**JavaTM Portlet Specification**

**- Final -**

**Version 3.0**

Send comments about this document to: **issues@portletspec3.java.net**

22.12.16 06:10:18

Martin Scott Nicklous (Scott.Nicklous@de.ibm.com)

**Java(TM) Portlet Specification ("Specification") Version: 3.0 Status: Final, Specification Lead: IBM Corp.**

**Copyright 2016 IBM Corp. All rights reserved.**

IBM Corporation (the "Spec Lead") for the Portlet (JSR 362) Specification 3.0 (the "Specification") hereby grants permission to copy and display the Specification, in any medium without fee or royalty, provided that you include the following on ALL copies, or portions thereof, that you make:

1. A link or URL to the Specification at this location: http://jcp.org/en/jsr/detail?id=362.

2. The copyright notice as shown herein.

The Spec Lead offers to grant a perpetual, non-exclusive, worldwide, fully paid-up, royalty free, irrevocable license under its licensable copyrights and patent claims for which there is no technically feasible way of avoiding infringement in the course of implementing the Specification, provided that you:

(a) fully implement the Specification including all its required interfaces and functionality.

(b) do not modify, subset, superset or otherwise extend the Specification's name space;

(c) pass the TCK for this Specification; and

(d) grant a reciprocal license to any of your patents necessary to implement required portions of the Specification on terms consistent with the provisions of Section 6.A of the Java Specification Participation Agreement.

THE SPECIFICATION IS PROVIDED "AS IS," AND THE SPEC LEAD AND ANY OTHER AUTHORS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS. THE SPEC LEAD AND ANY OTHER AUTHORS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE SPECIFICATION OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of the Spec Lead or any other Authors may NOT be used in any manner, including advertising or publicity pertaining to the Specification or its contents without specific, written prior permission. Title to copyright in the Specification will at all times remain with the Authors.

No other rights are granted by implication, estoppel or otherwise.

Preface

This document is the JavaTM Portlet Specification, v3.0. that describes the JavaTM Portlet API.

Additional Sources

The specification is intended to be a complete and clear explanation of Java portlets, but if questions remain the following may be consulted:

* A reference implementation (RI) has been made available which provides a behavioral benchmark for this specification. Where the specification leaves implementation of a particular feature open to interpretation, developers may use the reference implementation as a model of how to carry out the intention of the specification
* A Technology Compatibility Kit (TCK) has been provided for assessing whether implementations meet the compatibility requirements of the JavaTM Portlet API standard. The test results have normative value for resolving questions about whether an implementation is standard
* If further clarification is required, the working group for the JavaTM Portlet API under the Java Community Process should be consulted, and is the final arbiter of such issues

Comments and feedback are welcomed, and will be used to improve future versions.

Who Should Read This Specification

The intended audience for this specification includes the following groups:

* Portal server vendors who want to provide portlet containers that conform to this standard
* Authoring tool developers who want to support web applications that conform to this specification
* Experienced portlet authors who want to understand the underlying mechanisms of portlet technology

We emphasize that this specification is not a user’s guide for portlet developers and is not intended to be used as such.

API Reference

The full specifications of classes, interfaces, and method signatures that define the Java Portlet API, as well as the accompanying Javadoc™ documentation, is available online.

Other Java™ Platform Specifications

The following Java API specifications are referenced throughout this specification:

* Java Platform, Enterprise Edition ("Java EE"), version 7
* Java™ Servlet™, v3.1
* JavaServer Pages™, v2.2 (JSP™)
* JavaServer™ Faces, v2.2 (JSF™)
* The Java™ Architecture for XML Binding (JAXB) 2.2
* Contexts and Dependency Injection for the Java EE Platform 1.1

These specifications may be found at the Java Community Process website: <https://www.jcp.org/en/home/index>

Other Important References

The following Internet specifications provide information relevant to the development and implementation of the Portlet API and standard portlet engines:

* RFC 1630 Uniform Resource Identifiers (URI)
* RFC 5646 BCP 47, Tags for Identifying Languages
* RFC 1738 Uniform Resource Locators (URL)
* RFC 2396 Uniform Resource Identifiers (URI): Generic Syntax
* RFC 1808 Relative Uniform Resource Locators
* RFC 1945 Hypertext Transfer Protocol (HTTP/1.0)
* RFC 2045 MIME Part One: Format of Internet Message Bodies
* RFC 2046 MIME Part Two: Media Types
* RFC 2047 MIME Part Three: Message Header Extensions for non-ASCII text
* RFC 2048 MIME Part Four: Registration Procedures
* RFC 2049 MIME Part Five: Conformance Criteria and Examples
* RFC 2109 HTTP State Management Mechanism
* RFC 2145 Use and Interpretation of HTTP Version Numbers
* RFC 7230 Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing
* RFC 7231 Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content
* RFC 7232 Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests
* RFC 7233 Hypertext Transfer Protocol (HTTP/1.1): Range Requests
* RFC 7234 Hypertext Transfer Protocol (HTTP/1.1): Caching
* RFC 7235 Hypertext Transfer Protocol (HTTP/1.1): Authentication
* RFC 2616 Hypertext Transfer Protocol (HTTP/1.1)
* RFC 2617 HTTP Authentication: Basic and Digest Authentication
* ISO 639 Code for the representation of names of languages
* ISO 3166 Code (Country) list
* OASIS Web Services for Remote Portlets (WSRP)
* CC/PP Processing, JSR 188
* W3C: Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies

Online versions of these RFC and ISO documents are at:

* http://www.rfc-editor.org/
* <http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt>
* http://www.iso.org/iso/en/prods-services/iso3166ma/index.html

The World Wide Web Consortium <http://www.w3.org/> is a definitive source of HTTP related information affecting this specification and its implementations.

The WSRP Specification can be found in the OASIS web site

<http://www.oasis-open.org/>.

The Extensible Markup Language (XML) is used for the specification of the portlet deployment descriptor described in Chapter 27 Packaging and Deployment, and the portlet deployment descriptor schema presented in Appendix E Deployment Descriptor Schema. More information about XML can be found at the following website: <http://www.xml.org/>

Terminology

The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL in this document are to be interpreted as described in [RFC2119].

Providing Feedback

We welcome any and all feedback about this specification. Please e-mail your comments to   
jsr362-observers@portletspec3.java.net.

Please note that due to the volume of feedback that we receive, you will not normally receive a reply from an engineer. However, each and every comment is read, evaluated, and archived by the specification team.

Acknowledgements

The Java Portlet Specification version 3.0 is the result of diligent efforts of the JSR 362 expert group working under the auspices of the Java Community Process (JCP).

Contents

[Preface iii](#_Toc468260913)

[Additional Sources iii](#_Toc468260914)

[Who Should Read This Specification iii](#_Toc468260915)

[API Reference iii](#_Toc468260916)

[Other Java™ Platform Specifications iii](#_Toc468260917)

[Other Important References iv](#_Toc468260918)

[Terminology v](#_Toc468260919)

[Providing Feedback v](#_Toc468260920)

[Acknowledgements v](#_Toc468260921)

[Chapter 1 Overview 1](#_Toc468260922)

[1.1 What is a Portal? 1](#_Toc468260923)

[1.2 What is a Portlet? 1](#_Toc468260924)

[1.3 What is a Portlet Container? 1](#_Toc468260925)

[1.4 Client-Side Support 2](#_Toc468260926)

[1.5 An Example 2](#_Toc468260927)

[1.6 Render State and Backend State 2](#_Toc468260928)

[1.7 Compatibility 2](#_Toc468260929)

[1.8 Relationship to Java Enterprise Edition 3](#_Toc468260930)

[1.9 Configuration 3](#_Toc468260931)

[Chapter 2 Relationship to the Servlet Specification 4](#_Toc468260932)

[2.1 Bridging from Portlets to Servlets/JSPs 5](#_Toc468260933)

[2.2 Using Servlet Application Lifecycle Events 5](#_Toc468260934)

[2.3 The Servlet Container - Portlet Container Relationship 6](#_Toc468260935)

[Chapter 3 Portlet Concepts 7](#_Toc468260936)

[3.1 Portlets 7](#_Toc468260937)

[3.2 Portal Pages 8](#_Toc468260938)

[3.3 The Portlet Container 9](#_Toc468260939)

[3.4 Typical Processing Sequence 10](#_Toc468260940)

[3.5 Serving Resources 10](#_Toc468260941)

[3.6 Execution Stages and Phases 12](#_Toc468260942)

[3.6.1 The Preparation Stage 12](#_Toc468260943)

[3.6.2 The Aggregation Stage 13](#_Toc468260944)

[3.6.3 The Resource Stage 14](#_Toc468260945)

[3.7 Portlet Request Processing Phases 14](#_Toc468260946)

[3.7.1 Render Phase 14](#_Toc468260947)

[3.7.2 Header Phase 15](#_Toc468260948)

[3.7.3 Action Phase 15](#_Toc468260949)

[3.7.4 Event Phase 15](#_Toc468260950)

[3.7.5 Resource Phase 16](#_Toc468260951)

[3.8 Portlet URLs 16](#_Toc468260952)

[3.9 Portlet Parameters and Render State 17](#_Toc468260953)

[3.9.1 Portlet Parameters 17](#_Toc468260954)

[3.9.2 The Window State 17](#_Toc468260955)

[3.9.3 The Portlet Mode 17](#_Toc468260956)

[3.9.4 The Render State 18](#_Toc468260957)

[3.10 Portlet Stage Execution 18](#_Toc468260958)

[3.11 Client-Side Support 19](#_Toc468260959)

[3.12 Portlet Lifecycle Methods 19](#_Toc468260960)

[3.13 Portlets and Web Frameworks 20](#_Toc468260961)

[Chapter 4 Portlet Lifecycle Interfaces 21](#_Toc468260962)

[4.1 Portlet Identification 21](#_Toc468260963)

[4.2 Portlet Life Cycle Methods 21](#_Toc468260964)

[4.2.1 Loading and Instantiation 22](#_Toc468260965)

[4.2.2 Initialization 22](#_Toc468260966)

[4.2.3 End of Service 23](#_Toc468260967)

[4.3 Portlet Customization Levels 23](#_Toc468260968)

[4.3.1 Portlet Definition and Portlet Entity 23](#_Toc468260969)

[4.3.2 Portlet Window 24](#_Toc468260970)

[4.4 Portlet Class 24](#_Toc468260971)

[4.5 Example Request Sequence 24](#_Toc468260972)

[4.6 Request Handling 26](#_Toc468260973)

[4.6.1 Action Lifecycle Method 27](#_Toc468260974)

[4.6.2 Event Lifecycle Method 28](#_Toc468260975)

[4.6.3 Header Lifecycle Method 28](#_Toc468260976)

[4.6.4 Render Lifecycle Method 29](#_Toc468260977)

[4.6.5 Resource Lifecycle Method 29](#_Toc468260978)

[4.6.6 Multithreading Issues During Request Handling 29](#_Toc468260979)

[4.6.7 Exceptions During Request Handling 30](#_Toc468260980)

[4.6.8 Thread Safety 31](#_Toc468260981)

[4.7 GenericPortlet 31](#_Toc468260982)

[4.7.1 Dispatching to GenericPortlet Annotated Methods 31](#_Toc468260983)

[4.7.2 Action Dispatching 31](#_Toc468260984)

[4.7.3 Event Dispatching 32](#_Toc468260985)

[4.7.4 Resource Serving Dispatching 32](#_Toc468260986)

[4.7.5 Header Dispatching 32](#_Toc468260987)

[4.7.6 Render Dispatching 32](#_Toc468260988)

[4.8 Extended Annotation-Based Dispatching 33](#_Toc468260989)

[4.8.1 Initialization and Destruction 34](#_Toc468260990)

[4.8.2 Annotated Action Method Dispatching 35](#_Toc468260991)

[4.8.3 Annotated Event Method Dispatching 36](#_Toc468260992)

[4.8.4 Annotated Header Method Dispatching 37](#_Toc468260993)

[4.8.5 Annotated Render Method Dispatching 39](#_Toc468260994)

[4.8.6 Annotated Resource Method Dispatching 41](#_Toc468260995)

[Chapter 5 Portlet Applications 43](#_Toc468260996)

[5.1 Relationship with Web Applications 43](#_Toc468260997)

[5.2 Relationship to the Portlet Context 43](#_Toc468260998)

[5.3 Portlet Application Classloader 43](#_Toc468260999)

[5.4 Directory Structure and Portlet Application Archive File 43](#_Toc468261000)

[5.5 Portlet Application Deployment Descriptor 44](#_Toc468261001)

[5.6 Replacing a Portlet Application 44](#_Toc468261002)

[5.7 Error Handling 44](#_Toc468261003)

[5.8 Portlet Application Environment 44](#_Toc468261004)

[Chapter 6 The PortletConfig Interface 45](#_Toc468261005)

[6.1 Initialization Parameters 45](#_Toc468261006)

[6.2 Portlet Resource Bundle 45](#_Toc468261007)

[6.3 Default Event Namespace 46](#_Toc468261008)

[6.4 Public Render Parameters 46](#_Toc468261009)

[6.5 Publishing Event QNames 46](#_Toc468261010)

[6.6 Processing Event QNames 46](#_Toc468261011)

[6.7 Supported Locales 46](#_Toc468261012)

[6.8 Supported Container Runtime Options 46](#_Toc468261013)

[6.9 Portlet Modes 47](#_Toc468261014)

[6.9.1 Window States 47](#_Toc468261015)

[Chapter 7 Portal Context 48](#_Toc468261016)

[7.1 Portal Context Methods 48](#_Toc468261017)

[7.2 Support for Markup Head Elements 48](#_Toc468261018)

[Chapter 8 Portlet Context 49](#_Toc468261019)

[8.1 Scope of the Portlet Context 49](#_Toc468261020)

[8.2 Portlet Context functionality 49](#_Toc468261021)

[8.3 Relationship to the Servlet Context 49](#_Toc468261022)

[8.3.1 Correspondence between ServletContext and PortletContext methods 50](#_Toc468261023)

[8.4 Portlet Container Runtime Options 50](#_Toc468261024)

[8.4.1 Runtime Option javax.portlet.escapeXml 50](#_Toc468261025)

[8.4.2 Runtime Option javax.portlet.renderHeaders 50](#_Toc468261026)

[8.4.3 Runtime Option javax.portlet.servletDefaultSessionScope 51](#_Toc468261027)

[8.4.4 Runtime Option javax.portlet.actionScopedRequestAttributes 51](#_Toc468261028)

[Chapter 9 Portlet Modes 55](#_Toc468261029)

[9.1 VIEW Portlet Mode 55](#_Toc468261030)

[9.2 EDIT Portlet Mode 55](#_Toc468261031)

[9.3 HELP Portlet Mode 56](#_Toc468261032)

[9.4 Custom Portlet Modes 56](#_Toc468261033)

[9.5 Defining Portlet Modes Support 56](#_Toc468261034)

[9.6 Setting next possible Portlet Modes 57](#_Toc468261035)

[Chapter 10 Window States 58](#_Toc468261036)

[10.1 NORMAL Window State 58](#_Toc468261037)

[10.2 MAXIMIZED Window State 58](#_Toc468261038)

[10.3 MINIMIZED Window State 58](#_Toc468261039)

[10.4 Custom Window States 58](#_Toc468261040)

[10.5 Defining Window State Support 58](#_Toc468261041)

[Chapter 11 Portlet Parameters 60](#_Toc468261042)

[11.1 Portlet Parameters 61](#_Toc468261043)

[11.1.1 Extra Request Parameters 61](#_Toc468261044)

[11.1.2 PortletParameters Interface 61](#_Toc468261045)

[11.1.3 MutablePortletParameters Interface 62](#_Toc468261046)

[11.2 Render Parameters 63](#_Toc468261047)

[11.2.1 Public Render Parameters 64](#_Toc468261048)

[11.2.2 RenderParameters Interface 64](#_Toc468261049)

[11.2.3 MutableRenderParameters Interface 64](#_Toc468261050)

[11.3 Action Parameters 65](#_Toc468261051)

[11.3.1 ActionParameters Interface 65](#_Toc468261052)

[11.3.2 MutableActionParameters Interface 65](#_Toc468261053)

[11.4 Resource Parameters 65](#_Toc468261054)

[11.4.1 ResourceParameters Interface 66](#_Toc468261055)

[11.4.2 MutableResourceParameters Interface 66](#_Toc468261056)

[Chapter 12 Render State 67](#_Toc468261057)

[12.1 The RenderState Interface 67](#_Toc468261058)

[12.2 MutableRenderState Interface 67](#_Toc468261059)

[Chapter 13 Portlet URLs 69](#_Toc468261060)

[13.1 The Base URL 70](#_Toc468261061)

[13.1.1 BaseURL Interface 70](#_Toc468261062)

[13.2 The Resource URL 71](#_Toc468261063)

[13.2.1 ResourceURL Interface 72](#_Toc468261064)

[13.3 The Portlet URL 73](#_Toc468261065)

[13.3.1 PortletURL Interface 73](#_Toc468261066)

[13.4 The Render URL 74](#_Toc468261067)

[13.4.1 RenderURL Interface 74](#_Toc468261068)

[13.5 The Action URL 75](#_Toc468261069)

[13.5.1 ActionURL Interface 75](#_Toc468261070)

[13.6 The Portlet URL Generation Listener 75](#_Toc468261071)

[13.6.1 PortletURLGenerationListener Interface 75](#_Toc468261072)

[13.6.2 Configuration through the Deployment Descriptor 76](#_Toc468261073)

[13.6.3 Configuration through Annotation 76](#_Toc468261074)

[Chapter 14 Portlet Request Interfaces 78](#_Toc468261075)

[14.1 PortletRequest Interface 78](#_Toc468261076)

[14.1.1 Parameter Handling 79](#_Toc468261077)

[14.1.2 Request Attributes 79](#_Toc468261078)

[14.1.3 Request Properties 81](#_Toc468261079)

[14.1.4 Cookies 82](#_Toc468261080)

[14.1.5 Request Context Path 82](#_Toc468261081)

[14.1.6 Security Attributes 82](#_Toc468261082)

[14.1.7 Response Content Types 82](#_Toc468261083)

[14.1.8 Internationalization 83](#_Toc468261084)

[14.1.9 Portlet Mode 83](#_Toc468261085)

[14.1.10 Window State 83](#_Toc468261086)

[14.1.11 Access to the Portlet Window ID 84](#_Toc468261087)

[14.1.12 User Agent 84](#_Toc468261088)

[14.1.13 Deprecated Methods 84](#_Toc468261089)

[14.2 ClientDataRequest Interface 84](#_Toc468261090)

[14.2.1 Retrieving Uploaded Data 84](#_Toc468261091)

[14.3 ActionRequest Interface 85](#_Toc468261092)

[14.4 ResourceRequest Interface 85](#_Toc468261093)

[14.5 EventRequest Interface 86](#_Toc468261094)

[14.6 RenderRequest Interface 86](#_Toc468261095)

[14.7 HeaderRequest Interface 86](#_Toc468261096)

[14.8 Lifetime of the Request Objects 86](#_Toc468261097)

[Chapter 15 PortletResponse Interfaces 87](#_Toc468261098)

[15.1 PortletResponse Interface 88](#_Toc468261099)

[15.1.1 Response Properties 88](#_Toc468261100)

[15.1.2 Setting Cookies 88](#_Toc468261101)

[15.1.3 Setting HEAD Section markup 89](#_Toc468261102)

[15.1.4 Encoding URLs 89](#_Toc468261103)

[15.1.5 Namespacing 90](#_Toc468261104)

[15.2 StateAwareResponse Interface 90](#_Toc468261105)

[15.2.1 Render Parameters 90](#_Toc468261106)

[15.2.2 Portlet Modes and Window State Changes 90](#_Toc468261107)

[15.2.3 Publishing Events 91](#_Toc468261108)

[15.2.4 Deprecated Methods 91](#_Toc468261109)

[15.3 ActionResponse Interface 91](#_Toc468261110)

[15.3.1 Redirections 91](#_Toc468261111)

[15.4 EventResponse Interface 92](#_Toc468261112)

[15.4.1 Deprecated Methods 92](#_Toc468261113)

[15.5 MimeResponse Interface 92](#_Toc468261114)

[15.5.1 Content Type 92](#_Toc468261115)

[15.5.2 Output Stream and Writer Objects 93](#_Toc468261116)

[15.5.3 Buffering 93](#_Toc468261117)

[15.5.4 Predefined MimeResponse Properties 95](#_Toc468261118)

[15.5.5 Creating Portlet URLs 95](#_Toc468261119)

[15.6 RenderResponse Interface 96](#_Toc468261120)

[15.6.1 Next possible portlet modes 96](#_Toc468261121)

[15.6.2 Deprecated Methods 96](#_Toc468261122)

[15.7 ResourceResponse Interface 96](#_Toc468261123)

[15.7.1 Setting the Response Character Set 97](#_Toc468261124)

[15.8 HeaderResponse Interface 97](#_Toc468261125)

[15.8.1 Page Resources and Dependencies 97](#_Toc468261126)

[15.9 Lifetime of Response Objects 99](#_Toc468261127)

[Chapter 16 Portlet Filters 100](#_Toc468261128)

[16.1 What is a portlet filter? 100](#_Toc468261129)

[16.2 Main Concepts 100](#_Toc468261130)

[16.2.1 Filter Lifecycle 100](#_Toc468261131)

[16.2.2 Wrapping Requests and Responses 102](#_Toc468261132)

[16.2.3 Filter Environment 102](#_Toc468261133)

[16.2.4 Filter Configuration 102](#_Toc468261134)

[Chapter 17 Coordination between Portlets 106](#_Toc468261135)

[17.1.1 EventPortlet Interface 106](#_Toc468261136)

[17.1.2 Event Declaration 107](#_Toc468261137)

[17.1.3 Processing Events 108](#_Toc468261138)

[17.1.4 Sending Events 108](#_Toc468261139)

[17.1.5 Event Processing 108](#_Toc468261140)

[17.1.6 Exceptions during event processing 109](#_Toc468261141)

[17.2 Predefined Container Events 109](#_Toc468261142)

[Chapter 18 Portlet Preferences 111](#_Toc468261143)

[18.1 PortletPreferences Interface 111](#_Toc468261144)

[18.2 Preference Attributes Scopes 112](#_Toc468261145)

[18.3 Preference Attributes definition 112](#_Toc468261146)

[18.3.1 Localizing Preference Attributes 113](#_Toc468261147)

[18.4 Validating Preference Values 113](#_Toc468261148)

[18.4.1 Configuration through the Deployment Descriptor 114](#_Toc468261149)

[18.4.2 Configuration through Annotation 114](#_Toc468261150)

[Chapter 19 Sessions 115](#_Toc468261151)

[19.1 Creating a Session 115](#_Toc468261152)

[19.2 Portlet Session Scope 115](#_Toc468261153)

[19.3 Relationship to HttpSession 116](#_Toc468261154)

[19.4 Binding Attributes to a Portlet Session 116](#_Toc468261155)

[19.4.1 PortletSession Interface Methods 117](#_Toc468261156)

[19.5 Modifying Objects in the Portlet Session 117](#_Toc468261157)

[19.6 Reserved HttpSession Attribute Names 118](#_Toc468261158)

[19.7 Session Timeouts 118](#_Toc468261159)

[19.8 Last Accessed Times 118](#_Toc468261160)

[19.9 Important Session Semantics 118](#_Toc468261161)

[Chapter 20 Managed Bean Support 119](#_Toc468261162)

[20.1 Portlet Instantiation 119](#_Toc468261163)

[20.2 Custom Scopes 119](#_Toc468261164)

[20.2.1 Portlet Session Scope 119](#_Toc468261165)

[20.2.2 Portlet Request Scope 120](#_Toc468261166)

[20.2.3 Render State Scope 120](#_Toc468261167)

[20.3 Portlet Predefined Beans 121](#_Toc468261168)

[20.3.1 Portlet Request Scoped Beans 122](#_Toc468261169)

[20.3.2 Dependent Scoped Beans 123](#_Toc468261170)

[Chapter 21 Asynchronous Support 124](#_Toc468261171)

[21.1 Concepts 124](#_Toc468261172)

[21.1.1 Starting Asynchronous Processing 124](#_Toc468261173)

[21.1.2 The Asynchronous Processing Cycle 125](#_Toc468261174)

[21.1.3 The Asynchronous Dispatch Operation 125](#_Toc468261175)

[21.1.4 Portlet Request Dispatcher and Asynchronous Processing 126](#_Toc468261176)

[21.1.5 The Asynchronous Thread 126](#_Toc468261177)

[21.1.6 The Asynchronous Listener 126](#_Toc468261178)

[21.1.7 Portlet Filters and Asynchronous Processing 127](#_Toc468261179)

[21.1.8 Multiple Resource Methods 127](#_Toc468261180)

[21.1.9 Scenarios 127](#_Toc468261181)

[21.2 Asynchronous Programming API 129](#_Toc468261182)

[21.2.1 Resource Request Methods 130](#_Toc468261183)

[21.2.2 Portlet Asynchronous Context Methods 130](#_Toc468261184)

[21.2.3 Portlet Asynchronous Listener Methods 131](#_Toc468261185)

[21.2.4 Portlet Asynchronous Event Methods 132](#_Toc468261186)

[Chapter 22 Client-Side Support 133](#_Toc468261187)

[22.1 Basic Concepts 134](#_Toc468261188)

[22.1.1 Configuration 134](#_Toc468261189)

[22.1.2 Promises 134](#_Toc468261190)

[22.1.3 Changing the Render State 135](#_Toc468261191)

[22.1.4 Receiving Render State Updates 135](#_Toc468261192)

[22.1.5 Portlet Client Events 136](#_Toc468261193)

[22.1.6 Error Handling 136](#_Toc468261194)

[22.1.7 Important Considerations 136](#_Toc468261195)

[22.1.8 Blocking Operations 136](#_Toc468261196)

[22.2 Portlet Parameters and Render State 137](#_Toc468261197)

[22.3 Scenarios 138](#_Toc468261198)

[22.3.1 Initial Page Load 138](#_Toc468261199)

[22.3.2 Single Portlet Update 140](#_Toc468261200)

[22.3.3 Public Render Parameter Update 141](#_Toc468261201)

[22.3.4 Portlet Action 142](#_Toc468261202)

[22.3.5 Partial Action 143](#_Toc468261203)

[22.4 Portlet Hub API 145](#_Toc468261204)

[22.4.1 Portlet.register() 145](#_Toc468261205)

[22.4.2 action(actParams, element) 145](#_Toc468261206)

[22.4.3 addEventListener(type, func) 147](#_Toc468261207)

[22.4.4 createResourceUrl(resParams, cache, resid) 147](#_Toc468261208)

[22.4.5 dispatchClientEvent(type, payload) 148](#_Toc468261209)

[22.4.6 isInProgress() 149](#_Toc468261210)

[22.4.7 newParameters(p) 149](#_Toc468261211)

[22.4.8 newState(s) 149](#_Toc468261212)

[22.4.9 removeEventListener(handle) 149](#_Toc468261213)

[22.4.10 setRenderState(state) 149](#_Toc468261214)

[22.4.11 startPartialAction(actParams) 150](#_Toc468261215)

[22.4.12 onStateChange(type, renderState, renderData) 151](#_Toc468261216)

[Chapter 23 Caching 152](#_Toc468261217)

[23.1 Expiration Cache 152](#_Toc468261218)

[23.2 Validation Cache 152](#_Toc468261219)

[Chapter 24 Security 154](#_Toc468261220)

[24.1 Introduction 154](#_Toc468261221)

[24.2 Roles 154](#_Toc468261222)

[24.3 Programmatic Security 154](#_Toc468261223)

[24.4 Propagation of Security Identity in EJBTM Calls 155](#_Toc468261224)

[Chapter 25 Dispatching to JSPs and Servlets 156](#_Toc468261225)

[25.1 Obtaining a PortletRequestDispatcher 156](#_Toc468261226)

[25.1.1 Query Strings in Request Dispatcher Paths 156](#_Toc468261227)

[25.1.2 Path and Query Information 157](#_Toc468261228)

[25.2 Using a Request Dispatcher 158](#_Toc468261229)

[25.3 Error Handling 158](#_Toc468261230)

[25.4 Portlet-Specific Request Attributes 159](#_Toc468261231)

[25.4.1 Changing the Default Behavior for Session Scope 160](#_Toc468261232)

[25.5 The forward Method 160](#_Toc468261233)

[25.5.1 Forwarded Request Attributes 161](#_Toc468261234)

[25.6 The Include Method 161](#_Toc468261235)

[25.6.1 Included Request Attributes 162](#_Toc468261236)

[25.7 Asynchronous Processing 162](#_Toc468261237)

[25.8 Servlet filters and Request Dispatching 163](#_Toc468261238)

[25.9 HttpServletRequest and HttpServletResponse Objects 163](#_Toc468261239)

[25.9.1 Method Behavior during Portlet include Dispatching 164](#_Toc468261240)

[25.9.2 Method Behavior during Portlet forward and Asynchronous Dispatching 167](#_Toc468261241)

[Chapter 26 Portlet Tag Library 172](#_Toc468261242)

[26.1 defineObjects Tag 172](#_Toc468261243)

[26.2 actionURL Tag 174](#_Toc468261244)

[26.3 renderURL Tag 175](#_Toc468261245)

[26.4 resourceURL Tag 177](#_Toc468261246)

[26.5 namespace Tag 178](#_Toc468261247)

[26.6 param Tag 178](#_Toc468261248)

[26.7 property Tag 179](#_Toc468261249)

[26.8 Changing the Default Behavior for escapeXml 179](#_Toc468261250)

[Chapter 27 Packaging and Deployment 180](#_Toc468261251)

[27.1 Packaging 180](#_Toc468261252)

[27.1.1 Example Directory Structure 180](#_Toc468261253)

[27.1.2 Version Information 181](#_Toc468261254)

[27.2 Portlet Deployment Descriptor Structure 181](#_Toc468261255)

[27.2.1 Portlet Deployment Descriptor Elements 181](#_Toc468261256)

[27.2.2 Rules for processing the Portlet Deployment Descriptor 181](#_Toc468261257)

[27.2.3 Uniqueness of Deployment Descriptor Values 182](#_Toc468261258)

[27.2.4 Localization of Deployment Descriptor Values 182](#_Toc468261259)

[27.3 Deployment Descriptor Example 183](#_Toc468261260)

[Chapter 28 Configuration 184](#_Toc468261261)

[28.1 Portlet Application Configuration 184](#_Toc468261262)

[28.1.1 Portlet API Version 185](#_Toc468261263)

[28.1.2 Custom Portlet Mode 185](#_Toc468261264)

[28.1.3 Custom Window State 186](#_Toc468261265)

[28.1.4 User Attribute 186](#_Toc468261266)

[28.1.5 Resource Bundle 187](#_Toc468261267)

[28.1.6 Filter Configuration 189](#_Toc468261268)

[28.1.7 Default Namespace URI 189](#_Toc468261269)

[28.1.8 Event Configuration 190](#_Toc468261270)

[28.1.9 Public Render Parameters 191](#_Toc468261271)

[28.1.10 Portlet URL Generation Listener 192](#_Toc468261272)

[28.1.11 Portlet Container Runtime Options 192](#_Toc468261273)

[28.2 Portlet Configuration 193](#_Toc468261274)

[28.2.1 @PortletConfigurations Annotation 194](#_Toc468261275)

[28.2.2 Portlet Container Runtime Options 194](#_Toc468261276)

[28.2.3 Portlet Initialization Parameters 195](#_Toc468261277)

[28.2.4 Portlet Identification 195](#_Toc468261278)

[28.2.5 Portlet Resource Bundle 196](#_Toc468261279)

[28.2.6 Cache Settings 198](#_Toc468261280)

[28.2.7 Security Role Reference 198](#_Toc468261281)

[28.2.8 Page Resource Dependencies 199](#_Toc468261282)

[28.2.9 Public Render Parameter References 200](#_Toc468261283)

[28.2.10 Event References 201](#_Toc468261284)

[28.2.11 Supported Locales 202](#_Toc468261285)

[28.2.12 Portlet Modes and Window States 202](#_Toc468261286)

[28.2.13 Portlet Preferences 203](#_Toc468261287)

[28.2.14 Asynchronous Support 204](#_Toc468261288)

[28.2.15 Multipart Support 205](#_Toc468261289)

[Appendix A Change History 207](#_Toc468261290)

[A.1 Version 3.0 Changes 207](#_Toc468261291)

[A.2 Version 2.1.0 Changes 209](#_Toc468261292)

[A.3 Version 2.0 Changes 209](#_Toc468261293)

[A.3.1 Major changes introduced with V 2.0 209](#_Toc468261294)

[A.3.2 Clarifications that may make V1.0 Portlets Non-compliant 210](#_Toc468261295)

[Appendix B Markup Fragments 211](#_Toc468261296)

[Appendix C User Attribute Names 212](#_Toc468261297)

[Appendix D Custom Portlet Modes 214](#_Toc468261298)

[D.1 About Portlet Mode 214](#_Toc468261299)

[D.2 Config Portlet Mode 214](#_Toc468261300)

[D.3 Edit\_defaults Portlet Mode 215](#_Toc468261301)

[D.4 Preview Portlet Mode 216](#_Toc468261302)

[D.5 Print Portlet Mode 217](#_Toc468261303)

[Appendix E Deployment Descriptor Schema 218](#_Toc468261304)

# Overview

## What is a Portal?

A portal is a web based application that provides personalization, authentication, and content aggregation from different sources and hosts the presentation layer of information systems. Aggregation is the action of integrating content from different sources on a web page. A portal may have sophisticated personalization features to provide customized content to users. Portal pages may have different set of portlets creating content for different users.

## What is a Portlet?

A portlet is an application that provides a specific piece of content (information or service) to be included as part of a portal page. It is managed by a portlet container that processes requests to generate dynamic content. Portlets are used by portals as pluggable user interface components that provide a presentation layer to information systems.

The content generated by a portlet called a fragment. A fragment is a piece of markup (e.g. HTML, XHTML, and WML) that adheres to certain rules so that it can be aggregated with other fragments to form a complete document. The content of a portlet is normally aggregated with the content of other portlets along with additional theme markup to form the portal page.

Web clients interact with portlets via a request/response paradigm implemented by the portal. Normally, users interact with content produced by portlets, for example by following links or submitting forms, resulting in portlet actions being received by the portal. The portal forwards these actions to the targeted portlets.

The content generated by a portlet may vary from one user to another depending on the user configuration for the portlet.

This specification will deal with portlets as Java technology-based web components.

## What is a Portlet Container?

A portal provides infrastructure, known as the portlet container, for running portlets. The portlet container implements the programming interfaces described in this document to allow the portlets to obtain information about the request, to generate markup to be integrated into the complete portal web page, and to store state information.

A portlet container provides portlets with the required runtime environment and manages their lifecycle. It also provides persistent storage for portlet preferences. A portal sends requests to the portlet container in order to cause portlet method invocation.

A portlet container is not responsible for aggregating the content produced by the portlets. The portal handles content aggregation.

A portal and a portlet container can be built together as a single component or as separate components within a portal system.

## Client-Side Support

The Portlet Specification provides a JavaScript module that runs on the web browser client. This module, known as the portlet hub, supports development of responsive web pages consisting of independently-developed portlets aggregated by the portal.

A portlet can make use of the portlet hub by providing appropriate JavaScript code within its markup.

## An Example

The following is a typical sequence of events initiated when a user accesses the portal page:

* A client (e.g., a web browser) sends a form to the portal using an HTTP POST request.
* The request is received by the portal.
* The portal determines the target portlet on the portal page.
* The portal sends a request to the portlet container to cause the portlet to process the form.
* After form processing completes, the portal uses the portlet container to invoke each portlet in turn to obtain content fragments that can be included in the resulting portal page.
* The portal aggregates the output of the portlets along with theme markup to form a complete portal page and sends it back to the client.

## Render State and Backend State

Previous versions of the Portlet Specification defined the idea of render state implicitly. Portlet Specification 3.0 explicitly introduces the render state concept composed of the portlet mode, window state, and the render parameters. The render state can be read in all portlet lifecycle methods but can be updated only during event and action request processing. The render state can be thought of as a view state that defines the information the portlet should display.

The backend state concerns the information itself that is displayed by the portlet. This may be data from a database or from a content management system. The backend state can change even though the render state remains unchanged.

To understand the difference between the render state and the backend state, consider the following example. Assume that the render state stored in a link specifies that page three of a long paginated list of order information from a database is to be displayed. Using this render state, the portlet will always display page three of the order list every time the link is clicked, even though the actual order information displayed changes as the backend state is updated due to new orders being entered and due to processing of the current orders.

## Compatibility

The Java Portlet Specification version 3.0 extends the interfaces defined by its predecessor JSR 286 Portlet Specification 2.0 so as to support binary compatibility. Version 2.0 portlets (identified by a version 2.0 deployment descriptor) will run on a version 3.0 portlet container unchanged. Since the Portlet Specification 2.0 provides binary compatibility with JSR 168 Portlet Specification version 1.0 portlets, version 1.0 portlets will also run on a version 3.0 portlet container unchanged.

Portlet Specification version 3.0 portlet containers must support deployment of version 3.0, version 2.0 and version 1.0 portlets.

## Relationship to Java Enterprise Edition

The Portlet API v3.0 is based on the Java Standard Edition 7.0 and Java Enterprise Edition v7.0. Portlet containers should at least meet the requirements described by the Java EE 7.0 specification for executing in a Java EE environment.

Since portlets provide functionality in a sense similar to that of servlets, this specification will use analogous concepts and names for portlet features that correspond to servlet features.

Contexts and Dependency Injection (CDI) is an important component of the Java EE specification. Use of CDI beans has become popular within the Java development community.

The Portlet Specification provides features that make it easy to use CDI beans when writing portlets. When a CDI container is present, the portlet container will instantiate portlets that are contained in valid bean archive files through the CDI container so that the CDI injection mechanism can be used within portlet classes. The portlet container will provide custom CDI scopes to support the portlet lifecycle.

## Configuration

A portlet application is configured through a deployment descriptor in the same way that a servlet is configured. The portlet application corresponds to the web application. The portlet application consists of one or more portlets. The portlets are also declared within the deployment descriptor.

Beginning with Portlet Specification version 3.0, most configuration tasks can also be performed through use of annotations. This includes the portlet declarations as well.

# Relationship to the Servlet Specification

The *Servlet Specification* defines servlets as follows*[[1]](#footnote-1)*:

"A servlet is a Java technology based web component, managed by a container that generates dynamic content. Like other Java-based components, servlets are platform independent Java classes that are compiled to platform neutral bytecode that can be loaded dynamically into and run by a Java enabled web server. Containers, sometimes called servlet engines, are web server extensions that provide servlet functionality. Servlets interact with web clients via a request/response paradigm implemented by the servlet container."

Portlets share many similarities with servlets:

* Portlets are Java technology based web components.
* Portlets are managed by a specialized container.
* Portlets generate dynamic content.
* Portlets lifecycle is managed by a container.
* Portlets interact with web client via a request/response paradigm.

Portlets differ in the following aspects from servlets:

* The portlet render method only generates markup fragments, not complete documents. The portal aggregates portlet markup fragments into a complete portal page.
* Portlets can only be invoked through URLs constructed via the portlet API.
* Web clients interact with portlets through a portal system.
* Portlets have more refined request handling then servlets. The Portlet Specification defines Action, Event, Header, Render and Resource requests. The portlet container recognizes the portlet request type and dispatches the request to the appropriate portlet lifecycle method.
* Portlets have predefined portlet modes and window states that indicate the function the portlet is performing and the amount of real estate the portlet requires on the portal page.
* Portlets can appear multiple times on a portal page.

Portlets have access to functionality not provided by servlets:

* Portlets have a means of accessing and storing persistent configuration and customization data.
* Portlets have a means of storing render state information in the form of render parameters, the portlet mode, and the window state.
* Portlets have access to user profile information.
* Portlets have URL rewriting functions for creating hyperlinks within their content, which allow portal-server agnostic creation of links and actions in page fragments.
* Portlets can store transient data in the portlet session in two different scopes: the application-wide scope and the portlet private scope.
* Portlets can send and receive events from other portlets or can receive container defined events.

Portlets do not have access to the following functionality provided by servlets:

* Portlets may not set the character set encoding of the render response.
* Portlets may not directly access the URL of the client request to the portal.

The portlet has full control over the response when rendering resources via the serveResource call.

Because of these differences, the Expert Group has decided that portlets need to be a new component. Therefore, a portlet is not a servlet. This allows definition of a clear interface and behavior for portlets.

In order to reuse as much of the existing servlet infrastructure as possible, the Portlet Specification leverages functionality provided by the Servlet Specification wherever possible. This includes deployment, class loading, definition of web applications, web application lifecycle management, session management, and request dispatching. Many concepts and parts of the Portlet API have been modeled after the Servlet API.

Portlets, servlets, and JSPs are bundled in an extended web application called a portlet application. Portlets, servlets, and JSPs within the same portlet application share the same class loader, application context, and session.

## Bridging from Portlets to Servlets/JSPs

Portlets can leverage servlets, JSPs and JSP tag-libraries for generating content.

A portlet can call servlets and JSPs in the same manner that a servlet can invoke other servlets and JSPs using a request dispatcher (see Chapter 25 Dispatching to JSPs and Servlets). To enable a seamless integration between portlets and servlets the Portlet Specification leverages many of the servlet objects.

When a servlet or JSP is called from within a portlet, the servlet request passed to the servlet or JSP is based on the portlet request and the servlet response passed to the servlet or JSP is based on the portlet response. A few of the consequences of this are listed below:

* Attributes set in the portlet request are available in the included servlet request (see Chapter 25).
* The portlet and the included servlet or JSP share the same output stream (see Chapter 25).
* Attributes set in the portlet session are accessible from the servlet session and vice versa (see Chapter 19 Sessions).

## Using Servlet Application Lifecycle Events

The Java Servlet Specification describes a variety of servlet application lifecycle events. A servlet can register event listeners for these events. In the sense of the lifecycle events, the portlet objects PortletContext and PortletSession defined by this specification mirror their servlet counterparts. The lifecycle of the PortletContext is tied to that of the ServletContext of the web application. The attributes set in the PortletContext are mirrored in the ServletContext. The lifecycle of the PortletSession is tied to that of the HttpSession of the web application. The attributes set in the PortletSession are mirrored in the HttpSession. Since this is the case, the servlet lifecycle listeners for ServletContext and HttpSession can also be used for PortletContext and PortletSession notifications.

Given that the portlet request is independent of the servlet request, the servlet request lifecycle listeners do not have a simple mapping to portlet requests. In order to allow portlets to leverage the servlet request listeners for portlets, the portlet container must create a servlet request mirroring the portlet request. In order to allow the servlet request listeners to distinguish between the case of a plain servlet request and a servlet request targeted towards a portlet, the portlet container must set the javax.portlet.lifecycle\_phase request attribute in order to identify the servlet request representing a portlet request.

The following is the list of servlet listeners that also apply to portlets:

* The javax.servlet.ServletContextListener provides notifications about the servlet context and the corresponding portlet context.
* The javax.servlet.ServletContextAttributeListener provides notifications on attributes in the servlet context or the corresponding portlet context.
* The javax.servlet.http.HttpSessionActivationListener provides notifications on the activation or passivation of the HttpSession or the corresponding PortletSession.
* The javax.servlet.http.HttpSessionAttributeListener provides notifications on attributes of the HttpSession or the corresponding PortletSession.
* The javax.servlet.http.HttpSessionListener provides notifications about HttpSession or the corresponding PortletSession lifecycle changes.
* The javax.servlet.http.HttpSessionBindingListener provides notifications on binding of objects to the HttpSession or the corresponding PortletSession.
* The javax.servlet.ServletRequestListener provides notifications about changes to the HttpServletRequest or the mirrored portlet request of the current web application.
* The javax.servlet.ServletRequestAttributeEvent provides notifications about changes to the attributes of the HttpServletRequest or the mirrored portlet request of the current web application.

## The Servlet Container - Portlet Container Relationship

The portlet container is an extension of the servlet container. As such, a portlet container can be built on top of an existing servlet container or it may implement all the functionality of a servlet container. Regardless of how a portlet container is implemented, its runtime environment is assumed to support at least *Java™ Servlet Specification Version 3.1*.

# Portlet Concepts

## Portlets

Complex web sites often need to combine data from multiple sources to create a single web page. A shipping web site might need to show order information, user information, and a map, for example. These diverse types of information are accessed through different programming interfaces and protocols.

Often, the job of accessing, displaying, and updating a specific type of information is complex enough to warrant a separate application in its own right. In order to reduce complexity and also to make the application easier to use in other settings, the application should be able to render markup and perform updates independently of other applications while still being able to make use of information explicitly provided by other applications. To guide the user through a complex task, the application requires the ability to store state information.

A portlet is a web application that fulfills these requirements.

A portal system aggregates content rendered by multiple portlets to create a complete web page. When this happens, additional requirements come into play. The portlets should be able to operate independently of one another. For example, when the user information is updated, the order information should not be lost or change in an unplanned manner. A user should be able to set a bookmark on a portal page in order to return to a particular overall view. And browser back button support should work in an intuitive manner.

The portlet programming model allows a portal system to fulfill these requirements.

Web pages often incorporate JavaScript code to provide a responsive user experience. When a change is made, the JavaScript code only updates that portion of the web page actually requiring a change. When this principle is applied to a portal system, it means on one hand that a portlet needs to be able to provide JavaScript code that can update its own markup.

But portlets can share information among themselves, so that an update to one portlet can affect the state and rendered markup of another portlet. This means on the other hand that the JavaScript code provided by a given portlet also needs to be activated when it is affected by a change initiated by a different portlet. The portlet programming interface provides a mechanism that takes these requirements into account.

The portlet programming interface described by this specification provides the means to create independent portlet applications that a portal system can integrate with other portlet applications to create a complete complex web page. It provides the following services:

* It allows the portlet to render markup fragments that a portal can combine with others to form a complete web page.
* It allows the portlet to update its own information without affecting other portlets on the page.
* It allows portlets to share information with other portlets in a defined, standardized manner.
* It allows the portlet to store its current state in order to maintain a specific data display.
* It allows the portlet to use JavaScript to provide a responsive user experience.

## Portal Pages

A portlet generates markup fragments. A portal may add a title, control buttons and other decorations to the markup fragment generated by the portlet. This new fragment is called a portlet window. Then the portal may aggregate portlet windows into a complete document representing the portal page.

Figure 3–1 Elements of a Portal Page

Note that this is only one example on how a portal could make use of the portlet markup fragment. Portal implementations with a different rendering approach are possible. The important part of the portal page concept in regards to this specification is that the markup fragment generated by the portlet will in general not be the only document markup returned to the client. Thus the portlet markup needs to co-exist with whatever other markup the portal produces.

## The Portlet Container

Page rendering begins when a client device or user agent, such as a web browser or web-enabled phone, sends a request for a page to the portal. The portal accesses the portlet container in order to obtain the markup to be returned to the client.

Figure 3–2 The Portlet Container

The portlet container implements the interfaces described by the Portlet Specification in order to provide the portlet with the requisite runtime environment, in the same manner that the servlet container implements the Servlet Specification in order to provide a servlet with the requisite runtime environment.

The Portlet Specification describes the programming interface and its semantics as seen by a portlet. It describes neither the portlet container implementation nor the interaction between a portal system and the portlet container. When discussing portal system or portlet container behavior from the point of view of the portlet, this document will refer to the portlet container as though it provides the complete functionality in question.

The portlet container calls the portlet code to execute the required function. It does so by issuing requests to the portlet. The requests are analogous to servlet requests, but have more specific meanings. There are five types of portlet request:

* Header request – generates markup for inclusion into the head section of a portal page and sets HTTP header information.
* Render request – generates markup fragments for inclusion into a portal page body section.
* Action request – performs processing that can cause a change to persistent data.
* Event request – processes an event that was fired by a portlet or by the portlet container. Can cause a change to persistent data.
* Resource request – generates additional data related to the current render state.

## Typical Processing Sequence

The figure below illustrates a typical processing sequence. In this scenario, the user interacts with the user interface to trigger an action, for example by submitting a form. The request is transported to the portal server, where the following takes place:

1. The portlet container routes an action request to portlet B.
2. Portlet B processes the action request, and while doing so, fires a portlet event.
3. The portal system determines that portlet A is a recipient of the event.
4. The portlet container routes an event request to portlet A, which processes it.
5. After the action and event requests have been processed, the portal system renders each of the portlets on the page to obtain markup.
6. The portal system aggregates all content into a single document representing the portal page and returns it to the client.

Figure 3–3 Typical processing sequence

## Serving Resources

When portlets generate markup, they can add links to internal resources for retrieval by the browser.

The process of providing additional resources for inclusion into the portlet markup is known as resource serving, and such links are known as resource links or resource URLs. Portlets can create two different kinds of resource links in order to serve resources:

1. Direct links to the resource in the same portlet web application. The portlet constructs and encodes such links. They are not guaranteed to pass through the portal server and do not initiate portlet processing. They should be used in cases where the access to the portlet context and access through the portal is not needed, for example when serving a static resource such as an image. The portlet should use these links whenever possible.
2. Resource URL links that address the portlet. The portlet generates such links using methods defined by the Portlet Specification. In this case, activating the resource URL link causes a resource request to be routed to the target portlet. Since the portal handles the request triggered by the resource URL, it can impose user-specific security constraints on the request as it can for any other portlet request.

The portlet can use its portlet context and other resources to generate the appropriate data. During resource request processing, the portlet is not limited to generating markup, but has complete control of the response, and can send any appropriate data back to the client – HTML markup, JSON data, binary data, or whatever.

The figure below shows the resource processing sequence.

Figure 3–4 Resource Processing Sequence

The top part of the picture shows a normal action request that results in a complete page rendering. The resource serving sequence is executed as follows:

1. A resource URL for portlet A is triggered, resulting in an asynchronous XMLHttpRequest to the portal.
2. The portlet container invokes a resource request for portlet A.
3. Portlet A generates response data and returns it to the portlet container.
4. The portal returns the markup to the browser client
5. JavaScript code running on the client updates the user interface accordingly, for example by updating the browser DOM.

## Execution Stages and Phases

The portlet container must enforce an execution stage model to control portlet request invocation. Conceptually, a portlet execution stage is triggered by activation of a portlet URL. A portlet execution stage consists of one or more portlet request processing phases.

Figure 3–5 The Portlet Phase Model

### The Preparation Stage

The purpose of the preparation stage is to allow the portlet to make changes to the current render state that describes which data to display as well as to the actual data being served.

The preparation stage is triggered through use of either an action URL or a partial action URL. In either case, preparation stage processing is the same. Commonly, the client will use the HTTP POST method to initiate preparation stage execution.

To begin the preparation stage, the portlet container invokes the action phase for the target portlet. During action phase processing, the portlet may fire one or more events.

If the action target portlet fires an event, the portlet container invokes the event phase for each recipient portlet. For a given event, there may be zero or more event recipient portlets. The portlet firing the event may also receive the same event.

A portlet may fire additional events during event request processing. If it does so, the portlet container will dispatch the events to recipient portlets as previously described.

At the end of the preparation stage, the portlet will have prepared the render state and backend state for execution of the aggregation stage. If events were dispatched, the event recipient portlets as well as the action target portlet may have updated their render state and backend state.

Once the render state is set during preparation stage execution, it remains in force until it is explicitly changed through activation of a render URL containing new render state data or until it is explicitly changed through a subsequent preparation stage execution.

The render state of portlets that are neither the action target portlet nor an event recipient portlet remains unchanged except for public render parameters modified during the preparation stage.

If the preparation stage was initiated through an action URL, the portlet container can either continue directly with aggregation stage execution, or may redirect the client to a render URL containing the render state information in order to implement a POST-redirect-GET paradigm. Which of the methods is used is left as a portlet container implementation detail.

If the preparation stage was initiated through a partial action URL, the portlet container must execute the resource stage for the action target portlet rather than continuing with the aggregation stage. The portlet container will use the render state set during the preparation stage to execute the resource stage.[[2]](#footnote-2)

### The Aggregation Stage

The purpose of the aggregation stage is to produce markup that is aggregated into a complete portal page and returned to the client.

Conceptually, the aggregation stage is always initiated through a render URL containing the render state of all of the portlets on the page to be rendered. The render URL can be generated by a preceding preparation stage or can arrive at the portal through user interaction with a render URL link or through a page refresh.

Since the preparation stage does not change the render state of unaffected portlets, a portlet may be rendered multiple times for a given render state.

The portlet container will execute the aggregation stage for each portlet on the page being rendered.

The aggregation stage consists of two portlet request processing phases, the header phase and the render phase, that are executed in sequence.

The portlet container invokes the header phase to allow the portlet to generate HTTP header and HTML document head section data. After the header phase completes, the portlet container invokes the portlet render phase to allow the portlet to generate markup for the document body.

The portlet container must perform header phase processing for all portlets on the page before the aggregated portal response is committed. This can mean that the header phase invocations for all portlets on the page are completed before the render phase invocations begin.

### The Resource Stage

The purpose of the resource stage is to produce data for inclusion into a portal page that was generated during a preceding aggregation stage.

The resource stage is invoked when a resource URL is activated. This typically happens when user interaction causes JavaScript code executing on the browser to request resource data through use of a resource URL link.

The portlet container invokes the resource phase for the target portlet so that the portlet can produce the necessary response data.

The portlet has complete control over the response data produced during the resource phase. The portal application must not render any output in addition to the content returned by the resource request, although it may set additional HTTP response headers. The portal application should act only as a proxy for accessing the resource.

The resource stage can be viewed as a logical extension of the aggregation stage, since the render state contained in the resource URL is fixed to be the same render state that governed the aggregation stage during which the resource URL was created.

## Portlet Request Processing Phases

The portlet request processing action, event, header, render, and resource phases were introduced in the preceding section. The portlet container will generally invoke one or more portlet request execution methods during each request processing phase.

When executing a portlet request processing phase, the portlet container creates a portlet request and a portlet response object for use by the portlet. The portlet request and response objects are analogous to the servlet request and response objects. Since different portlet requests require different data and capabilities, the Portlet Specification defines appropriate request and response objects for each portlet request processing phase.

After creating the request and response objects, the portlet container then invokes the appropriate portlet request execution method or methods.

### Render Phase

When the portal is rendering a page, it invokes the portlet container. The portlet container in turn executes the render phase for each portlet on the page by creating the render request and response objects and invoking the appropriate portlet render method or methods. The portlet generates the appropriate markup based on the render state reflected in the render request and returns it to the portlet container through the render response. The generated markup type must be compatible with the markup type generated by the portal. The portlet container returns the markup to the portal for aggregation into the portal page.

The portlet cannot change the render state during the render phase.

When the portal has received markup from all portlets on the page, it writes aggregated page as response data for transmission to the client.

Note that the portal is not required to wait until it has obtained markup from all portlets on the page before returning markup to the client. It can instead stream content to the client as it is obtained from each portlet. The exact manner in which the portal provides markup to the client is left to the portal implementation.

### Header Phase

The portlet container executes the header phase by creating the header request and response objects and invoking the appropriate portlet render header method or methods. The portlet container must execute the header phase during page rendering, but before the portal application has written any response data for transmission to the client. The portlet may set cookies or other HTTP header information and may generate markup for inclusion into the head section of the overall portal response document returned to the client.

The portlet cannot change the render state during the header phase.

### Action Phase

When the user interacts with portlet markup displayed by a user agent so as to cause an update to information stored on the server, a portlet action is initiated. Typically this happens when the user submits a form that the portlet rendered as part of its markup.

The portal routes the submitted form to the portlet that generated the form markup. The portal recognizes the appropriate portlet through information encoded into the URL used for the form submission.

The portal passes the form to the portlet container, which then executes the action phase by creating the action request and response objects and invoking the appropriate portlet action method.

During action phase processing, the portlet does not generate markup to be returned to the client. Instead, the portlet updates its render state information and potentially also data stored on the server or on a backend system so that the appropriate markup can be generated during subsequent render request processing.

In the simplest case, after the portlet completes action phase processing, the portlet container invokes the render phase for each portlet on the page as described in a preceding section. During the rendering, the action request target portlet may have updated its render state, but the remaining portlets on the page will be rendered with the same private render state information as before.

### Event Phase

The portlet event mechanism provides a way for portlets to communicate with one another.

Event phase processing can be initiated either by a portlet or by the portlet container. The portlet can initiate event phase processing by firing one or more portlet events during action phase execution. Also, the portlet container may fire implementation-specific events targeted to a portlet. Such events may be triggered by a portal URL or through an implementation-specific mechanism.

Regardless of how the event phase is initiated, the portlet container can route the events to portlets able to receive them. The algorithm used for event routing is not defined in the Portlet Specification and is implementation specific.

When the portlet container routes an event to a recipient portlet, it invokes the portlet event phase by creating the event request and response objects and invoking the appropriate portlet event method. During event phase processing, the portlet does not generate markup to be returned to the client. Instead, the portlet updates its render state information and potentially also data stored on the server or on a backend system so that the appropriate markup can be generated during subsequent render request processing.

The portlet may also fire additional events during the event phase.

### Resource Phase

The resource phase differs from the remaining portlet request execution phases in that it is not strictly bound to the portal page rendering sequence.

When the client activates a resource URL, the portal invokes the portlet container for the target portlet. The portlet container invokes the portlet resource phase by creating the resource request and response objects and invoking the appropriate portlet resource method or methods. The portlet generates a response for the client based on the render state and any additional resource parameters.

Typically the resource URL is activated through use of JavaScript code. The JavaScript code triggering the resource URL receives the resource response and adds the information to the portal page in the appropriate manner.

Unlike render phase processing, where the generated markup type must match the markup type generated by the portal, during the resource phase, the portlet has complete control over the generated response. The portlet can return HTML markup, plain text, JSON data, image data, or other data as appropriate.

The portlet cannot change the render state during the resource phase.

## Portlet URLs

The portlet can use the Portlet API to generate URLs for inclusion into portlet markup. There are four types of portlet URL:

* The render URL initiates aggregation stage processing.
* The action URL initiates preparation stage processing
* The partial action URL initiates preparation stage processing.
* The resource URL initiates a resource stage processing.

All types of portlet URL can contain render state information. When a portlet URL is activated, the render state specified by the URL is made available to the portlet when the corresponding portlet request phase is invoked.

The Java classes that represent the render URL and action URL artifacts provide methods for modifying the render state. Any render state information attached to a resource URL cannot be modified.

The action URL can contain additional action parameters. The portlet can use portlet API methods to set action parameters. Also, parameters resulting from HTML form submission are also provided through the action parameters.

The client-side portlet API allows for creation of a partial action URL in order to support action-oriented frameworks. The portlet client-side code can specify additional action parameters for inclusion on the partial action URL. The render state information cannot be modified. Parameters resulting from HTML form submission are also provided through the action parameters on the partial action URL.

The resource URL is bound to the render state governing the portlet request processing phase during which the resource URL is created. The portlet can use portlet API methods to set the cacheability option for the resource URL. The resource URL will contain the render state depending on the cacheability setting. The resource URL can contain additional resource parameters. The portlet can use portlet API methods to set resource parameters.

The actual content and structure of the portlet URL is left as an implementation detail of the portal implementation and is not described further in this specification.

## Portlet Parameters and Render State

### Portlet Parameters

Portlet parameters are name value pairs modeled after servlet request parameters. The parameter name is represented by a string while the value is a string array. The Portlet Specification discerns between several types of parameters depending on how they are used.

Render parameters govern how the portlet is to generate data for aggregation into a portal page. The portlet can set render parameters during action phase processing to govern the following render and resource phases.

During render and resource phase execution, the portlet can generate render and action URLs and set render parameters on them. When the URL is activated, the portlet container uses the render parameters set on the URL to govern resulting render or action phase execution.

Action parameters are attached to an action or partial action URL and are independent from the render parameters. The portlet can set action parameters on the URL explicitly, or the action parameters can be set implicitly as a result of a form submission.

During action phase processing, the portlet container provides the action parameters to the portlet separately from the render parameters.

Resource parameters are portlet parameters independent from render parameters that can be attached to a resource URL. Resource parameters must be explicitly attached to a resource URL.

During resource phase processing, the portlet container provides the resource parameters to the portlet separately from the render parameters.

#### Public Render Parameters

Render parameters as described above are private to the portlet. The portlet container must not allow other portlets to read or modify private render parameters.

The portlet configuration contains fields that allow specific render parameters to be declared as public. A public render parameter can be read and modified by any portlet that declares that specific parameter to be public. See Section 11.2.1 Public Render Parameters for further information.

### The Window State

The window state is an indicator of the amount of page space the portlet is allowed to use on the portal page. The portlet container provides the window state to the portlet during the render and resource phases so that the portlet can adapt its display appropriately.

### The Portlet Mode

The portlet mode indicates the function that the portlet is to perform during rendering. For example, during ‘VIEW’ mode, the portlet should render its usual content for display, while during ‘HELP’ mode, the portlet should display information about how to use the portlet. The portlet container provides the portlet mode to the portlet during the render and resource phases so that the portlet can adapt its display appropriately.

### The Render State

The render state consists of the render parameters, the portlet mode, and the window state. The render parameters in the render state can be either public or private. The render state information is provided to the portlet in each request.

Conceptually, the render state is stored on the URL used to access the portal because the URL designates the page to be rendered or the operation to be performed. The portlet container must provide the render state indicated by the URL to the portlet during every portlet request. However, a portal implementation may choose to actually store the information in a location other than the URL.

## Portlet Stage Execution

If the client request is triggered by an action URL, the portlet container must first trigger preparation stage execution for the target portlet. After the preparation stage completes, the portlet container must trigger the aggregation stage for all the portlets on the portal page with the possible exception of portlets for which content is being cached. The header phase for each of the portlets may be executed sequentially or in parallel without any guaranteed order, but the header phases for all of the portlets must be complete before the portal begins writing the aggregated response data. The render phase for each of the portlets may be executed sequentially or in parallel without any guaranteed order.

If the client request is triggered by a partial action URL, the portlet container must first trigger preparation stage execution for the target portlet, executing both the action phase and any resulting event phase for all affected portlets. After the preparation stage completes, the portlet container must execute the resource stage for the partial action target portlet and return any resulting markup in the portal response.

If the client request is triggered by a render URL, the portlet container must execute the markup phase for all the portlets on the portal page with the possible exception of portlets for which content is being cached. The header phase for each of the portlets may be executed sequentially or in parallel without any guaranteed order, but the header phases for all of the portlets must be complete before the portal begins writing the aggregated response data. The render phase for each of the portlets may be executed sequentially or in parallel without any guaranteed order.

If the client request is triggered by a resource URL, the portal/portlet container must execute the render stage of the target portlet unless a valid cache entry is available.

If a portlet has caching enabled, the portlet container may choose not to invoke the markup or resource stages for that portlet. The portal/portlet container may instead use the portlet’s cached content. Refer to Chapter 23 Caching for details on caching.

## Client-Side Support

Use of JavaScript code on the client to improve the end-user experience has become very prevalent. Portlet Specification 2.0 introduced resource serving to allow portlets to access resources according to the Ajax paradigm. Portlet Specification 3.0 improves client-side support for portlets by introducing a dedicated client-side JavaScript API for portlet support.

Figure 3–6 Client Side Support

The client-side portlet API is known as the portlet hub. Portlet Specification 3.0 defines the API and its semantics, but does not define its implementation details. The portlet hub implementation along with any necessary communication protocols between the portlet hub and the portal server are implementation-specific and are not covered by this specification.

The portlet hub is a JavaScript module provided by the portal implementation that manages the state for all portlets on the page.

A portlet can provide JavaScript code known as the portlet client that registers itself with the portlet hub. Once registered, the portlet hub informs the portlet client through a callback function whenever the render state for that portlet changes. The portlet client can use portlet hub methods to set the render state, to execute portlet actions, and to obtain resource URLs that correspond to the current page state.

Refer to Chapter 22 Client-Side Support for additional details.

## Portlet Lifecycle Methods

The portlet container calls portlet lifecycle methods under well-defined circumstances to perform portlet initialization, execute portlet requests, and to take a portlet out of service.

The portlet lifecycle methods can be identified through use of the Portlet, EventPortlet, HeaderPortlet, and ResourceServingPortlet interfaces or through use of annotations. This specification describes the portlet interfaces and associated artifacts beginning in *Chapter 4, Portlet Lifecycle Interfaces.* The concepts and artifacts introduced through these interfaces define and underlie portlet behavior.

Portlet Specification 3.0 provides a new capability of identifying portlet lifecycle methods through use of annotations. The Portlet Specification describes this capability in Section 4.8 Extended Annotation-Based Dispatching.

## Portlets and Web Frameworks

The portlet model provides a clear separation of the state-changing logic that is embedded in the preparation stage and the rendering of markup which is performed during the markup and resource stages. The portlet model thus follows the popular Model-View-Controller pattern which separates the controller logic from the part that generates the view.

Portlets use JavaServer Pages (JSP) as the native technology for rendering views. However, the Portlet Specification accommodates use of other frameworks, notably JavaServer™ Faces (JSF™), as view technologies. When using such a web framework, the portlet acts as a bridge between the portlet environment and the web framework.

# Portlet Lifecycle Interfaces

The Portlet interface is the main abstraction of the Portlet API. All portlets implement this interface either directly, by extending a class that implements the interface, or indirectly through method identification by annotation.

The portlet can optionally implement the additional life cycle interfaces EventPortlet, HeaderPortlet, and ResourceServingPortlet in order to coordinate with other portlets through events, to set header information, or to serve resources, respectively.

The Portlet API includes a GenericPortlet class that implements the Portlet, EventPortlet, HeaderPortlet, and ResourceServingPortlet interfaces and provides default functionality. Developers can extend the GenericPortlet class to implement their portlets.

The Portlet API includes a mechanism for identifying portlet lifecycle methods through use of annotations. The portlet methods identified in this manner need not implement the portlet interfaces named above, but each portlet method annotation imposes method signature requirements that must be met.

## Portlet Identification

The portlet configuration along with the portlet method annotations define the portlets available in the portlet application. Each portlet in the portlet application is uniquely identified by the portlet name. The portlet name must be a non-empty string that does not contain leading or trailing white space characters.

## Portlet Life Cycle Methods

The portlet container manages the portlet through a well-defined lifecycle that defines how it is loaded, instantiated, initialized, handles requests from clients, and is taken out of service.

The Portlet interface init and destroy methods are used to initialize the portlet and take it out of service, respectively. The corresponding method annotations are @InitMethod and @DestroyMethod.

The request processing methods are the Portlet interface processAction and render methods, the HeaderPortlet interface renderHeader method, the EventPortlet interface processEvent method, and the ResourceServingPortlet interface serveResource method.

The corresponding portlet request processing method annotations are @ActionMethod, @RenderMethod, @HeaderMethod, @EventMethod, and @ServeResourceMethod, respectively. The portlet lifecycle methods that are not implemented by a portlet class but are instead identified through these annotations are referred to as annotated portlet lifecycle methods.

The portlet method annotations named above may in some cases be used to annotate methods defined through the portlet interfaces in order to contribute configuration data. However, their primary purpose is to identify portlet methods contained within managed beans, as will be discussed in Section 4.8 Extended Annotation-Based Dispatching on page 33.

Since the portlet lifecycle methods may be identified either through implementation of an interface or through application of an annotation, discussion in this chapter will refer to the lifecycle methods generically using the names init, destroy, action, event, render, header, and resource when it is not necessary to be more specific.

### Loading and Instantiation

The portlet container is responsible for loading and instantiating portlets. The loading and instantiation can occur when the portlet container starts the portlet application, or can be delayed until the portlet container determines the portlet is needed to service a request.

The portlet container must load the portlet class using the same ClassLoader the servlet container uses for the web application part of the portlet application. After loading the portlet classes, the portlet container instantiates them for use.

For portlets not contained in valid CDI bean archives, the portlet container may instantiate only one portlet object per unique portlet definition per virtual machine (VM). If the portlet is contained in a CDI bean archive, the portlet container must instantiate the classes containing portlet methods through the CDI container.

See also Section 20.1 Portlet Instantiation on page 119 for information on portlet instantiation when a CDI container is present.

### Initialization

After the portlet object is instantiated, the portlet container must initialize the portlet before invoking it to handle requests. Initialization is provided so that portlets can initialize costly resources (such as backend connections), and perform other one-time activities. The portlet container must initialize the portlet object by calling the init method with a unique (per portlet definition) object implementing the PortletConfig interface. This configuration object provides access to the initialization parameters and the ResourceBundle defined in the portlet configuration. Refer to Chapter 6 The PortletConfig Interface on page 45 for information about the PortletConfig interface. The configuration object also gives the portlet access to a context object that describes the portlet’s runtime environment. Refer to Chapter 7 Portal Context on page 48 for information about the PortalContext interface.

#### Error Conditions on Initialization

During initialization, the portlet object may throw an UnavailableException or a PortletException. In this case, the portlet container must not place the portlet object into active service and must release the portlet object. The portlet container must not call the destroy method because the initialization is considered to have been unsuccessful.

The portlet container may reattempt to instantiate and initialize the portlets at any time after a failure. The exception to this rule is when an UnavailableException indicates a minimum time of unavailability. When this happens the portlet container must wait for the specified time to pass before creating and initializing a new portlet object.

The portlet container must handle a RuntimeException thrown during initialization as a PortletException.

#### Tools Considerations

The triggering of static initialization methods when a tool loads and introspects a portlet application is to be distinguished from initialization performed by calling the init method. Developers should not assume that a portlet is in an active portlet container runtime until the init method is called. For example, a portlet should not try to establish connections to databases or Enterprise JavaBeans™ containers during static (class) initialization.

### End of Service

The portlet container is not required to keep a portlet loaded for any particular period of time. A portlet object may be kept active in a portlet container for a period of milliseconds, for the lifetime of the portlet container (which could be a number of days, months, or years), or any amount of time in between.

When the portlet container determines that a portlet should be removed from service, it calls the destroy method to allow the portlet to release any resources it is using and to save any persistent state. For example, the portlet container may do this when it wants to conserve memory resources or when it is being shut down.

Before the portlet container calls the destroy method, it should allow any threads that are currently processing requests within the portlet object to complete execution. To avoid waiting forever, the portlet container can optionally wait for a container-defined time period before destroying the portlet object.

Once the destroy method is called on a portlet object, the portlet container must not route any requests to that portlet object. Ifthe portlet container needs to enable the portlet again, it must do so with a new portlet object, which is a new instance of the portlet’s class.

Ifthe portlet object throws a RuntimeException within the execution of the destroy method the portlet container must consider the portlet object to be successfully destroyed.

After the destroy method completes, the portlet container must release the portlet object so that it is eligible for garbage collection. Portlet implementations should not use finalizers.

## Portlet Customization Levels

The portlet model leverages the flyweight pattern and provides the portlet instance with all required request-specific data during each portlet request execution phase. This keeps the number of portlet instances small and thus allows better scalability for large numbers of users. The portlet programming model allows for different levels of customization. In order to distinguish between the different levels of customization, this section introduces the terms portlet definition, portlet entity, and portlet window.

### Portlet Definition and Portlet Entity

The portlet definition is a set of configuration properties that govern the behavior of the deployed portlet. The portlet definition is identified by the unique portlet name. Property values are provided through the portlet configuration (see Chapter 28 Configuration). The portlet definition may include default values for the portlet preferences, which are name-value pairs with portlet-specific meaning (seeChapter 18Portlet Preferences).

The portlet entity is a portlet instance associated with a set of portlet preferences. A given portlet instance may be associated with multiple distinct sets of portlet preferences, resulting in multiple portlet entities for that portlet instance. The portlet container manages the unique portlet entities in an implementation-specific manner.

The portlet preferences customize portlet behavior and influence the content the portlet produces. The portlet may read, modify and add preferences.

The portal builds an initial set of portlet preferences based on the values from the portlet definition. A portal may provide means to create new sets of portlet preferences or to modify existing ones. Administration, management, and configuration of portlet preferences along with definition of advanced customization features, such as hierarchical management of portlet preferences or cascading changes to portlet preferences, is left to the portal implementation.

### Portlet Window

A portlet window is a portlet entity uniquely located on a portal page. A given portlet entity may appear on multiple portal pages or multiple times on the same page, resulting in the portlet entity appearing in multiple portlet windows.

The portlet container assigns a unique ID to each portlet window. The ID is constant, valid for the lifetime of the portlet window, and is used by the portlet container to key the portlet-scoped session data. The portlet can access the portlet window ID using the PortletRequest.getWindowID method.

A portal may manage the portlet preferences associated with a given portlet window in an implementation-specific hierarchical manner, but during portlet method execution, the portlet container must provide an aggregated set of portlet preferences to the portlet.

From a developer's perspective, portlet windows are important because they define distinct runtime views. The portlet container maintains the render state and the portlet-scoped session state based on the portlet window. Hence the same portlet entity appearing in different portlet windows may be associated with different render state and portlet session-scoped data.

## Portlet Class

With the introduction of the portlet method annotations listed in Section 4.2, it is no longer strictly necessary to identify a portlet class. The annotated portlet lifecycle methods may reside in any valid CDI managed bean class. It is possible to write a portlet using only such annotated methods.

However, if the developer implements a portlet by extending the GenericPortlet class or by implementing the portlet lifecycle interfaces directly without using the portlet method annotations, the portlet class must be identified to allow the portlet container to perform dispatching.

The portlet container must recognize a class as being a portlet class if it implements, directly or indirectly, the Portlet interface and is also identified as the portlet class associated with a portlet name in the portlet configuration. The portlet class may optionally implement the remaining portlet lifecycle interfaces HeaderPortlet, EventPortlet, and ResourceServingPortlet.

The developer can identify the portlet class in the configuration either by applying the @PortletConfiguration annotation to a class that implements the Portlet interface, or by naming the class in the portlet deployment descriptor using the <portlet-class> element. See Section 28.2.4 Portlet Identification on page 195 for further information.

## Example Request Sequence

The following figures show an example request processing sequence. Figure 4–1 illustrates a typical sequence for a buffering portal, which collects the portlet output in buffers before aggregating the page. Figure 4–2 illustrates a typical sequence for a streaming portal, which begins streaming markup to the client as early as possible.

In both Figure 4–1 and Figure 4–2, an action initiated by the client through interaction with portlet B results in preparation stage execution and an action method call to portlet B. During action phase processing, portlet B updates its render state appropriately and publishes an event. The portlet container routes the event to portlet A by calling its event method. Portlet B updates its render state and returns.

Finally, the portlet container executes the aggregation stage for each of the portlets.

Figure 4–1 Example Request Sequence – Buffering Portal

As shown in Figure 4–1, a buffering portal may execute the header and render phases within the aggregation stage for each portlet sequentially. Although the diagram shows aggregation stage execution for each portlet as being sequential, the Portlet Specification places no such requirement. The aggregation stages for each portlet can be invoked in any order or even in parallel as long as the portlet container assures that the header phase for each portlet is executed before the overall portal response has been committed.

As shown in Figure 4–2, a streaming portal will typically execute the header phase for all portlets before executing the render phase of any portlet in order to allow each portlet to contribute to the headers and document head section markup before the response is committed.

Header phase execution for each portlet can be invoked in any order or even in parallel with the header phase execution of other portlets. Similarly, render phase execution for each portlet can be invoked in any order or even in parallel with the render phase execution of other portlets.



Figure 4–2 Example Request Sequence – Streaming Portal

## Request Handling

After a portlet object is properly initialized, the portlet container may invoke the portlet to handle client requests. The portlet container invokes the portlet request methods in the context of the corresponding portlet request processing phase, see Section 3.7 Portlet Request Processing Phases on page 14. The portlet container executes the portlet request processing phase according to a fixed logical portlet request processing sequence (the actual implementation may differ, but the principle must remain the same).

1. The portlet container prepares the portlet request and response objects for the portlet request processing phase. These objects govern execution of the entire portlet request processing phase even when the portlet container invokes more than one portlet request method during that phase. The table below summarizes the request and response objects for each portlet request processing phase, indicates whether multiple method invocations are allowed, and identifies the corresponding portlet interface lifecycle method.

| **Phase** | **Request Object** | **Response Object** | **Multiple Methods Allowed** | **Lifecycle Interface Method** |
| --- | --- | --- | --- | --- |
| Action | ActionRequest | ActionResponse | No | processAction |
| Event | EventRequest | EventResponse | No | processEvent |
| Header | HeaderRequest | HeaderResponse | Yes | renderHeaders |
| Render | RenderRequest | RenderResponse | Yes | render |
| Resource | ResourceRequest | ResourceResponse | Yes | serveResource |

1. The portlet container makes portlet artifacts available for injection, see Section 20.3 Portlet Predefined Beans on page 121.
2. The portlet container invokes any configured portlet filters for the request, see Chapter 16 Portlet Filters on page 100.
3. The portlet container determines the portlet methods to be invoked.

If the portlet uses annotated portlet lifecycle methods, there may be multiple methods for invocation, as will be discussed shortly. Otherwise, only the corresponding portlet class lifecycle method will be invoked.

1. If annotated portlet lifecycle methods are present, the portlet container invokes them according to the rules defined in Section 4.8 Extended Annotation-Based Dispatching on page 33.
2. Otherwise, if the portlet class method corresponding to the execution phase is present, that method is invoked.
3. If no portlet class method can be found for the execution phase, the portlet container should recognize the error and handle it in an appropriate manner.

This can occur, for example, if a portlet that does not implement the ResourceServingPortlet interface renders a link using a resource URL and that link is activated.

### Action Lifecycle Method

During action phase processing, a portlet can update the render state based on the information provided through the action request parameters.

The action method can access the ActionRequest and the ActionResponse objects.

The ActionRequest object provides access to information such as the render state, the action parameters, the portal context, the portlet session, and the portlet preferences data.

While executing the action method, the portlet may instruct the portlet container to redirect the user to a specific URL. If the portlet issues a redirection, when the action method concludes, the portlet container must send the redirection back to the user agent and abort the preparation stage processing without executing the event phase for any portlet.

A portlet may change its render state consisting of portlet mode, window state, and render parameters during an action request using the ActionResponse object.

The change of portlet mode must be effective for subsequent portlet request execution phases. There are some exceptional circumstances, such as changes of access control privileges that could prevent the portlet mode change from happening.

The change of window state must be effective for subsequent portlet request execution phases if allowed by the portlet container. The portlet should not assume that the subsequent request will be in the window state set as the portlet container can override the window state due to implementation dependencies between portlet modes and window states.

The portlet may delegate the action processing to a servlet via a request dispatcher call (see Chapter 25 Dispatching to JSPs and Servlets).

The portlet may publish events using the ActionResponse.setEvent method and thus publish notifications to other portlets. See Chapter 17 Coordination between Portlets for more details on sending and receiving events.

### Event Lifecycle Method

Events can be used to coordinate state between different portlets. The event method can access the EventRequest and the EventResponse objects.

The EventRequest object provides access to information such as the event payload, the render state, the portal context, the portlet session, and the portlet preferences data.

A portlet may change its render state consisting of portlet mode, window state, and render parameters during an event request using the EventResponse object.

The change of portlet mode must be effective for subsequent portlet request execution phases. There are some exceptional circumstances, such as changes of access control privileges that could prevent the portlet mode change from happening.

The change of window state must be effective for subsequent portlet request execution phases if allowed by the portlet container. The portlet should not assume that the subsequent request will be in the window state set as the portlet container can override the window state due to implementation dependencies between portlet modes and window states.

The portlet may delegate the event processing to a servlet via a request dispatcher call (see Chapter 25 Dispatching to JSPs and Servlets).

The portlet may publish events via the EventResponse.setEvent method and thus publish state changes or other notifications to other portlets. See Chapter 17 Coordination between Portlets for more details on sending and receiving events.

### Header Lifecycle Method

The header method may access the HeaderRequest and HeaderResponse objects.

The HeaderRequest object provides access to information such as the render state, the portal context, the portlet session, and the portlet preferences data.

The portlet can set properties on the HeaderResponse object that cause the portal application to set corresponding HTTP header values on the overall portal response.

The portlet can produce content for the portal page document HEAD section using the PrintWriter object obtained from the RenderResponse or it may delegate the generation of content to a servlet or a JSP. Refer to Chapter 25 Dispatching to JSPs and Servlets for details.

The portlet should produce content such as <style> and <meta> tags that are valid for the HEAD section. The portlet container may filter or restrict HEAD section content. For example, the portlet container may disallow the <title> tag, as it could conflict with portal markup.

The portlet may not update the render state and should not trigger any backend state changes during header method execution. The header method invocation should be a safe operation as defined by the HTTP specification[[3]](#footnote-3).

### Render Lifecycle Method

Generally portlets generate content based on the current render state during render request execution. The render method may access the RenderRequest and RenderResponse objects.

The RenderRequest object provides access to information such as the render state, the portal context, the portlet session, and the portlet preferences data.

The portlet can produce content for the portal page document BODY section using the PrintWriter object obtained from the RenderResponse or it may delegate the generation of content to a servlet or a JSP. Refer to Chapter 25 Dispatching to JSPs and Servlets for details on this.

The portlet may not update the render state and should not trigger any backend state changes during render method execution. The render method invocation should be a safe operation as defined by the HTTP specification3.

### Resource Lifecycle Method

The resource method may access the ResourceRequest and ResourceResponse objects.

The ResourceRequest object provides access to information such as the render state, the portal context, the portlet session, the resource parameters, and the portlet preferences data.

The portlet can produce content using the ResourceResponse writer or output stream, or it may delegate the generation of content to a servlet or a JSP. Refer to Chapter 25 Dispatching to JSPs and Servlets for details on this.

The resource phase is logically tied to the preceding render phase through the render state. The portlet may not change the render state during resource phase processing. However, the portlet may make changes to information stored in the portlet session or stored as portlet preferences.

If the resource phase was initiated through the HTTP GET method, the portlet should not change any information stored as portlet preferences or stored in the portlet session.

If changes to such information is required, the resource phase should be initiated using a HTTP POST, PUT, or DELETE method when triggering the resource URL. The portlet should not change information that is shared with other portlets, such as portlet session scoped data or portlet preferences. Otherwise other portlets using the shared information could display stale markup. The portlet developer should note that such state changes impact cacheability of the resource and set the cacheability settings accordingly. See Section 13.2 The Resource URL for detailed discussion about cacheability.

Due to the potential for causing unintended side effects, updating data in the resource phase must be done with care.

### Multithreading Issues During Request Handling

The portlet container handles concurrent requests to the same portlet by concurrent execution of the request handling methods on different threads. Portlet developers must design their portlets to handle concurrent multithread execution of the action, event, header, render, and resource lifecycle methods.

### Exceptions During Request Handling

A portlet may throw a PortletException, a PortletSecurityException, or a UnavailableException during portlet request phase processing.

A PortletException signals that an error has occurred during portlet request phase processing and that the portlet container should take appropriate measures to clean up.

If a portlet throws an exception during action phase processing, the portlet container must ignore all operations on the ActionResponse including any events that were set.

If a portlet throws an exception during event phase processing, the portlet container must ignore all operations on the EventResponse including events that were set. Operations performed during the originating action request processing or during separate, successful event request processing cycles for the current or other portlets must remain unaffected. After the exception has been handled, the portlet container should continue processing other portlets on the portal page.

If a portlet throws an exception during render phase processing, any headers set and any data written by the portlet to the response that has not yet been flushed to the portal application should be cleared (see Section 15.5.3 Buffering on page 93 for a discussion on buffer handling). Data written by the portal itself or by other portlets as part of the overall client response must not be affected. The portlet container should continue processing other portlets on the portal page.

If a portlet throws an exception during resource request processing, any headers set and any data written by the portlet to the response that has not yet been flushed to the portal application should be cleared. The portlet container should respond to the client request with an appropriate HTTP status code.

A PortletSecurityException indicates that the request has been aborted because the user does not have sufficient rights. Upon receiving a PortletSecurityException, the portlet container should handle this exception in an appropriate manner.

An UnavailableException signals that the portlet is unable to handle requests either temporarily or permanently.

Ifthe UnavailableException indicates permanent unavailability, the portlet container must remove the portlet from service immediately, call the portlet’s destroy method, and release the portlet object. A portlet that throws a permanent UnavailableException must be considered unavailable until the portlet application containing the portlet is restarted.

Whenthe UnavailableException indicates temporary unavailability, the portlet container may choose not to route any requests to the portlet during the time period of the temporary unavailability.

The portlet container may choose to ignore the distinction between a permanent and temporary unavailability and treat any UnavailableException as permanent and remove a portlet object that throws any UnavailableException from service.

A RuntimeException thrown during request handling must be handled as a PortletException.

The manner in which the exception is communicated to the end user is left to the portal implementation.

### Thread Safety

Implementations of the portlet request and response objects are not guaranteed to be thread safe. This means that they must only be used within the scope of the thread invoking the portlet request handling methods.

Portlet applications should not pass references to the portlet request and response objects to other threads as the resulting behavior may be non-deterministic.

## GenericPortlet

The GenericPortlet abstract class provides default functionality and convenience methods for handling portlet requests. Portlets implemented through extending the GenericPortlet class are also more robust against future changes in the Java Portlet Specification as such changes can be mitigated by the GenericPortlet implementation. The GenericPortlet class implements the Portlet, HeaderPortlet, EventPortlet and ResourceServingPortlet interfaces.

The following sections describe the functionality provided by the GenericPortlet class

### Dispatching to GenericPortlet Annotated Methods

The GenericPortlet class will attempt to dispatch render, action, and event requests to generic portlet annotated methods based on the portlet mode, action name, and event name, respectively. The annotations processed by GenericPortlet are distinct from the portlet lifecycle method annotations mentioned in Section 4.2 Portlet Life Cycle Methods discussed above.

The GenericPortlet annotated method must be a public method within the class hierarchy of a class extending GenericPortlet and must be specified as the portlet class by the portlet configuration. The method must carry one of the following annotations:

* Action method: @ProcessAction(name=<action name>). The action method must have the following signature:

**public void <method name> (ActionRequest, ActionResponse) throws PortletException, java.io.IOException;**

* Render method: @RenderMode(name=<portlet mode name>). The render method must have the following signature:

**public void <method name> (RenderRequest, RenderResponse) throws PortletException, java.io.IOException;**

* Event method: @ProcessEvent(qname=<event name>). The event method must have the following signature:

**public void <method name> (EventRequest, EventResponse) throws PortletException, java.io.IOException;**

### Action Dispatching

When an action request is received, the processAction method in the GenericPortlet class reads the value of the action parameter with the name "javax.portlet.action" (represented by the constant ActionRequest.ACTION\_NAME). If this parameter is not null, the processAction method attempts to locate an action method carrying a @ProcessAction annotation that has a name element equal to the action parameter value. If such a method is found, it will be invoked. Otherwise, no operation will be performed.

### Event Dispatching

When an event request is received, the processEvent method in the GenericPortlet class reads the event qname and attempts to locate an event method carrying a @ProcessEvent annotation that has a qname element equal to the event qname. If such a method is found, it will be invoked. Otherwise, no operation will be performed.

### Resource Serving Dispatching

By default the serveResource method in the GenericPortlet class does nothing.

However, if a portlet initialization parameter with the reserved name "javax.portlet.automaticResourceDispatching" is set to true, the GenericPortlet serveResource method will attempt to forward the request to the resource ID set on the URL triggering the resource request. If no resource ID is set, the serveResource method does nothing.

Since GenericPortlet performs a Request Dispatcher forward, portlets extending GenericPortlet and wishing to use the automatic resource request dispatching function should override the serveResource method in order to verify the resource ID before calling the super.serveResource method for automatic dispatching.

### Header Dispatching

The default renderHeaders method does nothing.

### Render Dispatching

*Note: The support for the javax.portlet.renderHeaders portlet container runtime option described here and also in Section 8.4.2 Runtime Option javax.portlet.renderHeaders on page 50 pertains only to Version 2.0 portlets for compatibility purposes. The portlet container must recognize version 2.0 portlets based on the version information contained in the portlet deployment descriptor. For Version 3.0 portlets, the* RENDER\_PART *request attribute will not be set. Version 3.0 portlets must implement the HeaderPortlet interface renderHeaders method or a corresponding @HeaderMethod annotated method to write headers or to write output to the portal response document HEAD section.*

In order to allow portlets running on streaming portals to set header data, the portlet can set the javax.portlet.renderHeaders container runtime to true. A portlet container supporting this container runtime option will set the RENDER\_PART render request attribute to indicate that header requests will be processed. If the RENDER\_PART portlet request attribute is set, it indicates that the render request will be processed twice in order to allow portlets to set header information and document HEAD section data.

When the portlet container calls the GenericPortlet render method with the RENDER\_PART request attribute set to RENDER\_HEADERS, the GenericPortlet render method will invoke the doHeaders method. The default doHeaders method does nothing. Portlets extending GenericPortlet can override the doHeaders method in order to set headers information.

When the portlet container calls the GenericPortlet render method with the RENDER\_PART request attribute set to RENDER\_MARKUP, or if the RENDER\_PART attribute is not set, the GenericPortlet render method sets the title specified in the portlet definition in the deployment descriptor and invokes the doDispatch method.

The doDispatch method in the GenericPortlet class implements functionality to aid in the processing of requests based on the portlet mode the portlet is currently in (see Chapter 9 Portlet Modes on page 55).

First the doDispatch reads the portlet mode and attempts to locate a render method carrying a @RenderMode annotation that has a name element matching the portlet mode. If such a method is found, it will be invoked.

If no matching annotated method is found, GenericPortlet will dispatch to the following methods:

* doView for handling VIEW requests
* doEdit for handling EDIT requests
* doHelp for handling HELP requests

For any other portlet mode the GenericPortlet will throw a PortletException.

Ifthe window state of the portlet (see Chapter 10 Window States) is MINIMIZED, the render method of GenericPortlet does not invoke any of the portlet mode rendering methods.

Typically, portlets will extend the GenericPortlet class directly or indirectly and they will either use the @RenderMode annotation or override the doView, doEdit, doHelp and getTitle methods instead of the render and doDispatch methods.

## Extended Annotation-Based Dispatching

Portlet Specification 3.0 introduces a new request dispatching mechanism that extends the GenericPortlet-based dispatching concepts introduced in Section 4.7.1. The extended dispatching mechanism allows portlet methods to appear in any valid CDI managed bean provided by the portlet.

The managed bean containing the portlet method or methods is not required to implement a particular interface. However, each portlet method annotation imposes method signature requirements on the annotated method. If a portlet method annotation is applied to a method that does not meet the method signature requirement, the portlet container must not place the portlet in service. The appropriate error handling and message display is left as a portal implementation detail.

A portlet application may contain many portlets. The portlets within the portlet application are identified by a unique portlet name. Since the extended annotation-based dispatching mechanism emancipates the portlet methods from the portlet class, the developer must specify the portlet name as an annotation element. If the annotation applies to a single portlet only, the annotation contains a required String portletName element for that purpose. The portlet container must invoke the annotated method when processing a corresponding request for the portlet identified by the portlet name.

Some annotations can apply to more than one portlet within the portlet application. In this case, the portlet method annotation will contain a required String[] portletNames element. The portlet container must invoke the annotated method when processing a corresponding request for a portlet whose name is contained in the portletNames element.

If the first array entry of the portletNames annotation element contains the reserved string "\*", the annotated method must be invoked for all portlets defined in the portlet application.

Use of one of the portlet method annotations @RenderMethod, @HeaderMethod, or @ServeResourceMethod can define a portlet. The portlet container must register and make available for portal use all portlets whose names are defined in these annotations. However, the reserved string "\*" as a portlet name does not define a portlet.

Each portlet method annotation supports a method that accepts the native request and response types as arguments. Some portlet method annotations offer simplified method signatures. If the portlet uses a simplified method signature, it can obtain the request and response objects along with related portlet artifacts through the dependency injection mechanism. See Section 20.3 Portlet Predefined Beans on page 121 for a list of injectable portlet artifacts.

Annotated methods participating in the extended annotation-based dispatching mechanism need only declare a throws clause for exceptions that the method actually throws. If the method does not throw a checked exception, no throws clause is required. However, if a throws clause is present, the declared checked exceptions must be of type PortletException or IOException. If the annotated method declares any other type of checked exception in the throws clause, the portlet container must not place the portlet in service.

The method annotations @RenderMethod, @HeaderMethod, @ServeResourceMethod, @ActionMethod, and @EventMethod can be applied to methods within a class that extends the Portlet interface and is configured as a portlet class. This can be useful when the portlet requires multiple render methods, for example. If this is done, then the portlet container must assure that the annotated methods execute against the same single instance of the portlet class that the container instantiates for the portlet class methods.

If the method annotations are applied to methods of a class that does not extend the Portlet interface, then the portlet container should execute the methods against class instances according to CDI container rules.

The following sections describe how the portlet container dispatches requests to the annotated portlet lifecycle methods.

### Initialization and Destruction

The @InitMethod annotation designates a method corresponding to the Portlet interface init method. The annotation defines a single String attribute representing the portlet name. The annotated method must have the following signature:

public void <methodName>(PortletConfig) throws PortletException

where the method name can be freely selected.

The portlet container calls the annotated method to initialize the portlet when it is being placed into service. The portlet container calls the init method exactly once after instantiating the portlet but before request processing begins, regardless of the scope of the bean in which the annotated method is located. It is recommended that the method be placed in an @ApplicationScoped or a @Dependent-scoped bean. The initialization method must complete successfully before the portlet can receive any requests.

The portlet container cannot place the portlet into service if an annotated init method throws a PortletException or does not return within a time period defined by the portlet container. A portlet may optionally have a single init method.

The @DestroyMethod annotation designates a method corresponding to the Portlet interface destroy method. The annotation defines a single String attribute representing the portlet name. The annotated method must have the following signature:

public void <methodName>()

where the method name can be freely selected.

The portlet container calls the destroy method to indicate to a portlet that the portlet is being taken out of service.

Before the portlet container calls the destroy method, it should allow any threads that are currently processing requests within the portlet object to complete execution. To avoid waiting forever, the portlet container can optionally wait for a predefined time before destroying the portlet object. It is recommended that the method be placed in an @ApplicationScoped or a @Dependent-scoped bean.

This method enables the portlet to clean up any resources that it holds (for example, memory, file handles, threads) or to make sure that any persistent state is synchronized with the portlet current state in memory. A portlet may optionally have a single destroy method.

### Annotated Action Method Dispatching

The @ActionMethod annotation designates an action request processing method and corresponds to the Portlet interface processAction method. The annotated method must have the following signature:

public void <methodName>(ActionRequest, ActionResponse)

The method name can be freely selected.

The @ActionMethod annotation contains an actionName element which is used for more finely-grained request dispatching. A given portlet may define multiple @ActionMethod annotated methods that differ in the value of the actionName element. If the portlet defines more than one @ActionMethod annotated method with the same value of actionName, the portlet container must not place the portlet in service.

The portlet container must dispatch the @ActionMethod annotated method as follows:

1. When the portlet container receives a request from the client, it must check the value of the action parameter carrying the name "javax.portlet.action" (represented by the ActionRequest.ACTION\_NAME string constant). If a @ActionMethod annotated method exists whose actionName element matches the parameter value, the portlet container must dispatch the action request to that method.
2. If no such method exists, the portlet container must dispatch the request to a @ActionMethod annotated method containing an empty actionName element ("", the default).
3. If no method is defined whose @ActionMethod actionName element is empty, and if a portlet class implementing the Portlet interface is configured, then the portlet container must invoke the processAction method.
4. Otherwise the portlet container must do nothing.

The effect of this procedure for portlets that extend GenericPortlet is that the existence of a matching @ActionMethod annotated method will override the GenericPortlet processAction functionality.

The @ActionMethod annotation contains an optional publishingEvents array element that allows the portlet to specify the qualified names for the portlet events it can fire during method execution. The qualified names must be defined in the portlet application configuration. The portlet container must register the publishing events defined in this array as valid publishing events for the portlet. See Section 28.1.8 Event Configuration on page 190 and Section 28.2.10 Event References on page 201 for more information on event configuration. See Chapter 17 Coordination between Portlets on page 106 for more information on event processing.

The @ActionMethod annotation defines the following elements.

| **Element** | **Description** |
| --- | --- |
| portletName | The required portlet name element for which the annotated method applies. |
| actionName | String representing the action name. The default value is the empty string, "". The actionName must be unique for a given portletName. |
| publishingEvents | Array of PortletQName elements representing the events published during method processing. |

Example:

@ActionMethod(portletName="BeanPortlet", actionName="setName")

public void setName(ActionRequest req, ActionResponse resp)

throws PortletException, IOException {

...

}

### Annotated Event Method Dispatching

The @EventMethod annotation designates an event request processing method and corresponds to the Portlet interface processEvent method. The annotated method must have the following signature:

public void <methodName>(EventRequest, EventResponse)

The method name can be freely selected.

The @EventMethod annotation contains a required processingEvents element that defines the qualified names of the events the method can process. The qualified names must be defined in the portlet application configuration. The portlet container must register the processing events defined in this array as valid processing events for the portlet.

See Section 28.1.8 Event Configuration on page 190 and Section 28.2.10 Event References on page 201 for more information on event configuration. See Chapter 17 Coordination between Portlets for more information on event processing.

A given portlet may define multiple @EventMethod annotated methods that differ in the values provided by the processingEvents element. The values provided must be unique for the portlet. If the portlet defines duplicate entries for this element, or defines more than one @EventMethod annotated method with a given processingEvents element qualified name value, the portlet container must not place the portlet in service.

The portlet container dispatches event requests to @EventMethod annotated methods based on the configured values as follows:

1. The portlet container must determine that a given event is to be routed to the portlet. This is an implementation-specific determination.
2. The portlet container must route the event request to the @EventMethod annotated method that has a matching processingEvents entry.
3. If no such method exists, and if a portlet class implementing the EventPortlet interface is configured, then the portlet container must invoke the processEvent method.
4. Otherwise the portlet container must do nothing.

The effect of this procedure for portlets that extend GenericPortlet is that the existence of a matching @EventMethod annotated method will override the GenericPortlet class processEvent functionality.

The @EventMethod annotation contains a publishingEvents array element that allows the portlet to specify the qualified names for the portlet events it can fire during method execution. The qualified names must be defined in the portlet application configuration. The portlet container must register the publishing events defined in this array as valid publishing events for the portlet.

The @EventMethod annotation defines the following elements.

| **Element** | **Description** |
| --- | --- |
| portletName | The required portlet name element for which the annotated method applies. |
| processingEvents | Array of PortletQName elements representing the events processed by the method. |
| publishingEvents | Array of PortletQName elements representing the events published during method processing. |

Example:

@EventMethod(portletName="BeanPortlet", processingEvents= {

@PortletQName(

namespaceURI="http://www.apache.org/portals/pluto/ResourcePortlet",

localPart="Message")

})

public void processEvent(EventRequest req, EventResponse resp)

throws PortletException ,IOException {

...

}

### Annotated Header Method Dispatching

The @HeaderMethod annotation designates a header request processing method and corresponds to the GenericPortlet doHeaders method. The annotated method must have one of the following signatures:

|  |
| --- |
| **Method Signature** |
| public void <methodName>(HeaderRequest, HeaderResponse) |
| public String <methodName>() |
| public void <methodName>() |

The method name can be freely selected.

The purpose of this method is to allow the portlet to set HTTP headers, implementation-specific properties, and to allow the portlet to write markup to the overall document HEAD section. If the portlet defines @HeaderMethod annotated methods, the portlet container must invoke them before the overall portal application response is committed.

The portlet container should add any markup written to the given HeaderResponse object to the document HEAD section.

The @HeaderMethod annotation contains a portletMode element that the portlet container uses for request dispatching.

The portlet container must dispatch the @HeaderMethod annotated method as follows:

1. The portlet container must identify the @HeaderMethod annotated methods whose portletMode element matches the portlet mode of the current request and invoke them in the order determined by the ordinal number (see below).
2. If there is no such method, the portlet container must identify the @HeaderMethod annotated methods whose portletMode element is empty ("") and invoke them in the order determined by the ordinal number (see below).
3. If there is no such method, but if a portlet class implementing the HeaderPortlet interface is configured, then the portlet container must invoke the renderHeaders method.
4. Otherwise the portlet container must do nothing.

A given portlet may define multiple @HeaderMethod annotated methods that match the portlet mode information. In this case, the order of execution is determined by the ordinal element within the annotation. Annotated methods with a lower ordinal number are executed before methods with a higher ordinal number. If two annotated methods have the same ordinal number, both methods will be executed, but the execution order will be undetermined.

The annotated method can apply to multiple portlets within the portlet application. The names of the portlets for which the listener applies must be specified in the portletNames element. A wildcard character '\*' can be specified in the first portletName array element to indicate that the annotated method is to apply to all portlets in the portlet application. If specified, the wildcard character must appear alone in the first array element.

The @HeaderMethod annotation contains the following elements.

|  |  |  |
| --- | --- | --- |
| **Element** | **Description** | **Default** |
| portletNames | The portlet names for which the listener applies. | -none- |
| ordinal | The ordinal number for this annotated method. | 0 |
| portletMode | String value representing the portlet mode | "view" |
| contentType | The contentType for this request | "\*/\*" |

Example: Method applying to all portlets in the portlet application

@HeaderMethod(portletNames="\*")

public void header(HeaderRequest req, HeaderResponse resp) throws

throws PortletException ,IOException {

...

}

Example: Simplified method returning String

@HeaderMethod(portletNames="BeanPortlet")

public String header() throws PortletException ,IOException {

...

}

Example: Simplified method that includes a JSP

@HeaderMethod(portletNames="BeanPortlet", include="header.jsp")

public String header() throws PortletException ,IOException {

...

}

### Annotated Render Method Dispatching

The @RenderMethod annotation designates a render request processing method and corresponds to the Portlet interface render method. The annotated method must have one of the following signatures:

|  |
| --- |
| **Method Signature** |
| public void <methodName>(RenderRequest, RenderResponse) |
| public String <methodName>() |
| public void <methodName>() |

The method name can be freely selected.

If the @RenderMethod annotated method returns a String and if the returned string is not null, the portlet container must add the return value to the render response output stream.

The @RenderMethod annotation contains a portletMode element that the portlet container uses for request dispatching.

The portlet container must dispatch the @RenderMethod annotated method as follows:

1. The portlet container must identify the @RenderMethod annotated methods whose portletMode element matches the portlet mode of the current request and invoke them in the order determined by the ordinal number (see below).
2. If there is no such method, the portlet container must identify the @RenderMethod annotated methods whose portletMode element is empty ("") and invoke them in the order determined by the ordinal number (see below).
3. If there is no such method, but if a portlet class implementing the Portlet interface is configured, then the portlet container must invoke the render method.
4. Otherwise the portlet container must do nothing.

A given portlet may define multiple @RenderMethod annotated methods that match the portlet mode information. In this case, the order of execution is determined by the ordinal element within the annotation. Annotated methods with a lower ordinal number are executed before methods with a higher ordinal number. If two annotated methods have the same ordinal number, both methods will be executed, but the execution order will be undetermined.

The annotated method can apply to multiple portlets within the portlet application. The names of the portlets for which the listener applies must be specified in the portletNames element. A wildcard character '\*' can be specified in the first portletName array element to indicate that the listener is to apply to all portlets in the portlet application. If specified, the wildcard character must appear alone in the first array element.

The @RenderMethod annotation contains a contentType element. If this value is not empty, and does not contain the wildcard content type ('\*' or '\*/\*'), the portlet container must set the content type for the request prior to invoking the method according to the rules defined for the render request, see Section14.6 RenderRequest Interface. If this element is empty, the portlet container must do nothing.

The @RenderMethod annotation contains a portletMode element. This element is case-insensitive and must match one of the standard portlet mode values (view, help, or edit) or a configured custom portlet mode.

If the portlet configuration does not configure the portlet mode explicitly (see Section 28.2.12 Portlet Modes and Window States on page 202), the portlet container must implicitly configure the portlet mode given by the portletMode element using the content type specified by the contentType element.

If the portlet configuration configures the portlet mode, the implicit portlet mode configuration must not be carried out. This allows the explicit portlet configuration, for example specified using the portlet deployment descriptor, to override the implicit portlet mode configuration.

The @RenderMethod annotation contains an include element. If this element is not empty, the portlet container must first execute the body of the annotated method and then perform a request dispatcher include on the include element value.

The @RenderMethod annotation contains the following elements.

| **Element** | **Description** | **Default** |
| --- | --- | --- |
| portletNames | The portlet names for which the listener applies. | -none- |
| ordinal | The ordinal number for this annotated method. | 0 |
| portletMode | String value representing the portlet mode | "view" |
| contentType | The content type for this request | "\*/\*" |
| include | Specifies a JSP or servlet that will be included after the method body has been executed. | "" |

Example: Method applying to all portlets in the portlet application

@RenderMethod(portletNames="\*")

public void myView(RenderRequest req, RenderResponse resp) throws

throws PortletException ,IOException {

...

}

Example: Simplified method returning String

@RenderMethod(portletNames="BeanPortlet")

public String myView() throws PortletException ,IOException {

...

}

Example: Simplified method that includes a JSP

@RenderMethod(portletNames="BeanPortlet", include="view.jsp")

public String myView() throws PortletException ,IOException {

...

}

### Annotated Resource Method Dispatching

The @ServeResourceMethod annotation designates a resource request processing method and corresponds to the ResourceServingPortlet interface serveResource method. The annotated method must have one of the following signatures:

|  |
| --- |
| **Method Signature** |
| public void <methodName>(RenderRequest, RenderResponse) |
| public String <methodName>() |
| public void <methodName>() |

The method name can be freely selected.

If the @ServeResourceMethod annotated method returns a String and if the returned string is not null, the portlet container must add the return value to the resource response output stream.

The @ServeResourceMethod annotation contains a resourceID element that the portlet container uses for request dispatching.

The portlet container must dispatch the @ServeResourceMethod annotated method as follows:

1. The portlet container must identify the @RenderMethod annotated methods whose resourceID element matches the resource ID from the current request and invoke them in the order determined by the ordinal number (see below).
2. If there is no such method, the portlet container must identify the @RenderMethod annotated methods whose resourceID element is empty ("") and invoke them in the order determined by the ordinal number (see below).
3. If there is no such method, but if a portlet class implementing the ResourceServingPortlet interface is configured, then the portlet container must invoke the serveResource method.
4. Otherwise the portlet container must do nothing.

A given portlet may define multiple @ServeResourceMethod annotated methods that match the resource ID information. In this case, the order of execution is determined by the ordinal element within the annotation. Annotated methods with a lower ordinal number are executed before methods with a higher ordinal number. If two annotated methods have the same ordinal number, both methods will be executed, but the execution order will be undetermined.

The annotated method can apply to multiple portlets within the portlet application. The names of the portlets for which the listener applies must be specified in the portletNames element. A wildcard character '\*' can be specified in the first portletName array element to indicate that the listener is to apply to all portlets in the portlet application. If specified, the wildcard character must appear alone in the first array element.

The @ServeResourceMethod annotation contains a contentType annotation. If this value is not empty, the portlet container must set the content type for the request prior to invoking the annotated method according to the rules defined for the resource request, see Section 14.4 ResourceRequest Interface. If this element is empty, the portlet container must do nothing.

The @ServeResourceMethod annotation contains a characterEncoding annotation. If this value is not empty, the portlet container must set the character encoding for the request prior to invoking the annotated method according to the rules defined for the resource request, see Section 14.4 ResourceRequest Interface. If this element is empty, the portlet container must do nothing.

The @ServeResourceMethod annotation contains an include element. If this element is not empty, the portlet container must first execute the body of the annotated method and then perform a request dispatcher include on the element value.

The @ServeResourceMethod annotation contains the following elements.

| **Element** | **Description** | **Default** |
| --- | --- | --- |
| portletNames | The portlet names for which the listener applies. | -none- |
| ordinal | The ordinal number for this annotated method. | 0 |
| resourceID | String value representing the resource ID | "" |
| characterEncoding | The character encoding for this request | "UTF-8" |
| contentType | The content type for this request | "\*/\*" |
| include | Specifies a JSP or servlet that will be included after the method body has been executed. | "" |
| asyncSupported | Declares whether the serve resource method supports asynchronous operation mode. If this flag is set, any portlet to which this annotated method applies will be marked as supporting asynchronous operation. | false |

Example: Method applying to all portlets in the portlet application

@ResourceMethod(portletNames="\*")

public void myResource(ResourceRequest req, ResourceResponse resp) throws

throws PortletException ,IOException {

...

}

Example: Simplified method returning String

@ResourceMethod(portletNames="BeanPortlet")

public String myResource() throws PortletException ,IOException {

...

}

Example: Simplified method that includes a JSP

@ResourceMethod(portletNames="BeanPortlet", include="myResource.jsp")

public String myResource() throws PortletException ,IOException {

...

}

# Portlet Applications

A portlet application is a web application as defined in the *Servlet Specification.* It can contain one or more portlets identified by annotations or through the portlet deployment descriptor in addition to servlets, JSPs, HTML pages, Java classes and other resources normally found in a web application. A standard-conforming portlet application can run on standard portlet containers.

## Relationship with Web Applications

Portlets contained within the portlet application are managed by the portlet container. All other components and resources are managed by the servlet container underlying the portlet container.

Servlets, JSPTM pages, related Java classes, and static documents managed by the servlet container must be declared according to the *Servlet Specification 3.1*.

Portlets and other artifacts managed by the portlet container must be declared as described in this specification.

## Relationship to the Portlet Context

The portlet container must enforce a one-to-one correspondence between a portlet application and its portlet context. If the application is a distributed application, the portlet container must create a single PortletContext instance per VM. A PortletContext object provides a portlet with information about the application.

## Portlet Application Classloader

The portlet container must load the portlets and related resources within the portlet application with the same ClassLoader used by the underlying servlet container.

The portlet container must ensure that requirements defined in the servlets specification sections on ‘Dependencies on Extensions’ and ‘Web Application Class Loader’[[4]](#footnote-4) are fulfilled.

## Directory Structure and Portlet Application Archive File

A portlet application follows the same hierarchical directory structure as web applications.

Portlet classes, utility classes and other resources accessed through the portlet application class loader must reside within the /WEB-INF/classes directory or within a JAR file in the /WEB-INF/lib/ directory.

Portlet applications are packaged as web application archives (WAR) as defined in the *Servlet Specification[[5]](#footnote-5)* Chapter.

## Portlet Application Deployment Descriptor

In addition to a web application deployment descriptor, a portlet application may contain a /WEB-INF/portlet.xml deployment descriptor file. Configuration tasks can be carried out though use of annotations or through the portlet deployment descriptor. Configuration settings defined in the portlet deployment descriptor take precedence over corresponding settings from annotations.

Refer to Chapter 27 Packaging and Deployment for more details on the portlet application deployment descriptor.

## Replacing a Portlet Application

A portlet container should be able to replace a portlet application with a new version without restarting the container. In addition, the portlet container should provide a robust method for preserving session data for the portlet application when the portlet application is replaced.

## Error Handling

Error handling during request processing is left to the portlet container implementation.

For example, when an exception is thrown during request processing, the portal could render an error page instead of the portal page, render an error message in the portlet window of the portlet that threw the exception, or remove the portlet from the portal page and log an error message for the administrator.

## Portlet Application Environment

The Portlet Specification requires the execution environment described by the Servlet Specification as a prerequisite.[[6]](#footnote-6)

# The PortletConfig Interface

The PortletConfig object provides the portlet object with configuration information. The configuration can be taken from annotations or from the portlet deployment descriptor, see Chapter 28 Configuration. It also provides access to the portlet context, default event namespace, public render parameter names, and the resource bundle that provides title-bar resources.

## Initialization Parameters

The getInitParameterNames and getInitParameter methods of the PortletConfig interface return the initialization parameter names and values found in the portlet definition in the portlet configuration.

## Portlet Resource Bundle

The portlet configuration may specify some basic information that can be used for the portlet title-bar and for the portal’s categorization of the portlet. The specification defines a few resource elements for these purposes - the title, short-title, and keywords (see Section 28.2.5 Portlet Resource Bundle on page 196).

These resource elements can be directly included in the portlet configuration, or they can be placed in a resource bundle.

If the resources are defined in a resource bundle, the portlet configuration must provide the name of the resource bundle.

If the portlet configuration defines a resource bundle, the portlet container must look up these values in the ResourceBundle. If the root resource bundle does not contain the resources for these values and the values are defined in the configuration, the portlet container must add these values as resources of the root resource bundle. Values from the resource bundle have precedence over values defined directly in the configuration.

If the portlet definition does not define a resource bundle and the information is defined directly in the configuration, the portlet container must create a ResourceBundle and populate it with the directly defined values using the keys defined in Section 28.2.5 Portlet Resource Bundle.

The title, short title, keywords, and resource bundle elements are optional configuration elements.

If neither the title element nor a resource bundle element containing a tile are present, the portlet title will not be explicitly defined. In this case, it is left as a portlet container implementation detail to draw on other information, for example on the <portlet-name> or <display-name> values from the portlet descriptor, to define the portlet title.

The getResourceBundle method returns the resource bundle for a given Locale.

The render method of the GenericPortlet class uses the ResourceBundle object provided by the PortletConfig to retrieve the title of the portlet from the associated ResourceBundle or from the corresponding elements directly defined in the configuration.

## Default Event Namespace

The getDefaultNamespace method of the PortletConfig interface returns the default namespace for events and public render parameters set in the configuration or the XML default namespace XMLConstants.NULL\_NS\_URI if no default namespace is provided in the portlet configuration.

## Public Render Parameters

The getPublicRenderParameterNames method returns the public render parameter names defined in the portlet configuration or an empty enumeration if no public render parameters are defined for the current portlet definition.

The getPublicRenderParameterDefinitions method returns a map of the public render parameter names to their qualified names represented by a QName object or an empty map if no public render parameters are defined for the current portlet definition.

## Publishing Event QNames

The getPublishingEventQNames method of the PortletConfig interface returns the publishing event qualified names found in the portlet configuration or an empty enumeration if no publishing events are defined for the current portlet definition.

If the event was defined using the portlet deployment descriptor name element instead of the qname element, the defined default namespace must be added as namespace for the returned QName.

## Processing Event QNames

The getProcessingEventQNames method of the PortletConfig interface returns the processing event qualified found in the portlet configuration or an empty enumeration if no processing events are defined for the current portlet definition.

If the event was defined using the portlet deployment descriptor name element instead of the qname element, the defined default namespace must be added as namespace for the returned QName.

## Supported Locales

The getSupportedLocales method of the PortletConfig interface returns the supported locales found in the portlet configuration or an empty enumeration if no supported locales are defined for the current portlet definition.

## Supported Container Runtime Options

The getContainerRuntimeOptions method returns an immutable Map containing portlet application level container runtime options merged with the portlet level container runtime options. The runtime options defined at the portlet level take precedence over the values with the same names defined at the portlet application level.

Different portlet containers may support different sets of portlet container runtime options. The map returned by the getContainerRuntimeOptions method must only contain those values that are both defined for the portlet and supported by the portlet container. If no portlet container runtime options meet these criteria, the returned map must be empty.

The keys of the returned map are of type String. The values in the map are of type String array.

See Section 8.4 Portlet Container Runtime Options on page 50 for a list of all predefined container runtime options.

## Portlet Modes

The portlet configuration can specify the portlet modes supported by the portlet. The supported portlet modes may be a subset of the standard portlet modes, and may include custom portlet modes.

A portal implementation may define custom portlet modes that are managed by the portal. Portlets can declare additional custom portlet modes that are managed by the portlet. Custom portlet modes declared in the portlet descriptor as portal managed that are not supported by the portal are ignored.

The getPortletModes method returns an enumeration of the portlet modes available to the portlet. The enumeration includes the standard portlet modes supported by the portlet, the declared portal-managed modes supported by the portal, and the declared portlet-managed modes.

### Window States

The portlet configuration can specify the window states supported by the portlet. The supported window states always include the standard window states, and may include custom window states.

Portlets can declare support for custom window states in the deployment descriptor. Custom window states declared in the portlet descriptor that are not supported by the portal are ignored.

The getWindowStates method returns an enumeration of the window states available to the portlet. The enumeration includes the standard window states and the custom window states declared by the portlet and supported by the portal.

# Portal Context

The PortalContext interface provides read-only information about the portal that is invoking the portlet.

A portlet obtains a PortalContext object from the portlet request object using getPortalContext method.

## Portal Context Methods

The getPortalInfo method returns information such as the portal vendor and portal version.

The getProperty and getPropertyNames methods return portal properties.

The getSupportedPortletModes method returns all portlet modes supported by the portal. The portlet modes will include the standard portlet modes EDIT, HELP, VIEW.

The getSupportedWindowStates method returns all window states supported by the portal. The window states will include the standard window states MINIMIZED, NORMAL, MAXIMIZED.

## Support for Markup Head Elements

*If a portlet with a version 2.0 or earlier deployment descriptor is deployed, the portlet container must provide behavior described in this section in order to assure backward compatibility. For version 3.0 or later portlets, the behavior described in this section is to be disregarded.*

A portal can optionally support adding markup to the HTML document head section through use of the MimeResponse.setProperty method. The Portal should indicate support for this property through the PortalContext.MARKUP\_HEAD\_ELEMENT\_SUPPORT (value: "javax.portlet.markup.head.element.support") property on the PortalContext.

A non-null value of MARKUP\_HEAD\_ELEMENT\_SUPPORT indicates that the portal application supports the MARKUP\_HEAD\_ELEMENT property

# Portlet Context

The PortletContext interface provides a view of the portlet application within which the portlet is running. Using the PortletContext object, a portlet can log events, obtain portlet application resources, portlet application and portlet-specific runtime options, and set and store attributes that other portlets and servlets in the portlet application can access.

## Scope of the Portlet Context

There is one instance of the PortletContext interface associated with each portlet application deployed into a portlet container. In cases where the container is distributed over many virtual machines, a portlet application will have an instance of the PortletContext interface for each VM.

## Portlet Context functionality

The portlet can access context initialization parameters, retrieve and store context attributes, obtain static resources from the portlet application, and obtain a request dispatcher to include servlets and JSPs through the PortletContext interface.

## Relationship to the Servlet Context

A portlet application is an extended web application. As a web application, a portlet application also has a servlet context. The portlet context uses the servlet context object to provide much of its functionality. However, the portlet context object is distinct from the servlet context object.

The context-wide initialization parameters are the same as initialization parameters of the servlet context and the context attributes are shared with the servlet context.

Therefore, the portlet context initialization parameters must be defined in the web application deployment descriptor (the web.xml file). The initialization parameters accessible through the PortletContext must be the same as those accessible through the ServletContext of the portlet application.

Context attributes set through the PortletContext must be stored in the ServletContext of the portlet application. A direct consequence of this is that data stored in the ServletContext by servlets or JSPs is accessible to portlets through the PortletContext and vice versa.

The PortletContext must offer access to the same set of resources the ServletContext exposes.

The PortletContext must handle the same temporary working directory the ServletContext handles. It must be accessible as a context attribute using the same constant defined in the *Servlet Specification -* javax.servlet.context.tempdir. The portlet context must follow the same behavior and functionality defined for the servlet context for virtual hosting and reloading considerations. (See *Servlet Specification Version 3.1*):

### Correspondence between ServletContext and PortletContext methods

The following methods of the PortletContext should provide the same functionality as the methods of the ServletContext of similar name, but applied to the deployed portlet application: getAttribute, getAttributeNames, getInitParameter, getInitParameterNames, getMimeType, getRealPath, getResourceID, getResourcePaths, getResourceAsStream, log, removeAttribute, setAttribute, getClassLoader, getContextPath, getEffectiveMajorVersion, and getEffectiveMinorVersion.

## Portlet Container Runtime Options

The portlet can define additional runtime behavior in the portlet.xml on either the portlet application level or the portlet level with the container-runtime-option element. Runtime options that are defined on the application level should be applied to all portlets in the portlet application. Runtime options that are defined on the portlet level should be applied for this portlet only and override any runtime options defined on the application level with the same name.

The container runtime option javax.portlet.actionScopedRequestAttributes must be supported by the portlet container.

Support for all other container runtime options is optional. The portlet can find out which container runtime options are supported by the portlet container through the PortletContext.getContainerRuntimeOptions method. This method returns an enumeration of type String containing the keys of all container runtime options that the current portlet container supports.

Section 28.2.2 Portlet Container Runtime Options describes configuration of the portlet container runtime options.

### Runtime Option javax.portlet.escapeXml

The Java Portlet Specification V1.0 did not define XML escaping of URLs written by the tag library. Such portlets may have been coded with the assumption that the URLs were not XML escaped. In order to be able to run these portlets on a Java Portlet Specification V 2.0 or later container, the URL escaping behavior can be set through the javax.portlet.escapeXml container runtime option. The value of this setting can either be true for XML escaping URLs by default, or false for not XML escaping URLs by default.

Portlets that require URLs written to the output stream by the portlet tag library not to be XML escaped by default should therefore define the javax.portlet.escapeXml container runtime option in the portlet configuration.

If the portlet has defined the javax.portlet.escapeXml container runtime option, the portlet container should honor this setting as otherwise the portlet may not work correctly.

### Runtime Option javax.portlet.renderHeaders

There are cases in which the portlet may want to return header information, or other information that is required before getting the markup, like the portlet title or the next possible portlet modes, in the render phase. However, some portal implementations may choose a streaming implementation and thus do not buffer the output of the portlet. In order to support these implementations, the Java Portlet Specification provides the javax.portlet.renderHeaders container runtime setting and the RENDER\_PART request attribute that these streaming portal such implementations can set. Portlets that want to ensure that they run with maximum performance on all portal implementations should leverage this mechanism for:

* Setting cookies
* Setting headers
* Setting the title
* Returning new possible portlet modes

Portlets that need to write any headers in the render phase can set the additional container-runtime-option with name javax.portlet.renderHeaders and value true. The default for this setting is false. When set to true, streaming portal implementations should call the render method of the portlet twice with the RENDER\_PART attribute set in the render request (see Section 14.1.2.3 The Render Part Request Attribute).

### Runtime Option javax.portlet.servletDefaultSessionScope

By default, the session variable of included / forwarded servlets or JSPs maps to the portlet session with application scope. Some portlets may require that the session variable of included / forwarded servlets or JSPs maps instead to the portlet session with portlet scope in order to work correctly. This portlet container runtime option specifies the desired behavior. Setting it to PORTLET\_SCOPE maps the session variable to the portlet session with portlet scope, while setting it to APPLICATION\_SCOPE maps the session variable to the portlet session with application scope (the default). Example:

<container-runtime-option>

<name>javax.portlet.servletDefaultSessionScope</name>

<value>PORTLET\_SCOPE</value>

</container-runtime-option>

Portlet developers should note that not all portlet containers may be able to provide this feature as a portable Java EE solution does not currently exist. Therefore, relying on this feature may restrict the numbers of portlet containers the portlet can be executed on.

### Runtime Option javax.portlet.actionScopedRequestAttributes

The Java Portlet Specification follows a model of separating concerns in different lifecycle methods, like processAction, processEvent, render. This provides a clean separation of the action semantics from the rendering of the content. However, it may create some issues with servlet-based applications that don’t follow this strict Model-View-Controller pattern. Such applications might assume that attributes set in the action phase will be accessible during the render phase. The Java Portlet Specification provides the render parameters for such use cases, but some applications need to transport complex objects instead of strings.

One example would be a Java Server Faces (JSF) bridge portlet that expects to be executed in a single lifecycle phase for processing actions, events, and rendering from the JSF point of view and thus needs to transport attributes from action to subsequent event and render calls until the next action occurs.

For such use cases the Java Portlet Specification provides action-scoped request attributes as container runtime option with the intent to provide portlets with these request attributes until a new action occurs. This container runtime option must be supported by portlet containers.

Portlets should note that using this container runtime option will result in increased memory usage and thus may have a decreased performance as the portlet container needs to maintain and store these attributes across requests.

Portlets that want to leverage the action-scoped request attributes must to set the container runtime option javax.portlet.actionScopedRequestAttributes to true, default is false. In addition, the portlet may provide a value called numberOfCachedScopes where the following value element must be a positive number indicating the number of scopes the portlets wants to have cached by the portlet container. This value is a hint to the portlet container that the portlet container may not be able to honor because of resource constraints. The order of the values in the portlet deployment descriptor must be true, numberOfCachedScopes, <number of cached scopes>. Example:

<portlet>

...

<container-runtime-option>

<name>javax.portlet.actionScopedRequestAttributes</name>

<value>true</value>

<value>numberOfCachedScopes</value>

<value>5</value>

</container-runtime-option>

...

</portlet>

#### Action Scope ID Render Parameter

The portlet container must store the action scope ID as render parameter with the name "javax.portlet.as", defined as PortletRequest.ACTION\_SCOPE\_ID. When using the action-scoped request attribute extension the portlet must not use this render parameter name as a private or public render parameter name.

The portlet container must provide the action scope ID render parameter and its value when calling one of the portlet lifecycle methods and is responsible for setting the action scope ID at the end of the action phase.

If the portlet removes the PortletRequest.ACTION\_SCOPE\_ID render parameter through a PortletURLGenerationListener, the portlet container should create a portlet URL without this render parameter. This allows the portlet to create resource URLs that are cacheable across action scopes.

#### Lifetime of Action-scoped Request Attributes

The portlet can view attributes set on action, event, or resource requests in any of its lifecycle requests lasting until the next action occurs, or until some timeout or invalidation mechanism of the portlet container frees up the occupied memory, e.g. the user session has timed out.

A new action scope is started when

* An action request is processed: starts a new action scope with a new scope ID, all previous attributes are no longer accessible, new attributes can be stored.
* A render request with no existing scope ID is processed: starts a new scope without any scope ID, all previous attributes are no longer accessible, no new attributes can be stored.
* An event request with no existing scope ID is processed: starts a new action scope with a new scope ID, all previous attributes are no longer accessible, new attributes can be stored.
* An event request is processed after the first render for this scope has occurred: This event will likely have action semantics. All previous attributes are no longer accessible, new attributes can be stored.

The existing scope is preserved with the current scope ID and action-scoped attributes when

* A render request with an existing scope ID is processed
* An event request with an existing scope ID is processed before the first render request for this scope
* A resource request with an existing scope ID is processed

The following attributes are not stored in the action scope by the portlet container:

* all attributes starting with javax.portlet
* all Java Portlet Specification defined objects, like request, response, session, as they are only valid for the current request
* any other attributes the provided by the portlet container handling the request

The portlet may also filter out attributes that should not be stored in the action-scope at the end of the request either via removeAttribute or via a response filter.

Non-serializable objects used as attribute values might not be available across requests, for example when session replication is used. However, portlet containers should either provide the complete set of attributes to the portlet or discard the entire set of attributes in order to allow the portlet to always run in a consistent state.

#### ServeResource Calls

If a serveResource call is triggered by a resource URL with a cache level of FULL the action scope ID may not be included and thus the portlet may not have access to the action-scoped attributes.

#### Examples

Example 1:

* portlet receives a processAction call and sets attribute foo, new scope contains foo
* portlet receives a processEvent call reads foo and sets bar, scope contains foo, bar
* portlet receives a render call, scope contains foo, bar
* portlet receives a processEvent call and sets foo2, new scope contains foo2
* portlet receives a render call, scope contains foo2

Example 2:

* portlet receives a render call, empty scope
* portlet receives a processEvent call and sets foo and bar, new scope contains foo, bar
* portlet receives a serveResource call, scope contains foo, bar and sets foo’ and bar2, new scope contains foo’, bar and bar 2

#### Semantics for Portlet Containers

In order to provide a consistent user experience for end users, the portlet container should cache previous action-scoped attributes in order to allow the end user to navigate between different views with the browser forward and backward buttons. The portlet container should use the specified numberOfCachedScopes provided by the portlet or a meaningful default if the portlet has not provided this value.

In order to determine if a render has already occurred for the current action-scope it is assumed that the portlet container stores a bit invisible to the portlet in the action-scoped attributes that indicates if a render has already occurred.

# Portlet Modes

A portlet mode indicates the function a portlet is performing in the render phase. Normally, portlets perform different tasks and create different content depending on the function they are currently performing. A portlet mode advises the portlet what task it should perform and what content it should generate.When invoking a portlet, the portlet container provides the current portlet mode to the portlet. Portlets can programmatically change their portlet mode during the action phase.

The Portlet Specification defines three portlet modes - VIEW, EDIT, and HELP. The PortletMode class defines constants for these portlet modes.

The availability of the portlet modes, for a portlet, may be restricted to specific user roles by the portal. For example, anonymous users could be allowed to use the VIEW and HELP portlet modes but only authenticated users could use the EDIT portlet mode.

## VIEW Portlet Mode

The view mode represents the standard, or base, operating mode for a portlet. The view mode must be supported by all portlets, and can generally be viewed by all users who can access the portlet.

In view mode, the portlet should display its main operative content. For example, a stock ticker portlet should display the actual stock price information rather than configuration screens for adding stock symbols. However, this is just a recommendation rather than a rule.

The VIEW portlet mode of a portlet may include one or more screens that the user can navigate and interact with, or it may consist of static content that does not require any user interaction.

Portlet developers can implement the VIEW portlet mode functionality by overriding the doView method of the GenericPortlet class, by providing a method annotated with @RenderMode in a class extending GenericPortlet[[7]](#footnote-7), or providing a method annotated with @RenderMethod in a managed bean class[[8]](#footnote-8).

## EDIT Portlet Mode

Within the EDIT portlet mode, a portlet should provide content and logic that lets a user customize the behavior of the portlet. The EDIT portlet mode may include one or more screens among which users can navigate to enter their customization data.

Typically, portlets in EDIT portlet mode will set or update portlet preferences. Refer to Chapter 18 Portlet Preferences for details on portlet preferences.

Portlet developers can implement the EDIT portlet mode functionality by overriding the doEdit method of the GenericPortlet class, by providing a method annotated with @RenderMode in a class extending GenericPortlet7, or providing a method annotated with @RenderMethod in a managed bean class8.

Portlets are not required to support the EDIT portlet mode.

## HELP Portlet Mode

When in HELP portlet mode, a portlet should provide help information about the portlet. This help information could be a simple help screen explaining the entire portlet in coherent text or it could be context-sensitive help.

Portlet developers should implement the HELP portlet mode functionality by overriding the doHelp method of the GenericPortlet class, by providing a method annotated with @RenderMode in a class extending GenericPortlet7, or providing a method annotated with @RenderMethod in a managed bean class8.

Portlets are not required to support the HELP portlet mode.

## Custom Portlet Modes

Portal vendors may define custom portlet modes for vendor-specific functionality for modes that need to be managed by the portal. Portlets may define additional custom portlet modes that don’t need to be managed by the portal and correspond to the VIEW mode from a portal point of view.

The portlet must declare custom portlet modes that are not managed by the portal in the portlet configuration. Portlet modes are considered portal managed by default.

Portlets must define the custom portlet modes they intend to use in the portlet configuration, see Section 28.1.2 Custom Portlet Mode on page 185. At deployment time, the portal managed custom portlet modes defined by the portlet should be mapped to custom portlet modes supported by the portal implementation. Portlets that list custom portlet modes that are not managed by the portal may provide a localized decoration name as resource bundle entry with the key javax.portlet.app.custom-portlet-mode.<name>.decoration-name for this portlet mode. If no entry in the portlet resource bundle with such a name exists, the portlet container should use the portlet mode name as default decoration name.

If a custom portlet mode defined in the portlet configuration is not mapped to a custom portlet mode provided by the portal or otherwise supported as non-managed portlet mode, the portlet must not be invoked in that portlet mode.

Appendix D Custom Portlet Modes defines a list of portlet mode names and their suggested use. Portals implementing these predefined custom portlet modes could do an automatic mapping when custom portlet modes with those names are defined in the deployment descriptor. Therefore providing a decoration name or portal-managed element for the modes defined the appendix is not necessary.

## Defining Portlet Modes Support

The portlet must declare the supported portlet modes beyond the VIEW portlet mode for each markup type they support in the render method. As all portlets must support the VIEW portlet mode, VIEW does not have to be indicated. If the portlet configuration does not specify any portlet modes, the portlet container will assume VIEW mode support for all MIME types (MIME type \*/\*). The configuration of the portlet mode is described in Section 28.2.12 Portlet Modes and Window States on page 202.

The portlet must not be invoked in a portlet mode that has not been declared as supported for a given markup type.

The portlet container must ignore all references to custom portlet modes that are not supported by the portal implementation, or that have no mapping to portlet modes supported by the portal.

## Setting next possible Portlet Modes

Via the render response the portlet can set next possible portlet modes that make sense from the portlet point of view. If set, the portal should honor this enumeration of portlet modes and only provide the end user with choices to the provided portlet modes or a subset of these modes based on access control considerations. If the portlet does not set any next possible portlet modes the default is that all supported portlet modes that the portlet has defined are considered to be candidate new portlet modes.

To ensure that the next possible portlet modes are honored by all portal implementations, the portlet should either overwrite the GenericPortlet.getNextPossiblePortletModes method or set the next possible portlet modes in the header phase (see Section 4.7.5 Header Dispatching) via setNextPossiblePortletModes. Doing so ensures that the portal receives these suggested new modes before writing the portlet window decorations and thus is able to optimize the amount of buffering needed.

# Window States

A window state is an indicator of the amount of portal page space that will be assigned to the content generated by the portlet during rendering. When invoking a portlet, the portlet container provides the current window state to the portlet. The portlet may use the window state to decide how much information it should render. Portlets can programmatically change their window state during the action phase.

The Portlet Specification defines three window states - NORMAL, MAXIMIZED and MINIMIZED. The WindowState class defines constants for these window states.

## NORMAL Window State

The NORMAL window state indicates that a portlet may be sharing the page with other portlets. It may also indicate that the target device has limited display capabilities. Therefore, a portlet should restrict the size of its rendered output in this window state.

## MAXIMIZED Window State

The MAXIMIZED window state is an indication that a portlet may be the only portlet being rendered in the portal page, or that the portlet has more space compared to other portlets in the portal page. A portlet may generate richer content when its window state is MAXIMIZED.

## MINIMIZED Window State

When a portlet is in MINIMIZED window state, the portlet should only render minimal output or no output at all.

## Custom Window States

Portal vendors may define custom window states.

Portlets can only use window states that are defined by the portal. Portlets must define the custom window states they intend to use in the in the portlet configuration. At deployment time, the custom window states defined in the configuration should be mapped to custom window states supported by the portal implementation.

If a custom window state defined in the configuration is not mapped to a custom window state provided by the portal, portlets must not be invoked in that window state.

## Defining Window State Support

Portlets may specify the custom window states they can handle for each markup type they support in the render method through the configuration. If the portlet does not list explicitly which window states it supports, the portal / portlet container should assume that the portlet supports all pre-defined window states and all custom window states defined for this portlet application.

As all portlets must at least support the pre-defined window states NORMAL, MAXIMIZED, MINIMIZED, these window states do not have to be explicitly configured. The configuration of the window state is described in Section 28.2.12 Portlet Modes and Window States on page 202.

The portlet should not be invoked in a custom window state that has not been declared as supported for a given markup type.

The portlet container must ignore all references to custom window states that are not supported by the portal implementation, or that have no mapping to window states supported by the portal.

# Portlet Parameters

The PortletParameters and MutablePortletParameters interfaces define the API for accessing the parameters that are set for the portlet or on a portlet URL. Portlet parameters store state information that the portlet needs to render itself, generate content by serving resources, or make decisions when executing portlet actions.

Conceptually the portlet parameters correspond to query string parameters that are stored in the URL used to access the portlet, although they are not required to actually be present on the URL as visible parameters.

The Portlet Specification distinguishes between different types of portlet parameters according to their use.

* **Render Parameters** are set on the response during the Action and Event phases to govern content generation during the Render and Resource phases. They are also set on render URLs to move the portlet to a new render state when the URL is triggered.

Render parameters can always be read from the request object during request processing. However, they can only be changed during the action phase or when a render URL containing different render parameter values is activated.

Private render parameters are available exclusively to a single portlet. Public render parameters can be shared between portlets.

Example: Render URLs with differing render parameters can be used to implement tabbed navigation within a portlet.

* **Resource Parameters** provide additional information about the content to be generated when serving a resource for the governing render state. They are set on resource URLs and made available to the portlet through the resource request when the URL is triggered.

Example: Portlets may require several different pieces of content to be served for the governing render state. Resource URLs with differing resource parameters can be used to determine which content is to be served for a specific request.

* **Action Parameters** provide additional information about the action to be executed for the governing render state. They are set on action URLs and made available to the portlet through the action request when the URL is triggered, or they are provided to the portlet by the client (form parameters).

Example: Portlets may render a number of forms for the governing render state. Action URLs with differing action parameters can be used to determine which form is submitted or which action is to be executed for a specific request.

Example: Portlets may render forms that, when submitted, may cause parameters to be added to the portlet action request. During action request processing, these form parameters will be available as action parameters.

Portlet parameters can only be set and read in the appropriate execution phases. The Portlet Specifications defines interfaces for read-only and read-write access to the portlet parameters. The portlet container must implement these interfaces.

Figure 11–1 Portlet Parameters

## Portlet Parameters

Portlet parameters are name-value pairs modeled after servlet request parameters. The parameter name is represented by a String while the value is a String array. The parameter name may not be null.

The value null is a valid parameter value, both within a parameter values array and when set or returned as a single value. Setting a parameter value to null does not remove the parameter. Similarly, an empty array is a valid parameter value.

The parameters available through the PortletParameters interface must be "x-www-form-urlencoded" decoded. The parameters set through the MutablePortletParameters interface must be "x-www-form-urlencoded" encoded by the portal or portlet container as necessary.

The portlet must not be allowed to access private parameters set by other portlets.

If portlets namespace or encode URL parameters or form parameters they are also responsible for removing the namespace. The portlet container will not remove any namespacing the portlet has done on these parameters.

### Extra Request Parameters

The portlet container implementation may add extra parameters to portlet URLs to help the portlet container route and process client requests.

Extra parameters used by the portlet container must be invisible to the portlets receiving the request. It is the responsibility of the portlet container to properly namespace these extra parameters to avoid name collisions with parameters the portlets define.

Parameter names beginning with the "javax.portlet." prefix are reserved for definition by this specification and for use by portlet container implementations.

### PortletParameters Interface

The PortletParameters interface provides read-only access to the parameters. It provides the following methods.

| **Method** | **Description** |
| --- | --- |
| getValue | Returns a string value for the given parameter name. If the name designates a multivalued parameter, the first element of the values array is returned. |
| getNames | Returns a set of strings representing the parameter names. The set will be empty if no parameters are available. |
| getValues | Returns the values array for the given parameter name. Returns null if the parameter does not exist. |
| isEmpty | Returns true if the PortletParameters object is empty and false otherwise. |
| size | Returns the number of number of parameters contained in the object. |
| clone | Returns a MutablePortletParameters object containing the same parameters as this object. The underlying data structures are copied, so that changes to the returned object will not affect the original object. |

### MutablePortletParameters Interface

As shown in Figure 11–1 Portlet Parameters, the MutablePortletParameters interface extends the PortletParameters interface, adding write access to the parameters. It provides the methods below.

Changes made to a MutablePortletParameters object obtained from a portlet response or portlet URL immediately affect the underlying response or URL object. Changes made to a MutablePortletParameters object obtained by cloning a PortletParameters object only affect the object itself.

| **Method** | **Description** |
| --- | --- |
| getNames | Returns a set of strings representing the parameter names. The set will be empty if no parameters are available. Parameters cannot be added using the returned set, however, removing a parameter name from the returned set will remove the parameter from the MutablePortletParameters object. |
| setValue | Sets the parameter to the given value. If the parameter already exists, the existing parameter will be replaced. |
| setValues | Sets the parameter to the given values array. If the parameter already exists, the existing parameter will be replaced. |
| removeParameter | Removes the given parameter. |
| set | Accepts a PortletParameters object, replacing the current parameters with the parameters contained in the given object. The objects will remain independent, so that changes to one will not affect the other. |
| add | Accepts a PortletParameters object, adding the parameters contained in the given object to the current parameters. If a parameter from the input object is already present, its value will be updated with the input value. The objects will remain independent, so that changes to one will not affect the other. |
| clear | Removes all parameters. |

## Render Parameters

Render parameters govern portlet output during the render and resource phases. They can be read from the request object during the header, render, action, event or resource phases. Private render parameters are scoped to a specific portlet window. Public render parameters can be shared between portlets.

Render parameters can be set during the action and event phases through the ActionResponse or EventResponse, respectively.

Render parameters can be set on render or action URLs created during the render or resource phases. When this is done, the new render parameter values take effect when the render or action URL is activated.

Once render parameters have been set for a portlet during the action or event phase or by activating a render or action URL targeting the portlet, they remain available to the portlet until they are removed or until their values are changed. This holds true across subsequent action and event phase processing as well[[9]](#footnote-9).

If a portlet is re-rendered due to a client request targeted to another portlet in the portal page, its private render parameters must remain unchanged.

If a portlet processes an event request that is the result of a client request targeted to different portlet, its private render parameters received with the event request must be the render parameter values last set for the portlet.

If a portlet receives a render request following an action or event request, the parameters received with render request must be the render parameters as modified during the action or event request processing.

If a portlet receives a render or action request that is the result of invoking a corresponding URL targeting this portlet, the render parameters received with the request must be the parameters set on the render or action URL.

Portals often provide controls to change the portlet mode and the window state for the portlet. The URLs these controls use are generated by the portal. Client requests triggered by those URLs must be treated as render URLs in the sense that the existing render parameters are preserved.

### Public Render Parameters

Public render parameters can be shared with other portlets within the same portlet application or across portlet applications. Public render parameters can be viewed and changed by other portlets or components.

The portlet must declare public render parameters in its configuration by providing a qualified name along with an identifier. The portlet accesses the public render parameter through the configured identifier.

The qualified name uniquely identifies the public render parameter. Public render parameters defined by different portlet applications that have the same qualified name are considered to be the same public render parameter, even if the identifiers differ.

The portlet container must provide only the public render parameters specified in the portlet configuration to the portlet. The portlet container must share only those render parameters set by a portlet that are declared public in the portlet configuration.

Public render parameters can be set and removed on portlet URL and response objects using the MutableRenderState interface methods in the same manner as private render parameters. A portlet can use the RenderParameters interface isPublic method to determine if a given render parameter is a public render parameter.

The manner in which public render parameters are shared between portlet windows is portal implementation specific. For example, one portal implementation might consider only the current portal page for public render parameter sharing, while another portal implementation might share the public render parameter values across all portlet windows on all portal pages.

### RenderParameters Interface

The RenderParameters interface extends PortletParameters to provide read-only access to the render parameters. It provides the following methods.

| **Method** | **Description** |
| --- | --- |
| clone | Overrides the PortletParameters clone method to return a MutableRenderParameters object rather than a MutablePortletParameters object. |
| isPublic | Returns true if the given parameter name represents a public render parameter and false otherwise. |

### MutableRenderParameters Interface

The MutableRenderParameters interface extends RenderParameters and MutablePortletParameters to provide read-write access to the render parameters. It provides the following methods.

| **Method** | **Description** |
| --- | --- |
| clearPublic | Clears all public render parameters. |
| clearPrivate | Clears all private render parameters. |

## Action Parameters

Action parameters can be set on an action URL. Such parameters become available to the portlet as action parameters when the action URL is activated.

Action parameters may also result through an HTML form submission. Form data is handled according to the rules defined by *Servlet Specification 3.1[[10]](#footnote-10)*.

If the portlet is performing an HTML Form submission via HTTP method POST, the form data will be populated to the action parameters if the form content type is application/x-www-form-urlencoded. If the action URL used to initiate the request contains action parameters, the URL parameters will be aggregated with the form parameters. The URL parameters are presented before the POST body data.

If the POST form data are populated to the action parameters, the post form data will no longer be available for reading directly from the request object’s input stream. If the POST form data is not included in the parameter set, the post data must still be available to the portlet via the ActionRequest / ResourceRequest input stream.

A portlet container may optionally support form submission using the HTTP GET method. If the portlet is performing an HTML form submission via the HTTP GET method, the form data set is appended to the portlet URL used for the form submission and are therefore accessible as request parameters for the portlet.

As portlet URLs may be ECMA script functions that produce the required URL only on executing the URL the portlet should not simply add additional query parameters to a portlet URL on the client.

### ActionParameters Interface

The ActionParameters interface extends PortletParameters to provide read-only access to the action parameters. It provides the following methods.

| **Method** | **Description** |
| --- | --- |
| clone | Overrides the PortletParameters clone method to return a MutableActionParameters object rather than a MutablePortletParameters object. |

### MutableActionParameters Interface

The MutableActionParameters interface extends ActionParameters and MutablePortletParameters to provide read-write access to the action parameters. It provides no additional methods.

## Resource Parameters

When processing resource requests, the portlet must receive any resource parameters that were explicitly set on the ResourceURL that triggered the request.

### ResourceParameters Interface

The ResourceParameters interface extends PortletParameters to provide read-only access to the Resource parameters. It provides the following methods.

| **Method** | **Description** |
| --- | --- |
| clone | Overrides the PortletParameters clone method to return a MutableResourceParameters object rather than a MutablePortletParameters object. |

### MutableResourceParameters Interface

The MutableResourceParameters interface extends ResourceParameters and MutablePortletParameters to provide read-write access to the Resource parameters. It provides no additional methods.

# Render State

The render state consists of the portlet mode, the window state, and the public and private render parameters. The render state controls portlet content rendering. The portal URL addressing a portal page or portlet resource associated with the page generally contains the render state of each portlet on the page.

The Portlet Specification provides dedicated interfaces for accessing the render state. These interfaces are extended by the URL, request, and response interfaces that make the render state available.

The render state can be read in any portlet request phase (in the resource phase subject to the cacheability setting), but can be written only during the action or event phases. The render state can also be set on render and action URLs created during the resource or render phases.

The RenderState interface provides read-only access to the render state. The MutableRenderState interface extends RenderState to provide additional methods for read-write access.

Figure 12–1 Render State Interfaces

## The RenderState Interface

The RenderState interface provides the following methods.

| **Method** | **Description** |
| --- | --- |
| getRenderParameters | Returns the current RenderParameters object. If no render parameters are available, the object will be empty. |
| getPortletMode | Returns the current portlet mode. |
| getWindowState | Returns the current window state. |

## MutableRenderState Interface

The RenderState interface provides the following methods.

| **Method** | **Description** |
| --- | --- |
| getRenderParameters | Overrides the superclass method to return the current MutableRenderParameters object. If no render parameters are available, the object will be empty. |
| setPortletMode | Sets the portlet mode. Ifa portlet attempts to set a portlet mode that is not available or that is disallowed for the current user, this method must throw a PortletModeException. |
| setWindowState | Sets the window state. Ifa portlet attempts to set a window state that is not available or that is disallowed for the current user, this method must throw a WindowStateException. |

# Portlet URLs

The portlet may need to create URLs referencing itself when producing markup. For example, when the portlet renders a form to be submitted, or a link to another page of data within the portlet, it requires a URL that results in a request targeted to itself so that it can carry out the appropriate action. URLs that result in requests targeted to the portlet are called portlet URLs.

The Portlet Specification provides interfaces that allow the developer to manipulate and generate portlet URLs. This chapter presents those interfaces. Creation of portlet URL objects implementing these interfaces is covered in Chapter 15 PortletResponse Interfaces.

The Portlet Specification introduces three types of portlet URL for use by developers. When these URLs are activated, they initiate execution of the three portlet request phases.

* The render URL represented by the RenderURL interface initiates the render phase.
* The action URL represented by the ActionURL interface initiates the action phase.
* The resource URL represented by the ResourceURL interface initiates the resource phase.

The Portlet Specification introduces the additional interfaces BaseURL and PortletURL in order to aggregate common functionality. The developer uses these interfaces indirectly. The following figure illustrates the relationship between the portlet URL interfaces.

Figure 13–1 The Portlet URL Interfaces

The BaseURL interface provides read-only access to the render state along with other common functionality. The ResourceURL interface extends BaseURL as the resource URL is tied to a specific render state. The PortletURL interface extends BaseURL and MutableRenderState, adding the ability to change the render state. The RenderURL interface extends PortletURL to initiate rendering, while ActionURL extends PortletURL to trigger a portlet action.

Note that portlet URLs are only valid within the current request and need to be written to the output stream in order to allow re-writing the portlet URL token into a real URL.

## The Base URL

The BaseURL interface provides methods that are common for all URLs targeting the portlet.

The BaseURL interface extends the RenderState interface to provide access to the render parameters, portlet mode, and window state.

The toString method converts the URL to string form. The append method appends the URL to an Appendable object such as a StringBuilder. The write method writes output directly to the output stream. If the portlet wants to include a portlet URL in the portlet content it can use the write method to avoid the string object creation overhead associated with the toString method.

When the URL is written or appended to an output stream, a flag may be set to true in order to require XML escaping. The manner in which XML escaping is performed is implementation specific. If set to false, whether or not the URL will be XML escaped will be left to the portlet container implementation.

Portlet developers should be aware that the string representation of a portlet URL may not be a well formed URL but may be a special token at the time the portlet is generating its content. Portal servers often use a technique called URL rewriting that post-processes the content in order to resolve portlet URL tokens into real URLs. The portal may even render an ECMA script function to generate the URL when the user clicks on the link.

Properties can be used by portlets to set vendor specific information on the PortletURL object and thus use extended URL capabilities. A portlet can set properties using the setProperty and addProperty methods of the BaseURL interface.

The setProperty method sets a property with a given name and value. A previous property is replaced by the new property. Where a set of property values exist for the name, the values are cleared and replaced with the new value. The addProperty method adds a property value to the set with a given name. If there are no property values already associated with the name, a new property is created.

The setSecure method allows a portlet to indicate if the portlet URL has to be a secure URL or not (i.e. HTTPS or HTTP). If the setSecure method is not used, the portlet URL should be of the same security level of the current request. If setSecure is called with true, the transport for the request triggered with this URL must be secure (i.e. HTTPS). If set to false, the portlet indicates that it does not require a secure connection for the request triggered with such a URL.

### BaseURL Interface

The BaseURL interface provides the following methods.

| **Method** | **Version** | **Description** |
| --- | --- | --- |
| setParameter(String, String) | 2.0 | Deprecated. Behaves as described in JSR 286 Portlet Specification 2.0. |
| setParameter(String, String...) | 2.0 | Deprecated. Behaves as described in JSR 286 Portlet Specification 2.0. |
| setParameters(Map<String, String[]>) | 2.0 | Deprecated. Behaves as described in JSR 286 Portlet Specification 2.0. |
| getParameterMap() | 2.0 | Deprecated. Behaves as described in JSR 286 Portlet Specification 2.0. |
| setSecure(boolean) | 2.0 | If set to true, requests secure transmission. |
| toString() | 2.0 | Returns a string representation of the URL. |
| write(Writer) | 2.0 | Writes the portlet URL to the output stream. The URL written to the output stream is always XML escaped. |
| write(Writer, boolean) | 2.0 | Writes the portlet URL to the output stream. The URL written to the output stream is XML escaped if the boolean argument is set to true. |
| append(Appendable) | 3.0 | Appends the portlet URL to the given Appendable object. The appended URL is always XML escaped. |
| append(Appendable, boolean) | 3.0 | Appends the portlet URL to the given Appendable object. The appended URL is XML escaped if the boolean argument is set to true. |
| addProperty(String, String) | 2.0 | Adds a String property to an existing key on the URL. This method allows URL properties to have multiple values. |
| setProperty(String, String) | 2.0 | Sets a String property on the URL. This method resets all properties previously added with the same key. |

## The Resource URL

A resource URL triggers a resource request. During resource request processing, the portlet has access to the render state and has full control over the output stream. The portlet can render binary markup when serving resources.

A resource ID string can be attached to the resource URL using the setResourceID method. The resource ID is made available to the portlet during resource request processing. The portlet can query the resource ID using the getResourceID method.

The portlet can add additional resource parameters to the resource URL using the getResourceParameters method. Changes made to the MutableResourceParameters object returned by this method take effect on the resource URL immediately. Resource parameters are only made available during the resource request triggered by the resource URL.

Resource parameters can be used to differentiate between multiple resource URLs generated during the same portlet request processing phase. They are not affected by the resource URL cacheability setting.

The resource URL is tied to the render state governing the render or resource phase during which it is created. The render state on a resource URL cannot be modified. However, depending on the cacheability setting, render state information can be excluded from the URL.

The portlet can set the cacheability level for a resource URL using the setCacheability method. The portlet can query the cacheability setting using the getCacheability method.

A portal typically requires the render state of all portlets on the page in order to create portlet URLs. However, the more state attached to the URL, the less cacheable the content addressed by the URL will be. In order to allow resources to be cached to the greatest extent possible while still allowing creation of portlet URLs when necessary, the resource URL cacheability setting was introduced. In order of decreasing cacheability, the three cacheability settings follow:

* FULL – The resource URL contains no render state information.

While executing a resource request triggered by a resource URL with cacheability set to FULL, only resource URLs with the cacheability set to FULL can be created. Attempts to create resource URLs that are not of cacheability FULL, or to create action or render URLs, must result in an IllegalStateException.

URLs of the type FULL have the highest cacheability in the browser as they do not depend on any state of the portlet or page.

* PORTLET – The resource URL contains only the render state of the targeted portlet.

While executing a resource request triggered by a resource URL with cacheability set to PORTLET, only resource URLs with the cacheability set to FULL or PORTLET can be created. Attempts to create resource URLs that are not of cacheability FULL or PORTLET, or to create action or render URLs, must result in an IllegalStateException.

URLs of the type PORTLET are cacheable on the portlet level in the browser and can be served from the browser cache for as long as the state of this portlet does not change.

* PAGE – The resource URL contains the render state of all portlets on the page.

The markup returned by a resource request triggered through such a resource URL may contain any portlet URL.

Resource URLs of the type PAGE are only cacheable on the page level and can only be served from the browser cache as long as no state on the page changes.

If no cacheability option is set on the resource URL, the cacheability setting of the parent resource is used. If no parent resource is available, PAGE is the default.

### ResourceURL Interface

The ResourceURL interface provides the following fields and methods.

| **Method** | **Version** | **Description** |
| --- | --- | --- |
| PORTLET | 2.0 | String constant indicating cacheability level PORTLET |
| PAGE | 2.0 | String constant indicating cacheability level PAGE |
| SHARED | 2.0 | String constant to be used with the URL setProperty method to indicate that the resource can be shared |
| getResourceParameters() | 3.0 | Returns a MutableResourceParameters object. Parameter changes made through this object take effect on the resource URL immediately. |
| setResourceID(String) | 2.0 | Sets the resource ID |
| getResourceID() | 2.0 | Returns the currently set resource ID |
| getCacheability() | 2.0 | Returns the current cacheability setting |
| setCacheability(String) | 2.0 | Sets the cacheability |

## The Portlet URL

The PortletURL interface extends BaseURL to obtain common functionality and extends MutableRenderState to obtain methods to set the render state (window state, portlet mode, and render parameters).

When the portlet mode is set on the URL using the setPortletMode method, the change of portlet mode must be effective for the request triggered by the portlet URL. There are some exceptional circumstances, such as changes in access control privileges that could prevent the portlet mode change from happening. If the portlet mode is not explicitly set on a URL, the URL must have same the portlet mode as the current request.

When the window state is set on the URL using the setWindowState method, the change of window state should be effective for the request triggered by the portlet URL. The portlet should not assume that the request triggered by the portlet URL will be in the window state set as the portlet container could override the window state because of implementation dependencies between portlet modes and window states. If the window state is not explicitly set on a URL, the URL must have same the window state as the current request.

The portal/portlet container must ensure that all render parameters set on the portlet URL through the MutableRenderParameters interface methods become render parameters on requests resulting from render URL activation.

The PortletURL interface setBeanParameter(PortletSerializable) method sets the given @RenderStateScoped managed bean state on the URL. A @RenderStateScoped managed bean must implement the PortletSerializable interface. For a given portlet, a @RenderStateScoped bean is uniquely identified by its bean class, since the bean may not be further qualified. The values array obtained through the PortletSerializable interface serialize method is stored on the URL as a render parameter under the render parameter name associated with the bean. See Section 20.2.3 Render State Scope for further information.

### PortletURL Interface

The PortletURL interface provides the following methods.

| **Method** | **Version** | **Description** |
| --- | --- | --- |
| setBeanParameter | 3.0 | Sets the given @RenderStateScoped managed bean state on the URL. |
| removePublicRenderParameter | 2.0 | Deprecated. Behaves as described in JSR 286 Portlet Specification 2.0. |

## The Render URL

A render URL triggers portlet render phase execution. The RenderURL interface extends PortletURL to allow read-write access to the render state and to obtain the base portlet URL functionality.

Render URLs should only be used for idempotent tasks, i.e. tasks that do not change server state from the portlet perspective in order to allow the content to be cacheable from a network infrastructure point of view.

Error conditions, cache expirations, and changes of external data may affect the content generated by a portlet as result of a request triggered by a render URL.

Render URLs should be accessed using the HTTP GET method as the results of their invocation should not change any state on the server. As a consequence, it may be possible to bookmark render URLs. Render URLs should not be used for form submission.

The portlet can set a fragment identifier on a render URL. The fragment identifier consists of additional information appended to the URL after a '#' character. A URL can have only a single fragment identifier. The fragment identifier must be formed according to rfc3986 "Uniform Resource Identifier (URI): Generic Syntax".

The fragment identifier is often used to address a named anchor such as <a name="#fragmentIdentifier">, but it can also be used for other purposes such as to provide additional information for JavaScript routines on the client.

If the fragment identifier is not specified by the portlet, the portal implementation can optionally append a fragment identifier to the URL.

The setFragmentIdentifier method allows a fragment identifier to be appended to a render URL. The fragment identifier appended with this method will not be namespaced. The portlet is responsible for performing any required namespacing. However, the fragment identifier string will be escaped as necessary.

Setting the fragment identifier to null will remove a fragment identifier that was previously set using this method. Setting the fragment identifier to the empty string will create an empty fragment identifier.

The getFragmentIdentifier method retrieves a fragment identifier previously set through setFragmentIdentifier.

### RenderURL Interface

The RenderURL interface provides the following methods.

| **Method** | **Version** | **Description** |
| --- | --- | --- |
| setFragmentIdentifier | 3.0 | Sets the fragment identifier |
| getFragmentIdentifier | 3.0 | Queries the fragment identifier |

## The Action URL

An action URL triggers portlet action phase execution. The ActionURL interface extends PortletURL to allow read-write access to the render state and to obtain the base portlet URL functionality.

Action URLs are intended to be used for non-idempotent tasks, i.e. tasks that change server state. Action URLs are often used for form submission. Action URLs are generally accessed using the HTTP POST method, and by virtue of that, the content returned by a request initiated through an action URL is generally not cacheable.

The portlet can add additional action parameters to the action URL using the getActionParameters method. Changes made to the MutableActionParameters object returned by this method take effect on the resource URL immediately. Action parameters are only made available during the action request triggered by the action URL.

Action parameters can be used to differentiate between multiple action URLs generated during the same portlet request processing phase.

### ActionURL Interface

The ActionURL interface provides the following methods.

| **Method** | **Version** | **Description** |
| --- | --- | --- |
| getActionParameters() | 3.0 | Returns a MutableActionParameters object. Parameter changes made through this object take effect on the action URL immediately. |

## The Portlet URL Generation Listener

Portlets can register portlet URL generation listeners to filter URLs before they are generated using the BaseURL toString, write, or append methods. The portlet URL generation listener is also called for a render URL that is added to a redirect URL through the ActionResponse.sendRedirect(location, renderUrlParamName) method.

For example the portlet could use a portlet URL generation listener to set the cacheability of resource URLs in one central piece of code.

A portlet URL generation listener can be configured either through the deployment descriptor or by using the @PortletURLGenerationListener annotation.

### PortletURLGenerationListener Interface

The PortletURLGenerationListener interface provides callbacks for each portlet URL type. If the portlet application has specified one or more PortletURLGenerationListener classes in the portlet deployment descriptor, the portlet container must call:

* The filterActionURL method for all action URLs before executing the write, append, or toString method of these action URLs.
* The filterRenderURL method for all render URLs before executing the write, append, or toString method of these render URLs.
* The filterResourceURL method for all resource URLs before executing the write, append, or toString method of these resource URLs.

The portlet container must provide the PortletURL or ResourceURL to generate to the filter methods and execute the write or toString method on the updated PortletURL or ResourceURL that is the outcome of the filter method call.

### Configuration through the Deployment Descriptor

In order to receive a callback from the portlet container before a portlet URL is generated the listener class must implement the PortletURLGenerationListener interface and register it in the deployment descriptor.

Portlet applications must register Portlet URL listeners in the portlet deployment descriptor under the application section with the listener element and provide the class name that implements the PortletURLGenerationListener as value in the listener-class element.

If more than one listener is registered, the portlet container must chain the listeners in the order of appearance in the deployment descriptor.

### Configuration through Annotation

The @PortletListener annotation designates a portlet URL generation listener class. The portlet URL generation listener class must implement the PortletURLGenerationListener interface.

If more than one class is annotated, the order of execution is determined by the ordinal element within the @PortletListener annotation. Methods in annotated classes with a lower ordinal number are executed before methods in classes with a higher ordinal number. If two annotated classes have the same ordinal number, the methods of both will be executed, but the execution order will be undetermined.

Portlets should generally configure portlet URL generation listeners either exclusively through the deployment descriptor or exclusively through annotations. If portlet URL generation listeners are configured through both means, the annotated portlet URL generation listener methods will be executed before the listeners configured through the deployment descriptor.

The annotated listener can apply to multiple portlets within the portlet application. The names of the portlets for which the listener applies must be specified in the @PortletListener portletNames element. A wildcard character '\*' can be specified in the first portletName array element to indicate that the listener is to apply to all portlets in the portlet application. If specified, the wildcard character must appear alone in the first array element.

The @PortletListener annotation contains the following elements.

|  |  |
| --- | --- |
| **Element** | **Description** |
| portletNames | The portlet names for which the listener applies. |
| ordinal | The ordinal number for this annotated method. |
| description | The locale-specific description for tool use. |
| displayName | The locale-specific display name for tool use. |
| listenerName | The unique name for the listener. |

The listener name is not required. If a listener name is provided, the listener configuration may be addressed through the listener name in the portlet deployment descriptor to modify or remove the listener. The listener can be modified in the portlet deployment descriptor by specifying a different PortletURLGenerationListener class. If no listener class is provided in the portlet deployment descriptor for a given listener name, the listener is removed.

# Portlet Request Interfaces

Portlet request phases were introduced in Section 3.7 Portlet Request Processing and explained in detail in Chapter 4 Portlet Lifecycle Interfaces.

The request objects implement corresponding portlet request interfaces to encapsulate all information about the client request, parameters, request content data, portlet mode, and window state necessary for processing the request. The request object is passed to the portlet processAction, processEvent, serveResource, renderHeaders, and render methods. It is also explicitly or implicitly available during extended annotation-based method dispatching.

The Portlet Specification introduces five portlet request interfaces for direct use by developers.

* The HeaderRequest provides data for header request processing.
* The RenderRequest provides data for render request processing.
* The ActionRequest provides data for action request processing.
* The EventRequest provides data for event request processing.
* The ResourceRequest provides data for resource request processing.

The Portlet Specification introduces the additional interfaces PortletRequest and ClientDataRequest in order to aggregate common functionality. The developer uses these interfaces indirectly. The following figure illustrates the relationship between the portlet request interfaces.

Figure 14–1 Portlet Request Interfaces

## PortletRequest Interface

The PortletRequest interface at the root of the portlet request interface hierarchy provides read-only access to the render state along with other common functionality.

### Parameter Handling

The current render parameters must be made available to all request processing methods as described in Section 11.2 Render Parameters on page 63. Once set during action or event request processing, or through activation of a render URL, the render parameters remain set until they are modified or removed.

Availability of the render parameters during resource request processing depends on the resource URL cacheability setting as described in Section 13.2 The Resource URL on page 71.

The action parameters are available only during action request processing. Resource parameters are available only during resource request processing.

#### Version 2.0 Parameter Handling

*If a portlet with a version 2.0 or earlier deployment descriptor is deployed, the portlet container must provide behavior described in this section in order to assure backward compatibility. For version 3.0 or later portlets, the behavior described in this section is to be disregarded.*

The portlet container must not propagate parameters received in an action or event request to subsequent render requests of the portlet. The portlet container must not propagate parameters received in an action request to subsequent event requests of the portlet.

During action or event request processing, no parameters are set on the response by default. The portlet must use the setRenderParameter or setRenderParameters methods of the StateAwareResponse object to set the render parameters appropriately.

During event request processing, the render parameters last set for the portlet must be made available through the EventRequest object.

### Request Attributes

Request attributes are objects associated with a portlet during a single portlet request. Requests attributes are removed at the end of a request processing phase. Request attributes may be set by the portlet or the portlet container to express information that otherwise could not be expressed via the API. Request attributes can be used to share information with a servlet or JSP included through the PortletRequestDispatcher.

Attributes are set, obtained, and removed using the following PortletRequest methods:

* getAttribute
* getAttributeNames
* setAttribute
* removeAttribute

Only one attribute value may be associated with an attribute name.

Attribute names beginning with the "javax.portlet." prefix are reserved for definition by this specification. It is suggested that all attributes placed into the attribute set be named in accordance with the reverse domain name convention suggested by the Java Programming Language Specification 1 for package naming.

#### The User Information Request Attribute

Portlets can obtain an unmodifiable Map object containing the user attributes of the user associated with the current request from the request attributes. The Map object can be retrieved using the USER\_INFO constant defined in the PortletRequest interface.

If the request is done in the context of an unauthenticated user, the getAttribute method with the USER\_INFO constant must return null. If the user is authenticated and there are no user attributes available, the Map must be an empty Map.

The Map object must contain a String name-value pair for each available user attribute. The Map object should only contain the user attributes that have been configured for the portlet. See Section 28.1.4 User Attribute on page 186.

An example of a portlet retrieving user attributes would be:

...

Map userInfo = (Map) request.getAttribute(PortletRequest.USER\_INFO);

String givenName = (userInfo!=null)

? (String) userInfo.get(PortletRequest.P3PUserInfos.USER\_NAME\_GIVEN) : "";

String lastName = (userInfo!=null) ?

(String) userInfo.get(PortletRequest.P3PUserInfos.USER\_NAME\_FAMILY) : "";

...

#### The CC/PP Request Attribute

The portlet can access a Composite Capability/Preference Profile[[11]](#footnote-11) javax.ccpp.Profile object[[12]](#footnote-12) using the request attribute PortletRequest.CCPP\_PROFILE. The PortletRequest.CCPP\_PROFILE request attribute must return a javax.ccpp.Profile based on the current portlet request. It may contain additional CC/PP information set by the portlet container.

#### The Render Part Request Attribute

*If a portlet with a version 2.0 or earlier deployment descriptor is deployed, the portlet container must provide behavior described in this section in order to assure backward compatibility. For version 3.0 or later portlets, the behavior described in this section is to be disregarded.*

If the RENDER\_PART portlet request attribute is set, it indicates that the render request will be processed twice in order to allow portlets running on streaming portals to set header information and document HEAD section data.

1. During the first render request execution, the RENDER\_PART request attribute is set to the value RENDER\_HEADERS. This execution is performed before the underlying servlet response is committed. During this request execution, the portlet should only set the header-related data, cookies, markup for the document HEAD section, the next possible portlet modes, and the portlet title. The portlet can set cache information for this response that may differ from the one set on the RENDER\_MARKUP response.
2. During the second render request execution, the RENDER\_PART request attribute is set to the value RENDER\_MARKUP. The portlet should produce its markup during this execution.

Non-streaming portals will not set this attribute and thus the portlet should set all necessary headers and produce markup in a single render request.

If the javax.portlet.renderHeaders container runtime setting is set to false, the RenderRequest.RENDER\_PART request attribute should not be set.

The GenericPortlet class provides support for the RENDER\_PART request attribute. See Section 4.7.6 Render Dispatching.

Portlets making use of the extended annotation-based dispatching mechanism (see Section 4.8 Extended Annotation-Based Dispatching) are not affected by the RENDER\_PART attribute.

#### The Lifecycle Phase Request Attribute

The LIFECYCLE\_PHASE request attribute of the PortletRequest interface allows a portlet to determine the current lifecycle phase of this request. This attribute value must be ACTION\_PHASE if the current request is of type ActionRequest, EVENT\_PHASE if the current request is of type EventRequest, HEADER\_PHASE if the current request is of type HeaderRequest, RENDER\_PHASE if the current request is of type RenderRequest, and RESOURCE\_ PHASE if the current request is of type ResourceRequest.

The main intent of this attribute is to allow frameworks implemented on top of the Java Portlet Specification to perform the correct type casts from the PortletRequest/PortletResponse to a specific request/response pair, like ActionRequest/ActionResponse.

#### Action-scoped Request Attributes

The Java Portlet Specification follows a model of separating concerns into different request processing phases, such as the action phase, the event phase, and the render phase. This provides a clean separation of the action semantics from the rendering of the content, however, it may create some issues with servlet-based applications that don’t follow this strict Model-View-Controller pattern. Such applications in some cases assume that attributes that they set in the action phase will be accessible again when starting the rendering. The Java Portlet Specification provides the render parameters for such use cases, but some applications need to transport complex objects instead of strings.

For such use cases the Java Portlet Specification provides the action-scoped request attributes as container runtime option with the intent to provide portlets with these request attributes until a new action occurs.

Section 8.4.4 Runtime Option javax.portlet.actionScopedRequestAttributes on page 51 describes this option in more detail.

### Request Properties

A portlet can access portal/portlet container specific properties and, if available, the headers of the HTTP client request through the following PortletRequest methods:

* getProperty
* getProperties
* getPropertyNames

A property with a given name can have multiple values. If the property has multiple values, the getProperty method returns the first property value. The getProperties method allows access to all the property values associated with a particular property name by returning an Enumeration of String objects.

Depending on the underlying web-server/servlet-container and the portal/portlet container implementation, client request HTTP headers may not be always available. Portlets should not rely on the presence of headers to function properly.

The portlet interfaces provide specific methods to access some information that can be transmitted as HTTP headers fields. Portlets should use the portlet request methods for retrieving such values as the portal/portlet container implementation may use other means to determine that information. This holds true for the following information in particular.

|  |  |  |
| --- | --- | --- |
| **HTTP Header** | **Request Interface** | **Method** |
| content-length | ClientDataRequest | getContentLength |
| content-type | PortletRequest | getContentType |
| accept-language | PortletRequest | getLocale, getLocales |

### Cookies

The portlet can access cookies provided by the current request with the getCookies method. The returned cookie array provides the portlet with all cookie properties.

### Request Context Path

The PortletRequest interface provides the context path through the getContextPath method. The context path is the path prefix associated with the deployed portlet application. If the portlet application is rooted at the base of the web server URL namespace (also known as "default" context), this path must be an empty string. Otherwise, it must be the path the portlet application is rooted to, the path must start with a '/' and it must not end with a '/' character.

The getPortletContext method returns the portlet context for the portlet application.

### Security Attributes

The PortletRequest interface offers a set of methods that provide security information about the user and the connection between the user and the portal. These methods are:

* getAuthType
* getRemoteUser
* getUserPrincipal
* isUserInRole
* isSecure

The getAuthType method indicates the authentication scheme being used between the user and the portal. It may return one of the defined constants (BASIC\_AUTH, DIGEST\_AUTH, CERT\_AUTH, and FORM\_AUTH) or another String value that represents a vendor-provided authentication type. If the user is not authenticated, the getAuthType method must return null.

The getRemoteUser method returns the login name of the user making this request.

The getUserPrincipal method returns a java.security.Principal object containing the name of the authenticated user.

The isUserInRole method indicates if an authenticated user is included in the specified logical role.

The isSecure method indicates if the request has been transmitted over a secure protocol such as HTTPS.

### Response Content Types

The PortletRequest interface getResponseContentType method returns a string representing the default content type the portlet container assumes for the output. The getResponseContentTypes method returns all of the content types supported by the portlet container for the request. The returned Enumeration of strings should contain the content types the portlet container supports in order of preference. The first element of the enumeration must be the same content type returned by the getResponseContentType method.

The values returned by the getResponseContentType and getResponseContentTypes methods must be the same for action, event, and render request processing occurring within the same client request.

The portlet may declare supported content types in the portlet configuration using wildcards.

If a portlet defines support for all content types using a wildcard and the portlet container supports all content types, the getResponseContentType may return the wildcard or the content type preferred by the portlet container.

If the getResponseContentType or getResponseContentTypes methods are exposed through the ActionRequest, EventRequest, headerRequest or RenderRequest interfaces, the following additional restrictions apply:

* The content type must include only the MIME type, not the character set. The character set of the response can be retrieved using the RenderResponse interface getCharacterEncoding method.
* The getResponseContentTypes method must return only the content types supported by the portlet container for the current portlet mode.

If the getResponseContentType or getResponseContentTypes methods are exposed through the ResourceRequest interface, the return values should be based on the HTTP Accept header provided by the client.

### Internationalization

The PortletRequest interface getLocale method returns the preferred locale for the response as determined by the portlet container. The portlet container may use the Accept-Language header along with other information to make the determination. The getLocales method returns an Enumeration of Locale objects indicating, in decreasing order of preference, the locales in which the portlet container will accept content for this request.

### Portlet Mode

The getPortletMode method provides the current portlet mode. A portlet may be restricted to work with a subset of the portlet modes supported by the portlet container. A portlet can use the PortletRequest interface isPortletModeAllowed method to determine if the portlet is allowed to use a portlet mode. A portlet mode is not allowed if the portlet mode is not defined in the portlet configuration or if the available portlet modes for the portlet or for the user have been constrained further by the portal. Note that the VIEW mode is always allowed, even if not explicitly configured by the portlet.

### Window State

The getWindowState method provides the current window state. A portlet may be restricted to work with a subset of the window states supported by the portlet container. A portlet can use the PortletRequest interface isWindowStateAllowed method to determine if the portlet is allowed to use a window state.

### Access to the Portlet Window ID

The PortletRequest interface getWindowID method provides the current portlet window ID. The portlet window ID must be unique for the portlet window and constant for the lifetime of the portlet window. The portlet container must use the window ID this method returns for scoping the portlet-scope session attributes. The portlet window ID must not contain a ‘?’ character in order to comply with the portlet scope session ID requirements (see Section 19.4 Binding Attributes to a Portlet Session).

### User Agent

The getUserAgent() method makes the user agent string available to the portlet.

### Deprecated Methods

The following PortletRequest interface methods have been deprecated with Portlet Specification Version 3.0.

* getParameter
* getParameterValues
* getParameterNames
* getParameterMap
* getPrivateParameterMap
* getPublicParameterMap

These methods have been replaced by the parameter handling methods provided by the RenderParameters, ActionParameters, and ResourceParameters objects. See Chapter 11 Portlet Parameters on page 60.

## ClientDataRequest Interface

The ClientDataRequest interface extends the PortletRequest interface to provide additional information about the underlying HTTP request, such as access to the input stream.

### Retrieving Uploaded Data

The ClientDataRequest interface getPortletInputStream method returns the input stream. The portlet can use the input stream to read data when the client request contains HTTP POST data of a type other than application/x-www-form-urlencoded. For example, this might be the case when a file is uploaded to the portlet as part of a user interaction.

As a convenience to the portlet developer, the ClientDataRequest interface also provides a getReader method that retrieves the HTTP POST data as character data according to the character encoding defined in the request.

Only one of the two methods, getPortletInputStream or getReader, can be used during a single request phase. If the portlet obtains an input stream, a subsequent call to getReader must throw an IllegalStateException. Similarly, if the reader is obtained, a call to the getPortletInputStream must throw an IllegalStateException.

To help manage the input stream, the ClientDataRequest interface also provides the following methods:

* getContentType
* getCharacterEncoding
* setCharacterEncoding
* getContentLength
* getContentLengthLong

The setCharacterEncoding method sets the character set for the Reader returned by the getReader method and must be called prior to reading input using getReader or getPortletInputStream.

If the HTTP POST body data is of type application/x-www-form-urlencoded, it will have been already processed by the portlet container and will be available as request parameters. The getPortletInputStream and getReader methods must throw an IllegalStateException in this case.

The portlet API provides support for multipart requests through the getPart and getParts methods. The getPart method returns a javax.servlet.http.Part object corresponding to the specified name. The getParts method returns a collection of all javax.servlet.http.Part objects available for the request.

## ActionRequest Interface

The ActionRequest interface extends the ClientDataRequest interface for use during action request processing.

The ActionRequest interface getActionParameters method returns the action parameters for the request, see Section 11.3 Action Parameters. It also defines the ACTION\_NAME constant that can be used together with the @ProcessAction and @ActionMethod annotations.

The getRenderParameters method must return the render parameters that governed the request in which the action URL was created along with any changes made to the render parameters on the action URL.

## ResourceRequest Interface

The ResourceRequest interface extends the ClientDataRequest interface for use during resource request processing.

The ResourceRequest interface getResourceParameters method returns the resource parameters for the request, see Section 11.4 Resource Parameters.

The ResourceRequest interface getResponseContentType and getResponseContentTypes methods return the preferred content type and the set of acceptable content types for the response.

If the cacheability level of that resource URL (see Section 13.2 The Resource URL) is set to PORTLET or PAGE, the method must return the render parameters that governed the request in which the resource URL was created. Otherwise, the getRenderParameters method must return an empty RenderParameters object. The getCacheability method returns the cacheability setting for this request.

The ResourceRequest interface defines the ETAG constant and the getETag method for validation based caching as well as the getResourceID method that returns the resource ID set on the resource URL.

The ResourceRequest interface provide methods for asynchronous support. These are described in Chapter Asynchronous Support on page 124.

## EventRequest Interface

The EventRequest interface extends the PortletRequest interface and is used during event request processing.

The EventRequest interface getEvent method returns the Event object that triggered event processing. The Event object provides the event QName via getQName, the event local name through the getName method, and the event payload through the getValue method.

The EventRequest interface getMethod method returns the name of the HTTP method with which the original action request was made, for example POST, or PUT.

The getRenderParameters method must return the render parameters that governed at the end of the last action or render request for the portlet.

## RenderRequest Interface

The RenderRequest interface extends the PortletRequest interface and is used during render request processing.

The getRenderParameters method must return the render parameters that governed at the end of the last action, event, or render request for the portlet. If the render parameters were last set as the result of an action or event request targeting the portlet, these render parameters must be returned. If the render parameters were last set as the result of a render URL targeting the portlet, the render parameters set on the render URL must be returned.

The RenderRequest interface defines the ETAG constant and the getETag method for validation based caching.

## HeaderRequest Interface

The HeaderRequest interface extends the RenderRequest interface and is used during header request processing. It is a marker interface that provides no new methods.

## Lifetime of the Request Objects

Each request object is valid only within the scope of a single request phase execution. Containers commonly recycle request objects in order to avoid the performance overhead of request object creation. The developer must be aware that maintaining references to request objects outside the request phase scope may lead to non-deterministic behavior.

# PortletResponse Interfaces

Portlet request phases were introduced in Section 3.7 Portlet Request Processing, and explained in detail in Chapter 4 Portlet Lifecycle Interfaces.

The response objects implement corresponding portlet response interfaces to encapsulate all information to be returned from the portlet to the portlet container during request processing. This might be for example a redirection, a portlet mode change, setting the title, rendering content, etc. The portlet container will use this information to construct the response to be returned to the client. A response object is passed to the portlet processAction, processEvent, serveResource, renderHeaders, and render methods. It is also explicitly or implicitly available during extended annotation-based method dispatching.

The Portlet Specification introduces five portlet response interfaces for direct use by developers.

* The HeaderResponse allows the portlet to set properties such as HTTP headers and cookies and to generate markup to be included in the overall portal page document HEAD section.
* The RenderResponse allows the portlet to generate markup to be included in the overall portal page document BODY section in a manner that is compatible with other portal markup.
* The ActionResponse allows the portlet to set up the render state for a subsequent render request.
* The EventResponse allows the portlet to set up the render state for a subsequent render request.
* The ResourceResponse allows the portlet to write data to be passed back to the client so as to allow the resource request complete control over the response.

Figure 15–1 Portlet Response Interfaces

The figure above illustrates the relationship between the portlet response interfaces. The portlet response interfaces used directly by developers are shown in green. The Portlet Specification introduces the additional interfaces PortletResponse, MimeResponse and StateAwareResponse in order to aggregate common functionality. The developer uses these interfaces indirectly.

The PortletResponse interface provides functionality needed by all portlet responses. The MimeResponse interface adds additional capability needed by responses that render output. The StateAwareResponse interface extends MutableRenderState, allowing the render state to be modified.

## PortletResponse Interface

The PortletResponse interface defines the common functionality for all other portlet response interfaces.

### Response Properties

Portlets can use properties to send vendor specific information to the portlet container.

Response properties can be viewed as header values set for the portal application. If these header values are intended to be transmitted to the client as response headers, they should be set during the header phase or before the response is committed during the resource phase.

A portlet can set HTTP headers for the response using the setProperty or addProperty methods. Headers set on the response are not guaranteed to be transported to the client as the portal application may restrict headers due to security reasons or because they conflict with headers set by other portlets on the page.

A portlet can set properties using the following PortletResponse interface methods:

* setProperty
* addProperty

The setProperty method sets a property with a given name and value. A previous property value is replaced by the new value. If a set of property values exist for the name, the values are cleared and replaced with the new value. The addProperty method adds a property value to the set with a given name. If there are no property values already associated with the name, a new set is created.

The getProperty, getPropertyValues, and getPropertyNames methods can be used to access the properties set for this portlet.

### Setting Cookies

A portlet can set HTTP cookies on the response via the addProperty method that takes a javax.servlet.http.Cookie as parameter. The portal application is not required to transfer the cookie to the client, so the portlet should not assume that it can access the cookie on the client.

Cookies set in the response of one lifecycle call should be available to the portlet in the subsequent lifecycle calls. For example, setting a cookie during the action phase should make the cookie available during subsequent event or render phases.

For requests triggered via portlet URLs, the portlet should receive the cookie set during the last response. Cookies can be retrieved using the PortletRequest interface getCookies method.

Cookies are properties and all above restrictions about properties also apply for cookies, i.e. to be successfully transmitted back to the client, cookies must be set before the response is committed during header request or render request processing. The portlet container must ignore cookies set during the render or resource phase after the response buffer has been flushed to the portal application (see discussion on buffering, below).

### Setting HEAD Section markup

*The methods described in this section are provided for compatibility with previous versions of the specification. For the corresponding version 3.0 functionality, see Section 15.8.1* Page Resources and Dependencies *on page 97.*

The PortletResponse interface addProperty method provides a method signature that allows the portlet to add an XML DOM element property to the response. This method is intended for use during the header phase.

The portlet may create a DOM element using the PortletResponse interface createElement method.

The portlet uses the addProperty(String key, Element element) method to indicate that the DOM element provided as an argument should be added to the overall portal response document HEAD section. The key argument must have the value "javax.portlet.markup.head.element", which is provided by the MimeResponse.MARKUP\_HEAD\_ELEMENT string value.

Support for this property is optional. The portlet can verify if the portlet container provides this functionality by querying the MARKUP\_HEAD\_ELEMENT\_SUPPORT property on the PortalContext. Even if this property is supported, the portal application may choose to disregard certain DOM elements to avoid conflicts with other markup or for other implementation-specific reasons.

If a DOM element with the key specified in addProperty already exists, the element will be stored in addition to the existing element under the same key. If the element is null the key must be removed from the response.

As with properties and cookies, if these header values are intended to be transmitted to the client, they must be set before the response is committed. Header values set during a render or resource request after the response is committed will be ignored by the portlet container.

### Encoding URLs

Portlets may generate content with URLs referring to other resources within the portlet application, such as servlets, JSPs, images and other static files. Some portal/portlet container implementations may require those URLs to contain implementation-specific data. Because of this, portlets should use the encodeURL method to encode portlet URLs. The encodeURL method may include the session ID and other portlet container-specific information into the URL. If encoding is not needed, the method may return the URL unchanged.

Resources that are addressed by neither a URL encoded with encodeURL nor by a ResourceURL are not guaranteed to be accessible.

The portlet developer should be aware that the URL returned by encodeURL might not be a well formed URL but rather a special token at the time the portlet is generating its content. Thus portlets should not add additional parameters to the resulting URL or expect to be able to parse it. As a result, the outcome of the encodeURL call may be different than calling encodeURL in a servlet environment.

### Namespacing

Portlet markup may include elements, such as JavaScript function and variable names, that must be unique within the overall portal page.

The getNamespace method must return a unique string that the portlet can use to assure markup element uniqueness. The returned value must be unique for the portlet window, constant for the lifetime of the portlet window, and must be a valid JavaScript identifier[[13]](#footnote-13).

As an example, the portlet can use the getNamespace method return value to prefix JavaScript variable names.

## StateAwareResponse Interface

The StateAwareResponse interface extends the PortletResponse interface and the MutableRenderState interface to provide methods that allow the portlet to modify the render state and to fire events.

### Render Parameters

The StateAwareResponse interface getRenderParameters method must return a MutableRenderParameters object representing the render parameters. The render parameters to be returned are determined as follows:

* During action request processing, getRenderParameters must return the render parameters set on the initiating action URL.
* During event request processing, getRenderParameters must return the render parameters that were set at the end of the last preceding action, event, or render request.

The MutableRenderParameters object obtained from getRenderParameters allows render parameters to be set, cleared, and modified. Changes made through this object must be immediately applied to the response object.

For example, a render parameter on a StateAwareResponse object can be set as follows:

Response.getRenderParameters().setValue("name", "value");

The portlet container must make render parameters set on the StateAwareResponse object available to all subsequent requests until they are changed through action or event request processing, or until a render request is initiated through a render URL containing new parameter values.

### Portlet Modes and Window State Changes

The setPortletMode method allows a portlet to change its current portlet mode. Ifa portlet attempts to set a portlet mode that it is not allowed to switch to, a PortletModeException must be thrown. The portlet container may override the new portlet mode. If the portlet container accepts the new portlet mode, it must make the new portlet mode effective for subsequent requests.

The setWindowState method allows a portlet to change its current window state. Ifa portlet attempts to set a window state that it is not allowed to switch to, a WindowStateException must be thrown. The portlet container may override the new window state. If the portlet container accepts the new window state, it must make the new window state effective for subsequent requests.

If the portlet does not set a new portlet or window state through the StateAwareResponse interface, the portlet container must preserve the current portlet mode and window state.

### Publishing Events

The portlet can publish events via the setEvent method. It can also call setEvent multiple times during a single action or event phase in order to publish multiple events (see Chapter 17 Coordination between Portlets).

### Deprecated Methods

The following StateAwareResponse interface methods have been deprecated with Portlet Specification Version 3.0.

* setRenderParameter(String, String)
* setRenderParameter(String, String…)
* setRenderParameters(Map<String, String[]>)
* removePublicRenderParameter(String)
* getRenderParameterMap()

## ActionResponse Interface

The ActionResponse interface extends the StateAwareResponse interface and is used during action request processing. This interface provides additional methods for creating a URL for redirection and for redirecting the client to a URL.

### Redirections

The createRedirectURL(MimeResponse.Copy) method returns a render URL containing render parameters according to the MimeResponse.Copy argument. The portlet may modify the returned render URL. The returned render URL is intended to be used in the sendRedirect(String location) method to allow the portlet to force a redirect to the same page with modified render state.

The portlet may not both update the render state for subsequent rendering and perform a redirection, since the redirection interrupts the normal action phase followed by render phase portlet processing sequence.

If the portlet calls the createRedirectURL(MimeResponse.Copy) method after the render state has been modified, the portlet container must throw an IllegalStateException, since the sendRedirect(String location) method may only be used if the render state has not yet been modified.

The ActionResponse interface provide two methods for redirecting to a different URL.

* The sendRedirect(String location) method instructs the portal/portlet container to set the appropriate headers and content body to redirect the user to a different URL. The portlet must specify an absolute URL or a full path URL. Ifthe portlet supplies a relative path URL, the portlet container must throw an IllegalArgumentException.

If the portlet calls the sendRedirect(String location) method after the render state has been modified, the portlet container must throw an IllegalStateException and must not execute the redirection.

* The sendRedirect(String location, String renderUrlParamName) method instructs the portal/portlet container to set the appropriate headers and content body to redirect the user to a different URL. The portlet must specify an absolute URL or a full path URL. Ifthe portlet supplies a relative path URL, the portlet container must throw an IllegalArgumentException.

The portlet container must attach a render URL with the render state currently set on the ActionResponse. The attached URL must be available as query parameter value under the key provided with the renderUrlParamName argument.

The portlet can modify the render state on the ActionResponse before invoking this method. The modifications must be encoded in the attached render URL, but must not affect subsequent request processing.

If the portlet calls one of the sendRedirect methods, the portlet container must discard any events set during action phase execution.

## EventResponse Interface

The EventResponse interface extends the StateAwareResponse interface and provides no additional methods.

### Deprecated Methods

The EventResponse interface setRenderParameters(EventRequest request) method has been deprecated with Portlet Specification Version 3.0. It is no longer needed since the render state including the render parameters will automatically be available during subsequent requests.

## MimeResponse Interface

The MimeResponse interface extends the PortletResponse interface to add methods that allow portlets to create MIME-based content that is returned to the portal application.

### Content Type

A portlet can set the content type of the response using the MimeResponse interface setContentType method.

When called during render request processing, the setContentType method must throw an IllegalArgumentException if the given content type does not match (including wildcard matching) any of the content types returned by the getResponseContentTypes method of the PortletRequest object. During render request processing, the portlet container should ignore any character encoding specified as part of the content type and treat the content type as if the character encoding was not specified.

The setContentType method must be called before the getWriter or getPortletOutputStream methods. Otherwise, the method will have no effect.

The getContentType method must return the content type previously set through the setContentType method. If no content type has been set, the getContentType method must return null.

If the portlet does not specify a content type before using the getWriter or getPortletOutputStream methods, the portlet container must use the content type that would be returned by the PortletRequest.getResponseContentType method and must resolve content type wildcards on a best effort basis.

### Output Stream and Writer Objects

A portlet may generate its content by writing to either the OutputStream or to the PrintWriter object obtained through the MimeResponse interface getPortletOutputStream or getWriter methods, respectively.

Only one of the two methods, getPortletOutputStream or getWriter, can be used during a single request. If the portlet obtains an output stream, a subsequent call to getWriter must throw an IllegalStateException. Similarly, if the portlet obtains a writer, a call to the getPortletOutputStream must throw an IllegalStateException.

However, to support multiple portlet method invocations during a single request phase when using extended annotation-based dispatching, the portlet container must allow the portlet to call either getPortletOutputStream or getWriter multiple times. The portlet container should return the same object each time the method is called within the same portlet phase.

The termination of the portlet render or resource phase indicates that the portlet has satisfied the request and that the portlet container can provide the contents of the output buffer to the portal application for aggregation into the portal page.

The raw OutputStream is available during render request processing in order to satisfy some servlet container implementation requirements and to allow for portlets that do not generate markup fragments. Portlets should use the raw OutputStream only for binary content and should use the Writer for text-based markup. If a portlet utilizes the OutputStream, the portlet is responsible for using the proper character encoding.

### Buffering

The portlet container may buffer portlet output before it is provided to the portal application for aggregation into the overall portal response.

The following MimeResponse interface methods allow the portlet to access and set buffering information:

* getBufferSize
* setBufferSize
* isCommitted
* reset
* resetBuffer
* flushBuffer
* isFlushed

These methods allow buffering operations to be performed regardless of whether the portlet is using an OutputStream or a Writer.

The portlet can use these methods during the header, render, and resource phases. The relationship of the portlet response and properties set by the portlet to the overall portal response transmitted to the client is dependent on the execution phase.

* During the header phase, the overall portal response to the client is not committed by definition, since the headers phase is executed before the portal has written the status code and headers in order to allow portlets to contribute header information. The isCommitted method must return false.

Portlets may set properties that result in HTTP headers or cookies and may write output data for the overall document HEAD section.

The buffering methods other than the isCommitted method apply only to the buffer that the portlet container makes available to the portlet, not to the overall portal response.

* During the render phase, the overall portal response may be committed depending on the portal implementation. A streaming portal may commit the overall response before all portlet response data is available, while a buffering portal may wait to commit the overall response until all portlets have provided response data. The isCommitted method must return the committed state of the overall portal response.

Portlets may or may not be able to set properties that result in HTTP headers or cookies depending on whether or not the overall portal response has been committed, and may write output data for aggregation into the overall document BODY section.

The buffering methods other than the isCommitted method apply only to the buffer that the portlet container makes available to the portlet, not to the overall portal response.

* During the resource phase, the portlet has nearly complete control over the overall response returned to the client and may determine when the overall portal response is committed.

The buffering methods including the isCommitted method apply to the overall portal response.

The getBufferSize method returns the size of the underlying buffer provided by the portlet container. If the portlet container uses no buffering, this method must return 0 (zero).

The portlet can request a preferred buffer size by using the setBufferSize method. The buffer assigned is not required to be the size requested by the portlet, but must be at least as large as the size requested. This allows the portlet container to reuse a set of fixed size buffers, providing a larger buffer than requested if appropriate. The portlet should call this method before any content is written to the OutputStream or Writer. If content has already been written, this method may throw an IllegalStateException.

If the portlet fills the buffer or uses the flushBuffer method, the portlet container must flush content from the portlet buffer to the portal application. After this occurs, the portlet can no longer affect any data that was contained in the buffer nor any properties that the portlet set.

The isFlushed method returns true if the portlet container has flushed the contents of the portlet buffer to the portal application and false otherwise.

The isCommitted method returns a boolean value indicating whether the portal application has written the overall portal response headers for transmission to the client.

The reset method clears data in the buffer that has not yet been flushed to the portal application. Properties set by the portlet prior to the reset call must be cleared as well.

The resetBuffer method clears content in the buffer if the portlet container has not yet flushed the portlet buffer to the portlet application without clearing the properties.

If the reset or resetBuffer method is called after the buffer has been flushed to the portal application, the portlet container must throw an IllegalStateException. The response and its associated buffer must be unchanged.

The portlet should use the isFlushed rather than the isCommitted method to determine whether the reset or resetBuffer methods can be called without throwing an exception.

### Predefined MimeResponse Properties

The MimeResponse interface defines some constants portlets can use to set, add, and read certain properties.

#### Cache properties

The MimeResponse interface defines the property names CACHE\_SCOPE, EXPIRATION\_CACHE, ETAG, and USE\_CACHED\_CONTENT along with the property values PRIVATE\_SCOPE and PUBLIC\_SCOPE that can be used for validating expired content, setting new expiration times and cache scopes. See Chapter 23 Cachingfor more details.

#### Namespaced Response Property

The portlet may optionally use the NAMESPACED\_RESPONSE constant to indicate to the portal application that the generated content will be completely namespaced. The portlet should set this property only when all markup id elements, form fields, etc. will be completely namespaced. One example where this might be useful is for portal applications that are form-based and thus need to re-write any forms included in the portlet markup.

The portlet must set this property using the setProperty method with a non-null value. The value itself is not evaluated. The value of the NAMESPACED\_RESPONSE constant is "X-JAVAX-PORTLET-NAMESPACED-RESPONSE", indicating that it is intended to be a header in the portlet response to the portal application.

Portlets should set the namespaced property during the header phase.

### Creating Portlet URLs

The following MimeResponse interface methods allow the portlet to create portlet URLs for inclusion in portlet markup. See Chapter 13 Portlet URLs for a description of portlet URLs.

* createRenderURL()
* createRenderURL(Copy)
* createActionURL()
* createActionURL(Copy)
* createResourceURL()

The createRenderURL(Copy) method returns a portlet RenderURL object. The render parameters will be copied to the RenderURL object depending on the Copy option.

* If the copy option is ALL, the current public and private render parameters will be copied to the URL.
* If the copy option is set to NONE, the URL object will contain neither public nor private render parameters.
* If the copy option is set to PUBLIC, only the public render parameters will be added to the URL. This setting corresponds to Portlet Specification 2.0 behavior.

The createRenderURL() method returns a portlet RenderURL object containing only the public render parameters. This method corresponds to using the copy option PUBLIC as defined above.

The createActionURL(Copy) method returns a portlet ActionURL object. The render parameters will be copied to the ActionURL object depending on the Copy option. The copy option behaves as described for the createRenderURL(Copy) method.

The createActionURL() method returns a portlet ActionURL object containing only the public render parameters. This method corresponds to using the copy option PUBLIC as defined above.

The createResourceURL() method returns a portlet ResourceURL object. The ResourceURL initially always contains both the public and the private render parameters. See Section 13.2 The Resource URL for details about the resource URL.

## RenderResponse Interface

The RenderResponse interface extends the MimeResponse interface and is used during render request execution. This interface allows a portlet to indicate the next possible portlet modes and generate content for inclusion into the overall portal response document BODY section.

### Next possible portlet modes

A portlet may indicate to the portal application the next possible portlet modes that make sense from the portlet point of view by providing a collection of candidate portlet modes through the setNextPossiblePortletModes method.

If the portlet provides a collection of next possible portlet modes, the portal should honor them by providing the end user with portlet mode choices limited to the provided collection of portlet modes or a subset of those modes based on access control considerations.

If the portlet does not set any next possible portlet modes, the portlet container should consider all portlet modes defined as supported portlet modes in the portlet configuration as possible new portlet modes.

### Deprecated Methods

The RenderResponse interface setTitle method has been deprecated with Portlet Specification Version 3.0. This methods has been moved to the HeaderResponse interface.

## ResourceResponse Interface

The ResourceResponse interface extends the MimeResponse interface to provide additional methods useful during resource serving. This interface allows a portlet to generate content that is directly served to the client, including binary content.

The portlet container may pre-set character encoding and locale, however, the portlet can override the preset values by setting the character encoding and the locale of the response using the setCharacterEncoding and setLocale methods, respectively.

The setStatus method sets the HTTP status code for this request. The status code should be a valid value as defined in IETF rfc2616 "Hypertext Transfer Protocol -- HTTP/1.1" . The portlet may use the status code constants "SC\_\*" defined in the HttpServletResponse interface to designate status code values. This method has no effect if it is called after the response has been committed. The getStatus method retrieves the status code set for the response.

The setContentLength and setContentLengthLong methods set the length of the content body in the response.

### Setting the Response Character Set

The portlet can set the character encoding for a resource response in several ways:

* Using the setCharacterEncoding method.
* Using the setContentType method. Calls to setContentType set the character encoding only if the given content type string provides a value for the charset attribute.
* Using the setLocale method and a locale-encoding-mapping-list mapping in the web.xml deployment descriptor[[14]](#footnote-14). Calls to setLocale set the character encoding only if neither the setCharacterEncoding nor the setContentType method has previously set the character encoding.

If the portlet does not set a character encoding using one of the possibilities listed above before calling getWriter, the portlet container must use UTF-8 as the default character encoding.

## HeaderResponse Interface

The HeaderResponse interface extends the MimeResponse interface. The portlet can use this interface during the header phase, which the portlet container must execute before the overall portal response has been committed in order to allow the portlet to contribute header information.

The portlet can use the addProperty and setProperty methods described previously to contribute cookies and HTTP header data. However, the portlet container may restrict the type of HTTP header data that the portlet can set in order enforce the portlet programming model and to avoid conflicts with other portlets. For example, the portlet will generally not be allowed to set headers relating to redirection or the overall response length.

The portlet can use the HeaderResponse interface setTitle method to set the preferred title. The portlet container is not required to use the preferred title set by the portlet.

In addition, the portlet can use the OutputStream and PrintWriter objects obtained through the HeaderResponse interface to write content for inclusion into the overall portal response document HEAD section. However, the portlet container may restrict the type of content that the portlet can write in order enforce the portlet programming model and to avoid conflicts with other portlets. For example, when the overall portal page is an HTML document, the portlet container may restrict the allowed content to the HTML tags that are valid in the document <HEAD> section.

### Page Resources and Dependencies

Portlets are typically rendered by a portal as components on an HTML page. A portlet might require resources such as JavaScript files and style sheets in order to operate properly. Due to the nature of the HTML markup language, these resources will be available to all portlets on the page. In addition, they only need to be placed on the page once even if they are used by several different portlets. For the purposes of this specification, such resources will be known as page resources.

Earlier versions of the portlet specification provided limited capability for allowing a portlet to add a page resource, see Section 15.1.3 Setting HEAD Section markup on page 89. The portlet can use the addProperty(String, Element) method to add a resource to a page, but there is no way to prevent duplicate entries from being made.

The dependency mechanism provides a means for identifying page resources through use of a three-part identifier consisting of the mandatory page resource name, the optional scope, and the optional version[[15]](#footnote-15).

The portlet can declare its dependencies on page resources using the page resource identifier. When the portal aggregates the page, it can use the dependencies declared by each portlet to generate an optimal set of page resources to be placed on the page, preventing duplicate entries and providing a version of each page resource that will satisfy all portlet dependencies.

Page resources can have dependencies among themselves, and can require loading in a particular order. A comprehensive solution to this problem is beyond the scope of this specification and is left to the portal implementation. However, where possible, the portal implementation should attempt to load the page resources in the order they are declared by the portlet. Statically declared page resource dependencies should be loaded before dynamically declared dependencies.

The portlet can statically configure resources through the deployment descriptor or through annotations as described in Section 28.2.8 Page Resource Dependencies on page 199.

In addition, the portlet can dynamically declare dependencies at run time using the HeaderResponse interface addDependency(String name, String scope, String version) method. It can declare a dependency and at the same time add a resource to fulfill the dependency using the addDependency(String name, String scope, String version, String markup) method.

Dynamic dependency declaration example:

headerResponse.addDependency("jQuery", "com.jquery", "1.23.3");

Dynamic dependency declaration with resource example:

String contextRoot = headerRequest.getContextPath();

StringBuilder txt = new StringBuilder(128);

txt.append("<link href='").append(contextRoot);

txt.append("/resources/css/infobox.css' rel='stylesheet' type='text/css'>");

headerResponse.addDependency("infobox", "org.apache.pluto", "0.4.0",

txt.toString());

The addDependency method should be used in preference to the PortletResponse interface addProperty( String, Element) method even when the resource is only used by a single portlet, since the addDependency method allows the portal to prevent duplicate page resources when the portlet is placed on the page multiple times.

The PortletResponse interface addProperty( String, Element) method should only be used for namespaced resources that only target a single portlet window.

## Lifetime of Response Objects

Each response object is valid only within the scope of a single request processing phase. Containers commonly recycle response objects in order to avoid the performance overhead of response object creation. The developer must be aware that maintaining references to response objects outside the request phase scope may lead to non-deterministic behavior.

# Portlet Filters

Filters are Java components that allow on the fly transformations of information in both the request to and the response from a portlet.

## What is a portlet filter?

A filter is a reusable piece of code that can transform the content of portlet requests and portlet responses. Filters do not generally create a response or respond to a request as portlets do, rather they modify or adapt the requests, and modify or adapt the response.

Among the types of functionality available to the developer needing to use filters are the following:

* The modification of request data by wrapping the request in customized versions of the request object.
* The modification of response data by providing customized versions of the response object.
* The interception of an invocation of a portlet after its call.

Portlet filters are modeled after the servlet filters in order to make them easy to understand for people already familiar with the servlet model and to have one consistent filter concept in Java EE.

## Main Concepts

The main concepts of this filtering model are described in this section. The application developer creates a filter by implementing one of the javax.portlet.filter.XYZFilter interfaces and providing a public constructor taking no arguments. The class is packaged in the portlet application WAR along with the static content and portlets that make up the portlet application.

A filter can be declared using the <filter> element in the portlet deployment descriptor. A filter or collection of filters can be configured for invocation by defining <filter-mapping> elements in the portlet deployment descriptor. This is done by mapping filters to a particular portlet by the portlet’s logical name, or mapping to a group of portlets using the ‘\*’ as a wildcard.

A filter can also be declared using the @PortletLifecycleFilter annotation.

### Filter Lifecycle

After deployment of the portlet application, and before a request causes the portlet container to access a portlet, the portlet container must build an ordered list of the portlet filters to be applied to the portlet as defined in the portlet configuration. The portlet container must instantiate a filter of the appropriate class for each applicable filter and called the filter init(FilterConfig config) method. The filter may throw an exception to indicate that it cannot function properly. If the exception is of type javax.portlet.UnavailableException, the portlet container may examine the isPermanent attribute of the exception and may choose to retry the filter at some later time.

The portlet container must instantiate only one instance of each configured filter per Java Virtual Machine. The portlet container provides the filter configuration through a FilterConfig object. The FilterConfig object contains a reference to the PortletContext for the portlet application, and the set of initialization parameters provided through the filter configuration.

When the portlet container receives an incoming request, it takes the first filter instance in the list and calls its doFilter method, passing in the PortletRequest, the PortletResponse, and a reference to the FilterChain object it will use.

Depending on the target method of the doFilter call, the PortletRequest and PortletResponse must be instances of the following interfaces:

* ActionRequest and ActionResponse for action request processing
* EventRequest and EventResponse for event request processing
* HeaderRequest and HeaderResponse for header request processing
* RenderRequest and RenderResponse for render request processing
* ResourceRequest and ResourceResponse for resource request processing

The doFilter method of a filter will typically be implemented following this or some subset of the following pattern:

1. The method examines the request information.
2. The method may wrap the request object passed in to its doFilter method with a customized implementation of one of the request wrappers (ActionRequestWrapper, EventRequestWrapper, HeaderRequestWrapper, RenderRequestWrapper, ResourceRequestWrapper) in order to modify request data.
3. The method may wrap the response object passed in to its doFilter method with a customized implementation of one of the response wrappers (ActionResponseWrapper, EventResponseWrapper, HeaderResponseWrapper, RenderResponseWrapper, ResourceResponseWrapper) to modify response data.
4. The filter may invoke the next component in the filter chain. The next component may be another filter, or if the filter making the invocation is the last filter configured for this chain, the next component is the target method of the portlet. The invocation of the next component is effected by calling the doFilter method on the FilterChain object, and passing in the request and response with which it was called or passing in wrapped versions it may have created.

The FilterChain interface doFilter method, provided by the portlet container, must locate the next component in the filter chain and invoke its doFilter method, passing in the appropriate request and response objects. Alternatively, the filter chain can block the request by not making the call to invoke the next component, leaving the filter responsible for filling out the response object.

1. After invocation of the next filter in the chain, the filter may examine the response data.
2. If the filter throws a UnavailableException during its doFilter processing, the portlet container must not attempt continued processing down the filter chain. It may choose to retry the whole chain at a later time if the exception is not marked permanent.
3. When the last filter in the chain has been invoked, the next component accessed is the target method on the portlet at the end of the chain.
4. Before a filter instance can be removed from service by the portlet container, the portlet container must first call the destroy method on the filter to enable the filter to release any resources and perform other cleanup operations.

### Wrapping Requests and Responses

Central to the notion of filtering is the concept of wrapping a request or response in order that it can override behavior to perform a filtering task. In this model, the developer has the ability to override existing methods on the request and response objects. The portlet should not add additional methods to the wrapper as further downstream wrappers may not honor these.

When a filter invokes the FilterChain interface doFilter method, the portlet container must ensure that the request and response objects that it passes to the next filter in the filter chain, or to the target portlet if the filter was the last in the chain, are either the original objects that were passed into the doFilter method or instances of the appropriate wrapper classes that wrap the original request and response objects.

### Filter Environment

A set of initialization parameters can be associated with a filter using the <init-params> element in the portlet deployment descriptor or using the @PortletLifecycleFilter annotation initParams element. The names and values of these parameters are available to the filter at runtime via the getInitParameter and getInitParameterNames methods on the filter’s FilterConfig object. Additionally, the FilterConfig affords access to the PortletContext of the portlet application for the loading of resources, for logging functionality, and for storage of state in the PortletContext attribute list.

In a server environment where CDI is present, the portlet container must instantiate the filter class using the CDI container in order to support dependency injection.

### Filter Configuration

#### Configuration through the Deployment Descriptor

A filter is defined in the deployment descriptor using the <filter> element. In this element, the programmer declares the following:

* filter-name: used to map the filter to a portlet
* filter-class: used by the portlet container to identify the filter type
* lifecycle: used to determine for which lifecycles the filter should be applied
* init-params: initialization parameters for a filter

Optionally, the programmer can specify a textual description and a display name for tool manipulation. The portlet container must instantiate exactly one instance of the Java class defining the filter per filter declaration in the deployment descriptor. Hence, two instances of the same filter class will be instantiated by the portlet container if the developer makes two filter declarations for the same filter class.

Here is an example of a filter declaration:

<filter>

<filter-name>Log Filter</filter-name>

<filter-class>com.acme.LogFilter</filter-class>

<lifecycle>ACTION\_PHASE</lifecycle>

</filter>

Once a filter has been declared in the portlet deployment descriptor, the <filter-mapping> element is used to define portlets in the portlet application to which the filter is to be applied. Filters can be associated with a portlet using the <portlet-name> element. Each filter mapping matching the portlet should be applied for this portlet, even if that result in one filter being applied more than once.

For example, the following code example maps the Log Filter to the SamplePortlet portlet:

<filter-mapping>

<filter-name>Log Filter</filter-name>

<portlet-name>SamplePortlet</portlet-name>

</filter-mapping>

Filters can be associated with groups of portlets using the ‘\*’ character as a wildcard at the end of a string to indicate that the filter must be applied to any portlet whose name starts with the characters before the "\*" character. Example:

<filter-mapping>

<filter-name>Log Filter</filter-name>

<portlet-name>\*</portlet-name>

</filter-mapping>

Here the Log Filter is applied to all the portlets within the portlet application, because every portlet name matches the ‘\*’ pattern.

The portlet container must build the chain of filters to be applied to a particular request using the same order that the filter mapping elements applicable to the portlet appear in the deployment descriptor. The portlet container is free to add additional filters at any place in this filter chain, but must not remove filters matching a specific portlet. Portlet container implementations may cache filter chains.

A portlet filter can be applied to the portlet action, event, header, render, and resource request processing methods. Thus the filter must define the lifecycle method for which the filter is written in the <lifecycle> element of the <filter> element. A filter can be applied to one or more lifecycle methods. The following constants are valid values for the <lifecycle> element:

* ACTION\_PHASE requesting that the portlet container processes this filter for action request processing. The filter implementation must implement the ActionFilter interface.
* EVENT\_PHASE requesting that the portlet container processes this filter for event request processing. The filter implementation must implement the EventFilter interface.
* HEADER\_PHASE requesting that the portlet container processes this filter for render request processing. The filter implementation must implement the HeaderFilter interface.
* RENDER\_PHASE requesting that the portlet container processes this filter for render request processing. The filter implementation must implement the RenderFilter interface.
* RESOURCE\_PHASE requesting that the portlet container processes this filter for resource request processing. The filter implementation must implement the ResourceFilter interface.

If the lifecycle declaration and portlet filter type do not match the portlet container is free to either reject the portlet at deployment time or ignore the filter.

Example:

<filter>

<filter-name>Sample Filter</filter-name>

<filter-class>com.acme.SampleFilter</filter-class>

<lifecycle>ACTION\_PHASE</lifecycle>

<lifecycle>RENDER\_PHASE</lifecycle>

</filter>

In this example the portlet filter is applied to the action and render phase.

#### Filter Configuration through Annotation

The @PortletLifecycleFilter annotation is a type annotation that designates a portlet filter class. The filter class may be any valid CDI managed bean provided by the portlet that implements one or more of the following interfaces. Each of the interfaces is associated with a specific request processing type, and thus lifecycle phase, corresponding to the discussion in the preceding section. The portlet container must determine the lifecycle phases to which the filter is to be applied through the interfaces implemented by the filter.

|  |  |
| --- | --- |
| **Filter** | **Request Processing Type** |
| javax.portlet.filter.ActionFilter | action request processing |
| javax.portlet.filter.EventFilter | event request processing |
| javax.portlet.filter.HeaderFilter | header request processing |
| javax.portlet.filter.RenderFilter | render request processing |
| javax.portlet.filter.ResourceFilter | resource request processing |

If the @PortletLifecycleFilter annotation is applied to a class that does not implement one of these interfaces, the portlet container must not place the portlet in service. The appropriate error handling and message display is left as a portal implementation detail.

If more than one filter class is annotated for a given filter interface type, the order of execution is determined by the ordinal element within the @PortletLifecycleFilter annotation. Annotated methods with a lower ordinal number are executed before methods with a higher ordinal number. If two annotated methods have the same ordinal number, both methods will be executed, but the execution order will be undetermined.

Portlets should generally configure filters either exclusively through the deployment descriptor or exclusively through annotations. If portlet filters are configured through both means, the annotated filter class methods will be executed before the filters configured through the deployment descriptor.

The annotated filter class can apply to multiple portlets within the portlet application. The names of the portlets for which the listener applies must be specified in the @PortletLifecycleFilter portletNames element. A wildcard character '\*' can be specified in the first portletName array element to indicate that the listener is to apply to all portlets in the portlet application. If specified, the wildcard character must appear alone in the first array element.

The @PortletLifecycleFilter annotation contains the following elements.

| **Element** | **Description** |
| --- | --- |
| portletNames | The portlet names for which the listener applies. |
| ordinal | The ordinal number for this annotated method. |
| description | The locale-specific filter description for tool use |
| displayName | The locale-specific display name for tool use |
| locale | The locale applicable to the description and displayName elements |

The following example shows a filter configured for the portlets TestPortlet1 and TestPortlet2.

@PortletLifecycleFilter(portletNames = {"TestPortlet1", "TestPortlet2"},

ordinal=200)

public class FilterTest2 implements RenderFilter, ActionFilter {

@Override

public void init(FilterConfig config) throws PortletException {

}

@Override

public void destroy() {

}

@Override

public void doFilter(ActionRequest request, ActionResponse response,

FilterChain chain) throws IOException, PortletException {

// do something

}

@Override

public void doFilter(RenderRequest request, RenderResponse response,

FilterChain chain) throws IOException, PortletException {

// do something

}

}

# Coordination between Portlets

The Portlet Specification defines the following mechanisms for coordination between portlets:

* Portlets and other artifacts in the same web application can share data through the session in the application scope (see Section 19.2 Portlet Session Scope)
* Portlets can share render state through the public render parameters (see 11.2.1 Public Render Parameters).

Using public render parameters instead of events avoids the additional event request processing overhead and allows the end-user to use the browser navigation and bookmarking if the portal stores the render parameters in the URL.

As an example, a weather portlet might display the weather of a selected city. It therefore uses the public render parameters for encoding the zip code. Another portlet on the page could allow the location to be selected and set the public render parameter appropriately.

* Portlets can send and receive portlet events.

Portlet events are intended to allow portlets to react to actions or state changes not directly related to an interaction of the user with the portlet receiving the event. Events can be generated by the portal or portlet container, or can be generated as the result of a user interaction with other portlets.

The portlet event model is loosely coupled and brokered, allowing the event routing to be determined at runtime. The manner in which event routing between portlets is configured and performed is portal implementation specific.

Portlet events are not a replacement for reliable messaging (use other Java EE technologies such as the Java Message Service for this purpose). Portlet events are not guaranteed to be delivered and thus the portlet should always work in a meaningful manner even if some or all events are not delivered.

In response to an event, a portlet may update the render state and may publish new events for delivery to other portlets and thus trigger state changes on these other portlets.

As an example, a shopping cart portlet may fire an event when items are purchased. An inventory portlet might update its state when such an event is received.

Note that it is outside the scope of this specification to define how portlets are wired together for sharing public render parameters or routing events. Nor does the Portlet Specification define how portlets relate to each other or to a portal page. This is done on portal application level and is not reflected in the Java Portlet API or portlet configuration.

### EventPortlet Interface

In order to receive events the portlet must implement the EventPortlet interface in the javax.portlet package or implement a @EventMethod annotated method. The portlet container will invoke event processing for each event targeted to the portlet, and provide the portlet with an EventRequest and EventResponse object. Events are targeted to a specific portlet window in the current client request.

Portlet events can be set during the action or event processing phases. Events are processed during the event phase.

### Event Declaration

The portlet should declare all event definitions as well as the events that it publishes and processes in the portlet configuration. The declaration can take place through the portlet deployment descriptor or through use of annotations. See Sections 28.1.8 Event Configuration on page 190 and 28.2.10 Event on page 201 for more information on declaring events in the portlet application configuration and portlet configuration, respectively.

The event definition provides the portlet container with the event qualified name and the event payload type for each event. Event names are represented as QNames in order to make them uniquely identifiable.

The events that the portlet publishes and those that it can process are declared separately from the event definition. The set of publishing events can differ from the set of processing events.

Portlet containers will typically route only declared processing events to the portlet.

#### Event Payload

The default XML to Java mapping that every container should support is the JAXB mapping. Portlet containers are free to support additional mapping mechanisms beyond the JAXB mapping. For optimization purposes in local Java runtime environments the portlet container can use Java serialization or direct Java object passing for the event payload. The portlet must not make any assumptions on the mechanism the portlet container chooses to pass the event payload.

The Portlet Specification uses the JAXB XML Binding 2.0 for defining event payload data that may be transported across a network via remote protocols such as Web Services for Remote Portlets (WSRP) 2.0.

The event payload must be defined using the JAXB annotations in the Java class definition and by defining the Java class name of the event payload object in the portlet configuration using the deployment descriptor value-type element or using the @PortletConfiguration annotation.

The event payload must have a valid JAXB binding, or be in the list of Java primitive types and standard classes defined in the JAXB 2.0 specification section 8.5.1 or section 8.5.2, and must implement java.io.Serializable. Otherwise the portlet container must throw a java.lang.IllegalArgumentException. The primitive type xsd:anyURI must be mapped to java.net.URI rather than the JAXB default java.lang.String in order to avoid losing semantics

#### Events not declared in the Configuration

The portlet can publish events that are not declared in the portlet configuration. Note that if the events are not declared in the portlet configuration, the portlet container might not be able to route them. How the portlet container handles such events is left as an implementation detail.

### Processing Events

The portlet may optionally receive events from other portlets, such as an item being added to a shopping cart, or may receive container events, such as a portlet mode change event.

The portlet can access the triggering event object using the EventRequest interface getEvent method. This method returns an object of type Event encapsulating the current event name and value. The event must always have a name and may optionally have a value.

The event name can be either retrieved with the getQName method that returns the complete QName of the event, or with the getName method that only returns the local part of the event name.

If the event has a value it must be an object of the type defined in the portlet configuration.

### Sending Events

The portlet may publish events using the StateAwareResponse interface setEvent method during action or event request processing. The StateAwareResponse methods are exposed via the ActionResponse and EventResponse interfaces. The portlet can call the setEvent multiple times during action or event request execution in order to fire multiple events. The events fired will be processed by the portlet container after request processing has finished.

Events can be published using either the full qualified event name with the setEvent(QName, Serializable) method, or using the event qualified name local part with the setEvent(String, Serializable) method. If only the local part is specified, the portlet container must use the default namespace defined in the portlet configuration. If no default namespace is provided, the portlet container must use the XML default namespace javax.xml.XMLConstants.NULL\_NS\_URI.

The event payload must be an object of the type defined in the portlet configuration.

### Event Processing

Events are valid only within current client request processing. The portlet container must therefore deliver all events within the current client request. Event delivery is not guaranteed and the container may restrict event delivery in a meaningful manner, e.g. in order to prevent endless loops.

Events are not ordered and the container may re-order the received events before distributing them. Portal applications should distribute events returned by a single portlet in the order the portlet called the setEvent method but this ordering is not guaranteed. Thus portlet developers should rely on other mechanisms such as adding ordering information to the event payload if required.

Event distribution is non-blocking and can happen in parallel for different portlet windows.

Event distribution must be serialized for a specific portlet window per client request so that at any given time a portlet window is processing only one event for the current client request. Conceptually, the portlet container should queue the events for each portlet window per client request. When processing the queue the container should take any previously returned event response data, like render parameters, portlet mode, window state, into account and supply these updated values with the next event request.

Note that event processing for different portlets within the current client request may happen in parallel and that therefore if portlets update shared data like public render parameters or information in the application session during event processing, the last state change wins.

Container raised events are issued by the portlet container and not a portlet. The portlet should not publish container events, only process them. Container events published by the portlet should be ignored by the portlet container. If a portlet would like to receive a container raised event it should declare the event in the portlet configuration.

### Exceptions during event processing

A portlet may throw a PortletException, a PortletSecurityException or a UnavailableException during event processing.

A PortletException signals that an error has occurred during the processing of the event and that the portlet container should take appropriate measures to clean up the event processing. If a portlet throws an exception during event processing, all operations performed on the EventResponse object, such as setting render parameters and setting events, before the exception occurs must be ignored. The portlet container should continue processing other events targeted to the portlet and to other portlets participating in the current client request. Error handling is otherwise left to the portal implementation.

A UnavailableException signals that the portlet is unable to handle requests either temporarily or permanently.

Ifthe exception indicates permanent unavailability, the portlet container must remove the portlet from service immediately, call the portlet’s destroy method, and release the portlet object. A portlet that throws a permanent UnavailableException must be considered unavailable until the portlet application containing the portlet is restarted.

Ifthe exception indicates temporary unavailability, then the portlet container may choose not to route any requests to the portlet during the time period of the temporary unavailability.

The portlet container may choose to ignore the distinction between a permanent and temporary unavailability and treat all occurrences of UnavailableException as permanent and remove the portlet from service.

A RuntimeException thrown during the event handling must be handled as a PortletException.

## Predefined Container Events

The Web Service for Remote Portlets (WSRP) specification predefines some common events that should be leveraged when requiring an event for one of the following scenarios:

* Event handling failed (wsrp:eventHandlingFailed) –This is a portal application generated event which signals to the portlet that the portal application detected that errors occurred while distributing events. As a simple notification, this event carries no predefined payload, but does use an open content definition.
* Navigations context changed (wsrp:newNavigationalContextScope)– allowing the portlet to manage its own navigational context in a consistent manner with the navigational context managed by the portal application.
* New portlet mode (wsrp:newMode) – indicating to the portlet that it has been put into a new portlet mode and allowing the portlet to pre-set some state before getting rendered in this new mode.
* New window state(wsrp:newWindowState) – indicating to the portlet that it has been put into a new window state and allowing the portlet to pre-set some state before getting rendered in this window state.

See section 5.11 of the Web Services for Remote Portlets specification V2.0 for more details.

Portlet containers may optionally provide support for these events. Portlet containers that support the predefined events should deliver these events to all portlets that declare processing event support for them in the portlet configuration.

# Portlet Preferences

Portlets are commonly configured to provide a customized view or behavior for different users. This user-specific configuration is represented as a persistent set of name-value pairs and it is referred to as portlet preferences. The preference attribute name or key is a String and the preference attribute value is a String array. The portlet container is responsible for retrieving and storing the portlet preferences.

Portlet preferences are intended to store basic configuration data for portlets. It is not the purpose of the portlet preferences to replace general purpose databases.

## PortletPreferences Interface

Portlets access the portlet preference attributes through the PortletPreferences interface. Portlets can access to the associated PortletPreferences object while they are processing requests. Portlets may only modify preferences attributes during action phase or resource phase execution.

To access and manipulate preference attributes, the PortletPreferences interface provides the following methods:

* getNames
* getValue
* setValue
* getValues
* setValues
* getMap
* isReadOnly
* reset
* store

The getMap method returns a Map<String, String[]> object containing all current preference key – value pairs. The portlet container must assure that altering the returned Map object by modifying values or adding or deleting entries does not result in changes to the stored preference values.

The getValue and setValue methods are convenience methods for dealing with single values. If a preference attribute has multiple values, the getValue method returns the first value. The setValue method sets a single value into a preferences attribute, replacing any existing values.

The following code sample demonstrates how a stock quote portlet would retrieve from its preferences object the preferred stock symbols, the URL of the backend quoting services, and the quote refresh frequency.

PortletPreferences prefs = req.getPreferences();

String[] symbols =

prefs.getValues("preferredStockSymbols",

new String[]{"ACME","FOO"});

String url = prefs.getValue("quotesFeedURL", null);

int refreshInterval =

Integer.parseInt(prefs.getValue("refresh","10"));

The reset(String key) method must reset the preference attribute designated by the supplied key to its default value. The default value for a given portlet preference key can be set in the portlet configuration or though portlet container implementation specific means. If no default value is available, the preference attribute must be deleted.

If a preference attribute is read only, the setValue, setValues and reset methods must throw a ReadOnlyException when the portlet is in any of the standard modes.

The store method must persist all the changes made to the PortletPreferences object into the persistent store. If the call returns successfully, it is safe to assume the changes are permanent. The store method must be conducted as an atomic transaction regardless of how many preference attributes have been modified. The portlet container implementation is responsible for handling concurrent writes to avoid inconsistency in portlet preference attributes. All changes made to PortletPreferences object not followed by a call to the store method must be discarded when the portlet completes action request, event request, or resource request processing. If the store method is invoked within the render phase, it must throw an IllegalStateException.

The PortletPreferences object must reflect the current values of the persistent store when the portlet container invokes any portlet request method.

## Preference Attributes Scopes

The Portlet Specification assumes preference attributes are user specific, but it does not make any provision at the API level or at the semantic level for sharing preference attributes among users.

However, it enables sharing of preferences and enables definition of different levels of portlet entities (see Section 4.3.1 Portlet Definition and Portlet Entity). Portlet preference sharing and portlet entity level definition is not covered by the Portlet Specification and is implementation specific.

If a portal/portlet-container implementation provides an extension mechanism for sharing preference attributes, it should be well documented. Sharing preference attributes may have significant impact on the behavior of a portlet. In many circumstances, it could be inappropriate to share attributes that are meant to be private or confidential for a specific user.

## Preference Attributes definition

The configuration specified in the deployment descriptor defines portlet preference attribute names, default values, and an indicator that can mark a preference attribute to be read-only. In addition, preference attributes can be specified with the @PortletConfiguration annotation via the prefs annotation attribute.

If a preference attribute is not defined to be read-only, the preference attribute must be modifiable when the portlet is processing an action, event, or resource request in any of the standard portlet modes (VIEW, EDIT or HELP).

Portlets may change the value of modifiable preference attributes using the setValue, setValues and reset methods of the PortletPreferences interface. Administrators who deploy portlets may set the deployment descriptor preference read-only element to true to fix certain preference values at deployment time. Portals may allow changing read-only preference attributes while performing administration tasks.

Portlets are not restricted to use only the preference attributes defined in the portlet configuration. They can also programmatically add preference attributes using names not defined in the portlet configuration. These preferences attributes must be treated as modifiable attributes.

Portal administration and configuration tools may use and change default preference attributes when creating a new portlet preferences objects. In addition, the portal may further constrain the modifiability of preferences values.

### Localizing Preference Attributes

The Portlet Specification uses the Java SE ResourceBundle classes to provide localization.

To enable localization support of preference attributes for administration and configuration tools, developers should adhere to the following naming convention for entries in the portlet’s ResourceBundle (see Section 28.2.5 Portlet Resource Bundle).

Entries for preference attribute descriptions should be constructed as ‘javax.portlet.preference.description.<attribute-name>', where <attribute-name> is the preference attribute name.

Entries for preference attribute names should be constructed as ‘javax.portlet.preference.name.<attribute-name>', where <attribute-name> is the preference attribute name. These values should be used as localized preference display names.

Entries for preference attribute values that require localization should be constructed as 'javax.portlet.preference.value.<attribute-name>.<attribute-value>', where <attribute-name> is the preference attribute name and <attribute-value> is the localized preference attribute value.

## Validating Preference Values

The portlet can register a preferences validator in the portlet configuration that is invoked when the PortletPreferences interface store method is invoked. The registration can be performed through the portlet deployment descriptor or through annotations.

The preferences validator contains a method that accepts the current PortletPreferences object. The store method must invoke the preferences validator method before writing the changes to the persistent store. If the validation fails, the preferences validator must throw a ValidatorException. If a ValidatorException is thrown, the portlet container must cancel the store operation and it must propagate the exception to the portlet. If the validation is successful, the store operation must be completed. The portlet should not modify the portlet preferences during validation.

When creating a ValidatorException, portlet developers may include the set of preference attributes that caused the validator to fail. It is left to the developers to indicate the first preference attribute that failed or the name of all the invalid preference attributes.

The preferences validator must be coded in a thread safe manner as the portlet container may invoke it concurrently due to parallel request processing.

### Configuration through the Deployment Descriptor

To register a preferences validator in the deployment descriptor, the developer must provide a class implementing the PreferencesValidator interface. The PreferencesValidator interface contains a single validate method that accepts a PortletPreferences object.

The preferences validator class can be associated with the preferences definition in the deployment descriptor, as shown in the following example.

<!—- Portlet Preferences -->

<portlet-preferences>

...

<preferences-validator>

com.foo.portlets.XYZValidator

</preferences-validator>

</portlet-preferences>

### Configuration through Annotation

The @PortletPreferencesValidator annotation designates a preferences validator class. The method may be located in any valid CDI managed bean class provided by the portlet that implements the PreferencesValidator interface.

The annotated PreferencesValidator class can apply to multiple portlets within the portlet application. The names of the portlets for which the listener applies must be specified in the @PortletPreferencesValidator portletNames element. A wildcard character '\*' can be specified in the first portletName array element to indicate that the listener is to apply to all portlets in the portlet application. If specified, the wildcard character must appear alone in the first array element.

The @PortletPreferencesValidator annotation contains the following elements.

|  |  |
| --- | --- |
| **Element** | **Description** |
| portletNames | The portlet names for which the preference validator applies. |

# Sessions

To build effective portlet applications, it is imperative that requests from a particular client be associated with each other. The idea of a session was introduced in order to provide continuity between requests from the same client. Session tracking allows requests from a specific client to be associated with a session.

There are many session tracking approaches such as use of HTTP Cookies, SSL Sessions or URL rewriting. To free the programmer from having to deal with session tracking directly, this specification defines a PortletSession interface that allows the portlet container to use any of the approaches to track a user’s session without involving the developers in the nuances of any one approach. The actual session tracking approach used is a portlet container implementation detail and will not be covered further in this specification.

## Creating a Session

A session is considered "new" when it is only a prospective session and has not been established. Because the Portlet Specification is designed around a stateless request-response protocol (HTTP), a session is considered to be new until a client "joins" it. A client joins a session when session tracking information has been returned to the server indicating that a session has been established. Until the client joins a session, it cannot be assumed that the next request from the client will be recognized as part of a session.

The session is considered to be "new" ifeither of the following is true:

* The client does not yet know about the session
* The client chooses not to join a session

These conditions define the situation where the portlet container has no mechanism by which to associate a request with a previous request. The PortletSession interface isNew method allows the portlet to query whether a portlet session is new.

The portlet can obtain or create a portlet session through use of the PortletRequest interface getPortletSession method. The PortletSession object so obtained is only valid within the current client request. It should be noted that during asynchronous processing, the PortletSession object remains valid until asynchronous processing is completed. See Chapter 21 Asynchronous Support on page 124.

The portlet container must ensure that portlets within the same portlet application participate in the same portlet session. However, through namespacing performed by the portlet container. each portlet can have its own unique portlet session attributes, as will be described further below.

## Portlet Session Scope

PortletSession objects must be scoped at the portlet application context level. The portlet container must not share the PortletSession object or the attributes stored in it among different portlet applications or among different user sessions.

Each portlet application has its own distinct PortletSession object per user session.

## Relationship to HttpSession

A portlet application is a web application. The portlet application may contain resources such as servlets and JSPs in addition to portlets. The portlet session is based on the servlet session to allow information sharing between portlets and other web components. The portlet container must store information placed into the portlet session in a manner that makes the information available through the servlet session. The information must be available to any portlet, servlet or JSP within the same portlet application.

The portlet container must ensure that all attribute objects placed in the PortletSession object are also available through the portlet application HttpSession object. Also, all attributes placed into the HttpSession object must be made available through the PortletSession object. However, the attribute names used to access the objects through the HttpSession are subject to namespacing considerations described in the following section.

If the HttpSession object is invalidated, the portlet container must also invalidate the corresponding PortletSession object. If a portlet invalidates the PortletSession object, the portlet container must invalidate the corresponding HttpSession object.

Portlet applications can use servlet session lifecycle listeners to monitor the portlet session (see 2.2 Using Servlet Application Lifecycle Events).

## Binding Attributes to a Portlet Session

A portlet can bind an object attribute to a PortletSession by name.

The PortletSession interface defines two portlet session scopes for storing objects, APPLICATION\_SCOPE and PORTLET\_SCOPE. The APPLICATION\_SCOPE scopes the object to the portlet application, while the PORTLET\_SCOPE scopes the object to the portlet. The portlet session scope affects the name by which the portlet container stores the attribute in the underlying servlet session. The PortletSession interface methods use PORTLET\_SCOPE by default.

Any object stored in the portlet session with APPLICATION\_SCOPE is stored under the same attribute name in the HttpSession, and is available under that name to any other portlet or web component that belongs to the same portlet application.

Objects stored in the portlet session with PORTLET\_SCOPE must be available to the portlet during requests targeted to the portlet window that was active when the objects were stored. Since the same portlet might appear in multiple portlet windows within a single session, the portlet container must namespace such attributes when storing them in the HttpSession so that they can be identified as being associated with a specific portlet window.

The portlet container must encode the attribute names of PORTLET\_SCOPE scoped portlet session attributes in the HttpSession according to the following scheme:

javax.portlet.p.<ID>?<ATTRIBUTE\_NAME>

The <ID> field is a unique identifier for the portlet window assigned by the portlet container. This must the same ID string returned by the PortletRequest.getWindowID() method. The <ID> must not contain a ‘?’ character.

The <ATTRIBUTE\_NAME> field is the attribute name provided by the portlet when setting the PORTLET\_SCOPE scoped object on the portlet session.

The portlet need not be concerned with the PORTLET\_SCOPE naming scheme when accessing the attributes through the PortletSession object. The portlet uses the plain attribute name with no prefix to access the attribute. The portlet container must handle the required attribute name encoding and decoding transparently for the portlet.

However, PORTLET\_SCOPE portlet session scoped attributes are available through the HttpSession only through the encoded names as described above.

The PortletSessionUtil class provides utility methods to handle portlet session attributes read from the HttpSession object. They are intended to be used to decode portlet session attribute names within HTTP session listeners and other web components that provide access to the HttpSession. The decodeAttributeName method returns the attribute name used by the portlet when setting the attribute. Any prefix added to the attribute name by the portlet container for portlet session attributes stored with PORTLET\_SCOPE is removed. The decodeScope method returns the portlet session scope with which the given attribute name was stored. Portlet developers should always use the PortletSessionUtil class to deal with attributes stored with PORTLET\_SCOPE when accessing them through the servlet API.

### PortletSession Interface Methods

The PortletSession interface methods getCreationTime, getId, getLastAccessedTime, getMaxInactiveInterval, invalidate, isNew and setMaxInactiveInterval must provide the same functionality as the HttpSession interface methods with identical names.

The PortletSession interface getAttribute, setAttribute, removeAttribute and getAttributeNames methods must provide the same functionality as the methods of the HttpSession interface with identical names. The getAttributeMap method has no correspondence to an HttpSession method. It returns an immutable Map<String, String[]> containing the current attributes.

The behavior of the latter methods is influenced by the portlet session scope. Each of the methods provides a variant that accepts a portlet session scope argument and one that does not. The variant that does not accept a portlet session scope argument must assume PORTLET\_SCOPE portlet session scope. The portlet container must encode and decode the attribute name depending on the portlet session scope transparently for the portlet as described in Section 19.4.

The portlet container must execute PortletSession interface methods, for example the setAttribute and removeAttribute methods, in a thread-safe manner. The portlet container is responsible for handling concurrent writes to avoid inconsistency in portlet session attributes. However, the portlet developer is responsible for providing any necessary synchronization for the session attribute objects themselves. This follows the guidance for HttpSession attributes[[16]](#footnote-16).

An example of setting a portlet session attribute follows.

PortletSession session = request.getSession(true);

URL url = new URL("http://www.foo.com");

session.setAttribute("home.url",url,PortletSession.APPLICATION\_SCOPE);

session.setAttribute("bkg.color","RED",PortletSession.PORTLET\_SCOPE);

## Modifying Objects in the Portlet Session

The developer should consider the portlet execution phase when modifying portlet session attributes in order to avoid concurrency issues and inconsistent data.

Modifying or setting PORTLET\_SCOPE render state scoped attributes during the action phase will likely not create any concurrency issues, since the attribute is scoped to the portlet window and generally only a single request from a specific portlet window can be processed.

Modifying or setting APPLICATION\_SCOPE render state scoped attributes during the action phase is more likely to create concurrency issues, since these attributes are shared with other portlets and web components that may run in parallel and change the same attribute.

The portlet API does not prevent portlets from modifying portlet session attributes during the render or resource phases. However, developers are in general discouraged from doing so.

Portlet session attributes should only be modified during the resource phase if the client request was submitted using the HTTP POST, PUT, or DELETE method.

Modifying portlet session attributes during the render phase or during resource phase processing initiated through the HTTP GET method is strongly discouraged as it makes rendering non-idempotent. This is especially true for APPLICATION\_SCOPE attributes, since they can be shared across portlets and other web components.

## Reserved HttpSession Attribute Names

Session attribute names starting with "javax.portlet." are reserved for usage by the Portlet Specification and for portlet container vendors. A portlet container vendor may use this reserved namespace to store implementation-specific components. Application developers must not use attribute names starting with this prefix.

## Session Timeouts

The portlet session follows the timeout behavior of the servlet session as defined in the *Servlet Specification Version 3.1, Section .7.5 Session Timeouts*.

## Last Accessed Times

The portlet session follows the last accessed times behavior of the servlet session as defined in the *Servlet Specification Version 3.1 Section .7.6 Last Accessed Times*.

## Important Session Semantics

The portlet session follows the same semantic considerations as the servlet session as defined in the *Servlet Specification Version 3.1, Section .7.7 Important Session Semantics*.

These considerations include *Threading Issues*, *Distributed Environments,* and *Client Semantics*.

# Managed Bean Support

Support for Contexts and Dependency Injection for Java (CDI) is part of the Java EE web profile. The Portlet Specification provides support for managed beans through portlet instantiation, custom scopes for CDI, and by defining injectable portlet artifacts.

## Portlet Instantiation

The portlet must have a default constructor in order to be recognized as a valid CDI managed bean class. When CDI is available and the portlet classes are contained in a valid bean archive, the portlet container must instantiate the portlet through the CDI managed bean container in order to allow the portlet to use CDI features.

This holds for portlets that use the extended method annotations described in Section 4.8 Extended Annotation-Based Dispatching on page 33 as well as for portlets that extend GenericPortlet or that directly implement the Portlet interface.

When the portlet container uses the CDI container to instantiate the portlets, it must respect any scope annotations on the portlet beans. Generally, portlets should be @ApplicationScoped in order to avoid unnecessary object creation and destruction.

It should be noted that regardless of the bean scope, the Portlet interface init(PortletConfig) method will only be called a single time when the portlet is taken into service. If the portlet bean scope is other than @ApplicationScoped or @Dependent, this can potentially lead to unintended effects.

Portlets that do not directly or indirectly implement the Portlet interface can use the CDI @PostConstruct and @PreDestroy annotations to properly handle bean initialization and tear down. A PortletConfig object can be injected as necessary.

## Custom Scopes

The Portlet Specification provides three custom scopes for use with CDI managed beans.

### Portlet Session Scope

The portlet session scope is a passivating scope that associates managed beans with the portlet session rather than to the servlet session.

The CDI @SessionScoped built-in scope associates managed beans with the servlet session. The Portlet Specification introduces the @PortletSessionScoped annotation, which the portlet developer can use to designate beans that are to be placed into the portlet session.

The @PortletSessionScoped annotation provides the single value element that specifies the portlet scope into which the managed bean is to be placed. See Section 19.4 Binding Attributes to a Portlet Session for discussion of the portlet scopes. Possible values are:

* PortletSession.PORTLET\_SCOPE – places the managed bean into the portlet session.
* PortletSession.APPLICATION\_SCOPE – places the bean into the portlet application scope. This scope has the same effect as the CDI @SessionScoped built-in scope.

The default value is PortletSession.PORTLET\_SCOPE.

The portlet container must treat @SessionScoped beans as @PortletSessionScoped beans with APPLICATION\_SCOPE.

### Portlet Request Scope

The CDI @RequestScoped built-in scope associates managed beans with the servlet request. The Portlet Specification introduces the non-passivating @PortletRequestScoped annotation, which the portlet developer can use to designate beans that are to be associated with the portlet request.

The @PortletRequestScoped annotation provides no configuration elements.

The portlet container must treat @RequestScoped beans as @PortletRequestScoped beans within a portlet application.

### Render State Scope

The render state scope is a passivating scope that associates managed beans with the render state. The bean state is stored as a render parameter within the render state.

The Portlet Specification introduces the @RenderStateScoped annotation, which the portlet developer can use to designate beans that are to be stored as part of the render state. To enable passivation under observance of the portlet phase model and to allow the managed bean state to be stored in a render parameter values array, the managed bean annotated with @RenderStateScoped must implement the PortletSerializable interface. If the @RenderStateScoped annotation is applied to a managed bean that does not implement PortletSerializable, the portlet container must not place the portlet in service.

The @RenderStateScoped annotation provides the following elements.

* The paramName optional element specifies a render parameter name under which the managed bean state is to be stored. If no parameter name is specified, the portlet container must assign a parameter name that is unique within the portlet application.

The portlet container must make the managed bean state available through the RenderParameters object and through the client-side portlet API by the specified or assigned render parameter name.

If the specified render parameter name matches a public render parameter name specified in the portlet application configuration, the portlet container must store the managed bean state as the public render parameter value.

The PortletSerializable interface defines the methods serialize and deserialize. The serialize method returns a String[] object, while the deserialize method accepts a String[] object as an argument.

The portlet developer must implement the serialize method to return a String[] object that contains all information necessary to reconstruct the managed bean state. The portlet developer must implement the deserialize method to reconstruct the managed bean state from the given String[] object.

In accordance with the rules for updating render state, the portlet container must serialize the @RenderStateScoped managed bean state using the PortletSerializable interface serialize method during the action phase after each portlet action request or event request execution that uses the managed bean. The portlet container must store the serialized bean state as a render parameter in the render state using the associated render parameter name.

The portlet container must not serialize and store the managed bean state after render phase or resource phase execution. The portlet container must discard any changes made to the managed bean state during these execution phases.

When the managed bean is first accessed during portlet request processing, the portlet container must retrieve the stored bean state under the associated render parameter name from the render state and provide it as the argument to the PortletSerializable interface deserialize method.

A @RenderStateScoped managed bean may not be further differentiated through use of qualifiers, since doing so could lead to naming conflicts between the associated render parameter names.

## Portlet Predefined Beans

The portlet container must make certain portlet artifacts available as predefined beans for injection into portlet classes. The portlet container will generally do this by implementing appropriate CDI producer methods or fields for the artifacts, although the exact mechanism for doing so is an implementation detail not covered by this specification.

The injectable objects must be @PortletRequestScoped where possible so that they won’t become stale or conflict with corresponding objects from other portlets when used within common libraries. Also, many of the objects are tied to the portlet request processing lifecycle and become invalid when request processing completes.

The portlet should not override the bean scope assigned by the portlet container at the injection point, as the effect will be undefined and likely undesirable.

Some of the beans defined by the Portlet Specification must be dependent scoped due to technical limitations. The portlet developer should use the dependent scoped beans only within @PortletRequestScoped or @RenderStateScoped beans in order to avoid accessing stale data.

Some of the injectable objects only exist within certain portlet processing phases. For example, the ActionRequest object only exists during action request processing, while the RenderRequest object only exists during render request processing.

The portlet container producer method or field must produce or contain null if accessed during portlet request execution during which the portlet artifact does not logically exist. The portlet developer should note that the CDI container will throw a runtime exception if the corresponding injection point is accessed in this case.

Some of the injectable objects are of the same bean type. For example, the portlet namespace and context path but return String objects. Such objects must be distinguished through use of qualifiers.

The portlet container must provide the predefined beans defined in the table below. If the bean type differs from the artifact name, the actual bean type is specified in parentheses.

The portlet container must produce named beans that can be used in a JSP or a JavaServer™ Faces Facelet. The table below provides the required EL names.

### Portlet Request Scoped Beans

The portlet container must produce the beans described in this section as @PortletRequestScoped beans.

| **Artifact** | **Bean EL Name** | **Qualifier** | **Valid during** |
| --- | --- | --- | --- |
| PortletConfig | portletConfig | - | all |
| PortletRequest | portletRequest | - | all |
| PortletResponse | portletResponse | - | all |
| ActionRequest | actionRequest | - | action |
| ActionResponse | actionResponse | - | action |
| HeaderRequest | headerRequest | - | header |
| HeaderResponse | headerResponse | - | header |
| RenderRequest | renderRequest | - | render |
| RenderResponse | renderResponse | - | render |
| EventRequest | eventRequest | - | event |
| EventResponse | eventResponse | - | event |
| ResourceRequest | resourceRequest | - | resource |
| ResourceResponse | resourceResponse | - | resource |
| StateAwareResponse | stateAwareResponse | - | action, event |
| MimeResponse | mimeResponse | - | header, render, resource |
| ClientDataRequest | clientDataRequest | - | action, resource |
| RenderParameters | renderParams | - | all |
| MutableRenderParameters | mutableRenderParams | - | action, event |
| ActionParameters | actionParams | - | action |
| ResourceParameters | resourceParams | - | resource |
| PortletContext | portletContext | - | all |
| PortletMode | portletMode | - | all |
| WindowState | windowState | - | all |
| PortletPreferences | portletPreferences | - | all |
| Cookies (List<Cookie>) | cookies | - | all |
| PortletSession | portletSession | - | all |
| Locales (List<Locale>) | locales | - | all |

### Dependent Scoped Beans

The portlet container may produce the beans described in this section as @Dependent scoped beans.

|  |  |  |  |
| --- | --- | --- | --- |
| **Artifact** | **Bean EL Name** | **Qualifier** | **Valid during** |
| Namespace (String) | namespace | @Namespace | all |
| ContextPath (String) | contextPath | @ContextPath | all |
| WindowID (String) | windowId | @WindowId | all |
| Portlet name (String) | portletName | @PortletName | all |

# Asynchronous Support

Portlet containers are typically implemented using pools of resources, such as threads and buffers, to handle incoming requests. Resources are allocated for each incoming request and are released when the request is complete.

Some requests take a long time to respond since they need to wait for database access, a remote connection, or access to some other limited resource. Such requests will block the portlet container request resource pools while they are executing. If there are many such requests waiting at the same time, a great deal of portlet container request pool resources can become blocked, leading to overall slow response even for requests that do not require access to slow or limited resources.

Portlet Specification 3.0 introduces asynchronous support to help alleviate this problem. Using asynchronous support, the developer can delegate execution of a long-running request to a separate thread, allowing the initial portlet request to return in order to free up portlet container resources associated with the request without sending a response to the client. The long-running request can then send the response to the client and close the response at a later time when the required data becomes available.

The portlet specification describes asynchronous support for resource requests only. The portlet container must provide asynchronous support for resource requests.

Portlet asynchronous support is modeled after the asynchronous support introduced with Servlet Specification 3.0[[17]](#footnote-17). Many of the asynchronous support considerations, concepts , and restrictions described in the Servlet Specification apply to portlets as well.

## Concepts

When a resource request is received from the client, the portlet container will dispatch it to the target portlet as described in Chapter 4 Portlet Lifecycle Interfaces, and specifically in Section 4.8.6 Annotated Resource Method Dispatching on page 41. If the portlet does not use ResourceRequest methods to start asynchronous processing, portlet resource phase execution will take place as described there.

### Starting Asynchronous Processing

If the portlet uses asynchronous processing, it must start asynchronous processing within the body of the resource method invoked by the portlet container. If the portlet performs a PortletRequestDispatcher include or forward before starting asynchronous processing, behavior of the ServletRequest asynchronous methods within the method targeted by the include or forward is implementation-specific and outside the scope of this specification.

Once asynchronous processing has been started, the portlet container must not close the ResourceResponse until asynchronous processing has been completed. Upon asynchronous processing completion, the portlet container must close the ResourceResponse and call any registered listeners (as will be covered in the following sections).

After the portlet starts asynchronous processing, the portlet may complete asynchronous processing before exiting the resource method, may dispatch the request and exit, or may pass the request to an asynchronous thread and exit without completing asynchronous processing or dispatching.

### The Asynchronous Processing Cycle

Within the overall asynchronous processing bracket, that is, after the portlet has started asynchronous processing, but before asynchronous processing completion, there can be one or more asynchronous processing cycles.

The portlet can use PortletAsyncContext dispatch methods to end an asynchronous processing cycle without completing the overall asynchronous processing bracket. The portlet can use asynchronous dispatch methods to dispatch the request back to the same resource method that initialized asynchronous processing, to a servlet that supports asynchronous processing, or to a servlet or JSP that does not support asynchronous processing.

Dispatching transfers the request to the portlet container, which will then invoke the target servlet or portlet.

If the dispatch target is the portlet resource method itself, or is a servlet that supports asynchronous processing, that target method may use asynchronous methods to either start a new asynchronous processing cycle or to complete the overall asynchronous processing bracket. If the dispatch target method exits without starting a new asynchronous processing cycle and without completing asynchronous processing, the portlet container must complete the overall asynchronous processing bracket.

If the dispatch target does not support asynchronous processing, the portlet container must complete the overall asynchronous processing bracket after the target dispatch method exits.

If the portlet resource method performs a PortletRequestDispatcher forward after starting asynchronous processing but before performing an asynchronous dispatch, the portlet container must complete the overall asynchronous processing bracket after the target forward method exits.

Within the same container-dispatched method, performing a PortletRequestDispatcher forward after an asynchronous dispatch method has been executed is disallowed, as is an asynchronous dispatch after a PortletRequestDispatcher forward.

### The Asynchronous Dispatch Operation

The portlet container must make the following portlet artifacts available as request attributes to the asynchronous dispatch target in order to allow the portlet tag library to be used in a JSP targeted by the asynchronous dispatch operation:

| **Request Attribute Name** | **Type** |
| --- | --- |
| javax.portlet.config | javax.portlet.PortletConfig |
| javax.portlet.request | javax.portlet.ResourceRequest |
| javax.portlet.response | javax.portlet.ResourceResponse |

### Portlet Request Dispatcher and Asynchronous Processing

The portlet can obtain and use a portlet request dispatcher during asynchronous processing subject to the restrictions specified in Section 21.1.2 The Asynchronous Processing Cycle.

The portlet container must ensure that portlet request dispatcher includes and forwards work as specified in Chapter 25 Dispatching to JSPs and Servlets on page 156. When dispatching to a JSP, portlet artifacts must be available through the Expression Language as described in Section 20.3 Portlet Predefined Beans on page 121 and portlet tags must work as described in Chapter 26 Portlet Tag Library on page 172.

### The Asynchronous Thread

After starting asynchronous processing, the portlet may start an asynchronous thread without dispatching or completing the request before returning to the container.

It can do so by instantiating a portlet class implementing the Runnable interface and passing the object to the PortletAsyncContext interface start method. If the portlet runs an asynchronous thread in this manner, the portlet container should make contextual information available to the thread as described in Chapter 20 Managed Bean Support on page 119.

Since the asynchronous thread is logically executing the same request as the initiating resource method, the contextual objects available in the thread should be the same instances as those available to the originating method.

After the portlet calls the PortletAsyncContext interface start method, the portlet container should wait until the resource method exits before commencing thread execution.

As an alternative to using the start method, the portlet may run the asynchronous thread using other means, such as through use of a managed thread pool. In this case, availability of contextual information on the thread would be platform specific and outside the scope of this document.

### The Asynchronous Listener

The portlet asynchronous context provides methods for adding listeners that will be informed when an error occurs during processing, when the asynchronous request times out, when a new asynchronous processing cycle is started, and when asynchronous processing completes. The listener must implement the PortletAsyncListener interface. Listeners must be added within the container-dispatched method that starts the asynchronous processing cycle.

Multiple listeners can be added. If more than one listener is present, they will be executed in the order in which they were added.

If the asynchronous listener is instantiated through the PortletAsyncContext interface createListener method, the portlet container should make contextual information available to the thread as described in Chapter 20 Managed Bean Support on page 119.

Since the asynchronous listener is logically executing the same request as the initiating resource method, the contextual objects available in the thread should be the same instances as those available to the originating method.

During timeout or error processing, the listener may generate output, may perform an asynchronous dispatch operation, or may complete the request, providing that no earlier listener in the chain has already dispatched or completed the request.

If a timeout or error condition occurs and no PortletAsyncListener dispatches or completes the request, the portlet container must complete the request on behalf of the portlet, and may generate an implementation-specific error message.

When, after the initial asynchronous processing cycle, a new asynchronous processing cycle is started, it begins with no listeners registered for execution. All listeners that had been added to the old cycle will be informed that a new asynchronous cycle is starting. The listener can use this event to add itself for event notification during the new asynchronous processing cycle.

Registered listeners will be notified when asynchronous processing completes. When this occurs, the listener may not produce output, dispatch the request, or complete asynchronous processing. However, the listener may release any resources allocated for the request by the portlet.

### Portlet Filters and Asynchronous Processing

Portlet resource filters will be invoked before the resource method is executed during the initial request and also during resource requests resulting from an asynchronous dispatch.

The portlet resource filter can use the dispatcher type to determine whether the initial resource request or a request resulting from an asynchronous dispatch is being executed. The dispatcher type will be set to javax.servlet.DispatcherType.REQUEST for the initial request and to javax.servlet.DispatcherType.ASYNC for resource requests resulting from an asynchronous dispatch.

During outbound processing, the portlet resource filter can use the ResourceRequest interface isAsyncStarted() method to determine whether asynchronous processing has been started. If asynchronous processing has been started, the portlet resource filter may not produce output.

In general, if a portlet resource filter allocates resources during initial request inbound processing, it should not release those resources during outbound processing if asynchronous processing has been started. An asynchronous listener can be used to release the resources if it is expected that the request output will be generated by an asynchronous thread.

### Multiple Resource Methods

The portlet programming model allows multiple resource methods to be executed as described in Section 4.8.6 Annotated Resource Method Dispatching on page 41. However, if several resource methods are defined to be executed in sequence, and one of them starts asynchronous processing, any remaining resource methods will not be executed in order to avoid a potential conflict between output produced by the remaining resource methods and that produced asynchronously.

An asynchronous dispatch operation back to the resource method will cause execution of only the single resource method that initiated asynchronous processing.

### Scenarios

This section provides diagrams illustrating typical usage scenarios from the point of view of the portlet programmer. The diagrams provided for illustrative purposes only, and are non-normative. They do not imply or prescribe a specific container implementation.

For simplicity, the portlet container is shown as directly receiving resource requests from the client, although in reality this will often not be the case.

As shown in Figure 21–1, the client issues a resource request to the portlet container, which invokes the resource method for the portlet. The resource method starts asynchronous processing, adds an asynchronous listener, instantiates a Runnable, starts the asynchronous thread, and returns.

Figure 21–1 Asynchronous thread includes / forwards to JSP

The asynchronous thread performs its long-running work and uses a portlet request dispatcher to include or forward to a JSP to produce output, completes asynchronous processing, and exits. The portlet container fires an onComplete event to the asynchronous listener and returns the resource response to the client.

Figure 21–2 shows a similar scenario but rather than having the asynchronous thread produce output using a portlet request dispatcher, the asynchronous thread instead calls AsyncContext.dispatch() in order to dispatch back to the resource method.

Figure 21–2 Asynchronous dispatch to resource method

The portlet container performs the asynchronous dispatch by invoking the resource method that initiated asynchronous processing with the dispatcher type set to DispatcherType.ASYNC. The resource method generates output in the appropriate manner, for example, by invoking a JSP through use of a portlet request dispatcher. After output has been generated, the resource method completes asynchronous processing.

The portlet container fires an onComplete event to the asynchronous listener and returns the resource response to the client.

In the timeout scenario shown below, the client issues a resource request to the portlet container, which invokes the resource method for the portlet. The resource method starts asynchronous processing, adds an asynchronous listener, instantiates a Runnable, starts the asynchronous thread, and returns.

Figure 21–3 Timeout during asynchronous processing

The asynchronous thread performs its long-running work but does not finish in the allocated time. It does not complete asynchronous processing or perform an asynchronous dispatch operation.

The portlet container recognizes the timeout situation and fires an onTimeout event to the asynchronous listener.

The asynchronous listener onTimeout method performs an asynchronous dispatch to a JSP, possibly to generate an error message and returns to the container.

The portlet container invokes the JSP using an asynchronous dispatch. Since the JSP does not support asynchronous operation, the portlet container completes asynchronous processing after the JSP invocation.

The portlet container fires an onComplete event to the asynchronous listener and returns the resource response to the client.

## Asynchronous Programming API

The portlet ResourceRequest interface contains methods for asynchronous support that correspond to similarly-named asynchronous support methods defined by the ServletRequest interface. Portlet asynchronous support defines the PortletAsyncContext, PortletAsyncEvent, and PortletAsyncListener classes that correspond to the servlet API AsyncContext, AsyncEvent, and AsyncListener classes, respectively.

### Resource Request Methods

| **Method** | **Description** |
| --- | --- |
| PortletAsyncContext getPortletAsyncContext() | Gets the PortletAsyncContext that was created or reinitialized by the most recent invocation of a startPortletAsync method on this request. |
| DispatcherType getDispatcherType() | Gets the dispatcher type of this request. |
| boolean isAsyncStarted() | Checks if this request has been put into asynchronous mode. |
| boolean isAsyncSupported() | Checks if this request supports asynchronous operation. |
| PortletAsyncContext startPortletAsync() | Puts this request into asynchronous mode and initializes the PortletAsyncContext object. |
| PortletAsyncContext startPortletAsync(ResourceRequest, ResourceResponse) | Puts this request into asynchronous mode and initializes the PortletAsyncContext object using the given resource request and response objects. |

### Portlet Asynchronous Context Methods

The PortletAsyncContext interface provides methods for control of a running asynchronous processing cycle.

| **Method** | **Description** |
| --- | --- |
| void addListener(PortletAsyncListener) | Registers the given PortletAsyncListener with the most recent asynchronous cycle that was started by a call to one of the ResourceRequest.startAsync() methods. |
| void addListener(PortletAsyncListener, ResourceRequest, ResourceResponse) | Registers the given PortletAsyncListener with the most recent asynchronous cycle that was started by a call to one of the ResourceRequest.startAsync() methods, saving the provided request and response objects for delivery in an asynchronous event. |
| void complete() | Completes the portlet asynchronous operation and closes the response associated with this PortletAsyncContext object. |
| <T extends PortletAsyncListener> T createPortletAsyncListener(Class<T>) | Instantiates the given PortletAsyncListener class. |
| void dispatch() | Dispatches the request and response objects of this PortletAsyncContext to the portlet container. |
| void dispatch(String) | Dispatches the request and response objects of this PortletAsyncContext to the given path within the PortletContext. |
| ResourceRequest getResourceRequest() | Gets the request that was used to initialize this PortletAsyncContext. |
| ResourceResponse getResourceResponse() | Gets the response that was used to initialize this PortletAsyncContext. |
| long getTimeout() | Gets the timeout (in milliseconds) for this PortletAsyncContext. |
| boolean hasOriginalRequestAndResponse() | Checks if this PortletAsyncContext was initialized with the original or application-wrapped request and response objects. |
| void setTimeout(long time) | Sets the timeout (in milliseconds) for this PortletAsyncContext. |
| void start(Runnable run) | Causes the container to dispatch a thread, possibly from a managed thread pool, to run the specified Runnable. |

### Portlet Asynchronous Listener Methods

The PortletAsyncListener interface provides methods that can be implemented by a portlet class for registration as an asynchronous listener.

| **Method** | **Description** |
| --- | --- |
| void onComplete(PortletAsyncEvent) | Notifies this PortletAsyncListener that an asynchronous operation has been completed. |
| void onError(PortletAsyncEvent) | Notifies this PortletAsyncListener that an asynchronous operation has failed to complete. |
| void onStartAsync(PortletAsyncEvent) | Notifies this PortletAsyncListener that a new asynchronous cycle is being initiated via a call to one of the ResourceRequest#startAsync methods. |
| void onTimeout(PortletAsyncEvent) | Notifies this PortletAsyncListener that an asynchronous operation has timed out. |

### Portlet Asynchronous Event Methods

The PortletAsyncEvent class interface represents the event that is delivered to the listener when an asynchronous event occurs.

| **Method** | **Description** |
| --- | --- |
| PortletAsyncContext getPortletAsyncContext() | Gets the portlet asynchronous context object associated with the event. |
| ResourceRequest getSuppliedRequest() | Gets the resource request associated with the listener. Returns null if no request was provided to the addListener method. |
| ResourceResponse getSuppliedResponse() | Gets the resource response associated with the listener. Returns null if no request was provided to the addListener method. |
| Throwable getThrowable() | Gets the Throwable associated with the event. |

# Client-Side Support

Portlet Specification 2.0 introduced support for Ajax requests through resource serving. The resource requests are initiated through resource URLs that are tied to a specific render state. Portlet Specification 3.0 goes beyond this by allowing portlet JavaScript code running on the client to update the render state and obtain resource URLs corresponding to the resulting page state. Portlet Specification 3.0 improves client-side support for portlets by introducing a dedicated client-side JavaScript API for portlet support.

Figure 22–1 The Portlet Hub

The client-side portlet API is known as the portlet hub. The portlet hub represents the portal on the client. Portlet Specification 3.0 defines the API and its semantics, but does not define its implementation details. The portlet hub implementation along with any necessary communication protocols between the portlet hub and the portal server are implementation-specific and are not covered by this specification.

The portlet hub is a JavaScript module provided by the portal implementation that manages the render state for all portlets on the page. The manner in which the portlet hub is configured, packaged, and made available on the portal page is implementation specific.

A portlet can provide JavaScript code known as the portlet client that registers itself with the portlet hub. Once registered, the portlet hub informs the portlet client through a callback function whenever the render state for that portlet changes.

The portlet client can use portlet hub methods to set the render state, to execute portlet actions, and to obtain resource URLs that correspond to the current page state without causing a page refresh. The portlet hub thus allows the portal to serve pages containing many portlets that change their state and communicate with one another using portlet concepts such as portlet actions, public render parameters, and portlet events, while maintaining an overall single-page-application-like experience for the overall portal page.

The portlet hub JavaScript API documentation is provided as part of the portlet API documentation.

## Basic Concepts

The JavaScript API is designed using standard JavaScript constructs so as to place as few restrictions as possible on the vendor implementing the portlet hub. Similarly, it places as few restrictions as possible on the portlet client code provided by the portlet. The portlet hub and the portlet client are bound only by the portlet hub API definition. Either or both can be implemented in plain JavaScript or with the help of a JavaScript library.

The JavaScript namespace beginning with ‘portlet’ is reserved for use by the portlet hub. The portlet hub makes its register method available under the name ‘portlet.register’. Beyond reservation of this name, the portlet hub imposes no further namespace restrictions.

The Portlet Specification places no restrictions on portlet client packaging. The portlet client JavaScript code can be rendered inline in the portlet markup or it can be packaged in a separate file. The Portlet Specification places no restrictions on how the portlet client JavaScript code is structured.

The Portlet Specification describes client-side support for version 3.0 portlets only. Such portlets have a version 3.0 portlet descriptor or use the @PortletConfiguration configuration annotation defined in Portlet Specification Version 3.0. Support for version 2.0 portlets is left to the portal and portlet hub implementation.

The Portlet Specification does not describe how portlets not participating in the portlet hub client-side support are to be handled. However, the Portlet Specification recommends that the portlet hub initiates a page refresh if a portlet hub interaction results in a render state change for a portlet that does not participate in portlet hub client-side support. Otherwise, the non-participating portlet might display stale data.

### Configuration

The manner in which the portlet container places the portlet hub on the page is implementation specific. It may place the portlet hub on all portal pages, or only on a subset of pages.

If a portlet is configured with a dependency on the portlet hub, the portlet container should place the portlet hub on all pages containing that portlet.

A portlet can configure a dependency on the portlet hub either statically as described in Section 28.2.8 Page Resource Dependencies on page 199, or dynamically as described in Section 15.8.1 Page Resources and Dependencies on page 97.

The resource identifier designating the portlet hub is as follows.

<name>PortletHub</name>

<scope>javax.portlet</scope>

<version>3.0.0</version>

### Promises

The portlet hub makes use of promises to help with asynchronous operations. Promises are part of the ECMA Script standard[[18]](#footnote-18) and are supported by many browsers. However, some browsers do not support promises. The portal must provide a polyfill if such browsers are to be supported. The ECMA Script standard Promise support is distinct from Promise support provided by some JavaScript libraries. The portlet hub must provide Promise support that is compatible with the ECMA Script standard.

When the portlet client calls a portlet hub function and an error condition occurs immediately, the function can throw an exception to indicate the problem.

When the portlet client calls a portlet hub function that can potentially require communication with the portal server to complete, the portlet hub returns a Promise. The portlet hub implementation is not required to communicate with the server in such situations, but can do so if necessary. When the operation completes, the portlet hub can resolve the pending Promise in one of two ways:

1. It can *fulfill* the Promise if the operation was successful, passing the requested information back to the portlet client.
2. It can *reject* the Promise if the operation was not successful. In this case, the portlet hub must pass an Error object back to the portlet client. The Error object should contain an indication about the error that occurred.

In the description that follows, the reject path might be left out for clarity when the portlet hub returns a Promise. In such cases, the reader should note that a returned Promise can always be rejected.

### Changing the Render State

The portlet client can use portlet hub functions to initiate render state changes. The portlet client can set public and private render parameters as well as the portlet mode and window state.

In addition, the portlet client can submit a portlet action request that uses HTTP POST semantics. The portal will execute the portlet Action Phase and Event Phase processing on the server and return the updated page state to the portlet hub.

After the requested state change has been performed, the portlet hub will usually provide each affected portlet client with its updated state information.

However, regardless of whether the state change was initiated by setting parameters or through a portlet action, the portal may respond by completely refreshing the page. The portal may do so in order to support portlets that are affected by the state change but do not participate in the portlet client-side support, due to configuration settings, or for implementation-specific reasons.

If the portal responds to a state change request by refreshing the page, the portlets will not be updated with new page state information before the page refresh is carried out.

### Receiving Render State Updates

When a state change occurs that affects a portlet, the portlet hub informs the affected portlet client of its new state through use of a callback function. The change causing the update does not necessarily need to be initiated by the portlet client itself.

For example, when portlet A changes a public render parameter used by portlet B, the portlet hub will inform both the portlet A and the portlet B clients of that change.

### Portlet Client Events

Portlet client events consist of an event type and an event payload. Both are defined by the portlet clients themselves.

Portlet client events have no connection to the server-side portlet event mechanism.

The portlet hub provides utility functions that enable the portlet client to dispatch and listen for portlet client events.

### Error Handling

When the portlet hub can recognize an error during function execution, the error will be reported to the portlet client through an exception.

However, some methods initiate work that is performed asynchronously. Errors that occur during asynchronous processing will be reported to the portlet client by rejecting a Promise if possible. If the problem cannot be reported to the portlet client by rejecting a promise, the portlet hub will inform the portlet client of the error condition through the onError callback function.

### Important Considerations

The portlet hub calls the portlet client callback functions in several situations as described above. When the portlet hub calls the portlet client, the portlet client may navigate to a different page or may initiate another change to the render state that could potentially cause a page refresh.

Due to this behavior, the delivery of neither render state updates nor portlet client events can be guaranteed.

### Blocking Operations

The portlet hub implementation may optionally provide support for blocking operations.

The portlet hub provides for orderly state transitions by allowing only a single blocking operation (action, setRenderState, startPartialAction) to be active at any one time.

The state transition is considered to be active from the initial portlet client call to one of the blocking operations until the portlet hub has performed the requested state change and has informed all of the affected portlet clients by firing the corresponding onStateChange events.

This has the following implications:

* It is not possible to initiate a sequence of blocking operations.
* For example, once a portlet client calls the setRenderState method, it cannot call any additional blocking method until after its onStateChange listener function has been called.
* It is not possible to initiate a blocking operation during execution of the onStateChange listener function, since execution of that function belongs to the preceding state change operation.

The portlet client may use the isInProgress method to determine if a blocking operation is in progress. If blocking operations are not supported by the portlet hub implementation, this method must always return false.

## Portlet Parameters and Render State

The render state concept and the Java API for accessing the render state was introduced in Chapter 12 Render State on page 67. This section introduces the portlet hub JavaScript API for handling render state.

The idea of render state is central to the portlet hub concept. The portlet hub manages the overall page state on the client. The page state consists of the render state of all portlets on the page. The portlet hub informs the portlet client when the render state changes. This could occur as the result of a server-side action, for example. The portlet client can also make changes such as setting render parameters through the portlet hub.

The render state consists of the private and public render parameters, the portlet mode, and the window state. The render state is modeled by JavaScript objects that contain the necessary information along with functions that allow the render state to be read and updated.

The PortletParameters object is a JavaScript object whose property names correspond to parameter names and whose property values are string arrays. A property value may be null or may be an empty array. An example property object may appear as follows:

{

'parm1' : ['val1'],

'parm2' : ['val2', 'val3']

}

The portlet client uses property objects to pass action and resource parameters to portlet hub functions. A portlet parameters object also appears within the render state object to represent the render parameters.

The RenderState object contains a parameters property representing the render parameters, a portletMode property representing the portlet mode, and a windowState property representing the window state. In addition, it contains functions for accessing theses data items. An example for the data portion of the render state object might appear as follows:

{

'parameters' : {

'parm1' : ['val1'],

'parm2' : ['val2', 'val3']

},

'portletMode' : 'VIEW',

'windowState' : 'NORMAL'

}

The table below describes the functions available on the render state object.

| **Function** | **Description** |
| --- | --- |
| clone() | Returns a new copy of this object |
| setPortletMode(pm) | Sets the portlet mode to the specified value. The strings defined by the PortletConstants object should be used to specify the portlet mode. |
| getPortletMode() | Returns the current portlet mode |
| setWindowState(ws) | Sets the window state to the specified value |
| getWindowState() | Returns the current window state The strings defined by the PortletConstants object should be used to specify the window state. |
| setValue(n, v) | Sets a parameter with name n and value v. The value v may be a string or an array. |
| setValues(n, v) | Sets a parameter with name n and value v. The value v may be a string or an array. |
| getValue(n, d) | Gets the string parameter value for the name n. If n designates a multi-valued parameter, this function returns the first value in the values array. If parameter n is undefined, the function returns the optional default value d. |
| getValues(n, d) | Gets the string array parameter value for the name n. If parameter n is undefined, the function returns the optional default value array d. |
| remove(n) | Removes the parameter with name n. |

## Scenarios

The next sections illustrate the concepts behind the portlet hub through use of sequence diagrams. In the sequence diagrams in the sections that follow, communication or method calls not covered by the Portlet Specification is represented by dashed red lines (). Communication or method calls covered by the specification are represented by solid blue lines (). Illustrative text showing potential or optional communication between the portlet hub and the portal server is shown in parentheses (like this).

For details on the portlet hub methods, see Section 22.4 Portlet Hub API.

### Initial Page Load

When the page is initially loaded, the portlet hub and the portlet clients must be loaded and initialized. The following figure illustrates the page loading and initialization sequence.

Initialization is the same for each portlet client. The sequence is explained in the figure below using portlet B as an example.

Figure 22–2 Initial Page Load

1. The sequence begins when the user agent issues a request for a portal page that makes use of the portlet hub.
2. The portal causes the portlets on the page to be rendered and returns the page data containing the page markup, the portlet hub code, and the portlet client code for each portlet back to the user agent.
3. The portlet hub and the portlet client are initialized and begin execution. The method through which the initialization takes place is outside the scope of this specification. However, the portlet hub must ensure that its portlet.register function is available in the JavaScript global namespace before the portlet clients are initialized.
4. Each portlet client registers itself with the portlet hub by calling the portlet.register function and passing its portlet ID as an argument. The portlet ID is the unique namespace string returned by the PortletResponse interface getNamespace method.
5. The portlet hub must return a Promise object in order to allow potential communication with the portal.
6. The portlet hub fulfills the Promise by passing a PortletInit object to the portlet client.

The PortletInit object represents the actual portlet hub programming interface for use by the portlet. It contains all portlet hub methods that can be used after initialization.

The portlet hub must associate the portlet ID provided to the register function with the specific PortletInit object returned to the portlet client. This is done for convenience, so that the portlet client developer does not have to provide the portlet ID with each portlet hub function call. The portlet client should maintain a reference to the returned PortletInit object for future use.

1. The portlet client uses the PortletInit object addEventListener method to add a listener with the type containing the string ‘portlet.onStateChange’ and providing the appropriate callback function. The portlet hub must invoke the onStateChange callback function whenever the render state for the registered portlet changes.
2. The portlet client may additionally use the PortletInit object addEventListener method to add a listener with the type containing the string ‘portlet.onError’ and providing the appropriate callback function. The portlet hub must invoke the onError callback function to report an error that cannot be reported by an exception or by rejecting a promise.
3. After the portlet client adds an onStateChange listener, the portlet hub must fire an onStateChange event to that portlet, passing the current RenderState object as the event payload in order to provide the portlet with its initial render state information.

The following example shows portlet client initialization code:

// Handler for onStateChange event

update = function (type, state) {

// The type argument will be set to the string 'portlet.onStateChange'

// The state argument will be a RenderState object

};

// Register portlet with Portlet Hub.

// Add listener for onStateChange event.

portlet.register(pid).then(function (pi) {

hub = pi; // Store the PortletInit object

hub.addEventListener("portlet.onStateChange", update);

});

### Single Portlet Update

In this scenario, a single portlet updates its private render parameters and obtains a resource corresponding to the new page state in order to update its user interface.

Figure 22–3 Single Portlet Update

1. The scenario begins when a user interaction with portlet B requires a render state update, such as an update to private parameters that does not affect other portlets on the page.
2. The portlet updates its RenderState object and calls the portlet hub setRenderState method.
3. This can potentially cause a data exchange between the portlet hub and the portal.
4. The portlet hub accepts the update and fires an onStateChange event to the portlet in confirmation of the state change.
5. The portlet needs a new resource based on the new state, so it invokes the createResourceURL method with any necessary resource parameters and cacheability options.
6. The portlet hub returns a Promise for the resource URL.
7. This can potentially cause a data exchange between the portlet hub and the portal.
8. The portlet hub fulfills the promise by passing the resource URL containing the current page state as required to the portlet client.
9. The portlet client uses a native XHR or its JavaScript library function of choice to fire a request to the URL.
10. The portlet container executes resource request processing for portlet B using the page state and resource parameters provided through the resource URL.
11. The portal returns the resource response to the portlet client.

### Public Render Parameter Update

In this scenario, portlet B updates a public render parameter that is also used by portlet A.

Figure 22–4 Public Render Parameter Update

1. The scenario begins when a user interaction with portlet B requires a render state update that affects a public render parameter also used by portlet A.
2. Portlet B updates its RenderState object and calls the portlet hub setRenderState method.
3. This can potentially cause a data exchange between the portlet hub and the portal.
4. The portlet hub accepts the update and fires an onStateChange event to portlet B in confirmation of the state change.
5. The portlet hub fires an onStateChange event to portlet A to inform that portlet about the state change.
6. During onStateChange execution, both affected portlets can use portlet hub methods to create resource URLs and can use JavaScript library functions to update the user interface.

### Portlet Action

In this scenario, portlet B executes a portlet hub action that causes an event processing sequence on the server, resulting in render state updates for portlets A and B.

Figure 22–5 Portlet Action

1. The scenario begins when a user interaction with portlet client B initiates a portlet action, such as a form submission, that causes a server-side event affecting portlet A.
2. Portlet client B invokes the portlet hub action method with any necessary action parameters and form parameter arguments.
3. This causes the portlet hub to send an action request to the portal.
4. The portlet container fires an action request for portlet B on the server.
5. Portlet B executes the action request and fires an event.
6. The portal routes the event to portlet A, which performs event processing.
7. Instead of rendering the page or redirecting the client to the new resulting render URL, the portal transmits the new page state containing updated information for portlets A and B to the portlet hub.
8. The portlet hub fulfills the action Promise.
9. The portlet hub fires an onStateChange event to portlet client B with the updated RenderState object.
10. The portlet hub fires an onStateChange event to portlet client A with the updated RenderState object.
11. During onStateChange execution, both affected portlets can use portlet hub methods to create resource URLs and can use JavaScript library functions to update the user interface.

### Partial Action

The partial action was introduced to provide client-side support for action-based frameworks such as JSF which need to be in control in control of submitting the request and processing the response in order to work correctly.

The partial action sequence allows the portlet (or portlet bridge) client-side code to obtain an action URL that the framework can use to execute the request and obtain a response. The partial action response consists of a page state token, which is an object that can contain any value, including the empty string or null.

The framework passes the page state token to the portlet hub in order to complete the partial action processing sequence. Based on the new page state, obtained from the page state token or alternatively through additional communication with the portal, the portlet hub fires onStateChange events to all affected portlet clients on the page.

The onStateChange event fired to the portlet client initiating the partial action sequence must contain a RenderData object as described in Section 22.4.12 on page 151.

In this scenario, portlet B executes a partial action that causes an event processing sequence on the server, resulting in render state updates for portlets A and B and markup updates for portlet B.

In the figure below, portlet client B is colored to indicate that it might be executing framework bridge code, such as the JSF Portlet Bridge, to work with the framework.

Figure 22–6 Partial Action

1. The scenario begins when a user interaction with portlet client B initiates an action that uses the framework, such as a form submission, that that causes a server-side event affecting portlet A.
2. Portlet client B invokes the portlet hub startPartialAction method with any necessary action parameters.
3. The portlet hub returns a Promise.
4. The portlet hub potentially communicates with the portal.
5. The portlet hub fulfills the Promise with a PartialActionInit object.

The PartialActionInit object contains a URL and a callback function. The URL is called the partial action URL, and when it is fired, it invokes the partial action processing sequence on the portal server. The callback function is called the setPageState function, and it takes a String page state token as the only argument.

Later in the sequence, the portlet client invokes the callback function, passing the page state token provided by the portlet container as an argument, to conclude the partial action processing sequence.

1. Portlet client B uses the partial action URL with the framework to invoke partial action processing.
2. The portlet container issues an action request for portlet B on the server.
3. Portlet B executes the action request and fires an event.
4. The portal routes the event to portlet A, which performs event processing.
5. The portal invokes resource request processing for portlet B in order to generate the RenderData object to be provided to portlet client B during Step 16 below.

Note that the portal implementation is free to invoke the resource method at this point in order to include the resulting markup in the page state token, or at a later time through an implementation-specific mechanism.

1. Instead of rendering the page or redirecting the client to the new resulting render URL, the portal stops processing and returns the resulting page state token to the client.
2. The portal sends the partial action response, which contains the page state token, back to the framework.
3. The framework concludes its work and passes the page state token to portlet B.
4. Portlet B calls the setPageState function, passing the page state token to the portlet hub.
5. The portlet hub updates its internal state, which could potentially require communication with the portal.
6. The portlet hub fires an onStateChange event to portlet client B with the updated RenderState object and with a RenderData object containing the data produced by the resource method.
7. The portlet hub fires an onStateChange event to portlet client A with the updated RenderState object.
8. During onStateChange execution, both affected portlets can use portlet hub methods to create resource URLs and can use JavaScript library functions to update the user interface as necessary.

## Portlet Hub API

The portlet hub API consists of the portlet.register method along with the methods provided by the PortletInit object that are summarized in the following table. This section describes the API method behavior in detail.

| **Property** | **Returns** | **Description** |
| --- | --- | --- |
| action | Promise | Initiates a portlet action. |
| addEventListener | object | Adds a listener function for specified event type. |
| createResourceUrl | Promise | Returns a promise for a resource URL. |
| dispatchClientEvent | number | Dispatches a client event. |
| isInProgress | boolean | Tests whether a blocking operation is in progress. |
| newParameters | PortletParameters | Returns new PortletParameters object. |
| newState | RenderState | Returns a new RenderState object. |
| removeEventListener | n/a | Removes a previously added listener function. |
| setRenderState | n/a | Sets the render state. The argument must be a RenderState object. |
| startPartialAction | Promise | Starts partial action processing. |

### Portlet.register()

This method registers a portlet client with the portlet hub and returns a Promise object that is fulfilled with the PortletInit object for the portlet as argument when the registration has been completed.

The portlet client calling this method must provide a valid portlet ID. The portlet ID is identical to the unique namespace provided by the portal server for the portlet window. The portlet hub should validate the portlet ID and throw an exception or reject the Promise if the portlet ID is invalid.

The portlet hub implementation should allow multiple calls to the register method for a given portlet ID in order to provide support for independent JavaScript components.

### action(actParams, element)

This method initiates a portlet action using the specified action parameters and element arguments and returns a Promise that is fulfilled with no argument when the action completes. The action operation causes the action phase to be executed on the server for the initiating portlet and the event phase to be executed for all portlets affected by events fired during the action phase.

When the action has successfully completed, the portlet hub must provide the updated RenderState object to the portlet client through the onStateChange listener function. The portlet hub must also call the onStateChange callback function for every portlet client whose state has changed as result of this operation.

However, the portlet hub or portal may also completely refresh the page as a response to the action. This may occur in order to support portlets on the page that do not participate in the Portlet 3.0 Ajax support or due to configuration settings, for example. If the page is completely refreshed, it will be rendered according to render parameters set on the server.

The actParams argument represents action parameters, which are optional parameters attached to an action URL in addition to any render state values that may be present. Action parameters do not influence the render state. The portlet hub and portal must make the action parameters set through the portlet hub action method available on the server through the ActionParameters interface. See Section 11.3 Action Parameters.

The actParams argument must be a Parameters object containing properties representing parameter names whose values must be an array of string values, as described in Section 22.2 Portlet Parameters and Render State. Use of action parameters is optional. If the actParams argument is provided, but is not a valid Parameters object, the portlet hub must throw an exception.

If the optional element argument is present, it must refer to an HTML form to be submitted. The portlet hub must use this form to execute the action. The action attribute of the FORM tag is ignored and need not contain an action URL.

If the element argument is provided, but is not a valid HTML form element, the portlet hub must throw an exception.

If the form element is specified, the encoding type must be 'application/x-www-form-urlencoded' or 'multipart/form-data'. The encoding type 'text/plain' is not supported.

If the encoding type is 'multipart/form-data', the submission method must be 'POST'. Form 'INPUT' elements of type 'FILE' are supported.

If the encoding type is 'application/x-www-form-urlencoded', the submission method can be either 'GET' or 'POST'. However, form 'INPUT' elements of type 'FILE' are not supported.

If the element argument is not specified, the portlet hub will submit the action to the server by executing a 'POST' with an action URL containing any action parameters provided.

The parameters may be specified in either order, individually, or not at all. Examples of valid calls:

action();

action(actParams, element);

action(actParams);

action(element);

A portlet action is a blocking operation. If the portlet hub implementation supports blocking operations, it must throw an exception if this method called while a blocking operation is in progress.

If this method is called before the portlet client has registered an onStateChange listener, this method must throw an exception.

### addEventListener(type, func)

This method adds a listener function for specified event type and returns a handle representing the listener. The handle is needed in order to remove the event listener. The portlet client must specify the event type and the callback function that the portlet hub calls when the event occurs.

The portlet hub defines two classes of events - system events and portlet client events:

System events are generated by the portlet hub. They are used to pass portlet-specific information to the registered portlet client. The parameters passed to the system event callback functions are defined by the portlet hub.

Event types prefixed with "portlet." are reserved for system events. System event types may not be specified with a regular expression or wildcard. However, the same event listener may be added for all types of system events.

The following system event types are defined:

* portlet.onStateChange

Fired when the render state changes. In order to receive render state updates, a portlet client must register an onStateChange event listener for this event type.

After the portlet client adds an event listener for the onStateChange event, the portlet hub will call the onStateChange callback function to provide the portlet client with its initial state information. However, this will not occur before the call to addEventListener returns.

The onStateChange callback function is described in Section 22.4.12.

* portlet.onError

Fired when an error occurs that cannot be communicated through an exception. In general, this will be some type of asynchronous communication error. In order to receive notification about errors, a portlet must register an onError event listener for this event type.

Portlet Client Events are events initiated by the portlet client through the dispatchClientEvent method.

When adding a listener for a portlet client event, the event type may be specified by a regular expression string. The listener will be called for every event type that the regular expression string matches.

Example:

myHub.addEventListener("^myCompany\..\*", myListener); // registers myListener for all event types beginning with "myCompany."

An event listener can be added for multiple event types. This function returns a handle to identify the unique listener for the event type and for the portlet client associated with the function.

### createResourceUrl(resParams, cache, resid)

This method returns a Promise for a resource URL with parameters set appropriately for the page state according to the resource parameters and cacheability option provided.

The portlet hub must resolve the promise by providing a URL string that the portlet client may use a native XMLHttpRequest or with any appropriate JavaScript framework to retrieve content from the portlet through the server-side serveResource method.

The resParams argument represents resource parameters, which are optional parameters attached to a resource URL in addition to any render state values that may be present. Resource parameters do not influence the render state. The portlet hub and portal must make any resource parameters provided to the createResourceUrl function available on the server through the ResourceParameters interface. See Section 11.4 Resource Parameters

The resParams argument must be a PortletParameters object containing properties representing parameter names whose values must be an array of string values, as described in Section 22.2 Portlet Parameters and Render State. Use of resource parameters is optional. If the resParams argument is provided, but is not a valid Parameters object, the portlet hub must throw an exception.

The cacheability option designates the degree to which the content to be served can be cached and influences the type of content that can be served. There are three possible values – FULL, PORTLET, and PAGE as described in Section 13.2 The Resource URL.

Specification of cacheability (the cache argument) is optional. If the cacheability is not specified, cacheability for the URL will be set to "PAGE".

The resource ID (the resid argument) is an additional identifying string that the portlet container can use to select the correct resource serving method to execute, or that the resource serving method can use to determine the information to be provided. The resource ID argument is optional.

The arguments must be provided in the defined order. However, if a preceding argument is not needed, it may be specified as null. Examples of valid calls:

createResourceUrl();

createResourceUrl(resParams, cache);

createResourceUrl(resParams, cache, resid);

createResourceUrl(resParams);

createResourceUrl(null, cache);

createResourceUrl(null, null, resid);

### dispatchClientEvent(type, payload)

This method dispatches a client event and returns the number of events currently queued for delivery. The portlet client must supply the event type and the payload. Client events of the specified type are queued for delivery to registered event listeners of that type.

The portlet hub must match the event type against the type strings associated with registered event listeners and dispatch an event for each matching listener. The number of matching listeners will be returned.

The event payload is defined by the portlet client. It must be present, but may be of any type or value.

The portlet client may not dispatch event types beginning with the reserved string "portlet.".

The client is responsible for preventing race conditions. For example, a race condition can occur if portlet A dispatches an event to portlet B, causing an event to portlet A, which dispatches again to portlet B, etc.

Event delivery cannot be guaranteed, and may vary according to the situation.

The portlet hub implementation can optionally provide support for the dispatchClientEvent method. If support is not provided, the method must perform no operation and must return 0.

### isInProgress()

This method tests whether a blocking operation is in progress and if so returns true. Otherwise the method must return false.

If the portlet hub implementation does not support blocking operations, this method must always return false.

The portlet client can use this function to test whether a state change is in progress before initiating a blocking operation.

The developer should note that if the portlet client uses this function to implement a waiting function, the render state may be updated due to an onStateChange event during the time that the portlet client waits. Also note that the portal may choose to refresh the page as a response to a blocking operation, in which case the waiting function would not complete.

### newParameters(p)

This method returns a new PortletParameters object. If no argument is provided, an empty PortletParameters object will be returned. If an existing PortletParameters object is provided as argument, a clone of the input object will be returned. If the argument is provided, but is not a valid Parameters object, the portlet hub must throw an exception.

### newState(s)

This method returns a new RenderState object. If no argument is provided, an empty RenderState object will be returned. If an existing RenderState object is provided as argument, a clone of the input object will be returned. If the argument is provided, but is not a valid RenderState object, the portlet hub must throw an exception.

### removeEventListener(handle)

This method removes a previously added listener function designated by the handle. The handle must be the same object previously returned by the addEventListener function.

### setRenderState(state)

This method sets the render state. The portlet client must provide a RenderState object containing the updated render state as an argument.

When the operation has successfully completed, the portlet hub must provide the updated RenderState object to the portlet client through the onStateChange listener function. The portlet hub must also call the onStateChange callback function for every portlet client whose state has changed as result of this operation.

However, the portlet hub or portal may also completely refresh the page as a response to the setRenderState operation. This may occur in order to support portlets on the page that do not participate in the Portlet 3.0 Ajax support or due to configuration settings, for example. If the page is completely refreshed, it will be rendered according to render parameters set on the server.

The state argument must be a RenderState object as described in Section 22.2 Portlet Parameters and Render State. If the argument is not a valid RenderState object, the portlet hub must throw an exception.

A portlet setRenderState invocation is a blocking operation. If the portlet hub implementation supports blocking operations, it must throw an exception if this method called while a blocking operation is in progress.

### startPartialAction(actParams)

This method starts a partial action processing sequence and returns a Promise to the portlet client. The Promise is fulfilled with a PartialActionInit object as argument.

The partial action processing sequence provides client-side support for action-oriented frameworks such as JSF. It allows the framework to obtain URL that it can use to execute an action while remaining in an Ajax paradigm. Such a URL is called a partial action URL.

The PartialActionInit object returned through the promise contains a partial action URL and a portlet hub setPageState callback function that the framework must call to complete the processing sequence. The partial action URL can be used to initiate an action request. The setPageState callback function allows the portlet client to complete the partial action operation by updating the state of all portlet clients on the page that are affected by action and event processing on the server resulting from the partial action.

When the framework code invokes the setPageState callback function, the portlet hub must provide the updated RenderState object to the portlet client through the onStateChange listener function. . The portlet hub must also call the onStateChange callback function for every portlet client whose state has changed as result of this operation.

However, the portlet hub or portal may also completely refresh the page as a response to the partial action. This may occur in order to support portlets on the page that do not participate in the Portlet 3.0 Ajax support or due to configuration settings, for example. If the page is completely refreshed, it will be rendered according to render parameters set on the server.

The actParams argument represents action parameters, which are optional parameters attached to an action URL in addition to any render state values that may be present. Action parameters do not influence the render state. The portlet hub and portal must make any action parameters provided to the portlet hub action method available on the server through the ActionParameters interface. See Section 11.3 Action Parameters.

The actParams argument must be a Parameters object containing properties representing parameter names whose values must be an array of string values, as described in Section 22.2 Portlet Parameters and Render State. Use of action parameters is optional. If the actParams argument is provided, but is not a valid Parameters object, the portlet hub must throw an exception.

A portlet startPartialAction invocation is a blocking operation. If the portlet hub implementation supports blocking operations, it must throw an exception if this method called while a blocking operation is in progress.

If this method is called before the portlet client has registered an onStateChange listener, this method must throw an exception.

### onStateChange(type, renderState, renderData)

This describes the onStateChange callback function that each portlet client participating in the portlet hub Ajax support must implement.

The Portlet Specification speaks of the ‘onStateChange event’, however, the event has different semantics than normally associated with an ‘event’. Usually, an event distributes the same payload to all event subscribers. The portlet hub onStateChange event works differently. The portlet hub must fire an onStateChange event to each subscribed portlet that experiences a state change. The payload must contain the updated RenderState for the specific target portlet and no other.

The portlet client registers an onStateChange callback by adding a listener for the portlet.onStateChange event type through the addEventListener function.

The onStateChange callback function has two mandatory arguments and a third optional argument.

The mandatory type argument specifies the event type. This will always be the string ‘portlet.onStateChange’.

The mandatory renderState argument is the updated RenderState object.

The portlet hub can provide a third renderData argument, which is mandatory for onStateChange events fired to the portlet client initiating a partial action processing sequence, but is otherwise optional. This argument contains data that was returned directly in the action response. When the RenderData object is available, it contains the same data as would be available through a portlet resource request using the current render state with no additional resource parameters and with the resource URL cacheability option set to "PAGE".

The RenderData object contains two properties. The mimeType property contains the content type describing the data as set by the portlet resource method, while the renderData property contains the actual data.

{

'content' : '<p>Some markup</p>',

'mimeType' : 'text/html'

}

Portal implementations are not required to support this mode of operation for onStateChange events resulting from setRenderState or action calls, so the portlet client should obtain the resource data through use of a resource URL if the render data is not available.

# Caching

Caching content helps improve the portal response time for users. It also helps reduce the load on servers.

The Portlet Specification defines an expiration-based caching mechanism. This caching mechanism is for individual portlets. Cached content must not be shared across different user clients displaying the same portlet for the private cache scope.

Portlet containers are not required to implement expiration caching. Portlet containers implementing this caching mechanism may disable it, partially or completely, at any time to free memory resources.

Caching configuration is described in Section 28.2.6 Cache Settings on page 198.

## Expiration Cache

Portlets making use of the expiration cache should declare the default expiration time (in seconds) of the expiration cache in the portlet configuration. The portlet container should treat portlets with no default duration defined in the configuration as always expired.

A portlet may programmatically alter the expiration time or caching scope by setting a property in the RenderResponse or ResourceResponse object using the EXPIRATION\_CACHE or CACHE\_SCOPE constant defined in the MimeResponse interface in forwarded or included servlets/JSPs. Using Java code, the CacheControl object is available via the MimeResponse for setting the expiration time or caching scope via the calls setExpirationTime or setScope methods.

The portlet should set the expiration time or caching scope before writing to the output stream as otherwise the portlet container may ignore the values.

If the expiration property is set to 0, the returned markup fragment should be treated as always expired. If the expiration cache property is set to –1, the cache does not expire. If during render request processing the expiration cache property is not set, the expiration time defined in the deployment descriptor should be used. If the caching scope is set to PRIVATE\_SCOPE the cached data must not be shared across users. If the caching scope is set to PUBLIC\_SCOPE the cached data may be shared across users. The private scope is the default scope if no scope is provided in the portlet configuration or via the RenderResponse or ResourceResponse.

If the content of a portlet is cached, the cache has not expired, and the portlet is not the target of an action or event, the request handling methods of the portlet should not be invoked as part of the client request. Instead, the portlet-container should use the data from the cache.

If the content of a portlet is cached and the portlet is target of request with an action-type semantic (e.g. action or event processing), the portlet container should discard the cache and invoke the corresponding request handling methods of the portlet.

## Validation Cache

Portlets may use validation-based caching as an extension of the expiration-based caching mechanism. Validation-based caching allows portlets to return a validation token together with the markup response and cache expiration time. The portlet can set the validation token on the RenderResponse or ResourceResponse via the ETAG property from within servlets or JSPs, or via the CacheControl interface setETag method from within the portlet. The expiration time can be set as described in the previous section. If no expiration time is set, the content should be viewed by the portlet container as expired.

After the content is expired, the portlet container should provide the portlet with the validation token (called ETag in HTTP) of the expired content during render or resource request processing. The portlet can access the validation token provided by the portlet container either through the RenderRequest or ResourceRequest ETAG property, or using the getETag method.

The portlet can use the validation token to determine validity of the cached content. If the content is still valid, the portlet should not render any output but instead either set the RenderResponse or ResourceResponse interface USE\_CACHED\_CONTENT property with a new expiration time, or set a new expiration time using the CacheControl interface setUseCachedContent method.

Example programmatically setting the cache information:

protected void doHeaders (RenderRequest request, RenderResponse response)

throws PortletException, java.io.IOException

{

…

if ( request.getETag() != null ) { // validation request

if ( markupIsStillValid(request.getETag()) ) {

// markup is still valid

response.getCacheControl().setExpirationTime(30);

response.getCacheControl().setUseCachedContent(true);

return;

}

}

// create new content with new validation tag

response.getCacheControl().setETag(someID);

response.getCacheControl().setExpirationTime(60);

PortletRequestDispatcher rd = getPortletContext().getPortletRequestDispatcher("jsp/view.jsp");

rd.include(request, response);

}

# Security

Portlet applications are created by application developers who license the application to people who deploy the application into a runtime environment. Application developers need to define how the security is to be set up for the deployed application.

## Introduction

A portlet application contains resources that can be accessed by many users. These resources often traverse unprotected, open networks such as the Internet. In such an environment, a substantial number of portlet applications will have security requirements.

The portlet container is responsible for informing portlets of the roles users are in when accessing them. User authentication is outside the scope of this specification. Portlet containers may use the authentication mechanisms provided by the underlying servlet container[[19]](#footnote-19) or other means to perform authentication.

## Roles

The Portlet Specification uses the same definition of roles as provided by the servlet specification[[20]](#footnote-20). Security role reference configuration is described in Section 28.2.7 Security Role Reference on page 198.

## Programmatic Security

Programmatic security is supported through the following PortletRequest interface methods:

* getRemoteUser
* isUserInRole
* getUserPrincipal

See Section 14.1.6 Security Attributes.

The getRemoteUser method returns the user name the client used for authentication. The isUserInRole method determines ifa remote user is in a specified security role. The getUserPrincipal method determines the principal name of the current user and returns a java.security.Principal object. These APIs allow portlets to make business logic decisions based on the information obtained.

The values returned by the getRemoteUser and getUserPrincipal methods are the same as those returned by the equivalent methods of the servlet response object.

The isUserInRole method expects a string parameter with the role-name. A security-role-ref element must be declared by the portlet in deployment descriptor with a role-name sub-element containing the role-name to be passed to the method. The security-role-ref element should contain a role-link sub-element whose value is the name of the application security role that the user may be mapped into. This mapping is specified in the web.xml deployment descriptor file. The container uses the mapping of security-role-ref to security-role when determining the return value of the call.

For example, to map the security role reference "FOO" to the security role with   
role-name "manager" the syntax would be:

<portlet-app>

...

<portlet>

...

<security-role-ref>

<role-name>FOO</role-name>

<role-link>manager</role-link>

</security-role-ref>

</portlet>

...

...

</portlet-app>

In this case, ifthe portlet called by a user belonging to the "manager" security role made the API call isUserInRole("FOO"), then the result would be true.

Ifthe security-role-ref element does not define a role-link element, the container must default to checking the role-name element argument against the list of security-role elements defined in the web.xml deployment descriptor of the portlet application. The isUserInRole method references the list to determine whether the caller is mapped to a security role. The developer must be aware that the use of this default mechanism may limit the flexibility in changing role-names in the application without having to recompile the portlet making the call.

## Propagation of Security Identity in EJBTM Calls

A security identity, or principal, must always be provided for use in a call to an enterprise bean.

The default mode in calls to EJBs from portlet applications should be for the security identity of a user, in the portlet container, to be propagated to the EJBTM container.

Portlet containers running as part of a Java EE platform are required to allow users that are not known to the portlet container to make calls to the EJBTM container. In these scenarios, the portlet application may specify a run-as element in the web.xml deployment descriptor. When it is specified, the container must propagate the security identity of the caller to the EJB layer in terms of the security role name defined in the run-as element. The security role name must be one of the security role names defined for the web.xml deployment descriptor. Alternatively, portlet application code may be the sole processor of the sign on into the EJBTM container.

# Dispatching to JSPs and Servlets

Portlets can delegate the execution of logic or creation of content to servlets and JSPs. This is useful for implementing the Model-View-Controller pattern where the portlet may act as controller and dispatch to different JSPs for rendering the views.

The PortletRequestDispatcher interface provides a mechanism to accomplish this dispatching.

Servlets and JSPs invoked from within a portlet in the render phase should generate markup fragments following the recommendations in Appendix B Markup Fragments.

## Obtaining a PortletRequestDispatcher

PortletRequestDispatcher objects may be obtained using one of the following methods of the PortletContext object:

* getRequestDispatcher
* getNamedDispatcher

The getRequestDispatcher method takes a String argument describing a path within the scope of the PortletContext of a portlet application. This path must begin with a ‘/’ and it is relative to the PortletContext root.

The getNamedDispatcher method takes a String argument indicating the name of a servlet known to the PortletContext of the portlet application.

Ifno resource can be resolved based on the given path or name the methods must return null.

### Query Strings in Request Dispatcher Paths

The PortletContext interface getRequestDispatcher method creates PortletRequestDispatcher objects using path information and optional query string information. For example, a developer may obtain a PortletRequestDispatcher by using the following code:

String path = "/raisons.jsp?orderno=5";

PortletRequestDispatcher rd = context.getRequestDispatcher(path);

rd.include(renderRequest, renderResponse);

The portlet container must aggregate parameters specified in the query string used to create the PortletRequestDispatcher with the portlet render parameters. Query string parameters take precedence over other portlet render parameters of the same name passed to the servlet or JSP targeted by the forward or include.

The parameters associated with a PortletRequestDispatcher are scoped to apply only for the duration of the forward or include call.

#### Accessing Merged Parameters through the Servlet Request

The HttpServletRequest interface getParameter, getParameterMap, getParameterNames, and getParameterValues methods do not discern between portlet render parameters, action parameters, and resource parameters, so the portlet container must merge the portlet render parameters for access through these methods when more than one type of portlet parameter is present. The parameters must be merged in the following order of precedence.

1. Query string parameters specified when obtaining a portlet request dispatcher
2. Portlet action parameters or portlet resource parameters
3. Portlet render parameters

When there is no name clash between the different parameter types, the order of precedence plays no role and all parameters are made available according to the names and values set through the corresponding methods. However, when parameters of different types have the same parameter name, all parameter values must be added to the values array with the values of the parameter with higher precedence appearing before the values of the parameter with lower precedence.

Take for example the parameters in the following table:

|  |  |  |
| --- | --- | --- |
| **Parameter Type** | **Name** | **Values** |
| Query string | color | ["black", "white"] |
| Action parameter | color | ["green"] |
| Render parameter | color | ["white", "yellow"] |

The portlet container must merge these values into a single parameter with parameter name color and the values array ["black", "white", "green", "white", "yellow"] when accessed through the HttpServletRequest parameter methods named above.

#### Accessing Merged Parameters through the Portlet Request

When a query string is present, the portlet container must merge the query string parameters with the portlet render parameters with the query string parameters taking precedence over existing render parameters.

The query string parameters are considered to be private render parameters. Public render parameters cannot be set through the query string.

If a query string parameter name matches a public render parameter name, the query string parameter value is not made available to other portlets. However, both the query string parameter values and the public render parameter values will be available to the dispatching portlet, with the query string values appearing before the public render parameter values in the values array.

When portlet URLs are created during dispatch target execution, the query string parameters will be available as private render parameters on resource URLs, action URLs created using the MimeResponse#createActionURL(Copy.ALL) method, and on render URLs created using the MimeResponse#createRenderURL(Copy.ALL) method.

### Path and Query Information

When the portlet has started asynchronous processing through one of the ResourceRequest startPortletAsync methods, the HttpServletRequest object getPathInfo, getPathTranslated, getQueryString, getRequestURI and getServletPath methods must behave as defined by the servlet specification.

Otherwise, when the portlet is the starting point of the request dispatcher dispatch chain, the HttpServletRequest object getPathInfo, getPathTranslated, getQueryString, getRequestURI and getServletPath methods must return the path and query string information used to obtain the PortletRequestDispatcher object.

This differs from servlet processing when no portlet is involved. With forwards and includes originating from a servlet, these values are based on the path and query string of the client request. This makes sense in terms of the servlet programming model, since the included or forward target is treated as if it were running in place of the servlet issuing the request dispatcher include or forward call.

In the portlet programming model, the portlet does not have direct access to the path and query string information from the original client request, so the portlet container must provide the target servlet or JSP with the path and query string information used to obtain the PortletRequestDispatcher object.

When doing additional includes or forwards from within a servlet that was target of a portlet request dispatcher call, the semantics outlined in this section must continue to hold. In particular, further dispatch target servlets or JSPs must be provided with the path and query string information used to obtain the PortletRequestDispatcher object when using the above named methods.

## Using a Request Dispatcher

To include a servlet or a JSP, a portlet calls the PortletRequestDispatcher interface include method. To forward the request processing to a servlet or JSP the portlet calls the forward method.

The parameters to these methods must be the request and response objects that govern the request being processed, or must be instances of subclasses of the corresponding wrapper classes. In the latter case, the wrapper instances must wrap the original request or response objects created by the container.

The portlet container must ensure that the servlet or JSP called through a PortletRequestDispatcher is called in the same thread that invoked the PortletRequestDispatcher interface include or forward method.

## Error Handling

Ifthe servlet or JSP that is the target of portlet request dispatching throws a runtime exception, a PortletException, or a checked exception of type IOException, the exception must be propagated to the initiating portlet. All other exceptions, including a ServletException, must be wrapped with a PortletException and rethrown. The root cause of the exception must be set to the original exception before being propagated.

If a portlet throws an exception during action request processing, all operations on the ActionResponse must be ignored including set events.

If a portlet throws an exception during event processing, all operations on the EventResponse must be ignored including set events. Operations performed during the originating action request processing or during separate, successful event request processing cycles for the current or other portlets must remain unaffected. After the exception has been handled, the portlet container should continue processing other portlets on the portal page.

If a portlet throws an exception during render request or header request processing, the portlet container should clear any data written by the portlet that has not yet been flushed to the portal application[[21]](#footnote-21). If no data has been flushed to the portal application, the portlet container should clear any headers or page resources set by the portlet during the current request. Data written by the portal itself or by other portlets as part of the overall portal response must not be affected. The portlet container should continue processing other portlets on the portal page.

If a portlet throws an exception during resource request processing, the portlet container should clear any data written by the portlet that has not yet been flushed to the portal application. If no data has been flushed to the portal application, the portlet container should clear any headers set by the portlet. The portlet container should respond to the client request with an appropriate HTTP status code.

## Portlet-Specific Request Attributes

In addition to the request attributes specified in the Servlet Specification*,* the portlet request dispatching target servlet or JSP must have the following request attributes set:

|  |  |
| --- | --- |
| **Request Attribute Name** | **Type** |
| javax.portlet.config | javax.portlet.PortletConfig |

For includes during action request processing, the following additional attributes must be set:

|  |  |
| --- | --- |
| **Request Attribute Name** | **Type** |
| javax.portlet.request | javax.portlet.ActionRequest |
| javax.portlet.response | javax.portlet ActionResponse |

For includes during event processing, the following additional attributes must be set:

|  |  |
| --- | --- |
| **Request Attribute Name** | **Type** |
| javax.portlet.request | javax.portlet.EventRequest |
| javax.portlet.response | javax.portlet.EventResponse |

For includes during header request processing, the following additional attributes must be set:

|  |  |
| --- | --- |
| **Request Attribute Name** | **Type** |
| javax.portlet.request | javax.portlet.HeaderRequest |
| javax.portlet.response | javax.portlet.HeaderResponse |

For includes during render request processing, the following additional attributes must be set:

|  |  |
| --- | --- |
| **Request Attribute Name** | **Type** |
| javax.portlet.request | javax.portlet.RenderRequest |
| javax.portlet.response | javax.portlet.RenderResponse |

For includes during resource request processing, the following additional attributes must be set:

|  |  |
| --- | --- |
| **Request Attribute Name** | **Type** |
| javax.portlet.request | javax.portlet.ResourceRequest |
| javax.portlet.response | javax.portlet.ResourceResponse |

These attributes must be the same objects available to the portlet before invoking the PortletRequestDispatcher interface include or forward method, or instances of the appropriate wrapper classes wrapping the original objects. The dispatch target servlet or JSP can access the attributes using the HttpServletRequest object getAttribute method.

### Changing the Default Behavior for Session Scope

The session object that can be obtained through the HttpServletRequest object in an included JSP or servlet maps to the portlet session object with application scope by default. Some portlets may require that this object maps instead to the portlet session object with portlet scope in order to work correctly. This behavior can be influenced by the javax.portlet.servletDefaultSessionScope portlet container runtime option. See Section 8.4.3 Runtime Option javax.portlet.servletDefaultSessionScope on page 51.

## The forward Method

The PortletRequestDispatcher interface forward method may be called only when no output has been flushed to the portal application[[22]](#footnote-22). The portlet request dispatcher forward method allows the dispatch target servlet or JSP to set the response content.

Information like cookies, properties, portlet mode, window state, render parameters, or the portlet title that the portlet may have set before calling the portlet request dispatcher forward method should still be valid.

If output data exists in the response buffer that has not been flushed to the portal application, the content must be cleared before the target servlet service method is called. If the response has been flushed to the portal application, an IllegalStateException must be thrown.

Before the RequestDispatcher interface forward method returns, the portlet container must flush any response data to the portal application and close the output stream for the portlet, unless the request was put into asynchronous mode. If an error occurs in the target of a request dispatcher forward, the exception may be propagated back through all of the calling filters and servlets and eventually back to the portlet container.

When using a RequestDispatcher in a servlet that was target of a forward from a portlet, the servlet must request the RequestDispatcher via the ServletRequest and not the ServletContext. Using a RequestDispatcher that was retrieved from the ServletContext may behave in a way that does not comply with this specification.

### Forwarded Request Attributes

If the portlet request dispatcher was obtained using the getNamedDispatcher method, or if asynchronous processing was started before the dispatch was carried out, the request attributes described in this section must behave as described by the servlet specification.

Otherwise, a servlet that has been invoked by a portlet using the forward method of RequestDispatcher has access to the path used to obtain the PortletRequestDispatcher. The following request attributes must be set:

javax.servlet.forward.request\_uri

javax.servlet.forward.context\_path

javax.servlet.forward.servlet\_path

javax.servlet.forward.path\_info

javax.servlet.forward.query\_string

The values of these attributes must be equal to the return values of the HttpServletRequest methods getRequestURI, getContextPath, getServletPath, getPathInfo, getQueryString respectively, invoked on the request object passed to the first servlet object in the forward call chain.

These attributes are accessible from the forwarded servlet via the getAttribute method on the request object. Note that these attributes must always reflect the information in the target of the first forward servlet even in the situation that multiple forwards and subsequent includes are called.

## The Include Method

The PortletRequestDispatcher interface include method may be called at any time and multiple times during request processing. The servlet or JSP being included can make use of the received HttpServletRequest and HttpServletResponse objects subject to restrictions.

The portlet request dispatcher include target cannot set response headers. The portlet container must ignore any method call that would affect the response headers.

The included servlet may use the HttpServletRequest interface getSession or getSession(boolean) methods; however, if use of these methods would require adding a Cookie response header, the portlet container must throw a IllegalStateException if response data has already been flushed to the portal application.

Output written to the HttpServletResponse during the header phase must be added to the overall portal document head section.

Servlets and JSPs included by portlets should not use the servlet RequestDispatcher forward method as its behavior may be non-deterministic.

Servlets and JSPs included from portlets in the render method must be handled as HTTP GET requests.

The lookup of the servlet given a path is done according to the servlet path matching rule defined the Servlet Specification[[23]](#footnote-23).

### Included Request Attributes

Servlets and JSPs included by portlets must be provided with request attributes as defined by the Servlet Specification[[24]](#footnote-24).

Except for servlets obtained by using the getNamedDispatcher method, a servlet or JSP being used from within an include call has access to the path used to obtain the PortletRequestDispatcher. The following request attributes must be set:

javax.servlet.include.request\_uri

javax.servlet.include.context\_path

javax.servlet.include.servlet\_path

javax.servlet.include.path\_info

javax.servlet.include.query\_string

These attributes are accessible from the included servlet using the HttpServletRequest object getAttribute method and their values must be equal to the request URI, context path, servlet path, path info, and query string of the included servlet, respectively. If the request is subsequently included, these attributes are replaced for that include.

If the included servlet was obtained by using the getNamedDispatcher method, these attributes must not be set.

Note that this is the same behavior described by the Servlet Specification.

## Asynchronous Processing

As described in Section 21.1.1 Starting Asynchronous Processing on page 124, the portlet container is not required to support the initiation of an asynchronous processing sequence during a portlet request dispatcher include or forward.

However, if asynchronous processing was started in a resource method prior to a portlet request dispatcher include or forward, the servlet request asynchronous processing methods startAsync(), startAsync(ServletRequest, ServletResponse), getAsyncContext(), isAsyncStarted(), and isAsyncSupported()must map to the corresponding portlet asynchronous processing methods and return analogous objects.

The servlet AsyncContext object returned by the servlet request startAsync and getAsyncContext methods must provide the functionality described by the Servlet Specification with the following exception: The portlet container is not required to support the servlet AsyncContext dispatch(ServletContext, String) method.

During execution of servlet that is an asynchronous PortletAsyncContext interface dispatch(String) method target, any query string provided as part of the argument must be processed according to the rules specified in Section 25.1.1 Query Strings in Request Dispatcher Paths on page 156. The HttpServletRequest and HttpServletResponse methods available to the target servlet must be mapped to portlet methods according to the tables given in Section 25.9.2 Method Behavior during Portlet forward and Asynchronous Dispatching on page 167.

## Servlet filters and Request Dispatching

The Java Servlet Specification allows servlet filters to be specified for servlets that are request dispatching include and forward targets[[25]](#footnote-25). The portlet container must support this capability for PortletRequestDispatcher dispatching targets. The servlet filters for the servlets included via the PortletRequestDispatcher must be defined as described in the Java Servlet Specification.

## HttpServletRequest and HttpServletResponse Objects

When the portlet includes a servlet or JSP, the portlet container must ensure that the capabilities of the servlet request and response methods supplied to the JSP or servlet are restricted in order to maintain the proper portlet request processing semantics.

The tables below defines the required behavior of the HttpServletRequest and HttpServletResponse methods. For each servlet method in the left column, the subsequent columns define the required output based on the portlet interface named in the column header.

Sometimes the required output can be a constant, such as ‘0’, null, or ‘HTTP/1.1’. Where the table cell names a method, the behavior of the servlet method must be the same as the named method applied to the portlet interface named in the column header.

Taking the HttpServletRequest interface getContentLength method as an example, the method must return the same value as ActionRequest.getContentLength during action request processing, but must return 0 during render request processing.

The response of the deprecated HttpUtils.getRequestURL method is undefined and should not be used.

There are separate tables for the request dispatcher include and the request dispatcher forward. Where there is a difference in processing during an include as compared to a forward, the table cell is marked with this background color.

### Method Behavior during Portlet include Dispatching

| **HttpServletRequest method** | **ActionRequest mapping** | **EventRequest mapping** | **HeaderRequest and**  **RenderRequest mappings** | **ResourceRequest mapping** |
| --- | --- | --- | --- | --- |
| getAuthType | getAuthType | getAuthType | getAuthType | getAuthType |
| getContextPath | getContextPath | getContextPath | getContextPath | getContextPath |
| getCookies | getCookies | getCookies | getCookies | getCookies |
| getDateHeader | getProperties | getProperties | getProperties | getProperties |
| getHeader | getProperties | getProperties | getProperties | getProperties |
| getHeaderNames | getPropertyNames | getPropertyNames | getPropertyNames | getPropertyNames |
| getHeaders | getProperties | getProperties | getProperties | getProperties |
| getIntHeader | getProperties | getProperties | getProperties | getProperties |
| getMethod | getMethod | getMethod | ‘GET’ | getMethod |
| getPathInfo | Path and query string26 | Path and query string26 | Path and query string26 | Path and query string26 |
| getPathTranslated | Path and query string26 | Path and query string26 | Path and query string26 | Path and query string26 |
| getQueryString | Path and query string26 | Path and query string26 | Path and query string26 | Path and query string26 |
| getRemoteUser | getRemoteUser | getRemoteUser | getRemoteUser | getRemoteUser |
| getRequestedSessionId | getRequestedSessionId | getRequestedSessionId | getRequestedSessionId | getRequestedSessionId |
| getRequestURI | Path and query string[[26]](#footnote-26) | Path and query string26 | Path and query string26 | Path and query string26 |
| getRequestURL | null | null | null | null |
| getServletPath | Path and query string26 | Path and query string26 | Path and query string26 | Path and query string26 |
| getSession | getPortletSession[[27]](#footnote-27) | getPortletSession27 | getPortletSession27 | getPortletSession27 |
| getUserPrincipal | getUserPrincipal | getUserPrincipal | getUserPrincipal | getUserPrincipal |
| isRequestedSessionIdFromCookie | N/A[[28]](#footnote-28) | N/A28 | N/A28 | N/A28 |
| isRequestedSessionIdFromUrl | N/A28 | N/A28 | N/A28 | N/A28 |
| isRequestedSessionIdFromURL | N/A28 | N/A28 | N/A28 | N/A28 |
| isRequestedSessionIdValid | isRequestedSessionIdValid | isRequestedSessionIdValid | isRequestedSessionIdValid | isRequestedSessionIdValid |
| isUserInRole | isUserInRole | isUserInRole | isUserInRole | isUserInRole |
| getAttribute | getAttribute | getAttribute | getAttribute | getAttribute |
| getAttributeNames | getAttributeNames | getAttributeNames | getAttributeNames | getAttributeNames |
| getCharacterEncoding | getCharacterEncoding | null | null | getCharacterEncoding |
| getContentLength | getContentLength | 0 | 0 | getContentLength |
| getContentType | getContentType | null | null | getContentType |
| getInputStream | getPortletInputStream | null | null | getPortletInputStream |
| getLocalAddr | null | null | null | null |
| getLocale | getLocale | getLocale | getLocale | getLocale |
| getLocales | getLocales | getLocales | getLocales | getLocales |
| getLocalName | null | null | null | null |
| getLocalPort | 0 | 0 | 0 | 0 |
| getParameter | merged parameters[[29]](#footnote-29) | merged parameters29 | merged parameters29 | merged parameters29 |
| getParameterMap | merged parameters29 | merged parameters29 | merged parameters29 | merged parameters29 |
| getParameterNames | merged parameters29 | merged parameters29 | merged parameters29 | merged parameters29 |
| getParameterValues | merged parameters29 | merged parameters29 | merged parameters29 | merged parameters29 |
| getProtocol | HTTP/1.1 | HTTP/1.1 | HTTP/1.1 | HTTP/1.1 |
| getReader | getReader | null | null | getReader |
| getRealPath | null | null | null | null |
| getRemoteAddr | null | null | null | null |
| getRemoteHost | null | null | null | null |
| getRemotePort | 0 | 0 | 0 | 0 |
| getRequestDispatcher | N/A28 | N/A28 | N/A28 | N/A28 |
| getScheme | getScheme | getScheme | getScheme | getScheme |
| getServerName | getServerName | getServerName | getServerName | getServerName |
| getServerPort | getServerPort | getServerPort | getServerPort | getServerPort |
| isSecure | isSecure | isSecure | isSecure | isSecure |
| removeAttribute | removeAttribute | removeAttribute | removeAttribute | removeAttribute |
| setAttribute | setAttribute | setAttribute | setAttribute | setAttribute |
| setCharacterEncoding | setCharacterEncoding | no-op[[30]](#footnote-30) | no-op30 | setCharacterEncoding |
| getContentLengthLong | getContentLengthLong | 0L | 0L | getContentLengthLong |
| getServletContext | N/A28 | N/A28 | N/A28 | N/A28 |
| startAsync | IllegalStateException[[31]](#footnote-31) | IllegalStateException31 | IllegalStateException31 | startPortletAsync[[32]](#footnote-32) |
| startAsync | IllegalStateException31 | IllegalStateException31 | IllegalStateException31 | startPortletAsync32 |
| isAsyncStarted | false | false | false | isAsyncStarted |
| isAsyncSupported | false | false | false | isAsyncSupported |
| getAsyncContext | IllegalStateException31 | IllegalStateException31 | IllegalStateException31 | getPortletAsyncContext32 |
| getDispatcherType | getDispatcherType | getDispatcherType | getDispatcherType | getDispatcherType |
| changeSessionId | null | null | null | null |
| authenticate | false | false | false | false |
| login | no-op30 | no-op30 | no-op30 | no-op30 |
| logout | no-op30 | no-op30 | no-op30 | no-op30 |
| getParts | getParts | no-op30 | no-op30 | getParts |
| getPart | getPart | no-op30 | no-op30 | getPart |
| upgrade | ServletException[[33]](#footnote-33) | ServletException33 | ServletException33 | ServletException33 |

| **HttpServletResponse method** | **ActionResponse mapping** | **EventResponse mapping** | **HeaderResponse and**  **RenderResponse mappings** | **ResourceResponse mapping** |
| --- | --- | --- | --- | --- |
| addCookie | no-op30 | no-op30 | no-op30 | no-op30 |
| addDateHeader | no-op30 | no-op30 | no-op30 | no-op30 |
| addHeader | no-op30 | no-op30 | no-op30 | no-op30 |
| addIntHeader | no-op30 | no-op30 | no-op30 | no-op30 |
| containsHeader | false | false | false | false |
| encodeRedirectUrl | null | null | null | null |
| encodeRedirectURL | null | null | null | null |
| encodeUrl | encodeURL | encodeURL | encodeURL | encodeURL |
| encodeURL | encodeURL | encodeURL | encodeURL | encodeURL |
| sendError | no-op30 | no-op30 | no-op30 | no-op30 |
| sendRedirect | no-op30 | no-op30 | no-op30 | no-op30 |
| setDateHeader | no-op30 | no-op30 | no-op30 | no-op30 |
| setHeader | no-op30 | no-op30 | no-op30 | no-op30 |
| setIntHeader | no-op30 | no-op30 | no-op30 | no-op30 |
| setStatus | no-op30 | no-op30 | no-op30 | no-op30 |
| flushBuffer | no-op30 | no-op30 | flushBuffer | flushBuffer |
| getBufferSize | 0 | 0 | getBufferSize | getBufferSize |
| getCharacterEncoding | null | null | getCharacterEncoding | getCharacterEncoding |
| getContentType | null | null | getContentType | getContentType |
| getLocale | null | null | getLocale | getLocale |
| getOutputStream | null stream[[34]](#footnote-34) | null stream34 | getPortletOutputStream | getPortletOutputStream |
| getWriter | null writer34 | null writer34 | getWriter | getWriter |
| isCommitted | true | true | isCommitted | isCommitted |
| reset | no-op30 | no-op30 | reset | reset |
| resetBuffer | no-op30 | no-op30 | resetBuffer | resetBuffer |
| setBufferSize | no-op30 | no-op30 | setBufferSize | setBufferSize |
| setCharacterEncoding | no-op30 | no-op30 | no-op30 | no-op30 |
| setContentLength | no-op30 | no-op30 | no-op30 | no-op30 |
| setContentType | no-op30 | no-op30 | no-op30 | no-op30 |
| setLocale | no-op30 | no-op30 | no-op30 | no-op30 |
| setContentLengthLong | no-op30 | no-op30 | no-op30 | setContentLengthLong |
| getStatus | SC\_OK = 200 | SC\_OK = 200 | SC\_OK = 200 | getStatus |
| getHeader | getProperty | getProperty | getProperty | getProperty |
| getHeaders | getPropertValues | getPropertValues | getPropertValues | getPropertValues |
| getHeaderNames | getPropertyNames | getPropertyNames | getPropertyNames | getPropertyNames |

### Method Behavior during Portlet forward and Asynchronous Dispatching

| **HttpServletRequest method** | **ActionRequest mapping** | **EventRequest mapping** | **HeaderRequest and**  **RenderRequest mappings** | **ResourceRequest mapping** |
| --- | --- | --- | --- | --- |
| getAuthType | getAuthType | getAuthType | getAuthType | getAuthType |
| getContextPath | getContextPath | getContextPath | getContextPath | getContextPath |
| getCookies | getCookies | getCookies | getCookies | getCookies |
| getDateHeader | getProperties | getProperties | getProperties | getProperties |
| getHeader | getProperties | getProperties | getProperties | getProperties |
| getHeaderNames | getPropertyNames | getPropertyNames | getPropertyNames | getPropertyNames |
| getHeaders | getProperties | getProperties | getProperties | getProperties |
| getIntHeader | getProperties | getProperties | getProperties | getProperties |
| getMethod | getMethod | ActionRequest.getMethod | ‘GET’ | getMethod |
| getPathInfo | Path and query string26 | Path and query string26 | Path and query string26 | Path and query string26 |
| getPathTranslated | Path and query string26 | Path and query string26 | Path and query string26 | Path and query string26 |
| getQueryString | Path and query string26 | Path and query string26 | Path and query string26 | Path and query string26 |
| getRemoteUser | getRemoteUser | getRemoteUser | getRemoteUser | getRemoteUser |
| getRequestedSessionId | getRequestedSessionId | getRequestedSessionId | getRequestedSessionId | getRequestedSessionId |
| getRequestURI | Path and query string26 | Path and query string26 | Path and query string26 | Path and query string26 |
| getRequestURL | null | null | null | null |
| getServletPath | Path and query string26 | Path and query string26 | Path and query string26 | Path and query string26 |
| getSession | getPortletSession27 | getPortletSession27 | getPortletSession27 | getPortletSession27 |
| getUserPrincipal | getUserPrincipal | getUserPrincipal | getUserPrincipal | getUserPrincipal |
| isRequestedSessionIdFromCookie | N/A28 | N/A28 | N/A28 | N/A28 |
| isRequestedSessionIdFromUrl | N/A28 | N/A28 | N/A28 | N/A28 |
| isRequestedSessionIdFromURL | N/A28 | N/A28 | N/A28 | N/A28 |
| isRequestedSessionIdValid | isRequestedSessionIdValid | isRequestedSessionIdValid | isRequestedSessionIdValid | isRequestedSessionIdValid |
| isUserInRole | isUserInRole | isUserInRole | isUserInRole | isUserInRole |
| getAttribute | getAttribute | getAttribute | getAttribute | getAttribute |
| getAttributeNames | getAttributeNames | getAttributeNames | getAttributeNames | getAttributeNames |
| getCharacterEncoding | getCharacterEncoding | null | null | getCharacterEncoding |
| getContentLength | getContentLength | 0 | 0 | getContentLength |
| getContentType | getContentType | null | null | getContentType |
| getInputStream | getPortletInputStream | null | null | getPortletInputStream |
| getLocalAddr | null | null | null | null |
| getLocale | getLocale | getLocale | getLocale | getLocale |
| getLocales | getLocales | getLocales | getLocales | getLocales |
| getLocalName | null | null | null | null |
| getLocalPort | 0 | 0 | 0 | 0 |
| getParameter | merged parameters29 | merged parameters29 | merged parameters29 | merged parameters29 |
| getParameterMap | merged parameters29 | merged parameters29 | merged parameters29 | merged parameters29 |
| getParameterNames | merged parameters29 | merged parameters29 | merged parameters29 | merged parameters29 |
| getParameterValues | merged parameters29 | merged parameters29 | merged parameters29 | merged parameters29 |
| getProtocol | HTTP/1.1 | HTTP/1.1 | HTTP/1.1 | HTTP/1.1 |
| getReader | getReader | null | null | getReader |
| getRealPath | null | null | null | null |
| getRemoteAddr | null | null | null | null |
| getRemoteHost | null | null | null | null |
| getRemotePort | 0 | 0 | 0 | 0 |
| getRequestDispatcher | N/A28 | N/A28 | N/A28 | N/A28 |
| getScheme | getScheme | getScheme | getScheme | getScheme |
| getServerName | getServerName | getServerName | getServerName | getServerName |
| getServerPort | getServerPort | getServerPort | getServerPort | getServerPort |
| isSecure | isSecure | isSecure | isSecure | isSecure |
| removeAttribute | removeAttribute | removeAttribute | removeAttribute | removeAttribute |
| setAttribute | setAttribute | setAttribute | setAttribute | setAttribute |
| setCharacterEncoding | setCharacterEncoding | no-op30 | no-op30 | setCharacterEncoding |
| getContentLengthLong | getContentLengthLong | 0L | 0L | getContentLengthLong |
| getServletContext | N/A28 | N/A28 | N/A28 | N/A28 |
| startAsync | IllegalStateException31 | IllegalStateException31 | IllegalStateException31 | startPortletAsync32 |
| startAsync | IllegalStateException31 | IllegalStateException31 | IllegalStateException31 | startPortletAsync32 |
| isAsyncStarted | false | false | false | isAsyncStarted |
| isAsyncSupported | false | false | false | isAsyncSupported |
| getAsyncContext | IllegalStateException31 | IllegalStateException31 | IllegalStateException31 | getPortletAsyncContext32 |
| getDispatcherType | getDispatcherType | getDispatcherType | getDispatcherType | getDispatcherType |
| changeSessionId | null | null | null | null |
| authenticate | false | false | false | false |
| login | no-op30 | no-op30 | no-op30 | no-op30 |
| logout | no-op30 | no-op30 | no-op30 | no-op30 |
| getParts | getParts | no-op30 | no-op30 | getParts |
| getPart | getPart | no-op30 | no-op30 | getPart |
| upgrade | ServletException33 | ServletException33 | ServletException33 | ServletException33 |

| **HttpServletResponse method** | **ActionResponse mapping** | **EventResponse mapping** | **HeaderResponse and**  **RenderResponse mappings** | **ResourceResponse mapping** |
| --- | --- | --- | --- | --- |
| addCookie | addProperty | addProperty | addProperty | addProperty |
| addDateHeader | no-op30 | no-op30 | addProperty | addProperty |
| addHeader | no-op30 | no-op30 | addProperty | addProperty |
| addIntHeader | no-op30 | no-op30 | addProperty | addProperty |
| containsHeader | false | false | false | false |
| encodeRedirectUrl | null | null | null | null |
| encodeRedirectURL | null | null | null | null |
| encodeUrl | encodeURL | encodeURL | encodeURL | encodeURL |
| encodeURL | encodeURL | encodeURL | encodeURL | encodeURL |
| sendError | no-op30 | no-op30 | no-op30 | no-op30 |
| sendRedirect | sendRedirect30 | no-op30 | no-op30 | no-op30 |
| setDateHeader | no-op30 | no-op30 | setProperty | setProperty |
| setHeader | no-op30 | no-op30 | setProperty | setProperty |
| setIntHeader | no-op30 | no-op30 | setProperty | setProperty |
| setStatus | no-op30 | no-op30 | no-op30 | setStatus |
| flushBuffer | no-op30 | no-op30 | flushBuffer | flushBuffer |
| getBufferSize | 0 | 0 | getBufferSize | getBufferSize |
| getCharacterEncoding | null | null | getCharacterEncoding | getCharacterEncoding |
| getContentType | null | null | getContentType | getContentType |
| getLocale | null | null | getLocale | getLocale |
| getOutputStream | null stream34 | null stream34 | getPortletOutputStream | getPortletOutputStream |
| getWriter | null writer34 | null writer34 | getWriter | getWriter |
| isCommitted | false | false | isCommitted | isCommitted |
| reset | no-op30 | no-op30 | reset | reset |
| resetBuffer | no-op30 | no-op30 | resetBuffer | resetBuffer |
| setBufferSize | no-op30 | no-op30 | setBufferSize | setBufferSize |
| setCharacterEncoding | no-op30 | no-op30 | no-op30 | setCharacterEncoding |
| setContentLength | no-op30 | no-op30 | no-op30 | setContentLength |
| setContentType | no-op30 | no-op30 | setContentType | setContentType |
| setLocale | no-op30 | no-op30 | no-op30 | setLocale |
| setContentLengthLong | no-op30 | no-op30 | no-op30 | setContentLengthLong |
| getStatus | SC\_OK = 200 | SC\_OK = 200 | SC\_OK = 200 | getStatus |
| getHeader | getProperty | getProperty | getProperty | getProperty |
| getHeaders | getPropertValues | getPropertValues | getPropertValues | getPropertValues |
| getHeaderNames | getPropertyNames | getPropertyNames | getPropertyNames | getPropertyNames |

# Portlet Tag Library

The portlet tag library allows JavaServer Pages that are portlet request dispatcher targets to access portlet-specific artifacts such as the HeaderRequest, RenderRequest, ResourceRequest, ActionResponse, or RenderResponse. It also provides custom JSP tags that access portlet functionality such as portlet URL creation.

The portlet-container must provide an implementation of the portlet tag library. Portlet developers may indicate an alternate implementation using the mechanism defined in the JSP Specification[[35]](#footnote-35).

JSP pages that use the portlet tag library must declare its use through a taglib directive with the uri attribute set to "http://xmlns.jcp.org/portlet\_3\_0". The example below shows a taglib directive using the suggested prefix value "portlet".

<%@ taglib uri="http://xmlns.jcp.org/portlet\_3\_0" prefix="portlet" %>

The portlet container must support JavaServer Pages Version 2.3 and Expression Language Version 3.0 for the tags in the portlet tag library.

The portlet container must support taglib directives for older versions of the portlet tag library. The table below shows corresponding example taglib directives for the Portlet Specification Version 1.0 and 2.0.

|  |  |
| --- | --- |
| **Version 2.0** | <%@ taglib uri="http://java.sun.com/portlet\_2\_0" prefix="portlet" %> |
| **Version 1.0** | <%@ taglib uri="http://java.sun.com/portlet" prefix="portlet" %> |

In order to support migration, JSPs included by newer portlet versions may reference older tag library versions. For example, a version 3.0 portlet may include a JSP that contains a version 2.0 taglib directive.

## defineObjects Tag

The defineObjects tag must define the following variables in the JSP page:

1. RenderRequest renderRequest when included during render request processing, null or not defined otherwise.
2. ResourceRequest resourceRequest when included during resource request processing, null or not defined otherwise.
3. ActionRequest actionRequest when included during action request processing, null or not defined otherwise.
4. EventRequest eventRequest when included during event request processing, null or not defined otherwise.
5. RenderResponse renderResponse when included during render request processing, null or not defined otherwise.
6. ResourceResponse resourceResponse when included during resource request processing, null or not defined otherwise.
7. ActionResponse actionResponse when included during action request processing, null or not defined otherwise.
8. EventResponse eventResponse when included during event request processing, null or not defined otherwise.
9. PortletConfig portletConfig must always be defined.
10. PortletSession portletSession, providing access to the portlet session, does not create a new session, only returns an existing session or null if no session exists.
11. Map<String, Object> portletSessionScope, providing access to the portlet session attributes as a Map equivalent to the PortletSession.getAttributeMap() call, does not create a new session, and only returns an existing session. If no session attributes exist this method returns an empty Map.
12. PortletPreferences portletPreferences, providing access to the portlet preferences.
13. Map<String, String[]> portletPreferencesValues, providing access to the portlet preferences as a Map, equivalent to the PortletPreferences.getMap() call. If no portlet preferences exist this method returns an empty Map.
14. PortletRequest portletRequest must always be defined.
15. PortletResponse portletResponse must always be defined.
16. HeaderRequest headerRequest during header request processing, null or not defined otherwise.
17. HeaderResponse headerResponse during header request processing, null or not defined otherwise.
18. ClientDataRequest clientDataRequest during action and resource request processing, null or not defined otherwise.
19. MimeResponse mimeResponse during header, render, and resource request processing, null or not defined otherwise.
20. StateAwareResponse stateAwareResponse during action and event request processing, null or not defined otherwise.
21. RenderParameters renderParams must always be defined.
22. MutableRenderParameters mutableRenderParams must be defined during action and event request processing.
23. ResourceParameters resourceParams must be defined during resource request processing.
24. ActionParameters actionParams must be defined during action request processing.
25. PortletContext portletContext must always be defined.
26. PortletMode portletMode must always be defined.
27. WindowState windowState must always be defined.
28. Cookie[] cookies must always be defined.
29. Locale locale must always be defined.
30. Locale[] locales must always be defined.
31. String namespace must always be defined.
32. String contextPath must always be defined.
33. String windowId must always be defined.
34. String portletName must always be defined.

These variables must reference the same portlet artifacts stored as request attributes as defined in Section 25.4 Portlet-Specific Request Attributes.

A JSP using the defineObjects tag may use these variables in scriptlets throughout the page.

The defineObjects tag must not define any attribute and it must not contain any body content.

An example of a JSP using the defineObjects tag could be:

<portlet:defineObjects/>

<%=renderResponse.getCacheControl().setExpirationTime(10)%>

After using the defineObjects tag, the JSP invokes the renderResponse object getCacheControl() method to set the expiration time of the response to 10 seconds.

## actionURL Tag

The portlet actionURL tag creates a URL that must point to the current portlet and must trigger an action request with the supplied parameters.

Action parameters may be added to the URL by including the param tag between the actionURL start and end tags.

The following attributes are defined for this tag:

* **windowState** (Type: String, optional) – indicates the window state that the portlet should have when this link is executed. The following window states are predefined: minimized, normal, and maximized. If the specified window state is illegal for the current request, a JspException must be thrown. Reasons for a window state being illegal may include that the portal does not support this state, the portlet has not declared in its deployment descriptor that it supports this state, or the current user is not allowed to switch to this state. If a window state is not set for a URL, it should stay the same as the window state of the current request. The window state attribute is not case sensitive.
* **portletMode** (Type: String, optional) – indicates the portlet mode that the portlet must have when this link is executed, if no error condition ocurred. The following portlet modes are predefined: edit, help, and view. If the specified portlet mode is illegal for the current request, a JspException must be thrown. Reasons for a portlet mode being illegal may include that the portal does not support this mode, the portlet has not declared in its deployment descriptor that it supports this mode for the current markup, or the current user is not allowed to switch to this mode. If a portlet mode is not set for a URL, it must stay the same as the mode of the current request. The portlet mode attribute is not case sensitive.
* **var** (Type: String, optional) – name of the exported scoped variable for the action URL. The exported scoped variable must be a String. By default, the result of the URL processing is written to the current JspWriter. If the result is exported as a JSP scoped variable, defined via the var attributes, nothing is written to the current JspWriter.

*Note:* After the URL is created it is not possible to extend the URL or add any further parameter using the variable and String concatenation. If the given variable name already exists in the scope of the page or it is used within an iteration loop, the new value overwrites the old one.

* **secure** (Type: String, optional) – indicates if the resulting URL should be a secure connection (secure="true") or an insecure one (secure="false"). If the specified security setting is not supported by the run-time environment, a JspException must be thrown. If the security is not set for a URL, it must stay the same as the security setting of the current request.
* **copyCurrentRenderParameters** (Type: boolean, optional) – if set to true requests that the private render parameters of the portlet of the current request must be attached to this URL. It is equivalent to setting each of the current private render parameters via the <portlet:param> tag. If additional <portlet:param> tags are specified parameters with the same name as an existing render parameter will get merged and the value defined in additional <portlet:param> tags must be pre-pended.   
  The default for this attribute is false.
* **escapeXml** (Type: boolean, optional) – determines whether the resulting output is to be XML escaped. The manner in which the output is escaped is implementation specific. If this option is set to false, whether or not the output is XML escaped is left to the implementation. Default value is true.
* **name** (Type: String, optional) – specifies the value for the action name parameter that can be used during request dispatching. See Section 4.7.1 Dispatching to GenericPortlet Annotated Methods. Setting this name will result in adding a parameter to this action URL with the name javax.portlet.action.

A JspException with the PortletException that caused this error as root cause is thrown in the following cases:

* If an illegal window state is specified in the windowState attribute.
* If an illegal portlet mode is specified in the portletMode attribute.
* If an illegal security setting is specified in the secure attribute.

A JspException with the java.lang.IllegalStateException that caused this error as root cause is thrown in the following cases:

* If this tag is used in markup generated during resource request processing that was directly or indirectly triggered via a resource URL of type FULL or PORTLET.

An example of a JSP using the actionURL tag could be:

<portlet:actionURL copyCurrentRenderParameters="true" windowState="maximized" portletMode="edit" name="editStocks">

<portlet:param name="page" value="1"/>

</portlet:actionURL>

The example creates a URL that brings the portlet into EDIT mode and MAXIMIZED window state to edit the stocks quote list.

## renderURL Tag

The portlet renderURL tag creates a URL that must point to the current portlet and must trigger a render request with the supplied parameters.

Render parameters may be added by including the param tag between the renderURL start and end tags.

The following attributes are defined for this tag:

* **windowState** (Type: String, optional) – indicates the window state that the portlet should have when this link is executed. The following window states are predefined: minimized, normal, and maximized. If the specified window state is illegal for the current request, a JspException must be thrown. Reasons for a window state being illegal may include that the portal does not support this state, the portlet has not declared in its deployment descriptor that it supports this state, or the current user is not allowed to switch to this state. If a window state is not set for a URL, it should stay the same as the window state of the current request. The window state attribute is not case sensitive.
* **portletMode** (Type: String, optional) – indicates the portlet mode that the portlet must have when this link is executed, if not error condition ocurred. The following portlet modes are predefined: edit, help, and view. If the specified portlet mode is illegal for the current request, a JspException must be thrown. Reasons for a portlet mode being illegal may include that the portal does not support this mode, the portlet has not declared in its deployment descriptor that it supports this mode for the current markup, or the current user is not allowed to switch to this mode. If a portlet mode is not set for a URL, it must stay the same as the mode of the current request. The portlet mode attribute is not case sensitive.
* **var** (Type: String, optional) – name of the exported scoped variable for the render URL. The exported scoped variable must be a String. By default, the result of the URL processing is written to the current JspWriter. If the result is exported as a JSP scoped variable, defined via the var attributes, nothing is written to the current JspWriter.

*Note:* After the URL is created it is not possible to extend the URL or add any further parameter using the variable and String concatenation. If the given variable name already exists in the scope of the page or it is used within an iteration loop, the new value overwrites the old one.

* **secure** (Type: String, optional) – indicates if the resulting URL should be a secure connection (secure="true") or an insecure one (secure="false"). If the specified security setting is not supported by the run-time environment, a JspException must be thrown. If the security is not set for a URL, it must stay the same as the security setting of the current request.
* **copyCurrentRenderParameters** (Type: boolean, optional) – if set to true requests that the private render parameters of the portlet of the current request must attached to this URL. It is equivalent to setting each of the current private render parameters via the <portlet:param> tag. If additional <portlet:param> tags are specified parameters with the same name as an existing render parameter will get merged and the value defined in additional <portlet:param> tags must be pre-pended.   
  The default for this attribute is false.
* **escapeXml** (Type: boolean, optional) – determines whether the resulting output is to be XML escaped. The manner in which the output is escaped is implementation specific. If this option is set to false, whether or not the output is XML escaped is left to the implementation. Default value is true.

A JspException with the PortletException that caused this error as root cause is thrown in the following cases:

* If an illegal window state is specified in the windowState attribute.
* If an illegal portlet mode is specified in the portletMode attribute.
* If an illegal security setting is specified in the secure attribute.

A JspException with the java.lang.IllegalStateException that caused this error as root cause is thrown in the following cases:

* If this tag is used in markup generated during resource request processing that was directly or indirectly triggered via a resource URL of type FULL or PORTLET.

An example of a JSP using the renderURL tag could be:

<portlet:renderURL portletMode="view" windowState="normal">

<portlet:param name="showQuote" value="myCompany"/>

<portlet:param name="showQuote" value="someOtherCompany"/>

</portlet:renderURL>

The example creates a URL to provide a link that shows the stock quote of myCompany and someOtherCompany and changes the portlet mode to TTVIEWTT and the window state to TTNORMALTT.

## resourceURL Tag

The portlet resourceURL tag creates a URL that must point to the current portlet and must trigger a resource request with the supplied parameters.

The resourceURL must preserve the current portlet mode, window state and render parameters.

Resource parameters may be added by including the param tag between the resourceURL start and end tags.

The following attributes are defined for this tag:

* **var** (Type: String, optional) – name of the exported scoped variable for the resource URL. The exported scoped variable must be a String. By default, the result of the URL processing is written to the current JspWriter. If the result is exported as a JSP scoped variable, defined via the var attributes, nothing is written to the current JspWriter.

*Note:* After the URL is created it is not possible to extend the URL or add any further parameter using the variable and String concatenation. If the given variable name already exists in the scope of the page or it is used within an iteration loop, the new value overwrites the old one.

* **secure** (Type: String, optional) – indicates if the resulting URL should be a secure connection (secure="true") or an insecure one (secure="false"). If the specified security setting is not supported by the run-time environment, a JspException must be thrown. If the security is not set for a URL, it must stay the same as the security setting of the current request.
* **escapeXml** (Type: boolean, optional) – determines whether the resulting output is to be XML escaped. The manner in which the output is escaped is implementation specific. If this option is set to false, whether or not the output is XML escaped is left to the implementation. Default value is true.
* **id** (type:String, optional) – sets the ID for this resource. The ID can be retrieved the ResourceRequest object via the getResourceID method.
* **cacheability** (type: String, optional) – defines the cacheability of the markup returned by this resource URL. Valid values are: "FULL", "PORTLET", and "PAGE". See Section 13.2 The Resource URL for more details on these constants. If cacheability is not set, the default is PAGE.

A JspException with the PortletException that caused this error as root cause is thrown in the following case:

* If an illegal security setting is specified in the secure attribute.

A JspException with the java.lang.IllegalStateException that caused this error as root cause is thrown in the following cases:

* If this tag is used in markup generated during resource request processing that was directly or indirectly triggered via a resource URL of a weaker cacheability type.

An example of a JSP using the resourceURL tag could be:

<portlet:resourceURL cacheability="FULL" var="iconsURL"/>

<img src="<%=iconsURL%>" >

The example creates a URL to provide a link that renders the icon named mypict.gif via the default GenericPortlet resource serving mechanism.

## namespace Tag

This tag produces a unique value for the current portlet and must match the value of PortletResponse.getNamespace method.

This tag should be used for named elements in the portlet output (such as JavaScript functions and variables). The namespacing ensures that the given name is uniquely associated with this portlet and avoids name conflicts with other elements on the portal page or with other portlets on the page.

The namespace tag must not allow any body content.

An example of a JSP using the namespace tag could be:

<A HREF="javascript:<portlet:namespace/>doFoo()">Foo</A>

The example prefixes a JavaScript function with the name ‘doFoo’, ensuring uniqueness on the portal page.

## param Tag

This tag defines a parameter that may be added to an actionURL, renderURL or resourceURL.

The param tag must not contain any body content.

If the param tag has an empty value the specified parameter name must be removed from the URL. In the case of a resource URL, an empty value does not alter the render parameters automatically added by the portlet container to resource URLs.

This tag adds render parameters to a render URL, resource parameters to a resource URL, and either action parameters or render parameters to an action URL depending on the type attribute.

If the same name of a parameter occurs more than once within an actionURL, renderURL or resourceURL the values must be delivered as parameter value array with the values in the order of the declaration within the URL tag.

The following attributes are defined for this tag:

* **name** (Type: String, required) – the name of the parameter to add to the URL. If name is null or empty, no action is performed.
* **value** (Type: String, required) – the value of the parameter to add to the URL. If value is null, it is processed as an empty value.
* **type** (Type: String, optional) – the type of the parameter to add to the URL. If specified, this attribute is ignored if the param tag is used with a render URL or resource URL.

If the param tag is used with an action URL, the attribute value may be either "action" or "render". The attribute value is case-insensitive. If the type is set to "render", a render parameter is added, and if the type is set to "action", an action parameter is added.

If the param tag is used with an action URL but the type attribute is not specified, type="action" is assumed.

An example of a JSP using the param tag could be:

<portlet:actionURL>

<portlet:param name="myParam" type="action" value="someValue"/>

</portlet:actionURL>

## property Tag

This tag defines a property that may be added to an actionURL, renderURL or resourceURL and is equivalent to the API call addProperty().

The property tag should not contain any body content.

If the same name of a property occurs more than once within an actionURL, renderURL or resourceURL, the values should be delivered as properties value array with the values in the order of the declaration within the URL tag.

The following *required attributes* are defined for this tag:

* **name** (Type: String, required) – the name of the property to add to the URL. If name is null or empty, no action is performed.
* **value** (Type: String, required) – the value of the property to add to the URL. If value is null, it is processed as an empty value.

An example of a JSP using the param tag could be:

<portlet:actionURL>

<portlet:property name="myProperty" value="someValue"/>

</portlet:actionURL>

## Changing the Default Behavior for escapeXml

The portlet container performs XML escaping on URLs by default. The default behavior can be changed using a portlet container runtime option. See Section 8.4.1 Runtime Option javax.portlet.escapeXml for more information.

# Packaging and Deployment

The deployment descriptor conveys the elements and configuration information of a portlet application between people who develop, assemble, and deploy portlet applications. Portlet applications are self-contained applications that are intended to work without further resources. Portlet applications are managed by the portlet container.

A portlet application is a web application, so the portlet application can have a web application deployment descriptor (web.xml) as defined by the Servlet Specification[[36]](#footnote-36).

The version 3.0 portlet container is required provide backward compatibility for version 1.0 and version 2.0 portlets, so it is also required to support the corresponding versions of the portlet and web application deployment descriptors as well.

All web resources that are not portlets must be declared through annotations or through the web.xml deployment descriptor as described in the Servlet Specification. The following portlet web application properties can be set in the web.xml deployment descriptor:

* portlet application description using the <description> element
* portlet application name using the <display-name> element
* portlet application security role mapping using the <security-role> element
* portlet application locale-character set mapping for serving resources using the <locale-encoding-mapping-list>.

Portlet configuration can be carried out through use of annotations or through the portlet deployment descriptor (portlet.xml) as described in Chapter 28 Configuration. If the portlet can be completely configured using annotations, the portlet deployment descriptor is optional.

The mechanism by which portlets are actually deployed on the portal is implementation specific and not subject of this document.

## Packaging

All resources, portlets, and the deployment descriptors are packaged together in one web application archive (WAR file). This format is described by the Servlet Specification36. A portlet application requires the following resources in addition to those defined by the Servlet Specification:

* The /WEB-INF/portlet.xml deployment descriptor if the configuration is not completely defined through annotations.
* Portlet classes in the /WEB-INF/classes directory.
* Portlet Java Archive files /WEB-INF/lib/\*.jar

### Example Directory Structure

The following lists all files in a sample portlet application:

/images/myButton.gif

/META-INF/MANIFEST.MF

/WEB-INF/web.xml

/WEB-INF/portlet.xml

/WEB-INF/lib/myHelpers.jar

/WEB-INF/classes/com/mycorp/servlets/MyServlet.class

/WEB-INF/classes/com/mycorp/portlets/MyPortlet.class

/WEB-INF/jsp/myHelp.jsp

Portlet applications that need additional resources that cannot be packaged in the WAR file, like EJBs, may be packaged together with these resources in an EAR file.

### Version Information

Portlet application providers can provide version information about the portlet application through META-INF/MANIFEST.MF entries in the WAR file. The ‘Implementation-\*’ attributes should be used to define the version information. The version information should follow the format defined by the Java Product Versioning Specification[[37]](#footnote-37). An example follows.

Implementation-Title: myPortletApplication

Implementation-Version: 1.1.2

Implementation-Vendor: SunMicrosystems. Inc.

## Portlet Deployment Descriptor Structure

This section describes the overall structure of the portlet deployment descriptor. The individual data elements contained therein are described in Chapter 28 Configuration.

Appendix E Deployment Descriptor Schema shows the complete Portlet Specification 3.0 deployment descriptor schema.

### Portlet Deployment Descriptor Elements

The following types of configuration and deployment information are required to be supported in the portlet deployment descriptor for all portlet containers:

* Portlet Application Definition
* Portlet Definition

Security information, which may also appear in the deployment descriptor is not required to be supported unless the portlet container is part of an implementation of the Java EE Specification.

### Rules for processing the Portlet Deployment Descriptor

In this section is a listing of some general rules that the portlet container must observe concerning the processing of the deployment descriptor for a portlet application:

* Portlet containers should ignore all leading whitespace characters before the first non-whitespace character, and all trailing whitespace characters after the last non-whitespace character for PCDATA within text nodes of a deployment descriptor.
* Portlet containers and tools that manipulate portlet applications have a wide range of options for checking the validity of a WAR. This includes checking the validity of the web application and portlet deployment descriptor documents held within. It is recommended, but not required, that portlet containers and tools validate both deployment descriptors against the corresponding DTD and XML Schema definitions for structural correctness. Additionally, it is recommended that they provide a level of semantic checking. For example, it should be checked that a role referenced in a security constraint has the same name as one of the security roles defined in the deployment descriptor. In cases of non-conformant portlet applications, tools and containers should inform the developer with descriptive error messages. High end application server vendors are encouraged to supply this kind of validity checking in the form of a tool separate from the container.
* In elements whose value is an "enumerated type", the value is case sensitive.

### Uniqueness of Deployment Descriptor Values

The following deployment descriptor values must be unique in the scope of the portlet application definition:

* portlet <portlet-name>
* custom-portlet-mode <portlet-mode>
* custom-window-state <window-state>
* user-attribute <name>
* event-definition <name> and <qname>
* public-render-parameter <name> and <qname>
* filter <filter-name>

The following deployment descriptor values must be unique in the scope of the portlet definition:

* init-param <name>
* supports <mime-type>
* preference <name>
* security-role-ref <role-name>
* <supported-processing-event>
* <supported-publishing-event>
* <supported-public-render-parameter>

### Localization of Deployment Descriptor Values

Localization of deployment descriptor values allows the deployment tool to provide localized deployment messages to the person deploying the portlet application. The following deployment descriptor elements may exist multiple times with different locale information in the xml:lang attribute:

* all <description> elements
* portlet <display-name>

The default value for the xml:lang attribute is English ("en"). Portlet-container implementations using localized values of these elements should treat the English ("en") values as the default fallback value for all other locales.

The preferred method for localization of values in the deployment descriptor is providing a resource bundle via the <resource-bundle> element on the portlet application level (see Resource Bundle section below).

## Deployment Descriptor Example

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

<portlet-app xmlns="http://xmlns.jcp.org/xml/ns/portlet" version="3.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/portlet

http://xmlns.jcp.org/xml/ns/portlet/portlet-app\_3\_0.xsd">

<portlet>

<description xml:lang="en">Portlet displaying the time in different time zones</description>

<description xml:lang="de">Dieses Portlet zeigt die Zeit in verschiedenen Zeitzonen an. </description>

<portlet-name>TimeZoneClock</portlet-name>

<display-name xml:lang="en">Time Zone Clock Portlet</display-name>

<display-name xml:lang="de">ZeitzonenPortlet</display-name>

<portlet-class>com.myco.samplets.util.zoneclock.ZoneClock</portlet-class>

<expiration-cache>60</expiration-cache>

<supports>

<mime-type>text/html</mime-type>

<portlet-mode>config</portlet-mode>

<portlet-mode>edit</portlet-mode>

<portlet-mode>help</portlet-mode>

</supports>

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

<portlet-info>

<title>Time Zone Clock</title>

<short-title>TimeZone</short-title>

<keywords>Time, Zone, World, Clock</keywords>

</portlet-info>

<portlet-preferences>

<preference>

<name>time-server</name>

<value>http://timeserver.myco.com</value>

<read-only>true</read-only>

</preference>

</portlet-preferences>

<security-role-ref>

<role-name>trustedUser</role-name>

<role-link>auth-user</role-link>

</security-role-ref>

</portlet>

<custom-portlet-mode>

<description xml:lang="en">Pre-defined custom portlet mode CONFIG</description>

<portlet-mode>CONFIG</portlet-mode>

</custom-portlet-mode>

<custom-window-state>

<description xml:lang="en">Occupies 50% of the portal page</description>

<window-state>half-page</window-state>

</custom-window-state>

<user-attribute>

<description xml:lang="en">Pre-defined attribute for the telephone number of the user at work.</description>

<name>workInfo/telephone</name>

</user-attribute>

<security-constraint>

<portlet-collection>

<portlet-name>TimeZoneClock</portlet-name>

</portlet-collection>

<user-data-constraint>

<transport-guarantee>CONFIDENTIAL</transport-guarantee>

</user-data-constraint>

</security-constraint>

</portlet-app>

# Configuration

The configuration data can be provided through annotations or through the portlet deployment descriptor.

When values are provided through both the deployment descriptor and through annotations, the values from the deployment descriptor take precedence. The values are merged at the item level where possible. For example, if a configuration annotation provides initialization parameters ip1 and ip2, and the deployment descriptor overrides the initialization parameter ip2, the resulting configuration will have the initialization parameters ip1 with the value specified by the annotation and ip2 with the value specified by the deployment descriptor.

This chapter presents all configuration data and provides both annotation and deployment descriptor usage examples.

The configuration contains information described at the portlet application level that applies to all portlets in the portlet application and information described at the portlet level that applies only to the specific portlet. The following section refer to these two types of configuration data as the *portlet application configuration* and *portlet configuration*, respectively.

The portlet deployment descriptor allows specification of portlet application configuration within the <portlet-app> element. The @PortletApplication annotation contains many of the portlet application configuration elements provided by the <portlet-app> element. However, some of the portlet application configuration elements, such as those defining portlet filters and portlet URL generation listeners, are mapped to other configuration annotations as will be discussed below.

The <portlet-app> element also contains <portlet> child elements for each portlet in the portlet application.

The <portlet> element within the portlet deployment descriptor contains the portlet-specific configuration data. The @PortletConfiguration annotation contains many of the portlet configuration elements provided by the <portlet> element. However, some of the portlet configuration elements, such as those defining portlet events, are mapped to other configuration annotations as will be discussed below.

## Portlet Application Configuration

This section describes the portlet application configuration.

Since the <portlet-app> element encloses the portlet definitions within the portlet deployment descriptor, it must always be present. However, the configuration items contained within the <portlet-app> element are optional and need only appear when needed.

The @PortletApplication is a type annotation that can be applied to any class in the portlet application. It may appear at most once within the portlet application.

If the portlet provides a portlet deployment descriptor <portlet-app> element in addition to the @PortletApplication annotation, the portlet container must merge the portlet application configuration values with the configuration values provided through the @PortletApplication annotation. In cases where the same type of configuration data (such as a portlet container runtime option with a specific key) is provided in both the @PortletApplication annotation and the portlet deployment descriptor <portlet-app> element, the value from the portlet deployment descriptor must take precedence.

The @PortletApplication does not contain the individual @PortletConfiguration elements, so it is optional, and need only be used when the configuration items contained therein are needed.

### Portlet API Version

The version field declares the portlet API used by the portlet application. It is specified as an element attribute in the portlet deployment descriptor <portlet-app> element, and in the @PortletApplication annotation version element. The value must be "3.0" to obtain the functionality described in this specification.

Portlet deployment descriptor example:

<portlet-app xmlns="http://xmlns.jcp.org/xml/ns/portlet"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:rp=

"http://www.apache.org/portals/pluto/pub-render-params/ResourcePortlet"

xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/portlet http://xmlns.jcp.org/xml/ns/portlet/portlet-app\_3\_0.xsd"

version="3.0">

...

</portlet-app>

Annotation example:

The @PortletApplication version element is set to the value "3.0" by default and needs not be further declared.

### Custom Portlet Mode

The custom portlet mode configuration data consists of the mandatory custom portlet mode name, an optional flag that indicates whether the custom portlet mode is managed by the portal or by the portlet, and an optional set of localized description strings. Each custom portlet mode name must be unique within the configuration.

Within the portlet deployment descriptor <custom-portlet-mode> element, this data is represented by the <portlet-mode>, <portlet-managed>, and <description> elements, respectively.

Deployment descriptor usage example declaring an admin custom portlet mode:

<portlet-app>

...

<custom-portlet-mode>

<description>Provides administration functions</description>

<description xml:lang="de">Verwaltungsfunktionen</description>

<portlet-mode>admin</portlet-mode>

<portal-managed>true</portal-managed>

</custom-portlet-mode>

</portlet-app>

The @PortletApplication annotation uses an array of CustomPortletMode elements to define the custom portlet modes. Each defined CustomPortletMode element must contain a mandatory name element to define the custom portlet mode name, an optional boolean portalManaged flag that defaults to true, and an optional array of localized description elements.

Annotation usage example declaring an admin custom portlet mode:

@PortletApplication(customPortletModes={

@CustomPortletMode(description={

@LocaleString("Provides administration functions"),

@LocaleString(locale="de", value="Verwaltungsfunktionen")

},

name = "admin",

portalManaged=true)

})

### Custom Window State

The custom window state configuration data consists of the mandatory custom window state name and an optional set of localized description strings.

Each custom window state name must be unique within the configuration.

Within the portlet deployment descriptor <custom-window-state> element, this data is represented by the <window-state> and <description> elements, respectively.

Deployment descriptor usage example declaring a half\_page custom window state:

<portlet-app>

...

<custom-window-state>

<description>Occupies 50% of the portal page</description>

<window-state>half\_page</window-state>

</custom-window-state>

...

</portlet-app>

The @PortletApplication annotation uses an array of CustomWindowState elements to define the custom window states. Each defined CustomWindowState element must contain a mandatory name element to define the custom portlet mode name and an optional array of localized description elements.

Annotation usage example declaring an admin custom portlet mode:

@PortletApplication(

customWindowStates = {

@CustomWindowState(description = {

@LocaleString("Occupies 50% of the portal page"),

@LocaleString(locale="de", value =

"Verwendet 50% der Portal-Seite")

},

name = "half\_page")

})

### User Attribute

The portlet application configuration should define the user attribute names used by the portlets in the portlet application.

When the portlet is deployed, the logical user attributes must be mapped to the corresponding user attributes offered by the runtime environment. The manner in which this mapping is performed is implementation-specific and outside the scope of this specification.

At runtime, the portlet container uses this mapping to allow portlets to access the user attributes. User attributes of the runtime environment not mapped as part of the deployment process should not be exposed to portlets.

Refer to Appendix C User Attribute Names on page 212 for a list of recommended names.

The portlet deployment descriptor provides the <user-attribute> element with its child elements <description> and <name> for user attribute declaration.

Portlet deployment descriptor usage example:

<portlet-app>

…

<user-attribute>

<description>User Given Name</description>

<name>user.name.given</name>

</user-attribute>

<user-attribute>

<description>User Last Name</description>

<name>user.name.family</name>

</user-attribute>

<user-attribute>

<description>User eMail</description>

<name>user.home-info.online.email</name>

</user-attribute>

<user-attribute>

<description>Company Organization</description>

<name>user.business-info.postal.organization</name>

</user-attribute>

…

<portlet-app>

Annotation usage example:

@PortletApplication(

userAttributes = {

@UserAttribute(

description = @LocaleString("User Given Name"),

name = "user.name.given"

),

@UserAttribute(

description = @LocaleString("User Last Name"),

name = "user.name.family"

),

@UserAttribute(

description = @LocaleString("User eMail"),

name = "user.home-info.online.email"

),

@UserAttribute(

description = @LocaleString("Company Organization"),

name = "user.business-info.postal.organization"

),

},

)

### Resource Bundle

As an alternative to embedding all localized values in the deployment descriptor or the configuration annotations, the portlet can provide a separate resource bundle containing the localized values. Providing localized values via resource bundles is preferred, as it allows the separation of deployment descriptor values from localized values.

The Portlet Specification defines localized information at both the portlet application and the portlet level. This section describes the localized information for the portlet application level. A later section will describe the localized information at the portlet level.

The Java Portlet Specification defines the following constants for the application level resource bundle:

|  |  |
| --- | --- |
| javax.portlet.app.custom-portlet-mode. <portlet-mode>.description | Description of custom portlet mode <portlet-mode>. |
| javax.portlet.app.custom-window-state. <window-state>.description | Description of the custom window state <window-state>. |
| javax.portlet.app.user-attribute. <name>.description | Description of the user attribute <name>. |
| javax.portlet.app.event-definition. <name>.description | Description of the event <name>. <name> uses the string representation of the Java QName class with {namespace}localpart[[38]](#footnote-38). |
| javax.portlet.app.event-definition. <name>.display-name | Name under which this event is displayed to users or to tools. The display name need not be unique. <name> uses the string representation of the Java QName class with {namespace}localpart38. |
| javax.portlet.app. public-render-parameter. <name>.description | Description of the public render parameter <name>. |
| javax.portlet.app. public-render-parameter. <name>.display-name | Name under which this public render parameter is displayed to users or to tools. The display name need not be unique. |

For language-specific portlet application level information, the fully qualified class name of the resource bundle can be set in the deployment descriptor using the <resource-bundle> element on the portlet application level. The @PortletApplication annotation provides the resourceBundle element for setting the resource bundle.

Portlet deployment descriptor usage example:

<portlet-app>

...

<resource-bundle>com.acme.portlets.DisplayPortlet</resource-bundle>

...

</portlet-app>

Annotation usage example:

@PortletApplication(

resourceBundle="com.acme.portlets.DisplayPortlet",

)

### Filter Configuration

The portlet deployment descriptor allows for filter configuration through use of the <filter> and <filter-mapping> elements. The @PortletApplication annotation does not possess corresponding elements. Instead, the @PortletLifecycleFilter annotation combines the function of both the <filter> and <filter-mapping> elements.

See Section 16.2.4 Filter Configuration on page 102 for filter configuration discussion and examples.

### Default Namespace URI

The Portlet Specification makes use of qualified names (QNames)[[39]](#footnote-39) to uniquely identify portlet events and public render parameters. The qualified name consists of a local part and a namespace URI. When an event or public render parameter is declared, both the local part and the namespace URI must be provided.

Instead of specifying a namespace URI each time an event or public render parameter is declared, the developer can declare a default namespace URI through the portlet deployment descriptor <default-namespace> element or through the @PortletApplication defaultNamespaceURI element. If the default namespace URI is specified, the portlet container must use the provided value as the default for the namespace URI portion of qualified names identifying portlet events and public render parameters.

The default namespace URI may be configured a maximum of one time.

Portlet deployment descriptor usage example:

<portlet-app>

...

<default-namespace>

<http://www.apache.org/portals/portlets/DemoPortlet>

</default-namespace>

...

</portlet-app>

Annotation usage example:

@PortletApplication(

defaultNamespaceURI="http://www.apache.org/portals/portlets/DemoPortlet",

)

### Event Configuration

The portlet should declare events at the application level to make the event payload types known to the portlet container and to allow a description to be provided. The event configuration data at the application level consists of the mandatory event qualified name39, the optional event payload type, optional event qualified name aliases, the optional localized display name strings, and the optional localized description strings.

The portlet container must enforce uniqueness of the event qualified name at the portlet application level.

The event configuration is continued at the portlet configuration level, where the specific events published and processed by the portlet are declared.

Portlet container or portal defined events do not need to be declared on the application level, but can be directly referenced at the portlet configuration level.

For further discussion about events, see Chapter 17 Coordination between Portlets on page 106.

The portlet deployment descriptor provides the <event-definition> element for event declaration.

Within the <event-definition> element, either the <name> element or the <qname> element can be used to specify the qualified name. The <qname> element allows specification of both the qualified name local part and the namespace URI. The <name> element allows the qualified name local part to be specified. If the <name> element is used, the portlet container must take the namespace URI for the portlet event qualified name to be the default namespace URI (discussed above) if it is defined, otherwise it must take the namespace URI to be the constant value javax.xml.XMLConstants.NULL\_NS\_URI.

Portlet developers are encouraged to organize the local part of the event names declared in the event definitions in a hierarchical manner using the dot '.' as separator. The portlet must not specify events with the same name but different types. The event name local part should not end with a trailing '.' character as wildcards are not supported in the event definition.

Within the <event-definition> element, the <value-type> element configures the event payload, and may appear a maximum of one time. The <description>, <display-name>, and <alias> elements configure the localized description strings, the localized display name strings, and the alias qualified names, respectively, for the event, and can be repeated any required number of times.

Portlet deployment descriptor usage example:

<portlet-app>

...

<event-definition>

<qname xmlns:x="http:example.com/events">x:foo.bar</qname>

<value-type>com.example.Address</value-type>

</event-definition>

...

</portlet-app>

The @PortletApplication annotation provides the events element for event declaration which is an array of EventDefinition objects. The EventDefinition object contains corresponding data elements for declaration of a single event.

Annotation usage example:

@PortletApplication(

events = {

@EventDefinition(

qname = @PortletQName(

namespaceURI = "http:example.com/events",

localPart = "foo.bar"),

payloadType = Address.class)

},

)

### Public Render Parameters

The portlet must declare public render parameters used within the portlet application at the application level to make the public render parameter identifier used within the portlet application known to the portlet container. The public render parameter identifier is the render parameter name under which the portlet can access the public render parameter.

The public render parameter configuration data at the application level consists of the mandatory public render parameter qualified name39, the mandatory identifier, the optional localized display name strings, and the optional localized description strings.

The portlet container must enforce uniqueness of the public render parameter qualified name at the application level.

The public render parameter configuration is continued at the portlet configuration level, where the specific public render parameters used by each portlet are defined.

For further discussion about events, see Section 11.2.1 Public Render Parameters on page 64.

The portlet deployment descriptor provides the <public-render-parameter> element for public render parameter declaration.

Within the <public-render-parameter> element, either the <name> element or the <qname> element can be used to specify the qualified name. The <qname> element allows specification of both the qualified name local part and the namespace URI. The <name> element allows the qualified name local part to be specified. If the <name> element is used, the portlet container must take the namespace URI to be the default namespace URI (discussed above) if it is defined, otherwise it must take the namespace URI to be the constant value javax.xml.XMLConstants.NULL\_NS\_URI.

Also, the <identifier> element configures the public render parameter identifier, and may appear a maximum of one time. The <description> and <display-name> elements configure the localized description strings and the localized display name strings, respectively, for the public render parameter, and can be repeated any required number of times.

Portlet deployment descriptor usage example:

<portlet-app>

...

<public-render-parameter>

<identifier>imgName</identifier>

<qname xmlns:rp="http:example.com/PRPs">rp:imgName</qname>

</public-render-parameter>

...

</portlet-app>

The @PortletApplication annotation provides the publicParams element, which is an array of PublicRenderParameterDefinition objects, for public render parameter declaration. The PublicRenderParameterDefinition object contains corresponding data elements for declaration of a single public render parameter.

Annotation usage example:

@PortletApplication(

publicParams = {

@PublicRenderParameterDefinition(

qname = @PortletQName(

namespaceURI = "http:example.com/PRPs",

localPart = "imgName"),

identifier = "imgName"

),

},)

### Portlet URL Generation Listener

The portlet deployment descriptor allows for portlet URL generation listener configuration through use of the <listener> element. The @PortletApplication annotation does not possess corresponding elements. Instead, the @PortletURLGenerationListener annotation provides the function of the <listener> element and its children.

See Section 13.6 The Portlet URL Generation Listener on page 75 for portlet URL generation listener configuration discussion and examples.

### Portlet Container Runtime Options

Portlet container runtime options are name-value pairs that influence the operation of the portlet container. See Section 8.4 Portlet Container Runtime Options for discussion. Portlet container runtime options can be configured on the portlet application level, where they affect all portlets in the portlet application, or on the portlet level, where they affect only the specific portlet. Configuration at the portlet level will be discussed in a later section.

The portlet deployment descriptor allows configuration of portlet runtime options through the <container-runtime-option> element, which has mandatory <name> and <value> child elements. The @PortletApplication annotation allows configuration of the portlet container runtime options through the runtimeOption element, which may contain an array of @RuntimeOption annotations, each one specifying a name-value pair.

A given portlet container runtime option name may have multiple values associated with it. Each value is specified with a separate <value> element in the deployment descriptor and through use of the values array in the corresponding configuration annotation.

The portlet container must enforce uniqueness of the portlet container runtime option keys within the portlet application.

Portlet deployment descriptor usage example:

<portlet-app>

...

<container-runtime-option>

<name>javax.portlet.renderHeaders</name>

<value>true</value>

</container-runtime-option>

...

</portlet-app>

Annotation usage example:

@PortletApplication(

runtimeOptions = {

@RuntimeOption(name = "javax.portlet.renderHeaders", values = {"true"})

},

)

## Portlet Configuration

The @PortletConfiguration annotation provides configuration data for the portlet. It is a type annotation that can be applied to any class in the portlet application. It has a single required portletName element that specifies the portlet name and many optional elements for configuration data. The optional elements provide defaults that should be applicable in many situations.

The portlet container must enforce uniqueness of the portlet name within the portlet application.

If @PortletConfiguration annotates a class that directly or indirectly implements the Portlet interface, the portlet container must register the class as a portlet class under the given portlet name, and call all of the implemented portlet lifecycle methods from the Portlet, EventPortlet, and ResourceServingPortlet interfaces as described in Chapter 4 Portlet Lifecycle Interfaces. An example annotated portlet class:

@PortletConfiguration(portletName="TestPortlet4")

public class AnnotatedGenericPortlet extends GenericPortlet {

// add portlet methods

}

The portlet class may be both annotated with a @PortletConfiguration annotation and specified in one or more deployment descriptor portlet definitions as long as different portlet names are provided for each declared portlet.

If @PortletConfiguration annotates a class that does not implement the Portlet interface, the portlet container must register the configuration for the given portlet name. Since the portlet lifecycle methods defined by the Portlet interface are not available, the portlet must use the extended annotation-based dispatching functionality as described in Section 4.8 Extended Annotation-Based Dispatching. An example extended annotation-based dispatching class:

@PortletConfiguration(portletName="TestPortlet3")

public class MessageBoxPortlet {

}

If the same portlet name is specified in both the @PortletConfiguration annotation and in a portlet deployment descriptor <portlet> element, the portlet container aggregate the configuration values from the portlet deployment descriptor with those provided through the @PortletConfiguration annotation, with the values from the portlet deployment descriptor taking precedence. For example, the value of a portlet initialization parameter with a given key for a given portlet name defined in the portlet deployment descriptor must take precedence over a value for the same portlet initialization parameter key and portlet name provided through the @PortletConfiguration annotation.

However, when configuration data is provided through the @PortletConfiguration annotation, that annotation determines the portlet class or the portlet dispatching methods as described above. The portlet class or dispatching methods cannot be overridden through portlet deployment descriptor <portlet> entry with the same portlet name. In this case, the <portlet-class> element should not be present. If the portlet deployment descriptor <portlet-class> element is present, the portlet container must recognize the configuration error, but is free to either ignore the portlet deployment descriptor <portlet-class> element or reject the portlet at deployment time.

The portlet container must support configuration of multiple portlets within a portlet application through use of the @PortletConfiguration annotation applied to different portlet classes.

### @PortletConfigurations Annotation

The developer may sometimes need to configure more than one portlet for a given portlet class. Since Java SE Version 7 and earlier does not allow multiple annotations of the same type to be present on the same annotated artifact, the Portlet Specification provides the @PortletConfigurations annotation for that purpose.

The @PortletConfigurations annotation contains an array of @PortletConfiguration annotations. The portlet container must process each @PortletConfiguration annotation in the array in the manner described above, registering a portlet for each array element.

If the portlet application is using extended annotation-based dispatching rather than the portlet lifecycle methods provided by the Portlet interface, the @PortletConfigurations annotation can be used to consolidate portlet configuration information into a common location.

The portlet container must support use of the @PortletConfigurations annotation multiple times within a portlet application.

### Portlet Container Runtime Options

As discussed in a previous section, portlet container runtime options can be configured on the portlet application level, where they affect all portlets in the portlet application, or on the portlet level, where they affect only the specific portlet. Portlet container runtime options configured at the portlet level override portlet container runtime options of the same name configured at the portlet application level.

The portlet deployment descriptor allows configuration of portlet runtime options through the <container-runtime-option> element, which has mandatory <name> and <value> child elements. The @PortletConfiguration annotation allows configuration of the portlet container runtime options through the runtimeOption element, which contains a corresponding array of name-value pairs.

The portlet container must enforce uniqueness of the portlet container runtime option keys for the portlet.

Portlet deployment descriptor usage example:

<portlet>

...

<container-runtime-option>

<name>javax.portlet.renderHeaders</name>

<value>true</value>

</container-runtime-option>

...

</portlet>

Annotation usage example:

@PortletConfiguration(

runtimeOptions = {

@RuntimeOption(name = "javax.portlet.renderHeaders", values = {"true"})

},

)

### Portlet Initialization Parameters

Portlet initialization parameters are name-value pairs for use by the portlet. The portlet can access the portlet initialization parameters through the PortletConfig object. See Section 6.1 Initialization Parameters. Each portlet initialization parameter name-value pair may also have an associated array of localized descriptions strings..

The portlet container must enforce uniqueness of the portlet initialization parameter names for the portlet. The name may not be null.

The portlet deployment descriptor provides the <init-param> element for portlet initialization parameter declaration. It in turn contains the localized <description> element, which can be repeated as necessary, the <name> element for specification of the name, and the <value> element for specification of the value.

Portlet deployment descriptor usage example:

<portlet>

…

<init-param>

<name>city</name>

<value>New York</value>

</init-param>

<init-param>

<name>zip</name>

<value>10001</value>

</init-param>

</portlet>

The @PortletConfiguration annotation provides the initParams element, which contains an array of InitParameter elements, each of which contains corresponding fields for the localized description, the name, and the value.

Annotation usage example:

@PortletConfiguration(portletName="TestPortlet4", initParams={

@InitParameter(name="city", value="New York"),

@InitParameter(name="zip", value="10001")

})

### Portlet Identification

The portlet configuration contains information to identify and categorize the portlet. It consists of the mandatory portlet name, the optional portlet class, and the optional localized display name, description, title, short title, and keywords strings. See Section 6.2 Portlet Resource Bundle for additional information on the latter three items.

Within the portlet deployment descriptor, the portlet name, portlet class, display name, and description strings are configured through the <portlet> child elements <portlet-name>, <portlet-class>, <display-name>, and <description>, respectively.

The title, short title, and keyword strings are configured through the <portlet-info> child elements <title>, <short-title>, and <keyword>, respectively. The <portlet-info> element is contained within the <portlet> element. Portlet deployment descriptor usage example:

<portlet>

<description xml:lang="en">

Portlet displaying the time in different time zones</description>

<description xml:lang="de">

Dieses Portlet zeigt die Zeit in verschiedenen Zeitzonen an.

</description>

<portlet-name>TimeZoneClock</portlet-name>

<display-name xml:lang="en">Time Zone Clock Portlet</display-name>

<display-name xml:lang="de">ZeitzonenPortlet</display-name>

<portlet-class>com.myco.samplets.util.zoneclock.ZoneClock</portlet-class>

...

<portlet-info>

<title>Time Zone Clock</title>

<short-title>TimeZone</short-title>

<keywords>Time, Zone, World, Clock</keywords>

</portlet-info>

</portlet>

The @PortletConfiguration annotation provides the portletName, description, displayName, title, shortTitle, and keywords elements for configuration of the identification items. The portlet class is determined as described in Section 28.2 Portlet Configuration.

Annotation usage example:

@PortletConfiguration(portletName="TimeZoneClock",

description={

@LocaleString("Portlet displaying the time in different time zones"),

@LocaleString(locale="de",

value="Dieses Portlet zeigt die Zeit in verschiedenen Zeitzonen an")

}, displayName={

@LocaleString("Time Zone Clock Portlet"),

@LocaleString(locale="de", value="ZeitzonenPortlet")

}, title={

@LocaleString("Time Zone Clock")

}, shortTitle={

@LocaleString("TimeZone")

}, keywords={

@LocaleString("Time, Zone, World, Clock")

}

)

### Portlet Resource Bundle

Resource bundles can be used to provide language specific portlet-level information, like title and keywords. The fully qualified class name of the resource bundle can be set in the portlet definition in the deployment descriptor using the resource-bundle element.

The Java Portlet Specification defines the following constants for the portlet level resource bundle:

|  |  |
| --- | --- |
| javax.portlet.title | The title that should be displayed in the title bar of this portlet. Only one title per locale is allowed. Note that this title may be overridden by the portal or programmatically by the portlet. |
| javax.portlet.short-title | A short version of the title that may be used for devices with limited display capabilities. Only one short title per locale is allowed. |
| javax.portlet.keywords | Keywords describing the functionality of the portlet. Portals that allow users to search for portlets based on keywords may use these keywords. Multiple keywords per locale are allowed, but must be separated by commas ‘,’. |
| javax.portlet.description | Description of the portlet. |
| javax.portlet.display-name | Name under which this portlet is displayed at deployment time or to tools. The display name need not be unique. |
| javax.portlet.app.custom-portlet-mode.<name>.decoration-name | Decoration name for the portlet managed custom portlet mode <name>. |

For language-specific portlet level information, the fully qualified class name of the resource bundle can be set in the deployment descriptor within the <portlet> declaration using the <resource-bundle> element. The @PortletConfiguration annotation provides the resourceBundle element for setting the resource bundle.

Portlet deployment descriptor usage example:

<portlet>

...

<resource-bundle>com.acme.portlets.DisplayPortlet</resource-bundle>

...

</portlet>

Annotation usage example:

@PortletConfiguration(

resourceBundle="com.acme.portlets.DisplayPortlet",

)

#### Resource Bundle Example

This section shows the resource bundles for the world population clock portlet from the deployment descriptor example. The first resource bundle is for English and the second for German locales.

English Resource Bundle:

#

# filename: clock\_en.properties

# Portlet Info resource bundle example

javax.portlet.title=World Population Clock

javax.portlet.short-title=WorldPopClock

javax.portlet.keywords=World,Population,Clock

German Resource Bundle:

#

# filename: clock\_de.properties

# Portlet Info resource bundle example

javax.portlet.title=Weltbevölkerungsuhr

javax.portlet.short-title=Weltuhr

javax.portlet.keywords=Welt,Bevölkerung,Uhr

### Cache Settings

The portlet configuration allows declaration of the cache expiration and cache scope values supported by the portlet as defined in Chapter 23 Caching on page 152.

In the following examples, the content should be cached for 5 minutes (300 seconds) and must not be shared across users.

The portlet deployment descriptor provides the <expiration-cache> and <cache-scope> elements for cache setting configuration.

Portlet deployment descriptor usage example:

<portlet>

...

<expiration-cache>300</expiration-cache>

<cache-scope>private</cache-scope>

...

</portlet>

The @PortletConfiguration annotation provides the cacheExpirationTime and cacheScopePublic elements for cache setting configuration. If the cacheScopePublic element is set to true, cached information can be shared between users. The default is false.

Annotation usage example:

@PortletConfiguration(

cacheExpirationTime = 300,

cacheScopePublic = false,

)

### Security Role Reference

The portlet configuration allows declaration of security role references as described in Section 24.3 Programmatic Security on page 154. A security role reference contains a mandatory role name and an optional role link. The security role reference can be repeated as many times as needed.

The portlet container must enforce uniqueness of the security role names within the portlet configuration.

The portlet deployment descriptor provides the <security-role-ref> element with its child elements <role-name> and <role-link> elements for security role reference configuration.

Portlet deployment descriptor usage example:

<portlet>

...

<security-role-ref>

<role-name>FOO</role-name>

<role-link>manager</role-link>

</security-role-ref>

...

</portlet>

The @PortletConfiguration annotation provides the roleRefs for role reference configuration.

Annotation usage example:

@PortletConfiguration(

roleRefs = @SecurityRoleRef(roleName="FOO", roleLink="manager"),

)

### Page Resource Dependencies

The portlet configuration allows declaration of dependencies the portlet may have on page resource resources. The resources represent client-side prerequisites such as JavaScript libraries or stylesheet resources that are shared among portlets. A dependency is declared through mandatory resource name and minimum acceptable version strings. Generally the name should be the known common name for the resource, such as angular or bootstrap, and the version string should be in a format appropriate to the resource, although the mechanism can be used for proprietary portal or application resources as well.

When dependencies are configured, the portal system should take the appropriate measures to make the resources available to the portlet on the client.

The way the portal system is configured to provide the resources and the way it actually provides the resources to the portlet are beyond the scope of this specification.

The portlet deployment descriptor provides the <dependency> element and its child elements <name>, <scope> and <version> for dependency configuration. The <dependency> element can be repeated as required.

Portlet deployment descriptor usage example:

<portlet>

...

<dependency>

<name>jQuery</name>

<scope>com.jquery</scope>

<version>2.1.1</version>

</dependency>

...

</portlet>

The @PortletConfiguration annotation provides the dependencies element for dependency configuration.

Annotation usage example:

@PortletConfiguration(

dependencies = @Dependency(name = "jQuery", scope="com.jquery", version = "2.1.1"),

)

### Public Render Parameter References

The portlet developer must declare the public render parameter definitions used in the portlet application in the portlet application configuration as previously described. Each individual portlet within the portlet application must declare the public render parameters it uses by referencing the identifiers from the public render parameter definitions.

The portlet container must assure that the public render parameter identifiers declared in the portlet configuration are valid and are declared in the portlet application configuration. If this is not the case, the portlet container should reject the portlet at deployment time and provide the user with an appropriate error message.

The portlet deployment descriptor <portlet> element provides the <supported-public-render-parameter> child element for this purpose.

Portlet deployment descriptor usage example:

<portlet>

...

<supported-public-render-parameter>

imgName

</supported-public-render-parameter>

...

</portlet>

The @PortletConfiguration annotation provides the publicParams element for declaring the public render parameters.

Annotation usage example:

@PortletConfiguration(

publicParams = {"imgName"},

)

Public render parameters can also be declared implicitly for a portlet by using a @RenderStateScoped bean. Such beans are represented in the render state as a render parameter. See Section 20.2.3 Render State Scope on page 120 for more information.

The @RenderStateScoped annotation allows a parameter name to be specified. If the parameter name matches a public render parameter identifier as declared in the portlet application configuration, the portlet container must treat the render parameter representing the bean as a public render parameter for all portlets within the portlet application in which the bean is used.

A portlet that uses such an @RenderStateScoped bean will be implicitly using a public render parameter. In this case, the public render parameter identifier is not required to be explicitly declared in the @PortletConfiguration publicParams element.

Annotation usage example:

@RenderStateScoped(paramName = "imgName")

public class ImageBean implements PortletSerializable {

}

### Event References

The portlet developer must declare the event definitions used in the portlet application in the portlet application configuration as previously described. Each individual portlet within the portlet application must declare the events it can process and those it publishes by referencing the qualified names from the event definitions.

The portlet deployment descriptor provides the <supported-publishing-event> and <supported-processing-event> elements for publishing and processing event configuration. Within these elements, either the <name> element or the <qname> element can be used to specify the qualified name. The <qname> element allows specification of both the qualified name local part and the namespace URI. The <name> element allows the qualified name local part to be specified. If the <name> element is used, the portlet container must take the namespace URI for the portlet event qualified name to be the default namespace URI if it is defined, otherwise it must take the namespace URI to be the constant value javax.xml.XMLConstants.NULL\_NS\_URI.

The event declaration within the portlet configuration supports the dot '.' as a wildcard character on the qualified name local part. A trailing '.' tells the portlet container that the portlet is interested in all events with names in this branch of the hierarchy. The portlet container must not treat a '.' character appearing within a qualified name local part at a location other than the end of the string as a wildcard character.

The portlet container should be able to resolve a portlet event declaration in the portlet configuration ending with the wildcard character to an event definition without wildcards by matching event name local parts ending with a '.' character to any event definition whose event name local part starts with the characters before the '.' character specifying the same namespace. If this is not the case, the portlet container must recognize the configuration error and not place the portlet in service.

Portlet deployment descriptor usage example:

<portlet>

...

<supported-publishing-event>

<qname xmlns:x="http:example.com/events">x:foo.bar</qname>

</supported-publishing-event>

<supported-processing-event>

<qname xmlns:x="http:example.com/events">x:foo.bar</qname>

</supported-processing-event>

...

</portlet>

The @PortletConfiguration annotation does not provide means for portlet event configuration.

Instead, the @ActionMethod annotation applied to the action processing method provides the publishingEvents element for configuring the events published by the action method. The @EventMethod annotation applied to the event processing method provides the publishingEvents and processingEvents elements for configuring the events published and processed by the event method. See Sections 4.7.2 Action Dispatching on page 31 and 4.7.3 Event Dispatching on page 32 for more information on these annotations.

@ActionMethod usage example:

@ActionMethod(portletName = "BeanPortletA",

publishingEvents = @PortletQName(

localPart="foo.bar",

namespaceURI = "http:example.com/events"))

public void setPrefs(ActionRequest request, ActionResponse response) {

...

}

@EventMethod usage example:

@EventMethod(portletName = "BeanPortletA",

processingEvents = @PortletQName(

localPart="Message",

namespaceURI = "http:example.com/events"),

publishingEvents = @PortletQName(

localPart="foo.bar",

namespaceURI = "http:example.com/events"))

public void processEvent(EventRequest request, EventResponse response)

throws PortletException ,IOException {

...

}

### Supported Locales

The portlet configuration allows declaration of the locales supported by the portlet. The locale is specified as a language tag as defined in IETF BCP 47[[40]](#footnote-40). The language tag may contain wildcard characters as described in IETF BCP 47.

The portlet deployment descriptor provides the <supported-locale> element for locale declaration. It can be repeated as often as required.

Portlet deployment descriptor usage example:

<portlet>

...

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

<supported-locale>de</supported-locale>

...

</portlet>

The @PortletConfiguration annotation supportedLocale element allows configuration of the supported locales.

Annotation usage example:

@PortletConfiguration(portletName="ConfigTestPortlet",

supportedLocales = {"en", "de"},

)

### Portlet Modes and Window States

The portlet configuration allows the portlet modes and custom window supported by the portlet for each markup type to be specified according to the rules defined in Section 9.5 Defining Portlet Modes Support and Section 10.5 Defining Window State Support.

The supported portlet modes and window states must be defined for each MIME type that the portlet supports. The portlet modes and window states are defined using a supports element which contains the target MIME type along with the associated portlet modes and window states. The supports elements are not additive. In order to allow a portlet mode or window state defined by annotation to be disallowed through the deployment descriptor, a supports element appearing later in the deployment descriptor or @PortletConfiguration annotation will completely override a previous supports element for a given MIME type.

Portlet mode and window state strings specified in the configuration are not case-sensitive.

The portlet deployment descriptor allows the <supports> element to be specified once for each supported content type. Within the <supports> element, the <mime-type> element must be configured once to declare the supported content type. The <portlet-mode> and <window-state> elements can be repeated as necessary to declare the supported portlet modes and window states.

Portlet deployment descriptor usage example:

For HTML markup, this portlet supports the HALF-PAGE window state in addition to the required pre-defined window states. For WML markup, it supports only the pre-defined window states.

<portlet>

...

<supports>

<mime-type>text/html</mime-type>

<portlet-mode>edit</portlet-mode>

<portlet-mode>help</portlet-mode>

<window-state>half-page</window-state>

...

</supports>

<supports>

<mime-type>text/vnd.wap.wml</mime-type>

<portlet-mode>help</portlet-mode>

...

</supports>

...

</portlet>

The @PortletConfiguration annotation supports element provides corresponding fields for portlet mode and window state configuration. The mimeType element default is "text/html".

Annotation usage example:

@PortletConfiguration(portletName="ConfigTestPortlet",

supports = {

@Supports(portletModes={"help", "edit"},

windowStates="half\_page"),

@Supports(mimeType="text/vnd.wap.wml", portletModes="help")

})

### Portlet Preferences

Portlet preferences are name-value pairs that the portlet can use to store persistent preference data, where the values are string arrays. The portlet can access the portlet initialization parameters through the PortletRequest object. See Chapter 18 Portlet Preferences for more information.

The portlet configuration can provide portlet preference names along with their corresponding values read-only flags.

The portlet container must enforce uniqueness of the portlet preference names for the portlet.

The portlet deployment descriptor <portlet-preferences> element, which contains a <portlet-preference> element for each portlet preference to be declared, along with an optional <preference-validator> element for declaring a portlet preferences validator.

Each <portlet-preference> element allows configuration of the name, values, and read-only flag for a single portlet using the <name>, <value>, and <read-only> elements, respectively. The <value> element can be repeated once for each string in the portlet preference values array. The default value of the <read-only> element is false.

Section 18.4 Validating Preference Values on page 113 describes the portlet preferences validator configuration using either the portlet deployment descriptor or annotations.

Portlet deployment descriptor usage example:

<portlet>

...

<!—- Portlet Preferences -->

<portlet-preferences>

<preference>

<name>PreferredStockSymbols</name>

<value>FOO</value>

<value>XYZ</value>

<read-only>true</read-only>

</preference>

<preference>

<name>quotesFeedURL</name>

<value>http://www.foomarket.com/quotes</value>

</preference>

</portlet-preferences>

</portlet>

The @PortletConfiguration annotation prefs element provides corresponding fields for portlet preference configuration.

Annotation usage example:

@PortletConfiguration(portletName="ConfigTestPortlet",

prefs = {

@Preference(

name = "PreferredStockSymbols",

values = {"FOO", "XYZ"},

isReadOnly = true),

@Preference(

name = "quotesFeedURL",

values = {"http://www.foomarket.com/quotes"})

},

)

### Asynchronous Support

Asynchronous support can be configured either at the portlet level through the portlet deployment descriptor or by using the @PortletConfiguration annotation, or at the annotated method level through use of the @ServeResourceMethod annotation. In any case, asynchronous configuration consists of a binary flag set to true to indicate that asynchronous operation is supported. The default value is false.

Asynchronous support is only provided for the resource phase. An @ServeResourceMethod annotated method may apply to more than one portlet, see Section 4.8.6 Annotated Resource Method Dispatching on page 41. Since a resource method activating asynchronous support is likely to be coded to require such support, an asynchronous support flag set to true in the @ServeResourceMethod annotation will override the asynchronous support flag setting in the portlet configuration for any portlet to which the annotated method applies.

The portlet deployment descriptor provides the <async-supported> element to specify asynchronous support.

Portlet deployment descriptor usage example:

<portlet>

...

<async-supported>

true

</async-supported>

</portlet>

The @PortletConfiguration annotation asyncSupported element provides the corresponding field for asynchronous support configuration.

Annotation usage example:

@PortletConfiguration(portletName="ConfigTestPortlet", asyncSupported=true)

The @ServeResourceMethod annotation asyncSupported element provides corresponding allows asynchronous support configuration.

Annotation usage example:

@ServeResourceMethod(portletName="ConfigTestPortlet", asyncSupported=true)

### Multipart Support

Multipart form support can be configured at the portlet level through the portlet deployment descriptor or by using the @PortletConfiguration annotation. The multipart support configuration consists of the following elements:

|  |  |
| --- | --- |
| fileSizeThreshold | The size threshold after which the file will be written to disk. |
| location | The directory location where files will be stored. |
| maxFileSize | The maximum size allowed for uploaded files. |
| maxRequestSize | The maximum size allowed for multipart/form-data requests. |

The portlet deployment descriptor provides the <multipart-config> element to specify asynchronous support.

Portlet deployment descriptor usage example:

<portlet>

...

<multipart-config>

<location>/tmp</location>

<max-file-size>20848820</max-file-size>

<max-request-size>418018841</max-request-size>

<file-size-threshold>1048576</file-size-threshold>

</multipart-config>

</portlet>

The @PortletConfiguration annotation multipart element provides the corresponding fields for multipart support configuration. The @Multipart annotation can only be used within the @PortletConfiguration annotation. It contains an additional supported flag indicating whether or not the multipart support is enabled. By default it is set to false.

Annotation usage example:

@PortletConfiguration(portletName="ConfigTestPortlet",

multipart = @Multipart(supported=true, location="/tmp",

fileSizeThreshold=1024\*1024,

maxFileSize=1024\*1024\*4,

maxRequestSize=1024\*1024\*8)

)

The default value for the multipart element in the @PortletConfiguration annotation is:

@PortletConfiguration(portletName="ConfigTestPortlet",

multipart = Multipart(supported=false)

)

1. Change History
   1. Version 3.0 Changes

* Chapter 28 Configuration: New chapter. Aggregates all information on portlet and portlet application configuration.
* Chapter 27 Packaging and Deployment: Editorial changes. Removed portlet deployment descriptor schema.
* Chapter 26 Portlet Tag Library Added type attribute to param tag to allow action and render parameters to be added. Expanded list of objects made available through the defineObjects tag.
* Chapter 25 Dispatching to JSPs and Servlets Major rewrite while keeping the actual semantics compatible with JSR 286. Added behavior specification for new HTTP servlet request and response methods during an include or forward.
* Chapter 24 Security: Editorial changes.
* Chapter 23 Caching: Editorial changes.
* Chapter 22 Client-Side Support: New chapter. Describes client-side support provided by the portlet hub.
* Chapter 21 Asynchronous Support: New chapter describing asynchronous support during the resource phase.
* Chapter 20 Managed Bean Support: New chapter. Describes portlet initialization through the CDI container, custom scopes for portlets, and portlet artifacts as injectable predefined beans.
* Chapter 19 Sessions: Editorial changes.
* Chapter 18 Portlet Preferences: Editorial changes. Added annotation-based configuration for preferences validator.
* Chapter 16 Portlet Filters: Editorial changes, added configuration through annotations.
* Chapter 17 Coordination between Portlets: Editorial changes.
* Integrated material from JSR 286 Chapters "Fragment Serving" and Resource Serving" into the appropriate spec sections. These will not exist in the new specification as separate chapters in order to reduce redundancy and to group similar information accordingly.
* Chapter 15 PortletResponse Interfaces: Major rewrite to include to more accurately present JSR 362 concepts. Expanded introductory section. Added description for the URL creation methods. Added method on the action response to allow the portlet to request a redirect after action phase processing.
* Chapter 14 Portlet Request Interfaces: Major rewrite to include to more accurately present JSR 362 concepts. Expanded introductory section. Added PORTLET\_STATE resource request attribute for partial action request support.
* Chapter 13 Portlet URLs: Major rewrite to add JSR 362 concepts. Added ActionURL and RenderURL interfaces. Added configuration of the portlet URL generation listener through annotations.
* Chapter 12 Render State: New chapter.
* Chapter 11 Portlet Parameters: New chapter.
* Chapter 10 Window States: Editorial changes. Changed language to refer to portlet configuration rather than to the portlet deployment descriptor. Moved portlet descriptor examples to configuration chapter.
* Appendix D Custom Portlet Modes: Editorial changes. Added examples for annotation-based configuration.
* Appendix B Markup Fragments: Editorial changes.
* Chapter 9 Portlet Modes: Editorial changes. Changed language to refer to portlet configuration rather than to the portlet deployment descriptor. Moved portlet descriptor examples to configuration chapter.
* Chapter 8 Portlet Context: Editorial changes.

See PORTLETSPEC3-53 for discussion of further potential changes.

* Chapter 7 Portal Context: Editorial changes.
* Chapter 6 The PortletConfig Interface: Editorial changes to improve wording. Changed wording so as to refer to the ‘portlet configuration’ rather than to the ‘portlet deployment descriptor.   
  PORTLETSPEC3-18: Added method for obtaining public render parameter names and QNames under ‘Public Render Parameters’. Added ‘Portlet Modes’ section. Added ‘Window States’ section.
* Chapter 5 Portlet Applications: Editorial changes. Updated servlet specification references to the appropriate sections of Servlet Specification Version 3.1.
* Chapter 4 Portlet Lifecycle Interfaces: Editorial changes to improve wording. Changed wording so as to refer to the ‘portlet configuration’ rather than to the ‘portlet deployment descriptor’ where appropriate, since JSR 362 allows configuration through annotations. Rewrote ‘Portlet Customization Levels’ section. Rewrote ‘Request Handling’ section. Considerable editorial changes to ‘GenericPortlet’ section. Added ‘Dispatching to Annotated Methods’ section explaining GenericPortlet dispatching.
* Chapter 3 Portlet Concepts: Major rewrite to include to more accurately present JSR 362 concepts. Expanded introductory section. Clarified relationship between the Portlet Specification, the portal system, and the portlet container. Clarified the portlet phase model. Provided more detail on portlet requests. Added client-side support concepts. Changed wording so as to refer to the ‘portlet configuration’ rather than to the ‘portlet deployment descriptor’ where appropriate, since JSR 362 allows configuration through annotations. Described portlet parameters and render state. Described portlet URLs. Discussed portlet phase execution. Discussed portlet lifecycle methods.
* Chapter 2 Relationship to the Servlet Specification: Editorial changes to improve wording. Updated minimum runtime environment version to ‘Servlet Specification 3.1’.
* Chapter 1 Overview: Many editorial changes to improve language. Added ‘Render State’ section. Updated ‘Compatibility’ section. Moved change history to this appendix. Updated ‘Relationship to Java Enterprise Edition’ section to refer to target versions.
* Preface: Moved Chapter 1 from the JSR 286 document to be the new Preface. Updated the versions of the referenced Java specifications. Updated ‘Other Important References section. Updated ‘Providing Feedback’ to refer to the JSR 362 mailing list. Streamlined ‘Acknowledgements’.
* General: Restructured document to improve flow and layout.
* PLT7.1.1 PORTLETSPEC3-21: Added description of the BaseURL.write (Appendable) method.
* PLT6.2 PORTLETSPEC3-14: Added clarification about how the portlet title is defined if neither the <resource-bundle> nor the <portlet-info> elements are present in the portlet descriptor.
* PLT22.2 PORTLETSPEC3-19: Changed „doView" to „doHeaders" in the code example.
* PLT.26: PORTLETSPEC3-12; Clarified use of Portlet 1.0 Tag Library.
* PLT.11.1.4.3: PORTLETSPEC3-1: Clarified behavior when runtime setting renderHeaders is set to false.
* PLT.25.8.2: PORTLETSPEC3-6: Clarified how supported locales are to be specified.
  1. Version 2.1.0 Changes

Version 2.1.0 is a maintenance release amending the description of Resource Serving Dispatching in section PLT.5.4.5.3. This change, along with the associated Portlet API version 2.1.0 jar file update, closes a potential security vulnerability associated with Common Vulnerabilities and Exposures ID CVE-2015-1926.

* 1. Version 2.0 Changes

This section provides an overview of changes introduced with Portlet Specification 2.0 as compared to version 1.0. For additional details, consult the JSR 286 Portlet Specification 2.0 document.

* + 1. Major changes introduced with V 2.0

The major new features of version 2.0 include:

Events – enabling a portlet to send and receive events and perform state changes or send further events as a result of processing an event.

Public render parameters – allowing portlets to share parameters with other portlets.

Resource serving – provides the ability for a portlet to serve a resource.

Portlet filter – allowing on-the-fly transformations of information in both the request to and the response from a portlet.

* + 1. Clarifications that may make V1.0 Portlets Non-compliant

Depending on the implementation of the portlet of a specific runtime behavior of a portlet container the following clarifications may lead to different results when executing a portlet in either a JSR 168 or a JSR 286 container:

XML escaping of portlet URLs produced via the portlet tag library.   
V 2.0 clarifies that the default is all portlet URLs are XML escaped. The default can be changed with the new attribute escapeXML. JSR 168 portlets depending on the fact that portlet URLs created via the tag library are not XML escaped can change the default to non-escaped via the portlet container runtime option javax.portlet.escapeXml (see PLT.26.7)

Defining multiple values for the same parameter name in the Portlet param tag.  
V 2.0 clarifies that if the same name of a parameter occurs more than once within an actionURL, renderURL or resourceURL the values must be delivered as parameter value array with the values in the order of the declaration within the URL tag. Portlets assuming that the last occurrence wins and replaces the previous set values will behave differently in V2.0 containers.

getProtocol for included servlets / JSPs no longer returns null.  
V 2.0 defines that getProtocol now returns ‘HTTP/1.1’ and thus is better aligned with the servlet model that expects the getProtocol to return this value in the GenericServlet.

Parameters set on the portlet URL and the post body are aggregated into the request parameter set. Portlet URL parameters are presented before post body data. JSR 168 did not define if and how post body and portlet URL parameters are being merged. The added clarification mirrors the behavior defined in the servlet specification for servlets.

RenderResponse.setContentType is no longer required before calling getWriter or getOutputstream. Calling getWriter or getOutputstream without previously setting the content type will no longer result in an IllegalStateException.

PortletURL.setParameter called with a null value as value did throw an IllegalStateException whereas in V2.0 it results in removing that parameter from the PortletURL.

1. Markup Fragments

Portlets generate markup fragments that are aggregated in a portal page document. Because of this, there are some rules and limitations in the markup elements generated by portlets. Portlets should conform to these rules and limitations when generating content.

The disallowed tags indicated below are those tags that impact content generated by other portlets or may even break the entire portal page. Inclusion of such a tag invalidates the whole markup fragment.

Portlets generating HTML fragments must not use the following tags: base, body, frame, frameset, head, html and title. Using the iframe tag is not forbidden, but portlets using iframes should not expect portal/portlet context for the content of iframes.

Portlets generating XHTML and XHTML-Basic fragments must not use the following tags: base, body, iframe, head, html and title.

HTML, XHTML and XHTML-Basic specifications disallow the use of certain elements outside of the <head> element in the document. However, some browser implementations support some of these tags in other sections of the document. For example: current versions of Internet Explorer and Netscape Navigator both support the style tag anywhere within the document. Portlet developers should decide carefully the use of following markup elements that fit this description: link, meta and style.

1. User Attribute Names

This appendix defines a set of attribute names for user information. Portlet developers should use these attribute names in order to promote portability. To allow portals an automated mapping of commonly used user information attributes portlet programmers should use these attribute names. These attribute names are derived from the Platform for Privacy Preferences 1.0 (P3P 1.0) Specification by the W3C (http://www.w3c.org/TR/P3P).

|  |
| --- |
| **Attribute Name** |
| user.bdate.ymd.year |
| user.bdate.ymd.month |
| user.bdate.ymd.day |
| user.bdate.hms.hour |
| user.bdate.hms.minute |
| user.bdate.hms.second |
| user.bdate.fractionsecond |
| user.bdate.timezone |
| user.gender |
| user.employer |
| user.department |
| user.jobtitle |
| user.name.prefix |
| user.name.given |
| user.name.family |
| user.name.middle |
| user.name.suffix |
| user.name.nickName |
| user.login.id |
| user.home-info.postal.name |
| user.home-info.postal.street |
| user.home-info.postal.city |
| user.home-info.postal.stateprov |
| user.home-info.postal.postalcode |
| user.home-info.postal.country |
| user.home-info.postal.organization |
| user.home-info.telecom.telephone.intcode |
| user.home-info.telecom.telephone.loccode |
| user.home-info.telecom.telephone.number |
| user.home-info.telecom.telephone.ext |
| user.home-info.telecom.telephone.comment |
| user.home-info.telecom.fax.intcode |
| user.home-info.telecom.fax.loccode |
| user.home-info.telecom.fax.number |
| user.home-info.telecom.fax.ext |
| user.home-info.telecom.fax.comment |
| user.home-info.telecom.mobile.intcode |
| user.home-info.telecom.mobile.loccode |
| user.home-info.telecom.mobile.number |
| user.home-info.telecom.mobile.ext |
| user.home-info.telecom.mobile.comment |
| user.home-info.telecom.pager.intcode |
| user.home-info.telecom.pager.loccode |
| user.home-info.telecom.pager.number |
| user.home-info.telecom.pager.ext |
| user.home-info.telecom.pager.comment |
| user.home-info.online.email |
| user.home-info.online.uri |
| user.business-info.postal.name |
| user.business-info.postal.street |
| user.business-info.postal.city |
| user.business-info.postal.stateprov |
| user.business-info.postal.postalcode |
| user.business-info.postal.country |
| user.business-info.postal.organization |
| user.business-info.telecom.telephone.intcode |
| user.business-info.telecom.telephone.loccode |
| user.business-info.telecom.telephone.number |
| user.business-info.telecom.telephone.ext |
| user.business-info.telecom.telephone.comment |
| user.business-info.telecom.fax.intcode |
| user.business-info.telecom.fax.loccode |
| user.business-info.telecom.fax.number |
| user.business-info.telecom.fax.ext |
| user.business-info.telecom.fax.comment |
| user.business-info.telecom.mobile.intcode |
| user.business-info.telecom.mobile.loccode |
| user.business-info.telecom.mobile.number |
| user.business-info.telecom.mobile.ext |
| user.business-info.telecom.mobile.comment |
| user.business-info.telecom.pager.intcode |
| user.business-info.telecom.pager.loccode |
| user.business-info.telecom.pager.number |
| user.business-info.telecom.pager.ext |
| user.business-info.telecom.pager.comment |
| user.business-info.online.email |
| user.business-info.online.uri |

The PortletRequest interface provides these constants as an enum.

1. Custom Portlet Modes

Portals may provide support for custom portlet modes. Similarly, portlets may use custom portlet modes. This appendix describes a list of custom portlet modes and their intended functionality. Portals and portlets should use these custom portlet mode names if they provide support for the described functionality.

Portlets should use the getSupportedPortletModes method of the PortalContext interface to retrieve the portlet modes the portal supports.

* 1. About Portlet Mode

The about portlet mode should be used by the portlet to display information on the portlets purpose, origin, version etc.

Portlet developers should implement the about portlet mode functionality by using the @RenderMode(name="about")[[41]](#footnote-41) annotation supported by the GenericPortlet class.

In the deployment descriptor the support for the about portlet mode must be declared using

<portlet-app>

...

<portlet>

...

<supports>

...

<portlet-mode>about</portlet-mode>

</supports>

...

</portlet>

...

<custom-portlet-mode>

<portlet-mode>about</portlet-mode>

</custom-portlet-mode>

...

</portlet-app>

The corresponding annotation-based configuration could appear as follows:

@PortletConfiguration(portletName="ConfigTestPortlet",

supports = {

@Supports(locale="de", portletModes={"about"})

},

customPortletModes = {

@CustomPortletMode(description="about the portlet", name="about")

})

* 1. Config Portlet Mode

The config portlet mode should be used by the portlet to display one or more configuration views that let administrators configure portlet preferences that are marked non-modifiable in the deployment descriptor. This requires that the user must have administrator rights. Therefore, only the portal can create links for activating the config portlet mode.

Portlet developers should implement the config portlet mode functionality by using the @RenderMode(name="config")41 annotation supported by the GenericPortlet class.

The CONFIG mode of portlets operates typically on shared state that is common to many portlets of the same portlet definition. When a portlet modifies this shared state via the PortletPreferences, for all affected portlet entities, in the doView method the PortletPreferences must give access to the modified state.

In the deployment descriptor the support for the config portlet mode must be declared using

<portlet-app>

...

<portlet>

...

<supports>

...

<portlet-mode>config</portlet-mode>

</supports>

...

</portlet>

...

<custom-portlet-mode>

<portlet-mode>config</portlet-mode>

</custom-portlet-mode>

...

</portlet-app>

The corresponding annotation-based configuration could appear as follows:

@PortletConfiguration(portletName="ConfigTestPortlet",

supports = {

@Supports(locale="de", portletModes={"config"})

},

customPortletModes = {

@CustomPortletMode(name="config")

})

* 1. Edit\_defaults Portlet Mode

The edit\_defaults portlet mode signifies that the portlet should render a screen to set the default values for the modifiable preferences that are typically changed in the EDIT screen. The user must have the proper access rights to activate this mode. Therefore, only the portal can create links for activating edit\_defaults mode.

Portlet developers should implement the edit\_defaults portlet mode functionality by using the @RenderMode(name="edit\_defaults")41 annotation supported by the GenericPortlet class.

In the deployment descriptor the support for the edit\_defaults portlet mode must be declared using

<portlet-app>

...

<portlet>

...

<supports>

...

<portlet-mode> edit\_defaults </portlet-mode>

</supports>

...

</portlet>

...

<custom-portlet-mode>

<portlet-mode> edit\_defaults </portlet-mode>

</custom-portlet-mode>

...

</portlet-app>

The corresponding annotation-based configuration could appear as follows:

@PortletConfiguration(portletName="ConfigTestPortlet",

supports = {

@Supports(locale="de", portletModes={"edit\_defaults"})

},

customPortletModes = {

@CustomPortletMode(name="edit\_defaults")

})

* 1. Preview Portlet Mode

The preview portlet mode should be used by the portlet to render output without the need of having back-end connections or user specific data available. It may be used at page design time and in portlet development tools.

Portlet developers should implement the preview portlet mode functionality by using the @RenderMode(name="preview")41 annotation supported by the GenericPortlet class.

In the deployment descriptor the support for the preview portlet mode must be declared using

<portlet-app>

...

<portlet>

...

<supports>

...

<portlet-mode> preview </portlet-mode>

</supports>

...

</portlet>

...

<custom-portlet-mode>

<portlet-mode> preview </portlet-mode>

</custom-portlet-mode>

...

</portlet-app>

The corresponding annotation-based configuration could appear as follows:

@PortletConfiguration(portletName="ConfigTestPortlet",

supports = {

@Supports(locale="de", portletModes={"preview"})

},

customPortletModes = {

@CustomPortletMode(name="preview")

})

* 1. Print Portlet Mode

The print portlet mode signifies that the portlet should render a view that can be printed. Portlet developers should implement the print portlet mode functionality by using the @RenderMode(name="print")41 annotation supported by the GenericPortlet class.

In the deployment descriptor the support for the print portlet mode must be declared using

<portlet-app>

...

<portlet>

...

<supports>

...

<portlet-mode>print</portlet-mode>

</supports>

...

</portlet>

...

<custom-portlet-mode>

<portlet-mode>print</portlet-mode>

</custom-portlet-mode>

...

</portlet-app>

The corresponding annotation-based configuration could appear as follows:

@PortletConfiguration(portletName="ConfigTestPortlet",

supports = {

@Supports(locale="de", portletModes={"print"})

},

customPortletModes = {

@CustomPortletMode(name="print")

})

1. Deployment Descriptor Schema

The Portlet Specification 3.0 deployment descriptor schema is shown below.

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

<schema xmlns="http://www.w3.org/2001/XMLSchema"

xmlns:portlet="http://xmlns.jcp.org/xml/ns/portlet"

xmlns:xs="http://www.w3.org/2001/XMLSchema"

targetNamespace="http://xmlns.jcp.org/xml/ns/portlet"

elementFormDefault="qualified"

attributeFormDefault="unqualified"

version="3.0"

xml:lang="en">

<annotation>

<documentation>

This is the XML Schema for the Portlet 3.0 deployment descriptor.

</documentation>

</annotation>

<annotation>

<documentation>

The following conventions apply to all J2EE

deployment descriptor elements unless indicated otherwise.

- In elements that specify a pathname to a file within the

same JAR file, relative filenames (i.e., those not

starting with "/") are considered relative to the root of

the JAR file's namespace. Absolute filenames (i.e., those

starting with "/") also specify names in the root of the

JAR file's namespace. In general, relative names are

preferred. The exception is .war files where absolute

names are preferred for consistency with the Servlet API.

</documentation>

</annotation>

<!-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* -->

<import namespace="http://www.w3.org/XML/1998/namespace" schemaLocation="http://www.w3.org/2001/xml.xsd"/>

<element name="portlet-app" type="portlet:portlet-appType">

<annotation>

<documentation>

The portlet-app element is the root of the deployment descriptor

for a portlet application. This element has a required attribute version

to specify to which version of the schema the deployment descriptor

conforms. In order to be a valid JSR 362 portlet application the version

must have the value "3.0".

</documentation>

</annotation>

<unique name="portlet-name-uniqueness">

<annotation>

<documentation>

The portlet element contains the name of a portlet.

This name must be unique within the portlet application.

</documentation>

</annotation>

<selector xpath="portlet:portlet"/>

<field xpath="portlet:portlet-name"/>

</unique>

<unique name="custom-portlet-mode-uniqueness">

<annotation>

<documentation>

The custom-portlet-mode element contains the portlet-mode.

This portlet mode must be unique within the portlet application.

</documentation>

</annotation>

<selector xpath="portlet:custom-portlet-mode"/>

<field xpath="portlet:portlet-mode"/>

</unique>

<unique name="custom-window-state-uniqueness">

<annotation>

<documentation>

The custom-window-state element contains the window-state.

This window state must be unique within the portlet application.

</documentation>

</annotation>

<selector xpath="portlet:custom-window-state"/>

<field xpath="portlet:window-state"/>

</unique>

<unique name="user-attribute-name-uniqueness">

<annotation>

<documentation>

The user-attribute element contains the name the attribute.

This name must be unique within the portlet application.

</documentation>

</annotation>

<selector xpath="portlet:user-attribute"/>

<field xpath="portlet:name"/>

</unique>

<unique name="filter-name-uniqueness">

<annotation>

<documentation>

The filter element contains the name of a filter.

The name must be unique within the portlet application.

</documentation>

</annotation>

<selector xpath="portlet:filter"/>

<field xpath="portlet:filter-name"/>

</unique>

</element>

<complexType name="portlet-appType">

<sequence>

<element name="portlet" type="portlet:portletType" minOccurs="0" maxOccurs="unbounded">

<unique name="init-param-name-uniqueness">

<annotation>

<documentation>

The init-param element contains the name the attribute.

This name must be unique within the portlet.

</documentation>

</annotation>

<selector xpath="portlet:init-param"/>

<field xpath="portlet:name"/>

</unique>

<unique name="supports-mime-type-uniqueness">

<annotation>

<documentation>

The supports element contains the supported mime-type.

This mime type must be unique within the portlet.

</documentation>

</annotation>

<selector xpath="portlet:supports"/>

<field xpath="mime-type"/>

</unique>

<unique name="preference-name-uniqueness">

<annotation>

<documentation>

The preference element contains the name the preference.

This name must be unique within the portlet.

</documentation>

</annotation>

<selector xpath="portlet:portlet-preferences/portlet:preference"/>

<field xpath="portlet:name"/>

</unique>

<unique name="security-role-ref-name-uniqueness">

<annotation>

<documentation>

The security-role-ref element contains the role-name.

This role name must be unique within the portlet.

</documentation>

</annotation>

<selector xpath="portlet:security-role-ref"/>

<field xpath="portlet:role-name"/>

</unique>

</element>

<element name="custom-portlet-mode" type="portlet:custom-portlet-modeType" minOccurs="0" maxOccurs="unbounded"/>

<element name="custom-window-state" type="portlet:custom-window-stateType" minOccurs="0" maxOccurs="unbounded"/>

<element name="user-attribute" type="portlet:user-attributeType" minOccurs="0" maxOccurs="unbounded"/>

<element name="resource-bundle" type="portlet:resource-bundleType" minOccurs="0"/>

<element name="filter" type="portlet:filterType" minOccurs="0" maxOccurs="unbounded"/>

<element name="filter-mapping" type="portlet:filter-mappingType" minOccurs="0" maxOccurs="unbounded"/>

<element name="default-namespace" type="xs:anyURI" minOccurs="0"/>

<element name="event-definition" type="portlet:event-definitionType" minOccurs="0" maxOccurs="unbounded"/>

<element name="public-render-parameter" type="portlet:public-render-parameterType" minOccurs="0" maxOccurs="unbounded"/>

<element name="listener" type="portlet:listenerType" minOccurs="0" maxOccurs="unbounded"/>

<element name="container-runtime-option" type="portlet:container-runtime-optionType" minOccurs="0" maxOccurs="unbounded"/>

</sequence>

<attribute name="version" type="portlet:string" use="required"/>

<attribute name="id" type="portlet:string" use="optional"/>

</complexType>

<complexType name="cache-scopeType">

<annotation>

<documentation>

Caching scope, allowed values are "private" indicating that the content should not be shared

across users and "public" indicating that the content may be shared across users.

The default value if not present is "private".

Used in: portlet

</documentation>

</annotation>

<simpleContent>

<extension base="portlet:string"/>

</simpleContent>

</complexType>

<complexType name="custom-portlet-modeType">

<annotation>

<documentation>

A custom portlet mode that one or more portlets in

this portlet application supports.

If the portal does not need to provide some management functionality

for this portlet mode, the portal-managed element needs to be set

to "false", otherwise to "true". Default is "true".

Used in: portlet-app

</documentation>

</annotation>

<sequence>

<element name="description" type="portlet:descriptionType" minOccurs="0" maxOccurs="unbounded"/>

<element name="portlet-mode" type="portlet:portlet-modeType"/>

<element name="portal-managed" type="portlet:portal-managedType" minOccurs="0"/>

</sequence>

<attribute name="id" type="portlet:string" use="optional"/>

</complexType>

<complexType name="custom-window-stateType">

<annotation>

<documentation>

A custom window state that one or more portlets in this

portlet application supports.

Used in: portlet-app

</documentation>

</annotation>

<sequence>

<element name="description" type="portlet:descriptionType" minOccurs="0" maxOccurs="unbounded"/>

<element name="window-state" type="portlet:window-stateType"/>

</sequence>

<attribute name="id" type="portlet:string" use="optional"/>

</complexType>

<complexType name="expiration-cacheType">

<annotation>

<documentation>

Expiration-time defines the time in seconds after which the portlet output expires.

-1 indicates that the output never expires.

Used in: portlet

</documentation>

</annotation>

<simpleContent>

<extension base="int"/>

</simpleContent>

</complexType>

<complexType name="init-paramType">

<annotation>

<documentation>

The init-param element contains a name/value pair as an

initialization param of the portlet

Used in:portlet

</documentation>

</annotation>

<sequence>

<element name="description" type="portlet:descriptionType" minOccurs="0" maxOccurs="unbounded"/>

<element name="name" type="portlet:nameType"/>

<element name="value" type="portlet:valueType"/>

</sequence>

<attribute name="id" type="portlet:string" use="optional"/>

</complexType>

<complexType name="keywordsType">

<annotation>

<documentation>

Locale specific keywords associated with this portlet.

The kewords are separated by commas.

Used in: portlet-info

</documentation>

</annotation>

<simpleContent>

<extension base="portlet:string">

<attribute ref="xml:lang"/>

</extension>

</simpleContent>

</complexType>

<complexType name="mime-typeType">

<annotation>

<documentation>

MIME type name, e.g. "text/html".

The MIME type may also contain the wildcard

character '\*', like "text/\*" or "\*/\*".

Used in: supports

</documentation>

</annotation>

<simpleContent>

<extension base="portlet:string"/>

</simpleContent>

</complexType>

<complexType name="nameType">

<annotation>

<documentation>

The name element contains the name of a parameter.

Used in: init-param, ...

</documentation>

</annotation>

<simpleContent>

<extension base="portlet:string"/>

</simpleContent>

</complexType>

<complexType name="portletType">

<annotation>

<documentation>

The portlet element contains the declarative data of a portlet.

Used in: portlet-app

</documentation>

</annotation>

<sequence>

<element name="description" type="portlet:descriptionType" minOccurs="0" maxOccurs="unbounded"/>

<element name="portlet-name" type="portlet:portlet-nameType"/>

<element name="display-name" type="portlet:display-nameType" minOccurs="0" maxOccurs="unbounded"/>

<element name="portlet-class" type="portlet:portlet-classType" minOccurs="0" maxOccurs="1"/>

<element name="init-param" type="portlet:init-paramType" minOccurs="0" maxOccurs="unbounded"/>

<element name="expiration-cache" type="portlet:expiration-cacheType" minOccurs="0"/>

<element name="cache-scope" type="portlet:cache-scopeType" minOccurs="0"/>

<element name="supports" type="portlet:supportsType" maxOccurs="unbounded"/>

<element name="supported-locale" type="portlet:supported-localeType" minOccurs="0" maxOccurs="unbounded"/>

<element name="resource-bundle" type="portlet:resource-bundleType" minOccurs="0"/>

<element name="portlet-info" type="portlet:portlet-infoType" minOccurs="0"/>

<element name="portlet-preferences" type="portlet:portlet-preferencesType" minOccurs="0"/>

<element name="security-role-ref" type="portlet:security-role-refType" minOccurs="0" maxOccurs="unbounded"/>

<element name="supported-processing-event" type="portlet:event-definition-referenceType" minOccurs="0" maxOccurs="unbounded"/>

<element name="supported-publishing-event" type="portlet:event-definition-referenceType" minOccurs="0" maxOccurs="unbounded"/>

<element name="supported-public-render-parameter" type="portlet:string" minOccurs="0" maxOccurs="unbounded"/>

<element name="container-runtime-option" type="portlet:container-runtime-optionType" minOccurs="0" maxOccurs="unbounded"/>

<element name="dependency" type="portlet:dependencyType" minOccurs="0" maxOccurs="unbounded"/>

<element name="async-supported" type="boolean" minOccurs="0" maxOccurs="1">

<annotation>

<documentation>

If set to 'true', the portlet will support

asynchronous processing in resource requests.

</documentation>

</annotation>

</element>

<element name="multipart-config" type="portlet:multipartType" minOccurs="0" maxOccurs="1"/>

</sequence>

<attribute name="id" type="portlet:string" use="optional"/>

</complexType>

<simpleType name="portlet-classType">

<annotation>

<documentation>

The portlet-class element contains the fully

qualified class name of the portlet.

Used in: portlet

</documentation>

</annotation>

<restriction base="portlet:fully-qualified-classType"/>

</simpleType>

<complexType name="container-runtime-optionType">

<annotation>

<documentation>

The container-runtime-option element contains settings

for the portlet container that the portlet expects to be honored

at runtime. These settings may re-define default portlet container

behavior, like the javax.portlet.escapeXml setting that disables

XML encoding of URLs produced by the portlet tag library as

default.

Names with the javax.portlet prefix are reserved for the Java

Portlet Specification.

Used in: portlet-app, portlet

</documentation>

</annotation>

<sequence>

<element name="name" type="portlet:nameType"/>

<element name="value" type="portlet:valueType" minOccurs="0" maxOccurs="unbounded"/>

</sequence>

</complexType>

<complexType name="filter-mappingType">

<annotation>

<documentation>

Declaration of the filter mappings in this portlet

application is done by using filter-mappingType.

The container uses the filter-mapping

declarations to decide which filters to apply to a request,

and in what order. To determine which filters to

apply it matches filter-mapping declarations on the

portlet-name and the lifecyle phase defined in the

filter element. The order in which filters are invoked

is the order in which filter-mapping declarations

that match appear in the list of filter-mapping elements.

Used in: portlet-app

</documentation>

</annotation>

<sequence>

<element name="filter-name" type="portlet:filter-nameType"/>

<element name="portlet-name" type="portlet:portlet-nameType" maxOccurs="unbounded"/>

</sequence>

</complexType>

<complexType name="filterType">

<annotation>

<documentation>

The filter element specifies a filter that can transform the

content of portlet requests and portlet responses.

Filters can access the initialization parameters declared in

the deployment descriptor at runtime via the FilterConfig

interface.

A filter can be restricted to one or more lifecycle phases

of the portlet. Valid entries for lifecycle are:

ACTION\_PHASE, EVENT\_PHASE, RENDER\_PHASE,

RESOURCE\_PHASE.

The filter-name element allows an annotated filter to be addressed

and replaced through a corresponding portlet deployment descriptor

filter and / or filter mapping configuration.

If the filter class is null, the annotated filter with matching filter name will

be removed from the configuration. The ordinal element determines the order of execution

should there be multiple filters configured. The default ordinal number is 0.

Used in: portlet-app

</documentation>

</annotation>

<sequence>

<element name="description" type="portlet:descriptionType" minOccurs="0" maxOccurs="unbounded"/>

<element name="display-name" type="portlet:display-nameType" minOccurs="0" maxOccurs="unbounded"/>

<element name="filter-name" type="portlet:filter-nameType" minOccurs="0" maxOccurs="1"/>

<element name="filter-class" type="portlet:fully-qualified-classType"/>

<element name="ordinal" type="xs:int" minOccurs="0" maxOccurs="1"/>

<element name="lifecycle" type="portlet:string" maxOccurs="unbounded"/>

<element name="init-param" type="portlet:init-paramType" minOccurs="0" maxOccurs="unbounded"/>

</sequence>

</complexType>

<complexType name="event-definitionType">

<annotation>

<documentation>

The event-definitionType is used to declare events the portlet can either

receive or emit.

The name must be unique and must be the one the

portlet is using in its code for referencing this event.

Used in: portlet-app

</documentation>

</annotation>

<sequence>

<element name="description" type="portlet:descriptionType" minOccurs="0" maxOccurs="unbounded"/>

<element name="display-name" type="portlet:display-nameType" minOccurs="0" maxOccurs="unbounded"/>

<choice>

<element name="qname" type="xs:QName"/>

<element name="name" type="xs:NCName"/>

</choice>

<element name="alias" type="xs:QName" minOccurs="0" maxOccurs="unbounded"/>

<element name="value-type" type="portlet:fully-qualified-classType" minOccurs="0"/>

</sequence>

<attribute name="id" type="portlet:string" use="optional"/>

</complexType>

<complexType name="event-definition-referenceType">

<annotation>

<documentation>

The event-definition-referenceType is used to reference events

declared with the event-definition element at application level.

Used in: portlet

</documentation>

</annotation>

<choice>

<element name="qname" type="xs:QName"/>

<element name="name" type="xs:NCName"/>

</choice>

<attribute name="id" type="portlet:string" use="optional"/>

</complexType>

<complexType name="listenerType">

<annotation>

<documentation>

The listenerType is used to declare listeners for this portlet application.

The optional listener-name element allows an annotated listener to be addressed

and replaced through a corresponding portlet deployment descriptor configuration.

If a listener name is provided and the listener class is null, the annotated

listener will be removed from the configuration. The ordinal element determines the

order of execution should there be multiple listeners configured. The default

ordinal number is 0.

Used in: portlet-app

</documentation>

</annotation>

<sequence>

<element name="listener-name" type="portlet:string" minOccurs="0" maxOccurs="1"/>

<element name="ordinal" type="xs:int" minOccurs="0" maxOccurs="1"/>

<element name="description" type="portlet:descriptionType" minOccurs="0" maxOccurs="unbounded"/>

<element name="display-name" type="portlet:display-nameType" minOccurs="0" maxOccurs="unbounded"/>

<element name="listener-class" type="portlet:fully-qualified-classType" minOccurs="0" maxOccurs="1"/>

</sequence>

<attribute name="id" type="portlet:string" use="optional"/>

</complexType>

<complexType name="portlet-infoType">

<sequence>

<element name="title" type="portlet:titleType" minOccurs="0" maxOccurs="unbounded"/>

<element name="short-title" type="portlet:short-titleType" minOccurs="0" maxOccurs="unbounded"/>

<element name="keywords" type="portlet:keywordsType" minOccurs="0" maxOccurs="unbounded"/>

</sequence>

<attribute name="id" type="portlet:string" use="optional"/>

</complexType>

<simpleType name="portal-managedType">

<annotation>

<documentation>

portal-managed indicates if a custom portlet mode

needs to be managed by the portal or not.

Per default all custom portlet modes are portal managed.

Valid values are:

- true for portal-managed

- false for not portal managed

Used in: custom-portlet-modes

</documentation>

</annotation>

<restriction base="portlet:string">

<enumeration value="true"/>

<enumeration value="false"/>

</restriction>

</simpleType>

<complexType name="portlet-modeType">

<annotation>

<documentation>

Portlet modes. The specification pre-defines the following values

as valid portlet mode constants:

"edit", "help", "view".

Portlet mode names are not case sensitive.

Used in: custom-portlet-mode, supports

</documentation>

</annotation>

<simpleContent>

<extension base="portlet:string"/>

</simpleContent>

</complexType>

<complexType name="portlet-nameType">

<annotation>

<documentation>

The portlet-name element contains the canonical name of the

portlet. Each portlet name is unique within the portlet

application.

Used in: portlet, filter-mapping

</documentation>

</annotation>

<simpleContent>

<extension base="portlet:string"/>

</simpleContent>

</complexType>

<complexType name="portlet-preferencesType">

<annotation>

<documentation>

Portlet persistent preference store.

Used in: portlet

</documentation>

</annotation>

<sequence>

<element name="preference" type="portlet:preferenceType" minOccurs="0" maxOccurs="unbounded"/>

<element name="preferences-validator" type="portlet:preferences-validatorType" minOccurs="0"/>

</sequence>

<attribute name="id" type="portlet:string" use="optional"/>

</complexType>

<complexType name="preferenceType">

<annotation>

<documentation>

Persistent preference values that may be used for customization

and personalization by the portlet.

Used in: portlet-preferences

</documentation>

</annotation>

<sequence>

<element name="name" type="portlet:nameType"/>

<element name="value" type="portlet:valueType" minOccurs="0" maxOccurs="unbounded"/>

<element name="read-only" type="portlet:read-onlyType" minOccurs="0"/>

</sequence>

<attribute name="id" type="portlet:string" use="optional"/>

</complexType>

<simpleType name="preferences-validatorType">

<annotation>

<documentation>

The class specified under preferences-validator implements

the PreferencesValidator interface to validate the

preferences settings.

Used in: portlet-preferences

</documentation>

</annotation>

<restriction base="portlet:fully-qualified-classType"/>

</simpleType>

<simpleType name="read-onlyType">

<annotation>

<documentation>

read-only indicates that a setting cannot

be changed in any of the standard portlet modes

("view","edit" or "help").

Per default all preferences are modifiable.

Valid values are:

- true for read-only

- false for modifiable

Used in: preferences

</documentation>

</annotation>

<restriction base="portlet:string">

<enumeration value="true"/>

<enumeration value="false"/>

</restriction>

</simpleType>

<complexType name="resource-bundleType">

<annotation>

<documentation>

Name of the resource bundle containing the language specific

portlet informations in different languages (Filename without

the language specific part (e.g. \_en) and the ending (.properties).

Used in: portlet-info

</documentation>

</annotation>

<simpleContent>

<extension base="portlet:string"/>

</simpleContent>

</complexType>

<complexType name="role-linkType">

<annotation>

<documentation>

The role-link element is a reference to a defined security role.

The role-link element must contain the name of one of the

security roles defined in the security-role elements.

Used in: security-role-ref

</documentation>

</annotation>

<simpleContent>

<extension base="portlet:string"/>

</simpleContent>

</complexType>

<complexType name="security-role-refType">

<annotation>

<documentation>

The security-role-ref element contains the declaration of a

security role reference in the code of the web application. The

declaration consists of an optional description, the security

role name used in the code, and an optional link to a security

role. If the security role is not specified, the Deployer must

choose an appropriate security role.

The value of the role name element must be the String used

as the parameter to the

EJBContext.isCallerInRole(String roleName) method

or the HttpServletRequest.isUserInRole(String role) method.

Used in: portlet

</documentation>

</annotation>

<sequence>

<element name="description" type="portlet:descriptionType" minOccurs="0" maxOccurs="unbounded"/>

<element name="role-name" type="portlet:role-nameType"/>

<element name="role-link" type="portlet:role-linkType" minOccurs="0"/>

</sequence>

<attribute name="id" type="portlet:string" use="optional"/>

</complexType>

<complexType name="public-render-parameterType">

<annotation>

<documentation>

The public-render-parameters defines a render parameter that is allowed to be public

and thus be shared with other portlets.

The identifier must be used for referencing this public render parameter in the portlet code.

Used in: portlet-app

</documentation>

</annotation>

<sequence>

<element name="description" type="portlet:descriptionType" minOccurs="0" maxOccurs="unbounded"/>

<element name="display-name" type="portlet:display-nameType" minOccurs="0" maxOccurs="unbounded"/>

<element name="identifier" type="portlet:string"/>

<choice>

<element name="qname" type="xs:QName"/>

<element name="name" type="xs:NCName"/>

</choice>

</sequence>

<attribute name="id" type="portlet:string" use="optional"/>

</complexType>

<complexType name="short-titleType">

<annotation>

<documentation>

Locale specific short version of the static title.

Used in: portlet-info

</documentation>

</annotation>

<simpleContent>

<extension base="portlet:string">

<attribute ref="xml:lang"/>

</extension>

</simpleContent>

</complexType>

<complexType name="supportsType">

<annotation>

<documentation>

Supports indicates the portlet modes a

portlet supports for a specific content type. All portlets must

support the view mode.

Used in: portlet

</documentation>

</annotation>

<sequence>

<element name="mime-type" type="portlet:mime-typeType"/>

<element name="portlet-mode" type="portlet:portlet-modeType" minOccurs="0" maxOccurs="unbounded"/>

<element name="window-state" type="portlet:window-stateType" minOccurs="0" maxOccurs="unbounded"/>

</sequence>

<attribute name="id" type="portlet:string" use="optional"/>

</complexType>

<complexType name="dependencyType">

<annotation>

<documentation>

Dependency specifies a resource on which the portlet depends.

</documentation>

</annotation>

<sequence>

<element name="name" type="portlet:string"/>

<element name="scope" type="portlet:string" minOccurs="0" maxOccurs="1"/>

<element name="version" type="portlet:string" minOccurs="0" maxOccurs="1"/>

</sequence>

</complexType>

<complexType name="multipartType">

<annotation>

<documentation>

Provides multipart configuration information.

</documentation>

</annotation>

<sequence>

<element name="location" type="portlet:string" minOccurs="0" maxOccurs="1"/>

<element name="max-file-size" type="xs:long" minOccurs="0" maxOccurs="1"/>

<element name="max-request-size" type="xs:long" minOccurs="0" maxOccurs="1"/>

<element name="file-size-threshold" type="xs:int" minOccurs="0" maxOccurs="1"/>

</sequence>

</complexType>

<complexType name="supported-localeType">

<annotation>

<documentation>

Indicated the locales the portlet supports.

Used in: portlet

</documentation>

</annotation>

<simpleContent>

<extension base="portlet:string"/>

</simpleContent>

</complexType>

<complexType name="titleType">

<annotation>

<documentation>

Locale specific static title for this portlet.

Used in: portlet-info

</documentation>

</annotation>

<simpleContent>

<extension base="portlet:string">

<attribute ref="xml:lang"/>

</extension>

</simpleContent>

</complexType>

<complexType name="user-attributeType">

<annotation>

<documentation>

User attribute defines a user specific attribute that the

portlet application needs. The portlet within this application

can access this attribute via the request parameter USER\_INFO

map.

Used in: portlet-app

</documentation>

</annotation>

<sequence>

<element name="description" type="portlet:descriptionType" minOccurs="0" maxOccurs="unbounded"/>

<element name="name" type="portlet:nameType"/>

</sequence>

<attribute name="id" type="portlet:string" use="optional"/>

</complexType>

<complexType name="valueType">

<annotation>

<documentation>

The value element contains the value of a parameter.

Used in: init-param

</documentation>

</annotation>

<simpleContent>

<extension base="portlet:string"/>

</simpleContent>

</complexType>

<complexType name="window-stateType">

<annotation>

<documentation>

Portlet window state. Window state names are not case sensitive.

Used in: custom-window-state

</documentation>

</annotation>

<simpleContent>

<extension base="portlet:string"/>

</simpleContent>

</complexType>

<!--- everything below is copied from j2ee\_1\_4.xsd -->

<complexType name="descriptionType">

<annotation>

<documentation>

The description element is used to provide text describing the

parent element. The description element should include any

information that the portlet application war file producer wants

to provide to the consumer of the portlet application war file

(i.e., to the Deployer). Typically, the tools used by the

portlet application war file consumer will display the

description when processing the parent element that contains the

description. It has an optional attribute xml:lang to indicate

which language is used in the description according to

RFC 1766 (http://www.ietf.org/rfc/rfc1766.txt). The default

value of this attribute is English(“en”).

Used in: init-param, portlet, portlet-app, security-role

</documentation>

</annotation>

<simpleContent>

<extension base="portlet:string">

<attribute ref="xml:lang"/>

</extension>

</simpleContent>

</complexType>

<complexType name="display-nameType">

<annotation>

<documentation>

The display-name type contains a short name that is intended

to be displayed by tools. It is used by display-name

elements. The display name need not be unique.

Example:

...

<display-name xml:lang="en">Employee Self Service</display-name>

It has an optional attribute xml:lang to indicate

which language is used in the description according to

RFC 1766 (http://www.ietf.org/rfc/rfc1766.txt). The default

value of this attribute is English(“en”).

</documentation>

</annotation>

<simpleContent>

<extension base="portlet:string">

<attribute ref="xml:lang"/>

</extension>

</simpleContent>

</complexType>

<simpleType name="fully-qualified-classType">

<annotation>

<documentation>

The elements that use this type designate the name of a

Java class or interface.

</documentation>

</annotation>

<restriction base="portlet:string"/>

</simpleType>

<simpleType name="role-nameType">

<annotation>

<documentation>

The role-nameType designates the name of a security role.

The name must conform to the lexical rules for an NMTOKEN.

</documentation>

</annotation>

<restriction base="NMTOKEN"/>

</simpleType>

<simpleType name="string">

<annotation>

<documentation>

This is a special string datatype that is defined by JavaEE

as a base type for defining collapsed strings. When

schemas require trailing/leading space elimination as

well as collapsing the existing whitespace, this base

type may be used.

</documentation>

</annotation>

<restriction base="xs:token">

<whiteSpace value="collapse"/>

</restriction>

</simpleType>

<simpleType name="filter-nameType">

<annotation>

<documentation>

The logical name of the filter is declare

by using filter-nameType. This name is used to map the

filter. Each filter name is unique within the portlet

application.

Used in: filter, filter-mapping

</documentation>

</annotation>

<restriction base="portlet:string"/>

</simpleType>

</schema>

1. See the Servlet Specification Version 3.1, Section 1.1 What is a Servlet? [↑](#footnote-ref-1)
2. Partial action processing is part of the portlet client-side support and is described in more detail in Section 22.3.5 Partial Action. [↑](#footnote-ref-2)
3. See RFC 2616, http://www.w3.org/Protocols/rfc2616/rfc2616.html [↑](#footnote-ref-3)
4. *Servlet Specification 3.1 sections 10.7.1* and *10.7.2* [↑](#footnote-ref-4)
5. *Servlet Specification 3.1 sections 10.5* and *10.6* [↑](#footnote-ref-5)
6. See in particular *Servlet Specification 3.1 section 15.2.2* [↑](#footnote-ref-6)
7. See Section 4.7.1 Dispatching to GenericPortlet Annotated Methods on page 27. [↑](#footnote-ref-7)
8. See Section 4.8.4 Annotated Header Method Dispatching on page 37. [↑](#footnote-ref-8)
9. Note that this differs from Version 2.0 behavior. In Version 2.0, all render parameters had to be set on the ActionResponse or EventResponse, even if they were unchanged as compared to the values during the last render phase. [↑](#footnote-ref-9)
10. See Servlet Specification Version 3.1, Section 3.1 HTTP Protocol Parameters [↑](#footnote-ref-10)
11. See W3C: Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies, http://www.w3.org/TR/2001/WD-CCPP-struct-vocab-20010315/ [↑](#footnote-ref-11)
12. See JSR 188 CC/PP Processing, http://jcp.org/en/jsr/detail?id=188, for more details on CC/PP profile processing. [↑](#footnote-ref-12)
13. See the ECMAScript® Language Specification 5.1 Edition, Section 7.6 Identifier Names and Identifiers, http://www.ecma-international.org/ecma-262/5.1/#sec-7.6 [↑](#footnote-ref-13)
14. See Servlet Specification Version 3.1 Section 5.5 Internationalization for details. [↑](#footnote-ref-14)
15. Unfortunately, at this time there is no standard for page resource naming. We suggest that a name be used that designates the resource, such as 'jQuery'. The scope should be taken to be the reverse dotted domain name of the organization providing the resource, such as 'com.jquery'. The version should adhere to the semantic versioning specification located at <http://semver.org/> , such as '2.1.1'. [↑](#footnote-ref-15)
16. See *Servlet Specification Version 3.1, Section .7.7.1 Threading Issues*. [↑](#footnote-ref-16)
17. See the Java Servlet Specification Version 3.1, Section 2.3.3.3 Asynchronous Processing [↑](#footnote-ref-17)
18. See ECMAScript® 2015 Language Specification (ES6), at http://www.ecma-international.org/ecma-262/6.0/ [↑](#footnote-ref-18)
19. See Servlet Specification 3.1, Chapter 13 Security [↑](#footnote-ref-19)
20. See Servlet Specification Version 3.1, Section 13.5 Roles [↑](#footnote-ref-20)
21. See Section 15.5.3 Buffering on page 90 for discussion on response buffer handling. [↑](#footnote-ref-21)
22. Note that flushing output to the portal application does not necessarily imply that the output has been written to the client. Whether or not output was written to the client is an implementation specific detail. [↑](#footnote-ref-22)
23. See Servlet Specification Version 3.1, Chapter 12 Mapping Requests to Servlets [↑](#footnote-ref-23)
24. See Servlet Specification 3.1, Section 9.3.1 Included Request Parameters [↑](#footnote-ref-24)
25. See Servlet Specification 3.1, Chapter 6 Filtering and in particular Section 6.2.5 Filters and the Request Dispatcher [↑](#footnote-ref-25)
26. Based on the path and query string information used to obtain the PortletRequestDispatcher. [↑](#footnote-ref-26)
27. The portlet session returned by getPortletSession(APPLICATION\_SCOPE) [↑](#footnote-ref-27)
28. N/A indicates that such a method is not available in the portlet interface and the functionality defined by the Servlet Specification must be provided for this method. [↑](#footnote-ref-28)
29. Accesses merged query string and portlet parameters as described in Section 25.1.1.1 Accessing Merged Parameters [↑](#footnote-ref-29)
30. no-op indicates that this method does not perform any operation. [↑](#footnote-ref-30)
31. Async is only supported in the resource phase; method must throw IllegalStateException. [↑](#footnote-ref-31)
32. Asynchronous processing must be started in the resource method before a resource dispatcher include or forward. If async processing has been started, this method may be used to restart async processing. Otherwise it must throw an IllegalStateException. [↑](#footnote-ref-32)
33. Protocol upgrade is not supported. [↑](#footnote-ref-33)
34. A ‘null stream’ or ‘null writer’ designates an output stream or writer that ignores all output. [↑](#footnote-ref-34)
35. See JSP Specification Version 2.2, Section *JSP.7.3.9 Well-Known URIs*. [↑](#footnote-ref-35)
36. See Servlet Specification Version 3.1 Chapter 10 Web Applications and Chapter 13 Deployment Descriptor. [↑](#footnote-ref-36)
37. See the Jar File Specification, <http://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html>. Note that the Servlet Specification states “The format of the manifest entry should follow the standard JAR manifest format.” See the Java Product Versioning Specification, <https://docs.oracle.com/javase/7/docs/technotes/guides/versioning/spec/versioning2.html>. [↑](#footnote-ref-37)
38. If the namespace is missing, the defined default namespace is assumed. Note that the resource bundle name must comply with the java.util.Property.store method, i.e. the “:” must be escaped. [↑](#footnote-ref-38)
39. Qualified names are defined and discussed in detail in the following documents:

    [XML Schema Part2: Datatypes specification](http://www.w3.org/TR/xmlschema-2/#QName) (http://www.w3.org/TR/xmlschema-2/#QName)

    [Namespaces in XML](http://www.w3.org/TR/REC-xml-names/#ns-qualnames) (<http://www.w3.org/TR/REC-xml-names/#ns-qualnames>),

    [Namespaces in XML Errata](http://www.w3.org/XML/xml-names-19990114-errata) (<http://www.w3.org/XML/xml-names-19990114-errata>),

    TAG Finding: Using Qualified Names (QNames) as Identifiers in Content (http://www.w3.org/2001/tag/doc/qnameids-2002-06-17). [↑](#footnote-ref-39)
40. See IETF BCP 47, "Tags for Identifying Languages", https://tools.ietf.org/html/bcp47 [↑](#footnote-ref-40)
41. See Section Dispatching to GenericPortlet Annotated Methods [↑](#footnote-ref-41)