Java™ Servlet Specification

Version 3.1
Shing Wai Chan
Rajiv Mordani
April 2013

Oracle Corporation
www.oracle.com

Submit comments about this document to: users@servlet-spec.java.net
ORACLE IS WILLING TO LICENSE THIS SPECIFICATION TO YOU ONLY UPON THE CONDITION THAT YOU ACCEPT ALL OF THE TERMS CONTAINED IN THIS AGREEMENT. PLEASE READ THE TERMS AND CONDITIONS OF THIS AGREEMENT CAREFULLY. BY DOWNLOADING THIS SPECIFICATION, YOU ACCEPT THE TERMS AND CONDITIONS OF THE AGREEMENT. IF YOU ARE NOT WILLING TO BE BOUND BY IT, SELECT THE 'DECLINE' BUTTON AT THE BOTTOM OF THIS PAGE.

Version: 3.1
Status: Final Release
Specification Lead: Oracle America, Inc. ("Specification Lead")
Release: April 2013

Copyright 2013 Oracle America, Inc.
All rights reserved.

LIMITED LICENSE GRANTS

1. License for Evaluation Purposes. Specification Lead hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under Specification Lead's applicable intellectual property rights to view, download, use and reproduce the Specification only for the purpose of internal evaluation. This includes (i) developing applications intended to run on an implementation of the Specification, provided that such applications do not themselves implement any portion(s) of the Specification, and (ii) discussing the Specification with any third party; and (iii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Specification.

2. License for the Distribution of Compliant Implementations. Specification Lead also grants you a perpetual, non-exclusive, non-transferable, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights or, subject to the provisions of subsection 4 below, patent rights it may have covering the Specification to create and/or distribute an Independent Implementation of the Specification that: (a) fully implements the Specification including all its required interfaces and functionality; (b) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented; and (c) passes the Technology Compatibility Kit (including satisfying the requirements of the applicable TCK Users Guide) for such Specification ("Compliant Implementation"). In addition, the foregoing license is expressly conditioned on your not acting outside its scope. No license is granted hereunder for any other purpose (including, for example, modifying the Specification, other than to the extent of your fair use rights, or distributing the Specification to third parties). Also, no right, title, or interest in or to any trademarks, service marks, or trade names of Specification Lead or Specification Lead's licensors is granted hereunder. Java, and Java-related logos, marks and names are trademarks or registered trademarks of Oracle America, Inc. in the U.S. and other countries.

3. Pass-through Conditions. You need not include limitations (a)-(c) from the previous paragraph or any other particular "pass through" requirements in any license You grant concerning the use of your Independent Implementation or products derived from it. However, except with respect to Independent Implementations (and products derived from them) that satisfy limitations (a)-(c) from the previous paragraph, You may neither: (a) grant or otherwise pass through to your licensees any licenses under Specification Lead's applicable intellectual property rights; nor (b) authorize your licensees to make any claims concerning their implementation's compliance with the Specification in question.


a. With respect to any patent claims covered by the license granted under subparagraph 2 above that would be infringed by all technically feasible implementations of the Specification, such license is conditioned upon your offering on fair, reasonable and non-discriminatory terms, to
any party seeking it from You, a perpetual, non-exclusive, non-transferable, worldwide license under Your patent rights which are or would be infringed by all technically feasible implementations of the Specification to develop, distribute and use a Compliant Implementation.

b With respect to any patent claims owned by Specification Lead and covered by the license granted under subparagraph 2, whether or not their infringement can be avoided in a technically feasible manner when implementing the Specification, such license shall terminate with respect to such claims if You initiate a claim against Specification Lead that it has, in the course of performing its responsibilities as the Specification Lead, induced any other entity to infringe Your patent rights.

c Also with respect to any patent claims owned by Specification Lead and covered by the license granted under subparagraph 2 above, where the infringement of such claims can be avoided in a technically feasible manner when implementing the Specification such license, with respect to such claims, shall terminate if You initiate a claim against Specification Lead that its making, having made, using, offering to sell, selling or importing a Compliant Implementation infringes Your patent rights.

5. Definitions. For the purposes of this Agreement: "Independent Implementation" shall mean an implementation of the Specification that neither derives from any of Specification Lead's source code or binary code materials nor, except with an appropriate and separate license from Specification Lead, includes any of Specification Lead's source code or binary code materials; "Licensor Name Space" shall mean the public class or interface declarations whose names begin with "java", "javax", "com.sun" and "com.oracle" or their equivalents in any subsequent naming convention adopted by Oracle America, Inc. through the Java Community Process, or any recognized successors or replacements thereof; and "Technology Compatibility Kit" or "TCK" shall mean the test suite and accompanying TCK User's Guide provided by Specification Lead which corresponds to the Specification and that was available either (i) from Specification Lead's 120 days before the first release of Your Independent Implementation that allows its use for commercial purposes, or (ii) more recently than 120 days from such release but against which You elect to test Your implementation of the Specification.

This Agreement will terminate immediately without notice from Specification Lead if you breach the Agreement or act outside the scope of the licenses granted above.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS". SPECIFICATION LEAD MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT (INCLUDING AS A CONSEQUENCE OF ANY PRACTICE OR IMPLEMENTATION OF THE SPECIFICATION), OR THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE. This document does not represent any commitment to release or implement any portion of the Specification in any product. In addition, the Specification could include technical inaccuracies or typographical errors.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SPECIFICATION LEAD OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED IN ANY WAY TO YOUR HAVING, IMPLEMENTING OR OTHERWISE USING USING THE SPECIFICATION, EVEN IF SPECIFICATION LEAD AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend Specification Lead and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your Java application, applet and/or implementation; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND
U.S. Government: If this Specification is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

If you provide Specification Lead with any comments or suggestions concerning the Specification ("Feedback"), you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Specification Lead a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose.

GENERAL TERMS

Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.
Preface

This document is the Java™ Servlet Specification, version 3.1. The standard for the Java Servlet API is described herein.

Additional Sources

The specification is intended to be a complete and clear explanation of Java Servlets, but if questions remain, the following sources 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, implementors may use the reference implementation as a model of how to carry out the intention of the specification.

■ A compatibility test suite (CTS) has been provided for assessing whether implementations meet the compatibility requirements of the Java Servlet 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 Java Servlet API under the Java Community Process should be consulted, and is the final arbiter of such issues.

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

Who Should Read This Specification

The intended audience for this specification includes the following groups:
Web server and application server vendors that want to provide servlet engines that conform to this standard.

Authoring tool developers that want to support Web applications that conform to this specification

Experienced servlet authors who want to understand the underlying mechanisms of servlet technology.

We emphasize that this specification is not a user’s guide for servlet developers and is not intended to be used as such. References useful for this purpose are available from http://java.sun.com/products/servlet.

API Reference

The full specifications of classes, interfaces, and method signatures that define the Java Servlet API, as well as their 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
- JavaServer Pages™ ("JSP™"), version 2.2
- Java Naming and Directory Interface™ ("J.N.D.I.").
- Context and Dependency Injection for the Java EE Platform
- Managed Beans specification

These specifications may be found at the Java Platform, Enterprise Edition Web site: http://java.sun.com/javaee/.

Other Important References

The following Internet specifications provide information relevant to the development and implementation of the Java Servlet API and standard servlet engines:
RFC 1630 Uniform Resource Identifiers (URI)
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 2324 Hypertext Coffee Pot Control Protocol (HTCPCP/1.0)¹
RFC 2616 Hypertext Transfer Protocol (HTTP/1.1)
RFC 2617 HTTP Authentication: Basic and Digest Authentication
RFC 3986 Uniform Resource Identifier (URI): Generic Syntax

Online versions of these RFCs are at http://www.ietf.org/rfc/.

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

The eXtensible Markup Language (XML) is used for the specification of the Deployment Descriptors described in Chapter 13 of this specification. More information about XML can be found at the following Web sites:

http://java.sun.com/xml
http://www.xml.org/

Providing Feedback

We welcome any and all feedback about this specification. Please e-mail your comments to users@servleth-spc.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.

¹. This reference is mostly tongue-in-cheek although most of the concepts described in the HTCPCP RFC are relevant to all well-designed Web servers.
Expert Group members

- Deepak Anupalli (Pramati Technologies)
- Euigeun Chung (TmaxSoft, Inc)
- Robert Goff (IBM)
- Richard Hightower
- Seth Hodgson (Adobe Systems Inc.)
- Remy Maucherat (RedHat)
- Minoru Nitta (Fujitsu Limited)
- Ramesh PVK (Pramati Technologies)
- Alex Rojkov (Cauchò Technologies)
- Mark Thomas (VMware)
- Gregory John Wilkins
- Wenbo Zhu (Google Inc.)

Acknowledgements

Bill Shannon from Oracle has provided invaluable technical input to the specification. Ron Monzillo from Oracle has helped drive some of the proposals and technical discussions around security aspects.
Contents

1. Overview  1
   1.1 What is a Servlet?  1
   1.2 What is a Servlet Container?  1
   1.3 An Example  2
   1.4 Comparing Servlets with Other Technologies  3
   1.5 Relationship to Java Platform, Enterprise Edition  3
   1.6 Compatibility with Java Servlet Specification Version 2.5  4
      1.6.1 Processing annotations  4

2. The Servlet Interface  5
   2.1 Request Handling Methods  5
      2.1.1 HTTP Specific Request Handling Methods  5
      2.1.2 Additional Methods  6
      2.1.3 Conditional GET Support  6
   2.2 Number of Instances  6
      2.2.1 Note About The Single Thread Model  7
   2.3 Servlet Life Cycle  7
      2.3.1 Loading and Instantiation  7
2.3.2 Initialization 8
   2.3.2.1 Error Conditions on Initialization 8
   2.3.2.2 Tool Considerations 8

2.3.3 Request Handling 9
   2.3.3.1 Multithreading Issues 9
   2.3.3.2 Exceptions During Request Handling 9
   2.3.3.3 Asynchronous processing 10
   2.3.3.4 Thread Safety 20
   2.3.3.5 Upgrade Processing 20

2.3.4 End of Service 21

3. The Request 23
   3.1 HTTP Protocol Parameters 23
      3.1.1 When Parameters Are Available 24
   3.2 File upload 24
   3.3 Attributes 25
   3.4 Headers 25
   3.5 Request Path Elements 26
   3.6 Path Translation Methods 27
   3.7 Non Blocking IO 28
   3.8 Cookies 29
   3.9 SSL Attributes 30
   3.10 Internationalization 30
   3.11 Request data encoding 31
   3.12 Lifetime of the Request Object 31

4. Servlet Context 33
   4.1 Introduction to the ServletContext Interface 33
   4.2 Scope of a ServletContext Interface 33
4.3 Initialization Parameters 34

4.4 Configuration methods 34

4.4.1 Programatically adding and configuring Servlets 35

4.4.1.1 addServlet(String servletName, String className) 35

4.4.1.2 addServlet(String servletName, Servlet servlet) 35

4.4.1.3 addServlet(String servletName, Class<? extends Servlet> servletClass) 35

4.4.1.4 <T extends Servlet> T createServlet(Class<T> clazz) 35

4.4.1.5 ServletRegistration getServletRegistration(String servletName) 36

4.4.1.6 Map<String, ? extends ServletRegistration> getServletRegistrations() 36

4.4.2 Programatically adding and configuring Filters 36

4.4.2.1 addFilter(String filterName, String className) 36

4.4.2.2 addFilter(String filterName, Filter filter) 36

4.4.2.3 addFilter(String filterName, Class<? extends Filter> filterClass) 37

4.4.2.4 <T extends Filter> T createFilter(Class<T> clazz) 37

4.4.2.5 FilterRegistration getFilterRegistration(String filterName) 37

4.4.2.6 Map<String, ? extends FilterRegistration> getServletRegistrations() 37

4.4.3 Programatically adding and configuring Listeners 38

4.4.3.1 void addListener(String className) 38

4.4.3.2 <T extends EventListener> void addListener(T t) 38

4.4.3.3 void addListener(Class<? extends EventListener> listenerClass) 39

4.4.3.4 <T extends EventListener> void createListener(Class<T> clazz) 39

4.4.3.5 Annotation processing requirements for programmatically added Servlets, Filters and Listeners 40
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5</td>
<td>Context Attributes</td>
<td>40</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Context Attributes in a Distributed Container</td>
<td>41</td>
</tr>
<tr>
<td>4.6</td>
<td>Resources</td>
<td>41</td>
</tr>
<tr>
<td>4.7</td>
<td>Multiple Hosts and Servlet Contexts</td>
<td>42</td>
</tr>
<tr>
<td>4.8</td>
<td>Reloading Considerations</td>
<td>42</td>
</tr>
<tr>
<td>4.8.1</td>
<td>Temporary Working Directories</td>
<td>42</td>
</tr>
<tr>
<td>5.</td>
<td>The Response</td>
<td>45</td>
</tr>
<tr>
<td>5.1</td>
<td>Buffering</td>
<td>45</td>
</tr>
<tr>
<td>5.2</td>
<td>Headers</td>
<td>46</td>
</tr>
<tr>
<td>5.3</td>
<td>Non Blocking IO</td>
<td>47</td>
</tr>
<tr>
<td>5.4</td>
<td>Convenience Methods</td>
<td>48</td>
</tr>
<tr>
<td>5.5</td>
<td>Internationalization</td>
<td>49</td>
</tr>
<tr>
<td>5.6</td>
<td>Closure of Response Object</td>
<td>50</td>
</tr>
<tr>
<td>5.7</td>
<td>Lifetime of the Response Object</td>
<td>50</td>
</tr>
<tr>
<td>6.</td>
<td>Filtering</td>
<td>51</td>
</tr>
<tr>
<td>6.1</td>
<td>What is a filter?</td>
<td>51</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Examples of Filtering Components</td>
<td>52</td>
</tr>
<tr>
<td>6.2</td>
<td>Main Concepts</td>
<td>52</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Filter Lifecycle</td>
<td>52</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Wrapping Requests and Responses</td>
<td>54</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Filter Environment</td>
<td>54</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Configuration of Filters in a Web Application</td>
<td>55</td>
</tr>
<tr>
<td>6.2.5</td>
<td>Filters and the RequestDispatcher</td>
<td>58</td>
</tr>
<tr>
<td>7.</td>
<td>Sessions</td>
<td>61</td>
</tr>
<tr>
<td>7.1</td>
<td>Session Tracking Mechanisms</td>
<td>61</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Cookies</td>
<td>61</td>
</tr>
<tr>
<td>7.1.2</td>
<td>SSL Sessions</td>
<td>62</td>
</tr>
</tbody>
</table>
7.1.3 URL Rewriting 62
7.1.4 Session Integrity 62
7.2 Creating a Session 62
7.3 Session Scope 63
7.4 Binding Attributes into a Session 64
7.5 Session Timeouts 64
7.6 Last Accessed Times 65
7.7 Important Session Semantics 65
    7.7.1 Threading Issues 65
    7.7.2 Distributed Environments 65
    7.7.3 Client Semantics 66

8. Annotations and pluggability 67
8.1 Annotations and pluggability 67
    8.1.1 @WebServlet 67
    8.1.2 @WebFilter 69
    8.1.3 @WebInitParam 69
    8.1.4 @WebListener 69
    8.1.5 @MultipartConfig 70
    8.1.6 Other annotations / conventions 70
8.2 Pluggability 71
    8.2.1 Modularity of web.xml 71
    8.2.2 Ordering of web.xml and web-fragment.xml 72
    8.2.3 Assembling the descriptor from web.xml, web-fragment.xml and annotations 78
    8.2.4 Shared libraries / runtimes pluggability 91
8.3 JSP container pluggability 93
8.4 Processing annotations and fragments 93

9. Dispatching Requests 95
9.1 Obtaining a RequestDispatcher  95
  9.1.1 Query Strings in Request Dispatcher Paths  96
9.2 Using a Request Dispatcher  96
9.3 The Include Method  97
  9.3.1 Included Request Parameters  97
9.4 The Forward Method  98
  9.4.1 Query String  98
  9.4.2 Forwarded Request Parameters  98
9.5 Error Handling  99
9.6 Obtaining an AsyncContext  99
9.7 The Dispatch Method  100
  9.7.1 Query String  100
  9.7.2 Dispatched Request Parameters  100

10. Web Applications  103
  10.1 Web Applications Within Web Servers  103
  10.2 Relationship to ServletContext  103
  10.3 Elements of a Web Application  104
  10.4 Deployment Hierarchies  104
  10.5 Directory Structure  104
    10.5.1 Example of Application Directory Structure  106
  10.6 Web Application Archive File  106
  10.7 Web Application Deployment Descriptor  106
    10.7.1 Dependencies On Extensions  107
    10.7.2 Web Application Class Loader  107
  10.8 Replacing a Web Application  108
  10.9 Error Handling  108
    10.9.1 Request Attributes  108
    10.9.2 Error Pages  109
13.2 Declarative Security 126
13.3 Programmatic Security 126
13.4 Programmatic Security Policy Configuration 128
  13.4.1 @ServletSecurity Annotation 128
    13.4.1.1 Examples 132
    13.4.1.2 Mapping @ServletSecurity to security-constraint 133
    13.4.1.3 Mapping @HttpConstraint and @HttpMethodConstraint to XML. 135
  13.4.2 setServletSecurity of ServletRegistration.Dynamic 136
13.5 Roles 137
13.6 Authentication 138
  13.6.1 HTTP Basic Authentication 138
  13.6.2 HTTP Digest Authentication 138
  13.6.3 Form Based Authentication 139
    13.6.3.1 Login Form Notes 140
  13.6.4 HTTPS Client Authentication 141
  13.6.5 Additional Container Authentication Mechanisms 141
13.7 Server Tracking of Authentication Information 141
13.8 Specifying Security Constraints 142
  13.8.1 Combining Constraints 143
  13.8.2 Example 144
  13.8.3 Processing Requests 146
  13.8.4 Uncovered HTTP Protocol Methods 147
    13.8.4.1 Rules for Security Constraint Configuration 149
    13.8.4.2 Handling Uncovered HTTP Methods 149
13.9 Default Policies 150
13.10 Login and Logout 151

14. Deployment Descriptor 153
14.1 Deployment Descriptor Elements 153
14.2 Rules for Processing the Deployment Descriptor 154
14.3 Deployment Descriptor 155
14.4 Deployment Descriptor Diagram 155
14.5 Examples 178
  14.5.1 A Basic Example 179
  14.5.2 An Example of Security 180

15. Requirements related to other Specifications 183
15.1 Sessions 183
15.2 Web Applications 183
  15.2.1 Web Application Class Loader 183
  15.2.2 Web Application Environment 184
  15.2.3 JNDI Name for Web Module Context Root URL 184
15.3 Security 185
  15.3.1 Propagation of Security Identity in EJB™ Calls 186
  15.3.2 Container Authorization Requirements 186
  15.3.3 Container Authentication Requirements 186
15.4 Deployment 187
  15.4.1 Deployment Descriptor Elements 187
  15.4.2 Packaging and Deployment of JAX-WS Components 187
  15.4.3 Rules for Processing the Deployment Descriptor 189
15.5 Annotations and Resource Injection 189
  15.5.1 @DeclareRoles 190
  15.5.2 @EJB Annotation 191
  15.5.3 @EJBs Annotation 192
  15.5.4 @Resource Annotation 192
  15.5.5 @PersistenceContext Annotation 193
  15.5.6 @PersistenceContexts Annotation 193
15.5.7 @PersistenceUnit Annotation 194
15.5.8 @PersistenceUnits Annotation 194
15.5.9 @PostConstruct Annotation 194
15.5.10 @PreDestroy Annotation 195
15.5.11 @Resources Annotation 195
15.5.12 @RunAs Annotation 196
15.5.13 @WebServiceRef Annotation 197
15.5.14 @WebServiceRefs Annotation 197
15.5.15 Contexts and Dependency Injection for Java EE requirements 197

A. Change Log 199
A.1 Changes since Servlet 3.0 199
A.2 Changes since Servlet 3.0 Proposed Final Draft 201
A.3 Changes since Servlet 3.0 Public Review 202
A.4 Changes since Servlet 3.0 EDR 202
A.5 Changes since Servlet 2.5 MR6 202
A.6 Changes since Servlet 2.5 MR 5 203
  A.6.1 Clarify SRV 8.4 "The Forward Method" 203
  A.6.2 Update Deployment descriptor "http-method values allowed" 203
A.6.3 Clarify SRV 7.7.1 "Threading Issues" 204
A.7 Changes Since Servlet 2.5 MR 2 204
  A.7.1 Updated Annotation Requirements for Java EE containers 204
  A.7.2 Updated Java Enterprise Edition Requirements 204
  A.7.3 Clarified HttpServletRequest.getRequestURL() 204
  A.7.4 Removal of IllegalStateException from HttpSession.getId() 205
  A.7.5 ServletContext.getContextPath() 205
  A.7.6 Requirement for web.xml in web applications 206
A.8 Changes Since Servlet 2.4 206
A.8.1 Session Clarification 206
A.8.2 Filter All Dispatches 207
A.8.3 Multiple Occurrences of Servlet Mappings 207
A.8.4 Multiple Occurrences Filter Mappings 208
A.8.5 Support Alternative HTTP Methods with Authorization Constraints 209
A.8.6 Minimum J2SE Requirement 210
A.8.7 Annotations and Resource Injection 210
A.8.8 SRV.9.9 ("Error Handling") Requirement Removed 210
A.8.9 HttpServletRequest.isRequestedSessionIdValid() Clarification 210
A.8.10 SRV.5.5 ("Closure of Response Object") Clarification 210
A.8.11 ServletRequest.setCharacterEncoding() Clarified 211
A.8.12 Java Enterprise Edition Requirements 211
A.8.13 Servlet 2.4 MR Change Log Updates Added 211
A.8.14 Synchronized Access Session Object Clarified 211
A.9 Changes Since Servlet 2.3 211
Overview

1.1 What is a Servlet?

A servlet is a Java™ technology-based Web component, managed by a container, that generates dynamic content. Like other Java technology-based components, servlets are platform-independent Java classes that are compiled to platform-neutral byte code that can be loaded dynamically into and run by a Java technology-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.

1.2 What is a Servlet Container?

The servlet container is a part of a Web server or application server that provides the network services over which requests and responses are sent, decodes MIME-based requests, and formats MIME-based responses. A servlet container also contains and manages servlets through their lifecycle.

A servlet container can be built into a host Web server, or installed as an add-on component to a Web Server via that server’s native extension API. Servlet containers can also be built into or possibly installed into Web-enabled application servers.

All servlet containers must support HTTP as a protocol for requests and responses, but additional request/response-based protocols such as HTTPS (HTTP over SSL) may be supported. The required versions of the HTTP specification that a container must implement are HTTP/1.0 and HTTP/1.1. Because the container may have a caching mechanism described in RFC2616 (HTTP/1.1), it may modify requests from...
the clients before delivering them to the servlet, may modify responses produced by servlets before sending them to the clients, or may respond to requests without delivering them to the servlet under the compliance with RFC2616.

A servlet container may place security restrictions on the environment in which a servlet executes. In a Java Platform, Standard Edition (J2SE, v.1.3 or above) or Java Platform, Enterprise Edition (Java EE, v.1.3 or above) environment, these restrictions should be placed using the permission architecture defined by the Java platform. For example, high-end application servers may limit the creation of a Thread object to insure that other components of the container are not negatively impacted.

Java SE 7 is the minimum version of the underlying Java platform with which servlet containers must be built.

1.3 An Example

The following is a typical sequence of events:

1. A client (e.g., a Web browser) accesses a Web server and makes an HTTP request.

2. The request is received by the Web server and handed off to the servlet container. The servlet container can be running in the same process as the host Web server, in a different process on the same host, or on a different host from the Web server for which it processes requests.

3. The servlet container determines which servlet to invoke based on the configuration of its servlets, and calls it with objects representing the request and response.

4. The servlet uses the request object to find out who the remote user is, what HTTP POST parameters may have been sent as part of this request, and other relevant data. The servlet performs whatever logic it was programmed with, and generates data to send back to the client. It sends this data back to the client via the response object.

5. Once the servlet has finished processing the request, the servlet container ensures that the response is properly flushed, and returns control back to the host Web server.
1.4 Comparing Servlets with Other Technologies

In functionality, servlets lie somewhere between Common Gateway Interface (CGI) programs and proprietary server extensions such as the Netscape Server API (NSAPI) or Apache Modules.

Servlets have the following advantages over other server extension mechanisms:
- They are generally much faster than CGI scripts because a different process model is used.
- They use a standard API that is supported by many Web servers.
- They have all the advantages of the Java programming language, including ease of development and platform independence.
- They can access the large set of APIs available for the Java platform.

1.5 Relationship to Java Platform, Enterprise Edition

The Java Servlet API v.3.1 is a required API of the Java Platform, Enterprise Edition, 71. Servlet containers and servlets deployed into them must meet additional requirements, described in the Java EE specification, for executing in a Java EE environment.

---

1.6 Compatibility with Java Servlet Specification Version 2.5

1.6.1 Processing annotations

In Servlet 2.5, *metadata-complete* only affected the scanning of annotations at deployment time. The notion of web-fragments did not exist in servlet 2.5. However in servlet 3.0 and later, *metadata-complete* affects scanning of all annotations that specify deployment information and web-fragments at deployment time. The version of the descriptor MUST not affect which annotations you scan for in a web application. An implementation of a particular version of the specification MUST scan for all annotations supported in that configuration, unless *metadata-complete* is specified.
The Servlet Interface

The Servlet interface is the central abstraction of the Java Servlet API. All servlets implement this interface either directly, or more commonly, by extending a class that implements the interface. The two classes in the Java Servlet API that implement the Servlet interface are GenericServlet and HttpServlet. For most purposes, Developers will extend HttpServlet to implement their servlets.

2.1 Request Handling Methods

The basic Servlet interface defines a service method for handling client requests. This method is called for each request that the servlet container routes to an instance of a servlet.

The handling of concurrent requests to a Web application generally requires that the Web Developer design servlets that can deal with multiple threads executing within the service method at a particular time.

Generally the Web container handles concurrent requests to the same servlet by concurrent execution of the service method on different threads.

2.1.1 HTTP Specific Request Handling Methods

The HttpServlet abstract subclass adds additional methods beyond the basic Servlet interface that are automatically called by the service method in the HttpServlet class to aid in processing HTTP-based requests. These methods are:

- doGet for handling HTTP GET requests
- doPost for handling HTTP POST requests
- doGet for handling HTTP PUT requests
- doDelete for handling HTTP DELETE requests
The **doHead** method for handling HTTP HEAD requests
The **doOptions** method for handling HTTP OPTIONS requests
The **doTrace** method for handling HTTP TRACE requests

Typically when developing HTTP-based servlets, a Servlet Developer will only concern himself with the **doGet** and **doPost** methods. The other methods are considered to be methods for use by programmers very familiar with HTTP programming.

### 2.1.2 Additional Methods

The **doPut** and **doDelete** methods allow Servlet Developers to support HTTP/1.1 clients that employ these features. The **doHead** method in HttpServlet is a specialized form of the **doGet** method that returns only the headers produced by the **doGet** method. The **doOptions** method responds with which HTTP methods are supported by the servlet. The **doTrace** method generates a response containing all instances of the headers sent in the TRACE request.

### 2.1.3 Conditional GET Support

The HttpServlet interface defines the **getLastModified** method to support conditional GET operations. A conditional GET operation requests a resource be sent only if it has been modified since a specified time. In appropriate situations, implementation of this method may aid efficient utilization of network resources.

### 2.2 Number of Instances

The servlet declaration which is either via the annotation as described in Chapter 8, "Annotations and pluggability" or part of the deployment descriptor of the Web application containing the servlet, as described in Chapter 14, "Deployment Descriptor", controls how the servlet container provides instances of the servlet.

For a servlet not hosted in a distributed environment (the default), the servlet container must use only one instance per servlet declaration. However, for a servlet implementing the SingleThreadModel interface, the servlet container may instantiate multiple instances to handle a heavy request load and serialize requests to a particular instance.
In the case where a servlet was deployed as part of an application marked in the deployment descriptor as distributable, a container may have only one instance per servlet declaration per Java Virtual Machine (JVM™)\(^1\). However, if the servlet in a distributable application implements the SingleThreadModel interface, the container may instantiate multiple instances of that servlet in each JVM of the container.

2.2.1 Note About The Single Thread Model

The use of the SingleThreadModel interface guarantees that only one thread at a time will execute in a given servlet instance’s service method. It is important to note that this guarantee only applies to each servlet instance, since the container may choose to pool such objects. Objects that are accessible to more than one servlet instance at a time, such as instances of HttpSession, may be available at any particular time to multiple servlets, including those that implement SingleThreadModel.

It is recommended that a developer take other means to resolve those issues instead of implementing this interface, such as avoiding the usage of an instance variable or synchronizing the block of the code accessing those resources. The SingleThreadModel Interface is deprecated in this version of the specification.

2.3 Servlet Life Cycle

A servlet is managed through a well defined life cycle that defines how it is loaded and instantiated, is initialized, handles requests from clients, and is taken out of service. This life cycle is expressed in the API by the init, service, and destroy methods of the javax.servlet.Servlet interface that all servlets must implement directly or indirectly through the GenericServlet or HttpServlet abstract classes.

2.3.1 Loading and Instantiation

The servlet container is responsible for loading and instantiating servlets. The loading and instantiation can occur when the container is started, or delayed until the container determines the servlet is needed to service a request.

---

1. The terms “Java virtual machine” and “JVM” mean a virtual machine for the Java(TM) platform.
When the servlet engine is started, needed servlet classes must be located by the servlet container. The servlet container loads the servlet class using normal Java class loading facilities. The loading may be from a local file system, a remote file system, or other network services.

After loading the Servlet class, the container instantiates it for use.

2.3.2 Initialization

After the servlet object is instantiated, the container must initialize the servlet before it can handle requests from clients. Initialization is provided so that a servlet can read persistent configuration data, initialize costly resources (such as JDBC™ API-based connections), and perform other one-time activities. The container initializes the servlet instance by calling the init method of the Servlet interface with a unique (per servlet declaration) object implementing the ServletConfig interface. This configuration object allows the servlet to access name-value initialization parameters from the Web application’s configuration information. The configuration object also gives the servlet access to an object (implementing the ServletContext interface) that describes the servlet’s runtime environment. See Chapter 4, “ServletContext” for more information about the ServletContext interface.

2.3.2.1 Error Conditions on Initialization

During initialization, the servlet instance can throw an UnavailableException or a ServletException. In this case, the servlet must not be placed into active service and must be released by the servlet container. The destroy method is not called as it is considered unsuccessful initialization.

A new instance may be instantiated and initialized by the container after a failed initialization. The exception to this rule is when an UnavailableException indicates a minimum time of unavailability, and the container must wait for the period to pass before creating and initializing a new servlet instance.

2.3.2.2 Tool Considerations

The triggering of static initialization methods when a tool loads and introspects a Web application is to be distinguished from the calling of the init method. Developers should not assume a servlet is in an active container runtime until the init method of the Servlet interface is called. For example, a servlet should not try to establish connections to databases or Enterprise JavaBeans™ containers when only static (class) initialization methods have been invoked.
2.3.3 Request Handling

After a servlet is properly initialized, the servlet container may use it to handle client requests. Requests are represented by request objects of type ServletRequest. The servlet fills out response to requests by calling methods of a provided object of type ServletResponse. These objects are passed as parameters to the service method of the Servlet interface.

In the case of an HTTP request, the objects provided by the container are of types HttpServletRequest and HttpServletResponse.

Note that a servlet instance placed into service by a servlet container may handle no requests during its lifetime.

2.3.3.1 Multithreading Issues

A servlet container may send concurrent requests through the service method of the servlet. To handle the requests, the Servlet Developer must make adequate provisions for concurrent processing with multiple threads in the service method.

Although it is not recommended, an alternative for the Developer is to implement the SingleThreadModel interface which requires the container to guarantee that there is only one request thread at a time in the service method. A servlet container may satisfy this requirement by serializing requests on a servlet, or by maintaining a pool of servlet instances. If the servlet is part of a Web application that has been marked as distributable, the container may maintain a pool of servlet instances in each JVM that the application is distributed across.

For servlets not implementing the SingleThreadModel interface, if the service method (or methods such as doGet or doPost which are dispatched to the service method of the HttpServlet abstract class) has been defined with the synchronized keyword, the servlet container cannot use the instance pool approach, but must serialize requests through it. It is strongly recommended that Developers not synchronize the service method (or methods dispatched to it) in these circumstances because of detrimental effects on performance.

2.3.3.2 Exceptions During Request Handling

A servlet may throw either a ServletException or an UnavailableException during the service of a request. A ServletException signals that some error occurred during the processing of the request and that the container should take appropriate measures to clean up the request.

An UnavailableException signals that the servlet is unable to handle requests either temporarily or permanently.
If a permanent unavailability is indicated by the `UnavailableException`, the servlet container must remove the servlet from service, call its `destroy` method, and release the servlet instance. Any requests refused by the container by that cause must be returned with a `SC_NOT_FOUND` (404) response.

If temporary unavailability is indicated by the `UnavailableException`, the container may choose to not route any requests through the servlet during the time period of the temporary unavailability. Any requests refused by the container during this period must be returned with a `SC_SERVICE_UNAVAILABLE` (503) response status along with a `Retry-After` header indicating when the unavailability will terminate.

The container may choose to ignore the distinction between a permanent and temporary unavailability and treat all `UnavailableExceptions` as permanent, thereby removing a servlet that throws any `UnavailableException` from service.

### 2.3.3.3 Asynchronous processing

Some times a filter and/or servlet is unable to complete the processing of a request without waiting for a resource or event before generating a response. For example, a servlet may need to wait for an available JDBC connection, for a response from a remote web service, for a JMS message, or for an application event, before proceeding to generate a response. Waiting within the servlet is an inefficient operation as it is a blocking operation that consumes a thread and other limited resources. Frequently a slow resource such as a database may have many threads blocked waiting for access and can cause thread starvation and poor quality of service for an entire web container.

The asynchronous processing of requests is introduced to allow the thread may return to the container and perform other tasks. When asynchronous processing begins on the request, another thread or callback may either generate the response and call `complete` or dispatch the request so that it may run in the context of the container using the `AsyncContext.dispatch` method. A typical sequence of events for asynchronous processing is:

1. The request is received and passed via normal filters for authentication etc. to the servlet.
2. The servlet processes the request parameters and/or content to determine the nature of the request.
3. The servlet issues requests for resources or data, for example, sends a remote web service request or joins a queue waiting for a JDBC connection.
4. The servlet returns without generating a response.
5. After some time, the requested resource becomes available, the thread handling that event continues processing either in the same thread or by dispatching to a resource in the container using the `AsyncContext`.
Java Enterprise Edition features such as Section 15.2.2, “Web Application Environment” on page 15-184 and Section 15.3.1, “Propagation of Security Identity in EJB™ Calls” on page 15-186 are available only to threads executing the initial request or when the request is dispatched to the container via the AsyncContext.dispatch method. Java Enterprise Edition features may be available to other threads operating directly on the response object via the AsyncContext.start(Runnable) method.

The @WebServlet and @WebFilter annotations described in Chapter 8 have an attribute – asyncSupported that is a boolean with a default value of false. When asyncSupported is set to true the application can start asynchronous processing in a separate thread by calling startAsync (see below), passing it a reference to the request and response objects, and then exit from the container on the original thread. This means that the response will traverse (in reverse order) the same filters (or filter chain) that were traversed on the way in. The response isn't committed till complete (see below) is called on the AsyncContext. The application is responsible to handle concurrent access to the request and response objects if the async task is executing before the container-initiated dispatch that called startAsync has returned to the container.

Dispatching from a servlet that has asyncSupported=true to one where asyncSupported is set to false is allowed. In this case, the response will be committed when the service method of the servlet that does not support async is exited, and it is the container's responsibility to call complete on the AsyncContext so that any interested AsyncListener instances will be notified. The AsyncListener.onComplete notification should also be used by filters as a mechanism to clear up resources that it has been holding on to for the async task to complete.

Dispatching from a synchronous servlet to an asynchronous servlet would be illegal. However the decision of throwing an IllegalStateException is deferred to the point when the application calls startAsync. This would allow a servlet to either function as a synchronous or an asynchronous servlet.

The async task that the application is waiting for could write directly to the response, on a different thread than the one that was used for the initial request. This thread knows nothing about any filters. If a filter wanted to manipulate the response in the new thread, it would have to wrap the response when it was processing the initial request "on the way in", and passed the wrapped response to the next filter in the chain, and eventually to the servlet. So if the response was wrapped (possibly multiple times, once per filter), and the application processes the request and writes directly to the response, it is really writing to the response wrapper(s), i.e., any output added to the response will still be processed by the response wrapper(s). When an application reads from a request in a separate thread, and adds output to the response, it really reads from the request wrapper(s), and writes to the response wrapper(s), so any input and/or output manipulation intended by the wrapper(s) will continue to occur.
Alternately if the application chooses to do so it can use the AsyncContext to dispatch the request from the new thread to a resource in the container. This would enable using content generation technologies like JSP within the scope of the container.

In addition to the annotation attributes we have the following methods / classes added:

- ServletRequest

  - public AsyncContext startAsync(ServletRequest req, ServletResponse res). This method puts the request into asynchronous mode and initializes it’s AsyncContext with the given request and response objects and the time out returned by getAsyncTimeout. The ServletRequest and ServletResponse parameters MUST be either the same objects as were passed to the calling servlet’s service, or the filter’s doFilter method, or be subclasses of ServletRequestWrapper or ServletResponseWrapper classes that wrap them.

  A call to this method ensures that the response isn’t committed when the application exits out of the service method. It is committed when AsyncContext.complete is called on the returned AsyncContext or the AsyncContext times out and there are no listeners associated to handle the time out. The timer for async time outs will not start until the request and it’s associated response have returned from the container. The AsyncContext could be used to write to the response from the async thread. It can also be used to just notify that the response is not closed and committed.

  It is illegal to call startAsync if the request is within the scope of a servlet or filter that does not support asynchronous operations, or if the response has been committed and closed, or is called again during the same dispatch. The AsyncContext returned from a call to startAsync can then be used for further asynchronous processing. Calling the AsyncContext.hasOriginalRequestResponse() on the returned AsyncContext will return false, unless the passed ServletRequest and ServletResponse arguments are the original ones or do not carry application provided wrappers. Any filters invoked in the outboud direction after this request was put into asynchronous mode MAY use this as an indication that some of the request and / or response wrappers that they added during their inbound invocation MAY need to stay in place for the duration of the asynchronous operation, and their associated resources MAY not be released. A ServletRequestWrapper applied during the inbound invocation of a filter MAY be released by the outboud invocation of the filter only if the given ServletRequest which is used to initialize the AsyncContext and will be returned by a call to AsyncContext.getRequest(), does not contain the said ServletRequestWrapper. The same holds true for ServletResponseWrapper instances.
- public AsyncContext startAsync() is provided as a convenience that uses the original request and response objects for the async processing. Please note users of this method SHOULD flush the response if they are wrapped before calling this method if you wish, to ensure that any data written to the wrapped response isn’t lost.

- public AsyncContext getAsyncContext() - returns the AsyncContext that was created or re initialized by the invocation of startAsync. It is illegal to call getAsyncContext if the request has not been put in asynchronous mode.

- public boolean isAsyncSupported() - Returns true if the request supports async processing, and false otherwise. Async support will be disabled as soon as the request has passed a filter or servlet that does not support async processing (either via the designated annotation or declaratively).

- public boolean isAsyncStarted() - Returns true if async processing has started on this request, and false otherwise. If this request has been dispatched using one of the AsyncContext.dispatch methods since it was put in asynchronous mode, or a call to AsyncContext.complete is made, this method returns false.

- public DispatcherType getDispatcherType() - Returns the dispatcher type of a request. The dispatcher type of a request is used by the container to select the filters that need to be applied to the request. Only filters with the matching dispatcher type and url patterns will be applied. Allowing a filter that has been configured for multiple dispatcher types to query a request for it’s dispatcher type allows the filter to process the request differently depending on it’s dispatcher type. The initial dispatcher type of a request is defined as DispatcherType.REQUEST. The dispatcher type of a request dispatched via RequestDispatcher.forward(ServletRequest, ServletResponse) or RequestDispatcher.include(ServletRequest, ServletResponse) is given as DispatcherType.FORWARD or DispatcherType.INCLUDE respectively, while a dispatcher type of an asynchronous request dispatched via one of the AsyncContext.dispatch methods is given as DispatcherType.ASYNC. Finally the dispatcher type of a request dispatched to an error page by the container’s error handling mechanism is given as DispatcherType.ERROR.

- AsyncContext - This class represents the execution context for the asynchronous operation that was started on the ServletRequest. An AsyncContext is created and initialized by a call to ServletRequest.startAsync as described above. The following methods are in the AsyncContext:

  - public ServletRequest getRequest() - returns the request that was used to initialize the AsyncContext by calling one of the startAsync methods. Calling getRequest when complete or any of the dispatch methods has been previously called in the asynchronous cycle will result in an IllegalStateException.

  - public ServletResponse getResponse() - returns the response that was used to initialize the AsyncContext by calling one of the startAsync methods. Calling getResponse when complete or any of the dispatch methods has been previously called in the asynchronous cycle will result in an IllegalStateException.
- public void setTimeout(long timeoutMilliseconds) - Sets the time out for the asynchronous processing to occur in milliseconds. A call to this method overrides the time out set by the container. If the time out is not specified via the call to setTimeout, 30000 is used as the default. A value of 0 or less indicates that the asynchronous operation will never time out. The time out applies to the AsyncContext once the container-initiated dispatch during which one of the ServletRequest.startAsync methods was called has returned to the container. It is illegal to set the time out if this method is called after the container-initiated dispatch on which the asynchronous cycle was started has returned to the container and will result in an IllegalStateException.

- public long getTimeout() - Gets the time out, in milliseconds, associated with the AsyncContext. This method returns the container’s default time out, or the time out value set via the most recent invocation of setTimeout method.

- public void addListener(AsyncListener listener, ServletRequest req, ServletResponse res) - Registers the given listener for notifications of onTimeout, onError, onComplete or onStartAsync. The first three are associated with the most recent asynchronous cycle started by calling one of the ServletRequest.startAsync methods. The onStartAsync is associated to a new asynchronous cycle via one of the ServletRequest.startAsync methods. Async listeners will be notified in the order in which they were added to the request. The request and response objects passed in to the method are the exact same ones that are available from the AsyncEvent.getSuppliedRequest() and AsyncEvent.getSuppliedResponse() when the AsyncListener is notified. These objects should not be read from or written to, because additional wrapping may have occurred since the given AsyncListener was registered, but may be used in order to release any resources associated with them. It is illegal to call this method after the container-initiated dispatch on which the asynchronous cycle was started has returned to the container and before a new asynchronous cycle was started and will result in an IllegalStateException.

- public <T extends AsyncListener> createListener(Class<T> clazz) - Instantiates the given AsyncListener class. The returned AsyncListener instance may be further customized before it is registered with the AsyncContext via a call to one of the addListener methods specified below. The given AsyncListener class MUST define a zero argument constructor, which is used to instantiate it. This method supports any annotations applicable to the AsyncListener.

- public void addListener(AsyncListener) - Registers the given listener for notifications of onTimeout, onError, onComplete or onStartAsync. The first three are associated with the most recent asynchronous cycle started by calling one of the ServletRequest.startAsync methods. The onStartAsync is associated to a new asynchronous cycle via one of the ServletRequest.startAsync methods. If startAsync(req, res) or startAsync() is called on the request, the exact same request and response objects are available from the AsyncEvent when the AsyncListener is notified. The request and response may or may not be wrapped. Async listeners will be notified in the order in which they were added to the request. It is illegal to call this method after the container-initiated
dispatch on which the asynchronous cycle was started has returned to the container and before a new asynchronous cycle was started and will result in an `IllegalStateException`.

- **public void dispatch(String path)** - Dispatches the request and response that were used to initialize the `AsyncContext` to the resource with the given path. The given path is interpreted as relative to the `ServletContext` that initialized the `AsyncContext`. All path related query methods of the request MUST reflect the dispatch target, while the original request URI, context path, path info and query string may be obtained from the request attributes as defined in Section 9.7.2, “Dispatched Request Parameters” on page 9-100. These attributes MUST always reflect the original path elements, even after multiple dispatches.

- **public void dispatch()** - Provided as a convenience to dispatch the request and response used to initialize the `AsyncContext` as follows. If the `AsyncContext` was initialized via the `startAsync(ServletRequest, ServletResponse)` and the request passed is an instance of `HttpServletRequest`, then the dispatch is to the URI returned by `HttpServletRequest.getRequestURI()`. Otherwise the dispatch is to the URI of the request when it was last dispatched by the container. The examples CODE EXAMPLE 2-1, CODE EXAMPLE 2-2 and CODE EXAMPLE 2-3 shown below demonstrate what the target URI of dispatch would be in the different cases.

**CODE EXAMPLE 2-1**

```java
// REQUEST to /url/A
AsyncContext ac = request.startAsync();
...
ac.dispatch(); // ASYNC dispatch to /url/A
```

**CODE EXAMPLE 2-2**

```java
// REQUEST to /url/A
// FORWARD to /url/B
request.getRequestDispatcher("/url/B").forward(request, response);
// Start async operation from within the target of the FORWARD
AsyncContext ac = request.startAsync();
ac.dispatch(); // ASYNC dispatch to /url/A
```

**CODE EXAMPLE 2-3**

```java
// REQUEST to /url/A
// FORWARD to /url/B
request.getRequestDispatcher("/url/B").forward(request, response);
```
public void dispatch(ServletContext context, String path) -
dispatches the request and response used to initialize the AsyncContext to the
resource with the given path in the given ServletContext.

For all the 3 variations of the dispatch methods defined above, calls to the
methods returns immediately after passing the request and response objects to
a container managed thread, on which the dispatch operation will be
performed. The dispatcher type of the request is set to ASYNC. Unlike
RequestDispatcher.forward(ServletRequest, ServletResponse)
dispatches, the response buffer and headers will not be reset, and it is legal to
dispatch even if the response has already been committed. Control over the
request and response is delegated to the dispatch target, and the response will
be closed when the dispatch target has completed execution, unless
ServletRequest.startAsync() or
ServletRequest.startAsync(ServletRequest, ServletResponse) is called.
If any of the dispatch methods are called before the container-initiated dispatch
that called startAsync has returned to the container, then the call will not take
effect until after the container-initiated dispatch has returned to the container.
Invocation of the AsyncListener.onComplete(AsyncEvent),
AsyncListener.onTimeout(AsyncEvent) and
AsyncListener.onError(AsyncEvent) will also be delayed till after the
container-initiated dispatch has returned to the container. There can be at most
one asynchronous dispatch operation per asynchronous cycle, which is started
by a call to one of the ServletRequest.startAsync methods. There can be at
most one asynchronous dispatch per asynchronous cycle, which is started by a
call to ServletRequest.startAsync. Any attempt to perform additional
asynchronous dispatch operation within the same asynchronous cycle is illegal
and will result in an IllegalStateException. If startAsync is subsequently
called on the dispatched request, then any of the dispatch methods may be
called with the same restriction as above.

Any errors or exceptions that may occur during the execution of the dispatch
methods MUST be caught and handled by the container as follows:

i. invoke the AsyncListener.onError(AsyncEvent) method for all instances
   of the AsyncListener registered with the ServletRequest for which the
   AsyncContext was created and make the Throwable available via the
   AsyncEvent.getThrowable().

ii. If none of the listeners called AsyncContext.complete or any of the
    AsyncContext.dispatch methods, then perform an error dispatch with a
    status code equal to HttpServletResponse.SC_INTERNAL_SERVER_ERROR
    and make the Throwable available as the value of the
    RequestDispatcher.ERROR_EXCEPTION request attribute.
iii. If no matching error page is found, or the error page does not call
  AsyncContext.complete() or any of the AsyncContext.dispatch methods,
  then the container MUST call AsyncContext.complete.

- public boolean hasOriginalRequestAndResponse() - This method checks if the
  AsyncContext was initialized with the original request and response
  objects by calling ServletRequest.startAsync() or if it was initialized by
  calling ServletRequest.startAsync(ServletRequest, ServletResponse)
  and neither the ServletRequest nor the ServletResponse argument carried
  any application-provided wrappers, in which case it returns true. If the
  AsyncContext was initialized with wrapped request and/or response objects
  using ServletRequest.startAsync(ServletRequest, ServletResponse), it
  returns false. This information may be used by filters invoked in the outbound
  direction, after a request was put into asynchronous mode, to determine
  whether any request and/or response wrappers that they added during their
  inbound invocation need to be preserved for the duration of the asynchronous
  operation or may be released.

- public void start(Runnable r) - This method causes the container to
  dispatch a thread, possibly from a managed thread pool, to run the specified
  Runnable. The container may propagate appropriate contextual information to
  the Runnable.

- public void complete() - If request.startAsync is called then this method
  MUST be called to complete the async processing and commit and close the
  response. The complete method can be invoked by the container if the request
  is dispatched to a servlet that does not support async processing, or the target
  servlet called by AsyncContext.dispatch does not do a subsequent call to
  startAsync. In this case, it is the container's responsibility to call complete()
  as soon as that servlet's service method is exited. An IllegalStateException
  MUST be thrown if startAsync was not called. It is legal to call this method
  anytime after a call to ServletRequest.startAsync() or
  ServletRequest.startAsync(ServletRequest, ServletResponse) and
  before a call to one of the dispatch methods. If this method is called before the
  container-initiated dispatch that called startAsync has returned to the
  container, then the call will not take effect until after the container-initiated
  dispatch has returned to the container. Invocation of the
  AsyncListener.onComplete(AsyncEvent) will also be delayed till after the
  container-initiated dispatch has returned to the container.

- ServletRequestWrapper

- public boolean isWrapperFor(ServletRequest req)- Checks recursively if
  this wrapper wraps the given ServletRequest and returns true if it does, else
  it returns false

- ServletResponseWrapper

- public boolean isWrapperFor(ServletResponse res)- Checks recursively if
  this wrapper wraps the given ServletResponse and returns true if it does,
  else it returns false.

- AsyncListener
- public void onComplete(AsyncEvent event) - Is used to notify the listener of completion of the asynchronous operation started on the ServletRequest.
- public void onTimeout(AsyncEvent event) - Is used to notify the listener of a time out of the asynchronous operation started on the ServletRequest.
- public void onError(AsyncEvent event) - Is used to notify the listener that the asynchronous operation has failed to complete.
- public void onStartAsync(AsyncEvent event) - Is used to notify the listener that a new asynchronous cycle is being initiated via a call to one of the ServletRequest.startAsync methods. The AsyncContext corresponding to the asynchronous operation that is being reinitialized may be obtained by calling AsyncEvent.getAsyncContext on the given event.

In the event that an asynchronous operation times out, the container must run through the following steps:

- Invoke the AsyncListener.onTimeout method on all the AsyncListener instances registered with the ServletRequest on which the asynchronous operation was initiated.
- If none of the listeners called AsyncContext.complete() or any of the AsyncContext.dispatch methods, perform an error dispatch with a status code equal to HttpServletResponse.SC_INTERNAL_SERVER_ERROR.
- If no matching error page was found, or the error page did not call AsyncContext.complete() or any of the AsyncContext.dispatch methods, the container MUST call AsyncContext.complete().
- If an exception is thrown while invoking methods in an AsyncListener, it is logged and will not affect the invocation of any other AsyncListeners.
- Async processing in JSP would not be supported by default as it is used for content generation and async processing would have to be done before the content generation. It is up to the container how to handle this case. Once all the async activities are done, a dispatch to the JSP page using the AsyncContext.dispatch can be used for generating content.
- Figure 2-1 shown below is a diagram depicting the state transitions for various asynchronous operations.
FIGURE 2-1 State transition diagram for asynchronous operations
2.3.3.4 Thread Safety

Other than the `startAsync` and `complete` methods, implementations of the request and response objects are not guaranteed to be thread safe. This means that they should either only be used within the scope of the request handling thread or the application must ensure that access to the request and response objects are thread safe.

If a thread created by the application uses the container-managed objects, such as the request or response object, those objects must be accessed only within the object’s life cycle as defined in sections Section 3.12, “Lifetime of the Request Object” on page 3-31 and Section 5.7, “Lifetime of the Response Object” on page 5-50 respectively. Be aware that other than the `startAsync`, and `complete` methods, the request and response objects are not thread safe. If those objects were accessed in the multiple threads, the access should be synchronized or be done through a wrapper to add the thread safety, for instance, synchronizing the call of the methods to access the request attribute, or using a local output stream for the response object within a thread.

2.3.3.5 Upgrade Processing

In HTTP/1.1, the Upgrade general-header allows the client to specify the additional communication protocols that it supports and would like to use. If the server finds it appropriate to switch protocols, then new protocols will be used in subsequent communication.

The servlet container provides an HTTP upgrade mechanism. However the servlet container itself does not have knowledge about the upgraded protocol. The protocol processing is encapsulated in the `HttpUpgradeHandler`. Data reading or writing between the servlet container and the `HttpUpgradeHandler` is in byte streams.

When an upgrade request is received, the servlet can invoke the `HttpServletRequest.upgrade` method, which starts the upgrade process. This method instantiates the given `HttpUpgradeHandler` class. The returned `HttpUpgradeHandler` instance may be further customized. The application prepares and sends an appropriate response to the client. After exiting the `service` method of the servlet, the servlet container completes the processing of all filters and marks the connection to be handled by the `HttpUpgradeHandler`. It then calls the `HttpUpgradeHandler`’s `init` method, passing a `WebConnection` to allow the protocol handler access to the data streams.

The servlet filters only process the initial HTTP request and response. They are not involved in subsequent communications. In other words, they are not invoked once the request has been upgraded.
The `HttpUpgradeHandler` may use non-blocking IO to consume and produce messages.

The Developer has the responsibility for thread safe access to the `ServletInputStream` and `ServletOutputStream` while processing HTTP upgrade.

When the upgrade processing is done, `HttpUpgradeHandler.destroy` will be invoked.

### 2.3.4 End of Service

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

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

Before the servlet container calls the `destroy` method, it must allow any threads that are currently running in the `service` method of the servlet to complete execution, or exceed a server-defined time limit.

Once the `destroy` method is called on a servlet instance, the container may not route other requests to that instance of the servlet. If the container needs to enable the servlet again, it must do so with a new instance of the servlet’s class.

After the `destroy` method completes, the servlet container must release the servlet instance so that it is eligible for garbage collection.
The Request

The request object encapsulates all information from the client request. In the HTTP protocol, this information is transmitted from the client to the server in the HTTP headers and the message body of the request.

### 3.1 HTTP Protocol Parameters

Request parameters for the servlet are the strings sent by the client to a servlet container as part of its request. When the request is an HttpServletRequest object, and conditions set out in "When Parameters Are Available" on page 24 are met, the container populates the parameters from the URI query string and POST-ed data.

The parameters are stored as a set of name-value pairs. Multiple parameter values can exist for any given parameter name. The following methods of the ServletRequest interface are available to access parameters:

- `getParameter`
- `getParameterNames`
- `getParameterValues`
- `getParameterMap`

The `getParameterValues` method returns an array of String objects containing all the parameter values associated with a parameter name. The value returned from the `getParameter` method must be the first value in the array of String objects returned by `getParameterValues`. The `getParameterMap` method returns a java.util.Map of the parameter of the request, which contains names as keys and parameter values as map values.

Data from the query string and the post body are aggregated into the request parameter set. Query string data is presented before post body data. For example, if a request is made with a query string of `a=hello` and a post body of `a=goodbye&a=world`, the resulting parameter set would be ordered `a=(hello, goodbye, world)`. 
Path parameters that are part of a GET request (as defined by HTTP 1.1) are not exposed by these APIs. They must be parsed from the `String` values returned by the `getRequestURI` method or the `getPathInfo` method.

### 3.1.1 When Parameters Are Available

The following are the conditions that must be met before post form data will be populated to the parameter set:

1. The request is an HTTP or HTTPS request.
2. The HTTP method is POST.
3. The content type is `application/x-www-form-urlencoded`.
4. The servlet has made an initial call of any of the `getParameter` family of methods on the request object.

If the conditions are not met and the post form data is not included in the parameter set, the post data must still be available to the servlet via the request object’s input stream. If the conditions are met, post form data will no longer be available for reading directly from the request object’s input stream.

### 3.2 File upload

Servlet container allows files to be uploaded when data is sent as `multipart/form-data`.

The servlet container provides `multipart/form-data` processing if any one of the following conditions is met.

- The servlet handling the request is annotated with the `@MultipartConfig` as defined in Section 8.1.5, “@MultipartConfig” on page 8-70.
- Deployment descriptors contain a `multipart-config` element for the servlet handling the request.

How data in a request of type `multipart/form-data` is made available depends on whether the servlet container provides `multipart/form-data` processing:

- If the servlet container provides `multipart/form-data` processing, the data is made available through the following methods in `HttpServletRequest`:
  - `public Collection<Part> getParts()`
  - `public Part getPart(String name)`
Each part provides access to the headers, content type related with it and the content via the `Part.getInputStream` method.

For parts with `form-data` as the `Content-Disposition`, but without a filename, the string value of the part will also be available through the `getParameter` and `getParameterValues` methods on `HttpServletRequest`, using the name of the part.

- If the servlet container does not provide the multi-part/form-data processing, the data will be available through the `HttpServletRequest.getInputStream`.

### 3.3 Attributes

Attributes are objects associated with a request. Attributes may be set by the container to express information that otherwise could not be expressed via the API, or may be set by a servlet to communicate information to another servlet (via the `RequestDispatcher`). Attributes are accessed with the following methods of the `ServletRequest` interface:

- `getAttribute`
- `getAttributeNames`
- `setAttribute`

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

Attribute names beginning with the prefixes of `java.` and `javax.` are reserved for definition by this specification. Similarly, attribute names beginning with the prefixes of `sun.`, `com.sun.`, `oracle` and `com.oracle` are reserved for definition by Oracle Corporation. It is suggested that all attributes placed in the attribute set be named in accordance with the reverse domain name convention suggested by the Java Programming Language Specification\(^1\) for package naming.

### 3.4 Headers

A servlet can access the headers of an HTTP request through the following methods of the `HttpServletRequest` interface:

- `getHeader`
- `getHeaders`
- `getHeaderNames`

---

\(^1\) The Java Programming Language Specification is available at [http://docs.oracle.com/javase/specs/](http://docs.oracle.com/javase/specs/)
The `getHeader` method returns a header given the name of the header. There can be multiple headers with the same name, e.g. Cache-Control headers, in an HTTP request. If there are multiple headers with the same name, the `getHeader` method returns the first header in the request. The `getHeaders` method allows access to all the header values associated with a particular header name, returning an Enumeration of String objects.

Headers may contain String representations of int or Date data. The following convenience methods of the `HttpServletRequest` interface provide access to header data in a one of these formats:

- `getIntHeader`
- `getDateHeader`

If the `getIntHeader` method cannot translate the header value to an int, a `NumberFormatException` is thrown. If the `getDateHeader` method cannot translate the header to a Date object, an `IllegalArgumentException` is thrown.

### 3.5 Request Path Elements

The request path that leads to a servlet servicing a request is composed of many important sections. The following elements are obtained from the request URI path and exposed via the request object:

- **Context Path:** The path prefix associated with the `ServletContext` that this servlet is a part of. If this context is the “default” context rooted at the base of the Web server’s URL name space, this path will be an empty string. Otherwise, if the context is not rooted at the root of the server’s name space, the path starts with a `/` character but does not end with a `/` character.

- **Servlet Path:** The path section that directly corresponds to the mapping which activated this request. This path starts with a `/` character except in the case where the request is matched with the ‘/’ or “” pattern, in which case it is an empty string.

- **PathInfo:** The part of the request path that is not part of the Context Path or the Servlet Path. It is either null if there is no extra path, or is a string with a leading ‘/’.

The following methods exist in the `HttpServletRequest` interface to access this information:

- `getContextPath`
- `getServletPath`
- `getPathInfo`

It is important to note that, except for URL encoding differences between the request URI and the path parts, the following equation is always true:
requestURI = contextPath + servletPath + pathInfo

To give a few examples to clarify the above points, consider the following:

<table>
<thead>
<tr>
<th>TABLE 3-1</th>
<th>Example Context Set Up</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Context Path</strong></td>
<td>/catalog</td>
</tr>
</tbody>
</table>
| **Servlet Mapping** | Pattern: /lawn/*  
Servlet: LawnServlet |
| **Servlet Mapping** | Pattern: /garden/*  
Servlet: GardenServlet |
| **Servlet Mapping** | Pattern: *.jsp  
Servlet: JPServlet |

The following behavior is observed:

<table>
<thead>
<tr>
<th>TABLE 3-2</th>
<th>Observed Path Element Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Request Path</strong></td>
<td><strong>Path Elements</strong></td>
</tr>
</tbody>
</table>
| /catalog/lawn/index.html | ContextPath: /catalog  
ServletPath: /lawn  
PathInfo: /index.html |
| /catalog/garden/implements/ | ContextPath: /catalog  
ServletPath: /garden  
PathInfo: /implements/ |
| /catalog/help/feedback.jsp | ContextPath: /catalog  
ServletPath: /help/feedback.jsp  
PathInfo: null |

### 3.6 Path Translation Methods

There are two convenience methods in the API which allow the Developer to obtain the file system path equivalent to a particular path. These methods are:

- ServletContext.getRealPath
- HttpServletRequest.getPathTranslated

The `getRealPath` method takes a `String` argument and returns a `String` representation of a file on the local file system to which a path corresponds. The `getPathTranslated` method computes the real path of the `pathInfo` of the request.
In situations where the servlet container cannot determine a valid file path for these methods, such as when the Web application is executed from an archive, on a remote file system not accessible locally, or in a database, these methods must return null. Resources inside the META-INF/resources directory of JAR file must be considered only if the container has unpacked them from their containing JAR file when a call to `getRealPath()` is made, and in this case MUST return the unpacked location.

### 3.7 Non Blocking IO

Non-blocking request processing in the Web Container helps improve the ever increasing demand for improved Web Container scalability, increase the number of connections that can simultaneously be handled by the Web Container. Non-blocking IO in the Servlet container allows developers to read data as it becomes available or write data when possible to do so. Non-blocking IO only works with async request processing in Servlets and Filters (as defined in Section 2.3.3.3, “Asynchronous processing” on page 2-10), and upgrade processing (as defined in Section 2.3.3.5, “Upgrade Processing” on page 2-20). Otherwise, an `IllegalStateException` must be thrown when `ServletInputStream.setReadListener` or `ServletOutputStream.setWriteListener` is invoked.

The `ReadListener` provides the following callback methods for non blocking IO -

- **ReadListener**
  - onDataAvailable(). The `onDataAvailable` method is invoked on the `ReadListener` when data is available to read from the incoming request stream. The container will invoke the method the first time when data is available to read. The container will subsequently invoke the `onDataAvailable` method if and only if `isReady` method on `ServletInputStream`, described below, returns false.
  - onAllDataRead(). The `onAllDataRead` method is invoked when you have finished reading all the data for the `ServletRequest` for which the listener was registered.
  - onError(Throwable t). The `onError` method is invoked if there is any error or exception that occurs while processing the request.

The Servlet container must access methods in `ReadListener` in a thread safe manner.

In addition to the `ReadListener` defined above, the following methods have been added to `ServletInputStream` class -

- `ServletInputStream`
- boolean isFinished(). The isFinished method returns true when all the data for the request associated with the ServletInputStream has been read. Otherwise it returns false.

- boolean isReady(). The isReady method returns true if data can be read without blocking. If no data can be read without blocking it returns false. If isReady returns false it is illegal to call the read method and an IllegalStateException MUST be thrown.

- void setReadListener(ReadListener listener). Sets the ReadListener defined above to get invoked to read data in a non-blocking fashion. Once the listener is associated for the given ServletInputStream, the container invokes the methods on the ReadListener when data is available to read, all the data has been read or if there was an error processing the request. Registering a ReadListener will start non-blocking IO. It is illegal to switch to the traditional blocking IO at that point and an IllegalStateException MUST be thrown. A subsequent call to setReadListener in the scope of the current request is illegal and an IllegalStateException MUST be thrown.

3.8 Cookies

The HttpServletRequest interface provides the getCookies method to obtain an array of cookies that are present in the request. These cookies are data sent from the client to the server on every request that the client makes. Typically, the only information that the client sends back as part of a cookie is the cookie name and the cookie value. Other cookie attributes that can be set when the cookie is sent to the browser, such as comments, are not typically returned. The specification also allows for the cookies to be HttpOnly cookies. HttpOnly cookies indicate to the client that they should not be exposed to client-side scripting code (It's not filtered out unless the client knows to look for this attribute). The use of HttpOnly cookies helps mitigate certain kinds of cross-site scripting attacks.
3.9 SSL Attributes

If a request has been transmitted over a secure protocol, such as HTTPS, this information must be exposed via the `isSecure` method of the `ServletRequest` interface. The Web container must expose the following attributes to the servlet programmer:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Attribute Name</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>cipher suite</td>
<td>javax.servlet.request.cipher_suite</td>
<td>String</td>
</tr>
<tr>
<td>bit size of the algorithm</td>
<td>javax.servlet.request.key_size</td>
<td>Integer</td>
</tr>
<tr>
<td>SSL session id</td>
<td>javax.servlet.request.ssl_session_id</td>
<td>String</td>
</tr>
</tbody>
</table>

If there is an SSL certificate associated with the request, it must be exposed by the servlet container to the servlet programmer as an array of objects of type `java.security.cert.X509Certificate` and accessible via a `ServletRequest` attribute of `javax.servlet.request.X509Certificate`.

The order of this array is defined as being in ascending order of trust. The first certificate in the chain is the one set by the client, the next is the one used to authenticate the first, and so on.

3.10 Internationalization

Clients may optionally indicate to a Web server what language they would prefer the response be given in. This information can be communicated from the client using the `Accept-Language` header along with other mechanisms described in the HTTP/1.1 specification. The following methods are provided in the `ServletRequest` interface to determine the preferred locale of the sender:

- `getLocale`
- `getLocales`

The `getLocale` method will return the preferred locale for which the client wants to accept content. See section 14.4 of RFC 2616 (HTTP/1.1) for more information about how the `Accept-Language` header must be interpreted to determine the preferred language of the client.
The `getLocales` method will return an `Enumeration` of `Locale` objects indicating, in decreasing order starting with the preferred locale, the locales that are acceptable to the client.

If no preferred locale is specified by the client, the locale returned by the `getLocale` method must be the default locale for the servlet container and the `getLocales` method must contain an enumeration of a single `Locale` element of the default locale.

### 3.11 Request data encoding

Currently, many browsers do not send a `char` encoding qualifier with the `Content-Type` header, leaving open the determination of the character encoding for reading HTTP requests. The default encoding of a request the container uses to create the request reader and parse POST data must be “ISO-8859-1” if none has been specified by the client request. However, in order to indicate to the developer, in this case, the failure of the client to send a character encoding, the container returns `null` from the `getCharacterEncoding` method.

If the client hasn’t set character encoding and the request data is encoded with a different encoding than the default as described above, breakage can occur. To remedy this situation, a new method `setCharacterEncoding(String enc)` has been added to the `ServletRequest` interface. Developers can override the character encoding supplied by the container by calling this method. It must be called prior to parsing any post data or reading any input from the request. Calling this method once data has been read will not affect the encoding.

### 3.12 Lifetime of the Request Object

Each request object is valid only within the scope of a servlet’s `service` method, or within the scope of a filter’s `doFilter` method, unless the asynchronous processing is enabled for the component and the `startAsync` method is invoked on the request object. In the case where asynchronous processing occurs, the request object remains valid until `complete` is invoked on the `AsyncContext`. 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 for which `startAsync` has not been called outside the scope described above is not recommended as it may have indeterminate results.

In case of upgrade, the above is still true.
4.1 Introduction to the ServletContext Interface

The ServletContext interface defines a servlet’s view of the Web application within which the servlet is running. The Container Provider is responsible for providing an implementation of the ServletContext interface in the servlet container. Using the ServletContext object, a servlet can log events, obtain URL references to resources, and set and store attributes that other servlets in the context can access.

A ServletContext is rooted at a known path within a Web server. For example, a servlet context could be located at http://www.mycorp.com/catalog. All requests that begin with the /catalog request path, known as the context path, are routed to the Web application associated with the ServletContext.

4.2 Scope of a ServletContext Interface

There is one instance object of the ServletContext interface associated with each Web application deployed into a container. In cases where the container is distributed over many virtual machines, a Web application will have an instance of the ServletContext for each JVM.
Servlets in a container that were not deployed as part of a Web application are implicitly part of a “default” Web application and have a default ServletContext. In a distributed container, the default ServletContext is non-distributable and must only exist in one JVM.

4.3 Initialization Parameters

The following methods of the ServletContext interface allow the servlet access to context initialization parameters associated with a Web application as specified by the Application Developer in the deployment descriptor:

- `getInitParameter`
- `getInitParameterNames`

Initialization parameters are used by an Application Developer to convey setup information. Typical examples are a Webmaster’s e-mail address, or the name of a system that holds critical data.

4.4 Configuration methods

The following methods are added to ServletContext since Servlet 3.0 to enable programmatic definition of servlets, filters and the url pattern that they map to. These methods can only be called during the initialization of the application either from the `contextInitialized` method of a ServletContextListener implementation or from the `onStartup` method of a ServletContainerInitializer implementation. In addition to adding Servlets and Filters, one can also look up an instance of a Registration object corresponding to a Servlet or Filter or a map of all the Registration objects for the Servlets or Filters. If the ServletContext passed to the ServletContextListener’s `contextInitialized` method where the ServletContextListener was neither declared in web.xml or web-fragment.xml nor annotated with @WebListener then an `UnsupportedOperationException` MUST be thrown for all the methods defined in ServletContext for programmatic configuration of servlets, filters and listeners.
4.4.1 Programmatically adding and configuring Servlets

The ability to programmatically add a servlet to a context is useful for framework developers. For example a framework could declare a controller servlet using this method. The return value of this method is a ServletRegistration or a ServletRegistration.Dynamic object which further allows you to setup the other parameters of the servlet like init-params, url-mappings etc. There are three overloaded versions of the method as described below.

4.4.1.1 addServlet(String servletName, String className)

This method allows the application to declare a servlet programmatically. It adds the servlet with the given name, and class name to the servlet context.

4.4.1.2 addServlet(String servletName, Servlet servlet)

This method allows the application to declare a servlet programmatically. It adds the servlet with the given name, and servlet instance to the servlet context.

4.4.1.3 addServlet(String servletName, Class<? extends Servlet> servletClass)

This method allows the application to declare a servlet programmatically. It adds the servlet with the given name, and an instance of the servlet class to the servlet context.

4.4.1.4 <T extends Servlet> T createServlet(Class<T> clazz)

This method instantiates the given Servlet class. The method must support all the annotations applicable to Servlets except @WebServlet. The returned Servlet instance may be further customized before it is registered with the ServletContext via a call to addServlet(String, Servlet) as defined above.
4.4.1.5 ServletRegistration getServletRegistration(String servletName)

This method returns the ServletRegistration corresponding to the servlet with the given name, or null if no ServletRegistration exists under that name. An UnsupportedOperationException is thrown if the ServletContext was passed to the contextInitialized method of a ServletContextListener that was neither declared in the web.xml or web-fragment.xml, nor annotated with javax.servlet.annotation.WebListener.

4.4.1.6 Map<String, ? extends ServletRegistration> getServletRegistrations()

This method returns a map of ServletRegistration objects, keyed by name corresponding to all servlets registered with the ServletContext. If there are no servlets registered with the ServletContext an empty map is returned. The returned Map includes the ServletRegistration objects corresponding to all declared and annotated servlets, as well as the ServletRegistration objects corresponding to all servlets that have been added via one of the addServlet methods. Any changes to the returned Map MUST not affect the ServletContext. An UnsupportedOperationException is thrown if the ServletContext was passed to the contextInitialized method of a ServletContextListener that was neither declared in the web.xml or web-fragment.xml, nor annotated with javax.servlet.annotation.WebListener.

4.4.2 Programatically adding and configuring Filters

4.4.2.1 addFilter(String filterName, String className)

This method allows the application to declare a filter programatically. It adds the filter with the given name, and class name to the web application.

4.4.2.2 addFilter(String filterName, Filter filter)

This method allows the application to declare a filter programatically. It adds the filter with the given name, and filter instance to the web application.
4.4.2.3  addFilter(String filterName, Class <? extends Filter> filterClass)

This method allows the application to declare a filter programmatically. It adds the filter with the given name, and an instance of the filter class to the web application.

4.4.2.4  <T extends Filter> T createFilter(Class<T> clazz)

This method instantiates the given Filter class. The method must support all the annotations applicable to Filters. The returned Filter instance may be further customized before it is registered with the ServletContext via a call to addFilter(String, Filter) as defined above. The given Filter class must define a zero argument constructor, which is used to instantiate it.

4.4.2.5  FilterRegistration getFilterRegistration(String filterName)

This method returns the FilterRegistration corresponding to the filter with the given name, or null if no FilterRegistration exists under that name. An UnsupportedOperationException is thrown if the ServletContext was passed to the contextInitialized method of a ServletContextListener that was neither declared in the web.xml or web-fragment.xml, nor annotated with javax.servlet.annotation.WebListener.

4.4.2.6  Map<String, ? extends FilterRegistration> getServletRegistrations()

This method returns a map of ServletRegistration objects, keyed by name corresponding to all filters registered with the ServletContext. If there are no filters registered with the ServletContext an empty Map is returned. The returned Map includes the FilterRegistration objects corresponding to all declared and annotated filters, as well as the FilterRegistration objects corresponding to all filters that have been added via one of the addFilter methods. Any changes to the returned Map MUST not affect the ServletContext. An UnsupportedOperationException is thrown if the ServletContext was passed to the contextInitialized method of a ServletContextListener that was neither declared in the web.xml or web-fragment.xml, nor annotated with javax.servlet.annotation.WebListener.
4.4.3 Programatically adding and configuring Listeners

4.4.3.1 void addListener(String className)

Add the listener with the given class name to the ServletContext. The class with the given name will be loaded using the classloader associated with the application represented by the ServletContext, and MUST implement one or more of the following interfaces:

- javax.servlet.ServletContextAttributeListener
- javax.servlet.ServletRequestListener
- javax.servlet.ServletRequestAttributeListener
- javax.servlet.http.HttpSessionListener
- javax.servlet.http.HttpSessionAttributeListener
- javax.servlet.http.HttpSessionIdListener

If the ServletContext was passed to the ServletContainerInitializer’s onStartup method, then the class with the given name MAY also implement javax.servlet.ServletContextListener in addition to the interfaces listed above. As part of this method call, the container MUST load the class with the specified class name to ensure that it implements one of the required interfaces. If the class with the given name implements a listener interface whose invocation order corresponds to the declaration order, that is, if it implements javax.servlet.ServletRequestListener, javax.servlet.ServletContextListener or javax.servlet.http.HttpSessionListener, then the new listener will be added to the end of the ordered list of listeners of that interface.

4.4.3.2 <T extends EventListener> void addListener(T t)

Add the given listener to the ServletContext. The given listener MUST be an instance of one or more of the following interfaces:

- javax.servlet.ServletContextAttributeListener
- javax.servlet.ServletRequestListener
- javax.servlet.ServletRequestAttributeListener
- javax.servlet.http.HttpSessionListener
- javax.servlet.http.HttpSessionAttributeListener
- javax.servlet.http.HttpSessionIdListener
If the ServletContext was passed to the ServletContainerInitializer’s onStartup method, then the given listener MAY also be an instance of javax.servlet.ServletContextListener in addition to the interfaces listed above. If the given listener is an instance of a listener interface whose invocation order corresponds to the declaration order, that is, if it implements javax.servlet.ServletRequestListener, javax.servlet.ServletContextListener or javax.servlet.http.HttpSessionListener, then the new listener will be added to the end of the ordered list of listeners of that interface.

### 4.4.3.3 void addListener(Class <? extends EventListener> listenerClass)

Add the listener of the given class type to the ServletContext. The given listener class MUST implement one or more of the following interfaces:

- javax.servlet.ServletContextAttributeListener
- javax.servlet.ServletRequestListener
- javax.servlet.ServletRequestAttributeListener
- javax.servlet.http.HttpSessionListener
- javax.servlet.http.HttpSessionAttributeListener
- javax.servlet.http.HttpSessionIdListener

If the ServletContext was passed to the ServletContainerInitializer’s onStartup method, then the given listener class MAY also implement javax.servlet.ServletContextListener in addition to the interfaces listed above. If the given listener class implements a listener interface whose invocation order corresponds to the declaration order, that is, if it implements javax.servlet.ServletRequestListener, javax.servlet.ServletContextListener or javax.servlet.http.HttpSessionListener, then the new listener will be added to the end of the ordered list of listeners of that interface.

### 4.4.3.4 <T extends EventListener> void createListener(Class<T> clazz)

This method instantiates the given EventListener class. The specified EventListener class MUST implement at least one of the following interfaces:

- javax.servlet.ServletContextAttributeListener
- javax.servlet.ServletRequestListener
- javax.servlet.ServletRequestAttributeListener
- javax.servlet.http.HttpSessionListener
This method MUST support all annotations applicable to the above listener interfaces as defined by this specification. The returned EventListener instance may be further customized before it is registered with the ServletContext via a call to addListener(T t). The given EventListener class MUST define a zero argument constructor, which is used to instantiate it.

### 4.4.3.5 Annotation processing requirements for programmatically added Servlets, Filters and Listeners

When using the programmatic API to add a servlet or create a servlet, apart from the addServlet that takes an instance, the following annotations must be introspected in the class in question and the metadata defined in it MUST be used unless it is overridden by calls to the API in the ServletRegistration.Dynamic / ServletRegistration.

@ServletSecurity, @RunAs, @DeclareRoles, @MultipartConfig.

For Filters and Listeners no annotations need to be introspected.

Resource injection on all components (Servlets, Filters and Listeners) added programmatically or created programmatically, other than the ones added via the methods that takes an instance, will only be supported when the component is a CDI Managed Bean. For details please refer to Section 15.5.15, “Contexts and Dependency Injection for Java EE requirements” on page 15-197.

### 4.5 Context Attributes

A servlet can bind an object attribute into the context by name. Any attribute bound into a context is available to any other servlet that is part of the same Web application. The following methods of ServletContext interface allow access to this functionality:

- `setAttribute`
- `getAttribute`
- `getAttributeNames`
- `removeAttribute`
4.5.1 Context Attributes in a Distributed Container

Context attributes are local to the JVM in which they were created. This prevents ServletContext attributes from being a shared memory store in a distributed container. When information needs to be shared between servlets running in a distributed environment, the information should be placed into a session (See Chapter 7, “Sessions”), stored in a database, or set in an Enterprise JavaBeans™ component.

4.6 Resources

The ServletContext interface provides direct access only to the hierarchy of static content documents that are part of the Web application, including HTML, GIF, and JPEG files, via the following methods of the ServletContext interface:

- getResource
- getResourceAsStream

The getResource and getResourceAsStream methods take a String with a leading “/” as an argument that gives the path of the resource relative to the root of the context or relative to the META-INF/resources directory of a JAR file inside the web application’s WEB-INF/lib directory. These methods will first search the root of the web application context for the requested resource before looking at any of the JAR files in the WEB-INF/lib directory. The order in which the JAR files in the WEB-INF/lib directory are scanned is undefined. This hierarchy of documents may exist in the server’s file system, in a Web application archive file, on a remote server, or at some other location.

These methods are not used to obtain dynamic content. For example, in a container supporting the JavaServer Pages™ specification1, a method call of the form getResource("/index.jsp") would return the JSP source code and not the processed output. See Chapter 9, “Dispatching Requests” for more information about accessing dynamic content.

The full listing of the resources in the Web application can be accessed using the getResourcePaths(String path) method. The full details on the semantics of this method may be found in the API documentation in this specification.

---

1. The JavaServer Pages™ specification can be found at http://java.sun.com/products/jsp
4.7 Multiple Hosts and Servlet Contexts

Web servers may support multiple logical hosts sharing one IP address on a server. This capability is sometimes referred to as "virtual hosting". In this case, each logical host must have its own servlet context or set of servlet contexts. Servlet contexts cannot be shared across virtual hosts.

The `getVirtualServerName` method of `ServletContext` interface allows access to the configuration name of the logical host on which the `ServletContext` is deployed. Servlet containers may support multiple logical hosts. This method must return the same name for all the servlet contexts deployed on a logical host, and the name returned by this method must be distinct, stable per logical host, and suitable for use in associating server configuration information with the logical host.

4.8 Reloading Considerations

Although a Container Provider implementation of a class reloading scheme for ease of development is not required, any such implementation must ensure that all servlets, and classes that they may use, are loaded in the scope of a single class loader. This requirement is needed to guarantee that the application will behave as expected by the Developer. As a development aid, the full semantics of notification to session binding listeners should be supported by containers for use in the monitoring of session termination upon class reloading.

Previous generations of containers created new class loaders to load a servlet, distinct from class loaders used to load other servlets or classes used in the servlet context. This could cause object references within a servlet context to point at unexpected classes or objects, and cause unexpected behavior. The requirement is needed to prevent problems caused by demand generation of new class loaders.

4.8.1 Temporary Working Directories

A temporary storage directory is required for each servlet context. Servlet containers must provide a private temporary directory for each servlet context, and make it available via the `javax.servlet.context.tempdir` context attribute. The objects associated with the attribute must be of type `java.io.File`.

2. An exception is system classes that the servlet may use in a different class loader.
The requirement recognizes a common convenience provided in many servlet engine implementations. The container is not required to maintain the contents of the temporary directory when the servlet container restarts, but is required to ensure that the contents of the temporary directory of one servlet context is not visible to the servlet contexts of other Web applications running on the servlet container.
The Response

The response object encapsulates all information to be returned from the server to the client. In the HTTP protocol, this information is transmitted from the server to the client either by HTTP headers or the message body of the request.

5.1 Buffering

A servlet container is allowed, but not required, to buffer output going to the client for efficiency purposes. Typically servers that do buffering make it the default, but allow servlets to specify buffering parameters.

The following methods in the `ServletResponse` interface allow a servlet to access and set buffering information:

- `getBufferSize`
- `setBufferSize`
- `isCommitted`
- `reset`
- `resetBuffer`
- `flushBuffer`

These methods are provided on the `ServletResponse` interface to allow buffering operations to be performed whether the servlet is using a `ServletOutputStream` or a `Writer`.

The `getBufferSize` method returns the size of the underlying buffer being used. If no buffering is being used, this method must return the `int` value of 0 (zero).

The servlet can request a preferred buffer size by using the `setBufferSize` method. The buffer assigned is not required to be the size requested by the servlet, but must be at least as large as the size requested. This allows the container to reuse a set of fixed size buffers, providing a larger buffer than requested if appropriate. The
method must be called before any content is written using a ServletOutputStream or Writer. If any content has been written or the response object has been committed, this method must throw an IllegalStateException.

The isCommitted method returns a boolean value indicating whether any response bytes have been returned to the client. The flushBuffer method forces content in the buffer to be written to the client.

The reset method clears data in the buffer when the response is not committed. Headers, status codes and the state of calling getWriter or getOutputStream set by the servlet prior to the reset call must be cleared as well. The resetBuffer method clears content in the buffer if the response is not committed without clearing the headers and status code.

If the response is committed and the reset or resetBuffer method is called, an IllegalStateException must be thrown. The response and its associated buffer will be unchanged.

When using a buffer, the container must immediately flush the contents of a filled buffer to the client. If this is the first data that is sent to the client, the response is considered to be committed.

---

5.2 Headers

A servlet can set headers of an HTTP response via the following methods of the HttpServletResponse interface:

- setHeader
- addHeader

The setHeader method sets a header with a given name and value. A previous header is replaced by the new header. Where a set of header values exist for the name, the values are cleared and replaced with the new value.

The addHeader method adds a header value to the set with a given name. If there are no headers already associated with the name, a new set is created.

Headers may contain data that represents an int or a Date object. The following convenience methods of the HttpServletResponse interface allow a servlet to set a header using the correct formatting for the appropriate data type:

- setIntHeader
- setDateHeader
- addIntHeader
- addDateHeader
To be successfully transmitted back to the client, headers must be set before the response is committed. Headers set after the response is committed will be ignored by the servlet container.

Servlet programmers are responsible for ensuring that the Content-Type header is appropriately set in the response object for the content the servlet is generating. The HTTP 1.1 specification does not require that this header be set in an HTTP response. Servlet containers must not set a default content type when the servlet programmer does not set the type.

It is recommended that containers use the X-Powered-By HTTP header to publish its implementation information. The field value should consist of one or more implementation types, such as "Servlet/3.1". Optionally, the supplementary information of the container and the underlying Java platform can be added after the implementation type within parentheses. The container should be configurable to suppress this header.

Here's the examples of this header.

X-Powered-By: Servlet/3.1
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open Source Edition 4.0 Java/Oracle Corporation/1.7)

5.3 Non Blocking IO

Non-blocking IO only works with async request processing in Servlets and Filters (as defined in Section 2.3.3.3, “Asynchronous processing” on page 2-10), and upgrade processing (as defined in Section 2.3.3.5, “Upgrade Processing” on page 2-20). Otherwise, an IllegalStateException must be thrown when ServletInputStream.setReadListener or ServletOutputStream.setWriteListener is invoked. To support non-blocking writes in the Web container, in addition to the changes made in the ServletRequest as described in Section 3.7, “Non Blocking IO” on page 3-28, the following changes have been made to handle response related classes / interfaces.

The WriteListener provides the following callback methods which the container invokes appropriately.

- WriteListener
  - void onWritePossible(). When a WriteListener is registered with the ServletOutputStream, this method will be invoked by the container the first time when it is possible to write data. The container will subsequently invoke the onWritePossible method if and only if isReady method on ServletOutputStream, described below, returns false.
- onError(Throwable t). Invoked when an error occurs processing the response.

Along with the WriteListener, the following methods have been added to ServletOutputStream class to allow the developer to check with the runtime whether or not it is possible to write the data to be sent to the client.

- ServletOutputStream
  - boolean isReady(). This method returns true if a write to the ServletOutputStream will succeed, otherwise it will return false. If this method returns true, a write operation can be performed on the ServletOutputStream. If no further data can be written to the ServletOutputStream, then this method will return false till the underlying data is flushed at which point the container will invoke the onWritePossible method of the WriteListener. A subsequent call to this method will return true.
  
- void setWriteListener(WriteListener listener). Associates the WriteListener with this ServletOutputStream, for the container to invoke the callback methods on the WriteListener when it is possible to write data. Registering a WriteListener will start non-blocking IO. It is illegal to switch to the traditional blocking IO at that point.

The Servlet container must access methods in WriteListener in a thread safe manner.

### 5.4 Convenience Methods

The following convenience methods exist in the HttpServletResponse interface:

- sendRedirect
- sendError

The sendRedirect method will set the appropriate headers and content body to redirect the client to a different URL. It is legal to call this method with a relative URL path, however the underlying container must translate the relative path to a fully qualified URL for transmission back to the client. If a partial URL is given and, for whatever reason, cannot be converted into a valid URL, then this method must throw an IllegalArgumentException.

The sendError method will set the appropriate headers and content body for an error message to return to the client. An optional String argument can be provided to the sendError method which can be used in the content body of the error.
These methods will have the side effect of committing the response, if it has not already been committed, and terminating it. No further output to the client should be made by the servlet after these methods are called. If data is written to the response after these methods are called, the data is ignored.

If data has been written to the response buffer, but not returned to the client (i.e. the response is not committed), the data in the response buffer must be cleared and replaced with the data set by these methods. If the response is committed, these methods must throw an `IllegalStateException`.

### 5.5 Internationalization

Servlets should set the locale and the character encoding of a response. The locale is set using the `ServletResponse.setLocale` method. The method can be called repeatedly; but calls made after the response is committed have no effect. If the servlet does not set the locale before the page is committed, the container’s default locale is used to determine the response’s locale, but no specification is made for the communication with a client, such as `Content-Language` header in the case of HTTP.

```
<locale-encoding-mapping-list>
  <locale-encoding-mapping>
    <locale>ja</locale>
    <encoding>Shift_JIS</encoding>
  </locale-encoding-mapping>
</locale-encoding-mapping-list>
```

If the element does not exist or does not provide a mapping, `setLocale` uses a container dependent mapping. The `setCharacterEncoding`, `setContentType`, and `setLocale` methods can be called repeatedly to change the character encoding. Calls made after the servlet response’s `getWriter` method has been called or after the response is committed have no effect on the character encoding. Calls to `setContentType` set the character encoding only if the given content type string provides a value for the `charset` attribute. Calls to `setLocale` set the character encoding only if neither `setCharacterEncoding` nor `setContentType` has set the character encoding before.

If the servlet does not specify a character encoding before the `getWriter` method of the `ServletResponse` interface is called or the response is committed, the default ISO-8859-1 is used.

Containers must communicate the locale and the character encoding used for the servlet response’s writer to the client if the protocol in use provides a way for doing so. In the case of HTTP, the locale is communicated via the `Content-Language`
header, the character encoding as part of the Content-Type header for text media types. Note that the character encoding cannot be communicated via HTTP headers if the servlet does not specify a content type; however, it is still used to encode text written via the servlet response’s writer.

5.6 Closure of Response Object

When a response is closed, the container must immediately flush all remaining content in the response buffer to the client. The following events indicate that the servlet has satisfied the request and that the response object is to be closed:

- The termination of the service method of the servlet.
- The amount of content specified in the setContentLength or setContentLengthLong method of the response has been greater than zero and has been written to the response.
- The sendError method is called.
- The sendRedirect method is called.
- The complete method on AsyncContext is called.

5.7 Lifetime of the Response Object

Each response object is valid only within the scope of a servlet’s service method, or within the scope of a filter’s doFilter method, unless the associated request object has asynchronous processing enabled for the component. If asynchronous processing on the associated request is started, then the request object remains valid until complete method on AsyncContext is called. 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 for which startAsync on the corresponding request has not been called, outside the scope described above may lead to non-deterministic behavior.
Filtering

Filters are Java components that allow on the fly transformations of payload and header information in both the request into a resource and the response from a resource.

The Java Servlet API classes and methods that provide a lightweight framework for filtering active and static content. It describes how filters are configured in a Web application, and conventions and semantics for their implementation.

API documentation for servlet filters is provided online. The configuration syntax for filters is given by the deployment descriptor schema in Chapter 14, “Deployment Descriptor”. The reader should use these sources as references when reading this chapter.

6.1 What is a filter?

A filter is a reusable piece of code that can transform the content of HTTP requests, responses, and header information. Filters do not generally create a response or respond to a request as servlets do, rather they modify or adapt the requests for a resource, and modify or adapt responses from a resource.

Filters can act on dynamic or static content. For the purposes of this chapter, dynamic and static content are referred to as Web resources.

Among the types of functionality available to the developer needing to use filters are the following:
- The accessing of a resource before a request to it is invoked.
- The processing of the request for a resource before it is invoked.
- The modification of request headers and data by wrapping the request in customized versions of the request object.
■ The modification of response headers and response data by providing customized versions of the response object.
■ The interception of an invocation of a resource after its call.
■ Actions on a servlet, on groups of servlets, or static content by zero, one, or more filters in a specifiable order.

### 6.1.1 Examples of Filtering Components

- Authentication filters
- Logging and auditing filters
- Image conversion filters
- Data compression filters
- Encryption filters
- Tokenizing filters
- Filters that trigger resource access events
- XSL/T filters that transform XML content
- MIME-type chain filters
- Caching filters

### 6.2 Main Concepts

The main concepts of this filtering model are described in this section.

The application developer creates a filter by implementing the `javax.servlet.Filter` interface and providing a public constructor taking no arguments. The class is packaged in the Web Archive along with the static content and servlets that make up the Web application. A filter is declared using the `<filter>` element in the deployment descriptor. A filter or collection of filters can be configured for invocation by defining `<filter-mapping>` elements in the deployment descriptor. This is done by mapping filters to a particular servlet by the servlet’s logical name, or mapping to a group of servlets and static content resources by mapping a filter to a URL pattern.

### 6.2.1 Filter Lifecycle

After deployment of the Web application, and before a request causes the container to access a Web resource, the container must locate the list of filters that must be applied to the Web resource as described below. The container must ensure that it has instantiated a filter of the appropriate class for each filter in the list, and called its `init(FilterConfig config)` method. The filter may throw an exception to
indicate that it cannot function properly. If the exception is of type `UnavailableException`, the container may examine the `isPermanent` attribute of the exception and may choose to retry the filter at some later time.

Only one instance per `<filter>` declaration in the deployment descriptor is instantiated per JVM of the container. The container provides the filter config as declared in the filter’s deployment descriptor, the reference to the `ServletContext` for the Web application, and the set of initialization parameters.

When the container receives an incoming request, it takes the first filter instance in the list and calls its `doFilter` method, passing in the `ServletRequest` and `ServletResponse`, and a reference to the `FilterChain` object it will use.

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’s headers.

2. The method may wrap the request object with a customized implementation of `ServletRequest` or `HttpServletRequest` in order to modify request headers or data.

3. The method may wrap the response object passed in to its `doFilter` method with a customized implementation of `ServletRequest` or `HttpServletRequest` to modify response headers or data.

4. The filter may invoke the next entity in the filter chain. The next entity may be another filter, or if the filter making the invocation is the last filter configured in the deployment descriptor for this chain, the next entity is the target Web resource. The invocation of the next entity 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 filter chain’s implementation of the `doFilter` method, provided by the container, must locate the next entity 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 entity, leaving the filter responsible for filling out the response object.

The `service` method is required to run in the same thread as all filters that apply to the servlet.

5. After invocation of the next filter in the chain, the filter may examine response headers.

6. Alternatively, the filter may have thrown an exception to indicate an error in processing. If the filter throws an `UnavailableException` during its processing, the 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.
7. When the last filter in the chain has been invoked, the next entity accessed is the target servlet or resource at the end of the chain.

8. Before a filter instance can be removed from service by the container, the container must first call the destroy method on the filter to enable the filter to release any resources and perform other cleanup operations.

6.2.2 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 not only has the ability to override existing methods on the request and response objects, but to provide new API suited to a particular filtering task to a filter or target web resource down the chain. For example, the developer may wish to extend the response object with higher level output objects that the output stream or the writer, such as API that allows DOM objects to be written back to the client.

In order to support this style of filter the container must support the following requirement. When a filter invokes the doFilter method on the container’s filter chain implementation, the container must ensure that the request and response object that it passes to the next entity in the filter chain, or to the target web resource if the filter was the last in the chain, is the same object that was passed into the doFilter method by the calling filter.

The same requirement of wrapper object identity applies to the calls from a servlet or a filter to RequestDispatcher.forward or RequestDispatcher.include, when the caller wraps the request or response objects. In this case, the request and response objects seen by the called servlet must be the same wrapper objects that were passed in by the calling servlet or filter.

6.2.3 Filter Environment

A set of initialization parameters can be associated with a filter using the <init-params> element in the deployment descriptor. 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 ServletContext of the Web application for the loading of resources, for logging functionality, and for storage of state in the ServletContext’s attribute list. A Filter and the target servlet or resource at the end of the filter chain must execute in the same invocation thread.
6.2.4 Configuration of Filters in a Web Application

A filter is defined either via the @WebFilter annotation as defined in Section 8.1.2, “@WebFilter” on page 8-69 of the specification or 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 servlet or URL
- **filter-class**: used by the container to identify the filter type
- **init-params**: initialization parameters for a filter

Optionally, the programmer can specify icons, a textual description, and a display name for tool manipulation. The 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 container if the developer makes two filter declarations for the same filter class.

Here is an example of a filter declaration:

```xml
<filter>
  <filter-name>Image Filter</filter-name>
  <filter-class>com.acme.ImageServlet</filter-class>
</filter>
```

Once a filter has been declared in the deployment descriptor, the assembler uses the <filter-mapping> element to define servlets and static resources in the Web application to which the filter is to be applied. Filters can be associated with a servlet using the <servlet-name> element. For example, the following code example maps the Image Filter filter to the ImageServlet servlet:

```xml
<filter-mapping>
  <filter-name>Image Filter</filter-name>
  <servlet-name>ImageServlet</servlet-name>
</filter-mapping>
```

Filters can be associated with groups of servlets and static content using the <url-pattern> style of filter mapping:

```xml
<filter-mapping>
  <filter-name>Logging Filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Here the Logging Filter is applied to all the servlets and static content pages in the Web application, because every request URI matches the ‘/*’ URL pattern.
When processing a `<filter-mapping>` element using the `<url-pattern>` style, the container must determine whether the `<url-pattern>` matches the request URI using the path mapping rules defined in Chapter 12, “Mapping Requests to Servlets”.

The order the container uses in building the chain of filters to be applied for a particular request URI is as follows:

1. First, the `<url-pattern>` matching filter mappings in the same order that these elements appear in the deployment descriptor.

2. Next, the `<servlet-name>` matching filter mappings in the same order that these elements appear in the deployment descriptor.

If a filter mapping contains both `<servlet-name>` and `<url-pattern>`, the container must expand the filter mapping into multiple filter mappings (one for each `<servlet-name>` and `<url-pattern>`), preserving the order of the `<servlet-name>` and `<url-pattern>` elements. For example, the following filter mapping:

```
<filter-mapping>
  <filter-name>Multipe Mappings Filter</filter-name>
  <url-pattern>/foo/*</url-pattern>
  <servlet-name>Servlet1</servlet-name>
  <servlet-name>Servlet2</servlet-name>
  <url-pattern>/bar/*</url-pattern>
</filter-mapping>
```
is equivalent to:

```xml
<filter-mapping>
  <filter-name>Multipe Mappings Filter</filter-name>
  <url-pattern>/foo/*</url-pattern>
</filter-mapping>
<filter-mapping>
  <filter-name>Multipe Mappings Filter</filter-name>
  <servlet-name>Servlet1</servlet-name>
</filter-mapping>
<filter-mapping>
  <filter-name>Multipe Mappings Filter</filter-name>
  <servlet-name>Servlet2</servlet-name>
</filter-mapping>
<filter-mapping>
  <filter-name>Multipe Mappings Filter</filter-name>
  <url-pattern>/bar/*</url-pattern>
</filter-mapping>
```

The requirement about the order of the filter chain means that the container, when receiving an incoming request, processes the request as follows:

- Identifies the target Web resource according to the rules of “Specification of Mappings” on page 122.
- If there are filters matched by servlet name and the Web resource has a `<servlet-name>`, the container builds the chain of filters matching in the order declared in the deployment descriptor. The last filter in this chain corresponds to the last `<servlet-name>` matching filter and is the filter that invokes the target Web resource.
- If there are filters using `<url-pattern>` matching and the `<url-pattern>` matches the request URI according to the rules of Section 12.2, “Specification of Mappings”, the container builds the chain of `<url-pattern>` matched filters in the same order as declared in the deployment descriptor. The last filter in this chain is the last `<url-pattern>` matching filter in the deployment descriptor for this request URI. The last filter in this chain is the filter that invokes the first filter in the `<servlet-name>` matching chain, or invokes the target Web resource if there are none.

It is expected that high performance Web containers will cache filter chains so that they do not need to compute them on a per-request basis.
6.2.5 Filters and the RequestDispatcher

New since version 2.4 of the Java Servlet specification is the ability to configure filters to be invoked under request dispatcher \texttt{forward()} and \texttt{include()} calls.

By using the new \texttt{<dispatcher>} element in the deployment descriptor, the developer can indicate for a filter-mapping whether he would like the filter to be applied to requests when:

1. The request comes directly from the client.
   
   This is indicated by a \texttt{<dispatcher>} element with value \texttt{REQUEST}, or by the absence of any \texttt{<dispatcher>} elements.

2. The request is being processed under a request dispatcher representing the Web component matching the \texttt{<url-pattern>} or \texttt{<servlet-name>} using a \texttt{forward()} call.
   
   This is indicated by a \texttt{<dispatcher>} element with value \texttt{FORWARD}.

3. The request is being processed under a request dispatcher representing the Web component matching the \texttt{<url-pattern>} or \texttt{<servlet-name>} using an \texttt{include()} call.
   
   This is indicated by a \texttt{<dispatcher>} element with value \texttt{INCLUDE}.

4. The request is being processed with the error page mechanism specified in "Error Handling" on page 108 to an error resource matching the \texttt{<url-pattern>}. 
   
   This is indicated by a \texttt{<dispatcher>} element with the value \texttt{ERROR}.

5. The request is being processed with the async context dispatch mechanism specified in "Asynchronous processing" on page 10 to a web component using a \texttt{dispatch} call.
   
   This is indicated by a \texttt{<dispatcher>} element with the value \texttt{ASYNC}.

6. Or any combination of 1, 2, 3, 4 or 5 above.

For example:

\begin{verbatim}
<filter-mapping>
  <filter-name>Logging Filter</filter-name>
  <url-pattern>/products/*</url-pattern>
</filter-mapping>
\end{verbatim}
would result in the Logging Filter being invoked by client requests starting
/products/... but not underneath a request dispatcher call where the request
dispenser has path commencing /products/.... The LoggingFilter would be
invoked both on the initial dispatch of the request and on resumed request. The
following code:

```xml
<filter-mapping>
    <filter-name>Logging Filter</filter-name>
    <servlet-name>ProductServlet</servlet-name>
    <dispatcher>INCLUDE</dispatcher>
</filter-mapping>
```

would result in the Logging Filter not being invoked by client requests to the
ProductServlet, nor underneath a request dispatcher forward() call to the
ProductServlet, but would be invoked underneath a request dispatcher include()
call where the request dispatcher has a name commencing ProductServlet. The
following code:

```xml
<filter-mapping>
    <filter-name>Logging Filter</filter-name>
    <url-pattern>/products/*</url-pattern>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

would result in the Logging Filter being invoked by client requests starting
/products/... and underneath a request dispatcher forward() call where the
request dispatcher has path commencing /products/....

Finally, the following code uses the special servlet name ‘*’:

```xml
<filter-mapping>
    <filter-name>All Dispatch Filter</filter-name>
    <servlet-name>*</servlet-name>
    <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

This code would result in the All Dispatch Filter being invoked on request
dispenser forward() calls for all request dispatchers obtained by name or by path.
The Hypertext Transfer Protocol (HTTP) is by design a stateless protocol. To build effective Web applications, it is imperative that requests from a particular client be associated with each other. Many strategies for session tracking have evolved over time, but all are difficult or troublesome for the programmer to use directly.

This specification defines a simple HttpSession interface that allows a servlet container to use any of several approaches to track a user’s session without involving the Application Developer in the nuances of any one approach.

7.1 Session Tracking Mechanisms

The following sections describe approaches to tracking a user’s sessions.

7.1.1 Cookies

Session tracking through HTTP cookies is the most used session tracking mechanism and is required to be supported by all servlet containers.

The container sends a cookie to the client. The client will then return the cookie on each subsequent request to the server, unambiguously associating the request with a session. The standard name of the session tracking cookie must be JSESSIONID. Containers may allow the name of the session tracking cookie to be customized through container specific configuration.

All servlet containers MUST provide an ability to configure whether or not the container marks the session tracking cookie as HttpOnly. The established configuration must apply to all contexts for which a context specific configuration has not been established (see SessionCookieConfig javadoc for more details).
If a web application configures a custom name for its session tracking cookies, the same custom name will also be used as the name of the URI parameter if the session id is encoded in the URL (provided that URL rewriting has been enabled).

### 7.1.2 SSL Sessions

Secure Sockets Layer, the encryption technology used in the HTTPS protocol, has a built-in mechanism allowing multiple requests from a client to be unambiguously identified as being part of a session. A servlet container can easily use this data to define a session.

### 7.1.3 URL Rewriting

URL rewriting is the lowest common denominator of session tracking. When a client will not accept a cookie, URL rewriting may be used by the server as the basis for session tracking. URL rewriting involves adding data, a session ID, to the URL path that is interpreted by the container to associate the request with a session.

The session ID must be encoded as a path parameter in the URL string. The name of the parameter must be `jsessionid`. Here is an example of a URL containing encoded path information:

```
http://www.myserver.com/catalog/index.html;jsessionid=1234
```

URL rewriting exposes session identifiers in logs, bookmarks, referer headers, cached HTML, and the URL bar. URL rewriting should not be used as a session tracking mechanism where cookies or SSL sessions are supported and suitable.

### 7.1.4 Session Integrity

Web containers must be able to support the HTTP session while servicing HTTP requests from clients that do not support the use of cookies. To fulfill this requirement, Web containers commonly support the URL rewriting mechanism.

### 7.2 Creating a Session

A session is considered “new” when it is only a prospective session and has not been established. Because HTTP is a request-response based protocol, an HTTP 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” if either 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 servlet container has no mechanism by which to associate a request with a previous request.

A Servlet Developer must design his application to handle a situation where a client has not, can not, or will not join a session.

Associated with each session, there is a string containing a unique identifier, which is referred to as the session id. The value of the session id can be obtained by calling `javax.servlet.http.HttpSession.getId()` and can be changed after creation by invoking `javax.servlet.http.HttpServletRequest.changesessionId()`.

### 7.3 Session Scope

`HttpSession` objects must be scoped at the application (or servlet context) level. The underlying mechanism, such as the cookie used to establish the session, can be the same for different contexts, but the object referenced, including the attributes in that object, must never be shared between contexts by the container.

To illustrate this requirement with an example: if a servlet uses the `RequestDispatcher` to call a servlet in another Web application, any sessions created for and visible to the servlet being called must be different from those visible to the calling servlet.

Additionally, sessions of a context must be resumable by requests into that context regardless of whether their associated context was being accessed directly or as the target of a request dispatch at the time the sessions were created.
7.4 Binding Attributes into a Session

A servlet can bind an object attribute into an HttpSession implementation by name. Any object bound into a session is available to any other servlet that belongs to the same ServletContext and handles a request identified as being a part of the same session.

Some objects may require notification when they are placed into, or removed from, a session. This information can be obtained by having the object implement the HttpSessionBindingListener interface. This interface defines the following methods that will signal an object being bound into, or being unbound from, a session.

- valueBound
- valueUnbound

The valueBound method must be called before the object is made available via the getAttribute method of the HttpSession interface. The valueUnbound method must be called after the object is no longer available via the getAttribute method of the HttpSession interface.

7.5 Session Timeouts

In the HTTP protocol, there is no explicit termination signal when a client is no longer active. This means that the only mechanism that can be used to indicate when a client is no longer active is a time out period.

The default time out period for sessions is defined by the servlet container and can be obtained via the getMaxInactiveInterval method of the HttpSession interface. This time out can be changed by the Developer using the setMaxInactiveInterval method of the HttpSession interface. The time out periods used by these methods are defined in seconds. By definition, if the time out period for a session is set to 0 or lesser value, the session will never expire. The session invalidation will not take effect until all servlets using that session have exited the service method. Once the session invalidation is initiated, a new request must not be able to see that session.
7.6 Last Accessed Times

The `getLastAccessedTime` method of the `HttpSession` interface allows a servlet to determine the last time the session was accessed before the current request. The session is considered to be accessed when a request that is part of the session is first handled by the servlet container.

7.7 Important Session Semantics

7.7.1 Threading Issues

Multiple servlets executing request threads may have active access to the same session object at the same time. The container must ensure that manipulation of internal data structures representing the session attributes is performed in a thread safe manner. The Developer has the responsibility for thread safe access to the attribute objects themselves. This will protect the attribute collection inside the `HttpSession` object from concurrent access, eliminating the opportunity for an application to cause that collection to become corrupted.

7.7.2 Distributed Environments

Within an application marked as distributable, all requests that are part of a session must be handled by one JVM at a time. The container must be able to handle all objects placed into instances of the `HttpSession` class using the `setAttribute` or `putValue` methods appropriately. The following restrictions are imposed to meet these conditions:

- The container must accept objects that implement the `Serializable` interface.
- The container may choose to support storage of other designated objects in the `HttpSession`, such as references to Enterprise JavaBeans components and transactions.
- Migration of sessions will be handled by container-specific facilities.

The distributed servlet container must throw an `IllegalArgumentException` for objects where the container cannot support the mechanism necessary for migration of the session storing them.
The distributed servlet container must support the mechanism necessary for migrating objects that implement `Serializable`.

These restrictions mean that the Developer is ensured that there are no additional concurrency issues beyond those encountered in a non-distributed container.

The Container Provider can ensure scalability and quality of service features like load-balancing and failover by having the ability to move a session object, and its contents, from any active node of the distributed system to a different node of the system.

If distributed containers persist or migrate sessions to provide quality of service features, they are not restricted to using the native JVM Serialization mechanism for serializing `HttpSessions` and their attributes. Developers are not guaranteed that containers will call `readObject` and `writeObject` methods on session attributes if they implement them, but are guaranteed that the `Serializable` closure of their attributes will be preserved.

Containers must notify any session attributes implementing the `HttpSessionActivationListener` during migration of a session. They must notify listeners of passivation prior to serialization of a session, and of activation after deserialization of a session.

Application Developers writing distributed applications should be aware that since the container may run in more than one Java virtual machine, the developer cannot depend on static variables for storing an application state. They should store such states using an enterprise bean or a database.

### 7.7.3 Client Semantics

Due to the fact that cookies or SSL certificates are typically controlled by the Web browser process and are not associated with any particular window of the browser, requests from all windows of a client application to a servlet container might be part of the same session. For maximum portability, the Developer should always assume that all windows of a client are participating in the same session.
Annotations and pluggability

This chapter describes the use of annotations and other enhancements to enable pluggability of frameworks and libraries for use within a web application.

8.1 Annotations and pluggability

In a web application, classes using annotations will have their annotations processed only if they are located in the WEB-INF/classes directory, or if they are packaged in a jar file located in WEB-INF/lib within the application.

The web application deployment descriptor contains a new "metadata-complete" attribute on the web-app element. The "metadata-complete" attribute defines whether the web descriptor is complete, or whether the class files of the jar file should be examined for annotations and web fragments at deployment time. If "metadata-complete" is set to "true", the deployment tool MUST ignore any servlet annotations that specify deployment information present in the class files of the application and web fragments. If the metadata-complete attribute is not specified or is set to "false", the deployment tool must examine the class files of the application for annotations, and scan for web fragments.

Following are the annotations that MUST be supported by a Servlet 3.0 compliant web container.

8.1.1 @WebServlet

This annotation is used to define a Servlet component in a web application. This annotation is specified on a class and contains metadata about the Servlet being declared. The urlPatterns or the value attribute on the annotation MUST be present. All other attributes are optional with default settings (see javadocs for more.
details). It is recommended to use value when the only attribute on the annotation is the url pattern and to use the urlPatterns attribute when the other attributes are also used. It is illegal to have both value and urlPatterns attribute used together on the same annotation. The default name of the Servlet if not specified is the fully qualified class name. The annotated servlet MUST specify at least one url pattern to be deployed. If the same servlet class is declared in the deployment descriptor under a different name, a new instance of the servlet MUST be instantiated. If the same servlet class is added with a different name to the ServletContext via the programmatic API defined in Section 4.4.1, “Programmatically adding and configuring Servlets” on page 4-35, the attribute values declared via the @WebServlet annotation MUST be ignored and a new instance of the servlet with the name specified MUST be created.

Classes annotated with @WebServlet class MUST extend the javax.servlet.http.HttpServlet class.

Following is an example of how this annotation would be used.

**CODE EXAMPLE 8-1  @WebServlet Annotation Example**

```java
@WebServlet("/foo")
public class CalculatorServlet extends HttpServlet{
    //...
}
```

Following is an example of how this annotation would be used with some more of the attributes specified.

**CODE EXAMPLE 8-2  @WebServlet annotation example using other annotation attributes specified**

```java
@WebServlet(name="MyServlet", urlPatterns={"/foo", "/bar"})
public class SampleUsingAnnotationAttributes extends HttpServlet{
    public void doGet(HttpServletRequest req, HttpServletResponse res) {

    }
}
```
8.1.2 @WebFilter

This annotation is used to define a Filter in a web application. This annotation is specified on a class and contains metadata about the filter being declared. The default name of the Filter if not specified is the fully qualified class name. The urlPatterns attribute, servletNames attribute or the value attribute of the annotation MUST be specified. All other attributes are optional with default settings (see javadocs for more details). It is recommended to use value when the only attribute on the annotation is the url pattern and to use the urlPatterns attribute when the other attributes are also used. It is illegal to have both value and urlPatterns attribute used together on the same annotation.

Classes annotated with @WebFilter MUST implement javax.servlet.Filter.

Following is an example of how this annotation would be used.

```java
@WebFilter("/foo")
public class MyFilter implements Filter {
    public void doFilter(HttpServletRequest req, HttpServletResponse res) {
        ...
    }
}
```

8.1.3 @WebInitParam

This annotation is used to specify any init parameters that must be passed to the Servlet or the Filter. It is an attribute of the WebServlet and WebFilter annotation.

8.1.4 @WebListener

The WebListener annotation is used to annotate a listener to get events for various operations on the particular web application context. Classes annotated with @WebListener MUST implement one of the following interfaces:

- javax.servlet.ServletContextListener
- javax.servlet.ServletContextAttributeListener
- javax.servlet.ServletRequestListener
- javax.servlet.ServletRequestAttributeListener
■ javax.servlet.http.HttpSessionListener
■ javax.servlet.http.HttpSessionAttributeListener
■ javax.servlet.http.HttpSessionIdListener

An example:
@WebListener
public class MyListener implements ServletContextListener{
    public void contextInitialized(ServletContextEvent sce) {
        ServletContext sc = sce.getServletContext();
        sc.addServlet("myServlet", "Sample servlet",
                     "foo.bar.MyServlet", null, -1);
        sc.addServletMapping("myServlet", new String[] {
                     "/urlpattern/**");
    }
}

8.1.5 @MultipartConfig

This annotation, when specified on a Servlet, indicates that the request it expects
is of type mime/multipart. The HttpServletRequest object of the
corresponding servlet MUST make available the mime attachments via the
getParts and getPart methods to iterate over the various mime attachments. The
location attribute of the javax.servlet.annotation.MultipartConfig and
the <location> element of the <multipart-config> is interpreted as an absolute
path and defaults to the value of the javax.servlet.context.tempdir. If a
relative path is specified, it will be relative to the tempdir location. The test for
absolute path vs relative path MUST be done via java.io.File.isAbsolute.

8.1.6 Other annotations / conventions

In addition to these annotations all the annotations defined in Section 15.5,
"Annotations and Resource Injection" on page 15-189 will continue to work in the
context of these new annotations.

By default all applications will have index.htm(l) and index.jsp in the list of
welcome-file-list. The descriptor may be used to override these default
settings.
The order in which the Listeners, Servlets are loaded from the various framework jars / classes in the WEB-INF/classes or WEB-INF/lib is unspecified when using annotations. If ordering is important then look at the section for modularity of web.xml and ordering of web.xml and web-fragment.xml below. The order can be specified in the deployment descriptor only.

8.2 Pluggability

8.2.1 Modularity of web.xml

Using the annotations defined above makes the use of web.xml optional. However for overriding either the default values or the values set via annotations, the deployment descriptor is used. As before, if the metadata-complete element is set to true in the web.xml descriptor, annotations that specify deployment information present in the class files and web-fragments bundled in jars will not be processed. It implies that all the metadata for the application is specified via the web.xml descriptor.

For better pluggability and less configuration for developers, we introduce the notion of web module deployment descriptor fragments (web fragment). A web fragment is a part or all of the web.xml that can be specified and included in a library or framework jar's META-INF directory. A plain old jar file in the WEB-INF/lib directory with no web-fragment.xml is also considered a fragment. Any annotations specified in it will be processed according to the rules defined in 8.2.3. The container will pick up and use the configuration as per the rules defined below.

A web fragment is a logical partitioning of the web application in such a way that the frameworks being used within the web application can define all the artifacts without asking developers to edit or add information in the web.xml. It can include almost all the same elements that the web.xml descriptor uses. However the top level element for the descriptor MUST be web-fragment and the corresponding descriptor file MUST be called web-fragment.xml. The ordering related elements also differ between the web-fragment.xml and web.xml See the corresponding schema for web-fragments in the deployment descriptor section in Chapter 14.

If a framework is packaged as a jar file and has metadata information in the form of deployment descriptor then the web-fragment.xml descriptor must be in the META-INF/ directory of the jar file.

If a framework wants its META-INF/web-fragment.xml honored in such a way that it augments a web application's web.xml, the framework must be bundled within the web application's WEB-INF/lib directory. In order for any other types of
resources (e.g., class files) of the framework to be made available to a web application, it is sufficient for the framework to be present anywhere in the classloader delegation chain of the web application. In other words, only JAR files bundled in a web application's WEB-INF/lib directory, but not those higher up in the class loading delegation chain, need to be scanned for web-fragment.xml.

During deployment the container is responsible for scanning the location specified above and discovering the web-fragment.xml and processing them. The requirements about name uniqueness that exist currently for a single web.xml also apply to the union of a web.xml and all applicable web-fragment.xml files.

An example of what a library or framework can include is shown below:

```xml
<web-fragment>
  <servlet>
    <servlet-name>welcome</servlet-name>
    <servlet-class>
      WelcomeServlet
    </servlet-class>
  </servlet>
  <listener>
    <listener-class>
      RequestListener
    </listener-class>
  </listener>
</web-fragment>
```

The above web-fragment.xml would be included in the META-INF/ directory of the framework’s jar file. The order in which configuration from web-fragment.xml and annotations should be applied is undefined. If ordering is an important aspect for a particular application please see rules defined below on how to achieve the order desired.

### 8.2.2 Ordering of web.xml and web-fragment.xml

Since the specification allows the application configuration resources to be composed of multiple configuration files (web.xml and web-fragment.xml), discovered and loaded from several different places in the application, the question of ordering must be addressed. This section specifies how configuration resource authors may declare the ordering requirements of their artifacts.
A `web-fragment.xml` may have a top level `<name>` element of type `javaee:java-identifierType`. There can only be one `<name>` element in a `web-fragment.xml`. If a `<name>` element is present, it must be considered for the ordering of artifacts (unless the duplicate name exception applies, as described below).

Two cases must be considered to allow application configuration resources to express their ordering preferences.

1. Absolute ordering: an `<absolute-ordering>` element in the `web.xml`. There can only be one `<absolute-ordering>` element in a `web.xml`.
   a. In this case, ordering preferences that would have been handled by case 2 below must be ignored.
   b. The `web.xml` and WEB-INF/classes MUST be processed before any of the web-fragments listed in the `absolute-ordering` element.
   c. Any `<name>` element direct children of the `<absolute-ordering>` MUST be interpreted as indicating the absolute ordering in which those named web-fragments, which may or may not be present, must be processed.
   d. The `<absolute-ordering>` element may contain zero or one `<others/>` element. The required action for this element is described below. If the `<absolute-ordering>` element does not contain an `<others/>` element, any web-fragment not specifically mentioned within `<name/>` elements MUST be ignored. Excluded jars are not scanned for annotated servlets, filters or listeners. However, if a servlet, filter or listener from an excluded jar is listed in `web.xml` or a non-excluded `web-fragment.xml`, then its annotations will apply unless otherwise excluded by `metadata-complete`. ServletContextListeners discovered in TLD files of excluded jars are not able to configure filters and servlets using the programmatic APIs. Any attempt to do so will result in an `IllegalStateException`. If a discovered ServletContainerInitializer is loaded from an excluded jar, it will be ignored. Irrespective of the setting of `metadata-complete`, jars excluded by `<absolute-ordering>` elements are not scanned for classes to be handled by any ServletContainerInitializer.
   e. Duplicate name exception: if, when traversing the children of `<absolute-ordering>`, multiple children with the same `<name>` element are encountered, only the first such occurrence must be considered.

2. Relative ordering: an `<ordering>` element within the `web-fragment.xml`. There can only be one `<ordering>` element in a `web-fragment.xml`.
   a. A `web-fragment.xml` may have an `<ordering>` element. If so, this element must contain zero or one `<before>` element and zero or one `<after>` element. The meaning of these elements is explained below.
b. The web.xml and WEB-INF/classes MUST be processed before any of the web-fragments listed in the ordering element.

c. Duplicate name exception: if, when traversing the web-fragments, multiple members with the same <name> element are encountered, the application must log an informative error message including information to help fix the problem, and must fail to deploy. For example, one way to fix this problem is for the user to use absolute ordering, in which case relative ordering is ignored.

d. Consider this abbreviated but illustrative example. 3 web-fragments - MyFragment1, MyFragment2 and MyFragment3 are part of the application that also includes a web.xml.

web-fragment.xml
<web-fragment>
  <name>MyFragment1</name>
  <ordering><after><name>MyFragment2</name></after></ordering>
  ...
</web-fragment>

web-fragment.xml
<web-fragment>
  <name>MyFragment2</name>
  ...
</web-fragment>

web-fragment.xml
<web-fragment>
  <name>MyFragment3</name>
  <ordering><before><others/></before></ordering>
  ...
</web-fragment>

web.xml
<web-app>
  ...
</web-app>

In this example the processing order will be
The preceding example illustrates some, but not all, of the following principles.

- `<before>` means the document must be ordered before the document with the name matching what is specified within the nested `<name>` element.

- `<after>` means the document must be ordered after the document with the name matching what is specified within the nested `<name>` element.

- There is a special element `<others/>` which may be included zero or one time within the `<before>` or `<after>` element, or zero or one time directly within the `<absolute-ordering>` element. The `<others/>` element must be handled as follows.
  - If the `<before>` element contains a nested `<others/>`, the document will be moved to the beginning of the list of sorted documents. If there are multiple documents stating `<before><others/>`, they will all be at the beginning of the list of sorted documents, but the ordering within the group of such documents is unspecified.
  - If the `<after>` element contains a nested `<others/>`, the document will be moved to the end of the list of sorted documents. If there are multiple documents requiring `<after><others/>`, they will all be at the end of the list of sorted documents, but the ordering within the group of such documents is unspecified.
  - Within a `<before>` or `<after>` element, if an `<others/>` element is present, but is not the only `<name>` element within its parent element, the other elements within that parent must be considered in the ordering process.
  - If the `<others/>` element appears directly within the `<absolute-ordering>` element, the runtime must ensure that any web-fragments not explicitly named in the `<absolute-ordering>` section are included at that point in the processing order.
  - If a `web-fragment.xml` file does not have an `<ordering>` or the `web.xml` does not have an `<absolute-ordering>` element the artifacts are assumed to not have any ordering dependency.
  - If the runtime discovers circular references, an informative message must be logged, and the application must fail to deploy. Again, one course of action the user may take is to use absolute ordering in the `web.xml`.
  - The previous example can be extended to illustrate the case when the `web.xml` contains an ordering section.

```xml
<web-app>
  <absolute-ordering>
```

### Chapter 8  Annotations and pluggability  8-75
In this example, the ordering for the various elements will be

web.xml
MyFragment3
MyFragment2

Some additional example scenarios are included below. All of these apply to relative ordering and not absolute ordering

Document A:
<after>
  <others/>
  <name>
    C
  </name>
</after>

Document B
<before>
  <others/>
</before>

Document C:
<after>
  <others/>
</after>

Document D: no ordering
Document E: no ordering
Document F:
<before>
  <others/>
Resulting parse order:
web.xml, F, B, D, E, C, A.

Document <no id>:
<after>
  <others/>
</after>
<before>
  <name>
    C
  </name>
</before>

Document B:
<before>
  <others/>
</before>

Document C: no ordering

Document D:
<after>
  <others/>
</after>

Document E:
<before>
  <others/>
</before>

Document F: no ordering

Resulting parse order can be one of the following:
- B, E, F, <no id>, C, D
- B, E, F, <no id>, D, C
8.2.3 Assembling the descriptor from web.xml, web-fragment.xml and annotations

If the order in which the listeners, servlets, filters are invoked is important to an application then a deployment descriptor must be used. Also, if necessary, the ordering element defined above can be used. As described above, when using annotations to define the listeners, servlets and filters, the order in which they are invoked is unspecified. Below are a set of rules that apply for assembling the final deployment descriptor for the application:

1. The order for listeners, servlets, filters if relevant must be specified in either the web-fragment.xml or the web.xml.

2. The ordering will be based on the order in which they are defined in the descriptor and on the absolute-ordering element in the web.xml or an ordering element in the web-fragment.xml, if present.

   a. Filters that match a request are chained in the order in which they are declared in the web.xml.
b. Servlets are initialized either lazily at request processing time or eagerly during deployment. In the latter case, they are initialized in the order indicated by their `load-on-startup` elements.

c. The listeners are invoked in the order in which they are declared in the `web.xml` as specified below:

   i. Implementations of `javax.servlet.ServletContextListener` are invoked at their `contextInitialized` method in the order in which they have been declared, and at their `contextDestroyed` method in reverse order.

   ii. Implementations of `javax.servlet.ServletRequestListener` are invoked at their `requestInitialized` method in the order in which they have been declared, and at their `requestDestroyed` method in reverse order.

   iii. Implementations of `javax.servlet.http.HttpSessionListener` are invoked at their `sessionCreated` method in the order in which they have been declared, and at their `sessionDestroyed` method in reverse order.

   iv. The methods of implementation of `javax.servlet.ServletContextAttributeListener`, `javax.servlet.ServletRequestAttributeListener` and `javax.servlet.HttpSessionAttributeListener` are invoked in the order in which they are declared when corresponding events are fired.

3. If a servlet is disabled using the `enabled` element introduced in the `web.xml` then the servlet will not be available at the `url-pattern` specified for the servlet.

4. The `web.xml` of the web application has the highest precedence when resolving conflicts between the `web.xml`, `web-fragment.xml` and annotations.

5. If `metadata-complete` is not specified in the descriptors, or is set to `false` in the deployment descriptor, then the effective metadata for the application is derived by combining the metadata present in the annotations and the descriptors. The rules for merging are specified below -

   a. Configuration settings in web fragments are used to augment those specified in the main `web.xml` in such a way as if they had been specified in the same `web.xml`.

   b. The order in which configuration settings of web fragments are added to those in the main `web.xml` is as specified above in Section 8.2.2, “Ordering of `web.xml` and `web-fragment.xml`” on page 8-72.

   c. The `metadata-complete` attribute when set to `true` in the main `web.xml`, is considered complete and scanning of annotations and fragments will not occur at deployment time. The `absolute-ordering` and `ordering` elements will
be ignored if present. When set to true on a fragment, the metadata-complete attribute applies only to scanning of annotations in that particular jar.

d. Web fragments are merged into the main web.xml unless the metadata-complete is set to true. The merging takes place after annotation processing on the corresponding fragment.

e. The following are considered configuration conflicts when augmenting a web.xml with web fragments:

i. Multiple <init-param> elements with the same <param-name> but different <param-value>

ii. Multiple <mime-mapping> elements with the same <extension> but different <mime-type>

f. The above configuration conflicts are resolved as follows:

i. Configuration conflicts between the main web.xml and a web fragment are resolved such that the configuration in the web.xml takes precedence.

ii. Configuration conflicts between two web fragments, where the element at the center of the conflict is not present in the main web.xml, will result in an error. An informative message must be logged, and the application must fail to deploy.

g. After the above conflicts have been resolved, these additional rules are applied

i. Elements that may be declared any number of times are additive across the web-fragments in the resulting web.xml. For example, <context-param> elements with different <param-name> are additive.

ii. Elements that may be declared any number of times, if specified in the web.xml overrides the values specified in the web-fragments with the same name.

iii. If an element with a minimum occurrence of zero, and a maximum occurrence of one, is present in a web fragment, and missing in the main web.xml, the main web.xml inherits the setting from the web fragment. If the element is present in both the main web.xml and the web fragment, the configuration setting in the main web.xml takes precedence. For example, if both the main web.xml and a web fragment declare the same servlet, and the servlet declaration in the web fragment specifies a <load-on-startup> element, whereas the one in the main web.xml does not, then the <load-on-startup> element from the web fragment will be used in the merged web.xml.
iv. It is considered an error if an element with a minimum occurrence of zero, and a maximum occurrence of one, is specified differently in two web fragments, while absent from the main web.xml. For example, if two web fragments declare the same servlet, but with different <load-on-startup> elements, and the same servlet is also declared in the main web.xml, but without any <load-on-startup>, then an error must be reported.

v. <welcome-file> declarations are additive.

vi. <servlet-mapping> elements with the same <servlet-name> are additive across web-fragments. <servlet-mapping> specified in the web.xml overrides values specified in the web-fragments with the same <servlet-name>.

vii. <filter-mapping> elements with the same <filter-name> are additive across web-fragments. <filter-mapping> specified in the web.xml overrides values specified in the web-fragments with the same <filter-name>.

viii. Multiple <listener> elements with the same <listener-class> are treated as a single <listener> declaration.

ix. The web.xml resulting from the merge is considered <distributable> only if all its web fragments are marked as <distributable> as well.

x. The top-level <icon> and its children elements, <display-name>, and <description> elements of a web fragment are ignored.

xi. jsp-property-group is additive. It is recommended that jsp-config element use the url-pattern as opposed to extension mappings when bundling static resources in the META-INF/resources directory of a jar file. Further more JSP resources for a fragment should be in a sub-directory same as the fragment name, if there exists one. This helps prevent a web-fragment’s jsp-property-group from affecting the JSPs in the main docroot of the application and the jsp-property-group from affecting the JSPs in a fragment’s META-INF/resources directory.

h. For all the resource reference elements (env-entry, ejb-ref, ejb-local-ref, service-ref, resource-ref, resource-env-ref, message-destination-ref, persistence-context-ref, persistence-unit-ref) the following rules apply:

i. If any resource reference element is present in a web fragment, and is missing in the main web.xml, the main web.xml inherits the value from the web fragment. If the element is present in both the main web.xml and the web fragment, with the same name, the web.xml takes precedence. None of the child elements from the fragment are merged into the main web.xml except for the injection-target as specified below. For example, if both the main web.xml and a web fragment declare a <resource-ref> with
the same `<resource-ref-name>`, the `<resource-ref>` from the
`web.xml` will be used without any child elements being merged from the
fragment except `<injection-target>` as described below.

ii. If a resource reference element is specified in two fragments, while absent
from the main `web.xml`, and all the attributes and child elements of the
resource reference element are identical, the resource reference will be
merged into the main `web.xml`. It is considered an error if a resource
reference element has the same name specified in two fragments, while
absent from the main `web.xml` and the attributes and child elements are not
identical in the two fragments. An error must be reported and the
application MUST fail to deploy. For example, if two web fragments declare
a `<resource-ref>` with the same `<resource-ref-name>` element but
the type in one is specified as `javax.sql.DataSource` while the type in
the other is that of a JavaMail resource, it is an error and the application will
fail to deploy.

iii. For resource reference element with the same name `<injection-target>`
elements from the fragments will be merged into the main `web.xml`.

i. In addition to the merging rules for `web-fragment.xml` defined above, the
following rules apply when using the resource reference annotations
(`@Resource`, `@Resources`, `@EJB`, `@EJBS`, `@WebServiceRef`,
`@WebServiceRefs`, `@PersistenceContext`,
`@PersistenceContexts`, `@PersistenceUnit`, and
`@PersistenceUnits`)

If a resource reference annotation is applied on a class, it is equivalent to
defining a resource, however it is not equivalent to defining an
injection-target. The rules above apply for injection-target element in this case.

If a resource reference annotation is used on a field it is equivalent to defining
the injection-target element in the `web.xml`. However if there is no
injection-target element in the descriptor then the injection-target
from the fragments will still be merged into the `web.xml` as defined above.

If on the other hand there is an injection-target in the main `web.xml` and
there is a resource reference annotation with the same resource name, then it
is considered an override for the resource reference annotation. In this case
since there is an injection-target specified in the descriptor, the rules
defined above would apply in addition to overriding the value for the resource
reference annotation.

j. If a data-source element is specified in two fragments, while absent from the
main `web.xml`, and all the attributes and child elements of the data-source
element are identical, the data-source will be merged into the main
`web.xml`. It is considered an error if a data-source element has the same
name specified in two fragments, while absent from the main `web.xml` and
the attributes and child elements are not identical in the two fragments. In such a
case an error must be reported and the application MUST fail to deploy.
Below are some examples that show the outcome in the different cases.

**CODE EXAMPLE 8-4**

web.xml - no resource-ref definition

Fragment 1

web-fragment.xml

```xml
<resource-ref>
    <resource-ref-name="foo">
        ...
        <injection-target>
            <injection-target-class>
                com.foo.Bar.class
            </injection-target-class>
            <injection-target-name>
                baz
            </injection-target-name>
        </injection-target>
    </resource-ref>
</resource-ref>
```

The effective metadata would be

```xml
<resource-ref>
    <resource-ref-name="foo">
        ....
        <injection-target>
            <injection-target-class>
                com.foo.Bar.class
            </injection-target-class>
            <injection-target-name>
                baz
            </injection-target-name>
        </injection-target>
    </resource-ref>
</resource-ref>
```

**CODE EXAMPLE 8-5**

web.xml
<resource-ref>
  <resource-ref-name="foo">
    ...
  </resource-ref>

Fragment 1
web-fragment.xml
<resource-ref>
  <resource-ref-name="foo">
    ...
    <injection-target>
      <injection-target-class>
        com.foo.Bar.class
      </injection-target-class>
      <injection-target-name>
        baz
      </injection-target-name>
    </injection-target>
  </resource-ref>
</resource-ref>

Fragment 2
web-fragment.xml
<resource-ref>
  <resource-ref-name="foo">
    ...
    <injection-target>
      <injection-target-class>
        com.foo.Bar2.class
      </injection-target-class>
      <injection-target-name>
        baz2
      </injection-target-name>
    </injection-target>
  </resource-ref>
</resource-ref>

The effective metadata would be
<resource-ref>
<resource-ref-name="foo">
    ....
</resource-ref-name>

<injection-target>
    <injection-target-class>
        com.foo.Bar.class
    </injection-target-class>
    <injection-target-name>
        baz
    </injection-target-name>
</injection-target>

<injection-target>
    <injection-target-class>
        com.foo.Bar2.class
    </injection-target-class>
    <injection-target-name>
        baz2
    </injection-target-name>
</injection-target>

</resource-ref>

CODE EXAMPLE 8-6

web.xml

<resource-ref>
    <resource-ref-name="foo">
        <injection-target>
            <injection-target-class>
                com.foo.Bar3.class
            </injection-target-class>
            <injection-target-name>
                baz3
            </injection-target-name>
        </injection-target>
    </resource-ref-name="foo">

    ...
</resource-ref>
Fragment 1
web-fragment.xml<resource-ref>
  <resource-ref-name="foo">
    ...
    <injection-target>
      <injection-target-class>
        com.foo.Bar.class
      </injection-target-class>
      <injection-target-name>
        baz
      </injection-target-name>
    </injection-target>
  </resource-ref>
</resource-ref>

Fragment 2
web-fragment.xml<resource-ref>
  <resource-ref-name="foo">
    ...
    <injection-target>
      <injection-target-class>
        com.foo.Bar2.class
      </injection-target-class>
      <injection-target-name>
        baz2
      </injection-target-name>
    </injection-target>
  </resource-ref>
</resource-ref>

The effective metadata would be

<resource-ref>
  <resource-ref-name="foo">
    <injection-target>
      <injection-target-class>
        com.foo.Bar3.class
      </injection-target-class>
    </injection-target>
  </resource-ref>
</resource-ref>
The `<injection-target>` from fragment 1 and 2 will be merged into the main web.xml

k. If the main `web.xml` does not have any `<post-construct>` element specified and web-fragments have specified `<post-construct>` then the `<post-construct>` elements from the fragments will be merged into the main web.xml. However if in the main `web.xml` at least one `<post-construct>` element is specified then the `<post-construct>` elements from the fragment will not be merged. It is the responsibility of the author of the `web.xml` to make sure that the `<post-construct>` list is complete.

l. If the main `web.xml` does not have any `<pre-destroy>` element specified and web-fragments have specified `<pre-destroy>` then the `<pre-destroy>` elements from the fragments will be merged into the main `web.xml`. However if in the main `web.xml` at least one `<pre-destroy>` element is specified then the `<pre-destroy>` elements from the fragment will not be merged. It is the responsibility of the author of the `web.xml` to make sure that the `<pre-destroy>` list is complete.
m. After processing the `web-fragment.xml`, annotations from the corresponding fragment are processed to complete the effective metadata for the fragment before processing the next fragment. The following rules are used for processing annotations:

n. Any metadata specified via an annotation that isn’t already present in the descriptor will be used to augment the effective descriptor.

i. Configuration specified in the main `web.xml` or a web fragment takes precedence over the configuration specified via annotations.

ii. For a servlet defined via the `@WebServlet` annotation, to override values via the descriptor, the name of the servlet in the descriptor MUST match the name of the servlet specified via the annotation (explicitly specified or the default name, if one is not specified via the annotation).

iii. Init params for servlets and filters defined via annotations, will be overridden in the descriptor if the name of the init param exactly matches the name specified via the annotation. Init params are additive between the annotations and descriptors.

iv. `url-patterns`, when specified in a descriptor for a given servlet name overrides the url patterns specified via the annotation.

v. For a filter defined via the `@WebFilter` annotation, to override values via the descriptor, the name of the filter in the descriptor MUST match the name of the filter specified via the annotation (explicitly specified or the default name, if one is not specified via the annotation).

vi. `url-patterns` to which a filter is applied, when specified in a descriptor for a given filter name overrides the url patterns specified via the annotation.

vii. DispatcherTypes to which a filter applies, when specified in a descriptor for a given filter name overrides the DispatcherTypes specified via the annotation.

viii. The following examples demonstrates some of the above rules -

A Servlet declared via an annotation and packaged with the corresponding `web.xml` in the descriptor

```java
@WebServlet(urlPatterns="/MyPattern", initParams=
{@WebInitParam(name="ccc", value="333")})
public class com.acme.Foo extends HttpServlet
{
    ...
}
```
web.xml

<servlet>
  <servlet-class>com.acme.Foo</servlet-class>
  <servlet-name>Foo</servlet-name>
  <init-param>
    <param-name>aaa</param-name>
    <param-value>111</param-value>
  </init-param>
</servlet>

<servlet>
  <servlet-class>com.acme.Foo</servlet-class>
  <servlet-name>Fum</servlet-name>
  <init-param>
    <param-name>bbb</param-name>
    <param-value>222</param-value>
  </init-param>
</servlet>

<servlet-mapping>
  <servlet-name>Foo</servlet-name>
  <url-pattern>/foo/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>Fum</servlet-name>
  <url-pattern>/fum/*</url-pattern>
</servlet-mapping>

Since the name of the servlet declared via the annotation does not match the name of the servlet declared in the web.xml, the annotation specifies a new servlet declaration in addition to the other declarations in web.xml and is equivalent to:

<servlet>
  <servlet-class>com.acme.Foo</servlet-class>
  <servlet-name>com.acme.Foo</servlet-name>
</servlet>
<init-param>
  <param-name>ccc</param-name>
  <param-value>333</param-value>
</init-param>

If the above web.xml were replaced with the following

<servlet>
  <servlet-class>com.acme.Foo</servlet-class>
  <servlet-name>com.acme.Foo</servlet-name>
  <init-param>
    <param-name>aaa</param-name>
    <param-value>111</param-value>
  </init-param>
</servlet>

<servlet-mapping>
  <servlet-name>com.acme.Foo</servlet-name>
  <url-pattern>/foo/*</url-pattern>
</servlet-mapping>

Then the effective descriptor would be equivalent to

<servlet>
  <servlet-class>com.acme.Foo</servlet-class>
  <servlet-name>com.acme.Foo</servlet-name>
  <init-param>
    <param-name>aaa</param-name>
    <param-value>111</param-value>
  </init-param>
  <init-param>
    <param-name>ccc</param-name>
    <param-value>333</param-value>
  </init-param>
</servlet>

<servlet-mapping>
  <servlet-name>com.acme.Foo</servlet-name>
  <url-pattern>/foo/*</url-pattern>
</servlet-mapping>
8.2.4 Shared libraries / runtimes pluggability

In addition to supporting fragments and use of annotations one of the requirements is that not only we be able to plug-in things that are bundled in the WEB-INF/lib but also plugin shared copies of frameworks - including being able to plug-in to the web container things like JAX-WS, JAX-RS and JSF that build on top of the web container. The ServletContainerInitializer allows handling such a use case as described below.

The ServletContainerInitializer class is looked up via the jar services API. For each application, an instance of the ServletContainerInitializer is created by the container at application startup time. The framework providing an implementation of the ServletContainerInitializer MUST bundle in the META-INF/services directory of the jar file a file called javax.servlet.ServletContainerInitializer, as per the jar services API, that points to the implementation class of the ServletContainerInitializer.

In addition to the ServletContainerInitializer we also have an annotation - HandlesTypes. The HandlesTypes annotation on the implementation of the ServletContainerInitializer is used to express interest in classes that may have annotations (type, method or field level annotations) specified in the value of the HandlesTypes or if it extends / implements one those classes anywhere in the class’ super types. The HandlesTypes annotation is applied irrespective of the setting of metadata-complete.

When examining the classes of an application to see if they match any of the criteria specified by the HandlesTypes annotation of a ServletContainerInitializer, the container may run into class loading problems if one or more of the application’s optional JAR files are missing. Since the container is not in a position to decide whether these types of class loading failures will prevent the application from working correctly, it must ignore them, while at the same time providing a configuration option that would log them.

If an implementation of ServletContainerInitializer does not have the @HandlesTypes annotation, or if there are no matches to any of the HandlesType specified, then it will get invoked once for every application with null as the value of the Set. This will allow for the initializer to determine based on the resources available in the application whether it needs to initialize a servlet / filter or not.

The onStartup method of the ServletContainerInitializer will be invoked when the application is coming up before any of the servlet listener events are fired.

The onStartup method of the ServletContainerInitializer is called with a Set of Classes that either extend / implement the classes that the initializer expressed interest in or if it is annotated with any of the classes specified via the @HandlesTypes annotation.

A concrete example below showcases how this would work.
Let's take the JAX-WS web services runtime.

The implementation of JAX-WS runtime isn't typically bundled in each and every war file. The implementation would bundle an implementation of the ServletContainerInitializer (shown below) and the container would look that up using the services API (the jar file will bundle in it's META-INF/services directory a file called javax.servlet.ServletContainerInitializer that will point to the JAXWSServletContainerInitializer shown below).

```java
@HandlesTypes(WebService.class)
JAXWSServletContainerInitializer
    implements ServletContainerInitializer
{
    public void onStartup(Set<Class<?>> c, ServletContext ctx)
        throws ServletException {
        // JAX-WS specific code here to initialize the runtime
        // and setup the mapping etc.
        ServletRegistration reg = ctx.addServlet("JAXWSServlet",
            "com.sun.webservice.JAXWSServlet");
        reg.addServletMapping("/foo");
    }
}
```

The framework jar file can also be bundled in WEB-INF/lib directory of the war file. If the ServletContainerInitializer is bundled in a JAR file inside the WEB-INF/lib directory of an application, it's onStartup method will be invoked only once during the startup of the bundling application. If, on the other hand, the ServletContainerInitializer is bundled in a JAR file outside of the WEB-INF/lib directory, but still discoverable by the runtime's service provider lookup mechanism, it's onStartup method will be invoked every time an application is started.

Implementations of the ServletContainerInitializer interface will be discovered by the runtime's service lookup mechanism or a container specific mechanism that is semantically equivalent to it. In either case, ServletContainerInitializer services from web fragment JAR files that are excluded from an absolute ordering MUST be ignored, and the order in which these services are discovered MUST follow the application's class loading delegation model.
8.3 JSP container pluggability

The ServletContainerInitializer and programmatic registration features make it possible to provide a clear separation of responsibilities between the Servlet and JSP containers, by making the Servlet container responsible for parsing only web.xml and web-fragment.xml resources, and delegating the parsing of Tag Library Descriptor (TLD) resources to the JSP container.

Previously, a web container had to scan TLD resources for any listener declarations. With Servlet 3.0 and later versions, this responsibility may be delegated to the JSP container. A JSP container that is embedded in a Servlet container may provide its own ServletContainerInitializer implementation, search the ServletContext passed to its onStartup method for any TLD resources, scan those resources for listener declarations, and register the corresponding listeners with the ServletContext.

In addition, prior to Servlet 3.0, a JSP container used to have to scan an application's deployment descriptor for any jsp-config related configuration. With Servlet 3.0 and later versions, the Servlet container must make available, via the ServletContext.getJspConfigDescriptor method, any jsp-config related configuration from the application's web.xml and web-fragment.xml deployment descriptors.

Any ServletContextListeners that were discovered in a TLD and registered programmatically are limited in the functionality they provide. Any attempt to call a ServletContext API methods on them that was added since Servlet 3.0 will result in an UnsupportedOperationException.

In addition, a Servlet container compliant with Servlet 3.0 or later versions must provide a ServletContext attribute with name javax.servlet.context.orderedLibs, whose value (of type java.util.List&lt;java.lang.String&gt;) contains the list of names of JAR files in the WEB-INF/lib directory of the application represented by the ServletContext, ordered by their web fragment names (with possible exclusions if fragment JAR files have been excluded from absolute-ordering), or null if the application does not specify any absolute or relative ordering.

8.4 Processing annotations and fragments

Web applications can include both annotations and the web.xml / web-fragment.xml deployment descriptors. If there is no deployment descriptor, or there is one but does not have the metadata-complete set to true, web.xml, web-
fragment.xml and annotations if used in the application must be processed. The following table describes whether or not to process annotations and web.xml fragments.

<table>
<thead>
<tr>
<th>Deployment descriptor</th>
<th>metadata-complete</th>
<th>process annotations and web fragments</th>
</tr>
</thead>
<tbody>
<tr>
<td>web.xml 2.5</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>web.xml 2.5</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>web.xml 3.0 or later</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>web.xml 3.0 or later</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
Dispatching Requests

When building a Web application, it is often useful to forward processing of a request to another servlet, or to include the output of another servlet in the response. The RequestDispatcher interface provides a mechanism to accomplish this.

When asynchronous processing is enabled on the request, the AsyncContext allows a user to dispatch the request back to the servlet container.

9.1 Obtaining a RequestDispatcher

An object implementing the RequestDispatcher interface may be obtained from the ServletContext via the following methods:

- getRequestDispatcher
- getNamedDispatcher

The getRequestDispatcher method takes a String argument describing a path within the scope of the ServletContext. This path must be relative to the root of the ServletContext and begin with a ‘/’, or be empty. The method uses the path to look up a servlet, using the servlet path matching rules in Chapter 12, “Mapping Requests to Servlets”, wraps it with a RequestDispatcher object, and returns the resulting object. If no servlet can be resolved based on the given path, a RequestDispatcher is provided that returns the content for that path.

The getNamedDispatcher method takes a String argument indicating the name of a servlet known to the ServletContext. If a servlet is found, it is wrapped with a RequestDispatcher object and the object is returned. If no servlet is associated with the given name, the method must return null.
To allow RequestDispatcher objects to be obtained using relative paths that are relative to the path of the current request (not relative to the root of the ServletContext), the getRequestDispatcher method is provided in the ServletRequest interface.

The behavior of this method is similar to the method of the same name in the ServletContext. The servlet container uses information in the request object to transform the given relative path against the current servlet to a complete path. For example, in a context rooted at ‘/’ and a request to /garden/tools.html, a request dispatcher obtained via ServletRequest.getRequestDispatcher("header.html") will behave exactly like a call to ServletContext.getRequestDispatcher("/garden/header.html").

9.1.1 Query Strings in Request Dispatcher Paths

The ServletContext and ServletRequest methods that create RequestDispatcher objects using path information allow the optional attachment of query string information to the path. For example, a Developer may obtain a RequestDispatcher by using the following code:

```java
String path = "/raisins.jsp?orderno=5";
RequestDispatcher rd = context.getRequestDispatcher(path);
rd.include(request, response);
```

Parameters specified in the query string used to create the RequestDispatcher take precedence over other parameters of the same name passed to the included servlet. The parameters associated with a RequestDispatcher are scoped to apply only for the duration of the include or forward call.

9.2 Using a Request Dispatcher

To use a request dispatcher, a servlet calls either the include method or forward method of the RequestDispatcher interface. The parameters to these methods can be either the request and response arguments that were passed in via the service method of the javax.servlet.Servlet interface, or instances of subclasses of the request and response wrapper classes that were introduced for version 2.3 of the specification. In the latter case, the wrapper instances must wrap the request or response objects that the container passed into the service method.

The Container Provider should ensure that the dispatch of the request to a target servlet occurs in the same thread of the same JVM as the original request.
9.3 The Include Method

The include method of the RequestDispatcher interface may be called at any time. The target servlet of the include method has access to all aspects of the request object, but its use of the response object is more limited.

It can only write information to the ServletOutputStream or Writer of the response object and commit a response by writing content past the end of the response buffer, or by explicitly calling the flushBuffer method of the ServletResponse interface. It cannot set headers or call any method that affects the headers of the response, with the exception of the HttpServletRequest.getSession() and HttpServletRequest.getSession(boolean) methods. Any attempt to set the headers must be ignored, and any call to HttpServletRequest.getSession() or HttpServletRequest.getSession(boolean) that would require adding a Cookie response header must throw an IllegalStateException if the response has been committed.

If the default servlet is the target of a RequestDispatcher.include() and the requested resource does not exist, then the default servlet MUST throw FileNotFoundException. If the exception isn’t caught and handled, and the response hasn’t been committed, the status code MUST be set to 500.

9.3.1 Included Request Parameters

Except for servlets obtained by using the getNamedDispatcher method, a servlet that has been invoked by another servlet using the include method of RequestDispatcher has access to the path by which it was invoked.

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 via the getAttribute method on the request object 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.

### 9.4 The Forward Method

The `forward` method of the `RequestDispatcher` interface may be called by the calling servlet only when no output has been committed to the client. If output data exists in the response buffer that has not been committed, the content must be cleared before the target servlet’s `service` method is called. If the response has been committed, an `IllegalStateException` must be thrown.

The path elements of the request object exposed to the target servlet must reflect the path used to obtain the `RequestDispatcher`.

The only exception to this is if the `RequestDispatcher` was obtained via the `getNamedDispatcher` method. In this case, the path elements of the request object must reflect those of the original request.

Before the `forward` method of the `RequestDispatcher` interface returns without exception, the response content must be sent and committed, and closed by the servlet container, unless the request was put into the asynchronous mode. If an error occurs in the target of the `RequestDispatcher.forward()` the exception may be propagated back through all the calling filters and servlets and eventually back to the container.

#### 9.4.1 Query String

The request dispatching mechanism is responsible for aggregating query string parameters when forwarding or including requests.

#### 9.4.2 Forwarded Request Parameters

Except for servlets obtained by using the `getNamedDispatcher` method, a servlet that has been invoked by another servlet using the `forward` method of `RequestDispatcher` has access to the path of the original request.
The following request attributes must be set:

<table>
<thead>
<tr>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.servlet.forward.request_uri</td>
</tr>
<tr>
<td>javax.servlet.forward.context_path</td>
</tr>
<tr>
<td>javax.servlet.forward.servlet_path</td>
</tr>
<tr>
<td>javax.servlet.forward.path_info</td>
</tr>
<tr>
<td>javax.servlet.forward.query_string</td>
</tr>
</tbody>
</table>

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 call chain that received the request from the client.

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 original request even under the situation that multiple forwards and subsequent includes are called.

If the forwarded servlet was obtained by using the getNamedDispatcher method, these attributes must not be set.

---

### 9.5 Error Handling

If the servlet that is the target of a request dispatcher throws a runtime exception or a checked exception of type ServletException or IOException, it should be propagated to the calling servlet. All other exceptions should be wrapped as ServletException and the root cause of the exception set to the original exception, as it should not be propagated.

---

### 9.6 Obtaining an AsyncContext

An object implementing the AsyncContext interface may be obtained from the ServletRequest via one of startAsync methods. Once you have an AsyncContext, you can use it to either complete the processing of the request via the complete() method or use one of the dispatch methods described below.
9.7 The Dispatch Method

The following methods can be used to dispatch requests from the AsyncContext:

- `dispatch(path)`
  
The `dispatch` method takes a `String` argument describing a path within the scope of the ServletContext. This path must be relative to the root of the ServletContext and begin with a `'/'.`

- `dispatch(servletContext, path)`
  
The `dispatch` method takes a `String` argument describing a path within the scope of the ServletContext specified. This path must be relative to the root of the ServletContext specified and begin with a `'/'.`

- `dispatch()`
  
The `dispatch` method takes no argument. It uses the original URI as the path. If the AsyncContext was initialized via the `startAsync(ServletRequest, ServletResponse)` and the request passed is an instance of `HttpServletRequest`, then the dispatch is to the URI returned by `HttpServletRequest.getRequestURI()`. Otherwise the dispatch is to the URI of the request when it was last dispatched by the container.

One of the `dispatch` methods of the AsyncContext interface may be called by the application waiting for the asynchronous event to happen. If `complete()` has been called on the AsyncContext, an `IllegalStateException` must be thrown. All the variations of the dispatch methods returns immediately and do not commit the response.

The path elements of the request object exposed to the target servlet must reflect the path specified in the `AsyncContext.dispatch`.

9.7.1 Query String

The request dispatching mechanism is responsible for aggregating query string parameters when dispatching requests.

9.7.2 Dispatched Request Parameters

A servlet that has been invoked by using the `dispatch` method of `AsyncContext` has access to the path of the original request.
The following request attributes must be set:

<table>
<thead>
<tr>
<th>javax.servlet.async.request_uri</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.servlet.async.context_path</td>
</tr>
<tr>
<td>javax.servlet.async.servlet_path</td>
</tr>
<tr>
<td>javax.servlet.async.path_info</td>
</tr>
<tr>
<td>javax.servlet.async.query_string</td>
</tr>
</tbody>
</table>

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 call chain that received the request from the client.

These attributes are accessible from the dispatched servlet via the `getAttribute` method on the request object. Note that these attributes must always reflect the information in the original request even under the situation that multiple dispatches are called.
Web Applications

A Web application is a collection of servlets, HTML pages, classes, and other resources that make up a complete application on a Web server. The Web application can be bundled and run on multiple containers from multiple vendors.

10.1 Web Applications Within Web Servers

A Web application is rooted at a specific path within a Web server. For example, a catalog application could be located at http://www.mycorp.com/catalog. All requests that start with this prefix will be routed to the ServletContext which represents the catalog application.

A servlet container can establish rules for automatic generation of Web applications. For example a ~user/ mapping could be used to map to a Web application based at /home/user/public_html/.

By default, an instance of a Web application must run on one VM at any one time. This behavior can be overridden if the application is marked as “distributable” via its deployment descriptor. An application marked as distributable must obey a more restrictive set of rules than is required of a normal Web application. These rules are set out throughout this specification.

10.2 Relationship to ServletContext

The servlet container must enforce a one to one correspondence between a Web application and a ServletContext. A ServletContext object provides a servlet with its view of the application.
10.3 Elements of a Web Application

A Web application may consist of the following items:

■ Servlets
■ JSP™ Pages
■ Utility Classes
■ Static documents (HTML, images, sounds, etc.)
■ Client side Java applets, beans, and classes
■ Descriptive meta information that ties all of the above elements together

10.4 Deployment Hierarchies

This specification defines a hierarchical structure used for deployment and packaging purposes that can exist in an open file system, in an archive file, or in some other form. It is recommended, but not required, that servlet containers support this structure as a runtime representation.

10.5 Directory Structure

A Web application exists as a structured hierarchy of directories. The root of this hierarchy serves as the document root for files that are part of the application. For example, for a Web application with the context path /catalog in a Web container, the index.html file at the base of the Web application hierarchy or in a JAR file inside WEB-INF/lib that includes the index.html under META-INF/resources directory can be served to satisfy a request from /catalog/index.html. If an index.html is present both in the root context and in the META-INF/resources directory of a JAR file in the WEB-INF/lib directory of the application, then the file that is available in the root context MUST be used. The rules for matching URLs to context path are laid out in Chapter 12, “Mapping Requests to Servlets”. Since the context path of an application determines the URL namespace of the contents of the Web application, Web containers must reject Web applications defining a context path that could cause potential conflicts in this URL namespace. This may occur, for example, by attempting to deploy a second Web application with the same context

path. Since requests are matched to resources in a case-sensitive manner, this
determination of potential conflict must be performed in a case-sensitive manner as well.

A special directory exists within the application hierarchy named “WEB-INF”. This
directory contains all things related to the application that aren’t in the document root of the application. Most of the WEB-INF node is not part of the public document tree of the application. Except for static resources and JSPs packaged in the META-INF/resources of a JAR file that resides in the WEB-INF/lib directory, no other files contained in the WEB-INF directory may be served directly to a client by the container. However, the contents of the WEB-INF directory are visible to servlet code using the getResource and getResourceAsStream method calls on the ServletContext, and may be exposed using the RequestDispatcher calls. Hence, if the Application Developer needs access, from servlet code, to application specific configuration information that he does not wish to be exposed directly to the Web client, he may place it under this directory. Since requests are matched to resource mappings in a case-sensitive manner, client requests for ‘/WEB-INF/foo’, ‘/WEB-iNf/foo’, for example, should not result in contents of the Web application located under /WEB-INF being returned, nor any form of directory listing thereof.

The contents of the WEB-INF directory are:

- The /WEB-INF/web.xml deployment descriptor.
- The /WEB-INF/classes/ directory for servlet and utility classes. The classes in this directory must be available to the application class loader.
- The /WEB-INF/lib/*.jar area for Java ARchive files. These files contain servlets, beans, static resources and JSPs packaged in a JAR file and other utility classes useful to the Web application. The Web application class loader must be able to load classes from any of these archive files.

The Web application class loader must load classes from the WEB-INF/classes directory first, and then from library JARs in the WEB-INF/lib directory. Also, except for the case where static resources are packaged in JAR files, any requests from the client to access the resources in WEB-INF/ directory must be returned with a SC_NOT_FOUND(404) response.
10.5.1 Example of Application Directory Structure

The following is a listing of all the files in a sample Web application:

```
/index.html
/howto.jsp
/feedback.jsp
/images/banner.gif
/images/jumping.gif
/WEB-INF/web.xml
/WEB-INF/lib/jspbean.jar
/WEB-INF/lib/catalog.jar!/META-INF/resources/catalog/moreOffers/books.html
/WEB-INF/classes/com/mycorp/servlets/MyServlet.class
/WEB-INF/classes/com/mycorp/util/MyUtils.class
```

10.6 Web Application Archive File

Web applications can be packaged and signed into a Web ARchive format (WAR) file using the standard Java archive tools. For example, an application for issue tracking might be distributed in an archive file called `issuetracks.war`.

When packaged into such a form, a `META-INF` directory will be present which contains information useful to Java archive tools. This directory must not be directly served as content by the container in response to a Web client’s request, though its contents are visible to servlet code via the `getResource` and `getResourceAsStream` calls on the `ServletContext`. Also, any requests to access the resources in `META-INF` directory must be returned with a `SC_NOT_FOUND(404)` response.

10.7 Web Application Deployment Descriptor

The Web application deployment descriptor (see Chapter 14, “Deployment Descriptor”) includes the following types of configuration and deployment information:

- `ServletContext` Init Parameters
- Session Configuration
- Servlet/JSP Definitions
- Servlet/JSP Mappings
10.7.1 Dependencies On Extensions

When a number of applications make use of the same code or resources, they will typically be installed as library files in the container. These files are often common or standard APIs that can be used without sacrificing portability. Files used only by one or a few applications will be made available for access as part of the Web application. The container must provide a directory for these libraries. The files placed within this directory must be available across all Web applications. The location of this directory is container-specific. The class loader the servlet container uses for loading these library files must be the same for all Web applications within the same JVM. This class loader instance must be somewhere in the chain of parent class loaders of the Web application class loader.

Application developers need to know what extensions are installed on a Web container, and containers need to know what dependencies servlets in a WAR have on such libraries in order to preserve portability.

The application developer depending on such an extension or extensions must provide a \texttt{META-INF/MANIFEST.MF} entry in the WAR file listing all extensions needed by the WAR. The format of the manifest entry should follow standard JAR manifest format. During deployment of the Web application, the Web container must make the correct versions of the extensions available to the application following the rules defined by the \textit{Optional Package Versioning} mechanism (http://java.sun.com/j2se/1.4/docs/guide/extensions/).

Web containers must also be able to recognize declared dependencies expressed in the manifest entry of any of the library JARs under the \texttt{WEB-INF/lib} entry in a WAR.

If a Web container is not able to satisfy the dependencies declared in this manner, it should reject the application with an informative error message.

10.7.2 Web Application Class Loader

The class loader that a container uses to load a servlet in a WAR must allow the developer to load any resources contained in library JARs within the WAR following normal Java SE semantics using \texttt{getResource}. As described in the Java EE license agreement, servlet containers that are not part of a Java EE product should not allow the application to override Java SE platform classes, such as those in the java.* and javax.* namespaces, that Java SE does not allow to be modified. The container
should not allow applications to override or access the container’s implementation classes. It is recommended also that the application class loader be implemented so that classes and resources packaged within the WAR are loaded in preference to classes and resources residing in container-wide library JARs. An implementation MUST also guarantee that for every web application deployed in a container, a call to Thread.currentThread().getContextClassLoader() MUST return a ClassLoader instance that implements the contract specified in this section. Furthermore, the ClassLoader instance MUST be a separate instance for each deployed web application. The container is required to set the thread context ClassLoader as described above before making any callbacks (including listener callbacks) into the web application, and set it back to the original ClassLoader, once the callback returns.

10.8 Replacing a Web Application

A server should be able to replace an application with a new version without restarting the container. When an application is replaced, the container should provide a robust method for preserving session data within that application.

10.9 Error Handling

10.9.1 Request Attributes

A Web application must be able to specify that when errors occur, other resources in the application are used to provide the content body of the error response. The specification of these resources is done in the deployment descriptor.

If the location of the error handler is a servlet or a JSP page:

■ The original unwrapped request and response objects created by the container are passed to the servlet or JSP page.
■ The request path and attributes are set as if a RequestDispatcher.forward to the error resource had been performed.
The request attributes in TABLE 10-1 must be set.

<table>
<thead>
<tr>
<th>Request Attributes</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.servlet.error.status_code</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>javax.servlet.error.exception_type</td>
<td>java.lang.Class</td>
</tr>
<tr>
<td>javax.servlet.error.message</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>javax.servlet.error.exception</td>
<td>java.lang.Throwable</td>
</tr>
<tr>
<td>javax.servlet.error.request_uri</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>javax.servlet.error.servlet_name</td>
<td>java.lang.String</td>
</tr>
</tbody>
</table>

These attributes allow the servlet to generate specialized content depending on the status code, the exception type, the error message, the exception object propagated, and the URI of the request processed by the servlet in which the error occurred (as determined by the `getRequestURI` call), and the logical name of the servlet in which the error occurred.

With the introduction of the exception object to the attributes list for version 2.3 of this specification, the exception type and error message attributes are redundant. They are retained for backwards compatibility with earlier versions of the API.

10.9.2 Error Pages

To allow developers to customize the appearance of content returned to a Web client when a servlet generates an error, the deployment descriptor defines a list of error page descriptions. The syntax allows the configuration of resources to be returned by the container either when a servlet or filter calls `sendError` on the response for specific status codes, or if the servlet generates an exception or error that propagates to the container.

If the `sendError` method is called on the response, the container consults the list of error page declarations for the Web application that use the status-code syntax and attempts a match. If there is a match, the container returns the resource as indicated by the location entry.

A servlet or filter may throw the following exceptions during processing of a request:

- runtime exceptions or errors
- ServletException or subclasses thereof
- IOExceptions or subclasses thereof
The Web application may have declared error pages using the `exception-type` element. In this case the container matches the exception type by comparing the exception thrown with the list of error-page definitions that use the `exception-type` element. A match results in the container returning the resource indicated in the location entry. The closest match in the class hierarchy wins.

If no error-page declaration containing an `exception-type` fits using the class-hierarchy match, and the exception thrown is a `ServletException` or subclass thereof, the container extracts the wrapped exception, as defined by the `ServletException.getRootCause` method. A second pass is made over the error page declarations, again attempting the match against the error page declarations, but using the wrapped exception instead.

Error-page declarations using the `exception-type` element in the deployment descriptor must be unique up to the class name of the exception-type. Similarly, error-page declarations using the `status-code` element must be unique in the deployment descriptor up to the status code.

If an error-page element in the deployment descriptor does not contain an `exception-type` or an `error-code` element, the error page is a default error page.

The error page mechanism described does not intervene when errors occur when invoked using the `RequestDispatcher` or `filter.doFilter` method. In this way, a filter or servlet using the `RequestDispatcher` has the opportunity to handle errors generated.

If a servlet generates an error that is not handled by the error page mechanism as described above, the container must ensure to send a response with status 500.

The default servlet and container will use the `sendError` method to send 4xx and 5xx status responses, so that the error mechanism may be invoked. The default servlet and container will use the `setStatus` method for 2xx and 3xx responses and will not invoke the error page mechanism.

If the application is using asynchronous operations as described in Section 2.3.3.3, “Asynchronous processing” on page 2-10, it is the application’s responsibility to handle all errors in application created threads. The container MAY take care of the errors from the thread issued via `AsyncContext.start`. For handling errors that occur during `AsyncContext.dispatch` see Section n, “Any errors or exceptions that may occur during the execution of the dispatch methods MUST be caught and handled by the container as follows:” on page 2-16
10.9.3 Error Filters

The error page mechanism operates on the original unwrapped/unfiltered request and response objects created by the container. The mechanism described in Section 6.2.5, "Filters and the RequestDispatcher" may be used to specify filters that are applied before an error response is generated.

10.10 Welcome Files

Web Application developers can define an ordered list of partial URIs called welcome files in the Web application deployment descriptor. The deployment descriptor syntax for the list is described in the Web application deployment descriptor schema.

The purpose of this mechanism is to allow the deployer to specify an ordered list of partial URIs for the container to use for appending to URIs when there is a request for a URI that corresponds to a directory entry in the WAR not mapped to a Web component. This kind of request is known as a valid partial request.

The use for this facility is made clear by the following common example: A welcome file of 'index.html' can be defined so that a request to a URL like host:port/webapp/directory/, where 'directory' is an entry in the WAR that is not mapped to a servlet or JSP page, is returned to the client as 'host:port/webapp/directory/index.html'.

If a Web container receives a valid partial request, the Web container must examine the welcome file list defined in the deployment descriptor. The welcome file list is an ordered list of partial URLs with no trailing or leading /. The Web server must append each welcome file in the order specified in the deployment descriptor to the partial request and check whether a static resource in the WAR is mapped to that request URI. If no match is found, the Web server MUST again append each welcome file in the order specified in the deployment descriptor to the partial request and check if a servlet is mapped to that request URI. The Web container must send the request to the first resource in the WAR that matches. The container may send the request to the welcome resource with a forward, a redirect, or a container specific mechanism that is indistinguishable from a direct request.

If no matching welcome file is found in the manner described, the container may handle the request in a manner it finds suitable. For some configurations this may mean returning a directory listing or for others returning a 404 response.

Consider a Web application where:
The deployment descriptor lists the following welcome files.

```
<welcome-file-list>
  <welcome-file>index.html</welcome-file>
  <welcome-file>default.jsp</welcome-file>
</welcome-file-list>
```

The static content in the WAR is as follows.

```
/foo/index.html
/foo/default.jsp
/foo/orderform.html
/foo/home.gif
/catalog/default.jsp
/catalog/products/shop.jsp
/catalog/products/register.jsp
```

- A request URI of `/foo` will be redirected to a URI of `/foo/`.
- A request URI of `/foo/` will be returned as `/foo/index.html`.
- A request URI of `/catalog` will be redirected to a URI of `/catalog/`.
- A request URI of `/catalog/` will be returned as `/catalog/default.jsp`.
- A request URI of `/catalog/index.html` will cause a 404 not found.
- A request URI of `/catalog/products` will be redirected to a URI of `/catalog/products/`.
- A request URI of `/catalog/products/` will be passed to the “default” servlet, if any. If no “default” servlet is mapped, the request may cause a 404 not found, may cause a directory listing including `shop.jsp` and `register.jsp`, or may cause other behavior defined by the container. See Section 12.2, “Specification of Mappings” for the definition of “default” servlet.
- All of the above static content can also be packaged in a JAR file with the content listed above packaged in the META-INF/resources directory of the jar file. The JAR file can then be included in the WEB-INF/lib directory of the web application.

10.11 Web Application Environment

Servlet containers that are not part of a Java EE technology-compliant implementation are encouraged, but not required, to implement the application environment functionality described in Section 15.2.2, “Web Application
Environment and the Java EE specification. If they do not implement the facilities required to support this environment, upon deploying an application that relies on them, the container should provide a warning.

## 10.12 Web Application Deployment

When a web application is deployed into a container, the following steps must be performed, in this order, before the web application begins processing client requests.

- Instantiate an instance of each event listener identified by a `<listener>` element in the deployment descriptor.
- For instantiated listener instances that implement `ServletContextListener`, call the `contextInitialized()` method.
- Instantiate an instance of each filter identified by a `<filter>` element in the deployment descriptor and call each filter instance’s `init()` method.
- Instantiate an instance of each servlet identified by a `<servlet>` element that includes a `<load-on-startup>` element in the order defined by the load-on-startup element values, and call each servlet instance’s `init()` method.

## 10.13 Inclusion of a web.xml Deployment Descriptor

A web application is NOT required to contain a web.xml if it does NOT contain any Servlet, Filter, or Listener components or is using annotations to declare the same. In other words an application containing only static files or JSP pages does not require a web.xml to be present.
CHAPTER 11

Application Lifecycle Events

11.1 Introduction

The application events facility gives the Web Application Developer greater control over the lifecycle of the ServletContext and HttpSession and ServletRequest, allows for better code factorization, and increases efficiency in managing the resources that the Web application uses.

11.2 Event Listeners

Application event listeners are classes that implement one or more of the servlet event listener interfaces. They are instantiated and registered in the Web container at the time of the deployment of the Web application. They are provided by the Developer in the WAR.

Servlet event listeners support event notifications for state changes in the ServletContext, HttpSession and ServletRequest objects. Servlet context listeners are used to manage resources or state held at a JVM level for the application. HTTP session listeners are used to manage state or resources associated with a series of requests made into a Web application from the same client or user. Servlet request listeners are used to manage state across the lifecycle of servlet requests. Async listeners are used to manage async events such as time outs and completion of async processing.

There may be multiple listener classes listening to each event type, and the Developer may specify the order in which the container invokes the listener beans for each event type.
11.2.1 Event Types and Listener Interfaces

Events types and the listener interfaces used to monitor them are shown in the following tables:

### TABLE 11-1  Servlet Context Events

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Description</th>
<th>Listener Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lifecycle</td>
<td>The servlet context has just been created and is available to service its first request, or the servlet context is about to be shut down.</td>
<td>javax.servlet.ServletContextListener</td>
</tr>
<tr>
<td>Changes to attributes</td>
<td>Attributes on the servlet context have been added, removed, or replaced.</td>
<td>javax.servlet.ServletContextAttributeListener</td>
</tr>
</tbody>
</table>

### TABLE 11-2  HTTP Session Events

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Description</th>
<th>Listener Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lifecycle</td>
<td>An HttpSession has been created, invalidated, or timed out.</td>
<td>javax.servlet.http.HttpSessionListener</td>
</tr>
<tr>
<td>Changes to attributes</td>
<td>Attributes have been added, removed, or replaced on an HttpSession.</td>
<td>javax.servlet.http.HttpSessionAttributeListener</td>
</tr>
<tr>
<td>Changes to id</td>
<td>The id of HttpSession has been changed.</td>
<td>javax.servlet.http.HttpSessionIdListener</td>
</tr>
<tr>
<td>Session migration</td>
<td>HttpSession has been activated or passivated.</td>
<td>javax.servlet.http.HttpSessionActivationListener</td>
</tr>
<tr>
<td>Object binding</td>
<td>Object has been bound to or unbound from HttpSession</td>
<td>javax.servlet.http.HttpSessionBindingListener</td>
</tr>
</tbody>
</table>
11.2.2 An Example of Listener Use

To illustrate a use of the event scheme, consider a simple Web application containing a number of servlets that make use of a database. The Developer has provided a servlet context listener class for management of the database connection.

1. When the application starts up, the listener class is notified. The application logs on to the database, and stores the connection in the servlet context.

2. Servlets in the application access the connection as needed during activity in the Web application.

3. When the Web server is shut down, or the application is removed from the Web server, the listener class is notified and the database connection is closed.

11.3 Listener Class Configuration

11.3.1 Provision of Listener Classes

The Developer of the Web application provides listener classes implementing one or more of the listener interfaces in the javax.servlet API. Each listener class must have a public constructor taking no arguments. The listener classes are packaged into the WAR, either under the WEB-INF/classes archive entry, or inside a JAR in the WEB-INF/lib directory.
11.3.2 Deployment Declarations

Listener classes are declared in the Web application deployment descriptor using the listener element. They are listed by class name in the order in which they are to be invoked. Unlike other listeners, listeners of type AsyncListener may only be registered (with a ServletRequest) programmatically.

11.3.3 Listener Registration

The Web container creates an instance of each listener class and registers it for event notifications prior to the processing of the first request by the application. The Web container registers the listener instances according to the interfaces they implement and the order in which they appear in the deployment descriptor. During web application execution, listeners for the given events are mostly invoked in their registration orders, but there are some exceptions. For instance, HttpSessionListener.isDirectory are invoked in reverse order. See Section 8.2.3, “Assembling the descriptor from web.xml, web-fragment.xml and annotations” for details.

11.3.4 Notifications At Shutdown

On application shutdown, listeners are notified in reverse order to their declarations with notifications to session listeners preceding notifications to context listeners. Session listeners must be notified of session invalidations prior to context listeners being notified of application shutdown.

11.4 Deployment Descriptor Example

The following example is the deployment grammar for registering two servlet context lifecycle listeners and an HttpSession listener.

com.acme.MyConnectionManager to be notified of servlet context lifecycle events before com.acme.MyLoggingModule. Here is the deployment descriptor for this application:

```xml
<web-app>
  <display-name>MyListeningApplication</display-name>
  <listener>
    <listener-class>com.acme.MyConnectionManager</listener-class>
  </listener>
  <listener>
    <listener-class>com.acme.MyLoggingModule</listener-class>
  </listener>
  <servlet>
    <display-name>RegistrationServlet</display-name>
    ...etc
  </servlet>
</web-app>
```

### 11.5 Listener Instances and Threading

The container is required to complete instantiation of the listener classes in a Web application prior to the start of execution of the first request into the application. The container must maintain a reference to each listener instance until the last request is serviced for the Web application.

Attribute changes to `ServletContext` and `HttpSession` objects may occur concurrently. The container is not required to synchronize the resulting notifications to attribute listener classes. Listener classes that maintain state are responsible for the integrity of the data and should handle this case explicitly.

### 11.6 Listener Exceptions

Application code inside a listener may throw an exception during operation. Some listener notifications occur under the call tree of another component in the application. An example of this is a servlet that sets a session attribute, where the session listener throws an unhandled exception. The container must allow unhandled exceptions to be handled by the error page mechanism described in
Section 10.9, “Error Handling”. If there is no error page specified for those exceptions, the container must ensure to send a response back with status 500. In this case no more listeners under that event are called.

Some exceptions do not occur under the call stack of another component in the application. An example of this is a SessionListener that receives a notification that a session has timed out and throws an unhandled exception, or of a ServletContextListener that throws an unhandled exception during a notification of servlet context initialization, or of a ServletRequestListener that throws an unhandled exception during a notification of the initialization or the destruction of the request object. In this case, the Developer has no opportunity to handle the exception. The container may respond to all subsequent requests to the Web application with an HTTP status code 500 to indicate an application error.

Developers wishing normal processing to occur after a listener generates an exception must handle their own exceptions within the notification methods.

### 11.7 Distributed Containers

In distributed Web containers, HttpSession instances are scoped to the particular JVM servicing session requests, and the ServletContext object is scoped to the Web container’s JVM. Distributed containers are not required to propagate either servlet context events or HttpSession events to other JVMs. Listener class instances are scoped to one per deployment descriptor declaration per JVM.

### 11.8 Session Events

Listener classes provide the Developer with a way of tracking sessions within a Web application. It is often useful in tracking sessions to know whether a session became invalid because the container timed out the session, or because a Web component within the application called the invalidate method. The distinction may be determined indirectly using listeners and the HttpSession API methods.
CHAPTER 12

Mapping Requests to Servlets

The mapping techniques described in this chapter are required for Web containers mapping client requests to servlets.¹

12.1 Use of URL Paths

Upon receipt of a client request, the Web container determines the Web application to which to forward it. The Web application selected must have the longest context path that matches the start of the request URL. The matched part of the URL is the context path when mapping to servlets.

The Web container next must locate the servlet to process the request using the path mapping procedure described below.

The path used for mapping to a servlet is the request URL from the request object minus the context path and the path parameters. The URL path mapping rules below are used in order. The first successful match is used with no further matches attempted:

1. The container will try to find an exact match of the path of the request to the path of the servlet. A successful match selects the servlet.

2. The container will recursively try to match the longest path-prefix. This is done by stepping down the path tree a directory at a time, using the '/' character as a path separator. The longest match determines the servlet selected.

3. If the last segment in the URL path contains an extension (e.g. .jsp), the servlet container will try to match a servlet that handles requests for the extension. An extension is defined as the part of the last segment after the last '.' character.

¹ Versions of this specification prior to 2.5 made use of these mapping techniques as a suggestion rather than a requirement, allowing servlet containers to each have their different schemes for mapping client requests to servlets.
4. If neither of the previous three rules result in a servlet match, the container will attempt to serve content appropriate for the resource requested. If a "default" servlet is defined for the application, it will be used. Many containers provide an implicit default servlet for serving content.

The container must use case-sensitive string comparisons for matching.

### 12.2 Specification of Mappings

In the Web application deployment descriptor, the following syntax is used to define mappings:

- A string beginning with a `/` character and ending with a `/*` suffix is used for path mapping.
- A string beginning with a `*.` prefix is used as an extension mapping.
- The empty string (""") is a special URL pattern that exactly maps to the application’s context root, i.e., requests of the form http://host:port/<context-root>/.. In this case the path info is `/'` and the servlet path and context path is empty string (""").
- A string containing only the `/` character indicates the "default" servlet of the application. In this case the servlet path is the request URI minus the context path and the path info is null.
- All other strings are used for exact matches only.

If the effective web.xml (after merging information from fragments and annotations) contains any url-patterns that are mapped to multiple servlets then the deployment must fail.

#### 12.2.1 Implicit Mappings

If the container has an internal JSP container, the `*.jsp` extension is mapped to it, allowing JSP pages to be executed on demand. This mapping is termed an *implicit* mapping. If a `*.jsp` mapping is defined by the Web application, its mapping takes precedence over the implicit mapping.

A servlet container is allowed to make other implicit mappings as long as explicit mappings take precedence. For example, an implicit mapping of `*.shtml` could be mapped to include functionality on the server.
12.2.2 Example Mapping Set

Consider the following set of mappings:

**TABLE 12-1  Example Set of Maps**

<table>
<thead>
<tr>
<th>Path Pattern</th>
<th>Servlet</th>
</tr>
</thead>
<tbody>
<tr>
<td>/foo/bar/*</td>
<td>servlet1</td>
</tr>
<tr>
<td>/baz/*</td>
<td>servlet2</td>
</tr>
<tr>
<td>/catalog</td>
<td>servlet3</td>
</tr>
<tr>
<td>*.bop</td>
<td>servlet4</td>
</tr>
</tbody>
</table>

The following behavior would result:

**TABLE 12-2  Incoming Paths Applied to Example Maps**

<table>
<thead>
<tr>
<th>Incoming Path</th>
<th>Servlet Handling Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>/foo/bar/index.html</td>
<td>servlet1</td>
</tr>
<tr>
<td>/foo/bar/index.bop</td>
<td>servlet1</td>
</tr>
<tr>
<td>/baz</td>
<td>servlet2</td>
</tr>
<tr>
<td>/baz/index.html</td>
<td>servlet2</td>
</tr>
<tr>
<td>/catalog</td>
<td>servlet3</td>
</tr>
<tr>
<td>/catalog/index.html</td>
<td>“default” servlet</td>
</tr>
<tr>
<td>/catalog/racecar.bop</td>
<td>servlet4</td>
</tr>
<tr>
<td>/index.bop</td>
<td>servlet4</td>
</tr>
</tbody>
</table>

Note that in the case of /catalog/index.html and /catalog/racecar.bop, the servlet mapped to “/catalog” is not used because the match is not exact.
Security

Web applications are created by Application Developers who give, sell, or otherwise transfer the application to a Deployer for installation into a runtime environment. Application Developers communicate the security requirements to the Deployers and the deployment system. This information may be conveyed declaratively via the application’s deployment descriptor, by using annotations within the application code, or programmatically via the `setServletSecurity` method of the `ServletRegistration` interface.

This chapter describes the Servlet container security mechanisms and interfaces and the deployment descriptor, annotation, and programmatic mechanisms for conveying the security requirements of applications.

13.1 Introduction

A web 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 web applications will have security requirements.

Although the quality assurances and implementation details may vary, servlet containers have mechanisms and infrastructure for meeting these requirements that share some of the following characteristics:

- **Authentication**: The means by which communicating entities prove to one another that they are acting on behalf of specific identities that are authorized for access.
- **Access control for resources**: The means by which interactions with resources are limited to collections of users or programs for the purpose of enforcing integrity, confidentiality, or availability constraints.
- **Data Integrity**: The means used to prove that information has not been modified by a third party while in transit.
Confidentiality or Data Privacy: The means used to ensure that information is made available only to users who are authorized to access it.

13.2 Declarative Security

Declarative security refers to the means of expressing an application’s security model or requirements, including roles, access control, and authentication requirements in a form external to the application. The deployment descriptor is the primary vehicle for declarative security in web applications.

The Deployer maps the application’s logical security requirements to a representation of the security policy that is specific to the runtime environment. At runtime, the servlet container uses the security policy representation to enforce authentication and authorization.

The security model applies to the static content part of the web application and to servlets and filters within the application that are requested by the client. The security model does not apply when a servlet uses the RequestDispatcher to invoke a static resource or servlet using a forward or an include.

13.3 Programmatic Security

Programmatic security is used by security aware applications when declarative security alone is not sufficient to express the security model of the application. Programmatic security consists of the following methods of the HttpServletRequest interface:

- authenticate
- login
- logout
- getRemoteUser
- isUserInRole
- getUserPrincipal

The login method allows an application to perform username and password collection (as an alternative to Form-Based Login).

The authenticate methods allow an application to instigate authentication of the request caller by the container from within an unconstrained request context.
The **logout** method is provided to allow an application to reset the caller identity of a request.

The **getRemoteUser** method returns the name of the remote user (that is, the caller) associated, by the container, with the request.

The **isUserInRole** method determines if the remote user (that is, the caller) associated with the request is in a specified security role.

The **getUserPrincipal** method determines the principal name of the remote user (that is, the caller) and returns a `java.security.Principal` object corresponding to the remote user. Calling the **getName** method on the Principal returned by **getUserPrincipal** returns the name of the remote user. These APIs allow servlets to make business logic decisions based on the information obtained.

If no user has been authenticated, the **getRemoteUser** method returns null, the **isUserInRole** method always returns false, and the **getUserPrincipal** method returns null.

The **isUserInRole** method takes a String argument that references an application role. For each distinct role reference used in a call to **isUserInRole**, A security-role-ref element with role-name corresponding to the role reference should be declared in the deployment descriptor. Each security-role-ref should contain a role-link sub-element whose value is the name of the application security role to which the application embedded role reference is linked. The container uses the security-role-ref with role-name equal to the role reference to determine which security-role to test the user for membership in.

For example, to map the security role reference "FOO" to the security role with role-name "manager" the syntax would be:

```xml
<security-role-ref>
    <role-name>FOO</role-name>
    <role-link>manager</role-link>
</security-role-ref>
```

In this case, if a servlet called by a user belonging to the "manager" security role were to call **isUserInRole("FOO")** the result would be true.

If no matching security-role-ref exists for a role reference used in a call to **isUserInRole**, the container must default to testing the user for membership in the security-role with role-name equal to the role reference used in the call.

The role name "**" should never be used as an argument in calling **isUserInRole**. Any call to **isUserInRole** with "**" must return false. If the role-name of the security-role to be tested is "**", and the application has NOT declared an application security-role with role-name "**", **isUserInRole** must only return true.
if the user has been authenticated; that is, only when `getRemoteUser` and `getUserPrincipal` would both return a non-null value. Otherwise, the container must check the user for membership in the application role.

The declaration of `security-role-ref` elements informs the deployer of the role references used by the application and for which mappings must be defined.

13.4 Programmatic Security Policy Configuration

This section defines the annotations and apis provided to configure the security constraints enforced by the Servlet Container.

13.4.1 `@ServletSecurity` Annotation

The `@ServletSecurity` annotation provides an alternative mechanism for defining access control constraints equivalent to those that could otherwise have been expressed declaratively via `security-constraint` elements in the portable deployment descriptor or programmatically via the `setServletSecurity` method of the `ServletRegistration` interface. Servlet containers MUST support the use of the `@ServletSecurity` annotation on classes (and subclasses thereof) that implement the `javax.servlet.Servlet` interface.

```java
package javax.servlet.annotation;
@Inherited
@Documented
@Target(value=TYPE)
@Retention(value=RUNTIME)
public @interface ServletSecurity {
    HttpConstraint value();
    HttpMethodConstraint[] httpMethodConstraints();
}
```
The @HttpConstraint annotation is used within the @ServletSecurity annotation to represent the security constraint to be applied to all HTTP protocol methods for which a corresponding @HttpMethodConstraint does NOT occur within the @ServletSecurity annotation.

For the special case where an @HttpConstraint that returns all default values\(^1\) occurs in combination with at least one @HttpMethodConstraint that returns other than all default values, the @HttpConstraint represents that no security constraint is to be applied to any of the HTTP protocol methods to which a security constraint would otherwise apply. This exception is made to ensure that such potentially non-specific uses of @HttpConstraint do not yield constraints that will explicitly establish unprotected access for such methods; given that they would not otherwise be covered by a constraint.

```java
class HttpConstraint {
    ServletSecurity.EmptyRoleSemantic value();
    java.lang.String[] rolesAllowed();
    ServletSecurity.TransportGuarantee transportGuarantee();
}
```

---

\(^1\) From methods value(), rolesAllowed(), and transportGuarantee().
httpMethodConstraint

The @HttpMethodConstraint annotation is used within the @ServletSecurity annotation to represent security constraints on specific HTTP protocol messages.

```java
package javax.servlet.annotation;

@Documented
@Retention(value=RUNTIME)
public @interface HttpMethodConstraint {
    ServletSecurity.EmptyRoleSemantic value();
    java.lang.String[] rolesAllowed();
    ServletSecurity.TransportGuarantee transportGuarantee();
}
```

### TABLE 13-2  The HttpConstraint Interface

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>The default authorization semantic that applies (only) when rolesAllowed returns an-empty array.</td>
<td>PERMIT</td>
</tr>
<tr>
<td>rolesAllowed</td>
<td>An array containing the names of the authorized roles</td>
<td>{}</td>
</tr>
<tr>
<td>transportGuarantee</td>
<td>The data protection requirements that must be satisfied by the connections on which requests arrive.</td>
<td>NONE</td>
</tr>
</tbody>
</table>

### TABLE 13-3  The HttpMethodConstraint Interface

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>The HTTP protocol method name</td>
<td></td>
</tr>
<tr>
<td>emptyRoleSemantic</td>
<td>The default authorization semantic that applies (only) when rolesAllowed returns an empty array.</td>
<td>PERMIT</td>
</tr>
<tr>
<td>rolesAllowed</td>
<td>An array containing the names of the authorized roles</td>
<td>{}</td>
</tr>
<tr>
<td>transportGuarantee</td>
<td>The data protection requirements that must be satisfied by the connections on which requests arrive.</td>
<td>NONE</td>
</tr>
</tbody>
</table>
The `@ServletSecurity` annotation may be specified on (that is, targeted to) a Servlet implementation class, and its value is inherited by subclasses according to the rules defined for the `@Inherited` meta-annotation. At most one instance of the `@ServletSecurity` annotation may occur on a Servlet implementation class, and the `@ServletSecurity` annotation MUST NOT be specified on (that is, targeted to) a Java method.

When one or more `@HttpMethodConstraint` annotations are defined within a `@ServletSecurity` annotation, each `@HttpMethodConstraint` defines the security-constraint that applies to the HTTP protocol method identified within the `@HttpMethodConstraint`. Except for the case where its `@HttpConstraint` returns all default values, and where it contains at least one `@HttpMethodConstraint` that returns other than all default values, the `@ServletSecurity` annotation defines another security-constraint that applies to all HTTP protocol methods for which a corresponding `@HttpMethodConstraint` has not been defined.

The security-constraint elements defined in the portable deployment descriptors are authoritative for all the url-patterns occurring within the constraints.

When a security-constraint in the portable deployment descriptor includes a url-pattern that is an exact match for a pattern mapped to a class annotated with `@ServletSecurity`, the annotation must have no effect on the constraints enforced by the Servlet container on the pattern.

When `metadata-complete=true` is defined for a portable deployment descriptor, the `@ServletSecurity` annotation does not apply to any of the url-patterns mapped to (any servlet mapped to) the annotated class in the deployment descriptor.

The `@ServletSecurity` annotation is not applied to the url-patterns of a ServletRegistration created using the `addServlet(String, Servlet)` method of the ServletContext interface, unless the Servlet was constructed by the `createServlet` method of the ServletContext interface.

With the exceptions listed above, when a Servlet class is annotated with `@ServletSecurity`, the annotation defines the security constraints that apply to all the url-patterns mapped to all the Servlets mapped to the class.

When a class has not been annotated with the `@ServletSecurity` annotation, the access policy that is applied to a servlet mapped from that class is established by the applicable security-constraint elements, if any, in the corresponding portable deployment descriptor, or barring any such elements, by the constraints, if any, established programmatically for the target servlet via the `setServletSecurity` method of the ServletRegistration interface.
13.4.1.1 Examples

The following examples demonstrate the use of the `ServletSecurity` annotation.

**CODE EXAMPLE 13-1** for all HTTP methods, no constraints
```java
@ServletSecurity
public class Example1 extends HttpServlet {
}
```

**CODE EXAMPLE 13-2** for all HTTP methods, no auth-constraint, confidential transport required
```java
@ServletSecurity(@HttpConstraint(transportGuarantee = TransportGuarantee.CONFIDENTIAL))
public class Example2 extends HttpServlet {
}
```

**CODE EXAMPLE 13-3** for all HTTP methods, all access denied
```java
@ServletSecurity(@HttpConstraint(EmptyRoleSemantic.DENY))
public class Example3 extends HttpServlet {
}
```

**CODE EXAMPLE 13-4** for all HTTP methods, auth-constraint requiring membership in Role R1
```java
@ServletSecurity(@HttpConstraint(rolesAllowed = "R1"))
public class Example4 extends HttpServlet {
}
```

**CODE EXAMPLE 13-5** for all HTTP methods except GET and POST, no constraints; for methods GET and POST, auth-constraint requiring membership in Role R1; for POST, confidential transport required
```java
@ServletSecurity((httpMethodConstraints = {
    @HttpMethodConstraint(value = "GET", rolesAllowed = "R1"),
    @HttpMethodConstraint(value = "POST", rolesAllowed = "R1",
    transportGuarantee = TransportGuarantee.CONFIDENTIAL)
  })
public class Example5 extends HttpServlet {
}
```

**CODE EXAMPLE 13-6** for all HTTP methods except GET auth-constraint requiring membership in Role R1; for GET, no constraints
```java
@ServletSecurity(value = @HttpConstraint(rolesAllowed = "R1"),
    httpMethodConstraints = @HttpMethodConstraint("GET"))
public class Example6 extends HttpServlet {
```
for all HTTP methods except TRACE, auth-constraint requiring membership in Role R1; for TRACE, all access denied

```java
@ServletSecurity(value = @HttpConstraint(rolesAllowed = "R1"),
    httpMethodConstraints = @HttpMethodConstraint(value="TRACE",
      emptyRoleSemantic = EmptyRoleSemantic.DENY))
public class Example7 extends HttpServlet {
}
```

### Mapping `@ServletSecurity` to `security-constraint`

This section describes the mapping of the `@ServletSecurity` annotation to its equivalent representation as `security-constraint` elements. It is provided to facilitate enforcement using the existing `security-constraint` enforcement mechanism of the container. The enforcement by Servlet containers, of the `@ServletSecurity` annotation must be equivalent in effect to enforcement, by the container, of the `security-constraint` elements resulting from the mapping defined in this section.

The `@ServletSecurity` annotation is used to define one method-independent `@HttpConstraint` followed by a list of zero or more `@HttpMethodConstraint` specifications. The method-independent constraint is applied to all HTTP methods for which no HTTP method-specific constraint has been defined.

When no `@HttpMethodConstraint` elements are included, a `@ServletSecurity` annotation corresponds to a single `security-constraint` element containing a `web-resource-collection` that contains no `http-method` elements, and thus pertains to all HTTP methods.

The following example depicts the representation of a `@ServletSecurity` annotation with no contained `@HttpMethodConstraint` annotations as a single `security-constraint` element. The `url-pattern` elements defined by the corresponding servlet (registration) would be included in the `web-resource-collection`, and the presence and value of any contained `auth-constraint` and `user-data-constraint` elements would be determined by the mapping of the `@HttpConstraint` value as defined in Section 13.4.1.3, “Mapping `@HttpConstraint` and `@HttpMethodConstraint` to XML.” on page 13-135

```xml
<security-constraint>
    <web-resource-collection>
        <url-pattern>...</url-pattern>
    </web-resource-collection>
</security-constraint>
```
<auth-constraint>
  <role-name>Role1</role-name>
</auth-constraint>
</security-constraint>

When one or more @HttpMethodConstraint elements are specified, the method-independent constraint corresponds to a single security-constraint containing a web-resource-collection that contains on http-method-omission for each of the HTTP methods named in the @HttpMethodConstraint elements. The security-constraint containing http-method-omission elements must NOT be created if the method-independent constraint returns all default values and at least one @HttpMethodConstraint does not. Each @HttpMethodConstraint corresponds to another security-constraint containing a web-resource-collection containing an http-method element naming the corresponding HTTP method.

The following example depicts the mapping of a @ServletSecurity annotation with a single contained @HttpMethodConstraint to two security-constraint elements. The url-pattern elements defined by the corresponding servlet (registration) would be included in the web-resource-collection of both constraints, and the presence and value of any contained auth-constraint and user-data-constraint elements would be determined by the mapping of the associated @HttpConstraint and @HttpMethodConstraint values as defined in Section 13.4.1.3, “Mapping @HttpConstraint and @HttpMethodConstraint to XML” on page 13-135.

**CODE EXAMPLE 13-9** mapping @ServletSecurity with contained @HttpMethodConstraint

```java
@ServletSecurity(value=@HttpConstraint(rolesAllowed = "Role1"),
    httpMethodConstraints = @HttpMethodConstraint(value = "TRACE",
    emptyRoleSemantic = EmptyRoleSemantic.DENY))
```

```xml
<security-constraint>
  <web-resource-collection>
    <url-pattern>...</url-pattern>
    <http-method-omission>TRACE</http-method-omission>
  </web-resource-collection>
  <auth-constraint>
    <role-name>Role1</role-name>
  </auth-constraint>
</security-constraint>
```
13.4.1.3 Mapping @HttpConstraint and @HttpMethodConstraint to XML.

This section describes the mapping of the @HttpConstraint and @HttpMethodConstraint annotation values (defined for use within @ServletSecurity representations) to their corresponding auth-constraint and user-data-constraint representations. These annotations share a common model for expressing the equivalent of the auth-constraint and user-data-constraint elements used within the portable deployment descriptor. That model is composed of the following 3 elements:

- emptyRoleSemantic
  
  the authorization semantic, either PERMIT or DENY, that applies when no roles are named in rolesAllowed. The default value for this element is PERMIT, and DENY is not supported in combination with a non-empty rolesAllowed list.

- rolesAllowed
  
  A list containing the names of the authorized roles. When this list is empty, its meaning depends on the value of the emptyRoleSemantic. The role name "*" has no special meaning when included in the list of allowed roles. When the special role name "**" appears in rolesAllowed, it indicates that user authentication, independent of role, is required and sufficient. The default value for this element is an empty list.

- transportGuarantee
  
  The data protection requirements, either NONE or CONFIDENTIAL, that must be satisfied by the connections on which requests arrive. This element is equivalent in meaning to a user-data-constraint containing a transport-guarantee with the corresponding value. The default value for this element is NONE.

The following examples depict the correspondence between the @HttpConstraint model described above and auth-constraint and user-data-constraint elements in web.xml.

CODE EXAMPLE 13-10  
emptyRoleSemantic=PERMIT, rolesAllowed=[], transportGuarantee=NONE

no constraints

CODE EXAMPLE 13-11  
emptyRoleSemantic=PERMIT, rolesAllowed=[], transportGuarantee=CONFIDENTIAL
CODE EXAMPLE 13-12  emptyRoleSemantic=PERMIT, rolesAllowed={Role1}, transportGuarantee=NONE
<auth-constraint>
  <security-role-name>Role1</security-role-name>
</auth-constraint>

CODE EXAMPLE 13-13  emptyRoleSemantic=PERMIT, rolesAllowed={Role1}, transportGuarantee=CONFIDENTIAL
<auth-constraint>
  <security-role-name>Role1</security-role-name>
</auth-constraint>

CODE EXAMPLE 13-14  emptyRoleSemantic=DENY, rolesAllowed=[], transportGuarantee=NONE
<auth-constraint/>

CODE EXAMPLE 13-15  emptyRoleSemantic=DENY, rolesAllowed=[], transportGuarantee=CONFIDENTIAL
<auth-constraint/>

13.4.2 setServletSecurity of ServletRegistration.Dynamic

The setServletSecurity method may be used within a ServletContextListener to define the security constraints to be applied to the mappings defined for a ServletRegistration.

Collection<String> setServletSecurity(ServletSecurityElement arg);

The javax.servlet.ServletSecurityElement argument to setServletSecurity is analogous in structure and model to the ServletSecurity interface of the @ServletSecurity annotation. As such, the
mappings defined in Section 13.4.1.2, “Mapping @ServletSecurity to security-constraint” on page 13-133, apply analogously to the mapping of a ServletSecurityElement with contained HttpConstraintElement and HttpMethodConstraintElement values, to its equivalent security-constraint representation.

The setServletSecurity method returns the (possibly empty) Set of URL patterns that are already the exact target of a security-constraint element in the portable deployment descriptor (and thus were unaffected by the call).

This method throws an IllegalStateException if the ServletContext from which the ServletRegistration was obtained has already been initialized.

When a security-constraint in the portable deployment descriptor includes a url-pattern that is an exact match for a pattern mapped by a ServletRegistration, calls to setServletSecurity on the ServletRegistration must have no effect on the constraints enforced by the Servlet container on the pattern.

With the exceptions listed above and including when the Servlet class is annotated with @ServletSecurity, when setServletSecurity is called on a ServletRegistration it establishes the security constraints that apply to the url-patterns of the registration.

13.5 Roles

A security role is a logical grouping of users defined by the Application Developer or Assembler. When the application is deployed, roles are mapped by a Deployer to principals or groups in the runtime environment.

A servlet container enforces declarative or programmatic security for the principal associated with an incoming request based on the security attributes of the principal. This may happen in either of the following ways:

1. A deployer has mapped a security role to a user group in the operational environment. The user groups to which the calling principal belongs are retrieved from its security attributes. The principal is in the security role only if the principal belongs to the user group to which the security role has been mapped by the deployer.
2. A deployer has mapped a security role to a principal name in a security policy domain. In this case, the principal name of the calling principal is retrieved from its security attributes. The principal is in the security role only if the principal name is the same as a principal name to which the security role was mapped.

# 13.6 Authentication

A web client can authenticate a user to a web server using one of the following mechanisms:

- HTTP Basic Authentication
- HTTP Digest Authentication
- HTTPS Client Authentication
- Form Based Authentication

## 13.6.1 HTTP Basic Authentication

HTTP Basic Authentication, which is based on a username and password, is the authentication mechanism defined in the HTTP/1.0 specification. A web server requests a web client to authenticate the user. As part of the request, the web server passes the **realm** (a string) in which the user is to be authenticated. The web client obtains the username and the password from the user and transmits them to the web server. The web server then authenticates the user in the specified realm.

Basic Authentication is not a secure authentication protocol. User passwords are sent in simple base64 encoding, and the target server is not authenticated. Additional protection can alleviate some of these concerns: a secure transport mechanism (HTTPS), or security at the network level (such as the IPSEC protocol or VPN strategies) is applied in some deployment scenarios.

## 13.6.2 HTTP Digest Authentication

Like HTTP Basic Authentication, HTTP Digest Authentication authenticates a user based on a username and a password. However, unlike HTTP Basic Authentication, HTTP Digest Authentication does not send user passwords over the network. In HTTP Digest authentication the client sends a one-way cryptographic hash of the password (and additional data). Although passwords are not sent on the wire, HTTP Digest authentication requires that clear text password equivalents\(^2\) be available to

\(^2\) The password equivalents can be such that they can only be used to authenticate as the user at a specific realm.
the authenticating container so that it can validate received authenticators by calculating the expected digest. Servlet containers SHOULD support HTTP_DIGEST authentication.

13.6.3 Form Based Authentication

The look and feel of the “login screen” cannot be varied using the web browser’s built-in authentication mechanisms. This specification introduces a required form based authentication mechanism which allows a Developer to control the look and feel of the login screens.

The web application deployment descriptor contains entries for a login form and error page. The login form must contain fields for entering a username and a password. These fields must be named j_username and j_password, respectively.

When a user attempts to access a protected web resource, the container checks the user’s authentication. If the user is authenticated and possesses authority to access the resource, the requested web resource is activated and a reference to it is returned. If the user is not authenticated, all of the following steps occur:

1. The login form associated with the security constraint is sent to the client and the URL path and HTTP protocol method triggering the authentication is stored by the container.

2. The user is asked to fill out the form, including the username and password fields.

3. The client posts the form back to the server.

4. The container attempts to authenticate the user using the information from the form.

5. If authentication fails, the error page is returned using either a forward or a redirect, and the status code of the response is set to 200. The error page contains information about the failure.

6. If authentication succeeds, the client is redirected to the resource using the store URL path.

7. When the redirected and authenticated request arrives at the container, the container restores the request and HTTP protocol method, and the authenticated user’s principal is checked to see if it is in an authorized role for accessing the resource.

8. If the user is authorized, the request is accepted for processing by the container.
The HTTP protocol method of the redirected request that arrives in step 7, may differ from the HTTP method of the request that triggered the authentication. As such, following the redirection of step 6, the form authenticator must process the redirected request even if authentication is not required for the HTTP method with which the request arrives. To improve the predictability of the HTTP method of the redirected request, containers should redirect (in step 6) using the 303 (SC_SEE_OTHER) status code, except where interoperability with HTTP 1.0 user agents is required; in which cases the 302 status code should be used.

When conducted over an unprotected transport, Form Based Authentication is subject to some of the same vulnerabilities as Basic Authentication.

When the request that is triggering authentication arrives over a secure transport, or the login page is subject to a user-data-constraint of CONFIDENTIAL, the login page must be returned to the user, and submitted to the container over a secure transport.

The login page should be subject to a user-data-constraint of CONFIDENTIAL, and a user-data-constraint of CONFIDENTIAL should be included in every security-constraint that contains a requirement for authentication.

The login method of the HttpServletRequest interface provides an alternative means for an application to control the look and feel of it’s login screens.

### 13.6.3.1 Login Form Notes

Form based login and URL based session tracking can be problematic to implement. Form based login should be used only when sessions are being maintained by cookies or by SSL session information.

In order for the authentication to proceed appropriately, the action of the login form must always be `j_security_check`. This restriction is made so that the login form will work no matter which resource it is for, and to avoid requiring the server to specify the action field of the outbound form. The login form should specify `autocomplete="off"` on the password form field.

Here is an example showing how the form should be coded into the HTML page:

```html
<form method="POST" action="j_security_check">
  <input type="text" name="j_username">
  <input type="password" name="j_password" autocomplete="off">
</form>
```

If the form based login is invoked because of an HTTP request, the original request parameters must be preserved by the container for use if, on successful authentication, it redirects the call to the requested resource.
If the user is authenticated using form login and has created an HTTP session, the timeout or invalidation of that session leads to the user being logged out in the sense that subsequent requests must cause the user to be re-authenticated. The scope of the logout is the same as that of the authentication: for example, if the container supports single signon, such as Java EE technology compliant web containers, the user would need to reauthenticate with any of the web applications hosted on the web container.

13.6.4 HTTPS Client Authentication

End user authentication using HTTPS (HTTP over SSL) is a strong authentication mechanism. This mechanism requires the client to possess a Public Key Certificate (PKC). Currently, PKCs are useful in e-commerce applications and also for a single-signon from within the browser.

13.6.5 Additional Container Authentication Mechanisms

Servlet containers should provide public interfaces that may be used to integrate and configure additional HTTP message layer authentication mechanisms for use by the container on behalf of deployed applications. These interfaces should be offered for use by parties other than the container vendor (including application developers, system administrators, and system integrators).

To facilitate portable implementation and integration of additional container authentication mechanisms, it is recommended that all Servlet containers implement the Servlet Container Profile of The Java™ Authentication SPI for Containers (i.e., JSR 196). The SPI is available for download at: http://www.jcp.org/en/jsr/detail?id=196

13.7 Server Tracking of Authentication Information

As the underlying security identities (such as users and groups) to which roles are mapped in a runtime environment are environment specific rather than application specific, it is desirable to:
1. Make login mechanisms and policies a property of the environment the web application is deployed in.

2. Be able to use the same authentication information to represent a principal to all applications deployed in the same container, and

3. Require re-authentication of users only when a security policy domain boundary has been crossed.

Therefore, a servlet container is required to track authentication information at the container level (rather than at the web application level). This allows users authenticated for one web application to access other resources managed by the container permitted to the same security identity.

13.8 Specifying Security Constraints

Security constraints are a declarative way of defining the protection of web content. A security constraint associates authorization and or user data constraints with HTTP operations on web resources. A security constraint, represented as a security-constraint in a deployment descriptor, consists of the following elements:

- web resource collection (web-resource-collection in deployment descriptor)
- authorization constraint (auth-constraint in deployment descriptor)
- user data constraint (user-data-constraint in deployment descriptor)

The HTTP operations and web resources to which a security constraint applies (i.e. the constrained requests) are identified by one or more web resource collections. A web resource collection consists of the following elements:

- URL patterns (url-pattern in deployment descriptor)
- HTTP methods (http-method or http-method-omission elements in the deployment descriptor)

An authorization constraint establishes a requirement for authentication and names the authorization roles permitted to perform the constrained requests. A user must be a member of at least one of the named roles to be permitted to perform the constrained requests. The special role name "*" is a shorthand for all role names defined in the deployment descriptor. The special role name "**" is a shorthand for any authenticated user independent of role. When the special role name "**" appears in an authorization constraint, it indicates that any authenticated user, independent of role, is authorized to perform the constrained requests. An authorization constraint that names no roles indicates that access to the constrained requests must not be permitted under any circumstances. An authorization constraint consists of the following element:

- role name (role-name in deployment descriptor)
A user data constraint establishes a requirement that the constrained requests be received over a protected transport layer connection. The strength of the required protection is defined by the value of the transport guarantee. A transport guarantee of INTEGRAL is used to establish a requirement for content integrity and a transport guarantee of CONFIDENTIAL is used to establish a requirement for confidentiality. The transport guarantee of “NONE” indicates that the container must accept the constrained requests when received on any connection including an unprotected one. Containers may impose a confidential transport guarantee in response to the INTEGRAL value. A user data constraint consists of the following element:

- transport guarantee (transport-guarantee in deployment descriptor)

If no authorization constraint applies to a request, the container must accept the request without requiring user authentication. If no user data constraint applies to a request, the container must accept the request when received over any connection including an unprotected one.

### 13.8.1 Combining Constraints

For the purpose of combining constraints, an HTTP method is said to occur within a web-resource-collection when no HTTP methods are named in the collection, or the collection specifically names the HTTP method in a contained http-method element, or the collection contains one or more http-method-omission elements, none of which names the HTTP method.

When a url-pattern and HTTP method pair occurs in combination (i.e., within a web-resource-collection) in multiple security constraints, the constraints (on the pattern and method) are defined by combining the individual constraints. The rules for combining constraints in which the same pattern and method occur are as follows:

The combination of authorization constraints that name roles or that imply roles via the name “*” shall yield the union of the role names in the individual constraints as permitted roles. An authorization constraint that names the role “*” shall combine with authorization constraints that name or imply roles to permit any authenticated user independent of role. A security constraint that does not contain an authorization constraint shall combine with authorization constraints that name or imply roles to allow unauthenticated access. The special case of an authorization constraint that names no roles shall combine with any other constraints to override their affects and cause access to be precluded.

The combination of user-data-constraints that apply to a common url-pattern and http-method shall yield the union of connection types accepted by the individual constraints as acceptable connection types. A security constraint that does not contain a user-data-constraint shall combine with other user-data-constraint to cause the unprotected connection type to be an accepted connection type.
13.8.2 Example

The following example illustrates the combination of constraints and their translation into a table of applicable constraints. Suppose that a deployment descriptor contained the following security constraints.

```
<security-constraint>

  <web-resource-collection>
    <web-resource-name>precluded methods</web-resource-name>
    <url-pattern>/</url-pattern>
    <url-pattern>/acme/wholesale/*</url-pattern>
    <url-pattern>/acme/retail/*</url-pattern>
    <http-method-omission>GET</http-method-omission>
    <http-method-omission>POST</http-method-omission>
  </web-resource-collection>

  <auth-constraint/>

</security-constraint>

<security-constraint>

  <web-resource-collection>
    <web-resource-name>wholesale</web-resource-name>
    <url-pattern>/acme/wholesale/*</url-pattern>
    <http-method>GET</http-method>
    <http-method>PUT</http-method>
  </web-resource-collection>

  <auth-constraint>
    <role-name>SALESCLERK</role-name>
  </auth-constraint>

</security-constraint>
```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>wholesale 2</web-resource-name>
    <url-pattern>/acme/wholesale/*</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>CONTRACTOR</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>

<security-constraint>
  <web-resource-collection>
    <web-resource-name>retail</web-resource-name>
    <url-pattern>/acme/retail/*</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>CONTRACTOR</role-name>
    <role-name>HOMEOWNER</role-name>
  </auth-constraint>
</security-constraint>
The translation of this hypothetical deployment descriptor would yield the constraints defined in TABLE 13-4.

**TABLE 13-4  Security Constraint Table**

<table>
<thead>
<tr>
<th>url-pattern</th>
<th>http-method</th>
<th>permitted roles</th>
<th>supported connection types</th>
</tr>
</thead>
<tbody>
<tr>
<td>/*</td>
<td>all methods</td>
<td>access</td>
<td>not constrained</td>
</tr>
<tr>
<td></td>
<td>except GET, POST</td>
<td>precluded</td>
<td></td>
</tr>
<tr>
<td>/acme/wholesale/*</td>
<td>all methods</td>
<td>access</td>
<td>not constrained</td>
</tr>
<tr>
<td></td>
<td>except GET, POST</td>
<td>precluded</td>
<td></td>
</tr>
<tr>
<td>/acme/wholesale/*</td>
<td>GET</td>
<td>CONTRACTOR</td>
<td>not constrained</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SALESCLERK</td>
<td></td>
</tr>
<tr>
<td>/acme/wholesale/*</td>
<td>POST</td>
<td>CONTRACTOR</td>
<td>CONFIDENTIAL</td>
</tr>
<tr>
<td>/acme/retail/*</td>
<td>all methods</td>
<td>access</td>
<td>not constrained</td>
</tr>
<tr>
<td></td>
<td>except GET, POST</td>
<td>precluded</td>
<td></td>
</tr>
<tr>
<td>/acme/retail/*</td>
<td>GET</td>
<td>CONTRACTOR</td>
<td>not constrained</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HOMEOWNER</td>
<td></td>
</tr>
<tr>
<td>/acme/retail/*</td>
<td>POST</td>
<td>CONTRACTOR</td>
<td>not constrained</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HOMEOWNER</td>
<td></td>
</tr>
</tbody>
</table>

### 13.8.3 Processing Requests

When a Servlet container receives a request, it shall use the algorithm described in “Use of URL Paths” on page 121 to select the constraints (if any) defined on the `url-pattern` that is the best match to the request URI. If no constraints are selected, the container shall accept the request. Otherwise the container shall determine if the HTTP method of the request is constrained at the selected pattern. If it is not, the request shall be accepted. Otherwise, the request must satisfy the constraints that apply to the HTTP method at the `url-pattern`. Both of the following rules must be satisfied for the request to be accepted and dispatched to the associated servlet.
1. The characteristics of the connection on which the request was received must satisfy at least one of the supported connection types defined by the constraints. If this rule is not satisfied, the container shall reject the request and redirect it to the HTTPS port.3

2. The authentication characteristics of the request must satisfy any authentication and role requirements defined by the constraints. If this rule is not satisfied because access has been precluded (by an authorization constraint naming no roles), the request shall be rejected as forbidden and a 403 (SC_FORBIDDEN) status code shall be returned to the user. If access is restricted to permitted roles and the request has not been authenticated, the request shall be rejected as unauthorized and a 401 (SC_UNAUTHORIZED) status code shall be returned to cause authentication. If access is restricted to permitted roles and the authentication identity of the request is not a member of any of these roles, the request shall be rejected as forbidden and a 403 (SC_FORBIDDEN) status code shall be returned to the user.

13.8.4 Uncovered HTTP Protocol Methods

The security-constraint schema provides the ability to enumerate (including by omission) the HTTP protocol methods to which the protection requirements defined in a security-constraint apply. When HTTP methods are enumerated within a security-constraint, the protections defined by the constraint apply only to the methods established by the enumeration. We refer to the HTTP methods that are not established by the enumeration as "uncovered" HTTP methods. Uncovered HTTP methods are NOT protected at all request URLs for which a url-pattern of the security-constraint is a best match.

When HTTP methods are not enumerated within a security-constraint, the protections defined by the constraint apply to the complete set of HTTP (extension) methods. In that case, there are no uncovered HTTP methods at all request URLs for which a url-pattern of the security-constraint is a best match.

The examples that follow depict the three ways in which HTTP protocol methods may be left uncovered. The determination of whether methods are uncovered is made after all the constraints that apply to a url-pattern have been combined as described in Section 13.8.1, "Combining Constraints" on page 13-143.

3. As an optimization, a container should reject the request as forbidden and return a 403 (SC_FORBIDDEN) status code if it knows that access will ultimately be precluded (by an authorization constraint naming no roles).
1. A security-constraint names one or more HTTP methods in http-method elements. All HTTP methods other than those named in the constraint are uncovered.

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>wholesale</web-resource-name>
    <url-pattern>/acme/wholesale/*</url-pattern>
    <http-method>GET</http-method>
  </web-resource-collection>

  <auth-constraint>
    <role-name>SALESCLERK</role-name>
  </auth-constraint>
</security-constraint>
```

All HTTP Methods except GET are uncovered.

2. A security-constraint names one or more HTTP methods in http-method-omission elements. All HTTP methods named in the constraint are uncovered.

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>wholesale</web-resource-name>
    <url-pattern>/acme/wholesale/*</url-pattern>
    <http-method-omission>GET</http-method-omission>
  </web-resource-collection>

  <auth-constraint/>
</security-constraint>
```

GET is uncovered. All other methods are covered by the excluding auth-constraint.

3. A @ServletSecurity annotation includes an @HttpConstraint that returns all default values and it also includes at least one @HttpMethodConstraint that returns other than all default values. All HTTP methods other than those named in an @HTTPMethodConstraint are uncovered by the annotation. This case is
analogous to case 1, and equivalent use of the setServletSecurity method of the ServletRegistration interface will also produce an analogous result.

```java
@ServletSecurity((httpMethodConstraints = {
    @HttpMethodConstraint(value = "GET", rolesAllowed = "R1"),
    @HttpMethodConstraint(value = "POST", rolesAllowed = "R1",
                          transportGuarantee = TransportGuarantee.CONFIDENTIAL)
  })
public class Example5 extends HttpServlet {
}
```

All HTTP Methods except GET and POST are uncovered.

### 13.8.4.1 Rules for Security Constraint Configuration

Objective: Make sure all HTTP methods at all constrained URL patterns have the intended security protections (that is, are covered).

1. Do not name HTTP methods in constraints; in which case, the security protections defined for the URL patterns will apply to all HTTP methods.

2. If you can’t follow rule #1, add the `<deny-uncovered-http-methods>` and declare (using the `<http-method>` element, or equivalent annotation) all the HTTP methods (with security protections) that are to be allowed at the constrained URL patterns.

3. If you can’t follow rule #2, declare constraints to cover all HTTP methods at each constrained URL pattern. Use the `<http-method-omission>` element or the HttpMethodConstraint annotation to represent the set of all HTTP methods other than those named by `<http-method>` or HttpMethodConstraint. When using annotations, use the HttpConstraint annotation to define the security semantic to be applied to all other HTTP methods and configure EmptyRoleSemantic=DENY to cause all other HTTP methods to be denied.

### 13.8.4.2 Handling Uncovered HTTP Methods

During application deployment, the container must inform the deployer of any uncovered HTTP methods present in the application security constraint configuration resulting from the combination of the constraints defined for the application. The provided information must identify the uncovered HTTP protocol methods, and the corresponding URL patterns at which the HTTP methods are uncovered. The requirement to notify the deployer may be satisfied by logging the required information.
When the `deny-uncovered-http-methods` flag is set in the web.xml of an application, the container must deny any HTTP protocol method when it is used with a request URL for which the HTTP method is uncovered at the combined security constraint that applies to the url-pattern that is the best match for the request URL. The denied request shall be rejected as forbidden and a 403 (SC_FORBIDDEN) status code shall be returned.

To cause uncovered HTTP methods to be denied, the deployment system should establish additional excluding auth-constraints, to cover these HTTP methods at the constrained url-patterns at which the HTTP methods are uncovered.

When an application’s security configuration contains no uncovered methods, the `deny-uncovered-http-methods` flag must have no effect on the effective security configuration of the application.

Applying the `deny-uncovered-http-methods` flag to an application whose security configuration contains uncovered methods, may, in some cases, deny access to resources that must be accessible in order for the application to function. In such cases, the security configuration of the application should be completed such that all uncovered methods are covered by an appropriate constraint configuration.

Application Developers should define security constraint configurations that leave no HTTP methods uncovered, and they should set the `deny-uncovered-http-methods` flag to ensure that their applications do not become dependent on being accessible via uncovered methods.

A Servlet container may provide a configurable option to select whether the default behavior for uncovered methods is ALLOW or DENY. This option may be configured on a per-application granularity or larger. Note that setting this default to DENY may cause some applications to fail.

### 13.9 Default Policies

By default, authentication is not needed to access resources. Authentication is required when the security constraints (if any) that contain the `url-pattern` that is the best match for the request URI combine to impose an auth-constraint (naming roles) on the HTTP method of the request. Similarly, a protected transport is not required unless the security constraints that apply to the request combine to impose a user-data-constraint (with a protected transport-guarantee) on the HTTP method of the request.
13.10 Login and Logout

The container establishes the caller identity of a request prior to dispatching the request to the servlet engine. The caller identity remains unchanged throughout the processing of the request or until the application successfully calls authenticate, login or logout on the request. For asynchronous requests, the caller identity established at the initial dispatch remains unchanged until the processing of the overall request completes, or the application successfully calls authenticate, login or logout on the request.

Being logged into an application during the processing of a request, corresponds precisely to there being a valid non-null caller identity associated with the request as may be determined by calling getRemoteUser or getUserPrincipal on the request. A null return value from either of these methods indicates that the caller is not logged into the application with respect to the processing of the request.

Containers may create HTTP Session objects to track login state. If a developer creates a session while a user is not authenticated, and the container then authenticates the user, the session visible to developer code after login must be the same session object that was created prior to login occurring so that there is no loss of session information.
Deployment Descriptor

This chapter specifies the Java™ Servlet Specification requirements for Web container support of deployment descriptors. The deployment descriptor conveys the elements and configuration information of a Web application between Application Developers, Application Assemblers, and Deployers.

For Java Servlets v.2.4 and greater, the deployment descriptor is defined in terms of an XML schema document.

For backwards compatibility of applications written to the 2.2 version of the API, Web containers are also required to support the 2.2 version of the deployment descriptor. For backwards compatibility of applications written to the 2.3 version of the API, Web containers are also required to support the 2.3 version of the deployment descriptor. The 2.2 version is available at http://java.sun.com/j2ee/dtds/web-app_2_2.dtd and 2.3 version is available at http://java.sun.com/dtd/web-app_2_3.dtd.

14.1 Deployment Descriptor Elements

The following types of configuration and deployment information are required to be supported in the Web application deployment descriptor for all servlet containers:

- ServletContext Init Parameters
- Session Configuration
- Servlet Declaration
- Servlet Mappings
- Application Lifecycle Listener classes
- Filter Definitions and Filter Mappings
- MIME Type Mappings
- Welcome File list
- Error Pages
- Locale and Encoding Mappings
14.2 Rules for Processing the Deployment Descriptor

This section lists some general rules that Web containers and developers must note concerning the processing of the deployment descriptor for a Web application.

- Web containers must remove all leading and trailing whitespace, which is defined as “S(white space)” in XML 1.0 (http://www.w3.org/TR/2000/WD-xml-2e-20000814), for the element content of the text nodes of a deployment descriptor.
- The deployment descriptor must be valid against the schema. Web containers and tools that manipulate Web applications have a wide range of options for checking the validity of a WAR. This includes checking the validity of the deployment descriptor document held within.

Additionally, it is recommended that Web containers and tools that manipulate Web applications 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 Web 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.

- The sub elements under web-app can be in an arbitrary order in this version of the specification. Because of the restriction of XML Schema, the multiplicity of the elements distributable, session-config, welcome-file-list, jsp-config, login-config, and locale-encoding-mapping-list was changed from “optional” to “0 or more”. The containers must inform the developer with a descriptive error message when the deployment descriptor contains more than one element of session-config, jsp-config, and login-config. The container must concatenate the items in welcome-file-list and locale-encoding-mapping-list when there are multiple occurrences. The multiple occurrence of distributable must be treated exactly in the same way as the single occurrence of distributable.
- URI paths specified in the deployment descriptor are assumed to be in URL-decoded form. The containers must inform the developer with a descriptive error message when URL contains CR (#xD) or LF (#xA). The containers must preserve all other characters including whitespace in URL.
Containers must attempt to canonicalize paths in the deployment descriptor. For example, paths of the form /a/../b must be interpreted as /b. Paths beginning or resolving to paths that begin with ../ are not valid paths in the deployment descriptor.

URI paths referring to a resource relative to the root of the WAR, or a path mapping relative to the root of the WAR, unless otherwise specified, should begin with a leading /.

In elements whose value is an enumerated type, the value is case sensitive.

14.3 Deployment Descriptor

The deployment descriptor for this revision of the specification is available at http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd

14.4 Deployment Descriptor Diagram

This section illustrates the elements in deployment descriptor. Attributes are not shown in the diagrams. See Deployment Descriptor Schema for the detailed information.

1. web-app Element
The `web-app` element is the root deployment descriptor for a Web application. This element contains the following elements. This element has a required attribute `version` to specify to which version of the schema the deployment descriptor conforms. All sub elements under this element can be in an arbitrary order.

**FIGURE 14-1** web-app Element Structure
2. description Element

The description element is to provide a text describing the parent element. This element occurs not only under the web-app element but also under other multiple elements. It has an optional attribute xml:lang to indicate which language is used in the description. The default value of this attribute is English (“en”).

3. display-name Element

The display-name contains a short name that is intended to be displayed by tools. The display name need not to be unique. This element has an optional attribute xml:lang to specify the language.

4. icon Element

The icon contains small-icon and large-icon elements that specify the file names for small and large GIF or JPEG icon images used to represent the parent element in a GUI tool.

5. distributable Element

The distributable indicates that this Web application is programmed appropriately to be deployed into a distributed servlet container.

6. context-param Element

The context-param contains the declaration of a Web application’s servlet context initialization parameters.

7. filter Element

The filter declares a filter in the Web application. The filter is mapped to either a servlet or a URL pattern in the filter-mapping element, using the filter-name value to reference. Filters can access the initialization parameters declared in the deployment descriptor at runtime via the FilterConfig interface. The filter-name element is the logical name of the filter. It must be unique within the Web application. The element content of filter-name element must not be empty. The filter-class is the fully qualified class name of the filter. The
The `init-param` element contains name-value pairs as initialization parameters of this filter. The optional `async-supported` element, when specified, indicates that the filter supports asynchronous request processing.

**FIGURE 14-2** filter Element Structure

8. `filter-mapping` Element
The **filter-mapping** is used by the container to decide which filters to apply to a request in what order. The value of the **filter-name** must be one of the filter declarations in the deployment descriptor. The matching request can be specified either **url-pattern** or **servlet-name**.

**FIGURE 14-3** filter-mapping Element Structure

9. **listener Element**

The **listener** indicates the deployment properties for an application listener bean. The sub-element **listener-class** declares that a class in the application must be registered as a Web application listener bean. The value is the fully qualified classname of the listener class.

**FIGURE 14-4** listener Element Structure
10. servlet Element

The **servlet** is used to declare a servlet. It contains the declarative data of a servlet. The **jsp-file** element contains the full path to a JSP file within the web application beginning with a “/”. If a jsp-file is specified and the **load-on-startup** element is present, then the JSP should be precompiled and loaded. The **servlet-name** element contains the canonical name of the servlet. Each servlet name is unique within the web application. The element content of servlet-name must not be empty. The **servlet-class** contains the fully qualified class name of the servlet. The **run-as** element specifies the identity to be used for the execution of a component. It contains an optional description, and the name of a security role specified by the **role-name** element. The element **load-on-startup** indicates that this servlet should be loaded (instantiated and have its init() called) on the startup of the Web application. The element content of this element must be an integer indicating the order in which the servlet should be loaded. If the value is a negative integer, or the element is not present, the container is free to load the servlet whenever it chooses. If the value is a positive integer or 0, the container must load and initialize the servlet as the application is deployed. The container must guarantee that servlets marked with lower integers are loaded before servlets marked with higher integers. The container may choose the order of loading of servlets with the same load-on-startup value. The **security-role-ref** element declares the security role reference in a component’s or in a deployment component’s code. It consists of an optional description, the security role name used in the code (**role-name**), and an optional link to a security role (**role-link**). If the security role is not specified, the deployer must choose an appropriate security role. The optional **async-supported** element, when specified, indicates that the Servlet can support asynchronous request processing. If a servlet supports fileupload functionality and processing of mime-multipart requests, the configuration for the same can be provided via the **multipart-config** element in the descriptor. The
multipart-config element can be used to specify the location where the files can be stored, maximum size of the file being uploaded, maximum request size and the size threshold after which the file will be written to the disk.

**FIGURE 14-5** servlet Element Structure
11. servlet-mapping Element

The servlet-mapping defines a mapping between a servlet and a URL pattern. **FIGURE 14-6** servlet-mapping Element Structure

![Diagram of servlet-mapping Element Structure]

12. session-config Element

The session-config defines the session parameters for this Web application. The sub-element session-timeout defines the default session timeout interval for all sessions created in this Web application. The specified timeout must be expressed in a whole number of minutes. If the timeout is 0 or less, the container ensures the default behavior of sessions is never to time out. If this element is not specified, the container must set its default timeout period. **FIGURE 14-7** session-config Element Structure

![Diagram of session-config Element Structure]

13. mime-mapping Element
The `mime-mapping` defines a mapping between an extension and a mime type. The `extension` element contains a string describing an extension, such as “txt”.

**FIGURE 14-8** mime-mapping Element Structure

14. `welcome-file-list` Element

The `welcome-file-list` contains an ordered list of welcome files. The sub-element `welcome-file` contains a file name to use as a default welcome file, such as `index.html`.

**FIGURE 14-9** welcome-file-list Element Structure

15. `error-page` Element

The `error-page` contains a mapping between an error code or an exception type to the path of a resource in the Web application. However, the `error-code` or the `exception-type` element can be omitted to specify a default error page. The sub-element `exception-type` contains a fully qualified class name of a Java
exception type. The sub-element location element contains the location of the resource in the web application relative to the root of the web application. The value of the location must have a leading ‘/’.

FIGURE 14-10 error-page Element Structure

16. jsp-config Element

The jsp-config is used to provide global configuration information for the JSP files in a web application. It has two sub-elements, taglib and jsp-property-group. The taglib element can be used to provide information on a tag library that is used by a JSP page within the Web application. See JavaServer Pages specification version 2.1 for detail.

FIGURE 14-11 jsp-config Element Structure
17. security-constraint Element

The security-constraint is used to associate security constraints with one or more web resource collections. The sub-element web-resource-collection identifies a subset of the resources and HTTP methods on those resources within a Web application to which a security constraint applies. The auth-constraint indicates the user roles that should be permitted access to this resource collection. The role-name used here must either correspond to the role-name of one of the security-role elements defined for this Web application, or be the specially reserved role-name "*" that is a compact syntax for indicating all roles in the web application. If both "*" and role names appear, the container interprets this as all roles. If no roles are defined, no user is allowed access to the portion of the Web application described by the containing security-constraint. The container matches role names case sensitively when determining access. The user-data-constraint indicates how data communicated between the client
and container should be protected by the sub-element transport-guarantee. The legal values of the transport-guarantee is either one of NONE, INTEGRAL, or CONFIDENTIAL.

**FIGURE 14-12** security-constraint Element Structure

18. login-config Element

The login-config is used to configure the authentication method that should be used, the realm name that should be used for this application, and the attributes that are needed by the form login mechanism. The sub-element auth-method configures the authentication mechanism for the Web application. The element content must be either BASIC, DIGEST, FORM, CLIENT-CERT, or a
vendor-specific authentication scheme. The realm-name indicates the realm name to use for the authentication scheme chosen for the Web application. The form-login-config specifies the login and error pages that should be used in FORM based login. If FORM based login is not used, these elements are ignored.

**FIGURE 14-13** login-config Element Structure

19. security-role Element

The security-role defines a security role. The sub-element role-name designates the name of the security role. The name must conform to the lexical rules for NMTOKEN.

**FIGURE 14-14** security-role Element Structure

20. env-entry Element

The env-entry declares an application’s environment entry. The sub-element env-entry-name contains the name of a deployment component’s environment entry. The name is a JNDI name relative to the java:comp/env context. The name must be unique within a deployment component. The env-entry-type contains the fully-qualified Java type of the environment entry value that is
expected by the application’s code. The sub-element `env-entry-value` designates the value of a deployment component’s environment entry. The value must be a `String` that is valid for the constructor of the specified type that takes a single `String` as a parameter, or a single character for `java.lang.Character`. The optional `injection-target` element is used to define the injection of the named resource into fields or JavaBeans properties. An `injection-target` specifies a class and a name within that class into which a resource should be injected. The `injection-target-class` specifies the fully qualified class name that is the target of the injection. The `injection-target-name` specifies the target within the specified class. The target is first looked for as a JavaBean property name. If not found, the target is looked for as a field name. The specified resource will be injected into the target during initialization of the class by either calling the set method for the target property or by setting a value into the name
filed. If an injection-target is specified for the environment entry, the env-entry-type may be omitted or MUST match the type of the injection target. If no injection-target is specified, the env-entry-type is required.

FIGURE 14-15 env-entry Element Structure

21. ejb-ref Element

The ejb-ref declares the reference to an enterprise bean’s home. The ejb-ref-name specifies the name used in the code of the deployment component that is referencing the enterprise bean. The ejb-ref-type is the expected type of the referenced enterprise bean, which is either Entity or Session. The home defines the fully qualified name of the referenced enterprise bean’s home interface. The remote defines the fully qualified name of the referenced enterprise bean’s remote interface. The ejb-link specifies that an EJB reference is linked to the
enterprise bean. See Java Platform, Enterprise Edition, version 6 for more detail. In addition to these elements, the injection-target element can be used to define injection of the named enterprise bean into a component field or property.

**FIGURE 14-16 ejb-ref Element Structure**

22. ejb-local-ref Element
The `ejb-local-ref` declares the reference to the enterprise bean’s local home. The `local-home` defines the fully qualified name of the enterprise bean’s local home interface. The `local` defines the fully qualified name of the enterprise bean’s local interface.

**FIGURE 14-17 ejb-local-ref Element Structure**

---

23. service-ref Element

The `service-ref` declares the reference to a Web service. The `service-ref-name` declares the logical name that the components in the module use to look up the Web service. It is recommended that all service reference names start with `/service/`. The `service-interface` defines the fully qualified class name of the JAX-WS Service interface that the client depends on. In most cases, the value will be `javax.xml.rpc.Service`. A JAX-WS generated Service Interface class may also be specified. The `wsdl-file` element contains the URI location of a WSDL file. The location is relative to the root of the module. The `jaxrpc-mapping-file` contains the name of a file that describes the JAX-WS mapping between the Java interfaces used by the application and the WSDL description in the `wsdl-file`. The file name is a relative path within the module file. The service-
qname element declares the specific WSDL service element that is being referred to. It is not specified if no wsdl-file is declared. The port-component-ref element declares a client dependency on the container for resolving a Service Endpoint Interface to a WSDL port. It optionally associates the Service Endpoint Interface with a particular port-component. This is only used by the container for a Service.getPort(Class) method call. The handler element declares the handler for a port-component. Handlers can access the init-param name-value pairs using the HandlerInfo interface. If port-name is not specified, the handler is assumed to be associated with all ports of the service. See JSR-109 Specification [http://www.jcp.org/en/jsr/detail?id=109] for detail. The container that is not a part of a Java EE implementation is not required to support this element.

FIGURE 14-18 service-ref Element Structure
24. resource-ref Element

The resource-ref contains the declaration of a deployment component’s reference to the external resource. The res-ref-name specifies the name of a resource manager connection factory reference. The name is a JNDI name relative to the java:comp/env context. The name must be unique within a deployment file. The res-type element specifies the type of the data source. The type is the fully qualified Java language class or the interface expected to be implemented by the data source. The res-auth specifies whether the deployment component code signs on programmatically to the resource manager, or whether the container will sign on to the resource manager on behalf of the deployment component. In the latter case, the container uses the information supplied by the deployer. The res-sharing-scope specifies whether connections obtained through the given resource manager connection factory reference can be shared. The value, if specified, must be either Shareable or Unshareable. The optional injection-target element is used to define injection of the named resource into fields or JavaBeans properties.

FIGURE 14-19 resource-ref Element Structure
25. resource-env-ref Element

The `resource-env-ref` contains the deployment component’s reference to the administered object associated with a resource in the deployment component’s environment. The `resource-env-ref-name` specifies the name of the resource environment reference. The value is the environment entry name used in the deployment component code and is a JNDI name relative to the `java:comp/env` context and must be unique within the deployment component. The `resource-env-ref-type` specifies the type of the resource environment reference. It is the fully qualified name of a Java language class or the interface. The optional `injection-target` element is used to define injection of the named resource into fields or JavaBeans properties. The `resource-env-ref-type` MUST be supplied unless an injection target is specified, in which case the type of the target is used. If both are specified, the type MUST be assignment compatible with the type of the injection target.

**FIGURE 14-20** resource-env-ref Element Structure

26. message-destination-ref Element

The `message-destination-ref` element contains a declaration of deployment component’s reference to a message destination associated with a resource in deployment component’s environment. The `message-destination-ref-name` element specifies the name of a message destination reference; its value is the environment entry name used in deployment component code. The name is a
JNDI name relative to the java:comp/env context and must be unique within an ejb-jar for enterprise beans or a deployment file for others. The message-destination-type specifies the type of the destination. The type is specified by the Java interface expected to be implemented by the destination. The message-destination-usage specifies the use of the message destination indicated by the reference. The value indicates whether messages are consumed from the message destination, produced for the destination, or both. The Assembler makes use of this information in linking producers of a destination with its consumers. The message-destination-link links a message destination reference or message-driven bean to a message destination. The Assembler sets the value to reflect the flow of messages between producers and consumers in the application. The value must be the message-destination-name of a message destination in the same deployment file or in another deployment file in the same Java EE application unit. Alternatively, the value may be composed of a path name specifying a deployment file containing the referenced message destination with the message-destination-name of the destination appended and separated from the path name by "#". The path name is relative to the deployment file containing deployment component that is referencing the message destination. This allows multiple message destinations with the same name to be uniquely identified. The optional injection-target element is used to define injection of the named resource into fields or JavaBeans properties. The message-destination-type MUST be specified unless an injection target is specified, in which case the type of the target is used. If both are specified, the type MUST be assignment compatible with the type of the injection target.
Example:

```
<message-destination-ref>
  <message-destination-ref-name>
    jms/StockQueue
  </message-destination-ref-name>
  <message-destination-type>
    javax.jms.Queue
  </message-destination-type>
  <message-destination-usage>
    Consumes
  </message-destination-usage>
  <message-destination-link>
    CorporateStocks
  </message-destination-link>
</message-destination-ref>
```

**FIGURE 14-21** message-destination-ref Element Structure
27. message-destination Element

The message-destination specifies a message destination. The logical destination described by this element is mapped to a physical destination by the deployer. The message-destination-name element specifies a name for a message destination. This name must be unique among the names of message destinations within the deployment file.

Example:

```xml
<message-destination>
  <message-destination-name>
    CorporateStocks
  </message-destination-name>
</message-destination>
```

**FIGURE 14-22 message-destination Element Structure**

28. locale-encoding-mapping-list Element

The locale-encoding-mapping-list contains the mapping between the locale and the encoding, specified by the sub-element locale-encoding-mapping.
Example:

```xml
<locale-encoding-mapping-list>
  <locale-encoding-mapping>
    <locale>ja</locale>
    <encoding>Shift_JIS</encoding>
  </locale-encoding-mapping>
</locale-encoding-mapping-list>
```

**FIGURE 14-23** locale-encoding-mapping-list Element Structure

14.5 Examples

The following examples illustrate the usage of the definitions listed in the deployment descriptor schema.
14.5.1 A Basic Example

CODE EXAMPLE 14-1 Basic Deployment Descriptor Example

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                            http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd"
         version="2.5">
  <display-name>A Simple Application</display-name>
  <context-param>
    <param-name>Webmaster</param-name>
    <param-value>webmaster@mycorp.com</param-value>
  </context-param>
  <servlet>
    <servlet-name>catalog</servlet-name>
    <servlet-class>com.mycorp.CatalogServlet</servlet-class>
    <init-param>
      <param-name>catalog</param-name>
      <param-value>Spring</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>catalog</servlet-name>
    <url-pattern>/catalog/*</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>30</session-timeout>
  </session-config>
  <mime-mapping>
    <extension>pdf</extension>
    <mime-type>application/pdf</mime-type>
  </mime-mapping>
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
  </welcome-file-list>
  <error-page>
    <error-code>404</error-code>
    <location>/404.html</location>
  </error-page>
</web-app>
```
14.5.2 An Example of Security

CODE EXAMPLE 14-2 Deployment Descriptor Example Using Security

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd"
    version="2.5">
    <display-name>A Secure Application</display-name>
    <servlet>
        <servlet-name>catalog</servlet-name>
        <servlet-class>com.mycorp.CatalogServlet</servlet-class>
        <init-param>
            <param-name>catalog</param-name>
            <param-value>Spring</param-value>
        </init-param>
        <security-role-ref>
            <role-name>MGR</role-name>
            <!-- role name used in code -->
            <role-link>manager</role-link>
        </security-role-ref>
    </servlet>
    <security-role>
        <role-name>manager</role-name>
    </security-role>
    <servlet-mapping>
        <servlet-name>catalog</servlet-name>
        <url-pattern>/catalog/*</url-pattern>
    </servlet-mapping>
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>SalesInfo</web-resource-name>
            <url-pattern>/salesinfo/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>manager</role-name>
        </auth-constraint>
        <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
</web-app>
```
CODE EXAMPLE 14-2  Deployment Descriptor Example Using Security

```xml
  </user-data-constraint>
  </security-constraint>
</web-app>
```
Requirements related to other Specifications

This chapter lists the requirements for web containers that are included in products that also include other Java technologies.

In the following sections any reference to Java EE applies to not only the full Java EE profile but also any profile that includes support for Servlet, like the Java EE Web Profile. For more information on profiles please refer to the Java EE platform specification.

15.1 Sessions

Distributed servlet containers that are part of a Java EE implementation must support the mechanism necessary for migrating other Java EE objects from one JVM to another.

15.2 Web Applications

15.2.1 Web Application Class Loader

Servlet containers that are part of a Java EE product should not allow the application to override Java SE or Java EE platform classes, such as those in java.* and javax.* namespaces, that either Java SE or Java EE do not allow to be modified.
15.2.2 Web Application Environment

Java EE defines a naming environment that allows applications to easily access resources and external information without explicit knowledge of how the external information is named or organized.

As servlets are an integral component type of Java EE technology, provision has been made in the Web application deployment descriptor for specifying information allowing a servlet to obtain references to resources and enterprise beans. The deployment elements that contain this information are:

- env-entry
- ejb-ref
- ejb-local-ref
- resource-ref
- resource-env-ref
- service-ref
- message-destination-ref
- persistence-context-ref
- persistence-unit-ref

The developer uses these elements to describe certain objects that the Web application requires to be registered in the JNDI namespace in the Web container at runtime.

The requirements of the Java EE environment with regard to setting up the environment are described in Chapter 5 of the Java EE Specification.

Servlet containers that are part of a Java EE technology-compliant implementation are required to support this syntax. Consult the Java EE Specification for more details. This type of servlet container must support lookups of such objects and calls made to those objects when performed on a thread managed by the servlet container. This type of servlet container should support this behavior when performed on threads created by the developer, but are not currently required to do so. Such a requirement will be added in the next version of this specification. Developers are cautioned that depending on this capability for application-created threads is not recommended, as it is non-portable.

15.2.3 JNDI Name for Web Module Context Root URL

The Java EE Platform Specification defines a standardized global JNDI namespace and a series of related namespaces that map to various scopes of a Java EE application. These namespaces can be used by applications to portably retrieve references to components and resources. This section defines the JNDI names by which the base url for a web application is required to be registered.
The name of the pre-defined `java.net.URL` resource for the context root of a web application has the following syntax:

```java
java:global[/<app-name>]/<module-name>!ROOT in the global namespace and
java:app/<module-name>!ROOT in the application-specific namespace.
```

Please see section EE 8.1.1 (Component creation) and EE 8.1.2 (Application assembly) for the rules to determine the app name and module name.

The `<app-name>` is applicable only when the webapp is packaged within a `.ear` file.

The `java:app` prefix allows a component executing within a Java EE application to access an application-specific namespace. The `java:app` name allows a module in an enterprise application to reference the context root of another module in the same enterprise application. The `<module-name>` is a required part of the syntax for `java:app url`.

**Examples**

The above URL can then be used within an application as follows:

If a web application is deployed standalone with `module-name` as `myWebApp`. The URL can then be injected into another web module as follows:

**CODE EXAMPLE 15-1**

```java
@Resource(lookup="java:global/myWebApp!ROOT")
URL myWebApp;
```

When packaged in an ear file named `myApp` it can be used as follows:

**CODE EXAMPLE 15-2**

```java
@Resource(lookup="java:global/myApp/myWebApp!ROOT")
URL myWebApp;
```

15.3 Security

This section details the additional security requirements for web containers when included in a product that also contains EJB, JACC and or JASPIC. The following sections call out the requirements.
15.3.1 Propagation of Security Identity in EJB™ 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 enterprise beans from web applications is for the security identity of a web user to be propagated to the EJB container.

In other scenarios, web containers are required to allow web users that are not known to the web container or to the EJB container to make calls:

■ Web containers are required to support access to web resources by clients that have not authenticated themselves to the container. This is the common mode of access to web resources on the Internet.
■ Application code may be the sole processor of signon and customization of data based on caller identity.

In these scenarios, a web application deployment descriptor may specify a run-as element. When a run-as role is specified for a Servlet, the Servlet container must propagate a principal mapped to the role as the security identity in any call from the Servlet to an EJBs, including calls originating from the Servlet’s init and destroy methods. The security role name must be one of the security role names defined for the web application.

For web containers running as part of a Java EE platform, the use of run-as elements must be supported both for calls to EJB components within the same Java EE application, and for calls to EJB components deployed in other Java EE applications.

15.3.2 Container Authorization Requirements

In a Java EE product or in a product that includes support for Java Authorization Contracts for Containers (JACC, i.e, JSR 115), all Servlet containers MUST implement support for JACC. The JACC Specification is available for download at http://www.jcp.org/en/jsr/detail?id=115

15.3.3 Container Authentication Requirements

In a Java EE product, or a product that includes support for The Java Authentication SPI for Containers (JASPIC, i.e, JSR 196), all Servlet containers MUST implement the Servlet Container Profile of the JASPIC specification. The JASPIC Specification is available for download at http://www.jcp.org/en/jsr/detail?id=196
15.4 Deployment

This section details the deployment descriptor, packaging and deployment descriptor processing requirements of a Java EE technology compliant container and products that include support for JSP and or Web Services.

15.4.1 Deployment Descriptor Elements

The following additional elements exist in the Web application deployment descriptor to meet the requirements of Web containers that are JSP pages enabled or part of a Java EE application server. They are not required to be supported by containers wishing to support only the servlet specification:

- jsp-config
- Syntax for declaring resource references (env-entry, ejb-ref, ejb-local-ref, resource-ref, resource-env-ref)
- Syntax for specifying the message destination (message-destination, message-destination-ref)
- Reference to a Web service (service-ref)
- Reference to a Persistence context (persistence-context-ref)
- Reference to a Persistence Unit (persistence-unit-ref)

The syntax for these elements is now held in the JavaServer Pages specification version 2.2, and the Java EE specification.

15.4.2 Packaging and Deployment of JAX-WS Components

Web containers may choose to support running components written to implement a Web service endpoint as defined by the JAX-RPC and/or JAX-WS specifications. Web containers embedded in a Java EE conformant implementation are required to support JAX-RPC and JAX-WS web service components. This section describes the packaging and deployment model for web containers when included in a product which also supports JAX-RPC and JAX-WS.

JSR-109 [http://jcp.org/jsr/detail/109.jsp] defines the model for packaging a Web service interface with its associated WSDL description and associated classes. It defines a mechanism for JAX-WS and JAX-RPC enabled Web containers to link to a component that implements this Web service. A JAX-WS or JAX-RPC Web service implementation component uses the APIs defined by the JAX-WS and/or JAX-RPC
specifications, which defines its contract with the JAX-WS and/or JAX-RPC enabled Web containers. It is packaged into the WAR file. The Web service developer makes a declaration of this component using the usual `<servlet>` declaration.

JAX-WS and JAX-RPC enabled Web containers must support the developer in using the Web deployment descriptor to define the following information for the endpoint implementation component, using the same syntax as for HTTP Servlet components using the `servlet` element. The child elements are used to specify endpoint information in the following way:

- the `servlet-name` element defines a logical name which may be used to locate this endpoint description among the other Web components in the WAR
- the `servlet-class` element provides the fully qualified Java class name of this endpoint implementation
- the `description` element(s) may be used to describe the component and may be displayed in a tool
- the `load-on-startup` element specifies the order in which the component is initialized relative to other Web components in the Web container
- the `security-role-ref` element may be used to test whether the authenticated user is in a logical security role
- the `run-as` element may be used to override the identity propagated to EJBs called by this component

Any servlet initialization parameters defined by the developer for this Web component may be ignored by the container. Additionally, the JAX-WS and JAX-RPC enabled Web component inherits the traditional Web component mechanisms for defining the following information:

- mapping of the component to the Web container’s URL namespace using the `servlet mapping` technique
- authorization constraints on Web components using security constraints
- the ability to use servlet filters to provide low-level byte stream support for manipulating JAX-WS and/or JAX-RPC messages using the filter mapping technique
- the time out characteristics of any HTTP sessions that are associated with the component
- links to Java EE objects stored in the JNDI namespace

All of the above requirements can be met using the pluggability mechanism defined in Section 8.2, “Pluggability” on page 8-71.
15.4.3 Rules for Processing the Deployment Descriptor

The containers and tools that are part of Java EE technology-compliant implementation are required to validate the deployment descriptor against the XML schema for structural correctness. The validation is recommended, but not required for the web containers and tools that are not part of a Java EE technology compliant implementation.

15.5 Annotations and Resource Injection

The Java Metadata specification (JSR-175), which is part of J2SE 5.0 and later versions, provides a means of specifying configuration data in Java code. Metadata in Java code is also referred to as annotations. In Java EE, annotations are used to declare dependencies on external resources and configuration data in Java code without the need to define that data in a configuration file.

This section describes the behavior of annotations and resource injection in Java EE technology compliant Servlet containers. This section expands on the Java EE specification section 5 titled “Resources, Naming, and Injection.”

Annotations must be supported on the following container managed classes that implement the following interfaces and are declared in the web application deployment descriptor or using the annotations defined in Section 8.1, “Annotations and pluggability” on page 8-67 or added programmatically.

<table>
<thead>
<tr>
<th>TABLE 15-1 Components and Interfaces supporting Annotations and Dependency Injection</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Component Type</strong></td>
</tr>
<tr>
<td>Servlets</td>
</tr>
<tr>
<td>Filters</td>
</tr>
<tr>
<td>Listeners</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Web containers are not required to perform resource injection for annotations occurring in classes other than those listed above in TABLE 15-1.
References must be injected prior to any lifecycle methods being called and the component instance being made available the application.

In a web application, classes using resource injection will have their annotations processed only if they are located in the `WEB-INF/classes` directory, or if they are packaged in a jar file located in `WEB-INF/lib`. Containers may optionally process resource injection annotations for classes found elsewhere in the application’s classpath.

The web application deployment descriptor contains a `metadata-complete` attribute on the `web-app` element. The `metadata-complete` attribute defines whether the `web.xml` descriptor is complete, or whether other sources of metadata used by the deployment process should be considered. Metadata may come from the `web.xml` file, `web-fragment.xml` files, annotations on class files in `WEB-INF/classes`, and annotations on classes in jar files in the `WEB-INF/lib` directory. If `metadata-complete` is set to "true", the deployment tool only examines the `web.xml` file and must ignore annotations such as `@WebServlet`, `@WebFilter`, and `@WebListener` present in the class files of the application, and must also ignore any `web-fragment.xml` descriptor packaged in a jar file in `WEB-INF/lib`. If the `metadata-complete` attribute is not specified or is set to "false", the deployment tool must examine the class files and `web-fragment.xml` files for metadata, as previously specified.

The `web-fragment.xml` also contains the `metadata-complete` attribute on the `web-fragment` element. The attribute defines whether the `web-fragment.xml` descriptor is complete for the given fragment, or whether it should scan for annotations in the classes in the associated jar file. If `metadata-complete` is set to “true” the deployment tool only examines the `web-fragment.xml` and must ignore annotations such as `@WebServlet`, `@WebFilter` and `@WebListener` present in the class files of the fragment. If `metadata-complete` is not specified or is set to “false” the deployment tool must examine the class files for metadata.

Following are the annotations that are required by a Java EE technology compliant web container.

### 15.5.1 `@DeclareRoles`

This annotation is used to define the security roles that comprise the security model of the application. This annotation is specified on a class, and it is used to define roles that could be tested (i.e., by calling `isUserInRole`) from within the methods of the annotated class. Roles that are implicitly declared as a result of their use in a `@RolesAllowed` need not be explicitly declared using the `@DeclareRoles` annotation. The `@DeclareRoles` annotation may only be defined in classes implementing the `javax.servlet.Servlet` interface or a subclass thereof.
Following is an example of how this annotation would be used.

**CODE EXAMPLE 15-3  @DeclareRoles Annotation Example**

```java
@DeclareRoles("BusinessAdmin")
public class CalculatorServlet {
    //...
}
```

Declaring `@DeclareRoles ("BusinessAdmin")` is equivalent to defining the following in the web.xml.

**CODE EXAMPLE 15-4  @DeclareRoles web.xml**

```xml
<web-app>
    <security-role>
    <role-name>BusinessAdmin</role-name>
    </security-role>
</web-app>
```

This annotation is not used to relink application roles to other roles. When such linking is necessary, it is accomplished by defining an appropriate security-role-ref in the associated deployment descriptor.

When a call is made to `isUserInRole` from the annotated class, the caller identity associated with the invocation of the class is tested for membership in the role with the same name as the argument to `isCallerInRole`. If a security-role-ref has been defined for the argument role-name the caller is tested for membership in the role mapped to the role-name.

For further details on the `@DeclareRoles` annotation refer to the Common Annotations for the Java™ Platform™ specification (JSR 250) section 2.12.

### 15.5.2  @EJB Annotation

Enterprise JavaBeans™ 3.2 (EJB) components may be referenced from a web component using the `@EJB` annotation. The `@EJB` annotation provides the equivalent functionality of declaring the `ejb-ref` or `ejb-local-ref` elements in the deployment descriptor. Fields that have a corresponding `@EJB` annotation are injected with the a reference to the corresponding EJB component.

An example:

```java
@EJB private ShoppingCart myCart;
```
In the case above a reference to the EJB component “myCart” is injected as the value of the private field “myCart” prior to the classs declaring the injection being made available.

The behavior the @ EJB annotation is further detailed in section 11.5.1 of the EJB 3.2 specification (JSR 345).

### 15.5.3 @EJBs Annotation

The @EJBs annotation allows more than one @EJB annotations to be declared on a single resource.

An example:

```java
@EJBs({@EJB(Calculator), @EJB(ShoppingCart)})
public class ShoppingCartServlet {
    //...
}
```

The example above the EJB components ShoppingCart and Calculator are made available to ShoppingCartServlet. The ShoppingCartServlet must still look up the references using JNDI but the EJBs do not need to declared in the web.xml file.

### 15.5.4 @Resource Annotation

The @Resource annotation is used to declare a reference to a resource such as a data source, Java Messaging Service (JMS) destination, or environment entry. This annotation is equivalent to declaring a resource-ref, message-destination-ref or env-ref, or resource-env-ref element in the deployment descriptor.

The @Resource annotation is specified on a class, method or field. The container is responsible injecting references to resources declared by the @Resource annotation and mapping it to the proper JNDI resources. See the Java EE Specification Chapter 5 for further details.

An example of a @Resource annotation follows:

```java
@Resource private javax.sql.DataSource catalogDS;
public getProductsByCategory() {
    // get a connection and execute the query
```
In the example code above, a servlet, filter, or listener declares a field `catalogDS` of type `javax.sql.DataSource` for which the reference to the data source is injected by the container prior to the component being made available to the application. The data source JNDI mapping is inferred from the field name “`catalogDS`” and type (`javax.sql.DataSource`). Moreover, the `catalogDS` resource no longer needs to be defined in the deployment descriptor.

The semantics of the `@Resource` annotation are further detailed in the Common Annotations for the Java™ Platform™ specification (JSR 250) Section 2.3 and Java EE 7 Specification specification 5.2.5.

### 15.5.5 @PersistenceContext Annotation

This annotation specifies the container managed entity manager for referenced persistence units.

An example:

**CODE EXAMPLE 15-7  @PersistenceContext Example**

```java
@PersistenceContext (type=EXTENDED)
EntityManager em;
```

The behavior the `@PersistenceContext` annotation is further detailed in section 10.5.1 of the Java Persistence API, Version 2.1 (JSR 338).

### 15.5.6 @PersistenceContexts Annotation

The `PersistenceContexts` annotation allows more than one `@PersistenceContext` to be declared on a resource. The behavior the `@PersistenceContexts` annotation is further detailed in section 10.5.1 of the Java Persistence API, version 2.1 (JSR 338).
15.5.7 @PersistenceUnit Annotation

The @PersistenceUnit annotation provides Enterprise Java Beans components declared in a servlet a reference to a entity manager factory. The entity manager factory is bound to a separate persistence.xml configuration file as described in section 11.10 of the EJB 3.2 specification (JSR 345).

An example:

```java
@PersistenceUnit
EntityManagerFactory emf;
```

The behavior the @PersistenceUnit annotation is further detailed in section 10.5.2 of the Java Persistence API, version 2.1 (JSR 338).

15.5.8 @PersistenceUnits Annotation

This annotation allows for more than one @PersistenceUnit annotations to be declared on a resource. The behavior the @PersistenceUnits annotation is further detailed in section 10.5.2 of the Java Persistence API, version 2.1 (JSR 338).

15.5.9 @PostConstruct Annotation

The @PostConstruct annotation is declared on a method that does not take any arguments, and must not throw any checked exceptions. The return value must be void. The method MUST be called after the resources injections have been completed and before any lifecycle methods on the component are called.

An example:

```java
@PostConstruct
public void postConstruct() {
    ...
}
```

The example above shows a method using the @PostConstruct annotation.
The `@PostConstruct` annotation MUST be supported by all classes that support dependency injection and called even if the class does not request any resources to be injected. If the method throws an unchecked exception the class MUST not be put into service and no method on that instance can be called.

Refer to the Java EE specification section 2.5 and the Common Annotations for the Java™ Platform™ specification section 2.5 for more details.

### 15.5.10 @PreDestroy Annotation

The `@PreDestroy` annotation is declared on a method of a container managed component. The method is called prior to component being removed by the container.

An example:

**CODE EXAMPLE 15-10 @PreDestroy Example**

```java
@PreDestroy
public void cleanup() {
    // clean up any open resources
    ...
}
```

The method annotated with `@PreDestroy` must return `void` and must not throw a checked exception. The method may be public, protected, package private or private. The method must not be static however it may be final.

Refer to the JSR 250 section 2.6 for more details.

### 15.5.11 @Resources Annotation

The `@Resources` annotation acts as a container for multiple `@Resource` annotations because the Java MetaData specification does not allow for multiple annotations with the same name on the same annotation target.

An example:

**CODE EXAMPLE 15-11 @Resources Example**

```java
@Resources({
    @Resource(name="myDB" type=javax.sql.DataSource),
    @Resource(name="myMQ" type=javax.jms.ConnectionFactory)
})
```
In the example above a JMS connection factory and a data source are made available to the CalculatorServlet by means of an @Resources annotation.

The semantics of the @Resources annotation are further detailed in the Common Annotations for the Java™ Platform™ specification (JSR 250) section 2.4.

15.5.12 @RunAs Annotation

The @RunAs annotation is equivalent to the run-as element in the deployment descriptor. The @RunAs annotation may only be defined in classes implementing the javax.servlet.Servlet interface or a subclass thereof.

An example:

```java
@RunAs("Admin")
public class CalculatorServlet {
    //...
}

public class CalculatorServlet {
    @EJB private ShoppingCart myCart;

    public void doGet(HttpServletRequest req, HttpServletResponse res) {
        //....
        myCart.getTotal();
        //....
    }
    //....
}
```

The @RunAs("Admin") statement would be equivalent to defining the following in the web.xml.

```xml
<servlet>
    <servlet-name>CalculatorServlet</servlet-name>
    <run-as>Admin</run-as>
</servlet>
```
The example above shows how a servlet uses the @RunAs annotation to propagate the security identity “Admin” to an EJB component when the myCart.getTotal() method is called. For further details on propagating identities see Section 15.3.1, “Propagation of Security Identity in EJB™ Calls” on page 15-186.

For further details on the @RunAs annotation refer to the Common Annotations for the Java™ Platform™ specification (JSR 250) section 2.7.

15.5.13 @WebServiceRef Annotation

The @WebServiceRef annotation provides a reference to a web service in a web component in same way as a resource-ref element would in the deployment descriptor.

An example:
@WebServiceRef private MyService service;

In this example a reference to the web service “MyService” will be injected to the class declaring the annotation.

This annotation and behavior are further detailed in the JAX-WS Specification (JSR 224) section 7.

15.5.14 @WebServiceRefs Annotation

This annotation allows for more than one @WebServiceRef annotations to be declared on a single resource. The behavior of this annotation is further detailed in the JAX-WS Specification (JSR 224) section 7.

15.5.15 Contexts and Dependency Injection for Java EE requirements

In a product that supports Contexts and Dependency Injection for Java EE (CDI) and in which CDI is enabled, implementations MUST support the use of CDI managed beans. Servlets, Filters, Listeners and HttpUpgradeHandlers MUST support CDI injection and the use of interceptors as described in Section EE.5.24, "Support for Dependency Injection" of the Java EE 7 Platform specification.
Change Log

This document is the final release of the Java Servlet 3.1 Servlet specification developed under the Java Community Process℠ (JCP).

A.1 Changes since Servlet 3.0

1. Section 1.6, “Compatibility with Java Servlet Specification Version 2.5”. Remove subsection 1.6.1 “Listener ordering”.

2. Section 2.3.3.3, “Asynchronous processing”. And javadoc of AsyncContext.
   a. Clarified the behavior of AsyncListener.onStartAsync.
   b. Fixed errors and comments in Code Examples.
   c. Clarified the behavior of AsyncContext.getRequest and AsyncContext.getResponse after the asynchronous request is completed or dispatched.
   d. Specify the default async time out value.
   e. Clarified the behavior of AsyncListener when there is an error.

3. Added Section 2.3.3.5, “Upgrade Processing”, and new classes ProtocolHandler and WebConnection.


6. Clarified Section 4.4, “Configuration methods” that the ServletContextListener must be declared in the descriptor or annotated with @WebListener
7. Added HttpSessionIdListener to the list listeners in Section 4.4.3.1, “void addListener(String className)”, Section 4.4.3.2, “<T extends EventListener> void addListener(T t)”, Section 4.4.3.3, “void addListener(Class <? extends EventListener> listenerClass)”, Section 4.4.3.4, “<T extends EventListener> void createListener(Class<T> clazz)”, Section 8.1.4, “@WebListener” and Section 15.5, “Annotations and Resource Injection”.

8. Section 4.4.3.5, “Annotation processing requirements for programmatically added Servlets, Filters and Listeners”. Update the reference.


12. Section 6.2.1, “Filter Lifecycle” (4). Required filters and servlet processing in the same thread.


15. Section 8.1.1, “@WebServlet”. Programmatically adding servlet with a name different from that specified in annotation.

16. Section 8.2.2, “Ordering of web.xml and web-fragment.xml” and Section 8.2.4, “Shared libraries / runtimes pluggability”. The processing of HandlesTypes annotation is applied irrespective to setting of metadata-complete.

17. Section 8.2.3, “Assembling the descriptor from web.xml, web-fragment.xml and annotations”. Clarify the order in which listeners are invoked.

18. Section 8.2.4, “Shared libraries / runtimes pluggability” Clarify the creation of instance of ServletContainerInitializer.

19. Section 9.4, “The Forward Method”. Clarified the behavior of the response when the request is put in asynchronous mode.

20. TABLE 11-2. Add a “Changes to id” events.


24. Section 13.3, “Programmatic Security”, Section 13.4.1.3, “Mapping @HttpConstraint and @HttpMethodConstraint to XML.” and Section 13.8.1, “Combining Constraints”. Add descriptions for role “*” and “**”.


26. Section 13.6.3.1, “Login Form Notes”. Add autocomplete=”off’.


29. Section 15.3.1, “Propagation of Security Identity in EJB™ Calls”. Explicitly mentioned the Servlet.init and Servlet.destroy.

30. Section 15.5.15, “Contexts and Dependency Injection for Java EE requirements”. Add HttpUpgradeHandler and add reference to Java EE 7 specification.


33. Add new methods ServletRequest.getContentLengthLong and ServletResponse.setContentLengthLong.

34. Add the new Part.getSubmittedFileName.

---

### A.2 Changes since Servlet 3.0 Proposed Final Draft

1. Re-factored some of the Async APIs - moved addAsyncListener to AsyncContext and renamed it to addListener. Moved setAsyncTimeout to AsyncContext and renamed it to setTimeout.

2. Clarified some of the semantics around concurrent access to the request and response in async processing.


4. Added a new annotation - @ServletSecurity (and associated annotation for the fields) for defining security as opposed to re-using the @RolesAllowed, @PermitAll, @DenyAll
A.3 Changes since Servlet 3.0 Public Review

1. Updated isAsyncStarted to return false once a dispatch to the container or a call to complete is done from the async handler
2. Added ordering support for fragments
3. Added support for file upload
4. Added support for loading static resources and JSPs from JAR files that are included in the META-INF/resources directory of the JAR file which is then bundled in the WEB-INF/lib directory
5. Changed annotation names based on feedback on Public Review of the specification
6. Added programmatic login / logout support
7. Added support for security related common annotations - @RolesAllowed, @PermitAll, @DenyAll
8. Clarified welcome files

A.4 Changes since Servlet 3.0 EDR

1. The suspend / resume APIs are no longer present in the specification. They have been replaced by startAsync and AsyncContext which now has forward and complete methods.
2. Annotation names have changed and there are only top level annotations. The method level annotations for declaring the servlet methods are no longer being used.
3. The rules for assembling web.xml from fragments and annotations is described.

A.5 Changes since Servlet 2.5 MR6

1. Added support for annotations and web fragments
2. Added support for suspend / resume to allow async support in servlets.
3. Added support for initializing servlets and filters from the ServletContext at initialization time.

4. Added support for HttpOnly cookies and allow configuring cookies.

5. Added convenience methods to ServletRequest to get Response and ServletContext

A.6 Changes since Servlet 2.5 MR 5

A.6.1 Clarify SRV 8.4 "The Forward Method"

Change the last sentence of the section which currently is:

"Before the forward method of the RequestDispatcher interface returns, the response content must be sent and committed, and closed by the servlet container."

to read:

"Before the forward method of the RequestDispatcher interface returns without exception, the response content must be sent and committed, and closed by the servlet container. If an error occurs in the target of the RequestDispatcher.forward() the exception may be propagated back through all the calling filters and servlets and eventually back to the container."

A.6.2 Update Deployment descriptor "http-method values allowed"

The facet for http-method element in the deployment descriptor is currently more restrictive than the http specification. The following change is being made to the descriptor to allow the set of method names as defined by the http specification. The pattern value of http-methodType is being changed from

<xsd:pattern value="[\p{L}-[\p{Cc}\p{Z}]]+"/>

to closely match what the HTTP specification lists as allowable HTTP methods names.

<xsd:pattern value="[&#33;-&#126;-%00-%90;@,;:/\[\]?=\{"\}\p{Z}]]+"/>
A.6.3 Clarify SRV 7.7.1 "Threading Issues"

Change the paragraph which currently is:

"Multiple servlets executing request threads may have active access to a single session object at the same time. The Developer has the responsibility for synchronizing access to session resources as appropriate."

to read:

"Multiple servlets executing request threads may have active access to the same session object at the same time. The container must ensure that manipulation of internal data structures representing the session attributes is performed in a threadsafe manner. The Developer has the responsibility for threadsafe access to the attribute objects themselves. This will protect the attribute collection inside the HttpSession object from concurrent access, eliminating the opportunity for an application to cause that collection to become corrupted."

A.7 Changes Since Servlet 2.5 MR 2

A.7.1 Updated Annotation Requirements for Java EE containers

Added EJBs, PreDestroy, PersistenceContext, PersistenceContexts, PersistenceUnit, and PersistenceUnits with descriptions to the list of required Java EE container annotations in Section 15.5, “Annotations and Resource Injection”.

A.7.2 Updated Java Enterprise Edition Requirements

Updated the Annotations to the final Java EE annotation names. Also updated the "full" attribute in the web.xml to be "metadata-complete".

A.7.3 Clarified HttpServletRequest.getRequestURL()

The API documentation for javax.servlet.http.HttpServletRequest.getRequestURL() was clarified.

The text in italics was added:
If this request has been forwarded using
RequestDispatcher.forward(ServletRequest, ServletResponse), the
server path in the reconstructed URL must reflect the path used to obtain the
RequestDispatcher, and not the server path specified by the client. Because this method
returns a StringBuffer, not a string, you can modify the URL easily, for example, to
append query parameters.

A.7.4 Removal of IllegalStateException from HttpSession.getId()

The HttpSessionBindingListener calls the valueUnbound event after the session has
been expired, unfortunately, the HttpSession.getId() method is often used in this
scenario and is supposed to throw an IllegalStateException. The servlet EG agreed to
remove the exception from the API to prevent these types of exceptions.

A.7.5 ServletContext.getContextPath()

The method getContextPath() was added to the ServletContext in the API. The
description is as follows:

public java.lang.String getContextPath()

Returns the context path of the web application. The context path is the portion of
the request URI that is used to select the context of the request. The context path
always comes first in a request URI. The path starts with a "/" character but does not
end with a "/" character. For servlets in the default (root) context, this method
returns ".".

It is possible that a servlet container may match a context by more than one context
path. In such cases this method will return the actual context path used by the
request and it may differ from the path returned by this method. The context path
returned by this method should be considered as the prime or preferred context path
of the application.

Returns: The context path of the web application.

HttpServletRequest.getContextPath() was updated to clarify its relationship with the
ServletContext.getContextPath() method. The clarification is as follows.

It is possible that a servlet container may match a context by more than one context
path. In such cases this method will return the actual context path used by the
request and it may differ from the path returned by the
ServletContext.getContextPath() method. The context path returned by ServletContext.getContextPath() should be considered as the prime or preferred context path of the application.

A.7.6 Requirement for web.xml in web applications

Section 10.13, “Inclusion of a web.xml Deployment Descriptor” was added which removes requirement for Java EE compliant web applications. The section is as follows:

A web application is NOT required to contain a web.xml if it does NOT contain any Servlet, Filter, or Listener components. In other words an application containing only static files or JSP pages does not require a web.xml to be present.

A.8 Changes Since Servlet 2.4

A.8.1 Session Clarification

Clarified Section 7.3, “Session Scope” to allow for better support of session ids being used in more than one context. This was done to support the Portlet specification (JSR 168). Added the following paragraph at the end of Section 7.3:

“Additionally, sessions of a context must be resumable by requests into that context regardless of whether their associated context was being accessed directly or as the target of a request dispatch at the time the sessions were created.”

Made the changes in Section 9.3, “The Include Method” by replacing the following text:

"It cannot set headers or call any method that affects the headers of the response. Any attempt to do so must be ignored."

with the following:

"It cannot set headers or call any method that affects the headers of the response, with the exception of the HttpServletRequest.getSession() and HttpServletRequest.getSession(boolean) methods. Any attempt to set the headers must be ignored, and any call to HttpServletRequest.getSession() or HttpServletRequest.getSession(boolean) that would require adding a Cookie response header must throw an IllegalStateException if the response has been committed."
A.8.2 Filter All Dispatches

Modified Section 6.2.5, “Filters and the RequestDispatcher” to clarify a way to map a filter to all servlet dispatches by appending the following text to the end of the section:

Finally, the following code uses the special servlet name ‘*’:

**CODE EXAMPLE A-1**  Example of special servlet name ‘*’

```xml
<filter-mapping>
  <filter-name>All Dispatch Filter</filter-name>
  <servlet-name>*</servlet-name>
  <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

This code would result in the All Dispatch Filter being invoked on request dispatcher forward() calls for all request dispatchers obtained by name or by path.

A.8.3 Multiple Occurrences of Servlet Mappings

Previous versions of the servlet schema allows only a single url-pattern or servlet name per servlet mapping. For servlets mapped to multiple URLs this results in needless repetition of whole mapping clauses.

The deployment descriptor `servlet-mappingType` was updated to:

**CODE EXAMPLE A-2**  `servlet-mappingType` descriptor

```xml
  <xsd:complexType name="servlet-mappingType">
    <xsd:sequence>
      <xsd:element name="servlet-name" type="j2ee:servlet-nameType"/>
      <xsd:element name="url-pattern" type="j2ee:url-patternType" minOccurs="1" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
```
A.8.4 Multiple Occurrences Filter Mappings

Previous versions of the servlet schema allows only a single url-pattern in a filter mapping. For filters mapped to multiple URLs this results in needless repetition of whole mapping clauses.

The deployment descriptor schema the filter-mappingType was updated to:

CODE EXAMPLE A-3 Updated filter-mappingType schema

```xml
<xsd:complexType name="filter-mappingType">
  <xsd:sequence>
    <xsd:element name="filter-name" type="j2ee:filter-nameType"/>
    <xsd:choice minOccurs="1" maxOccurs="unbounded">
      <xsd:element name="url-pattern" type="j2ee:url-patternType"/>
      <xsd:element name="servlet-name" type="j2ee:servlet-nameType"/>
    </xsd:choice>
    <xsd:element name="dispatcher" type="j2ee:dispatcherType" minOccurs="0" maxOccurs="4"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
```

This change allows multiple patterns and servlet names to be defined in a single mapping as can be seen in the following example:

CODE EXAMPLE A-4 Filter mapping example

```xml
<filter-mapping>
  <filter-name>Demo Filter</filter-name>
  <url-pattern>/foo/*</url-pattern>
  <url-pattern>/bar/*</url-pattern>
  <servlet-name>Logger</servlet-name>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

Section 6.2.4, “Configuration of Filters in a Web Application” was updated to clarify the cases where there are multiple mappings with the following text:

"If a filter mapping contains both <servlet-name> and <url-pattern>, the container must expand the filter mapping into multiple filter mappings (one for each <servlet-name> and <url-pattern>), preserving the order of the <servlet-name> and <url-pattern> elements."
An example was also provided to clarify cases when there are multiple mappings.

### A.8.5 Support Alternative HTTP Methods with Authorization Constraints

The previous Servlet 2.4 schema restricted HTTP methods to GET, POST, PUT, DELETE, HEAD, OPTIONS, and TRACE. The schema http-methodType was changed from:

**Code Example A-5**  
Servlet 2.4 http-methodType schema

```xml
<xsd:complexType name="http-methodType">
  ...
  <xsd:simpleContent>
    <xsd:restriction base="j2ee:string">
      <xsd:enumeration value="GET"/>
      <xsd:enumeration value="POST"/>
      <xsd:enumeration value="PUT"/>
      <xsd:enumeration value="DELETE"/>
      <xsd:enumeration value="HEAD"/>
      <xsd:enumeration value="OPTIONS"/>
      <xsd:enumeration value="TRACE"/>
    </xsd:restriction>
  </xsd:simpleContent>
</xsd:complexType>
```

To the following:

**Code Example A-6**  
Servlet 2.5 http-methodType schema

```xml
<xsd:simpleType name="http-methodType">
  <xsd:annotation>
    <xsd:documentation>
      A HTTP method type as defined in HTTP 1.1 section 2.2.
    </xsd:documentation>
  </xsd:annotation>

  <xsd:restriction base="xsd:token">
    <xsd:pattern value="[\p{L}-[\p{Cc}\p{Z}]]+"/>
  </xsd:restriction>
</xsd:simpleType>
```

The http-method elements now need to be a token as described in HTTP 1.1 specification section 2.2.
A.8.6 Minimum J2SE Requirement

Servlet 2.5 Containers now require J2SE 5.0 as the minimum Java version. Section 1.2, “What is a Servlet Container?” was updated to reflect this requirement.

A.8.7 Annotations and Resource Injection

Java EE technology compliant containers require annotations and resource injection on servlets, filters, and listeners. Section 15.5, “Annotations and Resource Injection” describes the annotations and resource injection in further detail.

A.8.8 SRV.9.9 ("Error Handling") Requirement Removed

Section 10.9.1, “Request Attributes” defines the following requirement:

If the location of the error handler is a servlet or a JSP page:

[...]

The response setStatus method is disabled and ignored if called.

[...]

The JSP 2.1 EG has asked that this requirement above be removed to allow JSP error pages to update the response status.

A.8.9 HttpServletRequest.isRequestedSessionIdValid() Clarification

The API clarification better describes what happens when a client did not specify a session id. The API documentation was updated to specify when false is returned. The API documentation now states:

Returns false if the client did not specify any session ID.

A.8.10 SRV.5.5 ("Closure of Response Object") Clarification

The behavior in Section 5.6, “Closure of Response Object” the response’s content length is set to 0 via response.setHeader("Content-Length", "0") and any subsequently setHeader() calls are ignored.
Section 5.6, “Closure of Response Object” was updated to allow all headers to be set by changing:

"The amount of content specified in the setContentLength method of the response and has been written to the response"

To the following:

"The amount of content specified in the setContentLength method of the response has been greater than zero and has been written to the response"

A.8.11 ServletRequest.setCharacterEncoding() Clarified

The API was updated to described the behavior if the method is called after the getReader() was called. If the getReader() is called there will be no effect.

A.8.12 Java Enterprise Edition Requirements

Chapter 15, “Requirements related to other Specifications details all requirements of a Java EE container. Previously the requirements were mixed into each chapter.

A.8.13 Servlet 2.4 MR Change Log Updates Added

Added the changes from the Servlet 2.4 Maintenance Review. These changes include grammar and typographical fixes.

A.8.14 Synchronized Access Session Object Clarified

Section 7.7.1, “Threading Issues” was updated to clarify that access to the session object should be synchronized.

A.9 Changes Since Servlet 2.3

- Optional “X-Powered-By” header is added in the response (5.2)
- Clarification of “overlapping constraint” (12.8.1, 12.8.2)
- Add the section to clarify the process order at the time of web application deployment (9.12)
- Clarification that the security model is also applied to filter (12.2)
- Change the status code from 401 to 200 when FORM authentication is failed as there is no appropriate error status code in HTTP/1.1 (12.5.3)
- Clarification of the wrapper objects (6.2.2)
- Clarification of overriding the platform classes (9.7.2)
- Clarification of welcome file (9.10)
- Clarification of internationalization - the relationship among setLocale, setContentType, and setCharacterEncoding (5.4, 14.2.22)
- Clarification of ServletRequestListener and ServletRequestAttributeListener description (14.2.18, 14.2.20)
- Add HttpSessionActivationListener and HttpSessionBindingListener into the Table 10-1.
- Change the word "auth constraint" to "authorization constraint" (12.8)
- Add “Since” tag in the newly added methods in javadoc(14.2.16, 14.2.22)
- Fix the data type of <session-timeout> to xsdIntegerType in schema(13.3)
- Clarification when the listener throws the unhandled exception(10.6)
- Clarification of the “shared library”(9.7.1)
- Clarification of the container’s mechanism for the extension(9.7.1, third paragraph)
- HttpSession.logout method was removed. The portable authentication mechanism will be addressed in the next version of this specification and logout will also be discussed in that scope.(12.10)
- It is now a recommendation, instead of a requirement, that the reference to the request and response object should not be given to the object in other threads - based on the requirement from JSR-168. Warnings are added when the thread created by the application uses the objects managed by the container.(2.3.3.3)
- It is now a recommendation, that the dispatch should occur in the same thread of the same JVM as the original request - based on the requirement from JSR-168(8.2)
- Clarification of “wrap” (6.2.2)
- Clarification of handling the path parameter for the mapping(11.1)
- Add the description about the “HTTP chunk” in HttpServlet.doGet method(15.1.2)
- J2SE 1.3 is the minimum version of the underlying Java platform with which servlet containers must be built (1.2)
- Clarification of ServletResponse.setBufferSize method (5.1)
- Clarification of ServletRequest.getServerName and getServerPort (14.2.16.1)
- Clarification of Internationalization (5.4, 14.2.22)
- Clarification of the redirection of the welcome file (9.10)
- Clarification of ServletContextListener.contextInitialized (14.2.12.1)
- Clarification of HttpServletRequest.getRequestedSessionId - making it clear that it returns the session ID specified by the client (15.1.3.2)
- Clarification of the class loader for the extensions - the class loader must be the same for all web applications within the same JVM (9.7.1)
- Clarification of the case when ServletRequestListener throws an unhandled exception (10.6, 14.2.20)
- Clarification of the scope of ServletRequestListener (14.2.20)
- Add the description about the case when the container has a caching mechanism (1.2)
- Validating deployment descriptor against the schema is required for Java EE containers (13.2)
- Sub elements under <web-app> can be in an arbitrary order (13.2)
- One example of the container’s rejecting the web application was removed due to the contradiction with SRV.11.1 (9.5)
- url-patternType is changed from j2ee:string to xsd:string (13)
- The sub-elements under <web-app> in deployment descriptor can be in the arbitrary order (13)
- The container must inform a developer with a descriptive error message when deployment descriptor file contains an illegal character or multiple elements of <session-config>, <jsp-config>, or <login-config> (13)
- Extensibility of deployment descriptor was removed (13)
- Section SRV.1.6 added - describing the compatibility issue with the previous version of this specification (1.6)
- New attributes are added in RequestDispatcher.forward method (8.4.2)
- New methods in ServletRequest interface and ServletRequestWrapper (14.2.16.1)
- The interface SingleThreadModel was deprecated ((2.2.1, 2.3.3.1, 14.2.24)
- Change the name of the method ServletRequestEvent.getRequestDispatcher to ServletRequestEvent.getServletRequest (14.2.19.2)
- Clarification of the “request” to access to WEB-INF directory (9.5)
- Clarification of the behavior of ServletRequest.setAttribute - change “value” to “object” in “If the value passed in is null,” (14.2.16.1)
- Fix the inconsistency between this specification and HttpServletRequest, getServletPath - the return value starts with “/” (15.1.3.2)
- Fix the inconsistency between this specification and HttpServletRequest.getPathInfo - the return value starts with “/” (15.1.3.2)
- Fix the inconsistency between this specification and HttpServletRequest.getPathTranslated - add the case when the container cannot translate the path (15.1.3.2)
- Allow HttpServletRequest.getAuthType to return not only pre-defined four authentication scheme but also the container-specific scheme (15.1.3.2)
- Change the behavior of HttpSessionListener.sessionDestroyed to notify before the session is invalidated (15.1.14.1)
- Fix the wrong status code of 403 to 404 (9.5, 9.6)
- Element “taglib” should be “jsp-config” (13.2)
- Fix the version number of JSP specification to 2.0
- Fix the wrong formats (5.5, 6.2.5, 12.8.3, 12.9)
- HTTP/1.1 is now required (1.2)
- <url-pattern> in <web-resource-collection> is mandatory (13.4)
- Clarification of IllegalArgumentException in the distributed environments (7.7.2)
- Clarification of error page handling (9.9.1, 9.9.2, 9.9.3, 6.2.5)
- Clarification of Security Constraints, especially in the case of overlapping constraints (12.8)
- Clarification of the case when <session-timeout> element is not specified (13.4)
- Clarification of the case when the resource is permanently unavailable (2.3.3.2)
- Add missing getParameterMap() in the enumerated list (4.1)
- Clarification of the status code when /WEB-INF/ resource is accessed (9.5)
- Clarification of the status code when /META-INF/ resource is accessed (9.6)
- Change xsd:string to j2ee:string in deployment descriptor (13.4)
- Extensibility of deployment descriptors (SRV.13)
- XML Schema definition of deployment descriptor (SRV.13)
- Request listeners (SRV.10 and API change)
  - New API: ServletRequestListener, ServletRequestAttributeListener and associated event classes
- Ability to use Filters under the Request Dispatcher (6.2.5)
- Required class loader extension mechanism (9.7.1)
- Listener exception handling (10.6)
- Listener order vs. servlet init()/destroy() clarification (ServletContextListener javadoc change)
- Servlets mapped to WEB-INF / response handling (9.5)
- Request dispatcher / path matching rules (8.1)
- Welcome files can be servlets (9.10)
- Internationalization enhancements (5.4, 14.2, 22, 15.1.5)
- SC_FOUND(302) addition (15.1.5)
- “Relative path” in getRequestDispatcher() must be relative against the current servlet (8.1)
- Bug fix in the example of XML (13.7.2)
- Clarification of access by getResource “only to the resource” (3.5)
- Clarification of SERVER_NAME and SERVER_PORT in getServerName() and getServerPort() (14.2.16)
- Clarification: “run-as” identity must apply to all calls from a servlet including init() and destroy() (12.7)
- Login/logout description and methods added (12.10, 15.1.7)
Glossary

A

Application Developer
The producer of a web application. The output of an Application Developer is a set of servlet classes, JSP pages, HTML pages, and supporting libraries and files (such as images, compressed archive files, etc.) for the web application. The Application Developer is typically an application domain expert. The developer is required to be aware of the servlet environment and its consequences when programming, including concurrency considerations, and create the web application accordingly.

Application Assembler
Takes the output of the Application Developer and ensures that it is a deployable unit. Thus, the input of the Application Assembler is the servlet classes, JSP pages, HTML pages, and other supporting libraries and files for the web application. The output of the Application Assembler is a web application archive or a web application in an open directory structure.

D

Deployer
The Deployer takes one or more web application archive files or other directory structures provided by an Application Developer and deploys the application into a specific operational environment. The operational environment includes
a specific servlet container and web server. The Deployer must resolve all the external dependencies declared by the developer. To perform his role, the deployer uses tools provided by the Servlet Container Provider.

The Deployer is an expert in a specific operational environment. For example, the Deployer is responsible for mapping the security roles defined by the Application Developer to the user groups and accounts that exist in the operational environment where the web application is deployed.

**P**

**principal**  
A principal is an entity that can be authenticated by an authentication protocol. A principal is identified by a *principal name* and authenticated by using *authentication data*. The content and format of the principal name and the authentication data depend on the authentication protocol.

**R**

**role (development)**  
The actions and responsibilities taken by various parties during the development, deployment, and running of a web application. In some scenarios, a single party may perform several roles; in others, each role may be performed by a different party.

**role (security)**  
An abstract notion used by an Application Developer in an application that can be mapped by the Deployer to a user, or group of users, in a security policy domain.

**S**

**security policy domain**  
The scope over which security policies are defined and enforced by a security administrator of the security service. A security policy domain is also sometimes referred to as a *realm*.

**security technology domain**  
The scope over which the same security mechanism, such as Kerberos, is used to enforce a security policy. Multiple security policy domains can exist within a single technology domain.
Servlet Container Provider

A vendor that provides the runtime environment, namely the servlet container and possibly the web server, in which a web application runs as well as the tools necessary to deploy web applications.

The expertise of the Container Provider is in HTTP-level programming. Since this specification does not specify the interface between the web server and the servlet container, it is left to the Container Provider to split the implementation of the required functionality between the container and the server.

Servlet definition

A unique name associated with a fully qualified class name of a class implementing the Servlet interface. A set of initialization parameters can be associated with a servlet definition.

Servlet mapping

A servlet definition that is associated by a servlet container with a URL path pattern. All requests to that path pattern are handled by the servlet associated with the servlet definition.

System Administrator

The person responsible for the configuration and administration of the servlet container and web server. The administrator is also responsible for overseeing the well-being of the deployed web applications at run time.

This specification does not define the contracts for system management and administration. The administrator typically uses runtime monitoring and management tools provided by the Container Provider and server vendors to accomplish these tasks.
A compact string representation of resources available via the network. Once the resource represented by a URL has been accessed, various operations may be performed on that resource.¹ A URL is a type of uniform resource identifier (URI). URLs are typically of the form:

<protocol>://<servername>/<resource>

For the purposes of this specification, we are primarily interested in HTTP-based URLs which are of the form:

http[s]://<servername>[<port>]/<url-path>[?<query-string>]

For example:

https://javashop.sun.com/purchase

In HTTP-based URLs, the ‘/’ character is reserved to separate a hierarchical path structure in the URL-path portion of the URL. The server is responsible for determining the meaning of the hierarchical structure. There is no correspondence between a URL-path and a given file system path.

A collection of servlets, JSP pages, HTML documents, and other web resources which might include image files, compressed archives, and other data. A web application may be packaged into an archive or exist in an open directory structure.

All compatible servlet containers must accept a web application and perform a deployment of its contents into their runtime. This may mean that a container can run the application directly from a web application archive file or it may mean that it will move the contents of a web application into the appropriate locations for that particular container.

¹. See RFC 1738
web application archive  A single file that contains all of the components of a web application. This archive file is created by using standard JAR tools which allow any or all of the web components to be signed.

Web application archive files are identified by the .war extension. A new extension is used instead of .jar because that extension is reserved for files which contain a set of class files and that can be placed in the classpath or double clicked using a GUI to launch an application. As the contents of a web application archive are not suitable for such use, a new extension was in order.

web application, distributable  A web application that is written so that it can be deployed in a web container distributed across multiple Java virtual machines running on the same host or different hosts. The deployment descriptor for such an application uses the distributable element.