iterating over the rows in the table

```
#foreach ( $row in $table )
<tr>
  #if ($velocityCount % 2 == 0)
    #set ($rowstyle = "jetdbEven")
  #else
    #set ($rowstyle = "jetdbOdd")
  #end
```

The inner for loop walks through each column for the current row. This row/column model can be easily traversed using a generic algorithm that is easily customizable. Each iteration of this loop generates a column for the current row:

Listing 13.19 Rendering the database browser columns for the current row

```
#foreach ( $entry in $row )      |#1
  ## specialized for this CLAIMS query ONLY, skip over column 1
  #if ($velocityCount == 1)
    #set ($rowid = $entry)
  #elseif ($velocityCount == 2)
    #set ($claimLink = $renderResponse.createActionURL())      |#2
    $claimLink.setParameter("claim",$rowid.toString())
    <td nowrap class="$rowstyle" width="23">
      <div class="$rowstyle" align="center">
        <a href='$claimLink'$entry></a></div>      |#3
      </td>
    #else
    <td nowrap class="$rowstyle" width="23">
      <div class="$rowstyle" align="center">$entry</div>
    </td>
  #end
#end
(annotation) <#1 for each row, iterator over the columns>
(annotation) <#2 create an action URL for the Service column>
(annotation) <#3 display the content of the column>
```

The end result is a database browser over the CLAIMS table as shown in Figure 13.9:

Database Browser		
Service	Group	Status
Diagnostic Radiology	MedPlan 1000	PROCESSED
Emergency Op Room Charges	MedPlan 1000	PROCESSED
Other Medical Services	CompPlan 2000	** OPEN **
Prescriptions	MedPlan 1000	PROCESSED
Hospital Misc	MedPlan 1000	PROCESSED
Lab	CompPlan 2000	** HOLD **
Surgery	CompPlan 2000	PROCESSED
Medical Visit	MedPlan 1000	PROCESSED
Room and Board	MedPlan 1000	** OPEN **
0 of 9 Go Refresh		

Figure 13.9 The Claims Browser portlet in view mode

In figure 13.9, we show only the default navigational controls:

3. Go to Record
4. Refresh

Next, let's add some additional navigational controls to your database browser.

Adding navigation controls

The database browser supports the following controls:

- Refresh
- Go to Beginning of Result Set (>>)
- Go to End of Result Set (<<)
- Previous Page (<)
- Next Page (>)
- Go to Record #

<<	<	8	of 16	Go	>	>>	Refresh

Figure 13.10 Database Browser controls

Here is the Velocity template snippet required to do add the controls:

Listing 13.20 Rendering the Database Browser Controls

```
<table width="200" border="0" cellpadding="0" cellspacing="0" align="center">
  <tr>
    <td>
      <div align="center">
        <form action="$renderResponse.createActionURL()" method="post"> |#1
          <input type='hidden' name='db.browser.action' value='first' />
          <input class="jetdbButton" type="submit" value="<<" />
          <input type="hidden" name="start" value="0" />
        </form>
      </div>
    </td>
  </tr>
</table>
```

```

</td>
<td valign="middle" height="30">
  <div align="center">
    <form action="$renderResponse.createActionURL()" method="post"> | #2
      <input type='hidden' name='db.browser.action' value='prev' />
      <input class="jetdbButton" type="submit" value="<">
      <input type="hidden" name="start" value="$prev">
    </form>
  </div>
</td>
#end
#if ($tableSize > 0)
  <form action="$renderResponse.createActionURL()" method="post"> | #3
    <td valign="middle" height="30">
      <div align="center">
        <input type='hidden' name='db.browser.action' value='change' />
        <input type="input" name='start' size='5' value="$start">
      </div>
    </td>
    <td valign="middle" height="30">
      <div align="center">
        <input type="input" readonly size='10' value="of $tableSize">
      </div>
    </td>
    <td valign="middle" height="30">
      <div align="center">
        <input class="jetdbButton" type="submit" value="Go">
      </div>
    </td>

  </form>
#end
#if ($next)
  <td valign="middle">
    <div align="center">
      <form action="$renderResponse.createActionURL()" method="post"> | #4
        <input type='hidden' name='db.browser.action' value='next' />
        <input class="jetdbButton" type="submit" value=">">
        <input type="hidden" name="start" value="$next">
      </form>
    </div>
  </td>
  <td valign="middle" height="30">
    <div align="center">
      <form action="$renderResponse.createActionURL()" method="post"> | #5
        <input type='hidden' name='db.browser.action' value='last' />
        <input class="jetdbButton" type="submit" value=">>">
        <input type="hidden" name="start" value="$tableSize">
      </form>
    </div>
  </td>
#end
#if ($tableSize > 0)
  <td valign="middle">
    <div align="center">

```

```

        <form action="$renderResponse.createActionURL()" method="post"> | #6
            <input type='hidden' name='db.browser.action' value='refresh' />
            <input class="jetddbButton" type="submit" name="eventSubmit_doRefresh"
value="$MESSAGES.getString('dbrefresh')" />
        </form>
    </div>
</td>
#end
</tr>
</table>
(Annotation) <#1 go to First record>
(Annotation) <#2 go to Prev record>
(Annotation) <#3 go to Nth record>
(Annotation) <#4 go to Next record>
(Annotation) <#5 go to Last record>
(Annotation) <#6 Refresh>

```

#1 An action URL is created for each database navigation action. It takes no additional code in your portlet to handle these built in actions. However if you override the *processAction* method of your portlet, make sure to pass control to the *DatabaseBrowserPortlet* after handling your action processing:

```
super.processAction(actionRequest, actionResponse);
```

Browsing other data

The base class of the *DatabaseBrowserPortlet* is the *BrowserPortlet* class. The *BrowserPortlet* uses the same concepts as the *DatabaseBrowserPortlet*, however it is not limited to relational databases. By overriding the *getRows* method, derived implementations can easily browse over other kinds of data.

For example, the User popups as in Figure 13.11 in the Jetspeed-2 security management portlets use a *BrowserPortlet* combined with a *UserManager* API:



Figure 13.11 The User Chooser portlet is a Browser portlet

And here is the entire source to the UserChooserPortlet:

Listing 13.21 The UserChooserPortlet implementation

```

public class UserChooserPortlet extends BrowserPortlet
{
    private UserManager userManager;

    public void init(PortletConfig config)

```



```

throws PortletException
{
    super.init(config);
    userManager = (UserManager)    |#1
        getPortletContext().getAttribute(
            SecurityResources.CPS_USER_MANAGER_COMPONENT);
    if (null == userManager)
    {
        throw new PortletException(
            "Failed to find the User Manager on portlet initialization");
    }
}

public void getRows(RenderRequest request, String sql, int windowSize)
throws Exception
{
    List resultSetTitleList = new ArrayList();
    List resultSetTypeList = new ArrayList();
    try
    {
        Iterator users = userManager getUsers("");    |#2

        resultSetTypeList.add(String.valueOf(Types.VARCHAR));
        resultSetTitleList.add("User");

        List list = new ArrayList();
        while (users.hasNext())
        {
            User user = (User)users.next();
            Principal principal = getPrincipal(user.getSubject(),
                UserPrincipal.class);    |#3
            list.add(principal.getName());
        }
        BrowserIterator iterator = new DatabaseBrowserIterator(
            list, resultSetTitleList, resultSetTypeList,
            windowSize);
        setBrowserIterator(request, iterator);
        iterator.sort("User");    |#4
    }
    catch (Exception e)
    {
        //log.error("Exception in CMSBrowserAction.getRows: ", e);
        e.printStackTrace();
        throw e;
    }
}

```

(Annotation) <#1 acquire user manager service on portlet init>

(Annotation) <#2 getUsers API retrieve all users in portal>

(Annotation) <#3 build the list of users as JAAS principals>

(Annotation) <#4 sort the final user list>

As you can see in figure 13.11, we have quickly created a portlet for browsing over the users in the portal. Now that we have seen how to program a database browser portlet, let's look at how you can make the database browser portlet communicate with another portlet: the Claims Detail portlet.

Using inter-portlet communication: the claims browser and claims detail portlets

The Claims Browser portlet is a great way for an employee to review their claims. However, what if the employee needed to see more claims details? Or if the employee needed to update or interact with the claims record? In this section, we will show you how to solve these problems. By combining two portlets on one portal page, we give the employee the ability to both browse their claims, and update their claims detail status.

This is achieved with a basic inter-portlet communication solution based on action URLs and some helper classes from Portals Bridges. Note that this entire solution is portable to any Portlet API-compliant portal server.

First, let's look at how we send the message. This occurs when someone clicks on the service column in the Claims browser:

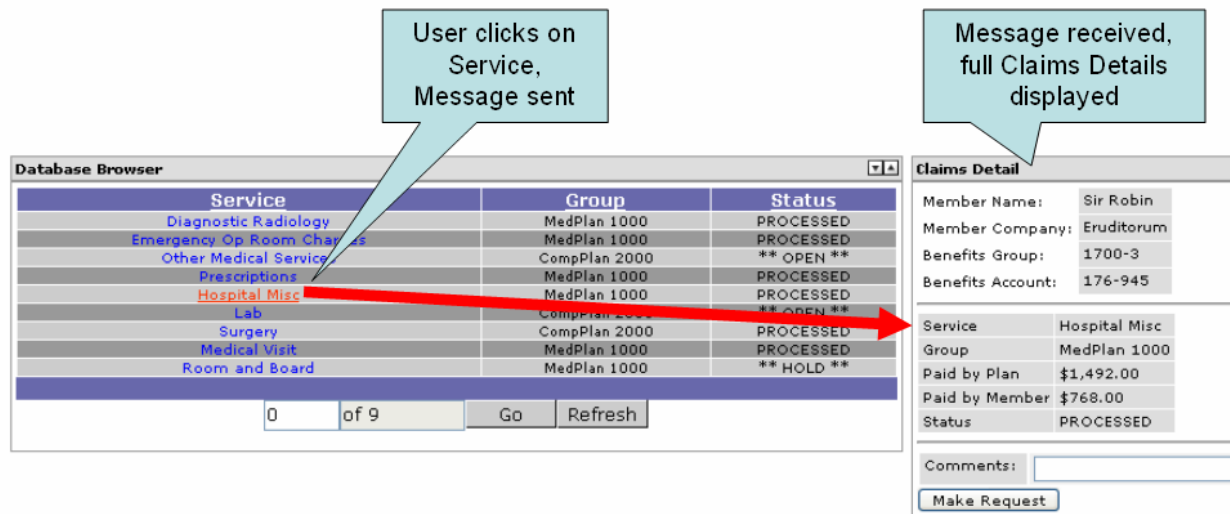


Figure 13.12 Inter-portlet communication: the claims browser sends a message to the claims detail

The service name is sent in a message to the Claims Detail portlet during the Claims Browser's action phase. The Claims Detail portlet receives the message, and knows to display the details for the selected claim.

Version 1.0 of the Portlet API does not address inter-portlet communication. For this book, we have made use of a fairly abstract implementation based on using the portlet application's session state. Since application scoped session state can be easily shared amongst portlets, we have provided a simple abstraction over the session API. The remainder of this section will demonstrate how we implemented our inter-portlet communication.

Sending a message

For the Service column of the database browser, we always create a URL around the column label "Service". The link is created using the *actionResponse* object (provided by the *DatabaseBrowserPortlet*'s base class, *GenericVelocityPortlet*). This URL is an action URL. The action URL has a parameter named *claim*.

```
#elseif ($velocityCount == 2)
#set ($claimLink = $renderResponse.createActionURL())
$claimLink.setParameter("claim", $rowid.toString())
```

```

<td nowrap class="$rowstyle" width="23">
    <div class="$rowstyle" align="center">
        <a href='$claimLink'$entry</a></div>
    </td>

```

The unique claim ID comes from the first column of the query:

```

#if ($velocityCount == 1)
    #set ($rowid = $entry)

```

The claim ID then passed as a parameter with the action URL:

```

$claimLink.setParameter("claim", $rowid.toString())

```

This parameter is picked up in our *ClaimBrowserPortlet*'s *processAction* method:

```

String claim = actionRequest.getParameter("claim");

// publish change to ClaimsDetail Portlet
if (claim != null)
    PortletMessaging.publish(actionRequest,
        "claims", "selected", claim);

```

We then make use of a helper class provided with Portals Bridges. The *PortletMessaging* class facilitates the passing of messaging between portlets within the same portlet application (Note that the default implementation will not support cross web-context messaging). The other message parameters include:

- The second parameter, “claims”, is a general message topic.
- The third parameter, “selected”, is the message name.
- The fourth parameter is the value of the message.

The next version of the Portlet API will support portlet messaging. The code that we have shown you in this section works as an abstraction towards the anticipated API change, minimizing the changes required to our application when the API is public.

Next, we will see how the *ClaimsDetailPortlet* picks up this message.

Receiving the message

The *ClaimsDetailPortlet* displays the full details of a Claims record. It also has the ability to update the comments field on the Claims record.

The Claims Detail portlet expects a primary key to be made available by another portlet. Looking at the *doView* method in Listing 13.22, we see that the portlet first retrieves user attributes(1). We discussed User Attributes in section 13.3. .

Listing 13.22 The doView method of the ClaimsDetailPortlet

```

public void doView(RenderRequest request, RenderResponse response)
    throws PortletException, IOException
{
    Map userInfo =      |#1
        (Map) request.getAttribute(PortletRequest.USER_INFO);

    Context context = super.getContext(request);
    String firstName = (String)userInfo.get("user.name.given");

```

```

context.put("firstName",      |#2
            userInfo.get("user.name.given"));
context.put("lastName", userInfo.get("user.name.family"));
context.put("company", userInfo.get("user.employer"));
context.put("group", userInfo.get("benefits.group"));
context.put("account", userInfo.get("benefits.account"));

String claimId =      |#3
            (String)PortletMessaging.consume(request, "claims", "selected");
Claim claim =      |#4
            (Claim)request.getPortletSession().getAttribute("claim");
if (claimId != null)
{
    claim = lookupClaimStatus(claimId);      |#5
    request.getPortletSession().setAttribute("claim", claim);
}

context.put("claim", claim);      |#6
super.doView(request, response);
}

```

(Annotation) <#1 get USER_INFO from Portlet API>

(Annotation) <#2 get required user attributes, then put them in Velocity context>

(Annotation) <#3 receive a message from Claims browser upon selection>

(Annotation) <#4 get the claim by ID from the portlet session>

(Annotation) <#5 if a new claim is selected, retrieve from database>

(Annotation) <#6 put the claim into the view for rendering in velocity template

Getting the claim selected message

Next, the portlet receives a message from the database browser portlet, informing the detail portlet that a row was clicked on in the browser:

```

String claimId = (String)PortletMessaging.consume(request, "claims",
"selected");
Claim claim = (Claim)request.getPortletSession().getAttribute("claim");
if (claimId != null)
{
    claim = lookupClaimStatus(claimId);
    request.getPortletSession().setAttribute("claim", claim);
}

```

Note the message is consumed, meaning it is removed from the message queue since the lifetime of a “claim selected” message spans only one portlet render cycle.

Looking up the correct claim status record

When a claim message is received, the portlet looks up a Claim Status record, using the primary key, *claimId*, passed in the message. The claim record is put into the Velocity context:

```
context.put("claim", claim);
```

The claim detail is rendered in the Velocity template:

Listing 13.23 The velocity template for the claims detail

```
<table border="1" cellspacing="1" cellpadding="3">
  <tr>
    <td class='portlet-section-alternate'>Service</td>
    <td class='portlet-section-alternate'>${!claim.Service}</td>
  </tr>
  <tr>
    <td class='portlet-section-alternate'>Group</td>
    <td class='portlet-section-alternate'>${!claim.Group}</td>
  </tr>
  <tr>
    <td class='portlet-section-alternate'>Paid by Plan</td>
    <td class='portlet-section-alternate'>${!claim.PlanPaid}</td>
  </tr>
  <tr>
    <td class='portlet-section-alternate'>Paid by Member</td>
    <td class='portlet-section-alternate'>
      ${!claim.MemberPaid}
    </td>
  </tr>
  <tr>
    <td class='portlet-section-alternate'>Status</td>
    <td class='portlet-section-alternate'>${!claim.Status}</td>
  </tr>
</table>
```

The final result is the rendered Claim Detail portlet (Figure 13.13):

Claims Detail	
Member Name:	Chip Sparrow
Member Company:	Eruditorum
Benefits Group:	1700-3
Benefits Account:	176-946
<hr/>	
Service	Diagnostic Radiology
Group	MedPlan 1000
Paid by Plan	\$111.00
Paid by Member	\$88.00
Status	PROCESSED
<hr/>	
Comments:	<input type="text" value="I need to get a more detailed lab report from the provider"/>
<input type="button" value="Make Request"/>	

Figure 13.13 The Claims Detail portlet in view mode

This concludes our coverage of inter-portlet communication. Next, let's look at how we can use the Claims Detail portlet to update a database record.

Updating the Comments field

The portlet also provides an example of updating the Comments field. An action URL is created on the HTML form's action element. The action URL directs the posted data back to our portlets *processAction* method.

Listing 13.24 Creating an action URL to post back to the portlet

```
<form name='documentForm' action="$renderResponse.createActionURL()" method="post">
<table>
  <tr colspan="2" align="right">
    <td nowrap class="portlet-section-alternate" align="right">Comments:&nbsp;  </td>
    <td class="portlet-section-body" align="left">
      <input type="text" name="comments" size="80" value="$!claim.Comments"
class="portlet-form-field-label">
    </td>
  </tr>
</table>
#if ($claim)
<input name='claimId' type='hidden' value='$!claim.ClaimId' />
<input name='save' type="submit" value="Make Request" class="portlet-form-button" />
#end
</form>
```

The *processAction* method looks for a hidden *claimId* parameter.

Listing 13.25 The processAction method for the ClaimsDetail portlet

```
public void processAction(ActionRequest actionRequest,
    ActionResponse actionResponse) throws PortletException, IOException
{
    String claimId = actionRequest.getParameter("claimId");
    String comments = actionRequest.getParameter("comments");
    if(claimId != null)
    {
        Claim claim = updateClaimComment(claimId, comments);
        actionRequest.getPortletSession().setAttribute("claim", claim);
        PortletMessaging.publish(actionRequest, "claims", "selected", claimId);
    }
}
```

When the hidden parameter is present, we attempt to update the database in the *updateClaimComment* method:

Listing 13.26 The updateClaimComment method

```
private Claim updateClaimComment(String claimId, String comments)
{
    Connection con = null;
    try
    {
        con = cms.getConnection();
        String sql = "UPDATE WWCLAIMS SET COMMENTS = ? WHERE CLAIM_ID = ?";
        PreparedStatement stm = con.prepareStatement(sql);
        stm.setString(1, comments);
        int id = Integer.parseInt(claimId);
        stm.setInt(2, id);
        int count = stm.executeUpdate();
        return lookupClaimStatus(claimId);
    }
    catch (SQLException e)
    {
        System.err.println("Exception updating claim : " + e);
    }
}
```

```

    }
    finally
    {
        releaseConnection(con);
    }
    return new Claim(0, "", "", 0, 0, "", "");
}

```

Claim is an inner class stored in our portlet:

Listing 13.27 The Claim inner class

```

    private final NumberFormat formatter = new DecimalFormat("$###,###.00");

    public Claim(int claimId, String service, String group, double planPaid,
double memberPaid,
                    String status, String comments)
    {
        this.claimId = claimId;
        this.service = service;
        this.group = group;
        this.planPaid = planPaid;
        this.memberPaid = memberPaid;
        this.status = status;
        this.comments = comments;
    }

    public String getPlanPaid()
    {
        return formatter.format(planPaid);
    }

    public String getMemberPaid()
    {
        return formatter.format(memberPaid);
    }

    /**
     * @return Returns the comments.
     */
    public String getComments()
    {
        return comments;
    }

    /**
     * @return Returns the group.
     */
    public String getGroup()
    {
        return group;
    }

    /**
     * @return Returns the service.
     */
    public String getService()

```

```

    {
        return service;
    }
    /**
     * @return Returns the status.
     */
    public String getStatus()
    {
        return status;
    }
    /**
     * @return Returns the claimId.
     */
    public int getClaimId()
    {
        return claimId;
    }
}

```

Portlet CSS style definitions

You may have noticed that some CSS style classes have names like *portlet-section-alternate*. The Portlet API recommends a common CSS style sheet naming convention for links, fonts, messages, sections and forms. See the Portlet API specification appendix, PLT.C for a complete list of recommended styles. If you would like your portlets to have the same overall style as other portlets in the portal, we recommend using these styles whenever possible in your portlet markup.

```
<td nowrap class="portlet-section-alternate" align="right">Comments:&nbsp;  </td>
```

Summary

In this chapter we learned about accessing data in backend systems and integrating that data in portlets using standard methods of data access. We showed you how to integrate portlets with relational databases, Portlet API user preferences, and the Jetspeed database browser portlet. We demonstrated how to use the Model View Controller design pattern in portlets. Additionally, we learned how to develop two portlets that work together, using a simple but effective and portable inter-portlet communication method.

In the process, we developed four portlets for the Waxwing Benefits portal:

5. Client Status portlet
6. Deductible Status portlet
7. Claims Browser portlet
8. Claims Detail portlet

Now that we have added these portlets to our Waxwing Benefit portal let's take a look at some advanced portlet features in the next chapter. We will integrate the Apache Graffito content management system, RSS portlets, the Apache Lucene search engine and access data via a web service.

Appendixes

In the appendixes, we will cover some additional information that may be helpful when developing portlets, but is not essential for the portlet development process itself.

In Appendix A, we will take a look at the portal market, which should give you some ideas on the areas that will create demand for portlets and the major platforms that your portlets will likely end up running upon.

In Appendix B, we will provide an introduction into the basics of web services that is the base technology for Chapter 9 – Web Services for Remote Portlets.

In Appendix C, we cover additional technologies that enable the portlet and portal technology, like servlets, JSPs, and other Java standards. Finally, we discuss how portlets benefit from web services technologies.

Appendix Portal market

The portal market started just a few years ago and today already the fourth generation of portals exist. Portals have already undergone the change from a standalone product towards a middleware component that is integrated in other products.

Portals serve a wide range of strategic and tactical business objectives. Portals integrate independent standalone applications into one consistent and connected set of tools. Businesses can leverage the new capabilities that portals provide to organize processes and daily work tasks. Portals also provide a platform for interacting with other people. Portals change how people interact with each other, handle information, and are integrated into processes.

In this section we will first give an overview of the current portal market, their major players, both commercial and open source and finally describe where the market is currently heading.

A.1 Current situation

Analysts say the portal market is still growing even as the IT economy in general is slowing down. Many analysts predict that the enterprise portal market will grow at around 20% until at least 2007. The current market for enterprise portals is estimated around one billion dollars for 2004.

Currently the portlet market is very fragmented, with more than 20 companies having a significant market share. Forecasts predict this will change and the portal market will become a mature market dominated by two or three companies that will make up around 80% of the market. Similar market mechanisms could be observed in the application server market; in its initial phase a variety of vendors competed, and finally it came down to BEA and IBM having more than 70% market share.

A.2 Major players in the portal market

Currently the portal market is in its early stage and thus quite fragmented. There are more than 20 portal vendors with significant portal revenue listed by market studies. The clear leader in these studies is IBM with WebSphere Portal, followed by SAP. Also of interest are BEA and Microsoft, as their portals had the largest growth rate besides the IBM portal. This section will highlight these four commercial leaders and some of the available open source portals.

Commercial portals

This book cannot really provide an overview for all commercial portal products. We will briefly present the four leading commercial portal products and describe the support for standards in the current versions of these products. .

BEA WebLogic Portal 8.1

The BEA WebLogic Portal is based on BEA's WebLogic application server and is available for Windows- and Unix-based platforms. It also ships with the BEA WebLogic Workshop Portal Extensions that allows you to visually create and design portal pages. BEA WebLogic Portal is Java-based and supports JSR 168 and JSPs. The BEA WebLogic portal includes all the major components needed for the enterprise market: content management, search, collaboration, and commerce capabilities. It also supports basic workflow capabilities to model these workflows inside the portal. The BEA WebLogic portal also supports WSRP producers and consumers. The BEA WebLogic portal is aimed at enterprise customers with medium to large portal deployments. For more information on the BEA WebLogic portal, see [1].

IBM WebSphere Portal 5.0.2.2

The IBM WebSphere Portal is based on IBM's WebSphere application server and is available on a wide range of platforms, from Windows-based platforms, Unix-based ones, up to the z/OS platform. WebSphere Portal is currently the market leader in the portal space and has the broadest set of functions, ranging from content management, search, extended collaboration features, work flow support, analysis tools, and translators. The extended collaboration capabilities let you create teamrooms for sharing data between teams, or add people awareness for instant messaging to portlets. The IBM WebSphere portal also supports JSR 168 and WSRP producers and consumers. There are also additions available that provide support for pervasive devices or accessing the portal via voice. The IBM WebSphere portal is available in different bundles to address all enterprise portal markets, from small to large portal deployments. For more information on the IBM WebSphere Portal see [2].

Microsoft SharePoint Portal Server 2003

Microsoft started late into the portal market, but with the 2003 version, Microsoft is catching up on established portal servers like BEA and IBM. The SharePoint Portal Server is only available for Windows-based platforms. The SharePoint Portal is focused on collaboration and integration of Microsoft office products into the portal; enterprise portal features, like workflow support or web site analysis tools, are currently not part of the product. SharePoint Portal does not currently support JSR 168, but it does support WSRP. The Microsoft portal targets the consumer market and small enterprise portal deployments. For more information on the Microsoft SharePoint Portal see [3].

SAP Enterprise Portal 5.0

The SAP Enterprise Portal is based on the mySAP infrastructure, including SAP's application server and is only available on Windows-based platforms. The SAP portal is focused on offering content from SAP systems via an enterprise portal and other enterprise portal functions, like workflow or collaboration are only available with very restricted capabilities. Enterprise Portal does not currently support JSR 168 nor WSRP, but we assume that both standards will be supported in later versions. For more information on the SAP Enterprise Portal see [4].

Open source portals

There are quite a lot of open source portals available right now. In this section we will only cover a part of this large group that are either very popular or have some interesting technology. For a more complete list of open source portals see [5].

Cocoon Portal

The Cocoon Portal is built on top of the Cocoon XML-based web development framework. Cocoon is an Apache project and can run on top of several servlet containers or application servers like Apache Tomcat, BEA WebLogic, IBM WebSphere, or JBoss Application Server. It currently only allows integrating XML-based content, but plans to support JSR 168 in a future version. More information about the Cocoon Portal can be found at [6].

eXo Portal 1.0

The eXo Portal is one of the first portals to use Java Server Faces as its aggregation engine. It contains content management and workflow capabilities and supports JSR 168 and WSRP. It comes in two flavors, as Express edition for small and medium companies and as Enterprise edition. More information on the eXo Portal can be found at [7].

JA-SIG uPortal 2.3

The JA-SIG uPortal comes with the Apache Tomcat servlet container, but can also run on top of full-fledged application servers, like BEA WebLogic, IBM WebSphere, or JBoss Application Server. The uPortal consist of basic support for content management, collaboration and workflow capabilities and supports JSR 168. More information on the JA-SIG uPortal can be found at [8].

Jetspeed-2

Jetspeed-2 was chosen as the open source example portal for this book and is explained in detail in chapter 8.

Liferay Enterprise Portal 2.0

The Liferay Enterprise Portal runs on top of all major application servers, like BEA WebLogic, IBM WebSphere, and JBoss Application Server. The Liferay portal is not comparable with the previously mentioned commercial portals, as it lacks all the enterprise features, like content management, search, collaboration or workflow support. However, Liferay supports JSR 168 and comes with some default portlets. Liferay is thus more suitable for portlet development or the consumer market. For more information on the Liferay Enterprise Portal see [9].

Pluto

Pluto is the JSR 168 reference implementation and provides, besides the portlet container, a simple portal driver to test out JSR 168 portlets. Pluto uses the Apache Tomcat server as underlying servlet container and is also an Apache project. More information about Pluto can be found in chapter 10.

Now that we have covered today's portal market and products available, we try to take a look at where the portal market will be heading in the future.

A.3 Where is the portal market heading?

Portal market studies from research companies like Gartner, Giga, or IDC predict that the portal market will grow faster than the average software market. We will not try to predict the future in this chapter, nor cover every aspect of these different studies. Instead we will focus on what we think are some of the key areas of the portal market growth in the next years: e-learning, process portals, web services, and micro portals.

e-learning

Market demand for portal and e-learning technologies is increasing and moving towards using portals and Web-based learning technology to directly link learning with the work issues that students face. Integration of portals into e-learning will lead to fully integrated learning where job activity and learning become one. Portals also make it easy for students to access and customize their individual learning experiences. Via corporate portals learning will also be integrated into the daily process of learning and on-the-job training, allowing people to easily and quickly access the information needed for a particular task.

Advances in e-learning will lead to a more task-specific learning pattern where students or employees have just-in-time access to the learning content they need in their current situation. Learning can now progress with the speed of the individual student and not with a fixed course layout.

Process portals

Process portals are about integrating business process monitors and portals into one offering. *Business process monitor* is a general term, describing a set of services and tools that provide for explicit process management, including process analysis, definition, execution, monitoring and administration with support for human and application-level interaction.

Process portals also include an integration broker suite that allows sending events and alarms to registered users based on the outcome of a process step.
(processPortal.tif)

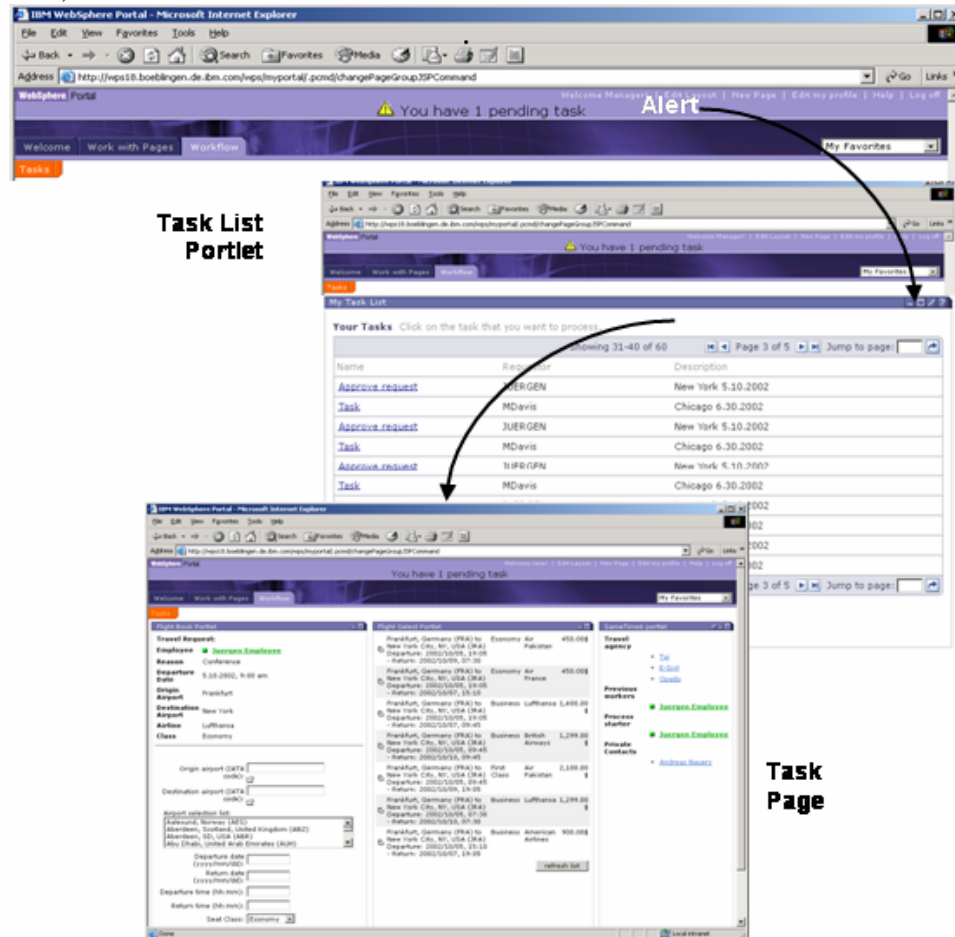


Figure A.1: This process portal alerts users of their pending tasks.

Figure A.1 depicts an example of a process portal where a user gets an alert that she has a pending task. When the user clicks on that alert she will be taken to a portlet that lists all current tasks. The user can select a task from the list and get a task page, with portlets displaying all information she needs to perform the task. In this example a flight for an employee needs to be booked. After this task step is finished, the process engine determines the next step in this process (e.g., flight ticket cost needs to be approved) and sends an alert to the person responsible for this task step.

In order to allow specifying the business process in standard manner future portals will support the Business Process Execution Language (BPEL) for web services that allows declaring business processes as web services. The current version of this standard can be found at OASIS (see [10]).

Tying the portal infrastructure increasingly with the business processes in a company will lead towards portals getting application platform suites. This means that portals will come as complete application suites including application servers, integration brokers, portals and other technologies, such as wireless, voices, personalization, caching, integrating development framework, and integrated systems management.

Web services

Web services will become more popular in the next few years and will open new areas for deploying portals. Web services allow businesses to easily distribute a system, providing their content and services that can be integrated into portals and served to the end user with a consistent user interface. Besides the

previously mentioned BPEL standard, Web Services for Remote Portlets (WSRP) will become very important in offering services and content as remote portlets. The whole web services technology is explained in Chapter 2 and WSRP is covered in detail in Chapter 9.

Micro portals

Micro portals are very small portals that are only used by one person or by a small group. They will enable the users to have an improved user experience, as many of the user interface elements can run on the client, with vastly reduced latency times as the network does not need to be consulted for every user action. In addition, micro portals will allow users to work offline with their portal and synchronize offline changes back to the server when they log back in.

Examples for these kinds of micro portals include the Longhorn vision from Microsoft that will make the operating system more portal-like, and the Workplace client based on Eclipse from IBM. More about micro portals can be found in Chapter 2.

A.4 References

- [1] BEA WebLogic Portal
<http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/portal>
- [2] IBM WebSphere Portal
http://www-306.ibm.com/software/info1/websphere/index.jsp?tab=products/portal&S_TACT=103BGW01&S_CMP=campaign
- [3] Microsoft SharePoint Portal
<http://office.microsoft.com/home/office.aspx?assetid=FX010909721033>
- [4] SAP Enterprise Portal
<http://www.sap.com/solutions/netweaver/enterpriseportal/index.asp>
- [5] List of available open source portals
http://www.manageability.org/blog/stuff/open_source_portal_servers_in_java/view
- [6] Cocoon Portal
<http://cocoon.apache.org/2.1/developing/webapps/portal.html>
- [7] eXo Portal
<http://www.exoplatform.org/>
- [8] JA-SIG uPortal
<http://www.uportal.org/>
- [9] Liferay Enterprise Portal
<http://www.liferay.com/home/index.jsp>
- [10] Web Services Business Process Execution Language (BPEL)
http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel

Appendix B. Introduction to Web services

This section offers a short introduction into web services to better understand the underpinnings of Web Services for Remote Portlets explained in chapter 9.

B.1 The basis for web services: XML documents

Probably the most important building block for web services is the XML standard. The Extensible Markup Language has its roots in SGML (Standard Generalized Markup Language defined by ISO 8879). SGML has been around for more than a decade and has been used in complex systems to maintain repositories of structured information. But it proved to be too complex to be widely accepted and seemed not well suited to exchange data over the web.

XML was developed as a simple dialect of SGML, lowering the hurdle for implementers and users. Basically XML is a language to describe data; it allows us to develop new document formats exchanged over the web in an easy way, similar to HTML. But XML is not just a HTML replacement. The goals for both languages are different. While HTML focuses on how data is displayed and how it looks, XML was designed to describe what the data actually is. Unlike HTML, XML has no predefined set of tags. Instead vocabularies can be defined that meet specific application or business needs. It is a meta-language; a language for defining other languages. XML is standardized at the World Wide Web Consortium (W3C).

The key point of XML is its language-, platform- and vendor-neutrality. An XML document generated by a Java application running on Linux is still an XML document for a C++ programmer using Microsoft Windows. This allows an application-neutral exchange of data. All applications need to do is develop a single tool to transform their internal data into XML and vice versa. Using a common representation of data eases interoperability and data exchange. And the good news is that commercial vendors and the open source community have already produced a bunch of tools that accomplish these tasks for you.

Assume a used car seller offering its cars over the Internet. The company uses XML to exchange the data about the cars available in stock. Listing B.1 shows a sample XML document used to describe the necessary data. Before we start to explain some basics in the next section, try to read and understand the document. We're pretty sure you captured the meaning and purpose of this document. There is nothing special about XML. It is plain text containing both, the data itself and information about the data structure.

Listing B.1 XML sample document containing cars available in stock

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This is the list of cars available in stock -->
<Cars>
  <Car ID="12345">
    <Price currency="USD">19000.00</Price>
    <Make>Audi</Make>
    <Model>TT</Model>
    <Color>Silver</Color>
    <ManufactureDate>2001-01</ManufactureDate>
    <Mileage>30000</Mileage>
    <BodyType>Coupe</BodyType>
    <Engine>
      <CubicCapacity unit="ccm">1781</CubicCapacity>
      <Cylinders>4</Cylinders>
      <Power unit="kW">165</Power>
      <FuelType>gasoline</FuelType>
    </Engine>
  </Car>
</Cars>
```



```

</Car>
<Car ID="98765">
  <Price currency="USD">29000.00</Price>
  <Make>Nissan</Make>
  <Model>350Z</Model>
  <Color>Blue</Color>
  <ManufactureDate>2004-02</ManufactureDate>
  <Mileage>1500</Mileage>
  <BodyType>Coupe</BodyType>
  <Engine>
    <CubicCapacity unit="ccm">3498</CubicCapacity>
    <Cylinders>6</Cylinders>
    <Power unit="PS">287</Power>
    <FuelType>gasoline</FuelType>
  </Engine>
</Car>
</Cars>

```

B.2 Making XML documents work together with Namespaces

The strength of XML is its flexibility. It allows us to define our own tags and create exchangeable documents suitable for our business needs. But there is one shortcoming in the XML specification, which made developers feel that the specification was incomplete. It lacked namespace support.

Why are namespaces so important? If everybody can invent tags and structures to describe data it is likely that name clashes occur. These name clashes can create ambiguity.

Consider the following two example document fragments describing a person. The first one might represent a person in an insurance database, while the second one might be describing an employee. Although the child elements of both `<Person>` elements are named the same, they carry different semantics. For example the `<ID>` element might be the social security number in the first case, while in the second case it's the employee number within a company. Also the `<Title>` element holds the courtesy title of a person in the first case, while in the second case it's the job title.

Listing B.2 XML sample showing name clashes

```

<Person>
  <ID>123-456-789</ID>
  <Title>Mr.</Title>
  <Name>John Doe</Name>
</Person>

<Person>
  <ID>123456789</ID>
  <Title>Software Developer</Title>
  <Name>John Doe</Name>
</Person>

```

While human beings probably could interpret the data correctly, machines cannot. They can't tell whether the `<Person>` element describes a client or an employee. This is where namespaces come into play. Namespaces remove ambiguity by creating domains in which all names are unique. Partitioning the global namespace into smaller domains makes it easier to come up with unique names. One has to care about uniqueness only within a limited context.

When dealing with namespaces, two things need to be considered. First, a namespace authority needs to make sure that a newly added name doesn't already exist in the namespace. If this is not ensured, then a

newly added name can corrupt the namespace, making entries indistinct. Per definition, a namespace must ensure the uniqueness of elements within it; otherwise it is no longer considered a namespace.

Second, the namespace itself must be given a unique name. Only the combination of namespace name and local name allows us to refer to a member within a namespace. For example Richard Jacob's jersey number playing for the Stuttgart Reds baseball team is 66. The namespace for that team could be "StuttgartReds". To refer to the player Richard Jacob, we could use the concatenation of namespace and local name like this: "StuttgartReds.66". In this way we can distinguish this player from the player "FreiburgKnights.66". Note that that player has the same local name, i.e. jersey number.

We assumed here that the namespace itself is unique. In order to ensure that the namespace name is unique, the name itself could be placed into another namespace. In our example, there could be another baseball team called "Stuttgart Reds" somewhere else in the world besides Germany. To make these distinguishable, we could place the team's namespace name into another namespace called "DBV" which is the German Baseball Association. This way we could easily identify the player Richard Jacob playing for the German team Stuttgart Reds: "DBV.StuttgartReds.66". We could continue this pattern until we build up a real unique namespace name.

This is exactly the way the Domain Name System (DNS) works. It partitions the global namespace into smaller namespaces which, in turn, partition themselves into further namespaces down to hostnames being unique within a domain. We can observe a similar pattern in Java class naming using package names to uniquely identify a class.

The definition of elements, attributes and types within a certain namespace is made within XML Schema. We will introduce XML Schema in a later section. Here you only need to understand that XML Schema provides a means to syntactically define a namespace and its members. The example below defines the namespace "http://www.manning.com/books/portal/employee" containing the elements Person, ID, Title and Name.

Listing B.3 XML namespace definition for a person

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
        targetNamespace="http://www.manning.com/books/portal/employee"
        elementFormDefault="qualified">

  <element name="Person">
    <complexType>
      <element name="ID" type="string"/>
      <element name="Title" type="string"/>
      <element name="Name" type="string"/>
    </complexType>
  </element>
</schema>
```

With this definition we can now use the element Person in a fully-qualified manner and thus can unambiguously tell which Person element we intend to use. Fully-qualified means we use the namespace identifier in combination with the local name to address a member within the namespace. Such a name consisting of the two parts is called a qualified name or QName.

In XML documents, namespace prefixes are used to qualify elements or attributes. A prefix is a placeholder for the namespace identifier. The prefix is mapped to a namespace identifier by using a namespace declaration. This declaration looks like an attribute of an element. The scope of the prefix is applicable on the declaration element as well as on any of its child elements. Once declared, the prefix can be used in front of any element or attribute name separated by a colon. In the following example we declare the

prefix “e” and map it to the namespace “http://www.manning.com/books/portal/employee.” We fully qualify the <Person> element and its child elements by using the prefix.

Listing B.4 XML sample using the person definition from listing 2.5

```
<?xml version="1.0" encoding="UTF-8"?>
<e:Person xmlns:e="http://www.manning.com/books/portal/employee">
  <e:ID>123456789</ID>
  <e:Title>Software Developer</Title>
  <e:Name>John Doe</Name>
</e:Person>
```

The usage of namespaces and the possibility to define namespaces and the elements, types, attributes within them enables us to share semantics. An XML document can utilize already-defined types and reuse them without reinventing the wheel. Indeed it is quite common to have multiple namespace declarations on one element and reuse already defined elements from various domains, i.e., namespaces.

Now that we have already stumbled over XML Schemas, let’s take a closer look on what XML Schemas are.

B.3 Getting types into the XML documents: XML Schemas

When processing XML documents, we generally process plain text files. But how can we tell that the data and structure within the document really meets the requirements and expectations we have when exchanging data? Without a type system we can’t. Type systems are used in many areas. For example, we use the relational model and integrity constraints to describe the valid content and data within a database. Having this model description enables us to validate the actual data against our model and to decide whether the data meets our requirements.

XML 1.0 lacked a good type system. The W3C XML 1.0 Recommendation contains the Document Type Definition (DTD) specification. DTDs use a different syntax from XML, which is a drawback to using them. DTDs have been created for many uses, however, including HTML, XHTML, and DocBook (for publishing and documentation).

The best alternative to DTD is the W3C XML Schema Recommendation. XML Schema definition language (XSD) provides a type system for XML environments. It enables us to describe the types that we intend to use in an XML document.

An XML document that conforms to the types described by XML Schema is referred to as an instance document. This is similar to the relationship between objects and classes in object-oriented languages. XML Schema documents are simple XML documents. They use predefined elements and attributes to describe the structure of yet another document, namely the instance documents. Another important fact is that XML Schema defines the types and binds them to a namespace also defined in the schema document.

Let’s take a look at the schema for our used car dealer’s list of cars available in stock. We will explain the various constructs at a later time. Just as our first XML document example, try to read and understand the schema. We’re pretty sure you can pretty well understand it right away.

Listing B.5 XML sample document containing cars available in stock

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.manning.com/CarDesc"
  xmlns:cd="http://www.manning.com/CarDesc">

  <complexType name="CarListType">
```

```

    <sequence>
      <element name="Car" type="cd:CarType" maxOccurs="unbounded"/>
    </sequence>
  </complexType>

  <element name="Cars" type="cd:CarListType"/>

  <complexType name="CarType">
    <sequence>
      <element name="Price" type="string"/>
      <element name="Make" type="string"/>
      <element name="Model" type="string"/>
      <element name="Color" type="string"/>
      <element name="ManufactureDate" type="gYearMonth"/>
      <element name="Mileage" type="unsignedInt"/>
      <element name="BodyType" type="string"/>
      <element name="Engine" type="cd:EngineType"/>
      <any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
    <attribute name="ID" type="string" use="required"/>
  </complexType>

  <complexType name="EngineType">
    <sequence>
      <element name="CubicCapacity" type="unsignedInt"/>
      <element name="Cylinders" type="unsignedInt"/>
      <element name="Power" type="cd:PowerType"/>
      <element name="FuelType" type="string"/>
    </sequence>
  </complexType>

  <complexType name="PowerType">
    <simpleContent>
      <extension base="unsignedInt">
        <attribute name="unit" type="string" use="required"/>
      </extension>
    </simpleContent>
  </complexType>

</schema>

```

The `<schema>` element sets the scope for the elements and types we define in this document to our namespace. The namespace name itself is defined in the `targetNamespace` attribute. Note that we use the default namespace `xmlns="http://www.w3.org/2001/XMLSchema"`. This namespace holds the elements, attributes and built-in types used in schema documents.

B.4 Communication with web services: the SOAP protocol

Now that we have the documents that are exchanged via web services covered in detail, we somehow need to connect to such a service and have a protocol to exchange these wonderful XML documents. The Simple Object Access Protocol (SOAP) provides such a mechanism for exchanging XML documents in a standardized manner between peers in a distributed environment.

SOAP does not define application semantics such as a programming model or implementation-specific semantics. Rather, SOAP defines a simple mechanism for expressing application semantics by providing a modular packaging model and encoding mechanisms for encoding data within modules.

SOAP consists of these parts:

SOAP envelope: The SOAP envelope defines a framework for the definition of messages. It allows specifying what is in a message, who should process it, and whether it is optional or mandatory.

SOAP encoding rules: The encoding rules in SOAP define a serialization mechanism for exchanging instances of application-defined data types.

SOAP Remote Procedure Call (RPC) representation: The RPC representation defines conventions that can be used to represent remote procedure calls and responses as SOAP messages.

Although these parts are described together in the SOAP specification, they are functionally orthogonal. To promote simplicity through modularity, the envelope and the encoding rules are defined in different namespaces. In addition to the SOAP envelope, encoding rules and Remote Procedure Call conventions, SOAP also defines protocol bindings that describe how to carry SOAP messages in HTTP messages.

(soap.tif)

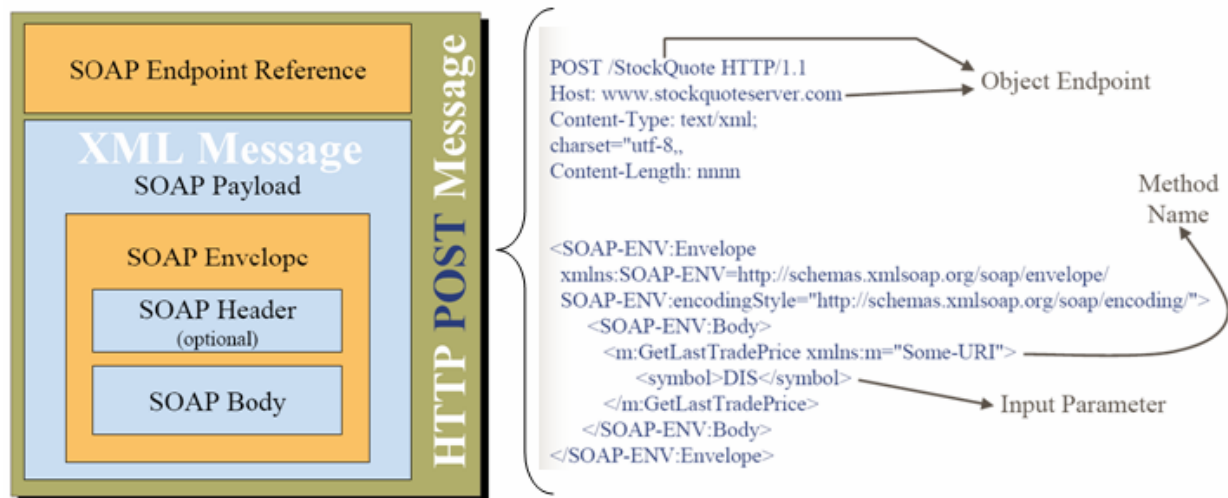


Figure B.1: Construction of a SOAP message for SOAP over HTTP

Figure B.1 shows what this looks like for the SOAP over HTTP example. The SOAP endpoint reference is encoded in the HTTP header and the SOAP envelope is encoded in the HTTP body. This envelope contains the SOAP body with the remote method name and the input parameter for that method.

Now that we have a mechanism that works technology-independent using XML, how does our portlet know which method to call on the service and which input parameter are required by the service? This question is answered in the next section about WSDL.

B.5 Describing web services: WSDL

So now we have a standardized document format and protocol to exchange these documents in a standard manner. What is missing to get web services going? Somehow the consumer of a web service needs to know what kind of service it is and what the input and output parameters are – the kind of information that is available in Java via the method signature and method description. To address this need in a language-independent manner the Web Services Description Language (WSDL [16]) was created. WSDL defines an XML grammar for describing network services as a set of endpoints that are operating on messages that contain either document-oriented or procedure-oriented information.

WSDL describes operations and messages abstractly, and binds them to concrete network protocols and message formats to define ports. It allows combining related ports into services. WSDL uses the following elements in the definition of network services:

- **Types:** A container for data type definitions using a type system (such as XSD).
- **Message:** An abstract, typed definition of the data being communicated.
- **Operation:** An abstract description of an action supported by the service.
- **Port Type:** An abstract set of operations supported by one or more endpoints.
- **Binding:** A concrete protocol and data format specification for a particular port type.
- **Port:** A single endpoint defined as a combination of a binding and a network address.
- **Service:** A collection of related endpoints.

(wsdl.tif)

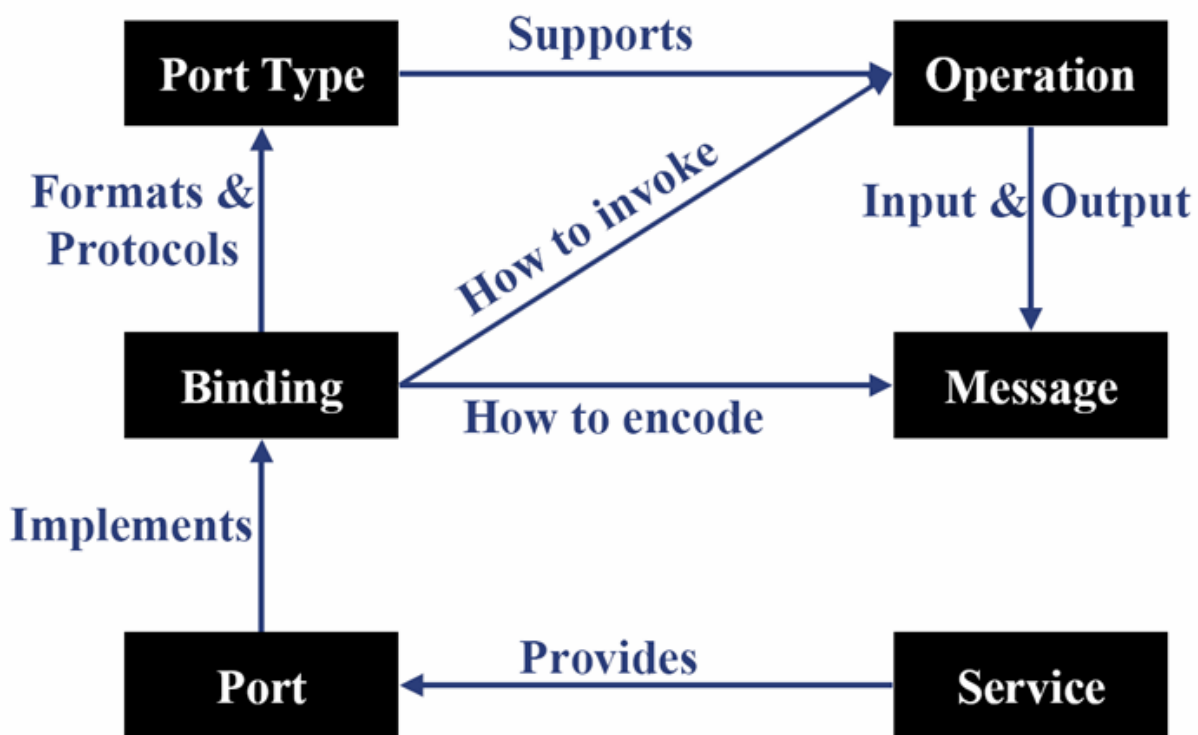


Figure B.2: The different parts that are defined in WSDL and how they relate to each other

Figure B.2 shows how all these definitions relate to each other. A service provides one or more ports by which it can be accessed. These ports implement a specific binding that describes the type of the port, how to invoke the operation on the service and how to encode the message being sent to the service. The port type defines the supported operations on the service and the operation describes the input and output parameters of the message.

Appendix C Enabling technologies

This section covers technologies that enable the portlet programming model and are therefore the basis on which portlet technology are build upon. This section is meant to give you all the basic information about these technologies so that you understand where the portlet programming model came from and where it is heading. These technologies range from the basic web programming model, to different Java standards in that area and finally to the web service and XML world.

To build portlets that are successful in the marketplace it is most important to reach as many customers as possible. In this context this means that portlets must be deployable in lots of environments. Therefore the technology the portlets are based on or use must be available to as many customers as possible. This is only possible if these technologies are standardized and platform independent. In the following section we will discuss some of the most important standards for portal environments. Let's start first with the roots of the portlet programming model: the web programming model with its servlets and JSPs.

C.1 Web programming model: servlets and JSPs

What could be more important for web portals than the common web programming model in the Java world? In this environment servlets and Java Server Pages (JSPs) have become the most widespread and accepted technologies. We assume that you are already familiar with servlets and JSPs, but as portlets build on top of these technologies (tools?) and have some slightly different behavior, we will cover the basic concepts here again and relate them to the portlet programming model. So let's take a look at servlets first.

C.1.1 Java servlet technology

We should clarify why the Java servlet technology was developed in the first place: Back some time ago people realized that the Internet can be used perfectly to provide services to end users all over the world. But to deliver services commonly, interaction is required. This implies that providers had to start to deliver dynamic content. A first attempt achieved this by introducing applets that focused on using the client platform to deliver dynamic user experiences. Common Gateway Interface (CGI) scripts were the main technology used to generate dynamic content on the server side. Although CGI has been widely used, CGI scripting technology has a number of shortcomings. The worst are platform dependence and lack of scalability. As Java is the choice to gain independence of the platform a runtime environment for Java server applications was developed that could better meet the requirements of modern web programming. The Java Servlet technology was created and is still the portable way to provide services and dynamic, user-oriented content. This means that servlets allow people to create complete dynamic web application. Servlets therefore are not tailored towards being a building block that can be integrated with other components, but to act as a specific service in a specific web application. As we will see later on this is one of the main differences in the portlet model. Portlets are designed from the beginning to be components that may be integrated in various applications.

You may now wonder if you still need these servlets when moving to the portlet world. The answer is yes, there are still use cases that servlets address, like generating binary content or complete documents or providing some non-UI related service.

C.1.2 Java Server Pages (JSP) technology

Servlets solve many of the abovementioned drawbacks of CGI scripts, and they are server independent and scalable. They still have one drawback: the combination of dynamic and static content is a quite tedious process using just servlets that are optimized for dynamic content. Static HTML content is typically created

by people who are not Java programmers and thus are not willing to write a servlet. To simplify adding static content the JSP (Java Server Pages) technology was invented. JSPs allow you to easily create Web content that has both static and dynamic components. This is possible as the JSP technology provides all the dynamic capabilities of Java Servlet technology but in addition provides a quite natural approach to define static content. Let's try to summarize the main features of the JSP technology. The JSP specification defines:

- a language for developing JSP pages, which are text-based documents that describe how to process a request and construct a response
- an expression language for accessing server-side objects
- a mechanism for defining extensions to the JSP language

When moving to the portlet programming paradigm, JSPs are an important component. As we will see later on portlets are designed to be implemented according to the Model-View-Controller pattern and JSPs are a natural fit for implementing the View part of this pattern. Thus you will see JSPs mentioned quite a lot in this book and your JSP experience is still of great value in the portlet world. For more information on the Model-View-Controller pattern and how this is used in the portlet programming model see Chapter 3.2.3 "Programming patterns for portlets: the Model-View-Controller pattern".

Now that we have covered the basic Java web standards let's take a look what other Java standards may be useful for you as a portlet developer.

C.2 Further Java standards: JSR 127, JSR 168 and more

The Java Community Process creates new standards through Java Specification Requests (JSRs). In this section we will explore which Java standards are available to help you in writing your portlets. Using these Java standards will make your portlets robust against infrastructure changes, as many of these standards are aimed to shield applications from underlying infrastructures. In addition, these standards let portlets run on other portal infrastructures than the one it was originally developed for, enhancing their portability. Finally, portlets using standard APIs become easier to maintain, as standard APIs are well known by other programmers. Your successor in maintaining the portlet will thank you for using a standard API.

C.2.1 JavaServer Faces (JSF), JSR 127, JSR 252

JavaServer Faces (JSF) is a framework for web UI components and thus very important for portlet development. Use JSF to build function-rich UIs for your portlet. JSF V 1.1 is defined in JSR 127 and V 1.2 is currently being defined in JSR 252. JSF is explained in great detail in Chapter 5.

C.2.2 Java Portlet Specification, JSR 168

This is the core specification needed for programming portlets and will be covered in detail in the next chapter. The Java Portlet Specification provides an API for portlets and a contract between portlets and the portlet container running the portlets. With JSR 168, portlets become pluggable components that can be deployed on any portal server supporting JSR 168 out-of-the-box.

C.2.3 Content Repository for Java technology API, JSR 170

JSR 170 provides a common Java API for all kinds of content repositories. By using this API, portlets can access documents, discussion forums and other content via one API, regardless of the infrastructure that really contains these documents. Using this API in portlets will thus decouple the portlet from the content repository and therefore make the portlets easier to reuse in other portal environments. It also makes the portlet robust against changes in the content repository infrastructure.

The abstraction from the underlying storage infrastructure is called Java Content Repository (JCR) and is like a file system. It has different nodes that can be grouped in hierarchies and for each node additional

attributes can be defined. Typically the nodes will represent a folder, a complete document or a chapter of a document. JSR 170 also defines two query languages, which allow applications to search for content in the repository. The first query language is based on a SQL-like syntax, whereas the second query language is a subset of XPath 2.0.

C.2.4 CC/PP Processing, JSR 188

The Composite Capability/Preference Profiles vocabulary defined in [9] allows clients to describe themselves to the portlet. This means a client can include more information about itself when it appears to the portal: “I’m a Firefox browser with these extensions installed” or “I’m a Smart Phone, I can display color images of this size and quality and convert text to speech.” An application may then tailor the produced markup towards these client capabilities.

JSR 188 defines a Java API that allows retrieving this profile from the Java world. Unfortunately, the portlet specification and JSR 188 were developed in parallel and thus are currently not aligned. This means that the portlet API currently does not return a client profile object and the JSR 188 API only operates on servlet requests and do not take portlet requests as an input parameter. However, JSR 188 support is on the agenda for the next portlet specification version so hopefully this hole is filled quickly. For now, this leaves portlet programmers with parsing the HTTP client field or depending on portal vendor extensions to get access to the JSR 188 profile.

C.2.5 Service Data Objects (SDO), JSR 235

What are Service Data Objects (SDO) all about? SDO provides a service mediator pattern for data access APIs and thus provides an abstraction layer for portlets on the data itself. This means that the data can reside on any storage or backend system, like databases, DOMs, XML files, JCA connectors, or web services, and the portlet is not affected by these implementation details. The SDO object provides static APIs to access these data, like simple Java bean-like methods to access these data or dynamic APIs based on HashMaps for cases where more flexibility on the data format is required than the static bean methods allow.

SDO also provides a metadata model that describes the data stored in the SDO and allows introspection of the SDO object and data modeling tooling support. Another useful functionality SDOs support is the use of SDOs in a disconnected mode where the SDO object will store all changes locally and send back these changes once connected to the data store again.

So if your portlet accesses data that may be stored today in databases, but may come in the future from other sources, like web services, or you want your portlet to be independent from the storage infrastructure SDO is a great way to achieve this. SDO is already used in open source projects, like in the Eclipse Modeling Framework (EMF).

As of writing this section of the book, in August 2004, this JSR was just in its starting phase. Please check <http://jcp.org> for the current status of this JSR. We are nevertheless covering this standard as there is already a specification version out by BEA and IBM that can be found at <http://www-128.ibm.com/developerworks/library/j-commonj-sdowmt/> including a reference implementation that will serve as input for JSR 235.

Now that we have covered all the Java-based technologies of value for portlet developers, we’ll take a look at the web services world and how they can be of value for portlet development.

Now that we have covered all the Java-based technologies of value for portlet developers, we’ll take a look at the web services world and how they can be of value for portlet development.

C.3 Getting interoperability with web services and XML

Until now we have been in the Java-based J2EE world, however there is more out there. Another important platform is Microsoft’s .NET platform. Wouldn’t it be nice if you could integrate .NET services into your

J2EE-based portal? If you don't like this one how about another scenario: wouldn't it be cool if you could also sell your Java portlet to the .NET-based customers? All this can be achieved via web services, so let's spend some minutes on the foundations of web services.

The term web services came up during the Internet and Web boom. Every company wanted to be present on the Web and build their offerings based on common Web technology. But web services are more than a marketing buzz word. There is the technical aspect, too. When thinking of the Web, we usually think of HTTP. In fact, most web services will leverage HTTP as the transport mechanism. Is this because HTTP is in a technical sense the best transport option? No, it's far from it. But HTTP is well known and widely accepted. Companies feel comfortable using HTTP, and leave firewalls open to HTTP communication. This allows web services to be delivered into every organization today without the need of reinventing and bringing up new IT infrastructures. All we need is already there, at least from the transport point of view.

What about the service part in web services? Think of Application Service Providers (ASPs). Usually ASPs offer products and solutions, which are not directly deployed in the customer's infrastructure. Rather than this, they offer services. Imagine a credit card company allowing re-sellers to check the credit card information and limits of their customers. The cash register application of a flower shop talks directly to the credit card checking application provided by the credit card company directly over the network.

Basically, we are speaking about business-to-business communication (B2B) here. B2B in fact means application-to-application communication. Business-to-business has been there for years, so what's new about web services? Web services bring the two pieces together. They enable application-to-application communication over well-known Internet technologies.

That said, we can simply define a web service as follows:

A web service is any piece of code that communicates with other pieces of code over the Internet.

While this definition is enough to pinpoint the idea of web services, the truth is more complicated. To really enable application-to-application communication we need to make sure that the technologies we use are platform-, language- and vendor-neutral. Programs written in Java, C++ or COBOL need to communicate with each other. No matter what platform an organization uses, whether it's J2EE, .NET or any other architecture you want to think of, we need interoperability.

Web services promise to bring a solution for these problems. Using web services, software can be truly modularized even across networks. Application developers can concentrate on solutions relative to their business rather than thinking about the technical details of how to communicate and exchange data with various business partners. But web services are not a single masterpiece. Indeed web services are an amalgamation of several standard technologies.

(webservicesscenario.tif)

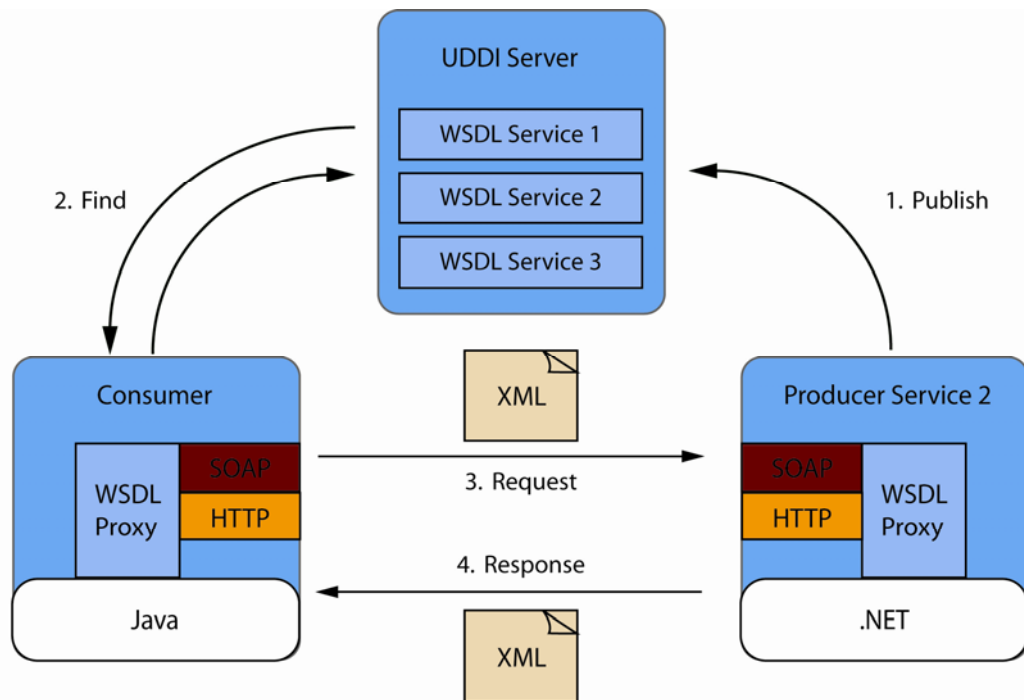


Figure C.1: The web service scenario with a producer publishing a web service to an UDDI Server and a consumer searching for a service in this UDDI directory, finding the service of the producer and binding to that service. The consumer can send directly, via XML, a request to the producer and receives a XML response.

Figure C.1 depicts the basic flow and involved technologies. First a producer of a service, say a stock quote service, register its service in a global registry. In our example this registry is a UDDI directory. The producer registers the service interface description, the WSDL, and maybe additional business information regarding this service. In our example the WSDL would define that the service would expect a company name as input parameter and would return the stock quote for that company as output parameter. Next our portlet, (the consumer that would like to provide stock quotes to the portal end user) is looking for a service that will enable the portlet to render the user's preferred stock quotes. The portlet developer searches in the UDDI directory for such a service and finds the entry of our producer. The corresponding WSDL file is downloaded to the consumer and a WSDL proxy is generated that translates between the Java and the XML world. When the portlet issues a request to the stock quote service the WSDL proxy translates this request into a platform independent XML document and connects via SOAP over HTTP to the service. The service on the other side receives the XML request and a WSDL stub translates the request into a .NET request in our example. The same mechanism is used for the response of the service and finally our portlet got the stock quote it requested. The portlet itself does not notice that it talked to a .NET service on the other side. This is the major benefit of using web services: your portlets can use services that need to be written in Java!

In this example a lot of abbreviations were mentioned, like XML, SOAP and WSDL. If you are not familiar with these, please take a look at Appendix B where we will cover these in more detail. Now let's take a look on why this web service stuff is something to know as portlet programmer.

C.3.1 How do portlets benefit from web services?

Maybe your are a Java programmer and now ask yourself why would a portlet programmer want to use this additional technology that seems to be quite complex on the first glance?

There may be different reasons why portlets want to use web services:

- the service may be written in a non-Java language
- the service is available via the Internet, like stock services or news services
- the service needs to be maintained on a central server that is different from the one running the portal

- the service may be of interest to other applications than just this portlet and thus should be programmed as a web service to allow arbitrary applications to use this service

All these are good reasons for Java programmers to use a web service and make a deeper dive into this technology. Another reason: we will leverage this knowledge for Web Service for Remote Portlets to which we come in the next section.

C.3.2 The next step: Web Services for Remote Portlets (WSRP)

As explained above, web services can be quite useful for portlets. However, this traditional way of including web services in portlets also has some disadvantages, like the need of proxy code in the portlet and the distribution of functionality between the service provider and the service consumer. In the stock quote service sample the portlet needs to understand the data format of the stock quote service and render the result accordingly. For portlets there is a more elegant solution to this problem: Web Services for Remote Portlets (WSRP) which we will explain in detail in Chapter 9 and that will build on top of the explained base web service technology.

Appendix D Table of Acronyms

A	
ANT, Apache	Another Neat Tool
API	Application Programming Interface
ASP	Application Service Provider
B	
B2B	Business to Business
C	
CSS	Cascading Style Sheet
CVS	Concurrent Versions System
D	
DB	DataBase
E	
EAR	Enterprise application ARchive
EJB	Enterprise Java Bean
EL	Expression Language
G	
GUI	Graphical User Interface
H	
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
I	
ID	Identification, Identifier
IDE	Integrated Development Environment
IOC	Inversion of Control
IP	Internet Protocol
IT	Information Technology
J	
J2EE	Java 2 Enterprise Edition
JAAS	Java Authentication and Authorization Service
JAR	Java Archive
JCP	Java Community Process
JDBC	Java Data Base Connectivity
JDK	Java Development Kit
JSF	JavaServer Faces
JSP	JavaServer Page
JSR	Java Specification Request
JSTL	JSP Standard Tag Library
JVM	Java Virtual Machine
L	
LDAP	Lightweight Directory Access Protocol
LTPA	Lightweight Third Party Authentication
M	
MIME	Multipurpose Internet Mail Extensions
MVC	Model View Controller
N	
NLS	National Language Support
O	
OASIS	Organization for the Advancement of Structured Information Standards
OM	Object Model
P	
P3P	Platform for Privacy Preferences
PC	Personal Computer
PDA	Personal Digital Assistant
PHP	PHP Hypertext Preprocessor (recursive acronym)
POJO	Plain Old Java Bean
R	
RC	Release Candidate
RI	Reference Implementation

S	
SSL	Secure Socket Layer
SSO	Single Sign On
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SPI	Service Provider Interface
SQL	Structured Query Language
T	
TCP	Transmission Control Protocol
TLD	Tag Library Descriptor
U	
UDDI	Universal Description, Discovery and Integration
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W	
WAP	Wireless Application Protocol
WAR	Web Archive
WML	Wireless Markup Language
WSDL	Web Service Description Language
WSRP	Web Services for Remote Portlets
WSRP4J	Web Services for Remote Portlets for Java, Apache project
X	
XHTML	Extensible HyperText Markup Language
XML	Extensible Markup Language

Appendix E References

[Apache]	Apache, Community, URL: http://www.apache.org/
[CC/PP]	<i>G. Klyne, F. Reynolds, C. Woodrow, H. Ohto, J. Hjelm, M. Butler, L. Tran</i> : Composite Capability/Preference Profiles (CC/PP) Structure and Vocabularies, W3C Working Draft 25 March 2003, URL: http://www.w3.org/TR/CCPP-struct-vocab/
[CERT]	CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests; URL: http://www.cert.org/advisories/CA-2000-02.html
[DBCP]	Jakarta Commons Database Connection Pool, http://jakarta.apache.org/commons/dbcp/
[Eclipse]	Eclipse; URL: http://www.eclipse.org/
[EMF]	Eclipse Modeling Framework, URL: http://eclipse.org/emf/
[Evans]	Evans Data Research about popularity of Eclipse; URL: http://www.evansdata.com/n2/pr/releases/EMEAAPAC04_01.shtml
[FaceAp]	My Faces, Moved to Apache, currently in incubator, URL: http://incubator.apache.org/myfaces/
[IBMRAD]	IBM Rational Application Developer; URL: http://www.ibm.com/software/awdtools/developer/application/index.html
[J2EE]	Java 2 Platform Enterprise Edition Specification, v1.4, URL: http://www.jcp.org/en/jsr/detail?id=151
[J2EEAp]	Designing Enterprise Applications with the J2EE Platform, Second Edition, URL: http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/index.html
[Jetsp2]	Jetspeed 2, Homepage of Jetspeed 2, URL: http://portals.apache.org/jetspeed-2/
[JDBC]	Java Database Connectivity , URL: http://java.sun.com/products/jdbc/
[JNDI]	Java Naming and Directory Interface, URL: http://java.sun.com/products/jndi/
[JSF]	<i>E. Burns, C. McClanahan, R. Kitain</i> : JavaServer Faces, URL: http://jcp.org/en/jsr/detail?id=127
[JSFCen]	JSF Central, Popular Java Server Faces Community, URL: http://www.jsfcentral.com
[JSFIntr]	Introduction to Java Server Faces, URL: http://java.sun.com/j2ee/jaserverfaces/jsfintro.html
[JSFRI]	JSF Reference Implementation from SUN, URL: http://java.sun.com/j2ee/jaserverfaces/index.jsp
[JSP]	<i>M. Roth, E. Pelegri-Llopart</i> : JavaServer Pages Specification Version 2.0, URL: http://www.jcp.org/en/jsr/detail?id=152
[JSR 235]	JSR 235: Service Data Objects, URL: http://www.jcp.org/en/jsr/detail?id=235
[JSR168]	<i>A. Abdelnur, S. Hepper</i> : JSR 168: The Java Portlet Specification, URL: http://jcp.org/en/jsr/detail?id=168 .
[JSR170]	<i>D. Nuescheler</i> : JSR 170: Content Repository for Java technology API, URL: http://www.jcp.org/en/jsr/detail?id=170

[JSR188]	<i>L. Tran</i> : JSR 188: Composite Capability/Preference Profiles (CC/PP) Processing Specification, URL: http://www.jcp.org/en/jsr/detail?id=188
[MVC]	Java BluePrints, Model-View-Controller, URL: http://java.sun.com/blueprints/patterns/MVC-detailed.html
[MyFace]	My Faces, Free Java Server Faces Implementation, URL: http://www.myfaces.org
[P3P]	<i>L. Cranor, M. Langheinrich, M. Marchiori, M. Presler-Marshall, J. Reagle</i> : The Platform for Privacy Preferences 1.0 (P3P1.0) Specification; URL: http://www.w3c.org/TR/P3P
[Patterns]	<i>E. Gamma, R. Helm, R. Johnson, J. Vlissides</i> : Design Patterns, Addison-Wesley, 1995
[Pluto]	JSR 168 Reference Implementation Pluto; URL: http://jakarta.apache.org/pluto/
[PlutoEc]	Eclipse plug-in for Apache Pluto; URL: http://plutoeclipse.sourceforge.net
[Schema]	XML Schema, URL: http://www.w3.org/XML/Schema
[SDO]	<i>J. Beatty, S. Brodsky, R. Ellersick, Martin Nally, R. Patel</i> : Service Data Objects specification, URL: http://www-106.ibm.com/developerworks/library/j-commonj-sdowmt/
[Servlet]	<i>D. Coward, Y. Yoshida</i> : Java Servlet Specification Version 2.4, URL: http://www.jcp.org/en/jsr/detail?id=154
[SOAP]	<i>D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, D. Winer</i> : Simple Object Access Protocol (SOAP) 1.1, URL: http://www.w3.org/TR/2000/NOTE-SOAP-20000508/
[SSO]	Single Sign On
[Tomcat]	Tomcat, Homepage of Tomcat, URL: http://jakarta.apache.org/tomcat/
[UDDI]	Universal Description, Discovery and Integration (UDDI), URL: http://www.uddi.org/
[WSDL]	<i>E. Christensen, F. Curbera, G. Meredith, S. Weerawarana</i> : Web Services Description Language (WSDL) 1.1, URL: http://www.w3.org/TR/2001/NOTE-wsdl-20010315
[WSRP]	Web Services for Remote Portlets ; URL: http://www.oasis-open.org/committees/wsrp/
[XML]	Extensible Markup Language (XML), URL: http://www.w3.org/XML/
[XPath]	XML Path Language (XPath) 2.0, URL: http://www.w3.org/TR/xpath20/

Further readings

- D. K. Fields, M. A. Kolb, S. Bayern: Web Development with JavaServer Pages, 2nd Edition, 2001, Manning Publications
- J. Falkner, K. Jones: Servlets and JavaServer Pages: The J2EE Technology Web Tier; URL: <http://www.theserverside.com/books/addisonwesley/ServletsJSP/index.tss>
- S.Hepper, M.Lamb, Best practices: Developing portlets using JSR 168 and WebSphere Portal V5.02; URL: http://www-106.ibm.com/developerworks/websphere/library/techarticles/0403_hepper/0403_hepper.html
- Kito D. Mann: JavaServer Faces in Action, Manning, October 2004
- www.theserverside.com – for discussions around J2EE topics
- <http://www.ibm.com/developerworks/java> – for additional technical articles around servlet and portlet programming