Building Portals with the Java Portlet API

JEFF LINWOOD, DAVE MINTER

Apress®

Building Portals with the Java Portlet API
Copyright © 2004 by Jeff Linwood, Dave Minter

ISBN (pbk): 1-59059-284-0

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin

Technical Reviewer: Carsten Ziegeler

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, John Franklin, Jason Gilmore, Chris Mills, Steve Rycroft, Dominic Shakeshaft, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Project Manager: Kylie Johnston

Copy Edit Manager: Nicole LeClerc

Copy Editor: Liz Welch

Production Manager: Kari Brooks

Production Editor: Ellie Fountain

Compositor: Kinetic Publishing Services, LLC

Proofreader: Nancy Sixsmith

Indexer: James Minkin

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, e-mail `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, e-mail `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

Contents at a Glance

Contents

Foreword

The phenomenal amount of information that networked computers can present to us is both the marvel and the bane of our time. Knowledge is commonly supposed to be power, but the reality is that we are often drowning in data, overwhelmed rather than enabled. The inexorable rise in the volume of facts and figures at our disposal should be A Good Thing, but unless we have the right tools to manage this information, we will struggle to keep our heads above water, let alone take full advantage of the data.

Arguably the single most important challenge in the computing world today is to provide users with the means to stay on top of the information they require. Connectivity is no longer enough—merely providing someone with a web browser and an Internet connection is roughly akin to supplying them with a small dinghy in order to circumnavigate the globe.

Two elements are crucial to solving this problem successfully: aggregation and selectivity. Aggregation technologies make multiple sources of information available in one place. Selectivity is the ability to exercise control over what is presented, and is necessary to exploit aggregation without being overwhelmed.

Search engines are the archetypical aggregation success story—the Internet would be orders of magnitude less useful without the ability to search the entire Web from one place. However, search engines are weak when it comes to selectivity. They necessarily cast their net very wide, which inevitably means that searches tend to return a lot of irrelevant data. This places the burden on users to sift through the results for data of value.

Search engines continue to improve their selection capabilities, with increasingly sophisticated algorithms for determining which pages are likely to be most relevant. However, search engines inevitably run up against the problem that different individuals are likely to be looking for different things when feeding in the same query. For example, someone I know was recently looking for information on dressage horses, and while the majority of results Google returned when she searched for "stallion german" were equine, a few of the results were catering to an entirely different market.

Recently, user-driven aggregation has been gaining ground, most notably in the form of RSS aggregators. These lack the all-encompassing reach of a search engine, but score much higher on selectivity—they retrieve information only from sources in which users have expressed an interest. This highly selective form of aggregation enables us to keep abreast of updates across hundreds of web sites without having to spend all day, every day visiting those sites in the browser.

The Java portlet architecture provides a framework for building systems that present users with the information they need. It offers the two key ingredients for success: aggregation (portals can aggregate information from multiple portlets) and selectivity (the architecture allows administrators and users to be selective about their sources of information by choosing which portlets will appear).

Moreover, portlets allow information from all kinds of sources to be aggregated, so a portal's reach is potentially much greater than that of either a web search engine or an RSS feed. Of course, portlets are available to handle both web content and RSS feeds, but portlets can also allow information from web services to be added to a portal, or from legacy systems. If you can retrieve information from a system with Java code, you can write a portlet for it.

Dave and Jeff have provided a comprehensive guide to the portlet architecture in this book. But of course, this technology will never be used in isolation—its basic purpose is to integrate diverse sources of information. Accordingly, they also describe the main technologies you are likely to come across when building portals. This book offers a guide to the various incompatible versions of RSS, shows you how to integrate the Lucene search engine into your site, and discusses various content management technologies.

Since the portlet specification is a fairly recent addition to the suite of Java specifications, very little portal-aware software exists right now. This means that for the time being, a lot of portlet development will involve integrating existing code into new portal environments. This book therefore provides a fully worked example, showing the effort required to take the open source YAZD forum software and wrap it as a portlet.

In short, the portlet specification provides the tools for building web sites that will enable users to exploit the potential of the information available to them, and this book tells you all you need to know to build great portals.

Ian Griffiths
Developer, consultant, and teacher
`www.interact-sw.co.uk/iangblog/`

About the Authors

Jeff Linwood is a software developer and consultant with the Gossamer Group (`www.gossamer-group.com`) in sunny Austin, Texas. Jeff has been in software programming since he had a 286 in high school. He was caught up with the Internet when he got access to a Unix shell account, and it has been downhill ever since. Jeff coauthored *Pro Struts Applications* (Apress, 2003), and was a technical reviewer for *Enterprise Java Development on a Budget* (Apress, 2004) and *Extreme Programming with Ant* (SAMS, 2003). He has a chemical engineering degree from Carnegie Mellon University.

Dave Minter is a freelance integration consultant and developer from rainy London, England. The first computer that he encountered was a Wang 2200 minicomputer, which at the time was roughly the same size as he was. Since then, he has worked for the largest of blue chip companies and the smallest of startups—encountering Jeff during the dotcom frenzy along the way. These days, he makes his living explaining to companies how they can build systems that "just work." He has a computer studies degree from the University of Glamorgan.

About the Technical Reviewer

Carsten Ziegeler is a member of the Apache Software Foundation and as such is involved in various open source communities, like Cocoon, Avalon, and Excalibur. He is a member of the Apache Portals project management committee and committer of the Pluto and the WSRP4J project.

In paid life, Carsten is the chief architect of the Open Source Group at S&N AG, Paderborn, Germany. The focus is on Java-based middleware functionality such as web frameworks, component architectures, and portal solutions and technologies.

The liaison to Apache started in 2000 when Carsten became committer of the Cocoon project and started to play an important role in designing and developing the current architecture. A major contribution to Cocoon is the standard-compliant Cocoon Portal.

Acknowledgments

THIS BOOK would not have been possible without the energy and enthusiasm of the Apress staff, especially John Zukowski for his decision to let us write it; Kylie Johnston, our project manager; and Steve Anglin, our editor. Special thanks to Liz Welch for a superb job of copyediting our chapters.

We owe particular thanks to Ian Griffiths (the gurus' guru) for his foreword to this book, and we owe a debt of gratitude to Carsten Ziegeler, our technical reviewer, whose comments have without exception been both pertinent and pithy.

Jeff would like to thank Karl Weinmeister, for his ideas on what should be in a portal development book, and Skip Walker, for reviewing the Single Sign-On material. Jeff would also like to thank his family for being supportive.

Dave would like to thank his parents for their patience with his little hobby, and his girlfriend Kalani Seymour for her tolerance and good humor as deadlines came and went. Thanks also to Bruce Robinson for the mug shot.

Any errors in the ensuing content are, of course, entirely our own doing.

Introduction

IT SHOULD BE possible to build a portal by plugging components from different vendors into a portal from any vendor. These components are portlets, and we explain how to build them in this book.

The noble aim of the portlet specification—which arose from Sun's Java Community Process with the collaboration of Sun, IBM, BEA, and others—was to simplify the process of tying applications into a portal by allowing them to cooperate. That so many vendors have come together to standardize their existing proprietary solutions bodes well for the future of this technology.

We believe that portlets and the portlet API will become at least as important to Java application developers as the servlet API has been because portlets make building a truly integrated system that much easier. Any new portal development projects should select a portal that supports the portlet API because independent software vendors now need to write portlets for only one API, not a dozen.

Both of us enjoy working with new technology, and there are a lot of new standards for portals, content management systems, business rules, and web services. We hope that you will enjoy learning about portal development as much as we enjoyed writing this book!

Who This Book Is For

This book is for developers who already have a command of the basics of web application development in Java. Ideally, you will have had some exposure to servlets and JSP pages. No prior knowledge of portlets or portal development is required. Some very basic knowledge of XML is useful.

All of our examples use standards or use open source software, so it will not be necessary for you to purchase any software to get started with portlet development. Because the portlet API is a standard, you can begin development on a free, open source portal, and then migrate your applications to a commercial portal.

This book is not an academic text—our focus is on providing extensive examples and taking a pragmatic approach to the technology that it covers.

How This Book Is Organized

We realize that many of our readers will be familiar with servlets and some of the core concepts of portlets when they come to this book. We recommend to such readers that they familiarize themselves with the following chapter guide so that they can quickly refer to the subjects they are interested in.

We have also tried to ensure that a portlet novice will find that these chapters are logically ordered, with the more advanced subjects covered only when the basics have been described in detail.

You will find the source code for the book's examples on the Apress web site (`www.apress.com`), on the Downloads page.

Chapter 1: Introduction to Portals and Portlets

This chapter outlines the basic concepts and terms that you will encounter in the book. We talk in broad terms about the strengths and weaknesses of portlets, and we give you an overview of some of the technologies that we cover more fully in later chapters.

Chapter 2: Portlet Basics

This chapter provides an example of a simple portlet, discusses how it works, and demonstrates how to build the application. We then introduce the open source Pluto portal and show how you can deploy the example portlet on Pluto.

Chapter 3: The Portlet Life Cycle

In this chapter, we discuss how a portlet interacts with a portal, from initialization to removal. We provide a simple example that walks you through the stages of the portlet life cycle, as well as a more complex example that illustrates the issues of multithreaded portlet applications.

Chapter 4: Portlet Concepts

This chapter introduces many of the basic portlet concepts for the first time, or in more detail, and much of the API is examined in depth. An example ties many of these concepts together to demonstrate file upload to a portlet.

Among many other topics, the chapter discusses

- Request and response objects
- Attributes and properties
- The portlet context

- Locales and internationalization
- Logging
- The API versioning scheme
- Sessions
- Default and custom modes
- Default and custom window states

Chapter 5: Using Servlets and JavaServer Pages with Portlets

Chapter 5 demonstrates how to invoke and include content from servlets and JSP pages. Session management, the creation and processing of HTML forms, and the portlet tag library are all addressed. We provide an example of a to-do list portlet to illustrate these techniques.

Chapter 6: Packaging and Deployment Descriptors

In this chapter, we show you how to use the portlet deployment descriptor. We also demonstrate XDoclet's portlet integration, which lets us build and deploy portlets easily.

Chapter 7: Portal and Portlet Configuration

This chapter describes the standard configuration information available to a portal and the portlets it contains. It discusses

- The `PortalContext` class
- Portal properties
- Window states and portlet modes configuration
- The `PortletConfig` class
- Portlet preferences

Chapter 8: Security and Single Sign-On

This chapter demonstrates how to integrate a portlet with a Single Sign-On solution using Kerberos as an example. We also discuss many of the other authentication and authorization technologies that are available to a portlet developer.

Chapter 9: RSS and Syndication

You'll learn how a portlet can incorporate syndicated links from other sites and how an application can present its own links to similarly capable external sites.

Chapter 10: Integrating the Lucene Search Engine

Lucene is a powerful, open source search engine. We show you how to create an index with Lucene, and then how to build a portlet that searches content in that index.

Chapter 11: Personalization and User Attributes

This chapter examines the information available to personalize portlets for the current user, and we describe the limited but useful facility for persisting user data. We discuss the use of a rules engine to govern portlet content decisions.

Chapter 12: Web Services for Remote Portlets (WSRP) and Application Syndication

We discuss the Web Services for Remote Portlets (WSRP) specification, and then tie WSRP into the broader problem of application syndication.

Chapter 13: Exposing an Existing Application As a Portlet

This chapter demonstrates how an existing real-world application, the YAZD forum software, can swiftly be converted into a portlet application using the techniques described in earlier chapters.

Chapter 14: Charting with JFreeChart

We apply the open source JFreeChart project to provide professional data-charting capabilities within a portlet.

Chapter 15: Content Management Systems

In our final chapter, we discuss integrating content management systems (CMSs) into portlets. We provide an overview of the new JSR 170 Java Content Repository API specification for CMS integration. WebDAV is a standard protocol for working with content management systems, and we build a portlet client for a WebDAV server.

CHAPTER 1

Introduction to Portals and Portlets

THIS BOOK IS FOR SOFTWARE developers and designers who develop Java applications for portals. We cover version 1.0 of the Java portlet API, also known as Java Specification Request (JSR) 168. Portlets are the individual components that provide content for a portal. Portals aggregate one or more portlets into web pages, which are usually personalized or customized for individual users or groups of users. Some portals also support mobile devices and voice support.

Before the release of this portlet API, each portal had a different API for developing portlets. Most Java portal vendors will support the JSR 168 standard in addition to their existing proprietary API. If you develop your portlets to the new portlet API standard, you can deploy them on any JSR 168-compatible server, just as any compatible servlet container can deploy servlets.

You may use the open source portal server Apache Pluto to run the portlets we write in this book. You are able to deploy your portlets on any other portals that support the standard, because none of the portlets will use any proprietary features. We use several open source software components to provide additional functionality beyond the portlet API.

Some of the problems we provide solutions for in later chapters are personalization, portal deployment, Single Sign-On (SSO), content syndication, and the porting of an existing application into a portal infrastructure. In this chapter, we discuss portals, information architecture, and background on the portlet API.

Providing a Solution with Portals

Usually, the decision to build a portal environment is made at a high level within an organization after users become frustrated with using applications that are not integrated and are not immediately visible. Other times, a project involving an extranet for suppliers and customers gets started, and the easiest way to aggregate security for all of these new users is through a portal's SSO feature. In this book, we do not discuss the business case for a portal within an organization. We wrote this book for developers and architects who have chosen to use a portal server that implements the Java portlet API and need to solve technical problems.

From a technical perspective, a portal provides a solution for aggregating content and applications from various systems for presentation to the end user. The users do not need to know how the content or functionality is provided; they just want to enjoy the benefits of a single web site and all of its services. Typically, a portal has an integrated user interface and an SSO approach for security. The software developer's job is to take all of the systems that provide these services and add interfaces to them to work with the portal. Portlets are the individual components displayed in the portal. Prior to the introduction of the standardized portlet API, portlets had to be custom-developed for each portal server because the API was different for each server. The leading portal vendors joined to create a standard to promote portal technology. Inside the Java Community Process (JCP), the name of the standard for the first version of the Java portlet API is JSR 168. Future versions of the portlet API will have different JSR numbers.

One of the problems for the designer or architect in charge of the portal project is that the existing systems do not always separate cleanly into presentation and business logic layers. Also, consider portal security and personalization when examining existing applications. In this book, we port an existing web application into a portlet application, so you can learn from some of our portlet integration problems. New software projects that integrate with the portal can use a services-oriented architecture with exposed web services, a stand-alone web application, or a portlet.

Portal projects have two major technical components designed in parallel: application architecture and information architecture. Both of these will flow from business requirements, and they require an integrated approach. If the portal applications do not support the common information architecture, the users will have a substandard experience. We discuss creating an information architecture for a portal environment in the next section.

Designing the Portal's Information Architecture

Moving all of your applications into a portal does not accomplish anything if your users are not able to solve their problems. One of the first steps for deploying a portal solution effectively is to gather requirements from the users and design the information architecture for your portal project. The information architecture includes the content displayed through the portal, the user interface, the available portlets, metadata, a search engine, and a classification system or taxonomy. The portal's information architecture defines the user-centered approach to the portal, while the technical architecture is what the developers use to build the portal. Aligning these two forces is a difficult task, but it is necessary for a successful project.

If you have not identified all of the users of the portal system yet, try to account for at least the three main types of users for a portal project: customers, suppliers, and employees. Most portal projects utilize phases or stages, with the initial phases usually being deployed only to a smaller group of users, usually side by side with existing systems. This will lower the risk profile for the project and cut initial support costs.

Identifying Content for the Portal

The business requirements for the portal determine the different collections of content. The content could be in content management or document management systems, in a database, on another web server, on the file system, or in any number of other places. Not all of your content is going to be web-ready, and you may need to write adapters to translate legacy content to XML or HTML. Some portal content may be syndicated using Rich Site Summary (RSS) feeds.

Another set of requirements determines who has access to what content. This can be set up with access rights, with pieces of content mapped into content collections, and users assigned into groups that can access these collections. Your content management system may already have all of this access control built in, and part of the portal project could be to integrate the portal authentication with the content management system security. Other implementations may have to build content security functionality on top of the portal's security model.

Identifying the Metadata for the Content

Most organizations do not have an enterprise-wide standard for their content metadata yet. Creating one makes the portal project much easier. Metadata is any descriptive information about content that can provide context, such as the title, creator, timestamp, or description. Traditionally, content cataloging has been a field where librarians excel, but it is certainly possible for content creators to learn how to provide correct metadata.

One standard for metadata is the Dublin Core Metadata Element Set (`www.dublincore.org/documents/dces/`). This set of 15 metadata elements contains fields for

- Title
- Creator
- Subject
- Description

- Publisher
- Contributor
- Date
- Type
- Format
- Identifier
- Source
- Language
- Relation
- Coverage
- Rights

The metadata can be stored inside the HTML document within `<META>` tags, as elements in an XML document, in a database, or in a Resource Description Framework (RDF) file that is separate from the content file.

If your content repositories do not have this information, you will need to create the metadata from the existing content. This can be a time-consuming manual process, but commercial tools for metadata extraction are available.

Designing the User Interface

Most portals (including the open source Apache Jetspeed 2 portal) are customizable using a set of skins, or themes, that provide look and feel. The HTML and style sheets for the portal are contained in the skin. The layout of the initial portal page the users will get when they log in is usually customizable through the administration tool. Many users will not customize the layout of this page, even if allowed, so an effective design is important. Determine which applications or content sources are going to be used most often (e-mail, human resources, engineering documents, etc.). Build prototypes of the proposed screens, and let users interact with the portal functionality.

From a technical perspective, changing the portal layout or look and feel after deployment is very simple, but the end users may require additional training and notification that the site layout is going to change. Some end users will always be resistant to change, even if it would improve their productivity, and this affects possible redesigns.

Creating an Effective Search Engine

We look at integrating the open source Jakarta Apache search engine Lucene with a portlet in Chapter 10. Your search engine should index as many of your content sources as possible, but it will probably need to be broken down into different collections for different classes of users. The metadata for the content will become fields in Lucene's search index for advanced searching. Commercial search engine vendors should provide portlets that plug into any portal server that implements the standard.

Portal Application Architecture

The information architecture leads to the technical architecture of a portal project. Part of the portal architect's job is to link the two together into a coherent design. For instance, if the business users require a natural language search engine or a structured view of all content in the system, the technical components used to build the portal must reflect this.

All portals that are compatible with the portlet API will have a similar structure. The portal will need to run inside a servlet container such as IBM WebSphere or Apache Tomcat. Because each portlet application deployed on the portal is also a web application, the servlet environment serves any web resources such as servlets or JavaServer Pages (JSP) files. The portal is responsible for providing administrative functionality, a layout for the portlets on the portal page, and the execution of the portlets within a portlet container. The portlet container may be a separate piece of software, but most portal implementations will integrate the container into the portal. Just as a servlet container is responsible for executing servlets, the portlet container will execute any portlets. The portlets have an execution life cycle that we will discuss in Chapter 3.

In this book, we use the terms *portal* and *portal server* interchangeably to reference the server-side application that runs the portlets, manages users, and displays portal pages.

Each visitor to the portal will receive a portal page, which will contain one or more portlets in a customizable layout. These portlets could be commercial portlets that integrate with your existing systems, open source portlets customized for your installation, or custom portlets you have created. Portlets are as easy to create as servlets, and your servlet programming background will help in creating effective portlets.

Building Portlets with the Portlet API

Portlets are components written in Java against the portlet API. The Java classes in the portlet API are in the `javax.portlet` package. Each portlet takes a request from the portal container and returns a response. The response contains content that the portal container will display as part of the portal page sent to the end user. Portlets may include JSP pages, Velocity templates, or another presentation layer technology. Just as with servlets, few developers will put any content directly into the Java code for any nontrivial portlets. In this book, we will use HTML inside of the portlet's Java code until we explain how to use JSP pages inside a portlet in Chapter 5.

The portlet container is responsible for sending requests from the portal to the portlet, and then passing the portlet response back to the portal. It also manages the initialization of the portlet, along with other life cycle events. The portal is responsible for taking the content from the portal container for each portlet and building a web page for the end user. The portal handles the layout, aggregation, and any personalization or SSO security features.

The portlet application is a standard Java 2 Enterprise Edition (J2EE) web application, with the addition of portlet classes and a portlet deployment descriptor named portlet.xml. The directory structure of the portlet application is the same as a web application's layout. A web application archive (WAR) file is also the format used for packaging the portlet application. The portal vendor is responsible for providing deployment and administration tools.

The standard build and packaging tools provided with your choice of Java programming environment or integrated development environment (IDE) for web applications are usable for building portlet applications. We are using the open source Java build tool Ant in this book. If you are not familiar with Ant, we recommend you check out *Java Development with Ant* by Erik Hatcher and Steve Loughran (Manning, 2002) or *Extreme Programming with Ant* by Glenn Niemeyer and Jeremy Poteet (Sams, 2003). Another source to check is the Apache Ant Manual on the Ant web site (`http://ant.apache.org/`).

Our example portlets deploy into the open source portal Apache Pluto (`http://jakarta.apache.org/pluto`), although they will work on any compatible implementation of the portal. Apache Jetspeed (`http://portals.apache.org/jetspeed-2/`) is a full-featured open source portal from Apache. There are currently two versions of Jetspeed: Jetspeed 1 does not support the Java portlet API, and Jetspeed 2 does support JSR-168. EXO (`www.exoplatform.org`) is another open source portal that supports the new Java portlet API, and it also supports Struts and JavaServer Faces (JSF). Most of the commercial Java portal vendors will have implementations of the Java portlet API out already or shortly. Some bundle JSR-168 support into a new version, and others are releasing support as an add-on module.

Providing Technical Solutions with Portals

Portal implementations usually involve a wide variety of interesting technical problems. Because they are generally systems for integrating a range of business systems, content stores, and web applications, the goals for the portal project are usually specific to the installation. Some of the goals that typically come up are integrating a search engine with the portal and providing an SSO interface for security. Other common development tasks for portal projects include extending personalization across all of the portal's applications, integrating content management systems, and creating portlets for systems that were not designed for portals.

In this book, we discuss all of these problems, plus several others, and provide solutions that work within a portal environment. Our hope is that you will be able to take our solutions and make them work for your problems.

Security and Single Sign-On

The portlet API has a basic security model based on the servlet security model. Security is an area that is going to depend heavily either on the proprietary features of your chosen portal, or on leveraging a third-party product that can handle a unified security model.

Single Sign-On is a key requirement for most portal projects, and we cover the different strategies that can be used for SSO projects. We cover SSO in Chapter 8.

Content Syndication and RSS

One common requirement for a portal deployment is to implement content syndication on the portal for multiple sources of content. This content could be from a content management system; from an internal groupware application; or from external sources like Reuters, Yahoo!, or CNN. In this book, we discuss using the RSS format for syndicating content on channels. The RSS portlet can consume RSS feeds that other sites publish.

We also discuss the mechanics of publishing an RSS feed for an existing content store. Our search engine integrates with RSS, so a content feed constantly updates a list of the top hits for a given search. This is useful for creating ad hoc knowledge management systems within your organization.

We will discuss RSS and content syndication in more detail in Chapter 9.

Searching Content from the Portal

In this book, we use the open source search engine Lucene from the Jakarta Apache project to create a portlet for searching content in a Lucene index. Our Lucene search portlet is compatible with any content indexed with Lucene. We demonstrate how to create a simple index from content on the file system. We also deliver the content for the search results inside the portlet.

We create the Lucene search portlet in Chapter 10.

Portals and Web Services

Portals are a natural fit for a services-oriented architecture. Portlets contain the user interface and controller logic, and call out to a service to retrieve information or execute a transaction. These services run on any platform that supports a Simple Object Access Protocol (SOAP) web services API, and the portlet calls out to them using a Java SOAP toolkit like Axis from the Web Services Apache project, or Glue from webMethods. These services can interface with existing mainframe or client server applications, and new enterprise applications should expose a SOAP web services API.

Figure 1-1 shows an example of a portal and web services architecture for a school or university. The enterprise systems for student information and course scheduling have a web services layer that exposes core functionality to the portlets. The web-based courseware service acts as a stand-alone service that can supply content to users of the portal. A desktop application uses SOAP to access the course scheduling system.

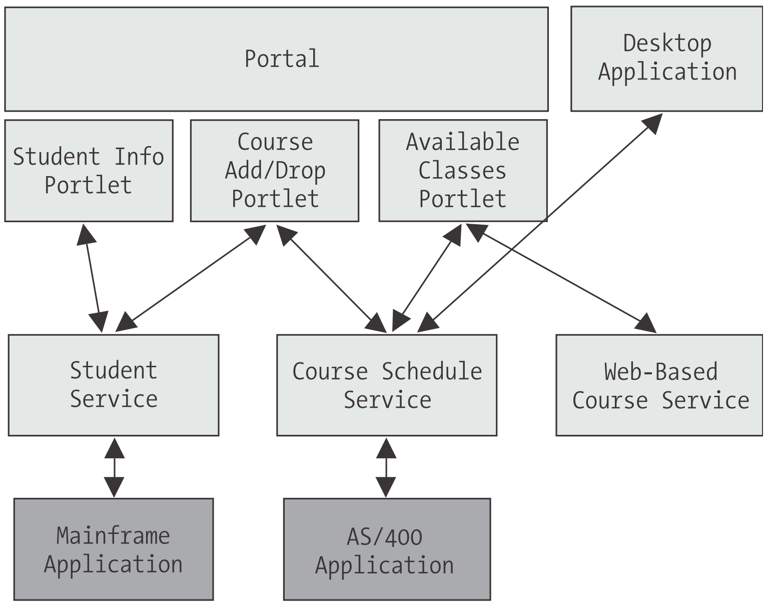

Figure 1-1. Example web services and portal architecture

Web Services for Remote Portals (WSRP)

Another approach is to create a distributed portal infrastructure. A portal that supports the Web Services for Remote Portals (WSRP) standard can display portlets from other WSRP-enabled portals. The producer portal publishes its available portlets to a registry, and a consumer portal can display a remote portlet from the registry. The protocols for displaying a portlet from a remote location were standardized with the WSRP specification from the Organization for the Advancement of Structured Information Standards (OASIS). We will cover WSRP in Chapter 12.

Integrating Existing Applications into the Portal

There are several approaches to bringing existing applications into a portal environment. The application can have a WSRP layer written on top, and host itself as a portlet producer. Another approach that uses SOAP is to completely write a thin web services layer on top of the core functionality, and then develop a portlet for that application.

For Java applications that do not have a services layer to expose, or where the value of the application is in the user interface, it makes sense to consider rewriting the application as a portlet or set of portlets. The business logic and persistence can be factored out of the existing application as it is rewritten. This creates a common base layer for both the new portlet user interface and the existing user interface.

Non-Java applications will have to be exposed using SOAP or another cross-platform API. For mainframe and AS/400 character mode applications, several commercial screen scraper products can translate a terminal interface into calls to and from a Java application. The portal may use a proxy portlet to serve existing web-based applications through the portal. The proxy portlet receives a request from the portal’s end user and translates it into a web request for the existing application. The existing application responds to the proxy portlet’s request with a response, which the proxy portlet then translates into a response for the portal. The portal aggregates that response with the rest of its content to produce a page for the end user. Some of the issues to consider when designing a proxied system like this one are SSO and security, personalization, a consistent look and feel, and the ability to keep track of sessions at the proxy portlet level.

We convert a web-based open source message board to use portlets in Chapter 13. We used the open source forums package YAZD as the beginning of our project. YAZD is built on a servlet and JSP architecture. We built a controller portlet that dispatches requests to the appropriate JSP page.

Using Charts in the Portal

Business intelligence and other analytical applications often use charts to communicate information. Some portal projects create "digital dashboards" that represent the current state of the organization for managers—for instance, a sales manager could see outstanding sales calls for each salesperson, sales margins for products, and profit projections for the quarter. We use the open source charting product JFreeChart to develop charts for portlets in Chapter 14.

Content Management and Portlets

In Chapter 15, we discuss two standards for communicating with content management systems: the Java Content Repository API and the WebDAV protocol. We create a portlet that uses the WebDAV protocol to integrate with a content management system. In this chapter, we use the open source WebDAV client library from the Apache Slide project to build our portlet.

Summary

Portal projects that use standards are going to be easily portable to new portal containers, and developers can use portlets from third-party vendors to create their portals. Before starting work on the production version of your portal, develop an information architecture from the user's perspective. It should show how the portal is going to look, what it is going to do, and what problems it solves. Start small, with a prototype or deployment to a limited number of users, and build out from there, to ensure that your portal deployment scales.

NOTE *The authors created a web site for this book at* `www.portalbook.com`. *We will have interesting articles, sample portlets, and more information about upcoming portal standards and APIs.*

CHAPTER 3

The Portlet Life Cycle

As we've seen in earlier chapters, portlets are conceptually very similar to servlets. Like servlets, they can only operate within a container. Both have obligations that their design must satisfy to allow them to interact with their container, and both demand clearly specified behavior from their containers.

The portlet's obligations are, broadly speaking, to provide implementations of specific methods, to respond appropriately when these are invoked by the container, and to handle error conditions gracefully.

This chapter describes the container's interactions with a portlet, starting with its creation and concluding with its destruction. It also considers the constraints that are incumbent upon both the portlet and the container at each step in this life cycle.

The Portlet Interface

To demonstrate the basic steps in the life cycle, let's first look at a simple portlet that implements the `Portlet` interface directly. Portlets need to implement this interface, either directly, or indirectly by extending a class that has already implemented the interface.

Here is a devastatingly simple portlet example:

```
package com.portalbook.crawler;

import java.io.*;
import javax.portlet.*;

public class SimplePortlet
   implements Portlet {

   public SimplePortlet() {
   }

   public void destroy() {
      portletCounter--;
   }
```

```
public void init(PortletConfig config)
   throws PortletException
{
   portletCounter++;
}

public void processAction(
      ActionRequest request,
      ActionResponse response)
   throws PortletException, IOException
{
   actionCounter++;
}

public void render(
      RenderRequest request,
      RenderResponse response)
   throws PortletException, IOException
{
   renderCounter++;

   response.setTitle("Simple Portlet");
   response.setContentType("text/html");
   PrintWriter out = response.getWriter();
   out.write("The server has instantiated " +
         portletCounter +
         " copies of the portlet<br>");

   out.write("This portlet has been rendered " +
         renderCounter +
         " times (including this one)<br>");

   out.write("This portlet has received " +
         actionCounter +
         " action requests<br>");

   PortletURL action = response.createActionURL();
   out.write("Click <a href=\"");
   out.write(action.toString());
   out.write("\">here</a> to trigger an action.<br>");
}
```

```
   private static int portletCounter = 0;
   private int renderCounter = 0;
   private int actionCounter = 0;
}
```

The portlet.xml file for the simple portlet follows:

```
<?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>PortletBook Simple Portlet</description>
      <portlet-name>simple</portlet-name>
      <display-name>Simple Portlet</display-name>
      <portlet-class>com.portalbook.crawler.SimplePortlet</portlet-class>
      <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>Simple Portlet</title>
         <short-title>Simple</short-title>
         <keywords>Simple, Example</keywords>
      </portlet-info>
   </portlet>
</portlet-app>
```

This portlet tracks the number of instances of the class that are being maintained by the portlet container at any given time, it counts the number of times that the specific instance has been rendered, and it counts the number of action requests that have been handled by the specific instance.

In the last section of this chapter, we will build a more realistic portlet application that demonstrates some of the issues involved in a threading application, and that builds upon the `GenericPortlet` class to make more complex portlet reactions possible.

Overview

It shouldn't come as much of a surprise to find that the life cycle of a portlet is broadly the same as that of a servlet. Servlets are generally responsible for rendering complete pages, and portlets are generally responsible for rendering fragments of pages, so there's an obvious correlation.

The life cycle of a portlet therefore breaks down into the following stages:

1. Creation of the portlet
2. Processing of a number of user requests (or possibly none)
3. Removal and garbage collection of the portlet

Creation of the Portlet

The creation of the portlet is probably the most complex "phase" in the life cycle since it involves three quite distinct steps. However, two of them—loading and instantiation—are very familiar Java concepts.

Loading the Classes

The container is able to load the classes required by the portlet at any point prior to invocation of the constructor.

A portlet application often consists of many classes and libraries, for which the actual portlet class is a relatively minor part of the whole. However, the portlet represents the user's way of interacting with the application. As such, it must have access to the rest of the application. The specification therefore demands that the portlet be loaded by the same classloader as the rest of the portlet application.

This guarantees that the servlets and other resources of the application may be accessed by the portlet that integrates it into the portal.

As with any class, at load time the class attributes will be initialized to their default values, so our variable `portletCounter` will be set to zero.

Invoking the Constructor

A portlet is required to provide a public default constructor—that is to say, a constructor taking no parameters.

Loading and instantiation (invoking the constructor) can take place either when the container starts the portlet application or when the container determines that the portlet is needed to service a request.

The option for delayed loading presents a benefit when a portlet will be used infrequently and consumes substantial resources, since they will not be acquired until they are actually needed. The trade-off is against performance, since the time taken by the portlet to service a request will be increased by the time taken to initialize its resources—but this will affect only the first user of the portlet. Where a portlet is initialized with the portlet application, the hit is taken "up front" when the application starts.

Our (minimal) constructor looks like this:

```
public SimplePortlet() {
}
```

It's hard to say anything interesting about our sample constructor. The normal instance initialization will take place, so that before the invocation of the constructor the attributes `renderCounter` and `actionCounter` will be set to zero, but then it does nothing, and, in fact, this is completely normal for a portlet implementation.

Initializing the Portlet

The container is required to initialize the portlet once it has been loaded and instantiated.

Although there's nothing to prevent you from doing useful initialization in the constructor, the configuration information isn't available to you until the `init()` method is called. As well as simplifying the implementation of the container, this allows the complete API of the portlet to be defined by the `Portlet` interface (interfaces don't allow the signature of the constructor to be specified):

```
public void init(PortletConfig config) throws PortletException
```

The `init()` method is passed an object implementing the `PortletConfig` interface. This object will be unique to the portlet definition and provides access to the initialization parameters and the `ResourceBundle` configured for the portlet in the portlet definition.

The `init()` method on a portlet instance is called only once by a portlet container.

Until the `init()` method has been invoked successfully, the portlet will not be considered active, so static initialization of the class should not trigger any methods that make this assumption. For example, the static (class rather than object scope) initializers of your class should not invoke connections to a database.

In our example, the `init()` method increases the number of portlet instances noted in the `portletCounter` attribute:

```
public void init(PortletConfig config)
            throws PortletException
{
            portletCounter++;
}
```

In a larger application, this method would be populated with code to extract configuration information in order to establish resources such as database connections and background threads. Our crawler example in the final section will demonstrate the initialization of a background thread in its `init()` method.

Exceptions During Initialization

The initialization process is error-prone. You are likely to be making connections to resources that may be unavailable and over which you have no control. For example, your database server might be unavailable. Without a mechanism for handling such errors, your portlet could end up in an invalid state. As you would expect, the usual exception-handling mechanism comes into play here.

The `init()` method is permitted to throw a `PortletException`. If it does so, the container is allowed to reattempt to load the portlet at any later time.

When constructing an `UnavailableException`, the portlet can provide a message describing the problem. In this case, the portlet must not be restarted. Use this exception if a configuration setting of the portlet will have to be changed to get the portlet work to properly. For instance, the portlet may require version 9 of a database to connect to, but the database it tried connecting to was version 7.

```
public UnavailableException(String text)
```

For example:

```
throw new UnavailableException("The database has been decommissioned");
```

Alternatively, you can specify a minimum period of time (in seconds) during which no attempt must be made to restart the portlet:

```
public UnavailableException(String text, int seconds)
```

For example:

```
throw new UnavailableException("The database is not currently available",5);
```

If the duration of the resource unavailability cannot be determined but is still considered to be a temporary problem, the portlet should return a zero or negative time.

For example:

```
throw new UnavailableException("A website resource could not be reached",0);
```

If any other `PortletException` is thrown, the container is allowed to attempt to restart the portlet at any time after the error. The container may either reuse the original instance, or discard the original and re-create it.

If a portlet needs to throw an exception from its initialization method, it must free any resources that it successfully acquired up to that point before doing so—this is because the `destroy()` method will not subsequently be called, as the portlet is considered to be uninitialized.

Request Handling

Once the portlet has been initialized, it is waiting for interactions with the users of the portal.

The container translates requests from the users of the portal into invocations of the `render()` and `processAction()` methods, thus elegantly breaking down the user requests into actions that command the portlet to change the state of its underlying application and render requests that display the application in its current state at any given point.

Users trigger actions by clicking on action URLs or submitting HTML forms that post data to an action URL. Upon receiving the action request from the user, the portal must invoke (via the container) the appropriate portlet's `processAction()` method. Once this method has completed, it must call the `render()` method for all of the portlets on the page. It is not required to invoke the `render()` methods in any particular order.

Users trigger render requests by either triggering action URLs as described previously, or by triggering a render URL. Again, upon receiving a request for a render URL from a user, the portal must invoke the `render()` method on all of the portlets in the page but is not obliged to follow any particular order.

The only exception to the invocations of the `render()` method as described is (optionally) for portlets that are cached by the portal and for which the state has not changed.

Since a single portlet is generally handling requests from multiple users, it must be able to handle simultaneous requests on different threads in each of these methods. In addition, it must not rely on any particular ordering of the calls to these methods.

Because the portlet cannot maintain all of the state information for a session, it is the container's responsibility to manage this and provide it when these methods are called.

Both action requests and render requests are similar, but each takes different classes for arguments. The action request cannot write any content to the portlet's response, because its `ActionResponse` does not have access to the output. The signatures of the two methods are very similar. First, here is the `processAction()` method signature:

```
public void processAction(
   ActionRequest request,
   ActionResponse response)
throws PortletException, IOException
```

And here is the `render()` method signature:

```
public void render(
   RenderRequest request,
   RenderResponse response)
throws PortletException, IOException
```

Each method receives a request object and a response object tailored to its function. In each case, the request represents the state of the session for the user, and the response object allows the method to interact with the portlet's response.

The `RenderRequest` object will not generally need to change the state of the underlying portlet application, so it provides the portlet with the information necessary to produce a view of it in its current state.

Specifically, these include

- The state of the portlet window (minimized, maximized, etc.)
- The mode of the portlet (e.g., VIEW mode)
- The context of the portlet
- The session associated with the portlet (including authorization information)
- The preferences information associated with the portlet
- Any render parameters that have been set on a render URL from a posted Form, or that have been set during the `processAction()` method

The `ActionRequest` object represents an opportunity to change the state of the portlet based on its current state, so this provides everything offered by the `PortletRequest` along with direct access to the content of the HTTP request made by the user of the portal.

Note that `ActionRequest` and `RenderRequest` are both interfaces, so it is the responsibility of the container to provide concrete implementation classes giving access to the appropriate information

To respond to `processAction()` the portlet should update its `ActionResponse` object. This provides methods to

- Redirect the client to a new page
- Change the mode of the portlet
- Add or modify rendering parameters for the user's session

You change the state of the portlet's container window in the portal. To respond to `render()`, the portlet should update its `RenderResponse` object. This provides methods to

- Render content into the container window displayed in the user's view of the portal
- Render URLs into that content, which will invoke actions on the portlet

Again, `ActionResponse` and `RenderResponse` are interfaces, and the container must provide a suitable implementation to be used by the portlet.

Here is our sample `processAction()` method:

```
public void processAction(
         ActionRequest request,
         ActionResponse response)
      throws PortletException, IOException
   {
      actionCounter++;
   }
```

The example `processAction()` makes only a trivial change to the state of the portlet; it increments the counter of actions handled, so it does not have to inform the container of any changes via the response.

Our sample `render()` method looks like this:

```
public void render(
         RenderRequest request,
         RenderResponse response)
      throws PortletException, IOException
   {
      renderCounter++;

      response.setTitle("Simple Portlet");
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.write("The server has instantiated " +
            portletCounter +
            " copies of the portlet<br>");
```

```
        out.write("This portlet has been rendered " +
              renderCounter +
              " times (including this one)<br>");

        out.write("This portlet has received " +
              actionCounter +
              " action requests<br>");

        PortletURL action = response.createActionURL ();
        out.write("Click <a href=\"");
        out.write(action.toString());
        out.write("\">here</a> to trigger an action.<br>");
    }
```

Because our sample portlet does not need to tailor its view to the different users of the system, it is able to ignore the request parameter, which contains the user-specific (session) state information. It does, however, need to render its current state to the browser, and must specify the type of content that it will produce.

Our sample portlet demonstrates an important relationship between the portlet and the portal: one portlet can be rendered multiple times on a single page. If an instance of the portlet is placed in a portal page in two distinct places, the portlet will be loaded once, rendered multiple times (twice each time the portal page as a whole is rendered), and destroyed once.

The last few lines of this example method are of particular interest, since they demonstrate how to provide a mechanism by which actions (and thus calls by the container to `processAction()`) can be rendered:

```
PortletURL action = response.createActionURL ();
```

This retrieves an object from the response object provided in the parameters of the `render()` method, which can render a URL representing a specific action. Our example has only one type of action, but methods such as `addParameter()` can be called on the `PortletURL` object to differentiate between calls to the various actions that you want to implement.

Again, `PortletURL` is an interface, and it is the responsibility of the container to provide a suitable implementation.

It is not appropriate to hardcode a URL into your portlet since the precise details of the mappings between URLs and portlets are configurable by the administrator of the portal. In addition, portlet URLs distinguish between different instances of a portlet running inside of a portlet. The portlet URLs are usually prefixed with a namespace or another unique ID. These details will be specific to the portal in which your portlet is running, so if you want to make your portlet compatible between portals (and even between versions of the same portal) you should always rely on the `createActionURL()` method.

Destroying the Portlet

The `destroy()` method will not be invoked until all other initialization or processing threads on the instance have completed. It will be invoked when the container determines that the portlet is no longer required—the container is not required to keep the portlet in service for any specific period of time.

The container invokes the `destroy()` method to release any resources that have been retained by the portlet. The portlet will then be de-referenced by the container and the garbage collector will be free to remove the portlet object from memory.

The `destroy()` method is guaranteed to be called (unless initialization failed with an exception), so this is also an appropriate place in which to notify other parts of the application that the portlet is becoming unavailable.

Finalizers should not be used since their invocation is not guaranteed.

Destroying our example portlet looks like this:

```
public void destroy() {
   portletCounter--;
}
```

Our example portlet uses the `destroy()` method to reduce the count of its running instances in the class attribute `portletCounter`.

Threading Issues

In this final section, we demonstrate the life cycle of a portlet that uses background threads of execution in a web crawler application.

Handling Concurrent Requests

Because the portlet container will handle concurrent requests from clients by invoking the methods on the portlet on separate threads of execution, your portlet must be able to handle any combination and number of simultaneous calls to `render()` and/or `processAction()`.

You must therefore implement your portlet to handle these concurrent requests safely. In practice this is not usually too tricky—all the information you need to process a request is provided in a thread-safe manner in the parameter list, so if your portlets don't use instance variables and they don't access other resources external to the portlet, your application will automatically be thread-safe.

It is guaranteed that your `init()` method will be called only once at the beginning of the life cycle and that no other methods will be invoked by the container until `init()` completes successfully, so your `init()` method does *not* have to be thread safe.

Your `render()` and `processAction()` methods will be invoked with request and response objects. These are guaranteed to be unique to that invocation of the method *during the lifetime of the method*. Containers are likely to recycle these objects once the method in question has completed, so retaining a reference to them outside the scope of the method to which they were passed may result in unexpected behavior.

Our Thread-Safe Crawler

Our crawler class is implemented to be thread safe. It implements `Runnable` so it can be created and started within a background thread.

Once the crawler is running, it can be queried at any time. The `get` methods return unmodifiable sets so it is not possible for the client to externally alter their state.

The crawler can be stopped by an external thread by calling the `stopCrawler()` method. This is essential so that our portlet can be unloaded safely.

Our crawler implementation follows, with a running commentary. Although this illustrates the functionality that's needed in a web crawler, you should note that it is a demonstration application only. We make a lot of assumptions and take shortcuts that would not be acceptable in a commercial product.

```
package com.portalbook.crawler;

import java.io.*;
import java.net.*;
import java.util.*;

public class Crawler
   implements Runnable
{
```

Our simplest constructor creates an instance to crawl a given path. This will search only within the host of the path specified:

```
   public Crawler(String path)
      throws MalformedURLException
   {
      this(path,DEFAULT_LINK_DEPTH);
   }
```

This more complex constructor creates an instance to crawl a given path. It will search within the host of the path specified, and up to the specified number of sites (`depth`) away. If `depth` is 2, the crawler will look within the host of the path specified and within the hosts of sites referenced directly from this site, but no further:

```
public Crawler(String path, int depth)
   throws MalformedURLException
{
   this(new HashSet(
         Arrays.asList(
            new Object[]
               { new URL(path) })),
        new HashSet(),
        new HashSet(),
        new HashSet(),
        new HashSet(),
        depth);
}
```

This internal constructor is used directly or indirectly by the public ones to create the instance to crawl a given set of paths. It will search within the hosts specified, and up to the specified number of sites away from those sites. It will not search forbidden hosts, failed hosts, or already visited hosts:

```
protected Crawler( Set links,
                   Set visited,
                   Set visitedHosts,
                   Set forbidden,
                   Set failed,
                   int depth )
{
   this.links        = links;
   this.visited      = visited;
   this.visitedHosts = visitedHosts;
   this.forbidden    = forbidden;
   this.failed       = failed;
   this.currentHost  =
      (URL)links.iterator().next();
   this.depth        = depth;
}
```

Our crawler is designed to run as a thread, so it implements the `Runnable` interface. It can therefore be passed in as a parameter to a new `Thread` object.

When the `start()` method of the containing `Thread` object is called, the following `run()` method will be started on the background thread of execution:

```
public void run() {
```

This code flags that work is in progress:

```
    setStopped(false);
    try {
```

until we instruct the thread to stop, or it runs out of links to search, or it reaches the edge of the network of links that we're allowing it to search:

```
        while( !isStopped() &&
                (links.size() > 0) &&
                (depth > 0) ) {
```

This code gets the first link from the queue of links to search:

```
            URL link =
                (URL)links.iterator().next();
```

If the link is on a different host

```
            if( !isCurrentHost(link) ) {
```

this code goes to that host and searches the link (if we're allowed):

```
                crawlNewHost(link);
            } else {
```

This code crawls the link:

```
                crawl(link);
            }
        }
```

Normal or abnormal termination should flag that work has completed regardless:

```
    } finally {
        setStopped(true);
    }
}
```

TIP *Most of the work of the crawler is carried out from within private methods. As with all good object-oriented designs, these hide the implementation details from the user of the class—this is particularly important with a multi-threaded design, since you will need to carefully isolate any code that might cause problems if two different threads were to execute it simultaneously.*

This private method processes a buffer containing the contents of a robots.txt file and adds forbidden URLs to the appropriate list. The link for the robots.txt file is required to convert the relative disallowed paths into absolute URLs:

```
private void processRobotBuffer(
      StringBuffer buffer,
      URL link)
{
```

Now we prepare to gather up potential URL strings:

```
   List disallows = new ArrayList();
```

and look for `DISALLOW` tokens. Any token immediately following a `DISALLOW` is presumed to be (potentially) a path:

```
   StringTokenizer tokenizer =
      new StringTokenizer(buffer.toString());
   while(tokenizer.hasMoreElements()) {
      String element = tokenizer.nextToken();

      if( element.equalsIgnoreCase(DISALLOW) &&
          tokenizer.hasMoreElements() ) {

         String path = tokenizer.nextToken();
         disallows.add(path);
      }
   }
```

The following code iterates over the disallow tokens gathered and converts them into absolute paths and then URLs to forbid access:

```
   Iterator i = disallows.iterator();
   while(i.hasNext()) {
      String path = (String)i.next();
      try {
         URL disallowedURL = new URL(link,path);
         forbidden.add(disallowedURL);
```

```
            } catch( MalformedURLException e ) {
               // Couldn't form a URL from this.
               // No point disallowing access to a
               // link we can't access anyway.
            }
         }
      }
```

This private method determines the robots.txt file that governs access to the host on which the link resides, parses, and then disallows access to links as required by the robots specification; however, we're polite and assume that *any* link that's disallowed to *anybody* must be forbidden to us. This makes the logic slightly simpler but would not normally be done in a production system. The robots.txt information is cached with the host as the key, so we look it up only when we encounter the first link from a site:

```
   private void processRobots(URL link) {
```

If the host has been visited before, we're not obliged to reread the robots.txt file. If this is the first visit, however, we need to determine which paths to discard:

```
      HttpURLConnection connection = null;
      try {
         if( !visitedHosts.contains(link.getHost()) ) {
```

This code creates an HTTP connection to the link:

```
            URL robotLink = new URL(link,"/robots.txt");

            connection =
               (HttpURLConnection)robotLink.openConnection();
```

This code gets the page:

```
            connection.setRequestMethod(GET);
            connection.setRequestProperty(
                  USER_AGENT,AGENT_IDENTIFIER);
```

And this reads the page into a buffer:

```
            InputStream input =
               (InputStream)connection.getContent();
            BufferedReader reader =
               new BufferedReader(
                  new InputStreamReader(input));

            int i = 0;
            StringBuffer buffer = new StringBuffer();
            String line = null;
            while((line = reader.readLine()) != null) {
               i++;
               buffer.append(line);
               buffer.append('\r');
            }
```

We now close the connection to the page and discard held resources:

```
            reader.close();
            connection.disconnect();
            connection = null;
```

Next we process the buffer to determine what links are permitted:

```
            processRobotBuffer(buffer,link);
         }
      } catch( IOException e ) {
```

Good. There is no robots.txt file, and we're allowed to view the site.
To ensure that the connection is always closed correctly, we use the following:

```
      } finally {
         if( connection != null ) {
            connection.disconnect();
         }
      }
   }
```

This private method crawls a specified URL and adds all valid HREF entries to the queue of links to be crawled (unless they are denied by the appropriate robots.txt file or have already been crawled):

```
   private void crawl(URL link) {
      HttpURLConnection connection = null;
      try {
```

Now let's check to see what is or isn't permitted on this host:

```
         processRobots(link);
```

This code creates an HTTP connection to the link:

```
connection =
   (HttpURLConnection)link.openConnection();
```

Now we get the page:

```
connection.setRequestMethod(GET);

String contentType = connection.getContentType();
if( (contentType != null) &&
    contentType.startsWith(TEXT_HTML) ) {

   InputStream input =
      (InputStream)connection.getContent();
```

And read the page into a buffer:

```
   BufferedReader reader =
   new BufferedReader(new InputStreamReader(input));

   int i = 0;
   StringBuffer buffer = new StringBuffer();
   String line = null;
   while((line = reader.readLine()) != null) {
      i++;
      buffer.append(line);
   }
```

To close the connection to the page and discard held resources, we use this code:

```
   reader.close();
   connection.disconnect();
   connection = null;
```

The next step is to process the page. This code assumes that the page is small enough to manage in memory:

```
   processBuffer(buffer,link);
} else {
   connection.disconnect();
   connection = null;
}
```

We've handled the item, so let's remove it from the queue of links:

```
      visited(link);
   } catch( IOException e ) {
```

The item caused a problem, so let's remove it from the queue:

```
      links.remove(link);
      failed.add(link);
   } finally {
```

To ensure that the connection is closed correctly, we use the following:

```
      if( connection != null ) {
         connection.disconnect();
      }
   }
}
```

This private method carries out actions once a link has been crawled. It removes the link from the queue of links to visit, adds it to the set of visited links, and adds its host to the set of visited hosts:

```
private void visited(URL link) {
   links.remove(link);
   visited.add(link);
   visitedHosts.add(link.getHost());
}
```

This private method processes a buffer containing an HTML page. The context is provided so that relative URLs can be resolved to their absolute URLs. HREFs within the HTML are extracted, converted to absolute URLs, and added to the queue of links to be crawled (if they have not already been visited and have not been forbidden by a robots.txt file):

```
private void processBuffer(StringBuffer buffer, URL url) {
```

Let's now prepare to gather up potential URL strings:

```
   List foundHREFs = new ArrayList();
```

And look for HREF tokens. Any token immediately following an HREF is presumed to be (potentially) a URL:

```
   StringTokenizer tokenizer =
      new StringTokenizer(buffer.toString(),DELIM);
```

```
while(tokenizer.hasMoreElements()) {
   String element = tokenizer.nextToken();

   if( element.equalsIgnoreCase(HREF) &&
       tokenizer.hasMoreElements() ) {

      String path = tokenizer.nextToken();
      foundHREFs.add(path);
   }
}
```

Now let's boil them down to absolute URLs:

```
Set absolute = new HashSet();
Iterator i = foundHREFs.iterator();
while(i.hasNext()) {
   String path = (String)i.next();

   try {
      URL toAdd = new URL(url,path);
      if( !toAdd.getProtocol().equalsIgnoreCase("http") ) {
```

If it's not a link beginning with "http" then it's not a protocol we're interested in:

```
      } else {
         absolute.add(toAdd);
      }
   } catch( MalformedURLException e ) {
```

If we encounter a `MalformedURLException`, then the crawler must have tried to load an invalid path. We should ignore the URL as it's probably a typo:

```
   }
}
```

Let's now remove all the URLs that we're not allowed to visit for one reason or another:

```
absolute.removeAll(forbidden);
```

And remove all the URLs that we've already visited anyway:

```
absolute.removeAll(visited);
```

We'll add the remainder to the visit queue:

```
      links.addAll(absolute);
   }
```

The following private method crawls a link that is on a host other than the one currently being processed. The simplest way to effect this is to instantiate a new crawler specifying the appropriate URL, but to pass it the information on visited and forbidden hosts so that already crawled links can be ignored. Note that we reduce the permitted depth by 1 for this next crawler so that it can't creep too far away from the original link. Eventually we'll reach 0, and there's no point in instantiating the crawler because it would simply complete immediately.

```
   private void crawlNewHost(URL url) {
      Set links = new HashSet();
      links.add(url);
      if( depth > 1 ) {
```

Now let's create a new crawler to crawl the external link:

```
         Crawler crawler =
            new Crawler(links,
                        visited,
                        visitedHosts,
                        forbidden,
                        failed,
                        (depth - 1));
```

The new crawler should be stopped whenever this crawler is stopped:

```
         addListener(crawler);
```

This code flags that the URL has been visited:

```
         crawler.run();
      }

      this.links.remove(url);
   }
```

This private method adds a crawler to a set of crawlers that are children of this crawler so that they can all by stopped when this one is stopped:

```
   private void addListener(Crawler crawler) {
      stopListeners.add(crawler);
   }
```

This private method stops all the crawlers that are children of this crawler:

```
private void notifyStopCrawlers() {
   Iterator i = stopListeners.iterator();
   while(i.hasNext() && depth > 0) {
      Crawler crawler = (Crawler)i.next();
      crawler.setStopped(true);
   }
}
```

This method allows the owner of the crawler to test to see if the crawler is currently running:

```
public boolean isStopped() {
   return stopped;
}
```

This method allows the owner to stop the crawler:

```
public void stopCrawler() {
   setStopped(true);
}
```

The following private method sets the flag that causes the crawler to stop, and if it's set to `true`, it instructs any child crawler threads to stop as well:

```
private void setStopped(boolean stopped) {
   if(stopped) notifyStopCrawlers();
   this.stopped = stopped;
}
```

The next method allows the owner of the crawler to retrieve the set of links (as URL objects) that the crawler has encountered so far and that were valid. Access to the set of visited URLs is synchronized, and a copy is returned, so that it is not possible for two threads to try to access the set simultaneously. By pursuing this policy for all of the publicly accessible stores of data provided by the crawler, we make it thread safe with very little use of synchronization. Judicious use of the `synchronized` keyword is important because it carries a significant performance penalty. While this isn't a big deal for our crawler (which is not limited by processing speed so much as by bandwidth), getting good responsiveness is often the primary goal when using threads:

```
public Set getVisitedURLs() {
   synchronized(this.visited) {
      return new HashSet(this.visited);
   }
}
```

Similarly, this code allows the owner to retrieves the set of hosts (as `Strings`) that the crawler has encountered so far:

```
public Set getVisitedHosts() {
   synchronized(this.visitedHosts) {
      return new HashSet(this.visitedHosts);
   }
}
```

The following code retrieves the set of links as URLs that the crawler was forbidden to search by their associated robots.txt files:

```
public Set getForbiddenURLs() {
   synchronized(this.forbidden) {
      return new HashSet(this.forbidden);
   }
}
```

Next we retrieve the set of invalid links as URLs that the crawler could not search (usually because of a network problem of some sort, such as "404 Host Not Found"):

```
public Set getFailedURLs() {
   synchronized(this.failed) {
      return Collections.unmodifiableSet(this.failed);
   }
}
```

This private method determines if a given link falls within the set of links being crawled by this crawler's instance:

```
private boolean isCurrentHost(URL url) {
   return currentHost.getHost().equalsIgnoreCase(url.getHost());
}
```

This field identifies the host that this crawler is searching:

```
private URL currentHost;
```

These fields are used to stop this thread and its child threads:

```
private Set stopListeners = new HashSet();
private boolean stopped = true;
```

These fields maintain information on where we've been so far:

```
   private Set links;
   private Set visited;
   private Set visitedHosts;
   private Set forbidden;
   private Set failed;
   private int depth;
```

The following are String constants required by the application mostly during tokenization or protocol negotiation:

```
   private static final String DELIM      =  "\t\r\n\" <>=";
   private static final String TEXT_HTML  =  "text/html";
   private static final String GET        =  "GET";
   private static final String HREF       =  "HREF";
   private static final String HTTP       =  "HTTP:";
   private static final String DISALLOW   =  "DISALLOW:";
   private static final String USER_AGENT =  "User-Agent";
```

This is the "official" name of this agent for identification in the logs of the servers we're searching:

```
   private static final String AGENT_IDENTIFIER="WoolGatherer";
```

Finally, we specify the default search depth to be used if the single-argument constructor is used:

```
   private static final int DEFAULT_LINK_DEPTH = 1;
}
```

Here is the portlet.xml file for the crawler portlet:

```
<?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>Link Crawler</description>

      <portlet-name>crawler</portlet-name>
      <display-name>Link Crawler</display-name>
      <portlet-class>
         com.portalbook.crawler.CrawlerPortlet
      </portlet-class>

      <init-param>
         <name>crawlPath</name>
         <value>http://localhost:8080/</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>Link Crawler Portlet</title>
         <short-title>Crawler</short-title>
         <keywords>Crawler, Link, Checker</keywords>
      </portlet-info>

   </portlet>
</portlet-app>
```

The GenericPortlet Abstract Class

Portlets are allowed to be displayed in the three standard modes—VIEW, EDIT, and HELP. They do not have to support any of them except VIEW. Many do not require any action handling. This results in a lot of boilerplate code to handle the requests for each of these modes and render the response.

The `GenericPortlet` abstract class therefore provides an initial framework around which you can build your custom portlet. At its simplest, a portlet derived from `GenericPortlet` must implement a constructor, a `getTitle()` method, and a `doView()` method. `GenericPortlet` then provides your application with a variety of helper methods that you will make frequent use of. It is generally recommended that you override `GenericPortlet` rather than directly creating your own implementation of the `Portlet` interface.

`GenericPortlet`'s `render()` method calls the `doDispatch()` method. This method determines the current mode of the portlet render request, and calls `doView()`, `doEdit()`, and `doHelp()` as appropriate (the modes that are permissible for a portlet are configured in its deployment descriptor). The `processAction()` method throws an exception if invoked, so your class must override this if you wish to handle any actions.

Since our crawler portlet provides only a view method and does not accept any actions, `GenericPortlet` provides the perfect starting point.

Our Threaded Crawler Portlet

Our crawler overrides `GenericPortlet`; this will simplify the implementation of our very conventional design.

The constructor is empty:

```
public CrawlerPortlet() {
   super();
}
```

The `init()` method retrieves the path from which the crawler will start from the portlet configuration:

```
String path = (String)config.getInitParameter("crawlPath");
```

It then creates a new crawler object, and invokes it on a background thread:

```
crawler = new Crawler(path);
Thread background = new Thread(crawler);
background.start();
```

Our code does not need to retain a reference to the thread object, since this will not be manipulated directly, but it does retain a reference to the crawler object, since it will need to access its methods in order to retrieve information and ultimately stop the crawler.

When the portlet is eventually unloaded by the container, the `destroy()` method will be invoked:

```
public void destroy() {
   crawler.stopCrawler();
   crawler = null;
}
```

This method cleans up the portlet's resources by stopping the crawler. The crawler's thread will exit, and no further action will be necessary (the container will manage the removal of the portlet from memory).

When running, the portlet may be required to render itself into its container and then onto the portal itself. This process would typically be triggered by the user browsing to a page on which the portlet is configured to be displayed.

The `render()` method of the `Portlet` interface will be invoked. This is caught by the `GenericPortlet` implementation, and it instead sets the title (which it determines by invoking `getTitle()`) and then invokes `doDispatch()`. The default implementation gets the title from the `ResourceBundle` of the `PortletConfig` of the portlet, but we will override this to return a specific string without requiring configuration:

```
   protected String getTitle(RenderRequest request) {
      return "Link Crawler";
   }
```

`doDispatch()` is provided as a default implementation by `GenericPortlet`. It determines the mode of the portlet (VIEW, EDIT, or HELP) and then calls the `doView()`, `doEdit()`, or `doHelp()` method as appropriate in order to render the portlet in its appropriate mode.

Our sample provides only a VIEW mode, so all render requests must result in a call to `doView()`:

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

      out.write("<table><tr><td>");
      out.write("<h2>Crawler</h2>");

      out.write("<table cellspacing=\"0\"")
      out.write("border=\"1\">");
      out.write("<tr><td align=\"right\">")
      out.write("<i>Status</i></td><td><b>");
      out.write(
         crawler.isStopped() ? "stopped" : "running");
      out.write("</b></td></tr>");
      out.write("</table>");

      renderCollection(out, "Hosts Crawled",
         crawler.getVisitedHosts());

      renderCollection(out, "Links Visited",
         crawler.getVisitedURLs());

      renderCollection(out, "Failed Links",
         crawler.getFailedURLs());
```

```
        renderCollection(out, "Forbidden Links",
           crawler.getForbiddenURLs());

        out.write("</td></tr></table>");
}
```

Whenever `doView()` is invoked, we render a simple set of tables demonstrating the current state of the crawler running in the background. Appropriate headers for the response are rendered, and then the output print writer is retrieved from the response object and all output is generated using this.

Both of our examples have used inline HTML to render output to the portlet window. Although this is straightforward, you can see with this example that it rapidly makes the structure of the page difficult to follow. In Chapter 5, we discuss the option of delegating the rendering of the page to a servlet or JavaServer Pages (JSP) page instead.

The full implementation of our threaded portlet follows. Because our crawler implementation is thread safe, we have only to ensure that we start the crawler thread in the `init()` method, and stop it in the `destroy()` method, and that we don't start any additional threads during the lifetime of our portlet.

Here is the threaded crawler portlet in full:

```
package com.portalbook.crawler;

import java.io.*;
import java.net.*;
import java.util.*;
import javax.portlet.*;

public class CrawlerPortlet
   extends GenericPortlet
{
   public CrawlerPortlet() {
      super();
   }

   public void init(PortletConfig config)
   throws PortletException
   {
      String path =
         (String)config.getInitParameter("crawlPath");

      try {
         crawler = new Crawler(path);

         // Here we create and kick off the background
         // thread of execution.
```

```
            Thread background = new Thread(crawler);
            background.start();

        } catch( MalformedURLException e ) {
            throw new PortletException(
                "Portlet could not be initialised",e);
        }
    }

    public void destroy() {
        // Here we ensure that the background
        // thread is terminated safely.
        crawler.stopCrawler();
        crawler = null;
    }

    private void renderCollection(
            PrintWriter out,
            String title,
            Collection collection )
        throws IOException
    {
        out.write("<table cellspacing=\"0\"");
        out.write("border=\"0\">");
        out.write("<tr><td><b>");
        out.write(title);
        out.write("</b></td></tr>");

        Iterator i = collection.iterator();
        while(i.hasNext()) {
            out.write("<tr><td>");
            out.write(i.next().toString());
            out.write("</td></tr>");
        }

        out.write("</table><br>");
    }

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

        out.write("<table><tr><td>");
        out.write("<h2>Crawler</h2>");

        out.write("<table cellspacing=\"0\"");
        out.write("border=\"1\">");
        out.write("<tr><td align=\"right\">");
        out.write("<i>Status</i></td><td><b>");
        out.write(
           crawler.isStopped() ? "stopped" : "running");
        out.write("</b></td></tr>");
        out.write("</table>");

        // Note that while we can access an object running
        // in a background thread from the doView method,
        // we must not create a new thread of execution here.
        renderCollection(out, "Hosts Crawled",
           crawler.getVisitedHosts());

        renderCollection(out, "Links Visited",
           crawler.getVisitedURLs());

        renderCollection(out, "Failed Links",
           crawler.getFailedURLs());

        renderCollection(out, "Forbidden Links",
           crawler.getForbiddenURLs());

        out.write("</td></tr></table>");
     }

     protected String getTitle(RenderRequest request) {
        return "Link Crawler";
     }

     private Crawler crawler;
  }
```

Summary

In this chapter, we have discussed the life cycle of a portlet. The portlet container calls the `init()` method on the portlet to initialize the portlet, the portlet handles any requests from the user, and the portlet is destroyed. The user can send either action or render requests, and we discussed the order in which these requests are handled. We also discussed error handling from the portlet's initialization phase.

Our web crawler example demonstrates how a portlet can create and manage a background thread. We discussed which methods on the portlet need to be made thread safe, and which objects are guaranteed to be unique during processing.

In the next chapter, we discuss the concepts that underlie the design of the portlet API and that account for the specifics of the portlet life cycle that we have just described.

CHAPTER 6

Packaging and Deployment Descriptors

Portlet application assemblers need to package each portlet application into a web application archive (WAR) file for distribution and deployment. Each portlet application contains a web and a portlet deployment descriptor, and we discuss the formats used for each in this chapter. Several tools assist with the portlet assembly and packaging process. We briefly discuss the Ant build tool, and cover the XDoclet portlet extensions in more detail.

Portlet Application Packaging

The portlet application consists of the portlet classes, any libraries or resources, a web application deployment descriptor, and a portlet application deployment descriptor. Both deployment descriptors are contained in the portlet application's WEB-INF directory.

The application assembler packages the portlet application into a WAR file. The portlet application's WAR file structure is identical to that of a standard Java 2 Enterprise Edition (J2EE) web application, with the addition of a portlet.xml portlet application deployment descriptor and the portlet classes.

Versioning

Versioning each release of a portlet application can help you manage your portlet distributions. If you are releasing the WAR file to other groups or other organizations, the version can help you track down specific bugs for a release. You can also provide other information about the portlet application implementation, such as the title and vendor name.

The versioning standard for a portlet application WAR archive is the same one used for Java Archive (JAR) files. The versioning information belongs in the WEB-INF/MANIFEST.MF file in the WAR archive. There are six different pieces of metadata that describe the versioning for a portlet application: title, vendor, and version both for the implementation and for the specification. It is up to you to specify the titles, versions, and vendors. Some portals may have tools that track these versions for you, and they can be useful if there is no other version information in the WAR file.

Here is an example MANIFEST.MF file for a portlet application:

```
Manifest-Version: 1.0
Name: DocumentManagementPortlet
Specification-Title: WebDAV
Specification-Vendor: WebDAV
Specification-Version: 1.0
Implementation-Title: DocumentationManagementPortlet
Implementation-Vendor: PortalBook.com
Implementation-Version: 0.9.3
```

Portlet Application Deployment Descriptor Structure

An XML Schema named portlet-app_1_0.xsd specifies the structure of the portlet application deployment descriptor. Your portal should include a copy of this schema, either as documentation or for deployment descriptor documentation. The full schema is too large to reprint in this book (over 15 pages), but we used Altova's XML Spy XML editor to generate diagrams from the schema. We discuss each element in the schema, and give a pointer to the relevant chapter of this book. Future versions of the portlet API may extend or change elements of this schema, so when you upgrade to a portal that supports a newer version, check the release notes to see what changed.

portlet-app

The root element of the portlet application deployment descriptor is the `<portlet-app>` element. The `<portlet-app>` element contains any portlet definitions, custom portlet modes or window states, the supported user attributes for the portlet application, and the security constraints. This element represents a distinct portlet application that is self-contained and can be deployed on a portal with no dependencies. Figure 6-1 shows the XML Schema for this element.

There are two attributes on the `<portlet-app>` element. The first attribute is named `version`, and it is required. The value of the `version` attribute is the version of the portlet API that this portlet application supports. The deployment descriptor must be valid with that version of the portlet application deployment descriptor schema, but it may be invalid with future releases. Until a new version of the portlet API is released, this value will be 1.0. The other attribute, `id`, is optional.

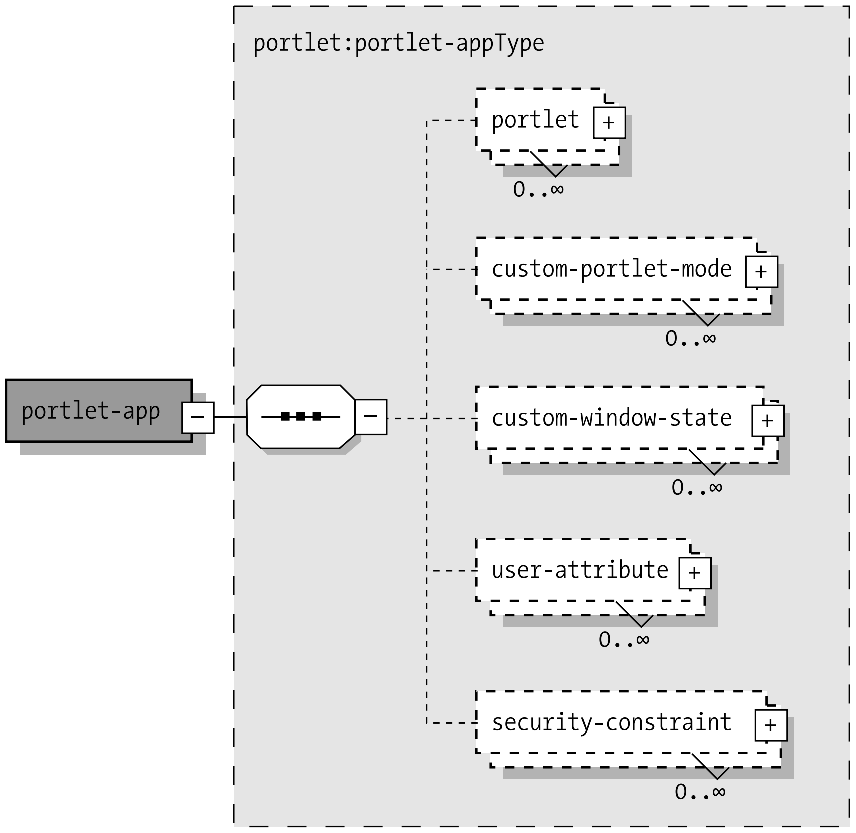

Figure 6-1. The `<portlet-app>` *XML Schema*

The `<portlet-app>` element may contain `<portlet>` elements, which represent portlet classes, `<custom-portlet-mode>` elements, `<custom-window-state>` elements, `<user-attribute>` elements, or `<security-constraint>` elements.

```
<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>
 </portlet-app>
```

portlet

The `<portlet>` element (see Figure 6-2) represents a portlet class and all of its metadata. The only attribute on the `<portlet>` element is `id`, and it is optional.

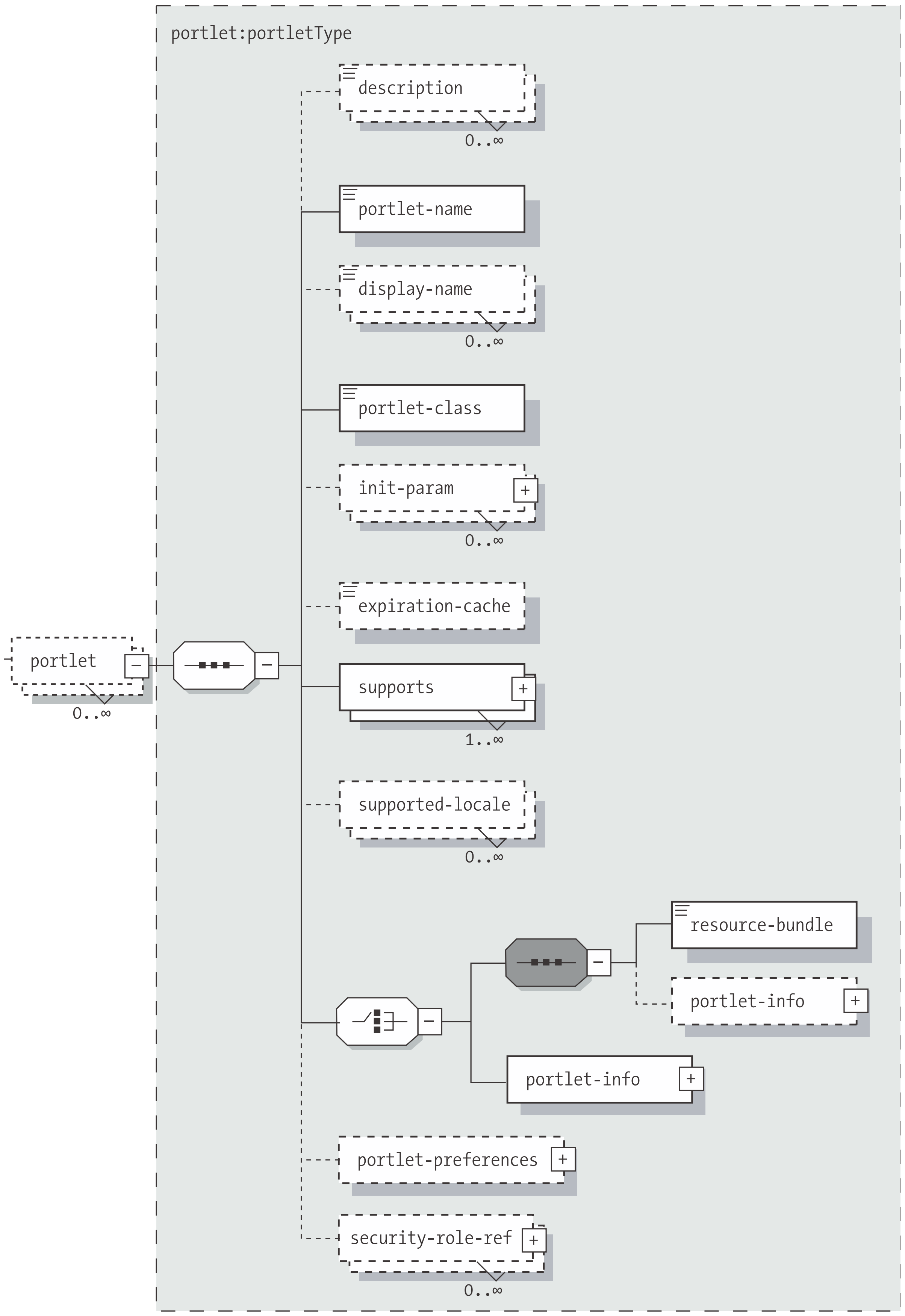

Figure 6-2. The `<portlet>` *XML Schema*

Each portlet may have an optional description, which is specified in the `<description>` element. Portal administration tools use the description to give some context about the portlet to a portlet application deployment.

TIP *The* `<description>` *element is a child of many of the tags in the portlet deployment descriptor. Its use is optional, but it is very handy for documentation or portlet deployment tools.*

The `<portlet-name>` element is required for the portlet. Each portlet's name in the portlet application has to be unique. This name should not contain any spaces or non-web-friendly special characters.

The `<display-name>` element provides a human-readable name for portal administration tools. It is optional.

The class name of the portlet belongs in the `<portlet-class>` element, which takes a fully qualified Java class name, as shown here:

```
    <portlet>
        <description>Preferences Validation Portlet</description>

        <portlet-name>PreferencesValidationPortlet</portlet-name>

        <display-name>Preferences Validation Portlet</display-name>

        <portlet-class>
com.portalbook.portlets.PreferencesValidationPortlet</portlet-class>
        ....
     </portlet>
```

Each portlet may have zero or more portlet initialization parameters, which are used to provide configuration information for all users. The `<init-param>` element (see Figure 6-3) has three child elements: `<description>`, `<name>`, and `<value>`. The `<name>` and `<value>` elements are required. For more on initialization parameters, see Chapter 7.

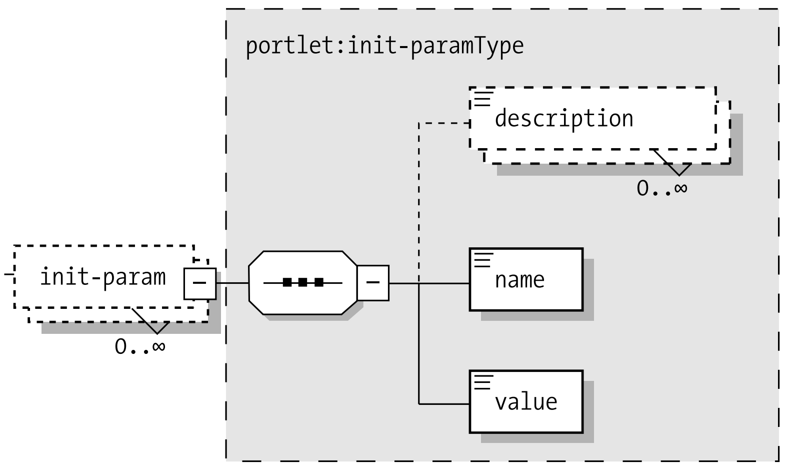

Figure 6-3. The `<init-param>` *XML Schema*

```
<init-param>
  <name>indexPath</name>
  <value>/java/index</value>
</init-param>
<init-param>
  <name>repository</name>
  <value>engineering</value>
</init-param>
```

The portlet container may cache the output of a portlet. The portlet defines the timeout it expects for the cache in the `<expiration-cache>` element. If the value is -1, the cached output is always valid:

```
<expiration-cache>0</expiration-cache>
```

The portlet may support more than one Multipurpose Internet Mail Extensions (MIME) type, and for each MIME type, the portlet needs to tell the portlet container which portlet modes are valid. These could be any of the standard portlet modes (VIEW, EDIT, HELP), or custom modes that are specified later in the deployment descriptor. Each MIME type should be specified only once, and there must be at least one MIME type for each portlet. The `<supports>` element (see Figure 6-4) groups MIME types, `<mime-type>`, and portlet modes, `<portlet-mode>`. The portlet modes should be a comma-delimited list of valid portlet modes. For more on portlet modes, see Chapter 4.

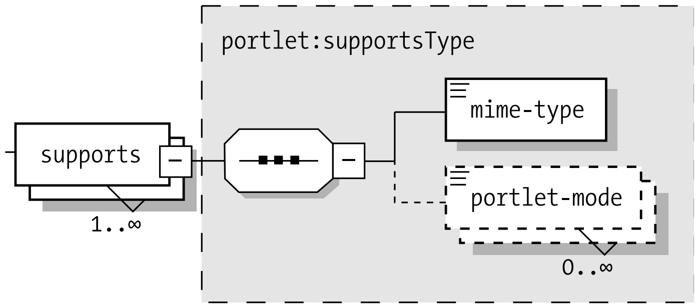

Figure 6-4. The `<supports>` *XML Schema*

```
<supports>
  <mime-type>text/html</mime-type>
  <portlet-mode>edit</portlet-mode>
  <portlet-mode>help</portlet-mode>
  <portlet-mode>view</portlet-mode>
</supports>
```

You may specify which locales the portlet supports with the `<supported-locale>` element. The element should contain the name of a valid Java locale. There can be more than one supported locale for a portlet. The portal may use these values to localize content for the end user:

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

The portlet metadata can also be localized with a resource bundle. The resource bundle should contain the information from the `<portlet-info>` tag (Figure 6-5), specified as javax.portlet.title, javax.portlet.short-title, and javax.portlet.keywords. The `<resource-bundle>` tag should contain the class name of the resource bundle. The bundle should be in the portlet application's classpath:

```
<resource-bundle>com.portalbook.Messages</resource-bundle>
```

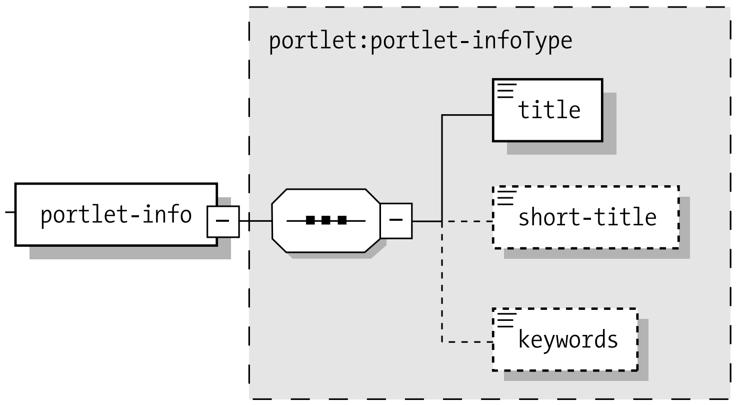

Figure 6-5. The `<portlet-info>` *XML Schema*

The `<portlet-info>` element represents the portlet metadata. There are three XML child elements that contain information: `<title>`, `<short-title>`, and `<keywords>`. The portlet's title bar uses the value in the `<title>` element, although some portlets will change the title dynamically. The short title is for mobile phones, PDAs, or other portal clients that do not have a lot of room for a title to display.

```
<portlet-info>
  <title>Taxonomy Portlet</title>
  <short-title>Taxonomy</short-title>
  <keywords>Taxonomy,Lucene</keywords>
</portlet-info>
```

The `<portlet-preferences>` element (Figure 6-6) contains zero or more `<preference>` elements and an optional `<preferences-validator>` element. The `<preferences-validator>` value should be the class name for a validator class. The `<preference>` element has a child `<name>` element, which is required, and zero or more optional initial values. There is also a read-only flag for preferences that cannot be modified. The preference value must be set in the deployment descriptor.

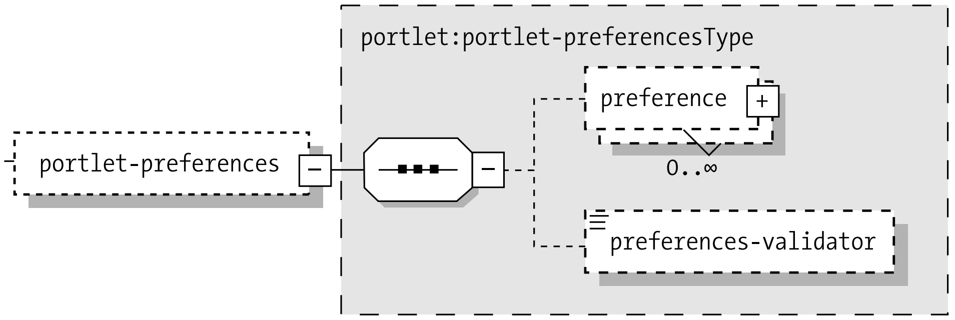

Figure 6-6. The `<portlet-preferences>` *XML Schema*

Each portlet preference name must be unique for the portlet. You may have portlet preferences with the same name for two or more portlets in a portlet application. For more on portlet preferences, see Chapter 7.

```
<portlet-preferences>
  <preference>
    <name>bookmark</name>
    <value>/content/marketing</value>
  </preference>

  <preferences-validator>
    com.portalbook.portlets.TaxonomyValidator
  </preferences-validator>
</portlet-preferences>
```

The <security-role-ref> element (Figure 6-7) maps portlet security roles to web application security roles. For more on portlet application security, see Chapter 8.

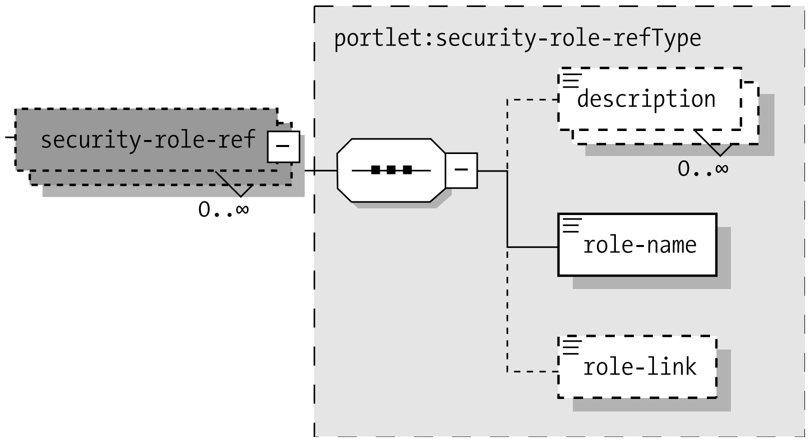

Figure 6-7. The <security-role-ref> *XML Schema*

```
<security-role-ref>
  <role-name>Administrator</role-name>
  <role-link>admin</role-link>
</security-role-ref>
```

custom-portlet-mode

The <custom-portlet-mode> element (Figure 6-8) defines a custom portlet mode that this portlet supports. Each portlet application can have as many custom portlet modes as it needs. The <custom-portlet-mode> element has a <portlet-mode> element that contains the name of the portlet mode, such as PRINT. The <description> element is optional.

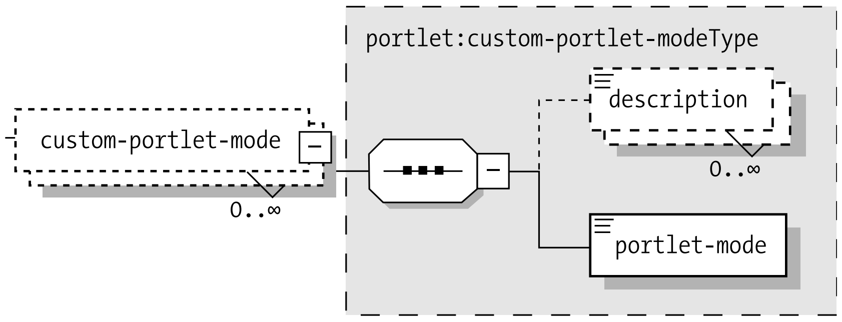

Figure 6-8. The <custom-portlet-mode> *XML Schema*

For more on custom portlet modes, see Chapter 4.

```
<custom-portlet-mode>
  <portlet-mode>PRINT</portlet-mode>
</custom-portlet-mode>
```

custom-window-state

The `<custom-window-state>` element (Figure 6-9) defines a custom window state supported by this portlet. The portlet application can have zero or more custom window states. The `<custom-window-state>` element has a `<window-state>` element that contains the name of the window state, such as ICON. The `<description>` element is optional.

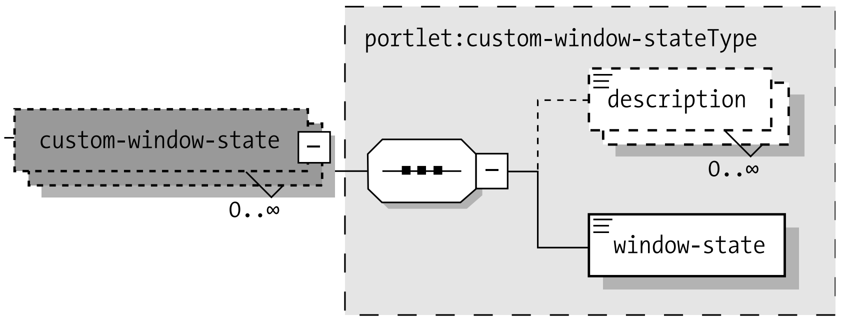

Figure 6-9. The `<custom-window-state>` *XML Schema*

For more on custom window states, see Chapter 4.

```
<custom-window-state>
  <window-state>docked</window-state>
</custom-window-state>
```

user-attribute

The portlet application can access information about the user, but the requested attributes must be explicit in the portlet deployment descriptor. Each user attribute the portlet requires needs to be defined in portlet.xml as a `<user-attribute>` element (Figure 6-10). The user attributes are optional, and there may be an unlimited number of them.

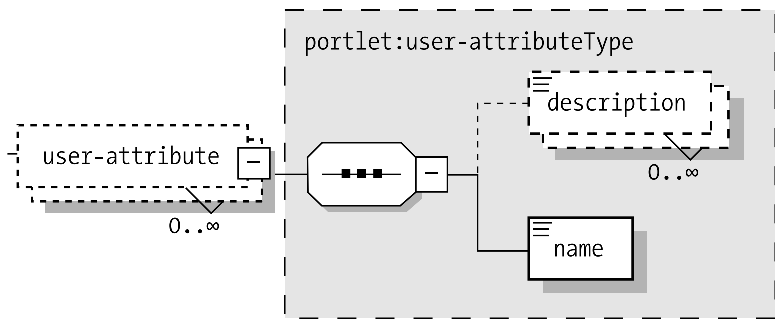

Figure 6-10. The `<user-attribute>` *XML Schema*

The `<user-attribute>` tag has a `<name>` child element, which must be unique for the portlet application. The `<description>` element is optional.

For more on user attributes, see Chapter 11.

```
<user-attribute>
  <name>user.name.given</name>
</user-attribute>
<user-attribute>
  <name>user.name.suffix</name>
</user-attribute>
<user-attribute>
  <name>user.home-info.postal.country</name>
</user-attribute>
```

security-constraint

The `<security-constraint>` element (Figure 6-11) has an optional display name, a portlet collection, and a user data constraint. The `<portlet-collection>` element has a set of `<portlet-name>` elements that reference portlets defined in the portlet application. The `<user-data-constraint>` element has a `<transport-guarantee>` element and an optional description.

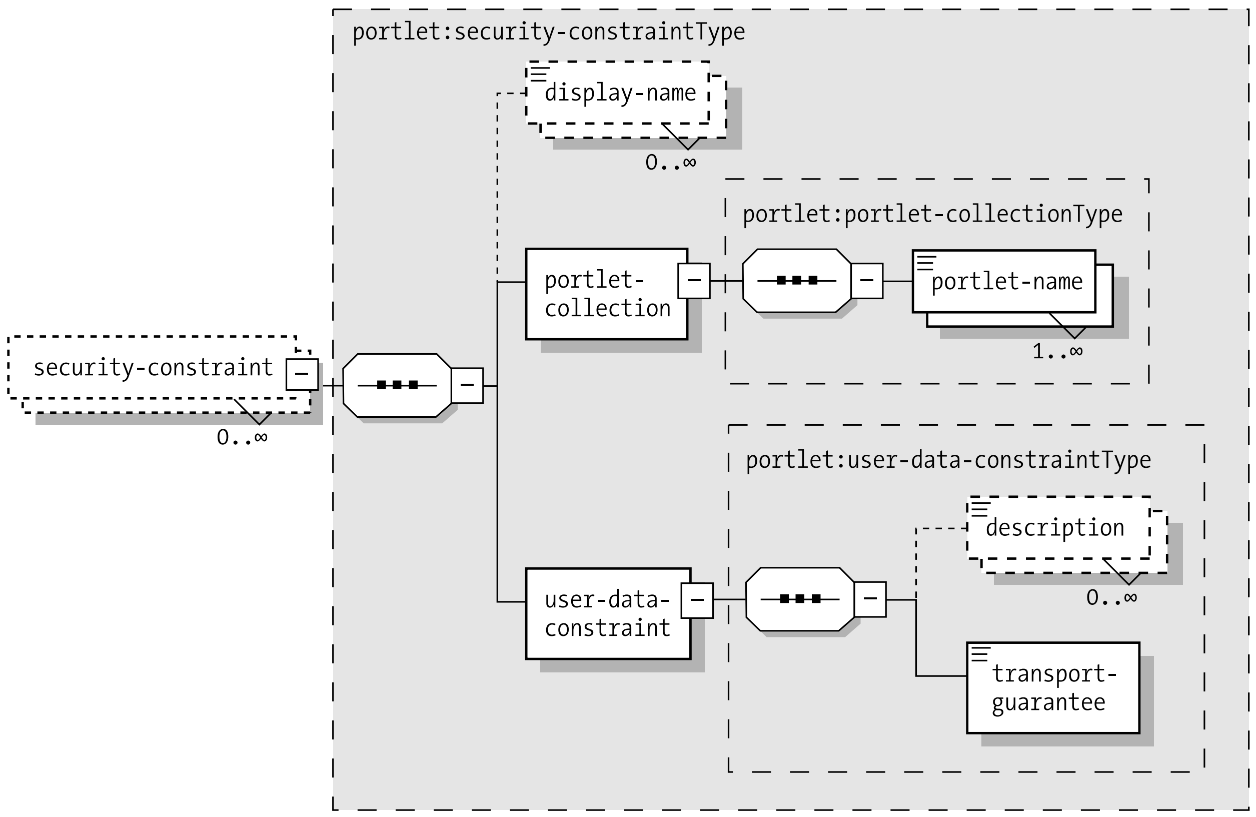

Figure 6-11. The `<security-constraint>` *XML Schema*

The `<transport-guarantee>` element may have a value of NONE, CONFIDENTIAL, or INTEGRAL. If the value is NONE, the portlets in the portlet collection do not require any transport-level security. The CONFIDENTIAL guarantee means that the data transmitted between the user and the portal cannot be seen by third parties. The INTEGRAL guarantee requires that another party cannot alter traffic between the user and the portal.

```
<security-constraint>
  <portlet-collection>
    <portlet-name>TaxonomyPortlet</portlet-name>
  </portlet-collection>
  <user-data-constraint>
    <transport-guarantee>INTEGRAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

Web Application Deployment Descriptor

Each portlet application needs a web application deployment descriptor named web.xml that resides in the WEB-INF directory. Three portlet application settings are common to the web application and belong in web.xml. The description should be specified in the `<description>` element. The `<display-name>` tag contains the name of the portlet application. Portlet security is controlled with the `<security-role>` tag, although the portlet application deployment descriptor also contains the `<security-role-ref>` tag for mapping roles.

In addition, any servlets or JSP pages descriptions belong in the web.xml file. We covered servlets and JSP in Chapter 5.

Some portlet containers will rewrite your portlet application's web.xml file as part of the deployment process, but this should not affect your portlet development. The following code snippet is from a web.xml deployment descriptor created during the Apache Pluto deployment process:

```
...
 <servlet>
   <servlet-name>ContentPortlet</servlet-name>
   <display-name>ContentPortlet Wrapper</display-name>
   <description>Automated generated Portlet Wrapper</description>
   <servlet-class>org.apache.pluto.core.PortletServlet</servlet-class>
   <init-param>
     <param-name>portlet-guid</param-name>
     <param-value>myapp.ContentPortlet</param-value>
   </init-param>
   <init-param>
     <param-name>portlet-class</param-name>
     <param-value>com.portalbook.portlets.content.ContentPortlet</param-value>
   </init-param>
 </servlet>
   ...
```

The web.xml deployment descriptor was generated from the values in the portlet.xml deployment descriptor.

XDoclet Portlet Support

XDoclet is an open source Java code-generation tool, with an Apache-style license. The XDoclet home page is located at `http://xdoclet.sourceforge.net/`. You need to download and install the XDoclet 1.2 package from SourceForge (`http://sourceforge.net/projects/xdoclet/`) to run the examples in this section. You also need Apache Ant 1.5 or greater (`http://ant.apache.org`) to run XDoclet. If you are not familiar with Ant, the online Apache Ant manual (`http://ant.apache.org/manual/index.html`) is a good place to start, as is *Enterprise Java Development on a Budget* by Christopher M. Judd and Brian Sam-Bodden (Apress, 2004).

We are going to use XDoclet to generate the portlet.xml deployment descriptor from our Java portlet class source code. Craig Walls wrote the portlet deployment descriptor integration for XDoclet, which speeds up the portlet application development process. To use XDoclet, you need to mark up your portlet classes with custom JavaDoc tags. XDoclet processes the JavaDoc tags and creates the deployment descriptor. The advantage is that the portlet description information belongs with the source code for the portlet class, so when you create or modify your portlet, it is easy to change the deployment descriptor information. For instance, if the class or package name changes, the generated deployment descriptor will contain the new name without any additional work on your part.

TIP *Keep the XDoclet-generated files out of source control. Re-create any generated files during the build process, because they will be derived from source files that may have changed.*

After working through the XDoclet example in this section, you should have some idea of whether using XDoclet is worth integrating into your build process. The portlet application deployment descriptor is not so complicated that automating its generation with XDoclet saves a lot of time on a normal project. For complex development projects with many portlet applications and portlet classes, standardization on XDoclet makes more sense.

NOTE *The main reason most developers use XDoclet is for generating Enterprise JavaBeans (EJB) classes. XDoclet cuts down on the number of Java classes that must be created and maintained for each EJB, so it can be a real time-saver. If you are interested in using XDoclet for EJB code generation, check out* Enterprise Java Development on a Budget *(Apress, 2004).*

We will show you how to set up a portlet class with XDoclet tags and then how to create an Ant build file that will generate a portlet.xml deployment descriptor. If you had more than one portlet class in your project with XDoclet tags, XDoclet will add all of them to the portlet.xml deployment descriptor.

XDoclet Tags for Portlets

We are going to reuse the `SessionPortlet` class from Chapter 4 for our XDoclet example. None of the Java code in the portlet needs to be modified for XDoclet support; we only have to add the XDoclet portlet tags to the class.

The first XDoclet tag we will use for `SessionPortlet` will be `@portlet.portlet`. This tag represents one portlet class in the portlet application. There are four parameters on the `@portlet.portlet` tag: `description`, `display-name`, `expiration-cache`, and `name`. These parameters correspond to several of the child elements of the

`<portlet>` tag in the deployment descriptor. The `name` parameter is required, and it corresponds to the `<portlet-name>` tag. The name of the portlet must be unique within the web application.

Here is the source code to the marked-up `SessionPortlet` class, with the `@portlet.portlet` tag. Notice that the XDoclet tag looks like the `@author` and `@version` JavaDoc tags in the source code. The `@portlet.portlet` tag takes up to four parameters, which can be specified on the same line as the tag name, or spread out over multiple lines. Each parameter has a name and a value—for this tag, all of the values are plain text.

```
package com.portalbook.xdoclet;

import java.io.IOException;
import java.io.PrintWriter;

import javax.portlet.GenericPortlet;
import javax.portlet.PortletException;
import javax.portlet.PortletSession;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;

/**
 * XDoclet example portlet
 *
 * @portlet.portlet
 *          description="This portlet demonstrates the use of the portlet session."
 *          display-name="Session Example"
 *          expiration-cache="0"
 *          name="SessionPortlet"
 *
 * @portlet.supports
 *          mime-type="text/html"
 *          modes="VIEW"
 *
 * @author Jeff Linwood and David Minter
 * @version 1.0
 */
public class SessionPortlet extends GenericPortlet
{
    public void doView(RenderRequest req, RenderResponse resp)
        throws PortletException, IOException
    {
        String newMessage = null;
        //set up for output
        resp.setContentType("text/html");
        PrintWriter writer = resp.getWriter();

        //get the session, or create it if needed
        PortletSession session = req.getPortletSession();

        //if there is already a value in the session, get it
        Object message = session.getAttribute("message");
        if (message == null)
        {
            //This is nothing in the session stored by the name 'message'
            newMessage = "Hi, This is the first visit to the portlet.";

        }
        else if (message instanceof String)
        {
            //change the message for repeat visitors
            newMessage = "Welcome back to this portlet!";
        }

        //Store it in the portlet session
        session.setAttribute("message", newMessage);

        //write it out
        writer.write(newMessage);
    }
}
```

As you can see, we now specify some of the metadata about the portlet in the Java class. We do not have to maintain this metadata in two different places; instead, we can just generate the portlet.xml deployment descriptor from the Java class.

Creating the Ant Build File with XDoclet Support

We've updated one of our standard portlet application Ant build files with XDoclet support. XDoclet can be run only from an Ant build file, because XDoclet is implemented as a set of Ant tasks. Here is a list of several of the available XDoclet Ant tasks:

- **EJBDoclet:** Used for generating local interfaces, home interfaces, application-server specific deployment files, and Struts/EJB integration code
- **WebDoclet:** Used for generating a web application deployment descriptor, JSP tag lib TLD files, struts-config.xml files for Struts, and servlet container-specific deployment files

- **HibernateDoclet:** Generates the bean to database mapping files for the open source persistence layer Hibernate

- **PortletDoclet:** Creates the portlet.xml deployment descriptor for a portlet application

To use any of these XDoclet tasks in an Ant build file, you will need to define the task in the build file with the `<taskdef>` element. We will use `<taskdef>` in our Ant build file to define the `<portletdoclet>` task. The `<portletdoclet>` task will use the `xdoclet.modules.portlet.PortletDocletTask` class from the XDoclet 1.2.x distribution. You will need to have downloaded and unzipped the XDoclet 1.2.x binary distribution somewhere on your file system so we can load the task. The portlet XDoclet task also requires the Java portlet API JAR file (portlet-api-1.0.jar) in the task's classpath. In our build file, we set up a classpath called xdoclet.portlet.classpath that references the JAR files in our XDoclet distribution and the portlet API JAR file. You will need to edit this classpath to point to these locations on your machine.

The `<portletdoclet>` task takes several attributes, the most important of which is `destdir`. The `destdir` attribute specifies the directory where XDoclet will generate the portlet.xml file. By default, XDoclet will not overwrite existing files, but if you set the `<portletdoclet>` element's `force` attribute to `true`, the new portlet.xml will overwrite an old portlet.xml. The `mergedir` attribute is the location of any additional XML files that will be merged into the generated portlet.xml. The portlet XDoclet supports custom window states, custom portlet modes, user attributes, and security constraints in the portlet deployment descriptor, if they are included in the XDoclet merge directory as XML files.

You will need to specify a set of Java source files for XDoclet to use to generate the portlet.xml. The `<portletdoclet>` task needs an Ant file set for the project's Java source code files. These can be anywhere, but in our Java build file, we used the regular source directory, with all Java classes. If your portlets are in one or two packages, or all of your portlet class names end in Portlet, you can use a more specific filter for the Ant file set include. Alternatively, you may also exclude certain classes from the file set if you have portlets with XDoclet markup that are not going to be deployed on the portal in this build. This could be useful in a scenario where you make different portlet application builds for production servers and development machines with the same Ant build file and source control repository.

The only subtask for the `<portletdoclet>` task is the `<portletxml>` subtask. The `<portletxml>` subtask will generate the portlet.xml from the XDoclet tags in the Java source code. Behind the scenes, it is filling in an XDoclet template with the correct information. The XDoclet template for portlet.xml is called portlet_xml.xdt, and it is available in the XDoclet source distribution or in the XDoclet portlet module JAR file. You do not ever need to edit this template, but if you are curious about how XDoclet works, the template is easy to understand.

The create-portlet-xml target is a dependency of our default Ant target, build-war, and it runs after compilation. That way, if our compilation fails, the build file does not generate a portlet.xml:

```
<?xml version="1.0" encoding="UTF-8"?>

<project default="build-war" name="xdoclet-build" basedir=".">
  <property file="build.properties"/>
  <property name="src.dir" value="src"/>
  <property name="build.dir" value="build"/>
  <property name="dist.dir" value="dist"/>
  <property name="classes.dir" value="${build.dir}/classes"/>
  <property name="lib.dir" value="lib"/>
  <property name="web-inf.dir" value="WEB-INF"/>
  <property name="web-inf.generated.dir" value="generated"/>
  <property name="war.name" value="xdocletexample"/>

  <property name="xdoclet.merge.dir" value="merge"/>

  <path id="xdoclet.portlet.classpath">
    <fileset dir="${xdoclet.install.dir}/lib">
        <include name="*.jar"/>
    </fileset>
        <fileset dir="${tomcat.shared.lib.dir}">
          <include name="*.jar"/>
        </fileset>
  </path>

  <target name="init">
    <mkdir dir="${build.dir}"/>
    <mkdir dir="${classes.dir}"/>
    <mkdir dir="${dist.dir}"/>
    <mkdir dir="${lib.dir}"/>
    <mkdir dir="${web-inf.generated.dir}"/>
  </target>

  <target name="compile" depends="init">
    <javac destdir="${classes.dir}" deprecation="true" debug="true" optimize="false">
      <src>
        <pathelement location="${src.dir}"/>
      </src>
      <classpath>
        <fileset dir="${lib.dir}">
          <include name="*.jar">
          </include>
        </fileset>
        <fileset dir="${tomcat.shared.lib.dir}">
          <include name="*.jar">
          </include>
        </fileset>
      </classpath>
    </javac>
  </target>

  <target name="create-portlet-xml">
    <taskdef name="portletdoclet"
             classname="xdoclet.modules.portlet.PortletDocletTask"
             classpathref="xdoclet.portlet.classpath"
     />

     <portletdoclet destdir="${web-inf.generated.dir}"
                    mergedir="${xdoclet.merge.dir}"
                    force="true"
     >
         <fileset dir="${src.dir}">
            <include name="**/*.java"/>
         </fileset>

         <portletxml/>
     </portletdoclet>
  </target>

  <target name="copy-portlet-xml">
    <copy file="${web-inf.generated.dir}/portlet.xml"
          todir="${web-inf.dir}"
     />
  </target>

  <target name="build-war" depends="compile,create-portlet-xml,copy-portlet-xml">
    <war destfile="${dist.dir}/${war.name}.war" webxml="WEB-INF/web.xml">
        <classes dir="${classes.dir}"/>
        <lib dir="${lib.dir}"/>
        <webinf dir="${web-inf.dir}"/>
    </war>
  </target>

  <target name="clean">
    <delete dir="${build.dir}"/>
    <delete dir="${dist.dir}"/>
    <delete dir="${web-inf.generated.dir}"/>
  </target>
</project>
```

You will need to edit the build.properties file referenced at the top of this Ant build file to correspond to your installation directories for the Tomcat shared library folder where you have the portlet API JAR (this could be any directory, not just Tomcat). You also need to update the XDoclet installation directory reference. Here is our example build.properties:

```
xdoclet.install.dir=/java/xdoclet
tomcat.shared.lib.dir=/java/tomcat/shared/lib
```

If the portlet API JAR is not in the task's classpath, the portlet XDoclet task will throw an error when the build is run. The API is necessary because XDoclet checks to see if the annotated class is an implementation of the `javax.portlet.Portlet` interface.

Running the Ant Build File

Once you have edited the build file to point to your portlet API JAR and XDoclet directory, you can run Ant to generate your portlet.xml and package your WAR file.

On a Windows XP machine with JDK 1.4 and Ant 1.5.1, the output looks like this:

```
C:\apress\packaging>ant -verbose -f xdoclet-build.xml create-portlet-xml

Apache Ant version 1.5.1 compiled on October 2 2002
Buildfile: xdoclet-build.xml
Detected Java version: 1.4 in: C:\java\j2sdk1.4.1_01\jre
Detected OS: Windows XP
parsing buildfile xdoclet-build.xml with URI = file:C:/apress/packaging/xdoclet-
build.xml
Project base dir set to: C:\apress\packaging
Build sequence for target 'create-portlet-xml' is [create-portlet-xml]
Complete build sequence is [create-portlet-xml, clean, init, compile, copy-portl
et-xml, build-war]

create-portlet-xml:
[portletdoclet] (XDocletMain.start               47  ) Running <portletxml/>

[portletdoclet] Generating portlet.xml.

BUILD SUCCESSFUL
Total time: 3 seconds
```

Here, we ran Ant in verbose mode to show extra information. We also set the `verbose` attribute to have the value "true" on the `<portletdoclet>` task in the Ant build file, but that did not give us any extra information about the portlet.xml generation.

Generated Portlet.xml Deployment Descriptor

The generated portlet.xml deployment descriptor follows. As you can see, the generated XML has placeholders for the custom portlet modes, custom window states, user attributes, and security constraints:

```
<?xml version="1.0" encoding="UTF-8"?>

<portlet-app 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" ➥
  xmlns="http://java.sun.com/xml/ns/portlet">

    <portlet>
      <description>This portlet demonstrates the use of the portlet session.
      </description>
      <portlet-name>SessionPortlet</portlet-name>
      <display-name>Session Example</display-name>
      <portlet-class>com.portalbook.xdoclet.SessionPortlet</portlet-class>

      <expiration-cache>0</expiration-cache>
      <supports>
        <mime-type>text/html</mime-type>
        <portlet-mode>VIEW</portlet-mode>
      </supports>

      <portlet-info>
        <title></title>
      </portlet-info>

      <portlet-preferences>

      </portlet-preferences>
    </portlet>
     <!--
       To add custom portlet modes to the portlet.xml file, add a file to your
       XDoclet merge directory called custom-portlet-modes.xml that contains the
       <custom-portlet-mode></custom-portlet-mode> markup.
     -->
       <!--
          To add custom window states to the portlet.xml file, add a file to your
         XDoclet merge directory called portlet-custom-window-states.xml that contains the
          <custom-window-state></custom-window-state> markup.
       -->

       <!--
          To add user attributes to the portlet.xml file, add a file to your
          XDoclet merge directory called portlet-user-attributes.xml that contains the
          <user-attribute></user-attribute> markup.
       -->
       <!--
          To add security constraints to the portlet.xml file, add a file to your
          XDoclet merge directory called portlet-security.xml that contains the
          <security-constraint></security-constraint> markup.
       -->

  </portlet-app>
```

NOTE *There is a bug in XDoclet 1.2 with the portlet.xml generation module. The XML Schema location is not specified correctly, and the generated portlet.xml will not deploy properly on Pluto (and probably most other portlet containers). If this is not fixed in a release when this book goes to print, we will make a working XDoclet portlet module available from our web site (*`www.portalbook.com/`*).*

The generated portlet.xml file in this chapter uses the correct XML Schema location, `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"`.

Other Portlet XDoclet Tags

In addition to `@portlet.portlet`, the other XDoclet tags for portlets are

- `@portlet.portlet-info`
- `@portlet.portlet-init-param`
- `@portlet.preference`
- `@portlet.preferences-validator`
- `@portlet.security-role-ref`
- `@portlet.supports`

All of these tags are for a portlet class, and are specified in the same place in the source code as the `@portlet.portlet` tag. None of them is required. All of the values for all of the parameters should be plain text.

The `@portlet.portlet-info` tag has three optional parameters: `title`, `keywords`, and `short-title`. There can be only one or none of each of these tags.

The `@portlet.portlet-init-param` tag takes two required parameters: `name` and `value`. The `description` parameter is optional. You may have zero or more portlet initialization parameters as XDoclet tags.

The `@portlet.preference` tag specifies portlet preferences. Any number of preferences is allowed, including none. There are two required parameters: `name` and `value`. The `read-only` parameter is optional.

There can be zero or one `@portlet.preferences-validator` XDoclet tags. This tag only requires one parameter, `class`, which is a Java class name.

The `@portlet.security-role-ref` tag has two required parameters: `role-link` and `role-name`. The `description` parameter is optional. A class may have zero or more of these tags.

The `@portlet.supports` tag specifies portlet mode to MIME type mappings. The tag has two required parameters: `mime-type` and `modes`. The `modes` parameter is a comma-delimited list of portlet modes. There may be zero or more of these tags in a class.

Extended XDoclet Example

We used the preceding XDoclet portlet tags to build a sample generated portlet.xml file for a taxonomy portlet. Here is the JavaDoc comment that belongs at the top of the class:

```
/**
 * @portlet.portlet
 *   name="TaxonomyPortlet"
 *   description="Browse Organized Content"
 *   display-name="Taxonomy Portlet"
 *   expiration-cache="0"
 *
 * @portlet.portlet-info
 *   title="Taxonomy Portlet"
 *   keywords="Taxonomy,Lucene"
 *   short-title="Taxonomy"
 *
 * @portlet.portlet-init-param
 *   name="indexPath"
 *   value="/java/index"
 *
 * @portlet.portlet-init-param
*   name="repository"
*   value="engineering"
*
* @portlet.preference
*   name="bookmark"
*   value="/content/marketing"
*
* @portlet.preferences-validator
*   class="com.portalbook.portlets.TaxonomyValidator"
*
* @portlet.security-role-ref
*   role-link="admin"
*   role-name="Administrator"
*
* @portlet.supports
*   mime-type="text/html"
*   modes="edit,help,view"
*
*/
```

Here is the generated portlet.xml deployment descriptor from those JavaDoc XDoclet tags:

```
<?xml version="1.0" encoding="UTF-8"?>

<portlet-app 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" ➥
xmlns="http://java.sun.com/xml/ns/portlet">

    <portlet>
      <description>Browse Organized Content</description>
      <portlet-name>TaxonomyPortlet</portlet-name>
      <display-name>Taxonomy Portlet</display-name>
      <portlet-class>com.portalbook.xdoclet.SessionPortlet</portlet-class>

      <init-param>
        <name>indexPath</name>
        <value>/java/index</value>
      </init-param>
      <init-param>
        <name>repository</name>
        <value>engineering</value>
      </init-param>
```

```
  <expiration-cache>0</expiration-cache>
  <supports>
    <mime-type>text/html</mime-type>
    <portlet-mode>edit</portlet-mode>
    <portlet-mode>help</portlet-mode>
    <portlet-mode>view</portlet-mode>
  </supports>

  <portlet-info>
    <title>Taxonomy Portlet</title>
    <short-title>Taxonomy</short-title>
    <keywords>Taxonomy,Lucene</keywords>
  </portlet-info>

  <portlet-preferences>
    <preference>
      <name>bookmark</name>
      <value>/content/marketing</value>
    </preference>

    <preferences-validator>
      com.portalbook.portlets.TaxonomyValidator
    </preferences-validator>
  </portlet-preferences>
  <security-role-ref>
    <role-name>Administrator</role-name>
    <role-link>admin</role-link>
  </security-role-ref>
</portlet>
 <!--
   To add custom portlet modes to the portlet.xml file, add a file to your
   XDoclet merge directory called custom-portlet-modes.xml that contains the
   <custom-portlet-mode></custom-portlet-mode> markup.
 -->
<!--
   To add custom window states to the portlet.xml file, add a file to your
   XDoclet merge directory called portlet-custom-window-states.xml
   that contains the <custom-window-state></custom-window-state>
   markup.
-->
```

```
    <!--
       To add user attributes to the portlet.xml file, add a file to your
       XDoclet merge directory called portlet-user-attributes.xml that contains the
       <user-attribute></user-attribute> markup.
    -->
    <!--
       To add security constraints to the portlet.xml file, add a file to your
       XDoclet merge directory called portlet-security.xml that contains the
       <security-constraint></security-constraint> markup.
    -->

</portlet-app>
```

Summary

Portlet application packaging is similar to web application packaging—both use web application archive (WAR) files and web.xml web application deployment descriptors. Portlet applications add the portlet.xml portlet application deployment descriptor and portlet classes.

XDoclet is a code-generation tool that generates the portlet application deployment descriptor from JavaDoc tags in the portlet source code. The `<portletdoclet>` XDoclet Ant task is straightforward, and the XDoclet tags in the portlet correspond to the elements in the deployment descriptor. The XDoclet portlet module also supports portlet application-wide deployment descriptor information: custom portlet modes, custom window states, user attributes, and security role references.