



Java™ Servlet Specification

Java Card™ Platform, Version 3.0.1

Connected Edition

Copyright © 2009 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at <http://www.sun.com/patents> and one or more additional patents or pending patent applications in the U.S. and in other countries.

U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

This distribution may include materials developed by third parties.

Sun, Sun Microsystems, the Sun logo, Java, Solaris, Java Card, Mozilla, Netscape, Javadoc, JDK, JVM, NetBeans and Servlet are trademarks or registered trademarks of Sun Microsystems, Inc. or its subsidiaries in the U.S. and other countries.

UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

The Adobe logo is a trademark or registered trademark of Adobe Systems, Incorporated.

Products covered by and information contained in this service manual are controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright © 2009 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, États-Unis. Tous droits réservés.

Sun Microsystems, Inc. détient les droits de propriété intellectuelle relatifs à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et ce sans limitation, ces droits de propriété intellectuelle peuvent inclure un ou plus des brevets américains listés à l'adresse <http://www.sun.com/patents> et un ou les brevets supplémentaires ou les applications de brevet en attente aux États - Unis et dans les autres pays.

Droits du gouvernement des États-Unis – Logiciel Commercial. Les droits des utilisateur du gouvernement des États-Unis sont soumis aux termes de la licence standard Sun Microsystems et aux conditions appliquées de la FAR et de ces compléments.

Cette distribution peut inclure des éléments développés par des tiers.

Sun, Sun Microsystems, le logo Sun, Java, Solaris, Java Card, Mozilla, Netscape, Javadoc, JDK, JVM, NetBeans et Servlet sont des marques de fabrique ou des marques déposées enregistrées de Sun Microsystems, Inc. ou ses filiales aux États-Unis et dans d'autres pays.

UNIX est une marque déposée aux États-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Le logo Adobe est une marque déposée de Adobe Systems, Incorporated.

Les produits qui font l'objet de ce manuel d'entretien et les informations qu'il contient sont régis par la législation américaine en matière de contrôle des exportations et peuvent être soumis au droit d'autres pays dans le domaine des exportations et importations. Les utilisations finales, ou utilisateurs finaux, pour des armes nucléaires, des missiles, des armes biologiques et chimiques ou du nucléaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou réexportations vers des pays sous embargo des États-Unis, ou vers des entités figurant sur les listes d'exclusion d'exportation américaines, y compris, mais de manière non exclusive, la liste de personnes qui font objet d'un ordre de ne pas participer, d'une façon directe ou indirecte, aux exportations de des produits ou des services qui sont régis par la législation américaine sur le contrôle des exportations et la liste de ressortissants spécifiquement désignés, sont rigoureusement interdites.

LA DOCUMENTATION EST FOURNIE "EN L'ÉTAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISÉE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE À LA QUALITÉ MARCHANDE, À L'APTITUDE À UNE UTILISATION PARTICULIÈRE OU À L'ABSENCE DE CONTREFAÇON.



Adobe PostScript™

Contents

Preface xix

1. Architecture Overview 1-1

- 1.1 Definition of a Servlet 1-1
- 1.2 Definition of a Servlet Container 1-2
- 1.3 Example of Servlet Invocation 1-2
- 1.4 Compatibility With Java Servlet Specification Version 2.4 1-3
 - 1.4.1 Features Not In The Java Card Platform Version 1-3
 - 1.4.2 Backward Compatibility Support of Web Applications 1-4
 - 1.4.3 Temporary Working Directories 1-4
 - 1.4.4 SSL Attributes 1-4
 - 1.4.5 Removed Elements of the Deployment Descriptor 1-5
 - 1.4.5.1 Removed Elements for Deployment in Web Container
JSP Pages Enabled or Part of a Java EE Application
Server 1-5
 - 1.4.5.2 distributable Element Removed 1-5
 - 1.4.5.3 run-as Element Removed 1-5
 - 1.4.5.4 dispatcher Element Removed 1-6
 - 1.4.5.5 auth-method Element Value CLIENT-CERT
Removed 1-6
 - 1.4.6 realm-name Element Extended Applicability 1-6

- 1.4.7 Filtering of Authorization Request Header 1-6
- 1.4.8 Web Application Deployment Hierarchy and Directory Structure 1-7
- 1.4.9 Usage of URL Patterns for Servlet Mapping, Filter Mapping and Security Constraints Restricted 1-7
- 1.4.10 Combination of Security Constraints Precluded 1-8
- 1.4.11 Default Servlet Implementation 1-8
- 1.4.12 Classes, Interfaces and Methods Depending on API Not Supported on the Java Card Platform 1-8
 - 1.4.12.1 `java.io.Serializable` Implementation Removed 1-8
 - 1.4.12.2 `java.lang.Cloneable` Implementation Removed 1-9
 - 1.4.12.3 `HttpSessionActivationListener` Interface Removed 1-9
 - 1.4.12.4 `ServletContext` methods `getResource` and `getResourcePaths` Removed 1-9
 - 1.4.12.5 `ServletRequest` and `ServletRequestWrapper` method `getParameterMap` Removed 1-9
 - 1.4.12.6 `getUserPrincipal` method of `HttpServletRequest` and `HttpServletRequestWrapper` Removed 1-10
 - 1.4.12.7 `ServletOutputStream` methods `print(float)`, `print(double)`, `println(float)` and `println(double)` Removed 1-10
- 1.4.13 Removed Classes, Interfaces and Methods Deprecated in Java Servlet API Specification Version 2.4 1-10
 - 1.4.13.1 `SingleThreadModel` Interface Removed 1-11
 - 1.4.13.2 `HttpSessionContext` Interface Removed 1-11
 - 1.4.13.3 `HttpUtils` Class Removed 1-11
 - 1.4.13.4 `ServletContext` methods `getServlet`, `getServletNames`, `getServlets`, and `log` Removed 1-11

- 1.4.13.5 `getRealPath` method of `ServletRequest` and `ServletRequestWrapper` Removed 1-12
- 1.4.13.6 `UnavailableException` constructors and method `getServlet` Removed 1-12
- 1.4.13.7 `isRequestedSessionIdFromUrl` method of `HttpServletRequest` and `HttpServletRequestWrapper` Removed 1-13
- 1.4.13.8 `HttpServletResponse` and `HttpServletResponseWrapper` methods `encodeRedirectUrl`, `encodeUrl` and `setStatus` Removed 1-13
- 1.4.13.9 `HttpSession` methods `getSessionContext`, `getValue`, `getValueNames`, `putValue` and `removeValue` Removed 1-13
- 1.4.13.10 Attributes Deprecated or Redundant in Java Servlet API Specification Version 2.4 1-14

2. The Servlet Interface 2-1

- 2.1 Request Handling Methods 2-1
 - 2.1.1 HTTP Specific Request Handling Methods 2-1
 - 2.1.2 Additional Methods 2-2
 - 2.1.3 Conditional GET Support 2-2
- 2.2 Number of Instances 2-2
- 2.3 Servlet Life Cycle 2-3
 - 2.3.1 Loading and Instantiation 2-3
 - 2.3.2 Initialization 2-3
 - 2.3.2.1 Error Conditions on Initialization 2-4
 - 2.3.2.2 Tool Considerations 2-4
 - 2.3.3 Request Handling 2-4
 - 2.3.3.1 Multithreading Issues 2-4
 - 2.3.3.2 Exceptions During Request Handling 2-5
 - 2.3.3.3 Thread Safety 2-5
 - 2.3.4 End of Service 2-6

3. Servlet Context 3-1

- 3.1 Introduction to the ServletContext Interface 3-1
- 3.2 Scope of a ServletContext Interface 3-1
- 3.3 Initialization Parameters 3-2
- 3.4 Context Attributes 3-2
- 3.5 Resources 3-2
- 3.6 Multiple Hosts and Servlet Contexts 3-3

4. The Request 4-1

- 4.1 HTTP Protocol Parameters 4-1
 - 4.1.1 When Parameters Are Available 4-2
- 4.2 Attributes 4-2
- 4.3 Headers 4-3
- 4.4 Request Path Elements 4-3
- 4.5 Path Translation Methods 4-5
- 4.6 Cookies 4-5
- 4.7 SSL Attributes 4-6
- 4.8 Internationalization 4-6
- 4.9 Request Data Encoding 4-7
- 4.10 Lifetime of the Request Object 4-7

5. The Response 5-1

- 5.1 Buffering 5-1
- 5.2 Headers of an HTTP Response 5-2
- 5.3 Convenience Methods 5-3
- 5.4 Internationalization 5-4
- 5.5 Closure of Response Object 5-5
- 5.6 Lifetime of the Response Object 5-5

6. Filtering 6-1

6.1	What is a Filter?	6-1
6.1.1	Examples of Filtering Components	6-2
6.2	Main Concepts of Filtering	6-2
6.2.1	Filter Life Cycle	6-3
6.2.2	Wrapping Requests and Responses	6-4
6.2.3	Filter Environment	6-4
6.2.4	Configuration of Filters in a Web Application	6-5
6.2.5	Filters and the RequestDispatcher	6-7
7.	Sessions	7-1
7.1	Session Tracking Mechanisms	7-1
7.1.1	Cookies	7-1
7.1.2	SSL Sessions	7-2
7.1.3	URL Rewriting	7-2
7.1.4	Session Integrity	7-2
7.2	Creating a Session	7-2
7.3	Session Scope	7-3
7.4	Binding Attributes into a Session	7-3
7.5	Session Timeouts	7-4
7.6	Last Accessed Times	7-4
7.7	Important Session Semantics	7-5
7.7.1	Threading Issues	7-5
7.7.2	Client Semantics	7-5
8.	Dispatching Requests	8-1
8.1	Obtaining a RequestDispatcher	8-1
8.1.1	Query Strings in Request Dispatcher Paths	8-2
8.2	Using a Request Dispatcher	8-2
8.3	The Include Method	8-3

8.3.1	Included Request Parameters	8-3
8.4	The Forward Method	8-4
8.4.1	Query String	8-4
8.4.2	Forwarded Request Parameters	8-4
8.5	Error Handling	8-5
9.	Web Applications	9-1
9.1	Web Applications Within Web Servers	9-1
9.2	Relationship to ServletContext	9-1
9.2.1	Elements of a Web Application	9-1
9.3	Deployment Hierarchies	9-2
9.4	Directory Structure	9-2
9.4.1	Example of Application Directory Structure	9-3
9.5	Web Application Archive File	9-3
9.6	Web Application Deployment Descriptor	9-4
9.6.1	Dependencies On Libraries External to WAR File	9-4
9.6.2	Web Application Class Loader	9-4
9.7	Error Handling	9-5
9.7.1	Request Attributes	9-5
9.7.2	Error Pages	9-5
9.8	Welcome Files	9-7
9.9	Web Application Deployment	9-8
10.	Application Life Cycle Events	10-1
10.1	Event Listeners	10-1
10.1.1	Event Types and Listener Interfaces	10-2
10.1.2	An Example of Listener Use	10-3
10.2	Listener Class Configuration	10-3
10.2.1	Provision of Listener Classes	10-3

10.2.2	Deployment Declarations	10-3
10.2.3	Listener Registration	10-4
10.2.4	Notifications At Shutdown	10-4
10.3	Deployment Descriptor Example	10-4
10.4	Listener Instances and Threading	10-5
10.5	Listener Exceptions	10-5
10.6	Session Events	10-6
11.	Mapping Requests To Servlets	11-1
11.1	Use of URL Paths	11-1
11.2	Specification of Mappings	11-2
11.2.1	Implicit Mappings	11-2
11.2.2	Example Mapping Set	11-3
12.	Security	12-1
12.1	Introduction	12-1
12.2	Declarative Security	12-2
12.3	Programmatic Security	12-2
12.4	Roles	12-3
12.5	Authentication	12-4
12.5.1	HTTP Basic Authentication	12-4
12.5.2	HTTP Digest Authentication	12-5
12.5.3	Form-Based Authentication	12-5
12.5.3.1	Login Form Notes	12-6
12.6	Server Tracking of Authentication Information	12-6
12.7	Specifying Security Constraints	12-7
12.7.1	Combining Constraints	12-8
12.7.2	Example of Applicable Constraints	12-9
12.7.3	Processing Requests	12-11

- 12.8 Default Policies 12-11
- 12.9 Login and Logout 12-12

13. Deployment Descriptor 13-1

- 13.1 Deployment Descriptor Elements 13-1
- 13.2 Rules for Processing the Deployment Descriptor 13-2
- 13.3 Deployment Descriptor 13-3
- 13.4 Deployment Descriptor Element Structure 13-3
 - 13.4.1 web-app Element 13-4
 - 13.4.2 description Element 13-5
 - 13.4.3 display-name Element 13-5
 - 13.4.4 icon Element 13-6
 - 13.4.5 context-param Element 13-6
 - 13.4.6 filter Element 13-6
 - 13.4.7 filter-mapping Element 13-7
 - 13.4.8 listener Element 13-8
 - 13.4.9 servlet Element 13-8
 - 13.4.10 servlet-mapping Element 13-9
 - 13.4.11 session-config Element 13-10
 - 13.4.12 mime-mapping Element 13-10
 - 13.4.13 welcome-file-list Element 13-10
 - 13.4.14 error-page Element 13-11
 - 13.4.15 security-constraint Element 13-11
 - 13.4.16 login-config Element 13-12
 - 13.4.17 security-role Element 13-13
 - 13.4.18 locale-encoding-mapping-list Element 13-13
- 13.5 Examples of Deployment Descriptor Usage 13-14
 - 13.5.1 A Basic Example 13-14
 - 13.5.2 An Example of Security 13-16

Glossary Glossary-1

Index Index-1

Figures

FIGURE 13-1	Conventions Used in Diagrams of Elements	13-4
FIGURE 13-2	web-app Element Structure	13-5
FIGURE 13-3	icon Element Structure	13-6
FIGURE 13-4	context-param Element Structure	13-6
FIGURE 13-5	filter Element Structure	13-7
FIGURE 13-6	filter-mapping Element Structure	13-7
FIGURE 13-7	listener Element Structure	13-8
FIGURE 13-8	servlet Element Structure	13-9
FIGURE 13-9	servlet-mapping Element Structure	13-9
FIGURE 13-10	session-config Element Structure	13-10
FIGURE 13-11	mime-mapping Element Structure	13-10
FIGURE 13-12	welcome-file-list Element Structure	13-10
FIGURE 13-13	error-page Element Structure	13-11
FIGURE 13-14	security-constraint Element Structure	13-12
FIGURE 13-15	login-config Element Structure	13-13
FIGURE 13-16	security-role Element Structure	13-13
FIGURE 13-17	locale-encoding-mapping-list Element Structure	13-14

Tables

TABLE 4-1	Example Context Setup	4–4
TABLE 4-2	Observed Path Element Behavior	4–5
TABLE 4-3	Protocol Attributes	4–6
TABLE 9-1	Request Attributes and Their Types	9–5
TABLE 10-1	Events and Listener Interfaces	10–2
TABLE 11-1	Example Set of Maps	11–3
TABLE 11-2	Incoming Paths Applied to Example Maps	11–3
TABLE 12-1	Security Constraint Table	12–10

Code Examples

- [CODE EXAMPLE 10-1](#) Deployment Descriptor Example 10-4
- [CODE EXAMPLE 12-1](#) Example of Applicable Constraints 12-9
- [CODE EXAMPLE 13-1](#) `locale-encoding-mapping-list` Element Code 13-13
- [CODE EXAMPLE 13-2](#) Deployment Descriptor 13-14
- [CODE EXAMPLE 13-3](#) Security Related Deployment Descriptor 13-16

Preface

This book provides a specification of the Java™ Servlet API for the Java Card™ 3 Platform, Connected Edition. This specification is a subset of the Java Servlet Specification, Version 2.4. For information on the differences between the two specifications, see [Section 1.4, “Compatibility With Java Servlet Specification Version 2.4” on page 1-3](#). In this book, Java Card 3 Platform refers to both versions 3.0 and 3.0.1 to distinguish them from all earlier versions.

Before You Read This Specification

To fully use the information in this document, you must have thorough knowledge of the topics discussed in the *Java Servlet Specification, Version 2.4*.

Before reading this guide, you should be familiar with the Java programming language, the other Java Card technology specifications, and smart card technology. A good resource for becoming familiar with Java technology and Java Card technology is the Sun Microsystems, Inc. web site, located at

<http://java.sun.com>

You should also be familiar with the Java Card technology web site at

<http://java.sun.com/products/javacard/>

How This Document Is Organized

[Chapter 1](#) describes the architecture of the Java Card 3 Platform, Connected Edition.

[Chapter 2](#) briefly describes the two application models.

[Chapter 3](#) describes the web application environment in more depth.

[Chapter 4](#) describes the APDU-based application environment in more depth.

[Chapter 5](#) describes card initialization and start-up.

[Chapter 6](#) describes security and access control mechanisms.

[Chapter 7](#) describes inter-application communication.

[Chapter 8](#) describes card management.

[Chapter 9](#) describes web applications.

[Chapter 10](#) describes events.

[Chapter 11](#) describes request mappings.

[Chapter 12](#) describes security issues.

[Chapter 13](#) describes how to use deployment descriptors.

[Glossary](#) provides definitions of selected terms used in the entire Connected Edition.

Using UNIX Commands

This document might not contain information about basic UNIX® commands and procedures such as shutting down the system, booting the system, and configuring devices. Refer to the following for this information:

- Software documentation that you received with your system
- Solaris™ Operating System documentation, which is at:

<http://docs.sun.com>

Shell Prompts

Shell	Prompt
C shell	<i>machine-name%</i>
C shell superuser	<i>machine-name#</i>
Bourne shell and Korn shell	\$
Bourne shell and Korn shell superuser	#

Typographic Conventions

Typeface*	Meaning	Examples
AaBbCc123	The names of commands, files, and directories; on-screen computer output	Edit your <code>.login</code> file. Use <code>ls -a</code> to list all files. <code>% You have mail.</code>
AaBbCc123	What you type, when contrasted with on-screen computer output	<code>% su</code> <code>Password:</code>
<i>AaBbCc123</i>	Book titles, new words or terms, words to be emphasized. Replace command-line variables with real names or values.	Read Chapter 6 in the <i>User's Guide</i> . These are called <i>class</i> options. You <i>must</i> be superuser to do this. To delete a file, type <code>rm filename</code> .

* The settings on your browser might differ from these settings.

Related Documentation

The following documents might be of interest.

- *Java Card Runtime Environment Specification, Java Card Platform, v3.0.1, Connected Edition* containing the runtime environment specification for the Connected Edition. In this book, “Java Card RE” is used to refer to the runtime environment in the Connected Edition.
- *Application Programming Interface Specification, Java Card Platform, Version 3.0.1, Connected Edition* containing full specifications of classes, interfaces, and method signatures that define the Java Servlet API for the Java Card Platform, Connected Edition, in Javadoc™ tool file format.

This specification is intended to be a complete and clear explanation of Java Servlets for the Java Card Platform, but if questions remain, the following sources may be consulted:

- A reference implementation (RI) has been made available by Sun Microsystems, Inc. that provides a behavioral benchmark for this specification. Where the specification leaves implementation of a particular feature open to interpretation, implementers may use the reference implementation as a model of how to carry out the intention of the specification.
- A Technology Compatibility Kit (TCK) suite has been provided by Sun Microsystems, Inc. for assessing whether implementations meet the compatibility requirements of the Java Servlet API specification for the Java Card Platform. The test results have normative value for resolving questions about whether an implementation is standard.

Documentation, Support, and Training

Sun Function	URL
Documentation	http://www.sun.com/documentation/
Support	http://www.sun.com/support/
Training	http://www.sun.com/training/

Third-Party Web Sites

Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with the use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

The following specifications provide information relevant to the development and implementation of the Java Servlet API and standard servlet engines. Online versions of the following RFCs are at <http://www.ietf.org/rfc/>.

- RFC 1630 Uniform Resource Identifiers (URI)
- RFC 1738 Uniform Resource Locators (URL)
- RFC 3986 Uniform Resource Identifiers (URI): Generic Syntax
- 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 2616 Hypertext Transfer Protocol (HTTP/1.1)
- RFC 2617 HTTP Authentication: Basic and Digest Authentication
- RFC 2818 HTTP Over TLS

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/>

Sun Welcomes Your Comments

Sun Microsystems is interested in improving its documentation and welcomes your comments and suggestions. You can submit your comments to jc-bandol-spec-feedback@sun.com.

Please include the title of your document with your feedback:

Java Servlet Specification, Java Card Platform, Version 3.0.1, Connected Edition.

Architecture Overview

This book is targeted for the Connected Edition. The Java Card 3 Platform consists of two editions.

- The Classic Edition is based on an evolution of the Java Card Platform, Version 2.2.2 and is backward compatible with it, targeting resource-constrained devices that solely support applet-based applications. You may disregard the specifications for the Connected Edition if you are interested in the functionality found only in the Classic Edition.
- The Connected Edition features a significantly enhanced runtime environment and a new virtual machine. It includes new network-oriented features, such as support for web applications, including the Java™ Servlet APIs, and also support for applets with extended and advanced capabilities. An application written for or an implementation of the Connected Edition may use features found in the Classic Edition. Therefore, you will need to use the specifications for both the Classic Edition and the Connected Edition.

This chapter introduces the use of servlets and servlet containers in the Java Card 3 Platform, Connected Edition.

1.1 Definition of 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 Definition of 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 life cycle.

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 Card Platform environment, these restrictions should be placed using the permission architecture. For example, some environments may limit the creation of a `Thread` object to insure that other components of the container are not negatively impacted.

1.3 Example of Servlet Invocation

The following is a typical sequence of interactions that occur when a servlet is invoked:

1. A client (for example, 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.
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 Compatibility With Java Servlet Specification Version 2.4

The Java Servlet API for the Java Card Platform is a subset of the Java Servlet API v2.4. This section describes this subset and the compatibility issues introduced in the Java Card Platform version of the specification.

1.4.1 Features Not In The Java Card Platform Version

The following features that were in the Java Servlet Specification, v2.4, are not supported by this specification and have been removed from it:

- Dependencies on unsupported APIs (`java.io.File`, `java.net.URL`, `java.util.Map`, `java.util.Set`, `java.security.cert.X509Certificate`...)
- Dependencies on floating point support
- Dependencies on serialization and cloning support (`java.io.Serializable`, `java.lang.Cloneable`)
- Configuration of filters that are invoked under request dispatcher forward and include calls or under the error page mechanism
- Support for distributed container and session migration
- Some of the programmatic and declarative security features
- Support for deployment in web containers that are JavaServer Pages™ (JSP™) enabled or part of a Java™ Platform, Enterprise Edition (“Java EE™”) application server, especially web application environment and dependencies on other Java platform specifications:
 - Java EE, version 1.4
 - Java Server Pages (JSP), version 2.0

- Java Naming and Directory Interface™ (“J.N.D.I.”).
- All deprecated features

1.4.2 Backward Compatibility Support of Web Applications

Servlet containers implementing this specification are not required to support backward compatibility of applications written to any prior version of the Servlet API, such as the 2.2 and 2.3 versions of the API.

Servlet containers implementing this specification must support applications written to the Servlet API, version 2.4, provided they do not use any of the removed features listed below. Such applications may be developed and packaged using tools such as Integrated Development Environments (IDE) supporting the Servlet API, version 2.4.

1.4.3 Temporary Working Directories

In this version of the specification, a servlet container is not required to provide a private, temporary directory for each servlet context and make it available via the `javax.servlet.context.tempdir` context attribute. The `javax.servlet.context.tempdir` context attribute was therefore removed.

1.4.4 SSL Attributes

In this version of the specification, the `javax.servlet.request.X509Certificate` request attribute used for retrieving the SSL certificate associated with a request as an array of objects of type `java.security.cert.X509Certificate` was removed.

Developers must instead use the `javax.servlet.request.X509Certificate` request attribute. The object associated with this attribute is an object of type `javax.microedition.pki.Certificate` which corresponds to the web client’s subject certificate.

1.4.5 Removed Elements of the Deployment Descriptor

The following sections describe the elements removed from the *Java Servlet Specification, Version 2.4* when creating this version of the specification, the *Java Servlet Specification for the Java Card Platform, Version 3.0.1, Connected Edition*.

1.4.5.1 Removed Elements for Deployment in Web Container JSP Pages Enabled or Part of a Java EE Application Server

The following elements, which exist in the web application deployment descriptor to meet the requirements of web containers that are JSP pages enabled or are part of a Java EE application server, were removed from this specification. They are not required to be supported by containers wishing to support only the servlet specification.

- Syntax for JSP configuration (`jsp-config`, `jsp-file`)
- Syntax for looking up JNDI objects (`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`)

1.4.5.2 `distributable` Element Removed

The following element was removed from the deployment descriptor schema in this version of the specification:

- The `distributable` element indicates that a web application is programmed appropriately to be deployed into a distributed servlet container.

This specification does not support session migration or distributed containers.

1.4.5.3 `run-as` Element Removed

The following element (part of the declarative security features) was removed from the deployment descriptor schema in this version of the specification:

- The `run-as` element specifies the identity to be used for the execution of a component (for example, a servlet).

1.4.5.4 dispatcher Element Removed

The following element was removed from the deployment descriptor schema in this version of the specification:

- The `dispatcher` element has four legal values: `FORWARD`, `REQUEST`, `INCLUDE`, and `ERROR` which indicate respectively that a Filter will be applied under `RequestDispatcher.forward()` calls, under ordinary client calls to the path or servlet, under `RequestDispatcher.include()` calls or under the error page mechanism. The absence of any dispatcher elements in a filter-mapping indicates a default of applying filters only under ordinary client calls to the path or servlet.

In this version of the specification, filters are only applied under ordinary client calls to the path or servlet.

1.4.5.5 auth-method Element Value CLIENT-CERT Removed

The following element value was removed from the deployment descriptor schema in this version of the specification:

- The `CLIENT-CERT` content in a sub-element `auth-method` element configures the authentication mechanism for the web application to use HTTPS (HTTP over SSL) client certificates for end user authentication.

1.4.6 realm-name Element Extended Applicability

The `realm-name` element may now be used for all authentication schemes and not just for BASIC authentication. The `realm-name` indicates the realm name to use for the authentication scheme chosen for the Web application.

1.4.7 Filtering of Authorization Request Header

This version of the specification requires that when an application is configured for container-managed HTTP Basic or Digest authentication, the web container filters out `Authorization` request headers so that they are not accessible to the application.

1.4.8 Web Application Deployment Hierarchy and Directory Structure

The following subdirectory of the web application directory structure is not supported in this version of the specification:

- The `/WEB-INF/lib/*.jar` area for Java™ Archive (JAR) files. These files contain servlets, beans, 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.

Web applications depending on libraries external to the Web Archive (WAR) file must use an alternate deployment mechanism specific to the Java Card Platform, Version 3, Connected Edition.

The web container is not required to recognize the dependencies of a web application on specific versions of libraries external to the Web Archive (WAR), expressed in the manifest entry of the WAR following the rules defined by the Optional Package Versioning mechanism (<http://java.sun.com/j2se/1.4/docs/guide/extensions/>).

The contents of the `WEB-INF/classes`, `WEB-INF/lib` and `META-INF` directories, and the deployment descriptor `WEB-INF/web.xml` are not visible to servlet code using the `getResourceAsStream` method calls on the `ServletContext`, and are not exposed using the `RequestDispatcher` calls.

1.4.9 Usage of URL Patterns for Servlet Mapping, Filter Mapping and Security Constraints Restricted

This Java Card Platform version of the specification imposes the following restriction on the use of `url-pattern` for servlet mapping: the `url-pattern` values of the `servlet-mapping` elements must not overlap.

This Java Card Platform version of the specification imposes the following restriction on the use of `url-pattern` for filter mapping and security-constrained web resource collections: the `url-pattern` value for filter mapping and security-constrained web resource collections must exactly correspond to the `url-pattern` of one of the `servlet-mapping` elements defined for mapping request URI to servlets in the deployment descriptor.

The web container on a Java Card Platform implementation must reject any application that declares:

- values of `servlet-mapping/url-pattern` elements that overlap;
- a value of a `filter-mapping/url-pattern` element that does not correspond to one of the declared values of `servlet-mapping/url-pattern` elements;

- a value of a `security-constraint/web-resource-collection/url-pattern` element that does not correspond to one of the declared values of `servlet-mapping/url-pattern` elements.

1.4.10 Combination of Security Constraints Precluded

This Java Card Platform version of the specification imposes the following restriction on security constraints: the same `url-pattern` and `http-method` value pair must not appear in multiple constraints. This restriction also applies to security constraints which do not have an `http-method` element as it stands for all the possible values of the `http-method` element.

The web container on a Java Card Platform implementation must reject applications declaring in their deployment descriptors multiple security constraints with the same `url-pattern` and `http-method` value pair.

1.4.11 Default Servlet Implementation

In this Java Card Platform version of the specification, the `doOptions` and `doTrace` methods respond by default with an HTTP status code 501 (Not implemented). These methods must be explicitly implemented by servlet developers who want to support these features.

1.4.12 Classes, Interfaces and Methods Depending on API Not Supported on the Java Card Platform

Note that the following removals cause source incompatibility in some cases, such as when a developer uses these methods in a Servlet API v2.4 development environment and intends to deploy on the Java Card Platform.

1.4.12.1 `java.io.Serializable` Implementation Removed

The following classes are not serializable in this version of the specification:

- `public abstract class GenericServlet` implements `javax.servlet.Servlet`, `javax.servlet.ServletConfig`
- `public abstract class HttpServlet` extends `javax.servlet.GenericServlet`

1.4.12.2 `java.lang.Cloneable` Implementation Removed

The following class is not cloneable in this version of the specification:

```
public class Cookie
```

1.4.12.3 `HttpSessionActivationListener` Interface Removed

The following interface was removed in this version of the specification:

```
public interface HttpSessionActivationListener extends  
java.util.EventListener
```

Objects that are bound to a session may listen to container events notifying them that sessions will be made passive and that session will be activated. A container that migrates session between VMs or persists sessions is required to notify all attributes bound to sessions implementing `HttpSessionActivationListener`.

This specification does not support session migration or distributed containers.

1.4.12.4 `ServletContext` methods `getResource` and `getResourcePaths` Removed

The following methods were removed from the `ServletContext` interface in this version of the specification:

- `public java.net.URL getResource(java.lang.String path)`
throws `MalformedURLException`

Returns a `URL` to the resource that is mapped to a specified path. The path must begin with a `"/"` and is interpreted as relative to the current context root.

- `public java.util.Set getResourcePaths(java.lang.String path)`

Returns a directory-like listing of all the paths to resources within the web application whose longest sub-path matches the supplied path argument.

1.4.12.5 `ServletRequest` and `ServletRequestWrapper` method `getParameterMap` Removed

The following method was removed from the `ServletRequest` interface and from the `ServletRequestWrapper` class in this version of the specification:

- `public java.util.Map getParameterMap()`

Returns a `java.util.Map` of the parameters of this request. Request parameters are extra information sent with the request.

1.4.12.6 `getUserPrincipal` method of `HttpServletRequest` and `HttpServletRequestWrapper` Removed

The following method (part of the programmatic security features) was removed from the `HttpServletRequest` interface and from the `HttpServletRequestWrapper` class in this version of the specification:

- `public java.security.Principal getUserPrincipal()`
Returns a `java.security.Principal` object containing the name of the current authenticated user.

1.4.12.7 `ServletOutputStream` methods `print(float)`, `print(double)`, `println(float)` and `println(double)` Removed

The following methods were removed from the `ServletOutputStream` class in this version of the specification:

- `public void print(double d) throws IOException`
Writes a `double` value to the client, with no carriage return-line feed (CRLF) at the end.
- `public void print(float f) throws IOException`
Writes a `float` value to the client, with no carriage return-line feed (CRLF) at the end.
- `public void println(double d) throws IOException`
Writes a `double` value to the client, followed by a carriage return-line feed (CRLF).
- `public void println(float f) throws IOException`
Writes a `float` value to the client, followed by a carriage return-line feed (CRLF).

1.4.13 Removed Classes, Interfaces and Methods Deprecated in Java Servlet API Specification Version 2.4

Note that the following removals cause source incompatibility in some cases, such as when a developer uses these methods in a Servlet API v2.4 development environment (where this deprecated classes, interfaces and methods may still be supported) and intends to deploy on the Java Card Platform.

1.4.13.1 SingleThreadModel Interface Removed

The following deprecated interface was removed in this version of the specification:

- `public interface SingleThreadModel`

Deprecated- As of Java Servlet API v2.4, with no direct replacement. Ensures that servlets handle only one request at a time. This interface has no methods.

1.4.13.2 HttpSessionContext Interface Removed

The following deprecated interface was removed in this version of the specification:

- `public interface HttpSessionContext`

Deprecated- As of Java Servlet API v2.1 for security reasons, with no replacement.

1.4.13.3 HttpUtils Class Removed

The following deprecated class was removed in this version of the specification:

- `public class HttpUtils`

Deprecated- As of Java Servlet API v2.3. These methods were only useful with the default encoding and have been moved to the request interfaces.

1.4.13.4 ServletContext methods getServlet, getServletNames, getServlets, and log Removed

The following deprecated methods were removed from the `ServletContext` interface in this version of the specification:

- `public Servlet getServlet(java.lang.String name
throws ServletException`

Deprecated- As of Java Servlet API v2.1, with no direct replacement. This method was originally defined to retrieve a servlet from a `ServletContext`.

- `public java.util.Enumeration getServletNames()`

Deprecated- As of Java Servlet API v2.1, with no replacement. This method was originally defined to return an `Enumeration` of all the servlet names known to this context.

- `public java.util.Enumeration getServlets()`

Deprecated- As of Java Servlet API v2.0, with no replacement. This method was originally defined to return an `Enumeration` of all the servlets known to this servlet context.

- `public void log(java.lang.Exception exception, java.lang.String msg)`

Deprecated- As of Java Servlet API v2.1, use `log(String, Throwable)` instead. This method was originally defined to write an exception's stack trace and an explanatory error message to the servlet log file.

1.4.13.5 `getRealPath` method of `ServletRequest` and `ServletRequestWrapper` Removed

The following deprecated method was removed from the `ServletRequest` interface and from the `ServletRequestWrapper` class in this version of the specification:

- `public java.lang.String getRealPath(java.lang.String path)`

Deprecated- As of Version 2.1 of the Java Servlet API, use `ServletContext.getRealPath(String)` instead.

1.4.13.6 `UnavailableException` constructors and method `getServlet` Removed

The following deprecated constructors and method were removed from the `UnavailableException` class in this version of the specification:

- `public UnavailableException(int seconds, Servlet servlet, java.lang.String msg)`

Deprecated- As of Java Servlet API v2.2, use `UnavailableException(String, int)` instead.

- `public UnavailableException(Servlet servlet, java.lang.String msg)`

Deprecated- As of Java Servlet API v2.2, use `UnavailableException(String)` instead.

- `public Servlet getServlet()`

Deprecated- As of Java Servlet API v2.2, with no replacement. Returns the servlet that is reporting its unavailability.

1.4.13.7 `isRequestedSessionIdFromUrl` method of `HttpServletRequest` and `HttpServletRequestWrapper` Removed

The following deprecated method was removed from the `HttpServletRequest` interface and from the `HttpServletRequestWrapper` class in this version of the specification:

- `public boolean isRequestedSessionIdFromUrl()`
Deprecated- As of Version 2.1 of the Java Servlet API, use `isRequestedSessionIdFromURL()` instead.

1.4.13.8 `HttpServletResponse` and `HttpServletResponseWrapper` methods `encodeRedirectUrl`, `encodeUrl` and `setStatus` Removed

The following deprecated methods were removed from the `HttpServletResponse` interface and from the `HttpServletResponseWrapper` class in this version of the specification:

- `public java.lang.String encodeRedirectUrl(java.lang.String url)`
Deprecated- As of version 2.1, use `encodeRedirectURL(String url)` instead.
- `public java.lang.String encodeUrl(java.lang.String url)`
Deprecated- As of version 2.1, use `encodeURL(String url)` instead.
- `public void setStatus(int sc, java.lang.String sm)`
Deprecated- As of version 2.1, due to ambiguous meaning of the message parameter. To set a status code use `setStatus(int)`, to send an error with a description use `sendError(int, String)`. Sets the status code and message for this response.

1.4.13.9 `HttpSession` methods `getSessionContext`, `getValue`, `getValueNames`, `putValue` and `removeValue` Removed

The following deprecated methods were removed from the `HttpSession` interface in this version of the specification:

- `public HttpSessionContext getSessionContext()`
Deprecated- As of Version 2.1, this method is deprecated and has no replacement. It will be removed in a future version of the Java Servlet API.

- `public java.lang.Object getValue(java.lang.String name)`
Deprecated- As of Version 2.2, this method is replaced by `getAttribute(String)`.
- `public java.lang.String[] getValueNames()`
Deprecated- As of Version 2.2, this method is replaced by `getAttributeNames()`.
- `public void putValue(java.lang.String name, java.lang.Object value)`
Deprecated- As of Version 2.2, this method is replaced by `setAttribute(String, Object)`.
- `public void removeValue(java.lang.String name)`
Deprecated- As of Version 2.2, this method is replaced by `removeAttribute(String)`.

1.4.13.10 Attributes Deprecated or Redundant in Java Servlet API Specification Version 2.4

The following attributes have been removed.

Error Handling Request Attributes exception-type and message Removed

In this version of the specification, the request attributes `javax.servlet.error.exception_type` and `javax.servlet.error.message` are not set by the container when the location of the error handler is a servlet. These attributes are redundant with the `javax.servlet.error.exception` request attribute.

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
- `doPut` for handling HTTP `PUT` requests

- `doDelete` for handling HTTP `DELETE` requests
- `doHead` for handling HTTP `HEAD` requests
- `doOptions` for handling HTTP `OPTIONS` requests
- `doTrace` for handling HTTP `TRACE` requests

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.

Caution – In this Java Card Platform version of the specification, the `doOptions` and `doTrace` methods respond by default with an HTTP status code 501 (Not implemented). These methods must be explicitly implemented by servlet developers who want to support these features.

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 part of the deployment descriptor of the web application containing the servlet, as described in [Chapter 13](#) controls how the servlet container provides instances of the servlet.

The servlet container must use only one instance per servlet declaration.

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.

Refer to the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for more details on the lifetime of servlet objects specific to the Java Card Platform.

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.

When the servlet engine is started, needed servlet classes must be located by the servlet container. The servlet container loads the servlet class using the class loading facilities of the runtime environment.

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, 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 3](#) 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 other services 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 a 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.

If the `init` method (or methods such as `doGet` or `doPost` which are dispatched to the `service` method of the `HttpServletRequest` abstract class) has been defined with the `synchronized` keyword, the servlet container cannot use an 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. It is recommended that a developer take other means to resolve those issues instead of synchronizing the `service` method, such as avoiding the usage of an instance variable or synchronizing the block of the code accessing those resources.

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 Thread Safety

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

References to the request and response objects should not be given to objects executing in other threads as the resulting behavior may be nondeterministic. If the 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 servlet's service life cycle and such a thread itself should have a life cycle within the life cycle of the servlet's `service` method because accessing those objects after the `service` method ends may cause undeterminable problems. Because 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 the wrapper to add the thread safety by, for instance, synchronizing the call of the methods to access the request attribute or by using a local output stream for the response object within a thread.

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 unloading an application.

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.

Servlet Context

3.1 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://<servername>/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`.

3.2 Scope of a ServletContext Interface

One instance object of the `ServletContext` interface is associated with each web application deployed into a container.

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`.

3.3 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. A typical example is the name of another peer system with which the application may interact.

3.4 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 the `ServletContext` interface allow access to this functionality:

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

3.5 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 `getResourceAsStream` method of the `ServletContext` interface.

The `getResourceAsStream` method takes a `String` with a leading `"/` as an argument that gives the path of the resource relative to the root of the context. This hierarchy of documents may exist in the server's file system, in a web application archive file, or at some other location.

This method is not used to obtain dynamic content. See [Chapter 8](#) for more information about accessing dynamic content.

3.6 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 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.

4.1 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 [Section 4.1.1, “When Parameters Are Available” on page 4-2](#) 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`

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 `getParameterNames` method returns a `java.util.Enumeration` of all the parameter names of the request.

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.

4.1.1 When Parameters Are Available

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

- The request is an HTTP or HTTPS request.
- The HTTP method is POST.
- The content type is `application/x-www-form-urlencoded`.
- 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 not be available for reading directly from the request object's input stream.

4.2 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 “javacard.” and “javacardx.”, “java.” and “javax.” are reserved for definition by this specification. Similarly, attribute names beginning with the prefixes of “sun.”, and “com.sun.” are reserved for definition by Sun Microsystems. It is suggested that all attributes placed in the attribute set be named in accordance with the reverse domain name convention for package naming suggested by *The Java Programming Language Specification*, available at <http://java.sun.com/docs/books/jls/>.

4.3 Headers

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

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

The `getHeader` method returns a header given the name of the header. There can be multiple headers with the same name, for example `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.

4.4 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 “/*” pattern, in which case it is an empty string.
- **Path Info:** 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:

$$\text{requestURI} = \text{contextPath} + \text{servletPath} + \text{pathInfo}$$

To give a few examples to clarify the above points, consider the setup shown in [TABLE 4-1](#).

TABLE 4-1 Example Context Setup

Context Path	/catalog
Servlet Mapping	Pattern: /lawn/* Servlet: LawnServlet
Servlet Mapping	Pattern: /garden/* Servlet: GardenServlet
Servlet Mapping	Pattern: *.shtml Servlet: SHTMLServlet

The behavior based on the context setup shown in [TABLE 4-1](#) results in the path elements shown in [TABLE 4-2](#).

TABLE 4-2 Observed Path Element Behavior

Request Path	Path Elements
/catalog/lawn/index.html	ContextPath: /catalog ServletPath: /lawn PathInfo: /index.html
/catalog/garden/implements/	ContextPath: /catalog ServletPath: /garden PathInfo: /implements/
/catalog/help/feedback.shtml	ContextPath: /catalog ServletPath: /help/feedback.shtml PathInfo: null

4.5 Path Translation Methods

There are two convenience methods in the API that 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, or an in-memory representation, these methods must return `null`.

4.6 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.



4.7 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 attributes listed in [TABLE 4-3](#) to the servlet programmer.

TABLE 4-3 Protocol Attributes

Attribute	Attribute Name	Java Type
cipher suite	<code>javax.servlet.request.cipher_suite</code>	<code>String</code>
bit size of the algorithm	<code>javax.servlet.request.key_size</code>	<code>Integer</code>

In this Java Card Platform version of the specification, if there is an SSL certificate associated with the request, it must be exposed by the servlet container to the servlet programmer as an object of type `javax.microedition.pki.Certificate` and accessible via a `ServletRequest` attribute of `javacardx.servlet.request.X509Certificate`. This certificate corresponds to the web client’s subject certificate.

In this Java Card Platform version of the specification, if there is a TLS PSK identity associated with the request, it must be exposed by the servlet container to the servlet programmer as an object of type `java.lang.String` and accessible via a `ServletRequest` attribute of `javacardx.servlet.request.PSKIdentity`.



4.8 Internationalization

Clients may optionally indicate to a web server the language in which they would prefer the response be given. 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.

4.9 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.

Breakage can occur if the client has not set character encoding and the request data is encoded with a different encoding than the default as described above. 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.

4.10 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. Containers commonly recycle request objects in order to avoid the performance overhead of request object creation. The developer must note that maintaining references to request objects outside the scope described above is not recommended as it may have indeterminate results.

Refer to the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for more details on the lifetime of request objects specific to the Java Card Platform.

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`. If the servlet container does not support buffering and the size requested is greater than 0 (zero), this method must throw an `IllegalArgumentException`.

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 and status codes 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 is sent to the client, the response is considered to be committed.

5.2 Headers of an HTTP Response

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 represent 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/2.4"`. 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.

Two examples of this header are:

`X-Powered-By: Servlet/2.4`

`X-Powered-By: Servlet/2.4 (JCRE/3.0.1)`

5.3 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 (meaning 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`.

5.4 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. However, 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.

5.5 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` method of the response has been written to the response.
 - The `sendError` method is called.
 - The `sendRedirect` method is called.
-

5.6 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. Containers commonly recycle response objects in order to avoid the performance overhead of response object creation. The developer must note that maintaining references to response objects outside the scope described above may lead to non-deterministic behavior.

Refer to the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for more details on the lifetime of response objects specific to the Java Card Platform.

Filtering

Filters are Java programming language components that, during runtime, allow transformations of payload and header information in both the request into a resource and the response from a resource.

This chapter describes 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 separately in Javadoc tool format. The configuration syntax for filters is given by the deployment descriptor schema in [Chapter 13](#). The reader should use these sources as references when reading this chapter.

This Java Card Platform version of the specification imposes that filters be mapped only to URL patterns to which servlets have been mapped. Therefore, filters cannot directly apply to static content. Filters can only apply indirectly to static content through a servlet such as an explicitly declared “default” servlet that is used to serve static content in response to requests from web clients when no other servlet applies. This indirect mapping is assumed for applying filters to static content.

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.

Some of the types of filtering functionality available 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 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 of Filtering

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 by the URL pattern the servlet is mapped to.

6.2.1 Filter Life Cycle

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 by 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 `ServletResponse` or `HttpServletResponse` 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.

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 `doFilter` 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.

Refer to the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for more details on the lifetime of filter objects specific to the Java Card Platform.

6.2.2 Wrapping Requests and Responses

Central to the notion of filtering is the concept of wrapping a request or response so 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 APIs 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 than the output stream or the writer.

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.

6.2.4 Configuration of Filters in a Web Application

A filter is defined in the deployment descriptor using the `<filter>` element. In this element, the programmer declares the following:

- **filter-name**: used to map the filter to a 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:

```
<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 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:

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

Filters can be associated with a servlet using the `<url-pattern>` style of filter mapping:

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

Caution – This Java Card Platform version of the specification imposes the following restriction on the use of the `<url-pattern>` style of filter mapping: the `<url-pattern>` value of a `<filter-mapping>` element must exactly correspond to the `<url-pattern>` value of one of the `<servlet-mapping>` elements defined for mapping request URI to servlets; moreover the `<url-pattern>` values of the `<servlet-mapping>` elements must not overlap. Because of this restriction, in the example above, the Logging Filter does not apply to all the servlets and static content pages in the web application but only to the content that is served by the servlet mapped to 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 11](#).

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.**

This requirement means that the container, when receiving an incoming request, processes the request as follows:

- Identifies the target web resource according to the rules of [Section 11.2, "Specification of Mappings" on page 11-2](#).
- 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 11.2, "Specification of Mappings" on page 11-2](#), 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, if there are none, invokes the target web resource.

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

Filters are only applied to a request when they come directly from the client.

Sessions

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 name of the session tracking cookie must be `JSESSIONID`.

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
```

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, cannot, or will not join a session.

Refer to the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for more details on the lifetime of session objects specific to the Java Card Platform.

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.

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 timeout period.

The default timeout period for sessions is defined by the servlet container and can be obtained via the `getMaxInactiveInterval` method of the `HttpSession` interface. This timeout can be changed by the developer using the `setMaxInactiveInterval` method of the `HttpSession` interface. The timeout periods used by these methods are defined in seconds. By definition, if the timeout period for a session is set to `-1`, 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 a single session object at the same time. The developer has the responsibility for synchronizing access to session resources as appropriate.

7.7.2 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.

Dispatching Requests

When building a web application, it is often useful to forward the 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.

8.1 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 `"/"`. The method uses the path to look up a servlet, using the servlet path matching rules in [Chapter 11](#), 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. If no resource is associated with the given path, the method must return `null`.

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")`.

8.1.1 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:

```
String path = "/raisins?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.

8.2 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. 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 as the original request.

8.3 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. Any attempt to do so must be ignored.

8.3.1 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 information, 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.

8.4 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, the response content must be sent, committed, and closed by the servlet container.

8.4.1 Query String

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

8.4.2 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:

- `javax.servlet.forward.request_uri`
- `javax.servlet.forward.context_path`
- `javax.servlet.forward.servlet_path`
- `javax.servlet.forward.path_info`
- `javax.servlet.forward.query_string`

The values of these attributes must be equal to the return values of the `HttpServletRequest` methods `getRequestURI`, `getContextPath`, `getServletPath`, `getPathInfo`, `getQueryString` respectively, invoked on the request object passed to the first servlet object in the 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.

8.5 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 `ServletExceptions` and the root cause of the exception set to the original exception because it should not be propagated.

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.

9.1 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://<servername>/catalog`. All requests that start with this prefix will be routed to the `ServletContext` that represents the catalog application.

9.2 Relationship to ServletContext

The servlet container must enforce a one-to-one correspondence between a web application and a `ServletContext` object. A `ServletContext` object provides a servlet with its view of the application.

9.2.1 Elements of a Web Application

A web application may consist of the following items:

- Servlets
- Utility Classes
- Static documents (HTML, images, sounds, etc.)

- Descriptive meta information that ties all of the above elements together.

9.3 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.

9.4 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 can be served to satisfy a request from `/catalog/index.html`. The rules for matching URLs to context path are laid out in [Chapter 11](#). 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 are not in the document root of the application. The `WEB-INF` node is not part of the public document tree of the application. No file contained in the `WEB-INF` directory may be served directly to a client by the container. However, the contents of the `WEB-INF` directory with the exception of the contents of `WEB-INF/classes`, `WEB-INF/lib` and the deployment descriptor `WEB-INF/web.xml` are visible to servlet code using the `getResourceAsStream` method calls on the `ServletContext`, and may be exposed using the `RequestDispatcher` calls. Therefore, if the application developer needs access from servlet code to application-specific configuration information that s/he does not wish to be exposed directly to the web client, s/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 the 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.

Any requests from the client to access the resources in the `WEB-INF/` directory must be returned with a `SC_NOT_FOUND(404)` response.

9.4.1 Example of Application Directory Structure

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

- `/index.html`
- `/howto.html`
- `/feedback.html`
- `/images/banner.gif`
- `/images/jumping.gif`
- `/WEB-INF/web.xml`
- `/WEB-INF/classes/com/mycorp/servlets/MyServlet.class`
- `/WEB-INF/classes/com/mycorp/util/MyUtils.class`

9.5 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 named `issuetrack.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, and its contents must not be visible to servlet code via the `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.

See the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for more information on the Java Card Platform-specific mechanism for deploying onto the Java Card Platform a web application's WAR file.

9.6 Web Application Deployment Descriptor

The web application deployment descriptor (see [Chapter 13](#)) includes the following types of configuration and deployment information:

- ServletContext Init Parameters
- Session Configuration
- Servlet Definitions
- Servlet Mappings
- MIME Type Mappings
- Welcome File list
- Error Pages
- Security

9.6.1 Dependencies On Libraries External to WAR File

When an application depends on code or resources external to the WAR file, such as when several applications may make use of the same code or resources, they will typically be installed as library files. These files are often common or standard APIs that can be used without sacrificing portability.

The mechanism used for deploying web applications and the libraries they depend on is described in the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition*.

9.6.2 Web Application Class Loader

Servlet containers should not allow applications to override system classes, such as those in the `java.*`, `javax.*`, `javacard.*` and `javacardx.*` namespaces, that the Java Card Platform does not allow to be modified. The container should not allow applications to override or access the container's implementation classes.

9.7 Error Handling

9.7.1 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, the following actions must occur:

- The original unwrapped request and response objects created by the container are passed to the servlet.
- The response `setStatus` method is disabled and ignored if called.
- The request path and attributes are set as if a `RequestDispatcher.forward` to the error resource had been performed.
- The request attributes in [TABLE 9-1](#) must be set.

TABLE 9-1 Request Attributes and Their Types

Request Attributes	Type
<code>javax.servlet.error.status_code</code>	<code>java.lang.Integer</code>
<code>javax.servlet.error.exception</code>	<code>java.lang.Throwable</code>
<code>javax.servlet.error.request_uri</code>	<code>java.lang.String</code>
<code>javax.servlet.error.servlet_name</code>	<code>java.lang.String</code>

These attributes allow the servlet to generate specialized content depending on the error code, 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.

9.7.2 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 error 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 uses the error-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:

- `RuntimeException` or errors
- `ServletExceptions` 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 `error-code` element must be unique in the deployment descriptor up to the error code.

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, by default - that is if no error code has already been set for the response.

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.

9.8 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 such as `host:port/webapp/directory/`, where `directory` is an entry in the WAR that is not mapped to a servlet, 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 or servlet in the WAR 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 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.html</welcome-file>
</welcome-file-list>
```

- The static content in the WAR is as follows
 - `/foo/index.html`
 - `/foo/default.html`
 - `/foo/orderform.html`

- /foo/home.gif
- /catalog/default.html
- /catalog/products/shop.html
- /catalog/products/register.html
- 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.html.
- 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, or may cause other behavior defined by the container. See [Section 11.2, “Specification of Mappings”](#) on page 11-2 for the definition of “default” servlet.

9.9 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.

1. Instantiate an instance of each event listener identified by a `<listener>` element in the deployment descriptor.
2. For instantiated listener instances that implement `ServletContextListener`, call the `contextInitialized()` method.
3. Instantiate an instance of each filter identified by a `<filter>` element in the deployment descriptor and call each filter instance’s `init()` method.
4. 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.

Application Life Cycle Events

The application events facility gives the web application developer greater control over the life cycle of the `ServletContext` and `HttpSession` and `ServletRequest`, allows for better code factorization, and increases efficiency in managing the resources that the web application uses.

10.1 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 platform 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 life cycle of servlet requests.

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.

10.1.1 Event Types and Listener Interfaces

Event types and the listener interfaces used to monitor them are shown in [TABLE 10-1](#).

TABLE 10-1 Events and Listener Interfaces

Event Type	Description	Listener Interface
<i>Servlet Context Events:</i>		
Lifecycle	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.	<code>javax.servlet. ServletContextListener</code>
Changes to attributes	Attributes on the servlet context have been added, removed, or replaced.	<code>javax.servlet. ServletContextAttributeListener</code>
<i>HTTP Session Events:</i>		
Lifecycle	An <code>HttpSession</code> has been created, invalidated, or timed out.	<code>javax.servlet.http. HttpSessionListener</code>
Changes to attributes	Attributes have been added, removed, or replaced on an <code>HttpSession</code> .	<code>javax.servlet.http. HttpSessionAttributeListener</code>
Object binding	Object has been bound to or unbound from <code>HttpSession</code> .	<code>javax.servlet.http. HttpSessionBindingListener</code>
<i>Servlet Request Events:</i>		
Lifecycle	A servlet request has started being processed by web components.	<code>javax.servlet. ServletRequestListener</code>
Changes to attributes	Attributes have been added, removed, or replaced on a <code>ServletRequest</code> .	<code>javax.servlet. ServletRequestAttributeListener</code>

10.1.2 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 common use of a service exposed by some other application through a service registry. The developer has provided a servlet context listener class for management of the lookup of the service in the registry.

1. When the application starts up, the listener class is notified. The application looks up the service in the service registry and stores the service client stub in the servlet context.
2. Servlets in the application access the service as needed during activity in the web application using the service client stub stored in the servlet context.
3. When the application is removed from the web server, the listener class is notified and the service client stub is released.

10.2 Listener Class Configuration

10.2.1 Provision of Listener Classes

The developer of the web application provides listener classes implementing one or more of the listener classes in the `javax.servlet` API. Each listener class must have a public constructor taking no arguments. The listener classes are packaged into the WAR, under the `WEB-INF/classes` archive entry.

10.2.2 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.

10.2.3 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 are invoked in the order of their registration.

10.2.4 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.

10.3 Deployment Descriptor Example

The following example is the deployment grammar for registering two servlet context life cycle listeners and a session listener.

Suppose that `com.acme.MyConnectionManager` and `com.acme.MyLoggingModule` both implement `javax.servlet.ServletContextListener`, and that `com.acme.MyLoggingModule` additionally implements `javax.servlet.HttpSessionListener`. Also, the developer wants `com.acme.MyConnectionManager` to be notified of servlet context life cycle events before `com.acme.MyLoggingModule`. [CODE EXAMPLE 10-1](#) shows the deployment descriptor for this application:

CODE EXAMPLE 10-1 Deployment Descriptor Example

```
<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>
```

10.4 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.

10.5 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 9.7.2, “Error Pages” on page 9-5](#). If there is no error page specified for those exceptions, the container must ensure that a response is sent 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.

10.6 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.

Mapping Requests To Servlets

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

11.1 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.

Caution – This Java Card Platform version of the specification imposes the following restriction on the use of `url-pattern` for servlet mapping: the `url-pattern` values of the `servlet-mapping` elements must not overlap. Therefore the web container on a Java Card Platform implementation must reject applications declaring in their deployment descriptors `url-pattern` values of `servlet-mapping` elements that overlap, such as the URL patterns `"/*` and `"/acme/wholesale/*"`.

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 (for example, .shtml), 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.
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.

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

11.2 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.
- 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 information is null.
- All other strings are used for exact matches only¹.

11.2.1 Implicit Mappings

A servlet container is allowed to make 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. If a *.shtml mapping is defined by the web application, its mapping takes precedence over the implicit mapping.

1. A servlet mapping string (other than an extension mapping string) not starting with a "/" will never be matched by a request URL as a servlet path must start with a "/".

11.2.2 Example Mapping Set

Consider the set of mappings in [TABLE 11-1](#).

TABLE 11-1 Example Set of Maps

Path Pattern	Servlet
/foo/bar/*	servlet1
/baz/*	servlet2
/catalog	servlet3
*.bop	servlet4

The behavior shown in [TABLE 11-2](#) would result.

TABLE 11-2 Incoming Paths Applied to Example Maps

Incoming Path	Servlet Handling Request
/foo/bar/index.html	servlet1
/foo/bar/index.bop	servlet1
/baz	servlet2
/baz/index.html	servlet2
/catalog	servlet3
/catalog/index.html	"default" servlet
/catalog/racecar.bop	servlet4
/index.bop	servlet4

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 need to communicate to deployers how the security is to be set up for the deployed application. This is accomplished declaratively by use of the deployment descriptors mechanism.

This chapter describes deployment representations for security requirements. The requirements for runtime representations are not specified, just as the requirements for web application directory layouts and deployment descriptors are not specified. It is recommended, however, that containers implement the elements set out here as part of their runtime representations.

This Java Card Platform version of the specification imposes that web resources to which a security constraint applies be only designated with URL patterns to which servlets have been mapped. Therefore, security constraints cannot directly apply to static content. Security constraints can only apply indirectly to static content through a servlet such as an explicitly declared “default” servlet that is used to serve static content in response to requests from web clients when no other servlet applies. This indirect mapping is assumed for any mention in this chapter of security constraints applying to static content.

12.1 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.
-

12.2 Declarative Security

Declarative security refers to the means of expressing an application's security structure, 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`.

12.3 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:

- `getRemoteUser`
- `isUserInRole`

The `getRemoteUser` method returns the user name the client used for authentication. The `isUserInRole` method determines if a remote user is in a specified security role. 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` and the `isUserInRole` method always returns `false`.

The `isUserInRole` method expects a `String` user role-name parameter. A `security-role-ref` element should be declared in the deployment descriptor with a `role-name` sub-element containing the role name to be passed to the method. A `security-role` element should contain a `role-link` sub-element whose value is the name of the security role that the user may be mapped into. The container uses the mapping of `security-role-ref` to `security-role` when determining the return value of the call.

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

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

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

If no `security-role-ref` element matching a `security-role` element has been declared, the container must default to checking the `role-name` element argument against the list of `security-role` elements for the web application. The `isUserInRole` method references the list to determine whether the caller is mapped to a security role. The developer must note that the use of this default mechanism may limit the flexibility in changing role names in the application without having to recompile the servlet making the call.

12.4 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 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.

Refer to the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for the actual declarative and programmatic security enforcement rules on the Java Card Platform.

12.5 Authentication

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

- HTTP Basic Authentication
- HTTP Digest Authentication
- Form-Based Authentication

The *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* introduces an additional authentication mechanism, which is required on the Java Card Platform. Refer to the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for details on this additional authentication mechanism specific to the Java Card Platform.

Caution – This Java Card Platform version of the specification requires that when an application is configured for container-managed HTTP Basic or Digest authentication, the web container filters out `Authorization` request headers so that they are not accessible to the application.

12.5.1 HTTP Basic Authentication

HTTP Basic Authentication, which is based on a user name 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 realm string of Basic Authentication does not have to reflect any particular security policy domain (confusingly also referred to as a realm). The web client obtains the user name 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.

12.5.2 HTTP Digest Authentication

Like HTTP Basic Authentication, HTTP Digest Authentication authenticates a user based on a user name and a password. However, with Digest Authentication the authentication is performed by transmitting the password in an encrypted form, which is much more secure than the simple base64 encoding used by Basic Authentication.

12.5.3 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 that 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 user name 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 triggering the authentication is stored by the container.
2. The user is asked to fill out the form, including the user name 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.
6. If authentication succeeds, the authenticated user’s principal is checked to see if it is in an authorized role for accessing the resource.
7. If the user is authorized, the client is redirected to the resource using the stored URL path.

The error page sent to a user that is not authenticated contains information about the failure.

Form-Based Authentication has the same lack of security as Basic Authentication since the user password is transmitted as plain text and the target server is not authenticated. Again 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.

12.5.3.1 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.

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 following example shows how the form should be coded into the HTML page:

```
<form method="POST" action="j_security_check">
  <input type="text" name="j_username">
  <input type="password" name="j_password">
</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 sign-on, the user would need to reauthenticate with any of the web applications hosted on the web container.

12.6 Server Tracking of Authentication Information

As the underlying security identities (such as users) 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.

12.7 Specifying Security Constraints

Security constraints are a declarative way of defining the protection of web content. A security constraint associates authorization and user data constraints, together or separately, with HTTP operations on web resources. A security constraint, which is represented by `security-constraint` in 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 (meaning 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` in deployment descriptor)

Caution – This Java Card Platform version of the specification imposes the following restriction on the use of `url-pattern` identifying constrained web resources: the `url-pattern` value of a `web-resource-collection` element must exactly correspond to the `url-pattern` value of one of the `servlet-mapping` elements defined for mapping request URL to servlets.

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. 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.

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.

12.7.1 Combining Constraints

Caution – This Java Card Platform version of the specification imposes the following restriction on security constraints: the same `url-pattern` and `http-method` value pair must not appear in multiple constraints. This restriction also applies to security constraints which do not have an `http-method` element as it stands for all the possible values of the `http-method` element. Therefore the web container on a Java Card Platform implementation must reject applications declaring in their deployment descriptors multiple security constraints with the same `url-pattern` and `http-method` value pair, as this is considered as a conflicting declaration.

12.7.2 Example of Applicable Constraints

[CODE EXAMPLE 12-1](#) illustrates the declaration of constraints and their translation into a table of applicable constraints. Suppose that a deployment descriptor contained the following security constraints.

CODE EXAMPLE 12-1 Example of Applicable Constraints

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>restricted methods</web-resource-name>
    <url-pattern>/acme/wholesale/*</url-pattern>
    <url-pattern>/acme/retail/*</url-pattern>
    <http-method>DELETE</http-method>
    <http-method>PUT</http-method>
  </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>-->
    <!--Uncommenting this PUT http-method element would result in
         the application being rejected as per the restrictions
         specific to the Java Card Platform. The previous security
         constraint has precluded the PUT http-method for access
         to that same url-pattern, this would therefore result in
         a conflict.-->
  </web-resource-collection>
  <auth-constraint>
    <role-name>SALESCLERK</role-name>
  </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>-->
    <!--Uncommenting this GET http-method element would result in
         the application being rejected as per the restrictions
         specific to the Java Card Platform. The previous security
         constraint has precluded the GET http-method for access
         to that same url-pattern, this would therefore result in
         a conflict.-->
    <http-method>POST</http-method>
```

CODE EXAMPLE 12-1 Example of Applicable Constraints *(Continued)*

```
</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 12-1](#).

TABLE 12-1 Security Constraint Table

url-pattern	http-method	permitted roles	supported connection types
/acme/wholesale/*	DELETE	Access precluded	Not constrained
/acme/wholesale/*	GET	SALESCLERK	Not constrained
/acme/wholesale/*	POST	CONTRACTOR	CONFIDENTIAL
/acme/wholesale/*	PUT	Access precluded	Not constrained
/acme/retail/*	DELETE	Access precluded	Not constrained
/acme/retail/*	GET	CONTRACTOR HOMEOWNER	Not constrained
/acme/retail/*	POST	CONTRACTOR HOMEOWNER	Not constrained
/acme/retail/*	PUT	Access precluded	Not constrained

12.7.3 Processing Requests

When a servlet container receives a request, it shall use the algorithm described in [Section 11.1, “Use of URL Paths” on page 11-1](#) 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:

- 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.¹
- 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. (This applies to HTTP Basic and Digest authentication schemes only, for other authentication schemes such a condition should trigger the authentication procedure specific to the scheme.) 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.

12.8 Default Policies

By default, authentication is not needed to access resources. Authentication is needed for requests for a web resource collection only when specified by the deployment descriptor.

1. 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).

12.9 Login and Logout

Being logged in to a web application corresponds precisely to there being a valid non-null value in `getRemoteUser` method, as described in [Section 12.3](#), “[Programmatic Security](#)” on page 12-2 and in the Javadoc tool files. A null value in that method indicates that a user is logged out.

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 for the Java Card Platform 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.

The deployment descriptor is defined in an XML schema document.

13.1 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 Life Cycle Listener classes
- Filter Definitions and Filter Mappings
- MIME Type Mappings
- Welcome File list
- Error Pages
- Locale and Encoding Mappings
- Security

13.2 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 white space, 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. Tools are required to validate deployment descriptor against the XML schema for structural correctness. The validation is recommended, but not required for the web containers.

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. Vendors are encouraged to supply this kind of validity checking in the form of a tool separate from the container.

- The subelements 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 `session-config`, `welcome-file-list`, `login-config`, and `locale-encoding-mapping-list` is set to “0 or more”. The containers or tools must inform the developer with a descriptive error message when the deployment descriptor contains more than one element of `session-config`, and `login-config`. The container must concatenate the items in `welcome-file-list` and `locale-encoding-mapping-list` when there are multiple occurrences.
- URI paths specified in the deployment descriptor are assumed to be in URL-decoded form. The containers or tools 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 white space in a URL.
- Containers or tools 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.

13.3 Deployment Descriptor

The XML Schema for the deployment descriptor of web applications intended to be deployed on the Java Card Platform is defined as a subset of the schema defined in the Servlet Specification, version 2.4.

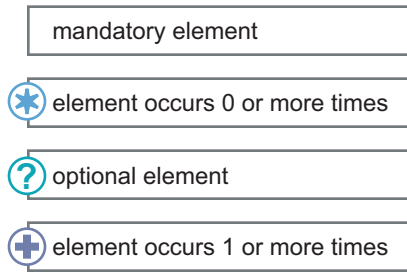
Deployment descriptors conforming to the XML Schema defined for the Java Card Platform are also conforming to the schema defined for the Servlet Specification, version 2.4.

Note – Deployment descriptors conforming to the XML Schema defined for the Servlet Specification, version 2.4, such as those generated by Servlet 2.4 compliant tools, may also be valid against the schema defined for the Java Card Platform version of this specification as long as they only use the supported subset.

13.4 Deployment Descriptor Element Structure

This section illustrates the elements in a deployment descriptor. All diagrams follow the convention displayed in [FIGURE 13-1](#). Attributes are not shown in the diagrams. See Deployment Descriptor Schema for the detailed information.

FIGURE 13-1 Conventions Used in Diagrams of Elements

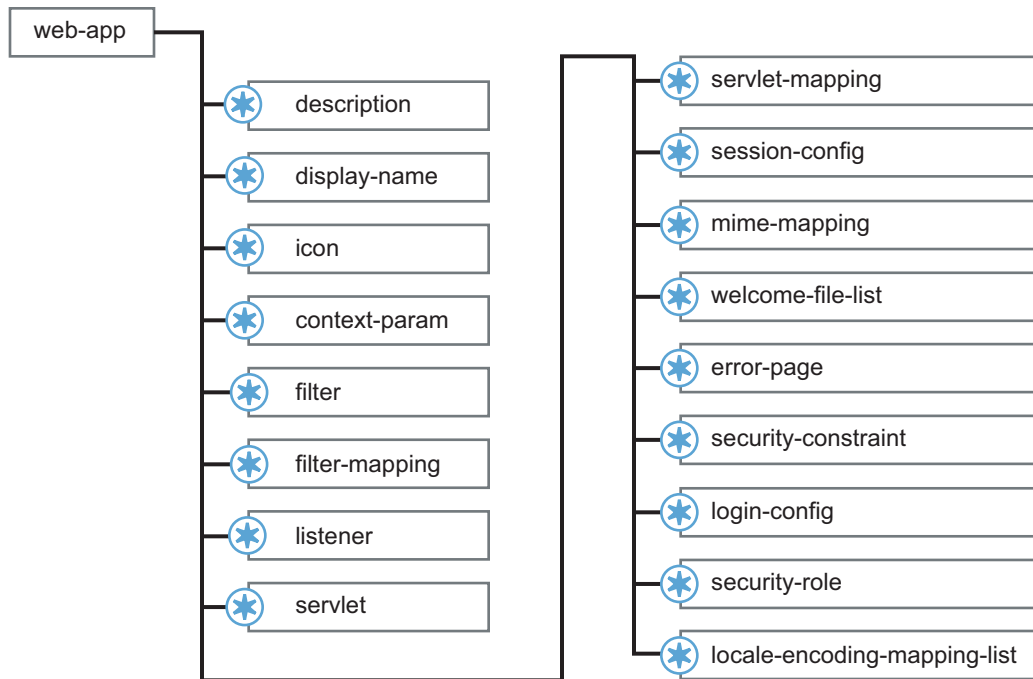


13.4.1 web-app Element

The `web-app` element is the root deployment descriptor for a web application. 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 13-2](#) shows the structure of the `web-app` element.

Caution – While the multiplicity of the elements `session-config`, `welcome-file-list`, `login-config`, and `locale-encoding-mapping-list` is set to “0 or more” in the XML Schema, they should actually occur only once at the most. See [Section 13.2, “Rules for Processing the Deployment Descriptor”](#) on [page 13-2](#) for more details.

FIGURE 13-2 web-app Element Structure



13.4.2 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").

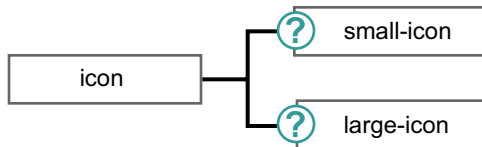
13.4.3 display-name Element

The display-name contains a short name that is intended to be displayed by tools. The display name need not be unique. This element occurs not only under the web-app element but also under other multiple elements. This element has an optional attribute `xml:lang` to specify the language.

13.4.4 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. This element occurs not only under the `web-app` element but also under other multiple elements. [FIGURE 13-3](#) shows structure of the `icon` element.

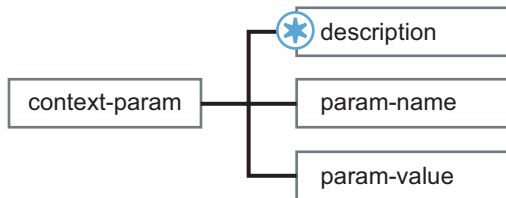
FIGURE 13-3 `icon` Element Structure



13.4.5 context-param Element

The `context-param` contains the declaration of a web application's servlet context initialization parameters.

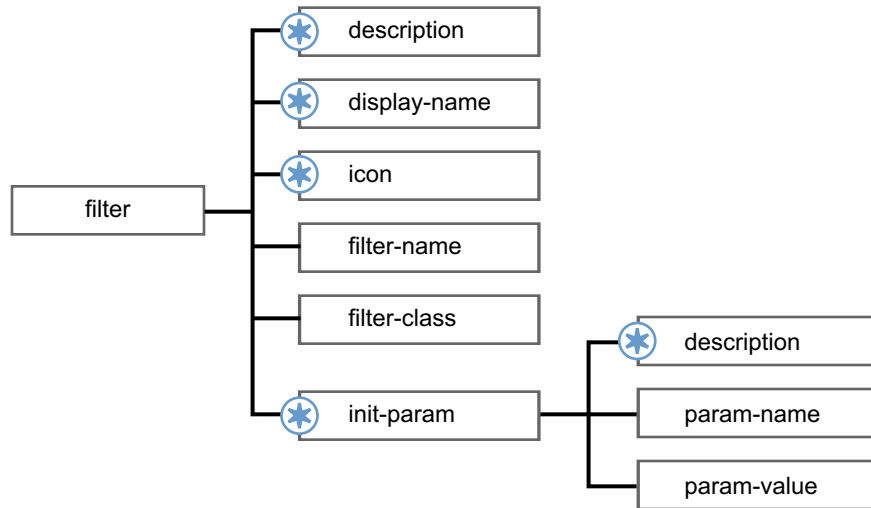
FIGURE 13-4 `context-param` Element Structure



13.4.6 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 the `filter-name` element must not be empty. The `filter-class` is the fully qualified class name of the filter. The `init-param` element contains name-value pair as an initialization parameter of this filter. [FIGURE 13-5](#) shows structure of the `filter` element.

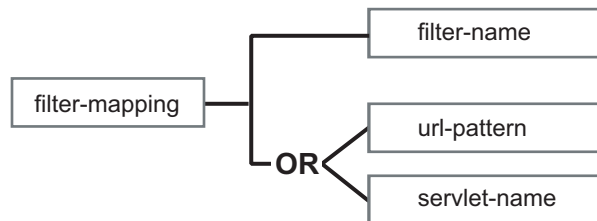
FIGURE 13-5 filter Element Structure



13.4.7 filter-mapping Element

The `filter-mapping` element 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 13-6](#) shows the structure of the `filter-mapping` element.

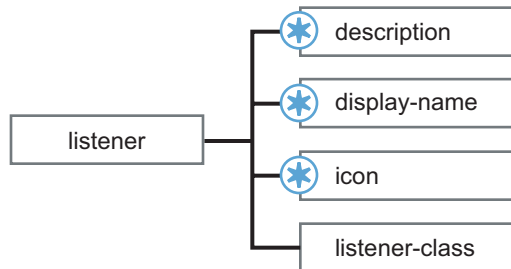
FIGURE 13-6 filter-mapping Element Structure



13.4.8 listener Element

The `listener` element 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 class name of the listener class. [FIGURE 13-7](#) shows the structure of the `listener` element.

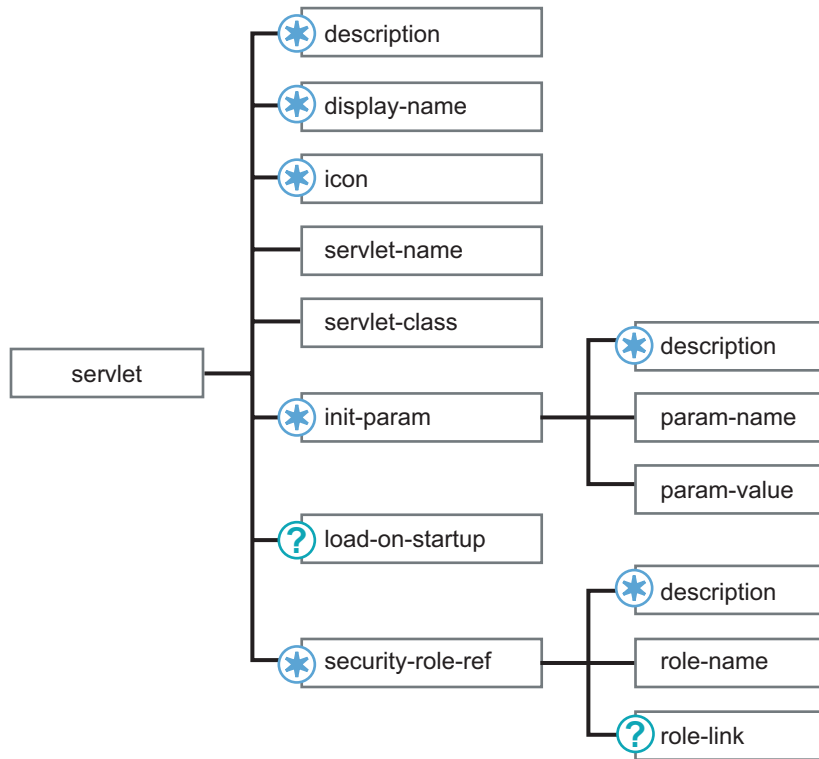
FIGURE 13-7 listener Element Structure



13.4.9 servlet Element

The `servlet` element is used to declare a servlet. It contains the declarative data of a servlet. 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 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. [FIGURE 13-8](#) shows the structure of the `servlet` element.

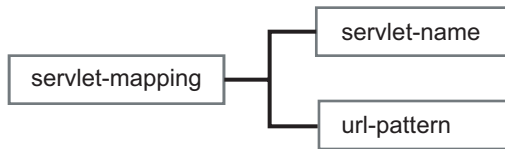
FIGURE 13-8 servlet Element Structure



13.4.10 servlet-mapping Element

The `servlet-mapping` element defines a mapping between a servlet and a URL pattern. [FIGURE 13-9](#) shows the structure of the `servlet-mapping` element.

FIGURE 13-9 servlet-mapping Element Structure



13.4.11 session-config Element

The `session-config` element 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 13-10](#) shows the structure of the `session-config` element.

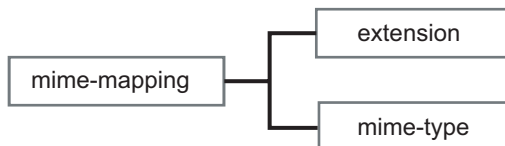
FIGURE 13-10 session-config Element Structure



13.4.12 mime-mapping Element

The `mime-mapping` element defines a mapping between an extension and a mime type. The `extension` element contains a string describing an extension, such as "txt". [FIGURE 13-11](#) shows the structure of the `mime-mapping` element.

FIGURE 13-11 mime-mapping Element Structure



13.4.13 welcome-file-list Element

The `welcome-file-list` element 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 13-12](#) shows the structure of the `welcome-file-list` element.

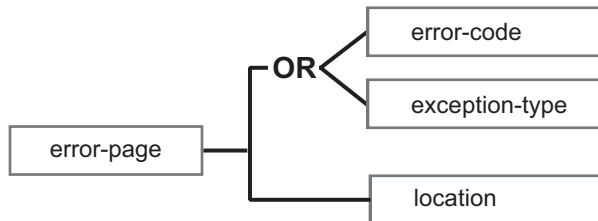
FIGURE 13-12 `welcome-file-list` Element Structure



13.4.14 error-page Element

The `error-page` element contains a mapping between an error code or an exception type to the path of a resource in the web application. The sub-element `exception-type` contains a fully qualified class name of a Java programming language 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 13-13](#) shows the structure of the `error-page` element.

FIGURE 13-13 `error-page` Element Structure

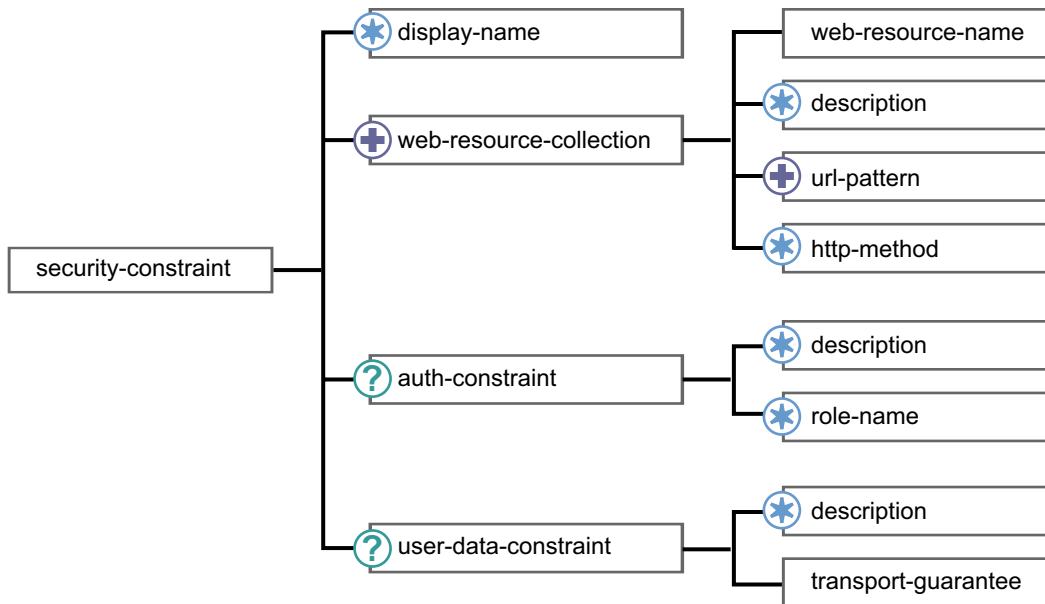


13.4.15 security-constraint Element

The `security-constraint` element 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 `"*"`, which 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 NONE, INTEGRAL or CONFIDENTIAL. [FIGURE 13-14](#) shows the structure of the security-constraint element.

FIGURE 13-14 security-constraint Element Structure



13.4.16 login-config Element

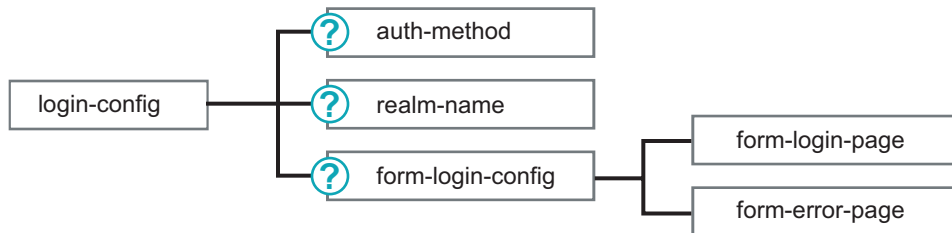
The login-config element is used to configure the authentication method that should be used, the realm name that should be used for this application, and the attributes that are needed by the form login mechanism. The sub-element auth-method configures the authentication mechanism for the web application. The element content must be either BASIC, DIGEST, FORM or a vendor-specific authentication scheme.

The *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* defines a Java Card Platform-specific extensible authentication scheme as a required extension to the default supported schemes listed above.

The `realm-name` indicates the realm name to use 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 (unless the scheme is the Java Card Platform-specific extensible authentication scheme, which is similar to the FORM-Based login and therefore reuses these elements).

FIGURE 13-15 shows the structure of the `login-config` element.

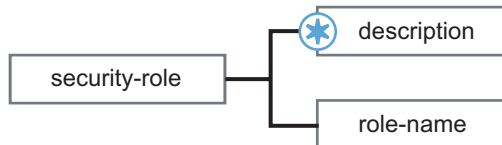
FIGURE 13-15 `login-config` Element Structure



13.4.17 security-role Element

The `security-role` element 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 13-16 shows the structure of the `security-role` element.

FIGURE 13-16 `security-role` Element Structure



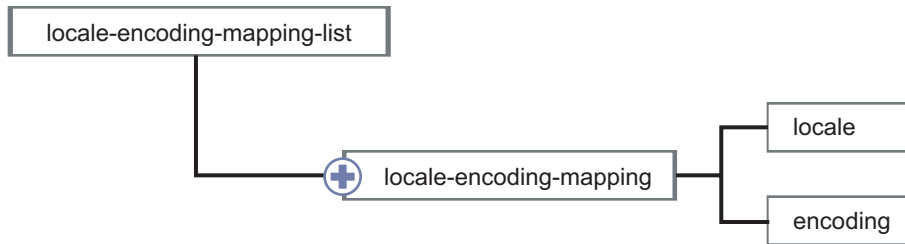
13.4.18 locale-encoding-mapping-list Element

The locale-encoding-mapping-list contains the mapping between the locale and the encoding, specified by the sub-element locale-encoding-mapping as shown in [CODE EXAMPLE 13-1](#) and [FIGURE 13-7](#).

CODE EXAMPLE 13-1 locale-encoding-mapping-list Element Code

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

FIGURE 13-17 locale-encoding-mapping-list Element Structure



13.5 Examples of Deployment Descriptor Usage

The following examples illustrate the usage of the definitions listed in the deployment descriptor schema.

13.5.1 A Basic Example

[CODE EXAMPLE 13-2](#) is a deployment descriptor conforming to the XML schema defined for this Java Card Platform version of the Java Servlet Specification. This deployment descriptor can be validated against the XML schema defined for this specification, as well as against the XML schema defined for Java Servlet Specification, version 2.4.

CODE EXAMPLE 13-2 Deployment Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<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/javacard/jcweb-app_3_0.xsd"
  version="2.4">
  <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.shtm</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>
```

CODE EXAMPLE 13-2 Deployment Descriptor (Continued)

```
</error-page>
</web-app>
```

The same example may be generated by a Java Servlet 2.4-compliant tool. This deployment descriptor can be validated against the XML Schema defined for Servlet Specification, version 2.4, as well as against the schema defined for Java Card Platform version of this specification.

```
<?xml version="1.0" encoding="UTF-8"?>
<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_4.xsd"
  version="2.4">
  ...
</web-app>
```

13.5.2 An Example of Security

See [CODE EXAMPLE 13-3](#)

CODE EXAMPLE 13-3 Security Related Deployment Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<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/javacard/jcweb-app_3_0.xsd"
  version="2.4">
  <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>
```


CODE EXAMPLE 13-3 Security Related Deployment Descriptor *(Continued)*

```
<role-name>manager</role-name>
</security-role>
<servlet-mapping>
  <servlet-name>catalog</servlet-name>
  <url-pattern>/catalog/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>catalog</servlet-name>
  <url-pattern>/salesinfo/*</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>
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>mgmt</realm-name>
</login-config>
</web-app>
```


Glossary

access control mechanism	a mechanism that permits or denies the access to a particular resource by a particular entity. An access control mechanism enforces a security policy.
active applet instance	an applet instance that is selected on at least one of the logical channels.
AID (application identifier)	<p>defined by ISO 7816, a string used to uniquely identify card applet applications and certain types of files in card file systems. An AID consists of two distinct pieces: a 5-byte RID (resource identifier) and a 0 to 11-byte PIX (proprietary identifier extension). The RID is a resource identifier assigned to companies by ISO. The PIX identifiers are assigned by companies.</p> <p>A unique AID is associated with each applet class in an applet application module. In addition, a unique AID is assigned to each applet instance during installation. This applet instance AID is used by an off-card client to select the applet instance for APDU communication sessions.</p> <p>Applet instance URIs are constructed from their applet instance AID using the "aid" registry-based namespace authority as follows:</p> <pre>//aid/<RID>/<PIX></pre> <p>where <RID> (resource identifier) and <PIX> (proprietary identifier extension) are components of the AID.</p>
APDU	an acronym for Application Protocol Data Unit as defined in ISO 7816-4.
APDU-based application environment	consists of all the functionalities and system services available to applet applications, such as the services provided by the applet container.
API	an acronym for Application Programming Interface. The API defines calling conventions by which an application program accesses the operating system and other services.

applet	within the context of this document, a Java Card applet, which is the basic component of applet-based applications and which runs in the APDU application environment.
applet application	an application that consists of one or more applets.
applet container	contains applet-based applications and manages their lifecycles through the applet framework API. Also provides the communication services over which APDU commands and responses are sent.
applet framework	an API that enables applet applications to be built.
applicable credential manager	the credential manager instance, application-assigned or card manager-assigned, that applies for a particular mode of communication. See credential manager .
applicable security requirements	the security requirements instance, application-assigned or card manager-assigned, that applies for a particular mode of communication. See security requirements .
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 application classes and resources, and other supporting libraries and files for the application. The output of the application assembler is an application archive.
application-defined event	an event that an application may define in its own namespace and may be the only one allowed to fire.
application-defined service	a service that an application may define in its own namespace and may be the only one allowed to register.
application descriptor	see descriptor .
application developer	The producer of an application. The output of an application developer is a set of application classes and resources, and supporting libraries and files for the application. The application developer is typically an application domain expert. The developer is required to be aware of the application environment and its consequences when programming, including concurrency considerations, and create the application accordingly.
application firewall	see firewall .
application framework class loader	a direct child of the extension library class loader, in charge of loading application framework libraries shared among a restricted set of application groups.
application group	a set of one or more applications executing in a common group context.

application-managed authentication	authentication that is programmatically triggered by an application's code based on some business logic.
application-managed connection endpoint	a client or server connection endpoint managed directly by an application.
application module class loader	a direct child in the class loader delegation hierarchy of either a group library class loader or of the classic library class loader, depending on the type of application model, in charge of loading the application module classes.
application protection domain	the set of permissions effectively granted to an application, that results from the combination of permissions granted by the platform security policy and the permissions granted by the card management security policy.
application security policy	a role-based security policy defined for a specific application and for which all the logical user and client security roles have been mapped to actual user identities and client application identities or characteristics on the platform to which the application is deployed.
application URI	a URI uniquely identifying an application instance on the platform.
atomic operation	an operation that either completes in its entirety or no part of the operation completes at all.
atomicity	state in which a particular operation is atomic. Atomicity of data updates guarantee that data are not corrupted in case of power loss or card removal.
authentication	the process of establishing or confirming an application or a user as authentic using some sort of credentials
authenticator	an authentication service that can be invoked both by applications for application-managed authentication and by the web container for container-managed authentication.
authorization	the process of allowing access to those resources by entities (applications or users) that have been granted authority to use them.
basic logical channel	logical channel 0, the only channel that is active at card reset in the APDU application environment. This channel is permanent and can never be closed.
bootstrap class loader	the root of the class loader delegation hierarchy in charge of loading the Java Card RE system classes.
bytecode	machine-independent code generated by the compiler and executed by the Java virtual machine.
canonicalization (URI)	the combined process of resolving a URI against a base URI, then normalizing it.

card holder	the primary user of a smart card.
card holder-facing client	a client that may directly and safely interact with the card holder. A card holder-facing client may typically be local, co-hosted on the card-hosting device, or in close proximity to the card.
card holder user	a user whose identity may be assumed by the card holder.
card manager	the on-card application to download and install applications and libraries. The card manager receives executable binary and metadata from the off-card installer, writes the binary into the smart card memory, links it with the other classes on the card, and creates and initializes any data structures used internally by the Java Card Runtime Environment.
card management facility	the Java Card platform layer responsible for securely adding and removing application code and instances onto the platform.
card management security policy	a permission-based security policy that is defined by a card management authority and that grants some permissions to an application or group of applications in accordance with the operational environment in which the application or group of applications is deployed.
card session	a card session begins when it is powered up or reset. The card is then able to exchange messages with external clients. The card session ends when the card loses power or is reset.
classic applet	applets with the same capabilities as those in previous versions of the Java Card platform and in the Classic Edition.
classic applet container mutex object	the object that is used by the Java Card RE to synchronize all concurrent accesses to a classic applet's code in order to guarantee its thread safety.
Classic Edition	one of the two editions in the Java Card Platform, Version 3. The Classic Edition is based on an evolution of the Java Card Platform, Version 2.2.2 and is backward compatible with it, targeting resource-constrained devices that solely support applet-based applications.
classic library	a Java programming language package that does not contain any non-abstract classes that extend the class <code>javacard.framework.Applet</code> . A classic applet application comprises a Java programming language package that contains one or more non-abstract classes that extend the <code>javacard.framework.Applet</code> class.
classic library class loader	a direct child of the shareable interface class loader in charge of loading classic library classes.
classic SIO proxy	see <i>classic SIO synchronization proxy</i> .

classic SIO synchronization proxy	an object that implements a shareable interface of a classic applet application and that synchronizes with all other concurrent accesses to the classic applet application before delegating to the actual SIO. An SIO synchronization proxy is returned to each client of the classic applet application that requests access to that shareable interface.
class loader	a Java Card RE component that defines and enforces a different class namespace for the classes it loads.
class loader delegation hierarchy	the hierarchy of class loaders that enforces code isolation among applications while allowing for sharing of system and library code.
client application	an on-card application that uses services provided by other applications (server applications).
client-role-based security	see <i>role-based security</i> .
Connected Edition	one of the two editions in the Java Card Platform, Version 3. The Connected Edition has a significantly enhanced runtime environment and a new virtual machine. It includes new network-oriented features, such as support for web applications, including the Java™ Servlet APIs, and also support for applets with extended and advanced capabilities. An application written for or an implementation of the Connected Edition may use features found in the Classic Edition.
connection endpoint (client, server)	see <i>application-managed connection endpoint</i> , <i>container-managed connection endpoint</i> .
container-managed authentication	authentication that is automatically triggered by the web container when a request to a protected resource is received, based on the declarative security configuration of the application.
container-managed connection endpoint	a server connection endpoint managed by a container, such as an HTTP or HTTPS server connection endpoint managed by the servlet container.
container-managed object	an object of which the lifecycle (creation, invocation, deletion...) is managed by a container. Examples are instances of <code>Applet</code> , <code>Servlet</code> and <code>Filter</code> .
context path	the path within the web server a servlet context is rooted at. The context path of a web application corresponds to its application URI.

context switch	a change from one currently active context to another. For example, a context switch is caused by an attempt to access an object that belongs to an application instance that resides in a different application group. The result of a context switch is a new currently active context.
converter	a piece of software that preprocesses all of the Java programming language class files of a classic applet application that make up a package, and converts the package into a standalone classic applet application module distribution format (CAP file). The Converter also produces an export file.
credential	material that can be used to ascertain the identity of a party (authenticate) in order to control access by that party to information or other resources and or to protect the integrity or confidentiality of information exchanges with that party. Examples of credentials are password, PIN or public-key certificates.
credential manager	an object that manages the key and trust material of an application when a secure communication is being established by either that application or by the web container on behalf of that web application.
currently active context	when an object instance method is invoked, an owning context of the object becomes the currently active context for that particular thread of execution.
currently active namespace	corresponds to the application owner identifier of the active context set upon entry into the group context for a particular thread of execution.
currently selected applet	the applet container keeps track of the currently selected Java Card applet. Upon receiving a SELECT FILE command with this applet's AID, the applet container makes this applet the currently selected applet. The applet container sends all APDU commands to the currently selected applet.
declarative security	a means of expressing an application's security structure, including roles, access control, and authentication requirements in a form external to the application, such as in the deployment descriptor of a web application.
default applet	an applet that is selected by default on a logical channel in the APDU application environment when it is opened. If an applet is designated the default applet on a particular logical channel in the APDU application environment on the Java Card platform, it becomes the active applet by default when that logical channel is opened using the basic channel.
default servlet	the application-defined servlet that is used to serve a request when no other servlet applies.
default default servlet	a servlet implementing the default container behavior that serves static resources of web applications. This servlet is used to serve a request to static content when no other servlet applies and no default servlet is defined by the application.

deployer	<p>The deployer takes one or more application archive files provided by an application developer and deploys the application into a card in a specific operational environment. The operational environment includes other installed applications and libraries, as well as standard bodies-defined frameworks. The deployer must resolve all the external dependencies declared by the developer.</p> <p>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 users that exist in the operational environment where the application is deployed.</p>
deployment unit	entity that can be distributed, deployed and installed on the Java Card platform.
deployment descriptor	see descriptor .
descriptor	a document that describes the configuration and deployment information of an application. A deployment descriptor conveys the elements and configuration information of an application between application developers, application assemblers, and deployers. A runtime descriptor describes the configuration and deployment information of an application that are specific to an operating environment to which the application is to be deployed.
distribution format	structure and encoding of a distribution or deployment unit intended for public distribution.
distribution unit	see deployment unit .
EEPROM	an acronym for Electrically Erasable, Programmable Read Only Memory.
entry point method	well-defined method of an object owned by an application (respectively the Java Card RE) that can be “legally” invoked by another application or the Java Card RE (respectively an application). SIO methods and other container-managed objects’ lifecycle methods are application entry point methods. Java Card RE entry point objects’ methods are Java Card RE entry point methods.
event	an object that encapsulates some occurring condition or situation. In the context of the event notification facility, an event is a shareable interface object that an application (event-producing application) uses to notify its clients (event-consuming applications) of an occurring condition.
event consuming application	an application that registers for notification of events fired by an event producing application.
event listener	an object that is registered to handle events when they occur. In the context of the event notification facility, an event listener is an object that a client application (event-consuming application) registers and uses to handle SIO-based events an application (event-producing application) produces.

event notification facility	a Java Card RE facility (or subsystem) that is used for event-driven inter-application communications.
event notification listener	see <i>event listener</i> .
event producing application	an application that fires events.
event registry	the core component of the event notification facility. The event registry is used for registering for notification of events and for notifying of events.
event URI	a URI that uniquely identifies an event produced by an event-producing application.
export file	a file produced by the Converter tool used during classic applet application development that represents the fields and methods of a package that can be imported by classes in other classic applet applications and classic libraries.
extended applet	an applet with extended and advanced capabilities (compared to a classic applet) such as the capabilities to manipulate <code>String</code> objects and open network connections.
extension library	library that extends the functionality of the platform.
extension library class loader	a direct child of the shareable interface class loader in the class loader delegation hierarchy in charge of loading extension libraries.
externally visible	<p>in the Java Card platform, any classes, interfaces, their constructors, methods, and fields of an application that can be accessed from another application according to the Java programming language semantics, as defined by the <i>Java Language Specification</i>, and code isolation restrictions (see the “Code Isolation” section in <i>Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition</i>).</p> <p>Externally visible items of a classic applet application are represented in an export file. For a classic library package, all classes, interfaces, their constructors, methods, and fields of an application that can be accessed from another application according to the Java programming language access control semantics, as defined by the <i>Java Language Specification</i> are listed in the export file.</p>
file permissions mode	an attribute of a file system object that indicates whether read or write operation on the object are permitted or denied.
filter	a web application component that is used to transform the content or header information of HTTP requests or responses.

finalization	<p>the process by which a Java virtual machine (VM) allows an unreferenced object instance to release non-memory resources (for example, close and open files) prior to reclaiming the object's memory. Finalization is only performed on an object when that object is ready to be garbage collected (meaning, there are no references to the object).</p> <p>Finalization is not supported by the Java Card virtual machine. The method <code>finalize()</code> is not called automatically by the Java Card virtual machine.</p>
firewall	the mechanism that prevents unauthorized accesses to objects in one application group context from another application group context.
flash memory	a type of persistent mutable memory. It is more efficient in space and power than EPROM. Flash memory can be read bit by bit but can be updated only as a block. Thus, flash memory is typically used for storing additional programs or large chunks of data that are updated as a whole.
garbage collection	the process by which dynamically allocated storage is automatically reclaimed during the execution of a program.
global array	an applet environment array objects accessible from any context.
global authentication	the scope of a user authentication that can be tracked globally (card-wide). Global authentication is restricted to card-holder-users. Authorization to access resources protected by a globally authenticated card-holder-user identity is granted to all users.
group context	protected object space associated with each application group and Java Card RE. All objects owned by an application belong to the context of the application group.
group-library class loader	a direct child of the extension library class loader in charge of loading the libraries private to an application group. Libraries, private to different application groups, are loaded by distinct group library class loaders, one per web or extended applet application group.
heap	<p>a common pool of free memory in volatile and persistent spaces usable by a program. A part of the computer's memory used for dynamic memory allocation, in which blocks of memory are used in an arbitrary order.</p> <p>The Java Card virtual machine's volatile heap is typically garbage collected on demand and on card tear.</p> <p>The Java Card virtual machine's persistent heap is typically garbage collected on a less frequent basis. Memory associated with objects allocated from the persistent heap are not necessarily reclaimed.</p>
instance variables	also known as non-static fields.

instantiation	in object-oriented programming, to produce a particular object from its class template. This involves allocation of a data structure with the types specified by the template, and initialization of instance variables with either default values or those provided by the class's constructor function.
instruction	a statement that indicates an operation for the computer to perform and any data to be used in performing the operation. An instruction can be in machine language or a programming language.
internally visible	items that are not externally visible to other applications on the card. See also <i>externally visible</i> .
inter-application communication facility	see <i>service facility</i> , <i>event notification facility</i> .
JAR file	an acronym for Java Archive file, which is a file format used for aggregating and compressing many files into one.
Java Card Platform Remote Method Invocation	a subset of the Java Platform Remote Method Invocation (RMI) system optionally supported by the APDU application environment. It provides a mechanism for a client application to invoke a method on a remote object of an applet application on the card.
Java Card Runtime Environment (Java Card RE)	consists of the Java Card virtual machine and the associated native methods.
Java Card Virtual Machine (Java Card VM)	a subset of the Java virtual machine, which is designed to be run on smart cards and other resource-constrained devices. The Java Card VM acts an engine that loads Java class files and executes them with a particular set of semantics.
Java Card RE context	the context of the Java Card RE has special system privileges so that it can perform operations that are denied to contexts of applications.

Java Card RE entry point object	<p>an object owned by the Java Card RE context that contains entry point methods. These methods can be invoked from any application group context and allows applications to request Java Card RE system services. A Java Card RE entry point object can be either temporary or permanent:</p> <p>temporary - references to temporary Java Card RE entry point objects cannot be stored in class variables, instance variables or array components. The Java Card RE detects and restricts attempts to store references to these objects as part of the firewall functionality to prevent unauthorized reuse. Examples of these objects are APDU objects and the APDU byte array.</p> <p>permanent - references to permanent Java Card RE entry point objects can be stored and freely reused. Examples of these objects are Java Card RE-owned AID instances.</p>
JDK™ software	an acronym for Java Development Kit. The JDK software is a Sun Microsystems, Inc. product that provides the environment required for software development in the Java programming language. The JDK software is available for a variety of operating systems, for example Sun Microsystems Solaris™ OS and Microsoft Windows.
local variable	a data item known within a block, but inaccessible to code outside the block. For example, any variable defined within a method is a local variable and cannot be used outside the method.
locally accessible web application	an application that may interact with the card holder.
logical channel	as seen at the card edge, works as a logical link to an applet application on the card. A logical channel establishes a communications session between a card applet and the terminal. Commands issued on a specific logical channel are forwarded to the active applet on that logical channel. For more information, see the <i>ISO/IEC 7816 Specification, Part 4</i> . (http://www.iso.org).
MAC	an acronym for Message Authentication Code. MAC is an encryption of data for security purposes.
mask production (masking)	refers to embedding the Java Card virtual machine, runtime environment, and applications in the read-only memory of a smart card during manufacture.
method	a procedure or routine associated with one or more classes in object-oriented languages.
mode (communication)	designates the type or protocol of communication (HTTPS, SSL/TLS, SIO...) and the mode of operation (client or server) that characterizes a communication endpoint.

module (application)	the logical unit of assembly of web or applet-based application. The components of a web application are assembled into a web application module. The components of an applet application are assembled into a applet application module.
multiselectable applets	implements the <code>javacard.framework.MultiSelectable</code> interface. Multiselectable applets can be selected on multiple logical channels in the APDU application environment at the same time. They can also accept other applets belonging to the same applet application being selected simultaneously.
multiselectd applet	an applet instance that is selected and, therefore, active on more than one logical channel in the APDU application environment simultaneously.
named permission	a permission that has a name but no actions list; the named permission is either granted or not. A named permission typically protects a function or functionality.
namespace	a set of names in which all names are unique.
native method	a method that is not implemented in the Java programming language, but in another language. The Card Manger does not load applications containing native methods.
nibble	four bits.
non-card holder-facing client	a client that does not directly interact with the card holder, but interacts with some other-users such as remote administrators. A non-card holder-facing client may typically be a remote system that may interact with the card through the network to which the card-hosting device itself is connected.
non-volatile memory	memory that is expected to retain its contents between card tear and power up events or across a reset event on the smart card device.
normalization (classic applet)	the process of transforming and repackaging a Java application packaged for the Java Card Platform, Version 2.2.2, for deployment on both the Java Card Platform, Version 3, Connected Edition and the Java Card Platform, Version 3, Classic Edition.
normalization (URI)	the process of removing unnecessary "." and ".." segments from the path component of a hierarchical URI.
Normalizer	a software tool that allows Java applications programmed for the Java Card Platform, Version 2.2.2, to be deployed on both the Java Card 3 Platform, Connected Edition and on the Java Card 3 Platform, Classic Edition. It also allows Java applications packaged for Version 2.2.2 to be transformed through the normalization process and then repackaged for deployment on both the Connected and Classic Editions.

object-oriented	a programming methodology based on the concept of an <i>object</i> , which is a data structure encapsulated with a set of routines, called <i>methods</i> , which operate on the data.
object owner	the applet instance context or web application context or the Java Card RE context which was the currently active context when the object was instantiated.
object	in object-oriented programming, unique instance of a data structure defined according to the template provided by its class. Each object has its own values for the variables belonging to its class and can respond to the messages (methods) defined by its class.
off-card client	see off-card client application .
off-card client application	an application that is not resident on the card, but runs at the request of a user's actions.
off-card installer	the off-card application that transmits the application and library executables to the card manager application running on the card.
off-card proxy generator	a program or tool used to generate classic SIO synchronization proxies prior to packaging and deploying a classic applet application.
on-card client	see client application .
origin logical channel	the logical channel in the APDU application environment on which an APDU command is issued.
"other user"	a user other than a card holder user, such as a remote card administrator.
owning context	the application or Java Card RE context in which an object is instantiated or created.
owner context	see owning context .
package	a namespace within the Java programming language that can have classes and interfaces.
permission	an object that represents access to specific protected resources, such as security-sensitive system resources, or application resources, such as services provided by applications. Permissions are instances of subclasses of the <code>Permission</code> class. A permission has a name and may have an actions list.
permission actions list	an attribute of a permission used to designate those actions for which the resources designated by the target name are protected.
permission-based security	measures defined by a permission-based security policy that restrict access to protected system and library resources.

permission-based security policy	a security policy that maps some of the characteristics of an application requesting access to a protected resource to a set of permissions granted to the application.
permission name	an attribute of a permission used to designate a protected function or resource, or a set thereof.
permission target name	the name attribute of a permission object (permission name) that designates the resource or set of resources that are protected with that permission.
permission type	a type defined by a permission class.
persistent object	persistent objects and their values persist from one card session to the next, indefinitely. Persistent object values are typically updated atomically using transactions. The term persistent does not mean there is an object-oriented database on the card or that objects are serialized and deserialized, just that the objects are not lost when the card loses power.
PIX	see AID (application identifier) .
platform event	a well-defined event fired by the platform. Examples are clock resynchronization events.
platform protection domain	a set of permissions granted to an application or group of applications by the platform security policy. A platform protection domain is defined by two sets of permissions: a set of included permissions that are granted and a set of excluded permissions that are denied and can never be granted.
platform security policy	the permission-based security policy that maps application models to sets of permissions granted to applications implementing these application models. For each of the application models, the platform security policy guarantees the consistency and integrity of the applications implementing the application model.
principal	an entity that can be authenticated by an authentication protocol. A principal is identified by a <i>principal name</i> and authenticated by using <i>authentication data</i> . The content and format of the principal name and the authentication data depend on the authentication protocol.
programmatic security	a means for a security aware application to express the security model of the application when declarative security alone is not sufficient.
protected content	see protected resource .
protected resource	an application or system resource that is protected by an access control mechanism.

protection domain	a set of permissions granted to an application or group of applications.
RAM (random access memory)	temporary working space for storing and modifying data. RAM is non-persistent memory; that is, the information content is not preserved when power is removed from the memory cell. RAM can be accessed an unlimited number of times and none of the restrictions of EEPROM apply.
reachability disrupting object	a special object that prevents the promotion of a volatile object to become a persistent object. If a volatile object is referenced by a persistent object, which is not a reachability disrupting object, or by a root of persistence, the volatile object is automatically promoted and becomes a persistent object. An example of reachability disrupting object is a <code>TransientReference</code> object.
reference implementation	a fully functional and compatible implementation of a given technology. It enables developers to build prototypes of applications based on the technology.
remote interface	<p>an interface of an applet application, which extends, directly or indirectly, the interface <code>java.rmi.Remote</code>.</p> <p>Each method declaration in the remote interface or its super-interfaces includes the exception <code>java.rmi.RemoteException</code> (or one of its superclasses) in its throws clause.</p> <p>In a remote method declaration, if a remote object is declared as a return type, it is declared as the remote interface, not the implementation class of that interface.</p> <p>In addition, Java Card RMI imposes additional constraints on the definition of remote methods of an applet application. See <i>Runtime Environment Specification, Java Card Platform, v3.0.1, Classic Edition</i>.</p>
remote methods	the methods of a remote interface of an applet application.
remote object	an object of an applet application whose remote methods can be invoked remotely from the off-card client. A remote object is described by one or more remote interfaces of an applet application.
remote user	an user whose identity may be assumed by a remote entity, such as a remote card administrator.
remotely accessible web application	an application that is not expected to interact with the card holder but with other-users, potentially remote.
resolution (URI)	the process of resolving one URI against another, base URI. The resulting URI is constructed from components of both URIs in the manner specified by RFC 3986, taking components from the base URI for those not specified in the original.

resource URI	a URI that uniquely identifies a resource on the platform. Examples are service URI, event URI, application URI and file URI.
RFU	acronym for Reserved for Future Use.
RID	see <i>AID (application identifier)</i> .
RMI	an acronym for Remote Method Invocation. RMI is a mechanism for invoking instance methods on objects located on remote virtual machines (meaning, a virtual machine other than that of the invoker).
role (development)	the actions and responsibilities taken by various parties during the development, deployment, and running of an 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.
role-based security	measures defined by a role-based security policy that restrict access by clients or by users to protected application resources.
role-based security policy	a security policy that maps some of the characteristics of an application requesting access to protected resources, such as its identity and the identity of the user on behalf of whom the access is requested to roles permitted to access the protected resources.
ROM (read-only memory)	memory used for storing the fixed program of the card. A smart card's ROM contains operating system routines as well as permanent data and user applications. No power is needed to hold data in this kind of memory. ROM cannot be written to after the card is manufactured. Writing a binary image to the ROM is called masking and occurs during the chip manufacturing process.
root URI	a URI that identifies the root of an application's namespace for a particular scheme. Examples are an application's service root URI, an application's event root URI.
runtime descriptor	see <i>descriptor</i> .
runtime environment	see <i>Java Card Runtime Environment (Java Card RE)</i> .
secure port redirector	a web application container that redirects HTTP requests for protected content sent over unsecure connections to the secure port over which that content can be served. Protected content must be served only over a secure port.
security constraint	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.

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 <i>realm</i> .
security policy	designates the protected resources that can be accessed by individual applications or groups of applications. These protected resources may be security-sensitive system resources or application resources such as services provided by other applications.
security requirements	the required security characteristics for a particular secure communication being established by either an application or by the web container on behalf of a web application.
server application	an on-card application that provides a service to its clients.
service	a shareable interface object that a server application uses to provide a set of well-defined functionalities to its clients.
service facility	a Java Card RE facility (or subsystem) that is used for inter-application communications.
service factory	an object that the Java Card RE invokes to create a service - on behalf of the server application that registered that service - for a client application that looked up the service.
service registry	the core component of the service facility. The service facility is used for registering and looking up services.
service URI	a URI that uniquely identifies a service provided by a server application.
servlet	a web application component, managed by a container, that generates dynamic web content and that runs in the web application environment.
servlet container	see web application container .
servlet context	a container-managed object that defines a servlet's view of the web application within which the servlet is running. A servlet context is rooted at a known path within a web server: a context path.
servlet definition	a unique name associated with a fully qualified class name of a class implementing the <code>servlet</code> interface. A set of initialization parameters can be associated with a servlet definition. See <i>Java Servlet Specification, Connected Edition</i> .
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. See <i>Java Servlet Specification, Connected Edition</i> .

session-scoped authentication	the scope of a user authentication that is tracked on a per-session basis. This prevents a user authenticated in a conversational session under one identity to gain unauthorized access to protected resources authorized to another, simultaneously authenticated, identity.
shareable interface	an interface that defines a set of shared methods. These interface methods can be invoked from an application in one group context when the object implementing them is owned by an application in another group context.
shareable interface class loader	the direct child of the bootstrap class loader in the class loader delegation hierarchy in charge of loading publicly exposed shareable interfaces.
shareable interface object (SIO)	an object that implements the shareable interface.
shareable interface object-based service	see <i>service</i> .
smart card	a card that stores and processes information through the electronic circuits embedded in silicon in the substrate of its body. Unlike magnetic stripe cards, smart cards carry both processing power and information. They do not require access to remote databases at the time of a transaction.
SPI	an acronym for Service Provider Interface or sometimes for System Programming Interface. The SPI defines calling conventions by which a platform implementer may implement system services.
standard event	a standard event with a well-defined semantic that an application may fire. Examples are standard application lifecycle events such as application creation and deletion events, and standard resource lifecycle events such as resource creation and deletion events.
standard service	a standard service with a well-defined interface that an application may provide and register. Examples are authenticators - authentication services.
restartable task	an object implementing the <code>Runnable</code> interface that has been registered for recurrent execution over card sessions. A task executes in its own thread.
restartable task registry	a Java Card RE facility that is used for registering tasks for recurrent execution over card sessions.
terminal	is typically a computer in its own right with an interface which connects with a smart card to exchange and process data.
thread	the basic unit of program execution. A process can have several threads running concurrently each performing a different job, such as waiting for events or performing a time consuming job that the program doesn't need to complete before going on. When a thread has finished its job, it is suspended or destroyed.

thread's active context	when an object instance method is invoked, the owning context of the object becomes the currently active context for that particular thread of execution. Synonymous with <i>currently active context</i> .
transaction	an atomic operation in which the developer defines the extent of the operation by indicating in the program code the beginning and end of the transaction.
transaction facility	a Java Card RE facility that enables an application to complete a single logical operation on application data atomically, consistently and durably within a transaction.
transient object	the state of transient objects do not persist from one card session to the next, and are reset to a default state at specified intervals. Updates to the values of transient objects are not atomic and are not affected by transactions.
transferable classes	<p>classes whose instances can have their ownership transferred to a context different from their currently owning context. Transferable classes are of two types:</p> <p>Implicitly transferable classes - Classes whose instances are not bound to any context (group contexts or Java Card RE context) and can, therefore, be passed and shared between contexts without any firewall restrictions. Examples are Boolean and literal String objects.</p> <p>Explicitly transferable classes - Classes whose instances must have their ownership explicitly transferred to another application's group context in order to be accessible to that other application. Examples are arrays and newly created String objects.</p>
transfer of ownership	a Java Card RE facility that allows for an application to transfer the ownership of objects it owns to an other application. Only instances of transferable classes can have their ownership transferred.
trusted client	an on-card or off-card application client that an on-card application trusts on the basis of credentials presented by the client.
trusted client credentials	credentials that an on-card application uses to ascertain the identity of clients it trusts.
uniform resource identifier (URI)	a compact string of characters used to identify or name an abstract or physical resource. A URI can be further classified as a uniform resource locator (URL), a uniform resource name (URN), or both. See RFC 3986 for more information.
uniform resource locator (URL)	a compact string representation used to locate resources available via network protocols or other protocols. Once the resource represented by a URL has been accessed, various operations may be performed on that resource. See RFC 1738 for more information. A URL is a type of uniform resource identifier (URI).

user role-based security	see <i>role-based security</i> .
verification	a process performed on an application or library executable that ensures that the binary representation of the application or library is structurally correct.
volatile memory	memory that is not expected to retain its contents between card tear and power up events or across a reset event on the smart card device.
volatile object	an object that is ideally suited to be stored in volatile memory. This type of object is intended for a short-lived object or an object which requires frequent updates. A volatile object is garbage collected on card tear (or reset).
web application	<p>a collection of servlets, HTML documents, and other web resources that might include image files, compressed archives, and other data. A web application is packaged into a web application archive.</p> <p>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. See <i>Java Servlet Specification, Connected Edition</i>.</p>
web application archive	<p>the physical representation of a web application module. A single file that contains all of the components of a web application. This archive file is created by using standard JAR file tools, which allow any or all of the web components to be signed.</p> <p>A web application archive file is identified by the <code>.war</code> extension and is often referred to as a WAR file. A new extension is used instead of <code>.jar</code> because that extension is reserved for files which contain a set of class files and that can be placed in the classpath. As the contents of a web application archive are not suitable for such use, a new extension was required. See <i>Java Servlet Specification, Connected Edition</i>.</p>
web application container	contains and manages web applications and their components (for example, servlets) through their lifecycle. Also provides the network services over which HTTP requests and responses are sent and manages security of web applications.
web application environment	in addition to the Java Card RE, consists of all the functionalities and system services available to web applications, such as the services provided by the web application container.
web client	an off-card entity that requests services from an on-card web application. A typical example is a web browser.

XML schema description of a type of XML document as a set of rules to which an XML document must conform in order to be considered valid according to that schema.

Index

A

Accept-Language header, 4-6
access control mechanism, Glossary-1
active applet instance, Glossary-1
addDateHeader method, 5-3
addHeader method, 5-2
addIntHeader method, 5-3
AID (application identifier), Glossary-1
APDU, Glossary-1
APDU-based application environment, Glossary-1
API, Glossary-1
applet, Glossary-2
applet application, Glossary-2
applet container, Glossary-2
applet framework, Glossary-2
applicable credential manager, Glossary-2
applicable security requirements, Glossary-2
application assembler, Glossary-2
application descriptor, Glossary-2
application developer, Glossary-2
application event listeners, 10-1
application firewall, Glossary-2
application framework class loader, Glossary-2
application group, Glossary-2
application module class loader, Glossary-3
application protection domain, Glossary-3
application security policy, Glossary-3
application URI, Glossary-3
application-defined event, Glossary-2
application-defined service, Glossary-2

application-managed authentication, Glossary-3
atomic operation, Glossary-3
atomicity, Glossary-3
attributes naming convention, 4-3
authentication, Glossary-3
 default policy, 12-11
 form-based, 12-5
 HTTP Basic, 12-4
 HTTP Digest, 12-5
 mechanisms of, 12-4
 tracking information, 12-6
authenticator, Glossary-3
authorization, Glossary-3
authorization constraint, 12-7
authorization role names, 12-8

B

basic logical channel, Glossary-3
binding notifications for sessions, 7-3
bootstrap class loader, Glossary-3
buffering output, 5-1
bytecode, Glossary-3

C

canonicalization (URI), Glossary-3
card holder, Glossary-4
card holder user, Glossary-4
card holder-facing client, Glossary-4
card management facility, Glossary-4
card management security policy, Glossary-4
card manager, Glossary-4

- card session, Glossary-4
- character encoding, 4-7
 - overriding, 4-7
 - setting default for requests, 4-7
 - setting default for responses, 5-4
 - setting for requests, 4-7
 - setting for responses, 5-4
- class loader, Glossary-5
- class loader delegation hierarchy, Glossary-5
- classes
 - GenericServlet, 1-8, 2-1
 - HttpServlet, 1-8, 2-1
 - not serializable, 1-8
- classic applet, Glossary-4
- classic applet container mutex object, Glossary-4
- Classic Edition, Glossary-4
- classic library, Glossary-4
- classic library class loader, Glossary-4
- classic SIO proxy, Glossary-4
- client requests, handling, 2-1, 2-4
- client-role-based security, Glossary-5
- clients, redirecting to different URLs, 5-3
- concurrent processing with multiple threads, 2-4
- concurrent requests, sending, 2-4
- conditional GET operations, 2-2
- Connected Edition, Glossary-5
- connection endpoint, Glossary-5
- connection endpoint (client, server), Glossary-5
- container-managed authentication, Glossary-5
- container-managed object, Glossary-5
- Content-Type header, 5-3
- context attributes
 - getAttribute, 3-2
 - getAttributeNames, 3-2
 - javax.cardx.security.request.X509Certificate, 1-4
 - removeAttribute, 3-2
 - setAttribute, 3-2
- context path, Glossary-5
- context path, request path element, 4-4
- context paths
 - conflicts, 9-2
 - examples of context setup, 4-4
- context switch, Glossary-6
- converter, Glossary-6

- cookies, 4-5
 - attributes of, 4-6
 - HTTP, 7-1
 - JSESSIONID, 7-1
 - session tracking, 7-1
 - unaccepted by clients, 7-2
- credential, Glossary-6
- credential manager, Glossary-6
- currently active context, Glossary-6
- currently active namespace, Glossary-6
- currently selected applet, Glossary-6

D

- declarative security, 12-2, Glossary-6
- default applet, Glossary-6
- default default servlet, Glossary-6
- default servlet, Glossary-6
- deployer, Glossary-7
- deployment descriptor, Glossary-7
- deployment descriptor schema
 - elements removed
 - CLIENT-CER value of auth-method, 1-6
 - dispatcher, 1-6
 - distributable, 1-5
 - run-as, 1-5
- deployment descriptors
 - configuration and deployment information, 9-4
 - declaring listener classes, 10-3
 - defining error page descriptions, 9-5
 - element structure, 13-3
 - elements of, 13-1
 - context-param, 13-6
 - description, 13-5
 - display-name, 13-5
 - error-page, 13-11
 - filter, 13-6
 - filter-mapping, 13-7
 - icon element, 13-6
 - listener, 13-8
 - locale-encoding-mapping-list, 13-13
 - login-config, 13-12
 - mime-mapping, 13-10
 - security-constraint, 13-11
 - security-role, 13-13
 - servlet, 13-8
 - servlet-mapping, 13-9
 - session-config, 13-10

- web-app, 13-4
- welcome-file-list, 13-10
- elements removed
 - JNDI objects lookup, 1-5
 - JSP configuration, 1-5
 - message destination, specifying, 1-5
 - web service reference, 1-5
- examples of
 - basic elements, 13-14
 - listener element, 10-4
 - security elements, 13-16
- mapping syntax, 11-2
- processing rules, 13-2
- schema version conformation, 13-4
- support requirements, 13-1
- welcome files, 9-7
- XML schema validity, 13-3
- deployment hierarchies, 9-2
- deprecated classes
 - HttpUtils, 1-11
- deprecated constructors
 - UnavailableException, 1-12
- deprecated interfaces
 - HttpSessionContext, 1-11
 - SingleThreadModel, 1-11
- deprecated methods
 - encodeRedirectUrl, 1-13
 - encodeUrl, 1-13
 - getRealPath, 1-12
 - getServlet, 1-11, 1-12
 - getServletNames, 1-11
 - getServlets, 1-11
 - getValue, 1-14
 - getValueNames, 1-14
 - getSessionContext, 1-13
 - isRequestedSessionIdFromUrl, 1-13
 - log, 1-12
 - putValue, 1-14
 - removeValue, 1-14
 - setStatus, 1-13
- destroy method, 2-4, 2-6
- distribution format, Glossary-7
- distribution unit, Glossary-7
- doDelete method, 2-2
- doFilter method, 4-7, 6-3
 - implementing, 6-3
- doGet method, 2-2, 2-4

- doHead method, 2-2
- doOptions method, 1-8, 2-2
- doPost method, 2-1, 2-4
- doPut method, 2-2
- doTrace method, 1-8, 2-2

E

- EEPROM, Glossary-7
- entry point method, Glossary-7
- Enumeration of String objects, 4-3
- error page mechanism, 9-6
- event consuming application, Glossary-7
- event notification listener, Glossary-8
- event producing application, Glossary-8
- event registry, Glossary-8
- event types, list of, 10-2
- event URI, Glossary-8
- export file, Glossary-8
- extended applet, Glossary-8
- extension library, Glossary-8

F

- file system path equivalents, obtaining, 4-5
- filter chains, caching, 6-6
- filter declarations, examples of, 6-5
- filter mapping
 - URL pattern use restrictions, 6-6
- FilterChain objects, 6-3
- FilterConfig objects, 6-5
- filtering, 6-2
 - main concepts of, 6-2
- filtering components
 - auditing, 6-2
 - authentication, 6-2
 - caching, 6-2
 - data compression, 6-2
 - encryption, 6-2
 - image conversion, 6-2
 - logging, 6-2
 - MIME-type chain filters, 6-2
 - resource access event triggers, 6-2
 - tokenizing, 6-2
 - XML content transformation, 6-2
- filtering content, 6-1
- filtering functionality, types of, 6-2

filters

- applying, 6-7
- associating with servlets, 6-5
- configuring, 6-2, 6-5
- creating a chain of, 6-6
- declaring, 6-5
- definition of, 6-1
- initialization parameters, 6-4
- initializing parameters for, 6-5
- lifecycle of, 6-3
- mapping to servlets, 6-2
- mapping to URL patterns, 6-2
- specifying class of, 6-5
- specifying names, 6-5
- types of exceptions thrown, 9-6

flushBuffer method, 5-1, 5-2, 8-3

Form-Based authentication, 12-5

form-based logins, 12-6

forward method, 8-4

forward request parameters, list of, 8-4

G

GenericServlet class, 2-1

getAttribute context attribute, 3-2

getAttribute request attributes, 4-2

getAttributeNames context attribute, 3-2

getAttributeNames request attributes, 4-2

getBufferSize method, 5-1

getCharacterEncoding method, 4-7

getContextPath method, 4-4

getCookies method, 4-5

getDateHeader method, 4-3

getHeader method, 4-3

getHeaderNames method, 4-3

getHeaders method, 4-3

getInitParameter initialization parameter, 3-2

getInitParameter method, 6-4

getInitParameterNames initialization
parameter, 3-2

getInitParameterNames method, 6-5

getIntHeader method, 4-3

getLastAccessedTime method, 7-4

getLastModified method, 2-2

getLocale method, 4-6

getLocales method, 4-6

getMaxInactiveInterval method, 7-4

getNamedDispatcher method, 8-1

getParameter method, 4-1

getParameterNames method, 4-1

getParameterValues method, 4-1

getPathInfo method, 4-2, 4-4

getRemoteUser method, 12-2

getRequestDispatcher method, 8-1

getRequestURI method, 4-2

getResourceAsStream method, 1-7, 3-2, 9-2, 9-3

getServletPath method, 4-4

getWriter method, 5-4

H

header methods, 4-3

HTTP Basic authentication, 12-4

HTTP cookies, 7-1

HTTP Digest authentication, 12-5

HTTP protocol parameters, 4-1

HTTP request objects, 2-4

HTTP requests

- default encoding, 4-7, 5-4

- headers of, 4-3

HTTP response headers, 5-2

HTTP session objects

- scoping of, 7-3

- tracking login states, 12-12

HttpServlet class, 2-1

HttpServletRequest interface, 4-3

- removed methods of, 1-13

HttpServletRequest.getPathTranslated
method, 4-5

HttpServletRequestInterface

- removed methods of, 1-10

HttpServletRequestWrapper class

- removed methods of, 1-10, 1-13

HttpServletRequestWrapper class, removed
methods of, 1-13

HttpSession interface, removed methods, 1-13

HttpSessionBindingListener interface, 7-3

I

implicit mappings, 11-2

include method, 8-3

include request parameters, list of, 8-3

init method, 2-3, 2-4

initialization parameters

function of, 3-2

getInitParameter, 3-2

getInitParameterNames, 3-2

interfaces

HttpServletRequest, 4-3

HttpSessionActivationListener, 1-9

HttpSessionBindingListener, 7-3

javax.servlet.Filter, 6-2

javax.servlet.Servlet, 8-2

removed, 1-9

RequestDispatcher, 8-1

Servlet, 2-1

ServletConfig, 2-3

ServletContext, 2-3, 3-1

ServletRequest, 4-1

invalidate method, 10-6

isCommitted method, 5-1, 5-2

isSecure method, 4-6

isUserInRole method, 12-2

J

Java Servlet Specification

application support by servlet containers, 1-4

compatibility issues, 1-3

distributed containers unsupported, 1-9

session migration unsupported, 1-9

unsupported features, list of, 1-3

javax.servlet.Filter interface, 6-2

javax.servlet.http.HttpSessionAttribute
Listener interface, 10-2

javax.servlet.http.HttpSessionBindingL
istener interface, 10-2

javax.servlet.http.HttpSessionListener
interface, 10-2

javax.servlet.Servlet interface, 8-2

javax.servlet.ServletContextAttributeL
istener interface, 10-2

javax.servlet.ServletContextListener
interface, 10-2

javax.servlet.ServletRequestAttributeL
istener, 10-2

javax.servlet.ServletRequestAttributeL
istener interface, 10-2

javax.servlet.ServletRequestListener
interface, 10-2

L

languages, specifying to web server, 4-6

listener classes, 10-3

deployment declarations, 10-3

instantiation of, 10-5

registering, 10-4

listener exceptions, 10-5

listener interfaces, list of, 10-2

locales, specifying, 4-6

logical hosts, supporting multiple, 3-3

login states, tracking, 12-12

logout states, tracking, 12-12

M

mapping examples, 11-3

mapping syntax, 11-2

mapping techniques, 11-1

methods

addDateHeader, 5-3

addHeader, 5-2

addIntHeader, 5-3

destroy, 2-4, 2-6

doDelete, 2-2

doFilter, 4-7, 6-3

doGet, 2-2, 2-4

doHead, 2-2

doOptions, 2-2

doPost, 2-1, 2-4

doPut, 2-2

doTrace, 2-2

flushBuffer, 5-1, 5-2, 8-3

forward, 8-4

getBufferSize, 5-1

getCharacterEncoding, 4-7

getContextPath, 4-4

getCookies, 4-5

getDateHeader, 4-3

getHeader, 4-3

getHeaderNames, 4-3

getHeaders, 4-3

getInitParameter, 6-4

getInitParameterNames, 6-5

getIntHeader, 4-3

getLastAccessedTime, 7-4

- getLastModified, 2-2
- getLocale, 4-6
- getLocales, 4-6
- getMaxInactiveInterval, 7-4
- getNamedDispatcher, 8-1
- getParameter, 4-1
- getParameterNames, 4-1
- getParameterValues, 4-1
- getPathInfo, 4-2, 4-4
- getRemoteUser, 12-2
- getRequestDispatcher, 8-1
- getRequestURI, 4-2
- getResourceAsStream, 1-7, 3-2, 9-2, 9-3
- getServletPath, 4-4
- getWriter, 5-4
- HttpServletRequest.getPathTranslated, 4-5
- include, 8-3
- init, 2-3, 2-4
- invalidate, 10-6
- isCommitted, 5-1, 5-2
- isSecure, 4-6
- isUserInRole, 12-2
- reset, 5-1, 5-2
- resetBuffer, 5-1, 5-2
- sendError, 5-3, 9-6
- sendRedirect, 5-3
- service, 2-1, 2-4
- ServletContext.getRealPath, 4-5
- ServletResponse.setLocale, 5-4
- setBufferSize, 5-1
- setCharacterEncoding, 4-7, 5-4
- setContentType, 5-4
- setDateHeader, 5-2
- setHeader, 5-2
- setIntHeader, 5-2
- setLocale, 5-4
- setMaxInactiveInterval, 7-4
- valueBound, 7-3
- valueUnbound, 7-3
- multiple threads
 - handling concurrent requests, 2-4
 - synchronizing session access, 7-5

N

- Normalizer, Glossary-12

P

- path info, request path element, 4-4
- path mapping procedure, 11-1
- path translation methods, 4-5
- post form data, conditions for adding, 4-2
- programmatic security, 12-2
- protected content, Glossary-14

Q

- query strings
 - adding to dispatcher paths, 8-2
 - aggregating parameters, 8-4

R

- realm, Glossary-17
- removeAttribute context attribute, 3-2
- removed attributes, 1-14
- removed classes
 - Cookie, 1-9
- removed methods
 - getParameterMap, 1-9
 - getResource, 1-9
 - getResourcePaths, 1-9
 - getUserPrincipal, 1-10
 - print(double d), 1-10
 - print(float f), 1-10
 - println(double d), 1-10
 - println(float f), 1-10
- request attributes, 4-2
 - error handling, 9-5
 - getAttribute, 4-2
 - getAttributeNames, 4-2
 - javax.servlet.error.exception, 9-5
 - javax.servlet.error.request_uri, 9-5
 - javax.servlet.error.servlet_name, 9-5
 - javax.servlet.error.status_code, 9-5
 - javax.servlet.forward.context_path, 8-4
 - javax.servlet.forward.path_info, 8-4
 - javax.servlet.forward.query_string, 8-4
 - javax.servlet.forward.request_uri, 8-4
 - javax.servlet.forward.servlet_path, 8-4
 - javax.servlet.include.context_path, 8-3

- `javax.servlet.include.path_info`, 8-3
 - `javax.servlet.include.query_string`, 8-3
 - `javax.servlet.include.request_uri`, 8-3
 - `javax.servlet.include.servlet_path`, 8-3
 - reserved names, 4-3
 - `setAttribute`, 4-2
- request dispatchers, 8-1
 - error handling, 8-5
 - servlets, using, 8-2
- request handling errors, 2-5
 - permanent, 2-5
 - temporary, 2-5
- request handling methods, 2-1
 - Http-specific, 2-1
- request objects, 2-4, 4-1
 - accessing in multiple threads, 2-5
 - adding query strings to paths, 8-2
 - obtaining, 8-1
 - recycling, 4-7
 - validity of, 4-7
 - wrapping, 6-4, 8-2
- request parameters
 - conditions for adding post form data, 4-2
 - multiple values, 4-1
 - storing, 4-1
- request paths
 - context path, 4-4
 - elements of, 4-3
 - examples of element behaviors, 4-5
 - path info, 4-4
 - servlet path, 4-4
- `RequestDispatcher` interface, 8-1
- `RequestDispatcher` object, 8-1
- requests
 - dispatching, 8-1
 - forwarding, 8-4
 - including, 8-3
- `requestURI` composition, 4-4
- `reset` method, 5-1, 5-2
- `resetBuffer` method, 5-1, 5-2
- response objects, 5-1
 - accessing in multiple threads, 2-5
 - closing, 5-5
 - closure indicators, 5-5

- committing responses, 5-2
 - recycling, 5-5
 - validity of, 5-5
 - wrapping, 6-4, 8-2
 - writing information to, 8-3
- role names, 12-8

S

- Secure Sockets Layer (SSL) sessions, 7-2
- security constraints, 12-7
 - combined constraints, example of, 12-9
- security requirements, characteristics of, 12-1
- security roles
 - definition of, 12-3
 - mapping example, 12-3
- `sendError` method, 5-3, 9-6
- `sendredirect` method, 5-3
- `service` method
 - handling concurrent requests, 2-1
 - multithreading, 2-4
 - synchronizing, 2-5
- servlet
 - definition, 1-1
- servlet containers
 - application support, 1-4
 - buffering
 - output, 5-1
 - unsupported, 5-2
 - creating a chain of filters, 6-6
 - default locale, use of, 5-4
 - defining timeout periods, 7-4
 - definition, 1-2
 - deployment hierarchies, 9-2
 - function of, 1-1
 - HTTP support, 1-2
 - initialization errors, 2-4
 - initializing servlets, 2-3
 - instantiating filters, 6-3
 - instantiating servlets, 2-3
 - loading servlets, 2-3
 - processing request URIs, 6-6
 - processing requests, rules for, 12-11
 - providing instances of a servlet, 2-2
 - publishing implementation information, 5-3
 - request handling errors, 2-5
 - security restrictions, 1-2
 - sending concurrent requests, 2-4

- SSL certificates, exposing, 4-6
- SSL sessions, 7-2
- terminating servlets, 2-6
- tracking authentication information, 12-6
- servlet contexts
 - multiple hosts, 3-3
 - resource access, 3-2
- servlet declarations, 2-2
- servlet event listeners, 10-1
- `Servlet` interface, 2-1
 - function of, 2-1
- servlet path, request path element, 4-4
- `ServletConfig` interface, 2-3
- `ServletContext` interface, 2-3
 - default web applications, 3-1
 - deprecated methods of, 1-11
 - function of, 3-1
 - path, 3-1
 - removed methods of, 1-9
- `ServletContext.getRealPath` method, 4-5
- `ServletOutputStream` class, removed methods of, 1-10
- `ServletRequest` interface, 4-1
 - removed methods of, 1-9, 1-12
 - request attributes, 4-2
- `ServletRequestWrapper` class, removed methods of, 1-9, 1-12
- `ServletResponse.setLocale` method, 5-4
- servlets
 - associating filters with, 6-5
 - binding object attributes, 7-3
 - containers, 1-1
 - creating instances of, 2-2
 - declaring, 13-8
 - ending service, 2-6
 - engines, 1-1
 - failed initialization, 2-4
 - forwarding requests, 8-4
 - HTTP request headers, 4-3
 - initializing, 2-3
 - invocation example, 1-2
 - invoking, 1-2
 - lifecycle of, 2-3
 - loading, 13-8
 - mapping client requests, 11-1
 - preferred buffer size, setting, 5-1
 - request handling errors, 2-5
 - request parameters, 4-1
 - request paths, 4-3
 - setting character encoding, 5-4
 - setting locales, 5-4
 - specifying load order, 13-8
 - thread lifecycle, 2-5
 - types of exceptions thrown, 9-6
 - using request dispatchers, 8-2
 - web clients, interacting with, 1-1
- session IDs, encoding, 7-2
- session parameters
 - defining, 13-10
- session tracking, 7-1
- sessions
 - access by multiple threads, 7-5
 - binding notifications, 7-3
 - creating, 7-2
 - default timeout periods, 7-4
 - defining default timeout periods, 13-10
 - determining access times, 7-4
 - `jsessionid` parameter, 7-2
 - new status, 7-2
 - scoping, 7-3
 - session IDs, 7-2
 - status, conditions of, 7-2
 - terminating, 7-4
 - tracking, 10-6
 - URL rewriting, 7-2
- `setAttribute` context attribute, 3-2
- `setAttribute` request attributes, 4-2
- `setBufferSize` method, 5-1
- `setCharacterEncoding` method, 4-7, 5-4
- `setContentType` method, 5-4
- `setDateHeader` method, 5-2
- `setHeader` method, 5-2
- `setIntHeader` method, 5-2
- `setLocale` method, 5-4
- `setMaxInactiveInterval` method, 7-4
- single signon, 12-6
- SSL attributes, 4-6
 - `javax.security.cert.X509Certificate`, 1-4
- static content
 - documents, direct access to, 3-2
- static initialization methods, triggering, 2-4
- `synchronized` keyword, 2-4

synchronizing the `service` method, 2-5

T

thread's active context, Glossary-19

timeout periods

 changing, 7-4

 setting, 7-4

U

`UnavailableException` class, removed
 constructors and methods, 1-12

URL patterns restriction, 12-7

URL patterns, usage restrictions, 1-7, 11-1

URL rewriting, 7-2

user data constraint, 12-7

V

valid partial requests, 9-7

`valueBound` method, 7-3

`valueUnbound` method, 7-3

virtual hosting, 3-3

W

web application directory

 unsupported JAR files subdirectory, 1-7

web application path, 9-1

web applications

 alternate deployment mechanism, 1-7

 archive files, 9-3

 configuring filters, 6-5

 conflicting context paths, 9-2

 defining session parameters, 13-10

 definition of, 9-1

 deployment descriptors, 9-4

 deployment steps, 9-8

 elements of, 9-1

 error handling, 9-5

 list of sample files, 9-3

 listener classes, 10-3

 META-INF directory, 9-3

 structured directories, 9-2

 tracking sessions, 10-6

 URL paths, 11-1

 Web ARchive (WAR) files, 9-3

 WEB-INF directory, 9-2

Web ARchive (WAR) files, 9-3

 dependencies, 9-4

web containers

 caching filter chains, 6-6

 deployment descriptors support, 13-1

 exposing SSL attributes, 4-6

 HTTP session support, 7-2

 processing rules for deployment containers, 13-2

 registering listener classes, 10-4

 single signon support, 12-6

 URL rewriting mechanism support, 7-2

web resource collection, 12-7

WEB-INF directory

 contents of, 9-3

 description of, 9-2

 exposure of contents, 9-2

welcome files, 9-7

 defining a default file, 13-10

X

X-Powered-By HTTP header, 5-3

 examples of, 5-3

