Java™ Servlet Specification

Version 2.3

Please send technical comments to: servletapi-feedback@eng.sun.com
Please send business comments to: danny.coward@sun.com

Final Release 8/13/01
Danny Coward (danny.coward@sun.com)

Java(TM) Servlet API Specification ("Specification")
Version: 2.3
Status: Final Release
Release: September 17, 2001

Copyright 2001 Sun Microsystems, Inc.
901 San Antonio Road, Palo Alto, California 94303, U.S.A.
All rights reserved.

NOTICE
The Specification is protected by copyright and the information described therein may be protected by one or more U.S. patents, foreign patents, or pending applications. Except as provided under the following license, no part of the Specification may be reproduced in any form by any means without the prior written authorization of Sun Microsystems, Inc. ("Sun") and its licensors, if any. Any use of the Specification and the information described therein will be governed by the terms and conditions of this license and the Export Control Guidelines as set forth in the Terms of Use on Sun's website. By viewing, downloading or otherwise copying the Specification, you agree that you have read, understood, and will comply with all of the terms and conditions set forth herein.

Sun hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (wit out the right to sublicense), under Sun's intellectual property rights that are essential to practice the Specification, to internally practice the Specification for the purpose of designing and developing your Java applets and applications intended to run on the Java platform or creating a clean room implementation of the Specification that: (i) includes a complete implementation of the current version of the Specification, without subsetting or supersetting; (ii) implements all of the interfaces and functionality of the Specification without subsetting or supersetting; (iii) includes a complete implementation of any optional components (as defined by the Specification) which you choose to implement, without subsetting or supersetting; (iv) implements all of the interfaces and functionality of such optional components, without subsetting or supersetting; (v) does not add any additional packages, classes or interfaces to the "java.*" or "javax.*" packages or subpackages or other packages defined by the Specification; (vi) satisfies all testing requirements available from Sun relating to the most recently published version of the Specification six (6) months prior to any release of the clean room implementation or upgrade thereto; (vii) does not derive from any Sun source code or binary code materials; and (viii) does not include any Sun source code or binary code materials without an appropriate and separate license from Sun. The Specification contains the proprietary information of Sun and may only be used in accordance with the license terms set forth herein. This license will terminate immediately without notice from Sun if you fail to comply with any provision of this license. Upon termination or expiration of this license, you must cease use of or destroy the Specification.

TRADEMARKS
No right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun's licensors is granted hereunder. Sun, Sun Microsystems, the Sun logo, Java, and the Java Coffee Cup logo, are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

DISCLAIMER OF WARRANTIES
THE SPECIFICATION IS PROVIDED "AS IS". SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED,

INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION,
IF ANY. SUN MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

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

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

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

REPORT
You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your use of the Specification ("Feedback"). To the extent that you provide Sun with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to subl cense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

(LFI#95810/Form ID#011801)

Contents

Preface

This document is the Java™ Servlet Specification, v2.3. The standard for the Java servlet API is described here.

SRV.P.1 Additional Sources

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

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

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

SRV.P.2 Who Should Read This Specification

The intended audience for this specification includes the following groups:

- Web server and application server vendors that want to provide servlet engines that conform to this standard.
- Authoring tool developers that want to support web applications that conform to this specification
- Experienced servlet authors who want to understand the underlying mechanisms of servlet technology.

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

SRV.P.3 API Reference

Chapter 14, "API Details" includes the full specifications of classes, interfaces, and method signatures, and their accompanying javadoc^{TM}, that define the servlet API.

SRV.P.4 Other Java™ Platform Specifications

The following Java API specifications are referenced throughout this specification:

- Java 2 Platform, Enterprise Edition, v1.3 (J2EE^{TM})
- JavaServer Pages™, v1.1 (JSP^{TM})
- Java Naming and Directory Interface^{TM} (JNDI)

These specifications may be found at the Java 2 Platform,Enterprise Edition website: `http://java.sun.com/j2ee/`.

SRV.P.5 Other Important References

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

- RFC 1630 Uniform Resource Identifiers (URI)
- RFC 1738 Uniform Resource Locators (URL)
- RFC 2396 Uniform Resource Identifiers (URI): Generic Syntax
- RFC 1808 Relative Uniform Resource Locators
- RFC 1945 Hypertext Transfer Protocol (HTTP/1.0)
- RFC 2045 MIME Part One: Format of Internet Message Bodies
- RFC 2046 MIME Part Two: Media Types
- RFC 2047 MIME Part Three: Message Header Extensions for non-ASCII text
- RFC 2048 MIME Part Four: Registration Procedures
- RFC 2049 MIME Part Five: Conformance Criteria and Examples
- RFC 2109 HTTP State Management Mechanism
- RFC 2145 Use and Interpretation of HTTP Version Numbers
- RFC 2324 Hypertext Coffee Pot Control Protocol (HTCPCP/1.0)[1]
- RFC 2616 Hypertext Transfer Protocol (HTTP/1.1)
- RFC 2617 HTTP Authentication: Basic and Digest Authentication

Online versions of these RFCs are at `http://www.rfc-editor.org/`.

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 websites:

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

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

SRV.P.6 Providing Feedback

We welcome any and all feedback about this specification. Please e-mail your comments to `servletapi-feedback@eng.sun.com`.

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

SRV.P.7 Acknowledgements

The formulation of this public draft is the result of the teamwork of the JSR053 expert group.

CHAPTER SRV.1 Overview

SRV.1.1 What is a Servlet?

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

SRV.1.2 What is a Servlet Container?

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

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

All servlet containers must support HTTP as a protocol for requests and responses, but additional request/response based protocols such as HTTPS (HTTP over SSL) may be supported. The minimum required version of the HTTP specification that a container must implement is HTTP/1.0. It is strongly suggested that containers implement the HTTP/1.1 specification as well.

A Servlet Container may place security restrictions on the environment in which a servlet executes. In a Java™ 2 Platform, Standard Edition 1.2 (J2SE™) or

Java^{TM} 2 Platform, Enterprise Edition 1.3 (J2EE) environment, these restrictions should be placed using the permission architecture defined by the Java 2 platform. For example, high-end application servers may limit the creation of a `Thread` object, to insure that other components of the container are not negatively impacted.

J2SE 1.2 is the minimum version of the underlying Java platform with which servlet containers must be built.

SRV.1.3 An Example

The following is a typical sequence of events:

1. A client (e.g., a web browser) accesses a web server and makes an HTTP request.
2. The request is received by the web server and handed off to the servlet container. The servlet container can be running in the same process as the host web server, in a different process on the same host, or on a different host from the web server for which it processes requests.
3. The servlet container determines which servlet to invoke based on the configuration of its servlets, and calls it with objects representing the request and response.
4. The servlet uses the request object to find out who the remote user is, what HTTP POST parameters may have been sent as part of this request, and other relevant data. The servlet performs whatever logic it was programmed with, and generates data to send back to the client. It sends this data back to the client via the response object.
5. Once the servlet has finished processing the request, the servlet container ensures that the response is properly flushed, and returns control back to the host web server.

SRV.1.4 Comparing Servlets with Other Technologies

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

Servlets have the following advantages over other server extension mechanisms:

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

SRV.1.5 Relationship to Java 2, Platform Enterprise Edition

The Servlet API v2.3 is a required API of the Java™ 2 Platform, Enterprise Edition, v1.3[1]. Servlet containers and servlets deployed into them must meet additional requirements, described in the J2EE specification, for executing in a J2EE environment.

1. Please see the Java™ 2 Platform, Enterprise Edition specification available at `http://java.sun.com/j2ee/`

CHAPTER SRV.2 The Servlet Interface

The `Servlet` interface is the central abstraction of the 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 servlet API that implement the `Servlet` interface are `GenericServlet` and `HttpServlet`. For most purposes, developers will extend `HttpServlet` to implement their servlets.

SRV.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 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.

SRV.2.1.1 HTTP Specific Request Handling Methods

The `HttpServlet` abstract subclass adds additional methods beyond the basic `Servlet` interface which 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

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

SRV.2.1.2 Additional Methods

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

In containers that support only HTTP/1.0, only the `doGet`, `doHead` and `doPost` methods are supported, as HTTP/1.0 does not define the `PUT`, `DELETE`, `OPTIONS`, and `TRACE` methods.

SRV.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.

SRV.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 SRV.13, "Deployment Descriptor", controls how the servlet container provides instances of the servlet.

For a servlet not hosted in a distributed environment (the default), the servlet container must use only one instance per servlet declaration. However, for a servlet implementing the `SingleThreadModel` interface, the servlet container may instantiate multiple instances to handle a heavy request load and serialize requests to a particular instance.

In the case where a servlet was deployed as part of an application marked in the deployment descriptor as distributable, a container may have only one instance per servlet declaration per virtual machine (VM). However, if the servlet in a distributable application implements the `SingleThreadModel` interface, the container may instantiate multiple instances of that servlet in each VM of the container.

SRV.2.2.1 Note About The Single Thread Model

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

SRV.2.3 Servlet Life Cycle

A servlet is managed through a well defined life cycle that defines how it is loaded, instantiated and initialized, handles requests from clients, and how it 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.

SRV.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 normal Java class loading facilities. The loading may be from a local file system, a remote file system, or other network services.

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

SRV.2.3.2 Initialization

After the servlet object is instantiated, the container must initialize the servlet before it can handle requests from clients. Initialization is provided so that a servlet can read persistent configuration data, initialize costly resources (such as JDBC™ API based connections), and perform other one-time activities. The container initializes the servlet instance by calling the `init` method of the `Servlet` interface with a unique (per servlet declaration) object implementing the `ServletConfig` interface. This configuration object allows the servlet to access name-value initialization parameters from the web application's configuration information. The configuration object also gives the servlet access to an object (implementing the `ServletContext` interface) that describes the servlet's runtime environment. See Chapter SRV.3, "Servlet Context" for more information about the `ServletContext` interface.

SRV.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.

SRV.2.3.2.2 Tool Considerations

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

SRV.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 respones 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.

SRV.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 developer of the servlet must make adequate provisions for concurrent processing with multiple threads in the `service` method.

An alternative for the developer is to implement the `SingleThreadModel` interface which requires the container to guarantee that there is only one request thread at a time in the `service` method. A servlet container may satisfy this requirement by serializing requests on a servlet, or by maintaining a pool of servlet instances. If the servlet is part of a web application that has been marked as distributable, the container may maintain a pool of servlet instances in each VM that the application is distributed across.

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

SRV.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.

If temporary unavailability is indicated by the `UnavailableException`, then 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 `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.

SRV.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 must not be given to objects executing in other threads as the resulting behavior may be nondeterministic.

SRV.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 itself is being shut down.

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

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

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

CHAPTER SRV.3

Servlet Context

SRV.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://www.mycorp.com/catalog`. All requests that begin with the `/catalog` request path, known as the *context path*, are routed to the web application associated with the `ServletContext`.

SRV.3.2 Scope of a ServletContext Interface

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

Servlets in a container that were not deployed as part of a web application are implicitly part of a "default" web application and have a default `ServletContext`. In a distributed container, the default `ServletContext` is non-distributable and must only exist in one VM.

SRV.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. Typical examples are a webmaster's e-mail address, or the name of a system that holds critical data.

SRV.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 `ServletContext` interface allow access to this functionality:

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

SRV.3.4.1 Context Attributes in a Distributed Container

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

SRV.3.5 Resources

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

- `getResource`
- `getResourceAsStream`

The `getResource` and `getResourceAsStream` methods take a `String` with a leading “/” as argument which 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, on a remote server, or at some other location.

These methods are not used to obtain dynamic content. For example, in a container supporting the JavaServer Pages^{TM} specification[1], a method call of the form `getResource("/index.jsp")` would return the JSP source code and not the processed output. See Chapter SRV.8, “Dispatching Requests” for more information about accessing dynamic content.

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

SRV.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 can not be shared across virtual hosts.

SRV.3.7 Reloading Considerations

Although a Container Provider implementation of a class reloading scheme for ease of development is not required, any such implementation must ensure that all servlets, and classes that they may use[2], are loaded in the scope of a single class loader. This requirement is needed to guarantee that the application will behave as

[1] The JavaServer Pages^{TM} specification can be found at `http://java.sun.com/products/jsp`

expected by the Developer. As a development aid, the full semantics of notification to session binding listeners should be be supported by containers for use in the monitoring of session termination upon class reloading.

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

SRV.3.7.1 Temporary Working Directories

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

The requirement recognizes a common convenience provided in many servlet engine implementations. The container is not required to maintain the contents of the temporary directory when the servlet container restarts, but is required to ensure that the contents of the temporary directory of one servlet context is not visible to the servlet contexts of other web applications running on the servlet container.

[2] An exception is system classes that the servlet may use in a different class loader.

CHAPTER SRV.4

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.

SRV.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 a `HttpServletRequest object`, and conditions set out below 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`.

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.

SRV.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:

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

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

SRV.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 "`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 into the attribute set be named in accordance with the reverse domain name convention suggested by the Java Programming Language Specification[1] for package naming.

SRV.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, e.g. `Cache-Control` headers, in an HTTP request. If there are multiple headers with the same name, the `getHeader` method returns the first head 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.

SRV.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:

1. The Java Programming Language Specification is available at `http://java.sun.com/docs/books/jls`

- **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 namespace, this path will be an empty string. Otherwise, if the context is not rooted at the root of the server's namespace, 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 the empty string.
- **PathInfo:** The part of the request path that is not part of the Context Path or the Servlet Path. It is either null if there is no extra path, or is a string with a leading '/'.

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

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

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

```
requestURI = contextPath + servletPath + pathInfo
```

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

Table 1: Example Context Set Up

`Context Path`	`/catalog`

`Servlet Mapping`	`Pattern: /lawn/*`
	`Servlet: LawnServlet`

`Servlet Mapping`	`Pattern: /garden/*`
	`Servlet: GardenServlet`

`Servlet Mapping`	`Pattern: *.jsp`
	`Servlet: JSPServlet`

The following behavior is observed:

Table 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.jsp`	`ContextPath: /catalog`
	`ServletPath: /help/feedback.jsp`
	`PathInfo: null`

SRV.4.5 Path Translation Methods

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

- `ServletContext.getRealPath`
- `HttpServletRequest.getPathTranslated`

The `getRealPath` method takes a `String` argument and returns a `String` representation of a file on the local file system to which a path corresponds. The `getPathTranslated` method computes the real path of the `pathInfo` of the request.

In situations where the servlet container cannot determine a valid file path for these methods, such as when the web application is executed from an archive, on a remote file system not accessible locally, or in a database, these methods must return null.

SRV.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.

SRV.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 following attributes to the servlet programmer:

Table 3: Protocol Attributes

Attribute	Attribute Name	Java Type
cipher suite	`javax.servlet.request.cipher_suite`	`String`
bit size of the algorithm	`javax.servlet.request.key_size`	`Integer`

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

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

SRV.4.8 Internationalization

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

- `getLocale`
- `getLocales`

The `getLocale` method will return the preferred locale that the client will accept content in. See section 14.4 of RFC 2616 (HTTP/1.1) for more information about how the `Accept-Language` header must 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.

SRV.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.

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

SRV.4.10 Lifetime of the Request Object

Each request object is valid only within the scpoe 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 be aware that maintaining references to request objects outside the scope described above may lead to non-deterministic behavior.

Final Version

CHAPTER SRV.5 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.

SRV.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, this method must throw an `IllegalStateException`.

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

The `reset` method clears data in the buffer when the response is not committed. Headers 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.

SRV.5.2 Headers

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

- `setHeader`
- `addHeader`

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

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

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

- `setIntHeader`
- `setDateHeader`
- `addIntHeader`
- `addDateHeader`

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

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

SRV.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 (i.e. the response is not committed), the data in the response buffer must be cleared and replaced with the data set by these methods. If the response is committed, these methods must throw an `IllegalStateException`.

SRV.5.4 Internationalization

A servlet will set the language attributes of a response with the `setLocale` method of the `ServletResponse` interface when the client has requested a document in a particular language, or has set a language preference. This method must correctly set the `Content-Language` header (along with other mechanisms described in the HTTP/1.1 specification), to accurately communicate the `Locale` to the client.

For maximum benefit, the `setLocale` method should be called by the Developer before the `getWriter` method of the `ServletResponse` interface is called. This ensures that the returned `PrintWriter` is configured appropriately for the target `Locale`.

Note that a call to the `setContentType` method with a `charset` component for a particular content type, will override the value set via a prior call to `setLocale`.

The default encoding of a response is “ISO-8859-1” if none has been specified by the servlet programmer.

SRV.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.

SRV.5.6 Lifetime of the Response Object

Each response object is valid only within the scpoe 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 be aware that maintaining references to response objects outside the scope described above may lead to non-deterministic behavior.

CHAPTER SRV.6

Filtering

Filters are a new feature of the Java servlet API in version 2.3. Filters allow on the fly transformations of payload and header information in both the request into a resource and the response from a resource.

This chapter describes the new 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 in the API definitions chapters of this document. The configuration syntax for filters is given by the document type definition (DTD) in Chapter SRV.13. The reader should use these sources as references when reading this chapter.

SRV.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 contents are referred to as web resources.

Among the types of functionality available to the filter author are

- The accessing of a resource before a request to it is invoked.
- The processing of the request for a resource before it is invoked.
- The modification of request headers and data by wrapping the request in customized versions of the request object.

- The modification of response headers and response data by providing customized versions of the response object.
- The interception of an invocation of a resource after its call.
- Actions on a servlet, on groups of servlets or static content by zero, one or more filters in a specifiable order.

SRV.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

SRV.6.2 Main Concepts

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

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

SRV.6.2.1 Filter Lifecycle

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

Only one instance per `filter` declaration in the deployment descriptor is instantiated per Java virtual machine of the container. The container provides the filter `config` as declared in the filter's deployment descriptor, the reference to the `ServletContext` for the web application, and the set of initialization parameters.

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

The `doFilter` method of a Filter will typically be implemented following this or some subset of the following pattern

Step 1: The method examines the request's headers.

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

Step 3: It 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.

Step 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 invokation 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, passing in the request and response it was called with, or 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.

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

Step 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.

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.

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.

SRV.6.2.2 Wrapping Requests and Responses

Central to the notion of filtering is the concept of wrapping a request or response in order that it can override behavior to perform a filtering task. In this model, the developer not only has the ability to override existing methods on the request and response objects, but to provide new API suited to a particular filtering task to a filter or target web resource down the chain. For example, the developer may wish to extend the response object with higher level output objects that the output stream or the writer, such as API that allows DOM objects to be written back to the client.

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

The same requirement of wrapper object identity applies to the case where the developer passes a wrapped request or response object into the request dispatcher; the request and response objects passed into the servlet invoked must be the same objects as were passed in.

SRV.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.

SRV.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

- `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

and optionally 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 descripor. 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 and static resources in the web application to which the Filter is to be applied. Filters can be associated with a servlet by using the `servlet-name` element. For example, the following 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 groups of servlets and static content using the `url-pattern` style of filter mapping:

```
<filter-mapping>
    <filter-name>Logging Filter</filter-name>
```

```
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```

Here the Logging Filter is applied to all the servlets and static content pages in the web application, because every request URI matches the '/*' URL pattern.

When processing a `filter-mapping` element using the `url-pattern` style, the container must determine whether the `url-pattern` matches the request URI using the path mapping rules defined in Chapter SRV.11, "CHAPTER.

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

1. The `url-pattern` matching filter-mappings in the same order that these elements appear in the deployment descriptor, and then
2. 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:

- Identifies the target web resource according to the rules of SRV.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 SRV.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` macthing chain, or invokes the target web resource if there are none.

It is expected that high performance web containers will cache filter chains so that they do not need to compute them on a per request basis.

CHAPTER SRV.7 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.

SRV.7.1 Session Tracking Mechanisms

The following sections describe approaches to tracking a user's sessions

SRV.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`.

SRV.7.1.2 SSL Sessions

Secure Sockets Layer, the encryption technology used in the HTTPS protocol, has a mechanism built into it 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.

SRV.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
```

SRV.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 fulfil this requirement, web containers commonly support the URL rewriting mechanism.

SRV.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.

The client does not yet know about the session

These conditions define the situation where the servlet container has no mechanism by which to associate a request with a previous request.

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

SRV.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 callee servlet must be different from those visible to the calling servlet.

SRV.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.

SRV.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.

SRV.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.

SRV.7.7 Important Session Semantics

J2EE.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.

SRV.7.7.2 Distributed Environments

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

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

The servlet container may throw an `IllegalArgumentException` if an object is placed into the session that is not `Serializable` or for which specific support has not been made available. The `IllegalArgumentException` must be thrown for objects where the container cannot support the mechanism necessary for migration of a session storing them.

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

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

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

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

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

SRV.7.7.3 Client Semantics

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

CHAPTER SRV.8

Dispatching Requests

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

SRV.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, wraps it with a `RequestDispatcher` object, and returns the resulting object. If no servlet can be resolved based on the given path, a `RequestDispatcher` is provided that returns the content for that path.

The `getNamedDispatcher` method takes a `String` argument indicating the name of a servlet known to the `ServletContext`. If a servlet is found, it is wrapped with a `RequestDispatcher` object and the object 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 following method is provided in the `ServletRequest` interface:

- getRequestDispatcher

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 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").

SRV.8.1.1 Query Strings in Request Dispatcher Paths

The ServletContext and ServletRequest methods that create RequestDispatcher objets 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 = “/raisons.jsp?orderno=5”;
RequestDispatcher rd = context.getRequestDispatcher(path);
rd.include(request, response);
```

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

SRV.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 Servlet interface, or instances of subclasses of the request and response wrapper classes that have been introduced for version 2.3 of the specification. In the latter case, the wrapper instances must wrap the request or response objects that the container passed into the service method.

The Container Provider must ensure that the dispatch of the request to a target servlet occurs in the same thread of the same VM as the original request.

SRV.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.

SRV.8.3.1 Included Request Parameters

Except for servlets obtained by using the `getNamedDispatcher` method, a servlet being used from within an `include` has access to the path by which it was invoked. The following request attributes are 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.

If the included servlet was obtained by using the `getNamedDispatcher` method these attributes are not set.

SRV.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 and committed, and closed by the servlet container.

SRV.8.4.1 Query String

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

SRV.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 before being propagated.

CHAPTER SRV.9 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.

SRV.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://www.mycorp.com/catalog`. All requests that start with this prefix will be routed to the `ServletContext` which represents the catalog application.

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

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

SRV.9.2 Relationship to ServletContext

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

SRV.9.3 Elements of a Web Application

A web application may consist of the following items:

- Servlets
- JSP™ Pages[1]
- Utility Classes
- Static documents (html, images, sounds, etc.)
- Client side Java applets, beans, and classes
- Descriptive meta information which ties all of the above elements together.

SRV.9.4 Deployment Hierarchies

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

SRV.9.5 Directory Structure

A web application exists as a structured hierarchy of directories. The root of this hierarchy serves as the document root for files that are part of the application. For example, for a web application with the context path `/catalog` in a web container, the `index.html` file at the base of the web application hierarchy can be served to satisfy a request from `/catalog/index.html`. The rules for matching URLs to context path are laid out in Chapter SRV.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 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, or two web applications where one context path is a substring of the other. Since requests are matched to resources case sensitively, this determination of potential conflict must be performed case sensitively as well.

[1] See the JavaServer Pages specification available from `http://java.sun.com/products/jsp`.

A special directory exists within the application hierarchy named "`WEB-INF`". This directory contains all things related to the application that aren't in the document root of the application. 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 are visible to servlet code using the `getResource` and `getResourceAsStream` method calls on the `ServletContext`. Hence, if the Application Developer needs access, from servlet code, to application specific configuration information that he does not wish to be exposed to the web client, he may place it under this directory. Since requests are matched to resource mappings case-sensitively, client requests for '`/WEB-INF/foo`', '`/WEb-iNf/foo`', for example, should not result in contents of the web application located under `/WEB-INF` being returned, nor any form of directory listing thereof.

The contents of the `WEB-INF` directory are:

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

The web application classloader must load classes from the `WEB-INF/ classes` directory first, and then from library JARs in the `WEB-INF/lib` directory.

SRV.9.5.1 Example of Application Directory Structure

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

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

SRV.9.6 Web Application Archive File

Web applications can be packaged and signed into a Web ARchive format (war) file using the standard Java Archive tools. For example, an application for issue tracking might be distributed in an archive file called `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, though its contents are visible to servlet code via the `getResource` and `getResourceAsStream` calls on the `ServletContext`.

SRV.9.7 Web Application Deployment Descriptor

The following are types of configuration and deployment information in the web application deployment descriptor (see Chapter SRV.13, "Deployment Descriptor"):

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

SRV.9.7.1 Dependencies On Extensions

When a number of applications make use of the same code or resources, they will typically be installed as library files in the container. These files are often common or standard APIs that can be used without portability being sacrificed. Files used only by one, or a few, applications will be made available for access as part of the web application.

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

Web containers are recommended to have a mechanism by which web applications can learn what JAR files containing resources and code are available, and for making them available to the application. Containers should provide a convenient procedure for editing and configuring library files or extensions.

It is recommended that Application developers provide a `META-INF/MANIFEST.MF` entry in the WAR file listing extensions, if any, needed by the WAR. The format of the manifest entry should follow standard JAR manifest format. In expressing dependencies on extensions installed on the web container, the manifest entry should follow the specification for standard extensions defined at `http://java.sun.com/j2se/1.3/docs/guide/extensions/versioning.html`.

Web Containers should be able to recognize declared dependencies expressed in the manifest entry of any of the library JARs under the `WEB-INF/lib` entry in a WAR. If a web container is not able to satisfy the dependencies declared in this manner, it should reject the application with an informative error message.

SRV.9.7.2 Web Application Classloader

The classloader that a container uses to load a servlet in a WAR must allow the developer to load any resources contained in library JARs within the WAR following normal J2SE semantics using `getResource`. It must not allow the WAR to override J2SE or Java servlet API classes. It is further recommended that the loader not allow servlets in the WAR access to the web container's implementation classes.

It is recommended also that the application class loader be implemented so that classes and resources packaged within the WAR are loaded in preference to classes and resources residing in container-wide library JARs.

SRV.9.8 Replacing a Web Application

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

SRV.9.9 Error Handling

SRV.9.9.1 Request Attributes

A web application must be able to specify that when errors occur other resources in the application are used. The specification of these resources is done in the deployment descriptor.

If the location of the error handler is a servlet or a JSP page, the request attributes in Table SRV.9-1 must be set.

Table SRV.9-1 Request Attributes and their types

Request Attributes	Type
`javax.servlet.error.status_code`	`java.lang.Integer`
`javax.servlet.error.exception_type`	`java.lang.Class`
`javax.servlet.error.message`	`java.lang.String`
`javax.servlet.error.exception`	`java.lang.Throwable`
`javax.servlet.error.request_uri`	`java.lang.String`
`javax.servlet.error.servlet_name`	`java.lang.String`

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

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

SRV.9.9.2 Error Pages

To allow developers to customize the appearance of content returned to a web client when a servlet generates an error, the deployment descriptor defines a list of error page descriptions. The syntax allows the configuration of resources to be returned by the container either when a servlet sets a status code to indicate an error on the reponse, or if the servlet generates an exception or error that propogates to the container.

If a status code indicating an error is set on the response, the container consults the list of error page declarations for the web application that use the status-code syntax and attempts a match. If there is a match, the container returns the resource as indicated by the location entry.

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

- runtime exceptions 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 heirarchy wins.

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

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

The error page mechanism described does not intervene when errors occur in servlets invoked using the `RequestDispatcher`. In this way, a servlet using the `RequestDispatcher` to call another servlet has the opportunity to handle errors generated in the servlet it calls.

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

SRV.9.10 Welcome Files

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

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

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

If a web container receives a valid partial request, the web container must examine the welcome file list defined in the deployment descriptor. The welcome file list is an ordered list of partial URLs with no trailing or leading /. The web server must append each welcome file in the order specified in the deployment descriptor to the partial request and check whether a resource 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.

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 invoking a default file servlet, or returning a directory listing. For other configurations it may return a 404 response.

Consider a web application where

- The deployment descriptor lists `index.html`, and `default.jsp` as its welcome files.
- Servlet A is an exact mapping to `/foo/bar`

The static content in the WAR is as follows

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

- A request URI of `/foo` or `/foo/` will be returned as `/foo/index.html`
- A request URI of `/catalog/` will be returned as `/catalog/default.jsp`
- A request URI of `/catalog/index.html` will cause a 404 not found
- A request URI of `/catalog/products/` may cause a 404 not found, may cause a directory listing of `shop.jsp` and/or `register.jsp`, or other behavior suitable for the container.

SRV.9.11 Web Application Environment

The Java™ 2 Platform, Enterprise Edition defines a naming environment that allows applications to easily access resources and external information without explicit knowledge of how the external information is named or organized.

As servlets are an integral component type of J2EE technology, provision has been made in the web application deployment descriptor for specifying information allowing a servlet to obtain references to resources and enterprise beans. The deployment elements that contain this information are:

- `env-entry`
- `ejb-ref`
- `ejb-local-ref`
- `resource-ref`
- `resource-env-ref`

These developer uses these elements describe certain objects that the web application requires to be registered in the JNDI namespace in the web container at runtime.

The requirements of the J2EE environment with regards to setting up the environment are described in Chapter J2EE.5 of the Java™ 2 Platform, Enterprise Edition v 1.3 specification[2]. Servlet containers that are not part of a J2EE technology compliant implementation are encouraged, but not required, to implement the application environment functionality described in the J2EE specification. If they do not implement the facilities required to support this environment, upon deploying an application that relies on them, the container should provide a warning.

Servlet containers that are part of a J2EE technology compliant implementation are required to support this syntax and should consult the Java™ 2 Platform, Enterprise Edition v 1.3 specification for more details.

Such servlet containers must support lookups of such objects and calls made to those objects when performed on a thread managed by the servlet container.

Such servlet containers should support this behavior when performed on threads created by the developer, but are not currently required to do so. Such a requirement will be added in the next version of this specification. Developers are

2. The J2EE specification is available at `http://java.sun.com/j2ee`

cautioned that depending on this capability for application-created threads is non-portable.

CHAPTER SRV.10

Application Lifecycle Events

SRV.10.1 Introduction

Support for application level events is new in version 2.3 of this specification. The application events facility gives the web application developer greater control over interactions with the `ServletContext` and `HttpSession` objects, allows for better code factorization, and increases efficiency in managing the resources that the web application uses.

SRV.10.2 Event Listeners

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

Servlet event listeners support event notifications for state changes in the `ServletContext` and `HttpSession` objects. Servlet context listeners are used to manage resources or state held at a VM 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.

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.

SRV.10.2.1 Event Types and Listener Interfaces

Events types and the listener interfaces used to monitor them are shown in **Table SRV.10-1**.

Table SRV.10-1 Events and Listener Interfaces

Event Type	Description	Listener Interface
Servlet Context Events		
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	`javax.servlet.ServletContextListener`
Changes to attributes	Attributes on the servlet context have been added, removed, or replaced.	`javax.servlet.ServletContextAttributesListener`
Http Session Events		
Lifecycle	An HttpSession has been created, invalidated, or timed out	`javax.servlet.http.HttpSessionListener`
Changes to attributes	Attributes have been added, removed, or replaced on an HttpSession	`javax.servlet.HttpSessionAttributesListener`

For details of the API, refer to the API reference in Chapter SRV.14 and Chapter SRV.15.

SRV.10.2.2 An Example of Listener Use

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

1. When the application starts up, the listener class is notified. The application logs on to the database, and stores the connection in the servlet context.
2. Servlets in the application access the connection as needed during activity in the web application.
3. When the web server is shut down, or the application is removed from the web server, the listener class is notified and the database connection is closed.

SRV.10.3 Listener Class Configuration

SRV.10.3.1 Provision of Listener Classes

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

SRV.10.3.2 Deployment Declarations

Listener classes are declared in the web application deployment descriptor using the `listener` element. They are listed by class name in the order in which they are to be invoked.

SRV.10.3.3 Listener Registration

The web container creates an instance of each listener class and registers it for event notifications prior to the processing of the first request by the application. The web container registers the listener instances according to the interfaces they implement and the order in which they appear in the deployment descriptor. During web application execution listeners are invoked in the order of their registration.

SRV.10.3.4 Notifications At Shutdown

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

SRV.10.4 Deployment Descriptor Example

The following example is the deployment grammar for registering two servlet context lifecycle listeners and an `HttpSession` 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 lifecycle events before com.acme.MyLoggingModule. Here is the deployment descriptor for this application:

```
<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>
```

SRV.10.5 Listener Instances and Threading

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

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

SRV.10.6 Distributed Containers

In distributed web containers, HttpSession instances are scoped to the particluar VM servicing session requests, and the ServletContext object is scoped to the web container's VM. Distributed containers are not required to propogate either servlet context events or HttpSession events to other VMs. Listener class instances are scoped to one per deployment descriptor declaration per Java virtual machine.

SRV.10.7 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 destinction may be determined indirectly using listeners and the `HTTPSession` API methods.

CHAPTER SRV.11 Mapping Requests to Servlets

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

SRV.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 the longest context path that matches the start of the request URL. The matched part of the URL is the context path when mapping to servlets.

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

The path used for mapping to a servlet is the request URL from the request object minus the context path. 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.

[1] Previous versions of this specification made use of these mapping techniques a suggestion rather than a requirement, allowing servlet containers to each have their different schemes for mapping client requests to servlets.

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

SRV.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 `'/*'` postfix 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 info is null.
- All other strings are used for exact matches only.

SRV.11.2.1 Implicit Mappings

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

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

SRV.11.2.2 Example Mapping Set

Consider the following set of mappings:

Table SRV.11-1 Example Set of Maps

path pattern	servlet
`/foo/bar/*`	`servlet1`
`/baz/*`	`servlet2`
`/catalog`	`servlet3`
`*.bop`	`servlet4`

The following behavior would result:

Table SRV.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.

CHAPTER SRV.12
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. Similarly to web application directory layouts and deployment descriptors, this section does not describe requirements for runtime representations. It is recommended, however, that containers implement the elements set out here as part of their runtime representations.

SRV.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.

SRV.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 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`.

SRV.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`
- `getUserPrincipal`

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. The `getUserPrincipal` method determines the principal name of the current user and returns a `java.security.Principal` object. These APIs allow servlets to make business logic decisions based on the information obtained.

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

The `isUserInRole` method 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 rolename 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</manager>
</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 be aware that the use of this default meachism may limit the flexibility in changing rolenames in the application wihout having to recompile the servlet making the call.

SRV.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 or groups in the runtime environment.

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

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

SRV.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
- HTTPS Client Authentication
- Form Based Authentication

SRV.12.5.1 HTTP Basic Authentication

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

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

SRV.12.5.2 HTTP Digest Authentication

Like HTTP Basic Authentication, HTTP Digest Authentication authenticates a user based on a username and a password. However 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, e.g. HTTPS Client Authentication. As Digest Authentication is not currently in widespread use, servlet containers are encouraged but not required to support it.

SRV.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 which allows a Developer to control the look and feel of the login screens.

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

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

1. The login form associated with the security constraint is sent to the client and the URL path triggering the authentication is stored by the container.
2. The user is asked to fill out the form, including the username and password fields.
3. The client posts the form back to the server.
4. The container attempts to authenticate the user using the information from the

form.

5. If authentication fails, the error page is returned using either a forward or a redirect, and the status code of the response is set to 401.
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.

J2EE.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.

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

Here is an example showing 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 that same as that of the authentication: for example, if the container supports single signon, such as J2EE technology compliant web containers, the user would need to reauthenticate with any of the web applications hosted on the web container.

SRV.12.5.4 HTTPS Client Authentication

End user authentication using HTTPS (HTTP over SSL) is a strong authentication mechanism. This mechanism requires the user to possess a Public Key Certificate (PKC). Currently, PKCs are useful in e-commerce applications and also for a single-signon from within the browser. Servlet containers that are not J2EE technology compliant are not required to support the HTTPS protocol.

SRV.12.6 Server Tracking of Authentication Information

As the underlying security identities (such as users and groups) to which roles are mapped in a runtime environment are environment specific rather than application specific, it is desirable to:

1. Make login mechanisms and policies a property of the environment the web application is deployed in.
2. Be able to use the same authentication information to represent a principal to all applications deployed in the same container, and
3. Require re-authentication of users only when a security policy domain boundary has been crossed.

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

SRV.12.7 Propagation of Security Identity in EJB™ Calls

A security identity, or principal, must always be provided for use in a call to an enterprise bean. The default mode in calls to enterprise beans from web applications is for the security identity of a web user to be propagated to the EJB™ container.

In other scenarios, web containers are required to allow web users that are not known to the web container or to the EJB™ container to make calls:

- Web containers are required to support access to web resources by clients that have not authenticated themselves to the container. This is the common mode of access to web resources on the Internet.
- Application code may be the sole processor of signon and customization of data based on caller identity.

In these scenarios, a web application deployment descriptor may specify a `run-as` element. When it is specified, the container must propagate the security identity of the caller to the EJB layer in terms of the security role name defined in the `run-as` element. The security role name must one of the security role names defined for the web application.

For web containers running as part of a J2EE platform, the use of run-as elements must be supported both for calls to EJB components within the same J2EE application, and for calls to EJB components deployed in other J2EE applications.

SRV.12.8 Specifying Security Constraints

Security constraints are a declarative way of annotating the intended protection of web content. A constraint consists of the following elements:

- web resource collection
- authorization constraint
- user data constraint

A web resource collection is a set of URL patterns and HTTP methods that describe a set of resources to be protected. All requests that contain a request path that matches a URL pattern described in the web resource collection is subject to the constraint. The container matches URL patterns defined in security constraints using the same algorithm described in this specification for matching client requests to servlets and static resources as described in SRV.11.1.

An authorization constraint is a set of security roles at least one of which users must belong for access to resources described by the web resource collection. If the user is not part of an allowed role, the user must be denied access to the resource requiring it. If the authorization constraint defines no roles, no user is allowed access to the portion of the web application defined by the security constraint.

A user data constraint describes requirements for the transport layer of the client server. The requirement may be for content integrity (preventing data tampering in the communication process) or for confidentiality (preventing reading while in transit). The container must at least use SSL to respond to requests to resources marked `integral` or `confidential`. If the original request was over HTTP, the container must redirect the client to the HTTPS port.

SRV.12.9 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.

CHAPTER SRV.13
Deployment Descriptor

This chapter specifies the Java^{TM} Servlet Specification, v 2.3 requirements for web container support of deployment descriptors. The deployment descriptor conveys the elements and configuration information of a web application between Application Developers, Application Assemblers, and Deployers.

For backwards compatibility of applications written to the 2.2 version of the API, web containers are also required to support the 2.2 version of the deployment descriptor. The 2.2 version is described in Appendix SRV.A.

SRV.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 Lifecyle Listener classes
- Filter Definitions and Filter Mappings
- MIME Type Mappings
- Welcome File list
- Error Pages

Security information which may also appear in the deployment descriptor is not required to be supported unless the servlet container is part of an implementation of the J2EE specification.

The following additional elements exist in the web application deployment descriptor to meet the requirements of web containers that are JSP pages enabled or part of a J2EE application server. They are not required to be supported by containers wishing to support only the servlet specification:

- `taglib`
- syntax for looking up JNDI objects (`env-entry`, `ejb-ref`, `ejb-local-ref`, `resource-ref`, `resource-env-ref`)

The DTD comments may be consulted for further description of deployment descriptor elements.

SRV.13.2 Rules for Processing the Deployment Descriptor

In this section is a listing of some general rules that web containers and developers must note concerning the processing of the deployment descriptor for a web application

- Web containers should ignore all leading whitespace characters before the first non-whitespace character, and all trailing whitespace characters after the last non-whitespace character for PCDATA within text nodes of a deployment descriptor.
- 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. It is recommended, but not required, that web containers and tools validate deployment descriptors against the DTD document for structural correctness.

  Additionally, it is recommended that they provide a level of semantic checking. For example, it should be checked that a role referenced in a security constraint has the same name as one of the security roles defined in the deployment descriptor.
  In cases of non-conformant web applications, tools and containers should inform the developer with descriptive error messages. High end application server vendors are encouraged to supply this kind of validity checking in the form of a tool separate from the container.

- URI paths specified in the deployment descriptor are assumed to be in URL-decoded form.
- Containers must attempt to canonicalize paths in the deployment descriptor. For example, paths of the form '`/a/../b`' must be interpreted as '`/b`'. Paths beginning or resolving to paths that begin with '..' are not valid paths in the deployment descriptor.
- URI paths referring to a resource relative to the root of the WAR, or a path mapping relative to the root of the WAR, unless otherwise specified, should begin with a leading '/'.
- In elements whose value is an "enumerated type", the value is case sensitive.

SRV.13.2.1 Deployment Descriptor DOCTYPE

All valid web application deployment descriptors for version 2.3 of this specification must contain the following `DOCTYPE` declaration:

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
```

SRV.13.3 DTD

The DTD that follows defines the XML grammar for a web application deployment descriptor.

```
<!--
The web-app element is the root of the deployment descriptor for
a web application.
-->

<!ELEMENT web-app (icon?, display-name?, description?,
   distributable?, context-param*, filter*, filter-mapping*,
   listener*, servlet*, servlet-mapping*, session-config?, mime-
   mapping*, welcome-file-list?, error-page*, taglib*, resource-
   env-ref*, resource-ref*, security-constraint*, login-config?,
   security-role*, env-entry*, ejb-ref*,  ejb-local-ref*)>

<!--
The auth-constraint element indicates the user roles that should
be permitted access to this resource collection. The role-name
```

```
used here must either correspond to the role-name of one of the
security-role elements defined for this web application, or be
the specially reserved role-name "*" that is a compact syntax for
indicating all roles in the web application. If both "*" and
rolenames 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.

Used in: security-constraint
-->

<!ELEMENT auth-constraint (description?, role-name*)>

<!--
The auth-method element is used to configure the authentication
mechanism for the web application. As a prerequisite to gaining
access to any web resources which are protected by an authorization
constraint, a user must have authenticated using the configured
mechanism. Legal values for this element are "BASIC", "DIGEST",
"FORM", or "CLIENT-CERT".

Used in: login-config
-->

<!ELEMENT auth-method (#PCDATA)>

<!--
The context-param element contains the declaration of a web
application’s servlet context initialization parameters.

Used in: web-app
-->

<!ELEMENT context-param (param-name, param-value, description?)>

<!--
The description element is used to provide text describing the parent
element.  The description element should include any information that
the web application war file producer wants to provide to the
consumer of the web application war file (i.e., to the Deployer).
Typically, the tools used by the web application war file consumer
will display the description when processing the parent element that
contains the description.
```

```
Used in: auth-constraint, context-param, ejb-local-ref, ejb-ref,
env-entry, filter, init-param, resource-env-ref, resource-ref, run-
as, security-role, security-role-ref, servlet, user-data-
constraint, web-app, web-resource-collection
-->

<!ELEMENT description (#PCDATA)>

<!--
The display-name element contains a short name that is intended to be
displayed by tools.  The display name need not be unique.

Used in: filter, security-constraint, servlet, web-app

Example:

<display-name>Employee Self Service</display-name>
-->

<!ELEMENT display-name (#PCDATA)>

<!--
The distributable element, by its presence in a web application
deployment descriptor, indicates that this web application is
programmed appropriately to be deployed into a distributed servlet
container

Used in: web-app
-->

<!ELEMENT distributable EMPTY>

<!--
The ejb-link element is used in the ejb-ref or ejb-local-ref
elements to specify that an EJB reference is linked to an
enterprise bean.

The name in the ejb-link element is composed of a
path name specifying the ejb-jar containing the referenced
enterprise bean with the ejb-name of the target bean appended and
separated from the path name by "#".  The path name is relative to
the war file containing the web application that is referencing the
enterprise bean.
This allows multiple enterprise beans with the same ejb-name to be
```

```
uniquely identified.

Used in: ejb-local-ref, ejb-ref

Examples:

    <ejb-link>EmployeeRecord</ejb-link>

    <ejb-link>../products/product.jar#ProductEJB</ejb-link>

-->
```

<!ELEMENT ejb-link (#PCDATA)>

```
<!--
The ejb-local-ref element is used for the declaration of a reference
to an enterprise bean's local home. The declaration consists of:

    - an optional description
    - the EJB reference name used in the code of the web application
      that's referencing the enterprise bean
    - the expected type of the referenced enterprise bean
    - the expected local home and local interfaces of the referenced
      enterprise bean
    - optional ejb-link information, used to specify the referenced
      enterprise bean

Used in: web-app
-->
```

<!ELEMENT ejb-local-ref (description?, ejb-ref-name, ejb-ref-type,
local-home, local, ejb-link?)>

```
<!--
The ejb-ref element is used for the declaration of a reference to
an enterprise bean's home. The declaration consists of:

    - an optional description
    - the EJB reference name used in the code of
      the web application that's referencing the enterprise bean
    - the expected type of the referenced enterprise bean
    - the expected home and remote interfaces of the referenced
      enterprise bean
    - optional ejb-link information, used to specify the referenced
```

```
     enterprise bean

Used in: web-app
-->
```

**<!ELEMENT ejb-ref (description?, ejb-ref-name, ejb-ref-type, home,
   remote, ejb-link?)>**

```
<!--
The ejb-ref-name element contains the name of an EJB reference. The
EJB reference is an entry in the web application’s environment and is
relative to the java:comp/env context.  The name must be unique
within the web application.

It is recommended that name is prefixed with "ejb/".

Used in: ejb-local-ref, ejb-ref

Example:

<ejb-ref-name>ejb/Payroll</ejb-ref-name>
-->
```

<!ELEMENT ejb-ref-name (#PCDATA)>

```
<!--
The ejb-ref-type element contains the expected type of the
referenced enterprise bean.

The ejb-ref-type element must be one of the following:

   <ejb-ref-type>Entity</ejb-ref-type>
   <ejb-ref-type>Session</ejb-ref-type>

Used in: ejb-local-ref, ejb-ref
-->
```

<!ELEMENT ejb-ref-type (#PCDATA)>

```
<!--
The env-entry element contains the declaration of a web application’s
environment entry. The declaration consists of an optional
description, the name of the environment entry, and an optional
value.  If a value is not specified, one must be supplied
during deployment.
```

```
-->

<!ELEMENT env-entry (description?, env-entry-name, env-entry-
   value?, env-entry-type)>

<!--
The env-entry-name element contains the name of a web applications’s
environment entry.  The name is a JNDI name relative to the
java:comp/env context.  The name must be unique within a web
application.

Example:

<env-entry-name>minAmount</env-entry-name>

Used in: env-entry
-->

<!ELEMENT env-entry-name (#PCDATA)>

<!--
The env-entry-type element contains the fully-qualified Java type of
the environment entry value that is expected by the web application’s
code.

The following are the legal values of env-entry-type:

   java.lang.Boolean
   java.lang.Byte
   java.lang.Character
   java.lang.String
   java.lang.Short
   java.lang.Integer
   java.lang.Long
   java.lang.Float
   java.lang.Double

Used in: env-entry
-->

<!ELEMENT env-entry-type (#PCDATA)>

<!--
The env-entry-value element contains the value of a web application’s
environment entry. The value must be a String that is valid for the
```

```
constructor of the specified type that takes a single String
parameter, or for java.lang.Character, a single character.

Example:

<env-entry-value>100.00</env-entry-value>

Used in: env-entry
-->

<!ELEMENT env-entry-value (#PCDATA)>

<!--
The error-code contains an HTTP error code, ex: 404

Used in: error-page
-->

<!ELEMENT error-code (#PCDATA)>

<!--
The error-page element contains a mapping between an error code
or exception type to the path of a resource in the web application

Used in: web-app
-->

<!ELEMENT error-page ((error-code | exception-type), location)>

<!--
The exception type contains a fully qualified class name of a
Java exception type.

Used in: error-page
-->

<!ELEMENT exception-type (#PCDATA)>

<!--
The extension element contains a string describing an
extension. example: "txt"

Used in: mime-mapping
-->
```

```
<!ELEMENT extension (#PCDATA)>

<!--
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.

Used in: web-app
-->

<!ELEMENT filter (icon?, filter-name, display-name?, description?,
    filter-class, init-param*)>

<!--
The fully qualified classname of the filter.

Used in: filter
-->

<!ELEMENT filter-class (#PCDATA)>

<!--
Declaration of the filter mappings in this web application. The
container uses the filter-mapping declarations to decide which
filters to apply to a request, and in what order. The container
matches the request URI to a Servlet in the normal way. To determine
which filters to apply it matches filter-mapping declarations either
on servlet-name, or on url-pattern for each filter-mapping element,
depending on which style is used. The order in which filters are
invoked is the order in which filter-mapping declarations that match
a request URI for a servlet appear in the list of filter-mapping
elements.The filter-name value must be the value of the <filter-name>
sub-elements of one of the <filter> declarations in the deployment
descriptor.

Used in: web-app
-->

<!ELEMENT filter-mapping (filter-name, (url-pattern | servlet-
    name))>

<!--
The logical name of the filter. This name is used to map the filter.
```

```
Each filter name is unique within the web application.

Used in: filter, filter-mapping
-->

<!ELEMENT filter-name (#PCDATA)>

<!--
The form-error-page element defines the location in the web app
where the error page that is displayed when login is not successful
can be found. The path begins with a leading / and is interpreted
relative to the root of the WAR.

Used in: form-login-config
-->

<!ELEMENT form-error-page (#PCDATA)>

<!--
The form-login-config element specifies the login and error pages
that should be used in form based login. If form based authentication
is not used, these elements are ignored.

Used in: login-config
-->

<!ELEMENT form-login-config (form-login-page, form-error-page)>

<!--
The form-login-page element defines the location in the web app
where the page that can be used for login can be found. The path
begins with a leading / and is interpreted relative to the root of
the WAR.

Used in: form-login-config
-->

<!ELEMENT form-login-page (#PCDATA)>

<!--
The home element contains the fully-qualified name of the enterprise
bean’s home interface.

Used in: ejb-ref
```

```
Example:

<home>com.aardvark.payroll.PayrollHome</home>
-->

<!ELEMENT home (#PCDATA)>

<!--
The http-method contains an HTTP method (GET | POST |...).

Used in: web-resource-collection
-->

<!ELEMENT http-method (#PCDATA)>

<!--
The icon element contains small-icon and large-icon elements that
specify the file names for small and a large GIF or JPEG icon images
used to represent the parent element in a GUI tool.

Used in: filter, servlet, web-app
-->

<!ELEMENT icon (small-icon?, large-icon?)>

<!--
The init-param element contains a name/value pair as an
initialization param of the servlet

Used in: filter, servlet
-->

<!ELEMENT init-param (param-name, param-value, description?)>

<!--
The jsp-file element contains the full path to a JSP file within
the web application beginning with a ‘/’.

Used in: servlet
-->

<!ELEMENT jsp-file (#PCDATA)>

<!--
The large-icon element contains the name of a file
```

```
containing a large (32 x 32) icon image. The file
name is a relative path within the web application's
war file.

The image may be either in the JPEG or GIF format.
The icon can be used by tools.

Used in: icon

Example:

<large-icon>employee-service-icon32x32.jpg</large-icon>
-->

<!ELEMENT large-icon (#PCDATA)>

<!--
The listener element indicates the deployment properties for a web
application listener bean.

Used in: web-app
-->

<!ELEMENT listener (listener-class)>

<!--
The listener-class element declares a class in the application must
be registered as a web application listener bean. The value is the
fully qualified classname of the listener class.

Used in: listener
-->

<!ELEMENT listener-class (#PCDATA)>

<!--
The load-on-startup element indicates that this servlet should be
loaded (instantiated and have its init() called) on the startup
of the web application. The optional contents of
these 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-start-up value.

Used in: servlet
-->

<!ELEMENT load-on-startup (#PCDATA)>

<!--

The local element contains the fully-qualified name of the
enterprise bean’s local interface.

Used in: ejb-local-ref
-->

<!ELEMENT local (#PCDATA)>

<!--
The local-home element contains the fully-qualified name of the
enterprise bean’s local home interface.

Used in: ejb-local-ref
-->

<!ELEMENT local-home (#PCDATA)>

<!--
The 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 ‘/’.

Used in: error-page
-->

<!ELEMENT location (#PCDATA)>

<!--
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.

Used in: web-app
```

```
-->

<!ELEMENT login-config (auth-method?, realm-name?, form-login-
    config?)>

<!--
The mime-mapping element defines a mapping between an extension
and a mime type.

Used in: web-app
-->

<!ELEMENT mime-mapping (extension, mime-type)>

<!--
The mime-type element contains a defined mime type. example:
"text/plain"

Used in: mime-mapping
-->

<!ELEMENT mime-type (#PCDATA)>

<!--
The param-name element contains the name of a parameter. Each
parameter name must be unique in the web application.

Used in: context-param, init-param
-->

<!ELEMENT param-name (#PCDATA)>

<!--
The param-value element contains the value of a parameter.

Used in: context-param, init-param
-->

<!ELEMENT param-value (#PCDATA)>


<!--
The realm name element specifies the realm name to use in HTTP
Basic authorization.
```

```
Used in: login-config
-->

<!ELEMENT realm-name (#PCDATA)>

<!--
The remote element contains the fully-qualified name of the
enterprise bean’s remote interface.

Used in: ejb-ref

Example:

<remote>com.wombat.empl.EmployeeService</remote>
-->

<!ELEMENT remote (#PCDATA)>

<!--
The res-auth element specifies whether the web application code signs
on programmatically to the resource manager, or whether the Container
will sign on to the resource manager on behalf of the web
application. In the latter case, the Container uses information that
is supplied by the Deployer.

The value of this element must be one of the two following:

    <res-auth>Application</res-auth>
    <res-auth>Container</res-auth>

Used in: resource-ref
-->

<!ELEMENT res-auth (#PCDATA)>

<!--
The res-ref-name element specifies the name of a resource manager
connection factory reference.  The name is a JNDI name relative to
the
java:comp/env context.  The name must be unique within a web
application.

Used in: resource-ref
-->
<!ELEMENT res-ref-name (#PCDATA)>
```

```
<!--
The res-sharing-scope element specifies whether connections obtained
through the given resource manager connection factory reference can
be
shared. The value of this element, if specified, must be one of the
two following:

    <res-sharing-scope>Shareable</res-sharing-scope>
    <res-sharing-scope>Unshareable</res-sharing-scope>

The default value is Shareable.

Used in: resource-ref
-->
```

<!ELEMENT res-sharing-scope (#PCDATA)>

```
<!--
The res-type element specifies the type of the data source. The type
is specified by the fully qualified Java language class or interface
expected to be implemented by the data source.

Used in: resource-ref
-->
```

<!ELEMENT res-type (#PCDATA)>

```
<!--
The resource-env-ref element contains a declaration of a web
application's reference to an administered object associated with a
resource in the web application's environment.  It consists of an
optional description, the resource environment reference name, and
an indication of the resource environment reference type expected by
the web application code.

Used in: web-app

Example:

<resource-env-ref>
    <resource-env-ref-name>jms/StockQueue</resource-env-ref-name>
    <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
</resource-env-ref>
-->
```

```
<!ELEMENT resource-env-ref (description?, resource-env-ref-name,

       resource-env-ref-type)>

<!--
The resource-env-ref-name element specifies the name of a resource
environment reference; its value is the environment entry name used
in the web application code.  The name is a JNDI name relative to the
java:comp/env context and must be unique within a web application.

Used in: resource-env-ref
-->

<!ELEMENT resource-env-ref-name (#PCDATA)>

<!--
The resource-env-ref-type element specifies the type of a resource
environment reference.  It is the fully qualified name of a Java
language class or interface.

Used in: resource-env-ref
-->

<!ELEMENT resource-env-ref-type (#PCDATA)>

<!--
The resource-ref element contains a declaration of a web
application’s reference to an external resource. It consists of an
optional description, the resource manager connection factory
reference name, the indication of the resource manager connection
factory type expected by the web application code, the type of
authentication (Application or Container), and an optional
specification of the shareability of connections obtained from the
resource (Shareable or Unshareable).

Used in: web-app

Example:

    <resource-ref>
   <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
   <res-type>javax.sql.DataSource</res-type>
   <res-auth>Container</res-auth>
   <res-sharing-scope>Shareable</res-sharing-scope>
    </resource-ref>
```

```
-->

<!ELEMENT resource-ref (description?, res-ref-name, res-type, res-
   auth, res-sharing-scope?)>

<!--
The role-link element is a reference to a defined security role. The
role-link element must contain the name of one of the security roles
defined in the security-role elements.

Used in: security-role-ref
-->

<!ELEMENT role-link (#PCDATA)>

<!--
The role-name element contains the name of a security role.
The name must conform to the lexical rules for an NMTOKEN.

Used in: auth-constraint, run-as, security-role, security-role-ref
-->

<!ELEMENT role-name (#PCDATA)>

<!--
The run-as element specifies the run-as identity to be used for the
execution of the web application. It contains an optional
description, and
the name of a security role.

Used in: servlet
-->

<!ELEMENT run-as (description?, role-name)>

<!--
The security-constraint element is used to associate security
constraints with one or more web resource collections

Used in: web-app
-->

<!ELEMENT security-constraint (display-name?, web-resource-
   collection+, auth-constraint?, user-data-constraint?)>
```

```
<!--
The security-role element contains the definition of a security
role. The definition consists of an optional description of the
security role, and the security role name.

Used in: web-app

Example:

    <security-role>
   <description>
       This role includes all employees who are authorized
       to access the employee service application.
   </description>
   <role-name>employee</role-name>
    </security-role>
-->
```

<!ELEMENT security-role (description?, role-name)>

```
<!--
The security-role-ref element contains the declaration of a security
role reference in the web application’s code. The declaration
consists
of an optional description, the security role name used in the code,
and an optional link to a security role. If the security role is not
specified, the Deployer must choose an appropriate security role.

The value of the role-name element must be the String used as the
parameter to the EJBContext.isCallerInRole(String roleName) method
or the HttpServletRequest.isUserInRole(String role) method.

Used in: servlet

-->
```

<!ELEMENT security-role-ref (description?, role-name, role-link?)>

```
<!--
The servlet element contains the declarative data of a
servlet. If a jsp-file is specified and the load-on-startup element
is present, then the JSP should be precompiled and loaded.

Used in: web-app
```

```
-->

<!ELEMENT servlet (icon?, servlet-name, display-name?, description?,
    (servlet-class|jsp-file), init-param*, load-on-startup?, run-
    as?, security-role-ref*)>

<!--
The servlet-class element contains the fully qualified class name
of the servlet.

Used in: servlet
-->

<!ELEMENT servlet-class (#PCDATA)>

<!--
The servlet-mapping element defines a mapping between a servlet
and a url pattern

Used in: web-app
-->

<!ELEMENT servlet-mapping (servlet-name, url-pattern)>

<!--
The servlet-name element contains the canonical name of the
servlet. Each servlet name is unique within the web application.

Used in: filter-mapping, servlet, servlet-mapping
-->

<!ELEMENT servlet-name (#PCDATA)>

<!--
The session-config element defines the session parameters for
this web application.

Used in: web-app
-->

<!ELEMENT session-config (session-timeout?)>

<!--
The session-timeout element 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
behaviour of sessions is never to time out.

Used in: session-config
-->
```

<!ELEMENT session-timeout (#PCDATA)>

```
<!--
The small-icon element contains the name of a file
containing a small (16 x 16) icon image. The file
name is a relative path within the web application’s
war file.

The image may be either in the JPEG or GIF format.
The icon can be used by tools.

Used in: icon

Example:

<small-icon>employee-service-icon16x16.jpg</small-icon>
-->
```

<!ELEMENT small-icon (#PCDATA)>

```
<!--
The taglib element is used to describe a JSP tag library.

Used in: web-app
-->
```

<!ELEMENT taglib (taglib-uri, taglib-location)>

```
<!--
the taglib-location element contains the location (as a resource
relative to the root of the web application) where to find the Tag
Libary Description file for the tag library.

Used in: taglib
-->
```

<!ELEMENT taglib-location (#PCDATA)>

```
<!--
The taglib-uri element describes a URI, relative to the location
of the web.xml document, identifying a Tag Library used in the Web
Application.

Used in: taglib
-->

<!ELEMENT taglib-uri (#PCDATA)>

<!--
The transport-guarantee element specifies that the communication
between client and server should be NONE, INTEGRAL, or
CONFIDENTIAL. NONE means that the application does not require any
transport guarantees. A value of INTEGRAL means that the application
requires that the data sent between the client and server be sent in
such a way that it can't be changed in transit. CONFIDENTIAL means
that the application requires that the data be transmitted in a
fashion that prevents other entities from observing the contents of
the transmission. In most cases, the presence of the INTEGRAL or
CONFIDENTIAL flag will indicate that the use of SSL is required.

Used in: user-data-constraint
-->

<!ELEMENT transport-guarantee (#PCDATA)>

<!--
The url-pattern element contains the url pattern of the mapping. Must
follow the rules specified in Section 11.2 of the Servlet API
Specification.

Used in: filter-mapping, servlet-mapping, web-resource-collection
-->

<!ELEMENT url-pattern (#PCDATA)>

<!--
The user-data-constraint element is used to indicate how data
communicated between the client and container should be protected.

Used in: security-constraint
-->

<!ELEMENT user-data-constraint (description?, transport-guarantee)>
```

```
<!--
The web-resource-collection element is used to identify a subset
of the resources and HTTP methods on those resources within a web
application to which a security constraint applies. If no HTTP
methods are specified, then the security constraint applies to all
HTTP methods.

Used in: security-constraint
-->

<!ELEMENT web-resource-collection (web-resource-name, description?,
    url-pattern*, http-method*)>


<!--
The web-resource-name contains the name of this web resource
collection.

Used in: web-resource-collection
-->

<!ELEMENT web-resource-name (#PCDATA)>

<!--
The welcome-file element contains file name to use as a default
welcome file, such as index.html

Used in: welcome-file-list
-->

<!ELEMENT welcome-file (#PCDATA)>

<!--
The welcome-file-list contains an ordered list of welcome files
elements.

Used in: web-app
-->

<!ELEMENT welcome-file-list (welcome-file+)>

<!--
The ID mechanism is to allow tools that produce additional deployment
information (i.e., information beyond the standard deployment
descriptor information) to store the non-standard information in a
separate file, and easily refer from these tool-specific files to the
information in the standard deployment descriptor.

Tools are not allowed to add the non-standard information into the
standard deployment descriptor.
-->


<!ATTLIST auth-constraint id ID #IMPLIED>

<!ATTLIST auth-method id ID #IMPLIED>

<!ATTLIST context-param id ID #IMPLIED>

<!ATTLIST description id ID #IMPLIED>

<!ATTLIST display-name id ID #IMPLIED>

<!ATTLIST distributable id ID #IMPLIED>

<!ATTLIST ejb-link id ID #IMPLIED>

<!ATTLIST ejb-local-ref id ID #IMPLIED>

<!ATTLIST ejb-ref id ID #IMPLIED>

<!ATTLIST ejb-ref-name id ID #IMPLIED>

<!ATTLIST ejb-ref-type id ID #IMPLIED>

<!ATTLIST env-entry id ID #IMPLIED>

<!ATTLIST env-entry-name id ID #IMPLIED>

<!ATTLIST env-entry-type id ID #IMPLIED>

<!ATTLIST env-entry-value id ID #IMPLIED>

<!ATTLIST error-code id ID #IMPLIED>

<!ATTLIST error-page id ID #IMPLIED>

<!ATTLIST exception-type id ID #IMPLIED>
```

```
<!ATTLIST extension id ID #IMPLIED>

<!ATTLIST filter id ID #IMPLIED>

<!ATTLIST filter-class id ID #IMPLIED>

<!ATTLIST filter-mapping id ID #IMPLIED>

<!ATTLIST filter-name id ID #IMPLIED>

<!ATTLIST form-error-page id ID #IMPLIED>

<!ATTLIST form-login-config id ID #IMPLIED>

<!ATTLIST form-login-page id ID #IMPLIED>

<!ATTLIST home id ID #IMPLIED>

<!ATTLIST http-method id ID #IMPLIED>

<!ATTLIST icon id ID #IMPLIED>

<!ATTLIST init-param id ID #IMPLIED>

<!ATTLIST jsp-file id ID #IMPLIED>

<!ATTLIST large-icon id ID #IMPLIED>

<!ATTLIST listener id ID #IMPLIED>

<!ATTLIST listener-class id ID #IMPLIED>

<!ATTLIST load-on-startup id ID #IMPLIED>

<!ATTLIST local id ID #IMPLIED>

<!ATTLIST local-home id ID #IMPLIED>

<!ATTLIST location id ID #IMPLIED>

<!ATTLIST login-config id ID #IMPLIED>

<!ATTLIST mime-mapping id ID #IMPLIED>

<!ATTLIST mime-type id ID #IMPLIED>
```

```
<!ATTLIST param-name id ID #IMPLIED>

<!ATTLIST param-value id ID #IMPLIED>

<!ATTLIST realm-name id ID #IMPLIED>

<!ATTLIST remote id ID #IMPLIED>

<!ATTLIST res-auth id ID #IMPLIED>

<!ATTLIST res-ref-name id ID #IMPLIED>

<!ATTLIST res-sharing-scope id ID #IMPLIED>

<!ATTLIST res-type id ID #IMPLIED>

<!ATTLIST resource-env-ref id ID #IMPLIED>

<!ATTLIST resource-env-ref-name id ID #IMPLIED>

<!ATTLIST resource-env-ref-type id ID #IMPLIED>

<!ATTLIST resource-ref id ID #IMPLIED>

<!ATTLIST role-link id ID #IMPLIED>

<!ATTLIST role-name id ID #IMPLIED>

<!ATTLIST run-as id ID #IMPLIED>

<!ATTLIST security-constraint id ID #IMPLIED>

<!ATTLIST security-role id ID #IMPLIED>

<!ATTLIST security-role-ref id ID #IMPLIED>

<!ATTLIST servlet id ID #IMPLIED>

<!ATTLIST servlet-class id ID #IMPLIED>

<!ATTLIST servlet-mapping id ID #IMPLIED>

<!ATTLIST servlet-name id ID #IMPLIED>

<!ATTLIST session-config id ID #IMPLIED>
```

```
<!ATTLIST session-timeout id ID #IMPLIED>

<!ATTLIST small-icon id ID #IMPLIED>

<!ATTLIST taglib id ID #IMPLIED>

<!ATTLIST taglib-location id ID #IMPLIED>

<!ATTLIST taglib-uri id ID #IMPLIED>

<!ATTLIST transport-guarantee id ID #IMPLIED>

<!ATTLIST url-pattern id ID #IMPLIED>

<!ATTLIST user-data-constraint id ID #IMPLIED>

<!ATTLIST web-app id ID #IMPLIED>

<!ATTLIST web-resource-collection id ID #IMPLIED>

<!ATTLIST web-resource-name id ID #IMPLIED>

<!ATTLIST welcome-file id ID #IMPLIED>

<!ATTLIST welcome-file-list id ID #IMPLIED>
```

SRV.13.4 Examples

The following examples illustrate the usage of the definitions listed above DTD.

SRV.13.4.1 A Basic Example

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Appli-
cation 2.3//EN" "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">

<web-app>
    <display-name>A Simple Application</display-name>
    <context-param>
        <param-name>Webmaster</param-name>
        <param-value>webmaster@mycorp.com</param-value>
    </context-param>
    <servlet>
        <servlet-name>catalog</servlet-name>
        <servlet-class>com.mycorp.CatalogServlet
        </servlet-class>
        <init-param>
            <param-name>catalog</param-name>
            <param-value>Spring</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>catalog</servlet-name>
        <url-pattern>/catalog/*</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>
    <mime-mapping>
        <extension>pdf</extension>
        <mime-type>application/pdf</mime-type>
    </mime-mapping>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
        <welcome-file>index.html</welcome-file>
        <welcome-file>index.htm</welcome-file>
    </welcome-file-list>
    <error-page>
        <error-code>404</error-code>
        <location>/404.html</location>
    </error-page>
</web-app>
```

SRV.13.4.2 An Example of Security

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Appli-
cation 2.2//EN" "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
    <display-name>A Secure Application</display-name>
    <security-role>
        <role-name>manager</role-name>
    </security-role>
    <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>
    <servlet-mapping>
        <servlet-name>catalog</servlet-name>
        <url-pattern>/catalog/*</url-pattern>
    </servlet-mapping>
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>SalesInfo
            </web-resource-name>
            <url-pattern>/salesinfo/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>manager</role-name>
        </auth-constraint>
        <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL
            </transport-guarantee>
        </user-data-constraint>
    </security-constraint>
</web-app>
```

CHAPTER SRV.14

javax.servlet

This chapter describes the javax.servlet package. The chapter includes content that is generated automatically from javadoc embedded in the actual Java classes and interfaces. This allows the creation of a single, authoritative, specification document.

SRV.14.1 Generic Servlet Interfaces and Classes

The *javax.servlet package* contains a number of classes and interfaces that describe and define the contracts between a servlet class and the runtime environment provided for an instance of such a class by a conforming servlet container.

The *Servlet* interface is the central abstraction of the 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 servlet API that implement the *Servlet* interface are *GenericServlet* and *HttpServlet* . For most purposes, developers will extend *HttpServlet* to implement their servlets while implementing web applications employing the HTTP protocol..

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.

SRV.14.2 The javax.servlet package

The following section summarizes the javax.servlet package:

Class Summary

Interfaces

Filter	A filter is an object than perform filtering tasks on either the request to a resource (a servlet or static content), or on the response from a resource, or both. Filters perform filtering in the `doFilter` method.
FilterChain	A FilterChain is an object provided by the servlet container to the developer giving a view into the invocation chain of a filtered request for a resource.
FilterConfig	A filter configuration object used by a servlet container used to pass information to a filter during initialization.
RequestDispatcher	Defines an object that receives requests from the client and sends them to any resource (such as a servlet, HTML file, or JSP file) on the server.
Servlet	Defines methods that all servlets must implement.
ServletConfig	A servlet configuration object used by a servlet container used to pass information to a servlet during initialization.
ServletContext	Defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file.
ServletContextAt-tributeListener	Implementations of this interface recieve notifications of changes to the attribute list on the servlet context of a web application.
ServletContextListener	Implementations of this interface recieve notifications about changes to the servlet context of the web application they are part of.
ServletRequest	Defines an object to provide client request information to a servlet.
ServletResponse	Defines an object to assist a servlet in sending a response to the client.

Class Summary	
SingleThreadModel	Ensures that servlets handle only one request at a time.
Classes	
GenericServlet	Defines a generic, protocol-independent servlet.
ServletContextAttributeEvent	This is the event class for notifications about changes to the attributes of the servlet context of a web application.
ServletContextEvent	This is the event class for notifications about changes to the servlet context of a web application.
ServletInputStream	Provides an input stream for reading binary data from a client request, including an efficient `readLine` method for reading data one line at a time.
ServletOutputStream	Provides an output stream for sending binary data to the client.
ServletRequestWrapper	Provides a convenient implementation of the ServletRequest interface that can be subclassed by developers wishing to adapt the request to a Servlet.
ServletResponseWrapper	Provides a convenient implementation of the ServletResponse interface that can be subclassed by developers wishing to adapt the response from a Servlet.
Exceptions	
ServletException	Defines a general exception a servlet can throw when it encounters difficulty.
UnavailableException	Defines an exception that a servlet or filter throws to indicate that it is permanently or temporarily unavailable.

SRV.14.2.1 Filter

`public interface` **`Filter`**

A filter is an object than perform filtering tasks on either the request to a resource (a servlet or static content), or on the response from a resource, or both.

Filters perform filtering in the `doFilter` method. Every Filter has access to a FilterConfig object from which it can obtain its initialization parameters, a reference to the ServletContext which it can use, for example, to load resources needed for filtering tasks.

Filters are configured in the deployment descriptor of a web application

Examples that have been identified for this design are
1) Authentication Filters
2) Logging and Auditing Filters
3) Image conversion Filters
4) Data compression Filters
5) Encryption Filters
6) Tokenizing Filters
7) Filters that trigger resource access events
8) XSL/T filters
9) Mime-type chain Filter

Since: Servlet 2.3

SRV.14.2.1.1 Methods

destroy()

```
public void destroy()
```

Called by the web container to indicate to a filter that it is being taken out of service. This method is only called once all threads within the filter's doFilter method have exited or after a timeout period has passed. After the web container calls this method, it will not call the doFilter method again on this instance of the filter.

This method gives the filter an opportunity to clean up any resources that are being held (for example, memory, file handles, threads) and make sure that any persistent state is synchronized with the filter's current state in memory.

doFilter(ServletRequest, ServletResponse, FilterChain)

```
public void doFilter(ServletRequest request,
    ServletResponse response, FilterChain chain)
    throws IOException, ServletException
```

The `doFilter` method of the Filter is called by the container each time a request/response pair is passed through the chain due to a client request for a resource at the end of the chain. The FilterChain passed in to this method allows the Filter to pass on the request and response to the next entity in the chain.

A typical implementation of this method would follow the following pattern:-
1. Examine the request
2. Optionally wrap the request object with a custom implementation to filter content or headers for input filtering
3. Optionally wrap the response object with a custom implementation to filter content or headers for output filtering
4. a) **Either** invoke the next entity in the chain using the FilterChain object (`chain.doFilter()`),
4. b) **or** not pass on the request/response pair to the next entity in the filter chain to block the request processing
5. Directly set headers on the response after invokation of the next entity in ther filter chain.

Throws:
`ServletException`, `IOException`

init(FilterConfig)

```
public void init(FilterConfig filterConfig)
    throws ServletException
```

Called by the web container to indicate to a filter that it is being placed into service. The servlet container calls the init method exactly once after instantiating the filter. The init method must complete successfully before the filter is asked to do any filtering work.

The web container cannot place the filter into service if the init method either
1.Throws a ServletException
2.Does not return within a time period defined by the web container

Throws:
`ServletException`

SRV.14.2.2 FilterChain

```
public interface FilterChain
```

A FilterChain is an object provided by the servlet container to the developer giving a view into the invocation chain of a filtered request for a resource. Filters use the FilterChain to invoke the next filter in the chain, or if the calling filter is the last filter in the chain, to invoke the rosource at the end of the chain.

Since: Servlet 2.3

See Also: `Filter`

SRV.14.2.2.1 Methods

doFilter(ServletRequest, ServletResponse)

```
public void doFilter(ServletRequest request,
    ServletResponse response)
    throws IOException, ServletException
```

Causes the next filter in the chain to be invoked, or if the calling filter is the last filter in the chain, causes the resource at the end of the chain to be invoked.

Parameters:
`request` - the request to pass along the chain.

`response` - the response to pass along the chain.

Throws:
`ServletException`, `IOException`

Since: 2.3

SRV.14.2.3 FilterConfig

```
public interface FilterConfig
```

A filter configuration object used by a servlet container used to pass information to a filter during initialization.

Since: Servlet 2.3

See Also: `Filter`

SRV.14.2.3.1 Methods

getFilterName()

```
public java.lang.String getFilterName()
```

Returns the filter-name of this filter as defined in the deployment descriptor.

getInitParameter(String)

```
public java.lang.String getInitParameter(java.lang.String name)
```

Returns a `String` containing the value of the named initialization parameter, or `null` if the parameter does not exist.

Parameters:
`name` - a `String` specifying the name of the initialization parameter

Returns: a `String` containing the value of the initialization parameter

getInitParameterNames()

`public java.util.Enumeration **getInitParameterNames**()`

Returns the names of the servlet's initialization parameters as an `Enumeration` of `String` objects, or an empty `Enumeration` if the servlet has no initialization parameters.

Returns: an `Enumeration` of `String` objects containing the names of the servlet's initialization parameters

getServletContext()

`public ServletContext **getServletContext**()`

Returns a reference to the `ServletContext` in which the caller is executing.

Returns: a `ServletContext` object, used by the caller to interact with its servlet container

See Also: `ServletContext`

SRV.14.2.4 GenericServlet

```
public abstract class GenericServlet implements
javax.servlet.Servlet, javax.servlet.ServletConfig,
java.io.Serializable
```

All Implemented Interfaces: `java.io.Serializable`, `Servlet`, `ServletConfig`

Direct Known Subclasses: `javax.servlet.http.HttpServlet`

Defines a generic, protocol-independent servlet. To write an HTTP servlet for use on the Web, extend `javax.servlet.http.HttpServlet` instead.

`GenericServlet` implements the `Servlet` and `ServletConfig` interfaces. `GenericServlet` may be directly extended by a servlet, although it's more common to extend a protocol-specific subclass such as `HttpServlet`.

`GenericServlet` makes writing servlets easier. It provides simple versions of the lifecycle methods `init` and `destroy` and of the methods in the `ServletConfig` interface. `GenericServlet` also implements the `log` method, declared in the `ServletContext` interface.

To write a generic servlet, you need only override the abstract `service` method.

SRV.14.2.4.1 Constructors

GenericServlet()

`public **GenericServlet**()`

Does nothing. All of the servlet initialization is done by one of the `init` methods.

SRV.14.2.4.2 Methods

destroy()

`public void destroy()`

Called by the servlet container to indicate to a servlet that the servlet is being taken out of service. See `Servlet.destroy()` .

Specified By: `Servlet.destroy()` in interface `Servlet`

getInitParameter(String)

`public java.lang.String getInitParameter(java.lang.String name)`

Returns a `String` containing the value of the named initialization parameter, or `null` if the parameter does not exist. See `ServletConfig.getInitParameter(String)` .

This method is supplied for convenience. It gets the value of the named parameter from the servlet's `ServletConfig` object.

Specified By: `ServletConfig.getInitParameter(String)` in interface `ServletConfig`

Parameters:
`name` - a `String` specifying the name of the initialization parameter

Returns: String a `String` containing the value of the initalization parameter

getInitParameterNames()

`public java.util.Enumeration getInitParameterNames()`

Returns the names of the servlet's initialization parameters as an `Enumeration` of `String` objects, or an empty `Enumeration` if the servlet has no initialization parameters. See `ServletConfig.getInitParameterNames()` .

This method is supplied for convenience. It gets the parameter names from the servlet's `ServletConfig` object.

Specified By: `ServletConfig.getInitParameterNames()` in interface `ServletConfig`

Returns: Enumeration an enumeration of `String` objects containing the names of the servlet's initialization parameters

getServletConfig()

`public ServletConfig **getServletConfig**()`

Returns this servlet's `ServletConfig` object.

Specified By: `Servlet.getServletConfig()` in interface `Servlet`

Returns: ServletConfig the `ServletConfig` object that initialized this servlet

getServletContext()

`public ServletContext **getServletContext**()`

Returns a reference to the `ServletContext` in which this servlet is running. See `ServletConfig.getServletContext()` .

This method is supplied for convenience. It gets the context from the servlet's `ServletConfig` object.

Specified By: `ServletConfig.getServletContext()` in interface `ServletConfig`

Returns: ServletContext the `ServletContext` object passed to this servlet by the `init` method

getServletInfo()

`public java.lang.String **getServletInfo**()`

Returns information about the servlet, such as author, version, and copyright. By default, this method returns an empty string. Override this method to have it return a meaningful value. See `Servlet.getServletInfo()` .

Specified By: `Servlet.getServletInfo()` in interface `Servlet`

Returns: String information about this servlet, by default an empty string

getServletName()

`public java.lang.String **getServletName**()`

Returns the name of this servlet instance. See `ServletConfig.getServletName()` .

Specified By: `ServletConfig.getServletName()` in interface `ServletConfig`

Returns: the name of this servlet instance

init()

```
public void init()
    throws ServletException
```

A convenience method which can be overridden so that there's no need to call `super.init(config)`.

Instead of overriding `init(ServletConfig)` , simply override this method and it will be called by `GenericServlet.init(ServletConfig config)`. The `ServletConfig` object can still be retrieved via `getServletConfig()` .

Throws:
`ServletException` - if an exception occurs that interrupts the servlet's normal operation

init(ServletConfig)

```
public void init(ServletConfig config)
    throws ServletException
```

Called by the servlet container to indicate to a servlet that the servlet is being placed into service. See `Servlet.init(ServletConfig)` .

This implementation stores the `ServletConfig` object it receives from the servlet container for later use. When overriding this form of the method, call `super.init(config)`.

Specified By: `Servlet.init(ServletConfig)` in interface `Servlet`

Parameters:
`config` - the `ServletConfig` object that contains configutation information for this servlet

Throws:
`ServletException` - if an exception occurs that interrupts the servlet's normal operation

See Also: `UnavailableException`

log(String)

```
public void log(java.lang.String msg)
```

Writes the specified message to a servlet log file, prepended by the servlet's name. See `ServletContext.log(String)` .

Parameters:
`msg` - a `String` specifying the message to be written to the log file

log(String, Throwable)

```
public void log(java.lang.String message, java.lang.Throwable t)
```

Writes an explanatory message and a stack trace for a given `Throwable` exception to the servlet log file, prepended by the servlet's name. See `ServletContext.log(String, Throwable)` .

Parameters:
`message` - a `String` that describes the error or exception

`t` - the `java.lang.Throwable` error or exception

service(ServletRequest, ServletResponse)

```
public abstract void service(ServletRequest req,
    ServletResponse res)
    throws ServletException, IOException
```

Called by the servlet container to allow the servlet to respond to a request. See `Servlet.service(ServletRequest, ServletResponse)` .

This method is declared abstract so subclasses, such as `HttpServlet`, must override it.

Specified By: `Servlet.service(ServletRequest, ServletResponse)` in interface `Servlet`

Parameters:
`req` - the `ServletRequest` object that contains the client's request

`res` - the `ServletResponse` object that will contain the servlet's response

Throws:
`ServletException` - if an exception occurs that interferes with the servlet's normal operation occurred

`IOException` - if an input or output exception occurs

SRV.14.2.5 RequestDispatcher

```
public interface RequestDispatcher
```

Defines an object that receives requests from the client and sends them to any resource (such as a servlet, HTML file, or JSP file) on the server. The servlet container creates the `RequestDispatcher` object, which is used as a wrapper around a server resource located at a particular path or given by a particular name.

This interface is intended to wrap servlets, but a servlet container can create `RequestDispatcher` objects to wrap any type of resource.

See Also: `ServletContext.getRequestDispatcher(String)`, `ServletContext.getNamedDispatcher(String)`, `ServletRequest.getRequestDispatcher(String)`

SRV.14.2.5.1 Methods

forward(ServletRequest, ServletResponse)

```
public void forward(ServletRequest request,
    ServletResponse response)
    throws ServletException, IOException
```

Forwards a request from a servlet to another resource (servlet, JSP file, or HTML file) on the server. This method allows one servlet to do preliminary processing of a request and another resource to generate the response.

For a `RequestDispatcher` obtained via `getRequestDispatcher()`, the `ServletRequest` object has its path elements and parameters adjusted to match the path of the target resource.

`forward` should be called before the response has been committed to the client (before response body output has been flushed). If the response already has been committed, this method throws an `IllegalStateException`. Uncommitted output in the response buffer is automatically cleared before the forward.

The request and response parameters must be either the same objects as were passed to the calling servlet's service method or be subclasses of the `ServletRequestWrapper` or `ServletResponseWrapper` classes that wrap them.

Parameters:

`request` - a `ServletRequest` object that represents the request the client makes of the servlet

`response` - a `ServletResponse` object that represents the response the servlet returns to the client

Throws:

`ServletException` - if the target resource throws this exception

`IOException` - if the target resource throws this exception

`IllegalStateException` - if the response was already committed

include(ServletRequest, ServletResponse)

```
public void include(ServletRequest request,
    ServletResponse response)
    throws ServletException, IOException
```

Includes the content of a resource (servlet, JSP page, HTML file) in the response. In essence, this method enables programmatic server-side includes.

The `ServletResponse` object has its path elements and parameters remain unchanged from the caller's. The included servlet cannot change the response status code or set headers; any attempt to make a change is ignored.

The request and response parameters must be either the same objects as were passed to the calling servlet's service method or be subclasses of the

`ServletRequestWrapper` or `ServletResponseWrapper` classes that wrap them.

Parameters:
`request` - a `ServletRequest` object that contains the client's request

`response` - a `ServletResponse` object that contains the servlet's response

Throws:
`ServletException` - if the included resource throws this exception

`IOException` - if the included resource throws this exception

SRV.14.2.6 Servlet

`public interface` **`Servlet`**

All Known Implementing Classes: `GenericServlet`

Defines methods that all servlets must implement.

A servlet is a small Java program that runs within a Web server. Servlets receive and respond to requests from Web clients, usually across HTTP, the HyperText Transfer Protocol.

To implement this interface, you can write a generic servlet that extends `javax.servlet.GenericServlet` or an HTTP servlet that extends `javax.servlet.http.HttpServlet`.

This interface defines methods to initialize a servlet, to service requests, and to remove a servlet from the server. These are known as life-cycle methods and are called in the following sequence:

1.The servlet is constructed, then initialized with the `init` method.

2.Any calls from clients to the `service` method are handled.

3.The servlet is taken out of service, then destroyed with the `destroy` method, then garbage collected and finalized.

In addition to the life-cycle methods, this interface provides the `getServlet-Config` method, which the servlet can use to get any startup information, and the `getServletInfo` method, which allows the servlet to return basic information about itself, such as author, version, and copyright.

See Also: `GenericServlet`, `javax.servlet.http.HttpServlet`

SRV.14.2.6.1 Methods

destroy()

`public void` **`destroy`**`()`

Called by the servlet container to indicate to a servlet that the servlet is being taken out of service. This method is only called once all threads within the servlet's `service` method have exited or after a timeout period has passed. After the servlet container calls this method, it will not call the `service` method again on this servlet.

This method gives the servlet an opportunity to clean up any resources that are being held (for example, memory, file handles, threads) and make sure that any persistent state is synchronized with the servlet's current state in memory.

getServletConfig()

`public ServletConfig` **`getServletConfig`**`()`

Returns a `ServletConfig` object, which contains initialization and startup parameters for this servlet. The `ServletConfig` object returned is the one passed to the `init` method.

Implementations of this interface are responsible for storing the `Servlet-Config` object so that this method can return it. The `GenericServlet` class, which implements this interface, already does this.

Returns: the `ServletConfig` object that initializes this servlet

See Also: `init(ServletConfig)`

getServletInfo()

`public java.lang.String` **`getServletInfo`**`()`

Returns information about the servlet, such as author, version, and copyright.

The string that this method returns should be plain text and not markup of any kind (such as HTML, XML, etc.).

Returns: a `String` containing servlet information

init(ServletConfig)

```
public void init(ServletConfig config)
    throws ServletException
```

Called by the servlet container to indicate to a servlet that the servlet is being placed into service.

The servlet container calls the `init` method exactly once after instantiating the servlet. The `init` method must complete successfully before the servlet can receive any requests.

The servlet container cannot place the servlet into service if the `init` method

1. Throws a `ServletException`

2. Does not return within a time period defined by the Web server

Parameters:
`config` - a `ServletConfig` object containing the servlet's configuration and initialization parameters

Throws:
`ServletException` - if an exception has occurred that interferes with the servlet's normal operation

See Also: `UnavailableException`, `getServletConfig()`

service(ServletRequest, ServletResponse)

```
public void service(ServletRequest req, ServletResponse res)
    throws ServletException, IOException
```

Called by the servlet container to allow the servlet to respond to a request.

This method is only called after the servlet's `init()` method has completed successfully.

The status code of the response always should be set for a servlet that throws or sends an error.

Servlets typically run inside multithreaded servlet containers that can handle multiple requests concurrently. Developers must be aware to synchronize access to any shared resources such as files, network connections, and as well as the servlet's class and instance variables. More information on multi-threaded programming in Java is available in the Java tutorial on multi-threaded programming (http://java.sun.com/Series/Tutorial/java/threads/multithreaded.html).

Parameters:
`req` - the `ServletRequest` object that contains the client's request

`res` - the `ServletResponse` object that contains the servlet's response

Throws:
`ServletException` - if an exception occurs that interferes with the servlet's normal operation

`IOException` - if an input or output exception occurs

SRV.14.2.7 ServletConfig

```
public interface ServletConfig
```

All Known Implementing Classes: `GenericServlet`

A servlet configuration object used by a servlet container used to pass information to a servlet during initialization.

SRV.14.2.7.1 Methods

getInitParameter(String)

`public java.lang.String` **`getInitParameter`**`(java.lang.String name)`

Returns a `String` containing the value of the named initialization parameter, or `null` if the parameter does not exist.

Parameters:
`name` - a `String` specifying the name of the initialization parameter

Returns: a `String` containing the value of the initialization parameter

getInitParameterNames()

`public java.util.Enumeration` **`getInitParameterNames`**`()`

Returns the names of the servlet's initialization parameters as an `Enumeration` of `String` objects, or an empty `Enumeration` if the servlet has no initialization parameters.

Returns: an `Enumeration` of `String` objects containing the names of the servlet's initialization parameters

getServletContext()

`public ServletContext` **`getServletContext`**`()`

Returns a reference to the `ServletContext` in which the caller is executing.

Returns: a `ServletContext` object, used by the caller to interact with its servlet container

See Also: `ServletContext`

getServletName()

`public java.lang.String` **`getServletName`**`()`

Returns the name of this servlet instance. The name may be provided via server administration, assigned in the web application deployment descriptor, or for an unregistered (and thus unnamed) servlet instance it will be the servlet's class name.

Returns: the name of the servlet instance

SRV.14.2.8 ServletContext

`public interface` **`ServletContext`**

Defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file.

There is one context per "web application" per Java Virtual Machine. (A "web application" is a collection of servlets and content installed under a specific subset of the server's URL namespace such as `/catalog` and possibly installed via a `.war` file.)

In the case of a web application marked "distributed" in its deployment descriptor, there will be one context instance for each virtual machine. In this situation, the context cannot be used as a location to share global information (because the information won't be truly global). Use an external resource like a database instead.

The `ServletContext` object is contained within the `ServletConfig` object, which the Web server provides the servlet when the servlet is initialized.

See Also: `Servlet.getServletConfig()`, `ServletConfig.getServletContext()`

SRV.14.2.8.1 Methods

getAttribute(String)

`public java.lang.Object` **`getAttribute`**`(java.lang.String name)`

Returns the servlet container attribute with the given name, or `null` if there is no attribute by that name. An attribute allows a servlet container to give the servlet additional information not already provided by this interface. See your server documentation for information about its attributes. A list of supported attributes can be retrieved using `getAttributeNames`.

The attribute is returned as a `java.lang.Object` or some subclass. Attribute names should follow the same convention as package names. The Java Servlet API specification reserves names matching `java.*`, `javax.*`, and `sun.*`.

Parameters:
`name` - a `String` specifying the name of the attribute

Returns: an `Object` containing the value of the attribute, or `null` if no attribute exists matching the given name

See Also: `getAttributeNames()`

getAttributeNames()

`public java.util.Enumeration` **`getAttributeNames`**`()`

Returns an `Enumeration` containing the attribute names available within this servlet context. Use the `getAttribute(String)` method with an attribute name to get the value of an attribute.

Returns: an `Enumeration` of attribute names

See Also: `getAttribute(String)`

getContext(String)

`public ServletContext` **`getContext`**`(java.lang.String uripath)`

Returns a `ServletContext` object that corresponds to a specified URL on the server.

This method allows servlets to gain access to the context for various parts of the server, and as needed obtain `RequestDispatcher` objects from the context. The given path must be begin with "/", is interpreted relative to the server's document root and is matched against the context roots of other web applications hosted on this container.

In a security conscious environment, the servlet container may return `null` for a given URL.

Parameters:
`uripath` - a `String` specifying the context path of another web application in the container.

Returns: the `ServletContext` object that corresponds to the named URL, or null if either none exists or the container wishes to restrict this access.

See Also: `RequestDispatcher`

getInitParameter(String)

`public java.lang.String` **`getInitParameter`**`(java.lang.String name)`

Returns a `String` containing the value of the named context-wide initialization parameter, or `null` if the parameter does not exist.

This method can make available configuration information useful to an entire "web application". For example, it can provide a webmaster's email address or the name of a system that holds critical data.

Parameters:
`name` - a `String` containing the name of the parameter whose value is requested

Returns: a `String` containing at least the servlet container name and version number

See Also: `ServletConfig.getInitParameter(String)`

getInitParameterNames()

`public java.util.Enumeration **getInitParameterNames**()`

Returns the names of the context's initialization parameters as an `Enumeration` of `String` objects, or an empty `Enumeration` if the context has no initialization parameters.

Returns: an `Enumeration` of `String` objects containing the names of the context's initialization parameters

See Also: `ServletConfig.getInitParameter(String)`

getMajorVersion()

`public int **getMajorVersion**()`

Returns the major version of the Java Servlet API that this servlet container supports. All implementations that comply with Version 2.3 must have this method return the integer 2.

Returns: 2

getMimeType(String)

`public java.lang.String **getMimeType**(java.lang.String file)`

Returns the MIME type of the specified file, or `null` if the MIME type is not known. The MIME type is determined by the configuration of the servlet container, and may be specified in a web application deployment descriptor. Common MIME types are "`text/html`" and "`image/gif`".

Parameters:
`file` - a `String` specifying the name of a file

Returns: a `String` specifying the file's MIME type

getMinorVersion()

`public int **getMinorVersion**()`

Returns the minor version of the Servlet API that this servlet container supports. All implementations that comply with Version 2.3 must have this method return the integer 3.

Returns: 3

getNamedDispatcher(String)

`public RequestDispatcher **getNamedDispatcher**(java.lang.String name)`

Returns a `RequestDispatcher` object that acts as a wrapper for the named servlet.

Servlets (and JSP pages also) may be given names via server administration or via a web application deployment descriptor. A servlet instance can determine its name using `ServletConfig.getServletName()` .

This method returns `null` if the `ServletContext` cannot return a `RequestDispatcher` for any reason.

Parameters:
`name` - a `String` specifying the name of a servlet to wrap

Returns: a `RequestDispatcher` object that acts as a wrapper for the named servlet

See Also: `RequestDispatcher`, `getContext(String)`, `ServletConfig.getServletName()`

getRealPath(String)

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

Returns a `String` containing the real path for a given virtual path. For example, the path "/index.html" returns the absolute file path on the server's filesystem would be served by a request for "http://host/contextPath/index.html", where contextPath is the context path of this ServletContext..

The real path returned will be in a form appropriate to the computer and operating system on which the servlet container is running, including the proper path separators. This method returns `null` if the servlet container cannot translate the virtual path to a real path for any reason (such as when the content is being made available from a `.war` archive).

Parameters:
`path` - a `String` specifying a virtual path

Returns: a `String` specifying the real path, or null if the translation cannot be performed

getRequestDispatcher(String)

```
public RequestDispatcher getRequestDispatcher(java.lang.String
    path)
```

Returns a `RequestDispatcher` object that acts as a wrapper for the resource located at the given path. A `RequestDispatcher` object can be used to forward a request to the resource or to include the resource in a response. The resource can be dynamic or static.

The pathname must begin with a "/" and is interpreted as relative to the current context root. Use `getContext` to obtain a `RequestDispatcher` for resources in foreign contexts. This method returns `null` if the `ServletContext` cannot return a `RequestDispatcher`.

Parameters:
`path` - a `String` specifying the pathname to the resource

Returns: a `RequestDispatcher` object that acts as a wrapper for the resource at the specified path

See Also: `RequestDispatcher`, `getContext(String)`

getResource(String)

```
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.

This method allows the servlet container to make a resource available to servlets from any source. Resources can be located on a local or remote file system, in a database, or in a `.war` file.

The servlet container must implement the URL handlers and `URLConnection` objects that are necessary to access the resource.

This method returns `null` if no resource is mapped to the pathname.

Some containers may allow writing to the URL returned by this method using the methods of the URL class.

The resource content is returned directly, so be aware that requesting a `.jsp` page returns the JSP source code. Use a `RequestDispatcher` instead to include results of an execution.

This method has a different purpose than `java.lang.Class.getResource`, which looks up resources based on a class loader. This method does not use class loaders.

Parameters:
`path` - a `String` specifying the path to the resource

Returns: the resource located at the named path, or `null` if there is no resource at that path

Throws:
`MalformedURLException` - if the pathname is not given in the correct form

getResourceAsStream(String)

`public java.io.InputStream` **`getResourceAsStream`**`(java.lang.String path)`

Returns the resource located at the named path as an `InputStream` object.

The data in the `InputStream` can be of any type or length. The path must be specified according to the rules given in `getResource`. This method returns `null` if no resource exists at the specified path.

Meta-information such as content length and content type that is available via `getResource` method is lost when using this method.

The servlet container must implement the URL handlers and `URLConnection` objects necessary to access the resource.

This method is different from `java.lang.Class.getResourceAsStream`, which uses a class loader. This method allows servlet containers to make a resource available to a servlet from any location, without using a class loader.

Parameters:
`name` - a `String` specifying the path to the resource

Returns: the `InputStream` returned to the servlet, or `null` if no resource exists at the specified path

getResourcePaths(String)

`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. Paths indicating subdirectory paths end with a '/'. The returned paths are all relative to the root of the web application and have a leading '/'. For example, for a web application containing

/welcome.html
/catalog/index.html
/catalog/products.html
/catalog/offers/books.html
/catalog/offers/music.html
/customer/login.jsp
/WEB-INF/web.xml
/WEB-INF/classes/com.acme.OrderServlet.class,

getResourcePaths("/") returns {"/welcome.html", "/catalog/", "/customer/", "/WEB-INF/"}
getResourcePaths("/catalog/") returns {"/catalog/index.html", "/catalog/products.html", "/catalog/offers/"}.

Parameters:
`the` - partial path used to match the resources, which must start with a /

Returns: a Set containing the directory listing, or null if there are no resources in the web application whose path begins with the supplied path.

Since: Servlet 2.3

getServerInfo()

```
public java.lang.String getServerInfo()
```

Returns the name and version of the servlet container on which the servlet is running.

The form of the returned string is *servername/versionnumber.* For example, the JavaServer Web Development Kit may return the string `JavaServer Web Dev Kit/1.0`.

The servlet container may return other optional information after the primary string in parentheses, for example, `JavaServer Web Dev Kit/1.0 (JDK 1.1.6; Windows NT 4.0 x86)`.

Returns: a `String` containing at least the servlet container name and version number

getServlet(String)

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

Deprecated. As of Java Servlet API 2.1, with no direct replacement.

This method was originally defined to retrieve a servlet from a `ServletContext`. In this version, this method always returns `null` and remains only to preserve binary compatibility. This method will be permanently removed in a future version of the Java Servlet API.

In lieu of this method, servlets can share information using the `ServletContext` class and can perform shared business logic by invoking methods on common non-servlet classes.

Throws:
`ServletException`

getServletContextName()

```
public java.lang.String getServletContextName()
```

Returns the name of this web application correponding to this ServletContext as specified in the deployment descriptor for this web application by the display-name element.

Returns: The name of the web application or null if no name has been declared in the deployment descriptor.

Since: Servlet 2.3

getServletNames()

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

Deprecated. As of Java Servlet API 2.1, with no replacement.

This method was originally defined to return an `Enumeration` of all the servlet names known to this context. In this version, this method always returns an empty `Enumeration` and remains only to preserve binary compatibility. This method will be permanently removed in a future version of the Java Servlet API.

getServlets()

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

Deprecated. As of Java Servlet API 2.0, with no replacement.

This method was originally defined to return an `Enumeration` of all the servlets known to this servlet context. In this version, this method always returns an empty enumeration and remains only to preserve binary compatibility. This method will be permanently removed in a future version of the Java Servlet API.

log(Exception, String)

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

Deprecated. As of Java Servlet API 2.1, 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.

log(String)

```
public void log(java.lang.String msg)
```

Writes the specified message to a servlet log file, usually an event log. The name and type of the servlet log file is specific to the servlet container.

Parameters:
`msg` - a `String` specifying the message to be written to the log file

log(String, Throwable)

```
public void log(java.lang.String message,
    java.lang.Throwable throwable)
```

Writes an explanatory message and a stack trace for a given `Throwable` exception to the servlet log file. The name and type of the servlet log file is specific to the servlet container, usually an event log.

Parameters:

`message` - a `String` that describes the error or exception

`throwable` - the `Throwable` error or exception

removeAttribute(String)

```
public void removeAttribute(java.lang.String name)
```

Removes the attribute with the given name from the servlet context. After removal, subsequent calls to `getAttribute(String)` to retrieve the attribute's value will return `null`.

If listeners are configured on the `ServletContext` the container notifies them accordingly.

Parameters:

`name` - a `String` specifying the name of the attribute to be removed

setAttribute(String, Object)

```
public void setAttribute(java.lang.String name,
    java.lang.Object object)
```

Binds an object to a given attribute name in this servlet context. If the name specified is already used for an attribute, this method will replace the attribute with the new to the new attribute.

If listeners are configured on the `ServletContext` the container notifies them accordingly.

If a null value is passed, the effect is the same as calling `removeAttribute()`.

Attribute names should follow the same convention as package names. The Java Servlet API specification reserves names matching `java.*`, `javax.*`, and `sun.*`.

Parameters:

`name` - a `String` specifying the name of the attribute

`object` - an `Object` representing the attribute to be bound

SRV.14.2.9 ServletContextAttributeEvent

```
public class ServletContextAttributeEvent extends
javax.servlet.ServletContextEvent
```

All Implemented Interfaces: `java.io.Serializable`

This is the event class for notifications about changes to the attributes of the servlet context of a web application.

Since: v 2.3

See Also: `ServletContextAttributeListener`

SRV.14.2.9.1 Constructors

ServletContextAttributeEvent(ServletContext, String, Object)

```
public ServletContextAttributeEvent(ServletContext source,
    java.lang.String name, java.lang.Object value)
```

Construct a ServletContextAttributeEvent from the given context for the given attribute name and attribute value.

SRV.14.2.9.2 Methods

getName()

```
public java.lang.String getName()
```

Return the name of the attribute that changed on the ServletContext.

getValue()

```
public java.lang.Object getValue()
```

Returns the value of the attribute that has been added removed or replaced. If the attribute was added, this is the value of the attribute. If the attrubute was removed, this is the value of the removed attribute. If the attribute was replaced, this is the old value of the attribute.

SRV.14.2.10 ServletContextAttributeListener

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

All Superinterfaces: `java.util.EventListener`

Implementations of this interface recieve notifications of changes to the attribute list on the servlet context of a web application. To recieve notification events, the implementation class must be configured in the deployment descriptor for the web application.

Since: v 2.3

See Also: `ServletContextAttributeEvent`

SRV.14.2.10.1 Methods

attributeAdded(ServletContextAttributeEvent)

`public void attributeAdded(ServletContextAttributeEvent scab)`

Notification that a new attribute was added to the servlet context. Called after the attribute is added.

attributeRemoved(ServletContextAttributeEvent)

`public void attributeRemoved(ServletContextAttributeEvent scab)`

Notification that an existing attribute has been remved from the servlet context. Called after the attribute is removed.

attributeReplaced(ServletContextAttributeEvent)

`public void attributeReplaced(ServletContextAttributeEvent scab)`

Notification that an attribute on the servlet context has been replaced. Called after the attribute is replaced.

SRV.14.2.11 ServletContextEvent

`public class ServletContextEvent extends java.util.EventObject`

All Implemented Interfaces: `java.io.Serializable`

Direct Known Subclasses: `ServletContextAttributeEvent`

This is the event class for notifications about changes to the servlet context of a web application.

Since: v 2.3

See Also: `ServletContextListener`

SRV.14.2.11.1 Constructors

ServletContextEvent(ServletContext)

`public ServletContextEvent(ServletContext source)`

Construct a ServletContextEvent from the given context.

Parameters:
`source` - - the ServletContext that is sending the event.

SRV.14.2.11.2 Methods

getServletContext()

`public ServletContext getServletContext()`

Return the ServletContext that changed.

Returns: the ServletContext that sent the event.

SRV.14.2.12 ServletContextListener

`public interface ServletContextListener extends java.util.EventListener`

All Superinterfaces: `java.util.EventListener`

Implementations of this interface recieve notifications about changes to the servlet context of the web application they are part of. To recieve notification events, the implementation class must be configured in the deployment descriptor for the web application.

Since: v 2.3

See Also: `ServletContextEvent`

SRV.14.2.12.1 Methods

contextDestroyed(ServletContextEvent)

`public void contextDestroyed(ServletContextEvent sce)`

Notification that the servlet context is about to be shut down.

contextInitialized(ServletContextEvent)

`public void contextInitialized(ServletContextEvent sce)`

Notification that the web application is ready to process requests.

SRV.14.2.13 ServletException

`public class ServletException extends java.lang.Exception`

All Implemented Interfaces: `java.io.Serializable`

Direct Known Subclasses: `UnavailableException`

Defines a general exception a servlet can throw when it encounters difficulty.

SRV.14.2.13.1 Constructors

ServletException()

`public ServletException()`

Constructs a new servlet exception.

ServletException(String)

```
public ServletException(java.lang.String message)
```

Constructs a new servlet exception with the specified message. The message can be written to the server log and/or displayed for the user.

Parameters:
`message` - a `String` specifying the text of the exception message

ServletException(String, Throwable)

```
public ServletException(java.lang.String message,
   java.lang.Throwable rootCause)
```

Constructs a new servlet exception when the servlet needs to throw an exception and include a message about the "root cause" exception that interfered with its normal operation, including a description message.

Parameters:
`message` - a `String` containing the text of the exception message

`rootCause` - the `Throwable` exception that interfered with the servlet's normal operation, making this servlet exception necessary

ServletException(Throwable)

```
public ServletException(java.lang.Throwable rootCause)
```

Constructs a new servlet exception when the servlet needs to throw an exception and include a message about the "root cause" exception that interfered with its normal operation. The exception's message is based on the localized message of the underlying exception.

This method calls the `getLocalizedMessage` method on the `Throwable` exception to get a localized exception message. When subclassing `ServletException`, this method can be overridden to create an exception message designed for a specific locale.

Parameters:
`rootCause` - the `Throwable` exception that interfered with the servlet's normal operation, making the servlet exception necessary

SRV.14.2.13.2 Methods

getRootCause()

```
public java.lang.Throwable getRootCause()
```

Returns the exception that caused this servlet exception.

Returns: the `Throwable` that caused this servlet exception

SRV.14.2.14 ServletInputStream

`public abstract class` **`ServletInputStream`** `extends java.io.InputStream`

Provides an input stream for reading binary data from a client request, including an efficient `readLine` method for reading data one line at a time. With some protocols, such as HTTP POST and PUT, a `ServletInputStream` object can be used to read data sent from the client.

A `ServletInputStream` object is normally retrieved via the `ServletRequest.getInputStream()` method.

This is an abstract class that a servlet container implements. Subclasses of this class must implement the `java.io.InputStream.read()` method.

See Also: `ServletRequest`

SRV.14.2.14.1 Constructors

ServletInputStream()

`protected` **`ServletInputStream`**`()`

Does nothing, because this is an abstract class.

SRV.14.2.14.2 Methods

readLine(byte[], int, int)

```
public int readLine(byte[] b, int off, int len)
    throws IOException
```

Reads the input stream, one line at a time. Starting at an offset, reads bytes into an array, until it reads a certain number of bytes or reaches a newline character, which it reads into the array as well.

This method returns -1 if it reaches the end of the input stream before reading the maximum number of bytes.

Parameters:
`b` - an array of bytes into which data is read

`off` - an integer specifying the character at which this method begins reading

`len` - an integer specifying the maximum number of bytes to read

Returns: an integer specifying the actual number of bytes read, or -1 if the end of the stream is reached

Throws:
`IOException` - if an input or output exception has occurred

SRV.14.2.15 ServletOutputStream

```
public abstract class ServletOutputStream extends
java.io.OutputStream
```

Provides an output stream for sending binary data to the client. A `ServletOutputStream` object is normally retrieved via the `ServletResponse.getOutputStream()` method.

This is an abstract class that the servlet container implements. Subclasses of this class must implement the `java.io.OutputStream.write(int)` method.

See Also: `ServletResponse`

SRV.14.2.15.1 Constructors

ServletOutputStream()

```
protected ServletOutputStream()
```

Does nothing, because this is an abstract class.

SRV.14.2.15.2 Methods

print(boolean)

```
public void print(boolean b)
    throws IOException
```

Writes a `boolean` value to the client, with no carriage return-line feed (CRLF) character at the end.

Parameters:
`b` - the `boolean` value to send to the client

Throws:
`IOException` - if an input or output exception occurred

print(char)

```
public void print(char c)
    throws IOException
```

Writes a character to the client, with no carriage return-line feed (CRLF) at the end.

Parameters:
`c` - the character to send to the client

Throws:
`IOException` - if an input or output exception occurred

print(double)

```
public void print(double d)
    throws IOException
```

Writes a `double` value to the client, with no carriage return-line feed (CRLF) at the end.

Parameters:
`d` - the `double` value to send to the client

Throws:
`IOException` - if an input or output exception occurred

print(float)

```
public void print(float f)
    throws IOException
```

Writes a `float` value to the client, with no carriage return-line feed (CRLF) at the end.

Parameters:
`f` - the `float` value to send to the client

Throws:
`IOException` - if an input or output exception occurred

print(int)

```
public void print(int i)
    throws IOException
```

Writes an int to the client, with no carriage return-line feed (CRLF) at the end.

Parameters:
`i` - the int to send to the client

Throws:
`IOException` - if an input or output exception occurred

print(long)

```
public void print(long l)
    throws IOException
```

Writes a `long` value to the client, with no carriage return-line feed (CRLF) at the end.

Parameters:
`l` - the `long` value to send to the client

Throws:

`IOException` - if an input or output exception occurred

print(String)

```
public void print(java.lang.String s)
    throws IOException
```

Writes a `String` to the client, without a carriage return-line feed (CRLF) character at the end.

Parameters:
`s` - the `String</code to send to the client`

Throws:
`IOException` - if an input or output exception occurred

println()

```
public void println()
    throws IOException
```

Writes a carriage return-line feed (CRLF) to the client.

Throws:
`IOException` - if an input or output exception occurred

println(boolean)

```
public void println(boolean b)
    throws IOException
```

Writes a `boolean` value to the client, followed by a carriage return-line feed (CRLF).

Parameters:
`b` - the `boolean` value to write to the client

Throws:
`IOException` - if an input or output exception occurred

println(char)

```
public void println(char c)
    throws IOException
```

Writes a character to the client, followed by a carriage return-line feed (CRLF).

Parameters:
`c` - the character to write to the client

Throws:
`IOException` - if an input or output exception occurred

println(double)

```
public void println(double d)
    throws IOException
```

Writes a `double` value to the client, followed by a carriage return-line feed (CRLF).

Parameters:
`d` - the `double` value to write to the client

Throws:
`IOException` - if an input or output exception occurred

println(float)

```
public void println(float f)
    throws IOException
```

Writes a `float` value to the client, followed by a carriage return-line feed (CRLF).

Parameters:
`f` - the `float` value to write to the client

Throws:
`IOException` - if an input or output exception occurred

println(int)

```
public void println(int i)
    throws IOException
```

Writes an int to the client, followed by a carriage return-line feed (CRLF) character.

Parameters:
`i` - the int to write to the client

Throws:
`IOException` - if an input or output exception occurred

println(long)

```
public void println(long l)
    throws IOException
```

Writes a `long` value to the client, followed by a carriage return-line feed (CRLF).

Parameters:
`l` - the `long` value to write to the client

Throws:

`IOException` - if an input or output exception occurred

println(String)

```
public void println(java.lang.String s)
    throws IOException
```

Writes a `String` to the client, followed by a carriage return-line feed (CRLF).

Parameters:
`s` - the String to write to the client

Throws:
`IOException` - if an input or output exception occurred

SRV.14.2.16 ServletRequest

```
public interface ServletRequest
```

All Known Subinterfaces: `javax.servlet.http.HttpServletRequest`

All Known Implementing Classes: `ServletRequestWrapper`

Defines an object to provide client request information to a servlet. The servlet container creates a `ServletRequest` object and passes it as an argument to the servlet's `service` method.

A `ServletRequest` object provides data including parameter name and values, attributes, and an input stream. Interfaces that extend `ServletRequest` can provide additional protocol-specific data (for example, HTTP data is provided by `javax.servlet.http.HttpServletRequest` .

See Also: `javax.servlet.http.HttpServletRequest`

SRV.14.2.16.1 Methods

getAttribute(String)

```
public java.lang.Object getAttribute(java.lang.String name)
```

Returns the value of the named attribute as an `Object`, or `null` if no attribute of the given name exists.

Attributes can be set two ways. The servlet container may set attributes to make available custom information about a request. For example, for requests made using HTTPS, the attribute `javax.servlet.request.X509Certificate` can be used to retrieve information on the certificate of the client. Attributes can also be set programatically using `setAttribute(String, Object)` . This allows information to be embedded into a request before a `RequestDispatcher` call.

Attribute names should follow the same conventions as package names. This specification reserves names matching `java.*`, `javax.*`, and `sun.*`.

Parameters:
`name` - a `String` specifying the name of the attribute

Returns: an `Object` containing the value of the attribute, or `null` if the attribute does not exist

getAttributeNames()

```
public java.util.Enumeration getAttributeNames()
```

Returns an `Enumeration` containing the names of the attributes available to this request. This method returns an empty `Enumeration` if the request has no attributes available to it.

Returns: an `Enumeration` of strings containing the names of the request's attributes

getCharacterEncoding()

```
public java.lang.String getCharacterEncoding()
```

Returns the name of the character encoding used in the body of this request. This method returns `null` if the request does not specify a character encoding

Returns: a `String` containing the name of the chararacter encoding, or `null` if the request does not specify a character encoding

getContentLength()

```
public int getContentLength()
```

Returns the length, in bytes, of the request body and made available by the input stream, or -1 if the length is not known. For HTTP servlets, same as the value of the CGI variable CONTENT_LENGTH.

Returns: an integer containing the length of the request body or -1 if the length is not known

getContentType()

```
public java.lang.String getContentType()
```

Returns the MIME type of the body of the request, or `null` if the type is not known. For HTTP servlets, same as the value of the CGI variable CONTENT_TYPE.

Returns: a `String` containing the name of the MIME type of the request, or null if the type is not known

getInputStream()

```
public ServletInputStream getInputStream()
    throws IOException
```

Retrieves the body of the request as binary data using a `ServletInputStream` . Either this method or `getReader()` may be called to read the body, not both.

Returns: a `ServletInputStream` object containing the body of the request

Throws:
`IllegalStateException` - if the `getReader()` method has already been called for this request

`IOException` - if an input or output exception occurred

getLocale()

```
public java.util.Locale getLocale()
```

Returns the preferred `Locale` that the client will accept content in, based on the Accept-Language header. If the client request doesn't provide an Accept-Language header, this method returns the default locale for the server.

Returns: the preferred `Locale` for the client

getLocales()

```
public java.util.Enumeration getLocales()
```

Returns an `Enumeration` of `Locale` objects indicating, in decreasing order starting with the preferred locale, the locales that are acceptable to the client based on the Accept-Language header. If the client request doesn't provide an Accept-Language header, this method returns an `Enumeration` containing one `Locale`, the default locale for the server.

Returns: an `Enumeration` of preferred `Locale` objects for the client

getParameter(String)

```
public java.lang.String getParameter(java.lang.String name)
```

Returns the value of a request parameter as a `String`, or `null` if the parameter does not exist. Request parameters are extra information sent with the request. For HTTP servlets, parameters are contained in the query string or posted form data.

You should only use this method when you are sure the parameter has only one value. If the parameter might have more than one value, use `getParameterValues(String)` .

If you use this method with a multivalued parameter, the value returned is equal to the first value in the array returned by `getParameterValues`.

If the parameter data was sent in the request body, such as occurs with an HTTP POST request, then reading the body directly via `getInputStream()` or `getReader()` can interfere with the execution of this method.

Parameters:
`name` - a `String` specifying the name of the parameter

Returns: a `String` representing the single value of the parameter

See Also: `getParameterValues(String)`

getParameterMap()

`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. For HTTP servlets, parameters are contained in the query string or posted form data.

Returns: an immutable java.util.Map containing parameter names as keys and parameter values as map values. The keys in the parameter map are of type String. The values in the parameter map are of type String array.

getParameterNames()

`public java.util.Enumeration` **`getParameterNames`**`()`

Returns an `Enumeration` of `String` objects containing the names of the parameters contained in this request. If the request has no parameters, the method returns an empty `Enumeration`.

Returns: an `Enumeration` of `String` objects, each `String` containing the name of a request parameter; or an empty `Enumeration` if the request has no parameters

getParameterValues(String)

`public java.lang.String[]` **`getParameterValues`**`(java.lang.String name)`

Returns an array of `String` objects containing all of the values the given request parameter has, or `null` if the parameter does not exist.

If the parameter has a single value, the array has a length of 1.

Parameters:
`name` - a `String` containing the name of the parameter whose value is requested

Returns: an array of `String` objects containing the parameter's values

See Also: `getParameter(String)`

getProtocol()

```
public java.lang.String getProtocol()
```

Returns the name and version of the protocol the request uses in the form *protocol/majorVersion.minorVersion*, for example, HTTP/1.1. For HTTP servlets, the value returned is the same as the value of the CGI variable `SERVER_PROTOCOL`.

Returns: a `String` containing the protocol name and version number

getReader()

```
public java.io.BufferedReader getReader()
    throws IOException
```

Retrieves the body of the request as character data using a `BufferedReader`. The reader translates the character data according to the character encoding used on the body. Either this method or `getInputStream()` may be called to read the body, not both.

Returns: a `BufferedReader` containing the body of the request

Throws:
`UnsupportedEncodingException` - if the character set encoding used is not supported and the text cannot be decoded

`IllegalStateException` - if `getInputStream()` method has been called on this request

`IOException` - if an input or output exception occurred

See Also: `getInputStream()`

getRealPath(String)

```
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.

getRemoteAddr()

```
public java.lang.String getRemoteAddr()
```

Returns the Internet Protocol (IP) address of the client that sent the request. For HTTP servlets, same as the value of the CGI variable `REMOTE_ADDR`.

Returns: a `String` containing the IP address of the client that sent the request

getRemoteHost()

`public java.lang.String **getRemoteHost**()`

Returns the fully qualified name of the client that sent the request. If the engine cannot or chooses not to resolve the hostname (to improve performance), this method returns the dotted-string form of the IP address. For HTTP servlets, same as the value of the CGI variable `REMOTE_HOST`.

Returns: a `String` containing the fully qualified name of the client

getRequestDispatcher(String)

`public RequestDispatcher **getRequestDispatcher**(java.lang.String path)`

Returns a `RequestDispatcher` object that acts as a wrapper for the resource located at the given path. A `RequestDispatcher` object can be used to forward a request to the resource or to include the resource in a response. The resource can be dynamic or static.

The pathname specified may be relative, although it cannot extend outside the current servlet context. If the path begins with a "/" it is interpreted as relative to the current context root. This method returns `null` if the servlet container cannot return a `RequestDispatcher`.

The difference between this method and `ServletContext.getRequestDispatcher(String)` is that this method can take a relative path.

Parameters:
`path` - a `String` specifying the pathname to the resource

Returns: a `RequestDispatcher` object that acts as a wrapper for the resource at the specified path

See Also: `RequestDispatcher`, `ServletContext.getRequestDispatcher(String)`

getScheme()

`public java.lang.String **getScheme**()`

Returns the name of the scheme used to make this request, for example, `http`, `https`, or `ftp`. Different schemes have different rules for constructing URLs, as noted in RFC 1738.

Returns: a `String` containing the name of the scheme used to make this request

getServerName()

`public java.lang.String` **`getServerName`**`()`

Returns the host name of the server that received the request. For HTTP servlets, same as the value of the CGI variable `SERVER_NAME`.

Returns: a `String` containing the name of the server to which the request was sent

getServerPort()

`public int` **`getServerPort`**`()`

Returns the port number on which this request was received. For HTTP servlets, same as the value of the CGI variable `SERVER_PORT`.

Returns: an integer specifying the port number

isSecure()

`public boolean` **`isSecure`**`()`

Returns a boolean indicating whether this request was made using a secure channel, such as HTTPS.

Returns: a boolean indicating if the request was made using a secure channel

removeAttribute(String)

`public void` **`removeAttribute`**`(java.lang.String name)`

Removes an attribute from this request. This method is not generally needed as attributes only persist as long as the request is being handled.

Attribute names should follow the same conventions as package names. Names beginning with `java.*`, `javax.*`, and `com.sun.*`, are reserved for use by Sun Microsystems.

Parameters:
`name` - a `String` specifying the name of the attribute to remove

setAttribute(String, Object)

`public void` **`setAttribute`**`(java.lang.String name, java.lang.Object o)`

Stores an attribute in this request. Attributes are reset between requests. This method is most often used in conjunction with `RequestDispatcher` .

Attribute names should follow the same conventions as package names. Names beginning with `java.*`, `javax.*`, and `com.sun.*`, are reserved for use by Sun Microsystems.
If the value passed in is null, the effect is the same as calling `removeAttribute(String)` .

Parameters:
`name` - a `String` specifying the name of the attribute

`o` - the `Object` to be stored

setCharacterEncoding(String)

```
public void setCharacterEncoding(java.lang.String env)
    throws UnsupportedEncodingException
```

Overrides the name of the character encoding used in the body of this request. This method must be called prior to reading request parameters or reading input using getReader().

Parameters:
`a` - `String` containing the name of the chararacter encoding.

Throws:
`java.io.UnsupportedEncodingException` - if this is not a valid encoding

SRV.14.2.17 ServletRequestWrapper

```
public class ServletRequestWrapper implements
javax.servlet.ServletRequest
```

All Implemented Interfaces: `ServletRequest`

Direct Known Subclasses: `javax.servlet.http.HttpServletRequestWrapper`

Provides a convenient implementation of the ServletRequest interface that can be subclassed by developers wishing to adapt the request to a Servlet. This class implements the Wrapper or Decorator pattern. Methods default to calling through to the wrapped request object.

Since: v 2.3

See Also: `ServletRequest`

SRV.14.2.17.1 Constructors

ServletRequestWrapper(ServletRequest)

```
public ServletRequestWrapper(ServletRequest request)
```

Creates a ServletRequest adaptor wrapping the given request object.

Throws:
`java.lang.IllegalArgumentException` - if the request is null

SRV.14.2.17.2 Methods

getAttribute(String)

```
public java.lang.Object getAttribute(java.lang.String name)
```

The default behavior of this method is to call getAttribute(String name) on the wrapped request object.

Specified By: `ServletRequest.getAttribute(String)` in interface `ServletRequest`

getAttributeNames()

```
public java.util.Enumeration getAttributeNames()
```

The default behavior of this method is to return getAttributeNames() on the wrapped request object.

Specified By: `ServletRequest.getAttributeNames()` in interface `ServletRequest`

getCharacterEncoding()

```
public java.lang.String getCharacterEncoding()
```

The default behavior of this method is to return getCharacterEncoding() on the wrapped request object.

Specified By: `ServletRequest.getCharacterEncoding()` in interface `ServletRequest`

getContentLength()

```
public int getContentLength()
```

The default behavior of this method is to return getContentLength() on the wrapped request object.

Specified By: `ServletRequest.getContentLength()` in interface `ServletRequest`

getContentType()

```
public java.lang.String getContentType()
```

The default behavior of this method is to return getContentType() on the wrapped request object.

Specified By: `ServletRequest.getContentType()` in interface `ServletRequest`

getInputStream()

```
public ServletInputStream getInputStream()
    throws IOException
```

The default behavior of this method is to return getInputStream() on the wrapped request object.

Specified By: `ServletRequest.getInputStream()` in interface `ServletRequest`

Throws:
`IOException`

getLocale()

```
public java.util.Locale getLocale()
```

The default behavior of this method is to return getLocale() on the wrapped request object.

Specified By: `ServletRequest.getLocale()` in interface `ServletRequest`

getLocales()

```
public java.util.Enumeration getLocales()
```

The default behavior of this method is to return getLocales() on the wrapped request object.

Specified By: `ServletRequest.getLocales()` in interface `ServletRequest`

getParameter(String)

```
public java.lang.String getParameter(java.lang.String name)
```

The default behavior of this method is to return getParameter(String name) on the wrapped request object.

Specified By: `ServletRequest.getParameter(String)` in interface `ServletRequest`

getParameterMap()

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

The default behavior of this method is to return getParameterMap() on the wrapped request object.

Specified By: `ServletRequest.getParameterMap()` in interface `ServletRequest`

getParameterNames()

```
public java.util.Enumeration getParameterNames()
```

The default behavior of this method is to return getParameterNames() on the wrapped request object.

Specified By: `ServletRequest.getParameterNames()` in interface `ServletRequest`

getParameterValues(String)

```
public java.lang.String[] getParameterValues(java.lang.String name)
```

The default behavior of this method is to return getParameterValues(String name) on the wrapped request object.

Specified By: `ServletRequest.getParameterValues(String)` in interface `ServletRequest`

getProtocol()

```
public java.lang.String getProtocol()
```

The default behavior of this method is to return getProtocol() on the wrapped request object.

Specified By: `ServletRequest.getProtocol()` in interface `ServletRequest`

getReader()

```
public java.io.BufferedReader getReader()
    throws IOException
```

The default behavior of this method is to return getReader() on the wrapped request object.

Specified By: `ServletRequest.getReader()` in interface `ServletRequest`

Throws:
`IOException`

getRealPath(String)

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

The default behavior of this method is to return getRealPath(String path) on the wrapped request object.

Specified By: `ServletRequest.getRealPath(String)` in interface `ServletRequest`

getRemoteAddr()

```
public java.lang.String getRemoteAddr()
```

The default behavior of this method is to return getRemoteAddr() on the wrapped request object.

Specified By: `ServletRequest.getRemoteAddr()` in interface `ServletRequest`

getRemoteHost()

```
public java.lang.String getRemoteHost()
```

The default behavior of this method is to return getRemoteHost() on the wrapped request object.

Specified By: `ServletRequest.getRemoteHost()` in interface `ServletRequest`

getRequest()

```
public ServletRequest getRequest()
```

Return the wrapped request object.

getRequestDispatcher(String)

```
public RequestDispatcher getRequestDispatcher(java.lang.String
    path)
```

The default behavior of this method is to return getRequestDispatcher(String path) on the wrapped request object.

Specified By: `ServletRequest.getRequestDispatcher(String)` in interface `ServletRequest`

getScheme()

```
public java.lang.String getScheme()
```

The default behavior of this method is to return getScheme() on the wrapped request object.

Specified By: `ServletRequest.getScheme()` in interface `ServletRequest`

getServerName()

```
public java.lang.String getServerName()
```

The default behavior of this method is to return getServerName() on the wrapped request object.

Specified By: `ServletRequest.getServerName()` in interface `ServletRequest`

getServerPort()

```
public int getServerPort()
```

The default behavior of this method is to return getServerPort() on the wrapped request object.

Specified By: `ServletRequest.getServerPort()` in interface `ServletRequest`

isSecure()

```
public boolean isSecure()
```

The default behavior of this method is to return isSecure() on the wrapped request object.

Specified By: `ServletRequest.isSecure()` in interface `ServletRequest`

removeAttribute(String)

```
public void removeAttribute(java.lang.String name)
```

The default behavior of this method is to call removeAttribute(String name) on the wrapped request object.

Specified By: `ServletRequest.removeAttribute(String)` in interface `ServletRequest`

setAttribute(String, Object)

```
public void setAttribute(java.lang.String name, java.lang.Object o)
```

The default behavior of this method is to return setAttribute(String name, Object o) on the wrapped request object.

Specified By: `ServletRequest.setAttribute(String, Object)` in interface `ServletRequest`

setCharacterEncoding(String)

```
public void setCharacterEncoding(java.lang.String enc)
    throws UnsupportedEncodingException
```

The default behavior of this method is to set the character encoding on the wrapped request object.

Specified By: `ServletRequest.setCharacterEncoding(String)` in interface `ServletRequest`

Throws:
`UnsupportedEncodingException`

setRequest(ServletRequest)

```
public void setRequest(ServletRequest request)
```

Sets the request object being wrapped.

Throws:
`java.lang.IllegalArgumentException` - if the request is null.

SRV.14.2.18 ServletResponse

`public interface` **`ServletResponse`**

All Known Subinterfaces: `javax.servlet.http.HttpServletResponse`

All Known Implementing Classes: `ServletResponseWrapper`

Defines an object to assist a servlet in sending a response to the client. The servlet container creates a `ServletResponse` object and passes it as an argument to the servlet's `service` method.

To send binary data in a MIME body response, use the `ServletOutputStream` returned by `getOutputStream()` . To send character data, use the `PrintWriter` object returned by `getWriter()` . To mix binary and text data, for example, to create a multipart response, use a `ServletOutputStream` and manage the character sections manually.

The charset for the MIME body response can be specified with `setContentType(String)` . For example, "text/html; charset=Shift_JIS". The charset can alternately be set using `setLocale(Locale)` . If no charset is specified, ISO-8859-1 will be used. The `setContentType` or `setLocale` method must be called before `getWriter` for the charset to affect the construction of the writer.

See the Internet RFCs such as RFC 2045 (http://info.internet.isi.edu/in-notes/rfc/files/rfc2045.txt) for more information on MIME. Protocols such as SMTP and HTTP define profiles of MIME, and those standards are still evolving.

See Also: `ServletOutputStream`

SRV.14.2.18.1 Methods

flushBuffer()

```
public void flushBuffer()
    throws IOException
```

Forces any content in the buffer to be written to the client. A call to this method automatically commits the response, meaning the status code and headers will be written.

Throws:
`IOException`

See Also: `setBufferSize(int)`, `getBufferSize()`, `isCommitted()`, `reset()`

getBufferSize()

`public int getBufferSize()`

Returns the actual buffer size used for the response. If no buffering is used, this method returns 0.

Returns: the actual buffer size used

See Also: `setBufferSize(int)`, `flushBuffer()`, `isCommitted()`, `reset()`

getCharacterEncoding()

`public java.lang.String getCharacterEncoding()`

Returns the name of the charset used for the MIME body sent in this response.

If no charset has been assigned, it is implicitly set to `ISO-8859-1` (`Latin-1`).

See RFC 2047 (http://ds.internic.net/rfc/rfc2045.txt) for more information about character encoding and MIME.

Returns: a `String` specifying the name of the charset, for example, `ISO-8859-1`

getLocale()

`public java.util.Locale getLocale()`

Returns the locale assigned to the response.

See Also: `setLocale(Locale)`

getOutputStream()

```
public ServletOutputStream getOutputStream()
    throws IOException
```

Returns a `ServletOutputStream` suitable for writing binary data in the response. The servlet container does not encode the binary data.

Calling flush() on the ServletOutputStream commits the response. Either this method or `getWriter()` may be called to write the body, not both.

Returns: a `ServletOutputStream` for writing binary data

Throws:
`IllegalStateException` - if the `getWriter` method has been called on this response

`IOException` - if an input or output exception occurred

See Also: `getWriter()`

getWriter()

```
public java.io.PrintWriter getWriter()
    throws IOException
```

Returns a `PrintWriter` object that can send character text to the client. The character encoding used is the one specified in the `charset=` property of the `setContentType(String)` method, which must be called *before* calling this method for the charset to take effect.

If necessary, the MIME type of the response is modified to reflect the character encoding used.

Calling flush() on the PrintWriter commits the response.

Either this method or `getOutputStream()` may be called to write the body, not both.

Returns: a `PrintWriter` object that can return character data to the client

Throws:
`UnsupportedEncodingException` - if the charset specified in `setContentType` cannot be used

`IllegalStateException` - if the `getOutputStream` method has already been called for this response object

`IOException` - if an input or output exception occurred

See Also: `getOutputStream()`, `setContentType(String)`

isCommitted()

```
public boolean isCommitted()
```

Returns a boolean indicating if the response has been committed. A commited response has already had its status code and headers written.

Returns: a boolean indicating if the response has been committed

See Also: `setBufferSize(int)`, `getBufferSize()`, `flushBuffer()`, `reset()`

reset()

```
public void reset()
```

Clears any data that exists in the buffer as well as the status code and headers. If the response has been committed, this method throws an `IllegalStateException`.

Throws:
`IllegalStateException` - if the response has already been committed

See Also: `setBufferSize(int)`, `getBufferSize()`, `flushBuffer()`, `isCommitted()`

resetBuffer()

`public void` **`resetBuffer`**`()`

Clears the content of the underlying buffer in the response without clearing headers or status code. If the response has been committed, this method throws an `IllegalStateException`.

Since: 2.3

See Also: `setBufferSize(int)`, `getBufferSize()`, `isCommitted()`, `reset()`

setBufferSize(int)

`public void` **`setBufferSize`**`(int size)`

Sets the preferred buffer size for the body of the response. The servlet container will use a buffer at least as large as the size requested. The actual buffer size used can be found using `getBufferSize`.

A larger buffer allows more content to be written before anything is actually sent, thus providing the servlet with more time to set appropriate status codes and headers. A smaller buffer decreases server memory load and allows the client to start receiving data more quickly.

This method must be called before any response body content is written; if content has been written, this method throws an `IllegalStateException`.

Parameters:
`size` - the preferred buffer size

Throws:
`IllegalStateException` - if this method is called after content has been written

See Also: `getBufferSize()`, `flushBuffer()`, `isCommitted()`, `reset()`

setContentLength(int)

`public void` **`setContentLength`**`(int len)`

Sets the length of the content body in the response In HTTP servlets, this method sets the HTTP Content-Length header.

Parameters:
`len` - an integer specifying the length of the content being returned to the client; sets the Content-Length header

setContentType(String)

`public void` **`setContentType`**`(java.lang.String type)`

Sets the content type of the response being sent to the client. The content type may include the type of character encoding used, for example, `text/html; charset=ISO-8859-4`.

If obtaining a `PrintWriter`, this method should be called first.

Parameters:
`type` - a `String` specifying the MIME type of the content

See Also: `getOutputStream()`, `getWriter()`

setLocale(Locale)

`public void setLocale(java.util.Locale loc)`

Sets the locale of the response, setting the headers (including the Content-Type's charset) as appropriate. This method should be called before a call to `getWriter()` . By default, the response locale is the default locale for the server.

Parameters:
`loc` - the locale of the response

See Also: `getLocale()`

SRV.14.2.19 ServletResponseWrapper

`public class ServletResponseWrapper implements javax.servlet.ServletResponse`

All Implemented Interfaces: `ServletResponse`

Direct Known Subclasses: `javax.servlet.http.HttpServletResponseWrapper`

Provides a convenient implementation of the ServletResponse interface that can be subclassed by developers wishing to adapt the response from a Servlet. This class implements the Wrapper or Decorator pattern. Methods default to calling through to the wrapped response object.

Since: v 2.3

See Also: `ServletResponse`

SRV.14.2.19.1 Constructors

ServletResponseWrapper(ServletResponse)

`public ServletResponseWrapper(ServletResponse response)`

Creates a ServletResponse adaptor wrapping the given response object.

Throws:

`java.lang.IllegalArgumentException` - if the response is null.

SRV.14.2.19.2 Methods

flushBuffer()

```
public void flushBuffer()
    throws IOException
```

The default behavior of this method is to call flushBuffer() on the wrapped response object.

Specified By: `ServletResponse.flushBuffer()` in interface `ServletResponse`

Throws:
`IOException`

getBufferSize()

```
public int getBufferSize()
```

The default behavior of this method is to return getBufferSize() on the wrapped response object.

Specified By: `ServletResponse.getBufferSize()` in interface `ServletResponse`

getCharacterEncoding()

```
public java.lang.String getCharacterEncoding()
```

The default behavior of this method is to return getCharacterEncoding() on the wrapped response object.

Specified By: `ServletResponse.getCharacterEncoding()` in interface `ServletResponse`

getLocale()

```
public java.util.Locale getLocale()
```

The default behavior of this method is to return getLocale() on the wrapped response object.

Specified By: `ServletResponse.getLocale()` in interface `ServletResponse`

getOutputStream()

```
public ServletOutputStream getOutputStream()
    throws IOException
```

The default behavior of this method is to return getOutputStream() on the wrapped response object.

Specified By: `ServletResponse.getOutputStream()` in interface `ServletResponse`

Throws:
`IOException`

getResponse()

```
public ServletResponse getResponse()
```

Return the wrapped ServletResponse object.

getWriter()

```
public java.io.PrintWriter getWriter()
    throws IOException
```

The default behavior of this method is to return getWriter() on the wrapped response object.

Specified By: `ServletResponse.getWriter()` in interface `ServletResponse`

Throws:
`IOException`

isCommitted()

```
public boolean isCommitted()
```

The default behavior of this method is to return isCommitted() on the wrapped response object.

Specified By: `ServletResponse.isCommitted()` in interface `ServletResponse`

reset()

```
public void reset()
```

The default behavior of this method is to call reset() on the wrapped response object.

Specified By: `ServletResponse.reset()` in interface `ServletResponse`

resetBuffer()

```
public void resetBuffer()
```

The default behavior of this method is to call resetBuffer() on the wrapped response object.

Specified By: `ServletResponse.resetBuffer()` in interface `ServletResponse`

setBufferSize(int)

```
public void setBufferSize(int size)
```

The default behavior of this method is to call setBufferSize(int size) on the wrapped response object.

Specified By: `ServletResponse.setBufferSize(int)` in interface `ServletResponse`

setContentLength(int)

```
public void setContentLength(int len)
```

The default behavior of this method is to call setContentLength(int len) on the wrapped response object.

Specified By: `ServletResponse.setContentLength(int)` in interface `ServletResponse`

setContentType(String)

```
public void setContentType(java.lang.String type)
```

The default behavior of this method is to call setContentType(String type) on the wrapped response object.

Specified By: `ServletResponse.setContentType(String)` in interface `ServletResponse`

setLocale(Locale)

```
public void setLocale(java.util.Locale loc)
```

The default behavior of this method is to call setLocale(Locale loc) on the wrapped response object.

Specified By: `ServletResponse.setLocale(Locale)` in interface `ServletResponse`

setResponse(ServletResponse)

```
public void setResponse(ServletResponse response)
```

Sets the response being wrapped.

Throws:
`java.lang.IllegalArgumentException` - if the response is null.

SRV.14.2.20 SingleThreadModel

`public interface` **`SingleThreadModel`**

Ensures that servlets handle only one request at a time. This interface has no methods.

If a servlet implements this interface, you are *guaranteed* that no two threads will execute concurrently in the servlet's `service` method. The servlet container can make this guarantee by synchronizing access to a single instance of the servlet, or by maintaining a pool of servlet instances and dispatching each new request to a free servlet.

This interface does not prevent synchronization problems that result from servlets accessing shared resources such as static class variables or classes outside the scope of the servlet.

SRV.14.2.21 UnavailableException

`public class` **`UnavailableException`** `extends`
`javax.servlet.ServletException`

All Implemented Interfaces: `java.io.Serializable`

Defines an exception that a servlet or filter throws to indicate that it is permanently or temporarily unavailable.

When a servlet or filter is permanently unavailable, something is wrong with the it, and it cannot handle requests until some action is taken. For example, a servlet might be configured incorrectly, or a filter's state may be corrupted. The component should log both the error and the corrective action that is needed.

A servlet or filter is temporarily unavailable if it cannot handle requests momentarily due to some system-wide problem. For example, a third-tier server might not be accessible, or there may be insufficient memory or disk storage to handle requests. A system administrator may need to take corrective action.

Servlet containers can safely treat both types of unavailable exceptions in the same way. However, treating temporary unavailability effectively makes the servlet container more robust. Specifically, the servlet container might block requests to the servlet or filter for a period of time suggested by the exception, rather than rejecting them until the servlet container restarts.

SRV.14.2.21.1 Constructors

UnavailableException(int, Servlet, String)

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

Deprecated. As of Java Servlet API 2.2, use `UnavailableException(String, int)` instead.

Parameters:
`seconds` - an integer specifying the number of seconds the servlet expects to be unavailable; if zero or negative, indicates that the servlet can't make an estimate

`servlet` - the `Servlet` that is unavailable

`msg` - a `String` specifying the descriptive message, which can be written to a log file or displayed for the user.

UnavailableException(Servlet, String)

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

Deprecated. As of Java Servlet API 2.2, use `UnavailableException(String)` instead.

Parameters:
`servlet` - the `Servlet` instance that is unavailable

`msg` - a `String` specifying the descriptive message

UnavailableException(String)

```
public UnavailableException(java.lang.String msg)
```

Constructs a new exception with a descriptive message indicating that the servlet is permanently unavailable.

Parameters:
`msg` - a `String` specifying the descriptive message

UnavailableException(String, int)

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

Constructs a new exception with a descriptive message indicating that the servlet is temporarily unavailable and giving an estimate of how long it will be unavailable.

In some cases, the servlet cannot make an estimate. For example, the servlet might know that a server it needs is not running, but not be able to report how long it will take to be restored to functionality. This can be indicated with a negative or zero value for the `seconds` argument.

Parameters:
`msg` - a `String` specifying the descriptive message, which can be written to a log file or displayed for the user.

`seconds` - an integer specifying the number of seconds the servlet expects to be unavailable; if zero or negative, indicates that the servlet can't make an estimate

SRV.14.2.21.2 Methods

getServlet()

`public Servlet getServlet()`

Deprecated. As of Java Servlet API 2.2, with no replacement. Returns the servlet that is reporting its unavailability.

Returns: the `Servlet` object that is throwing the `UnavailableException`

getUnavailableSeconds()

`public int getUnavailableSeconds()`

Returns the number of seconds the servlet expects to be temporarily unavailable.

If this method returns a negative number, the servlet is permanently unavailable or cannot provide an estimate of how long it will be unavailable. No effort is made to correct for the time elapsed since the exception was first reported.

Returns: an integer specifying the number of seconds the servlet will be temporarily unavailable, or a negative number if the servlet is permanently unavailable or cannot make an estimate

isPermanent()

`public boolean isPermanent()`

Returns a `boolean` indicating whether the servlet is permanently unavailable. If so, something is wrong with the servlet, and the system administrator must take some corrective action.

Returns: `true` if the servlet is permanently unavailable; `false` if the servlet is available or temporarily unavailable

CHAPTER SRV.15

javax.servlet.http

This chapter describes the javax.servlet.http package. The chapter includes content that is generated automatically from the javadoc embedded in the actual Java classes and interfaces. This allows the creation of a single, authoritative, specification document.

SRV.15.1 Servlets Using HTTP Protocol

The *javax.servlet.http package* contains a number of classes and interfaces that describe and define the contracts between a servlet class running under the HTTP protocol and the runtime environment provided for an instance of such a class by a conforming servlet container.

The class *HttpServlet implements the Servlet interface and provides a base* developers will extend *t* o implement servlets for implementing web applications employing the HTTP protocol. In addition to generic Servlet interface methods, the class *HttpServlet* implements interfaces providing HTTP functionality.

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.

Class Summary	
Interfaces	
HttpServletRequest	Extends the javax.servlet.ServletRequest interface to provide request information for HTTP servlets.

Class Summary	
HttpServletResponse	Extends the javax.servlet.ServletResponse interface to provide HTTP-specific functionality in sending a response.
HttpSession	Provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.
HttpSessionActivation-Listener	Objects that are bound to a session may listen to container events notifying them that sessions will be passivated and that session will be activated.
HttpSessionAt-tributeListener	This listener interface can be implemented in order to get notifications of changes to the attribute lists of sessions within this web application.
HttpSessionBindingLis-tener	Causes an object to be notified when it is bound to or unbound from a session.
HttpSessionContext	
HttpSessionListener	Implementations of this interface may are notified of changes to the list of active sessions in a web application.
Classes	
Cookie	Creates a cookie, a small amount of information sent by a servlet to a Web browser, saved by the browser, and later sent back to the server.
HttpServlet	Provides an abstract class to be subclassed to create an HTTP servlet suitable for a Web site.
HttpServletRequestWrap-per	Provides a convenient implementation of the HttpServletRequest interface that can be subclassed by developers wishing to adapt the request to a Servlet.
HttpServletResponse-Wrapper	Provides a convenient implementation of the HttpServletResponse interface that can be subclassed by developers wishing to adapt the response from a Servlet.

Class Summary	
HttpSessionBindingEvent	Events of this type are either sent to an object that implements HttpSessionBindingListener when it is bound or unbound from a session, or to a HttpSessionAttributeListener that has been configured in the deployment descriptor when any attribute is bound, unbound or replaced in a session.
HttpSessionEvent	This is the class representing event notifications for changes to sessions within a web application.
HttpUtils	

SRV.15.1.1 Cookie

`public class **Cookie** implements java.lang.Cloneable`

All Implemented Interfaces: `java.lang.Cloneable`

Creates a cookie, a small amount of information sent by a servlet to a Web browser, saved by the browser, and later sent back to the server. A cookie's value can uniquely identify a client, so cookies are commonly used for session management.

A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number. Some Web browsers have bugs in how they handle the optional attributes, so use them sparingly to improve the interoperability of your servlets.

The servlet sends cookies to the browser by using the `HttpServletResponse.addCookie(Cookie)` method, which adds fields to HTTP response headers to send cookies to the browser, one at a time. The browser is expected to support 20 cookies for each Web server, 300 cookies total, and may limit cookie size to 4 KB each.

The browser returns cookies to the servlet by adding fields to HTTP request headers. Cookies can be retrieved from a request by using the `HttpServletRequest.getCookies()` method. Several cookies might have the same name but different path attributes.

Cookies affect the caching of the Web pages that use them. HTTP 1.0 does not cache pages that use cookies created with this class. This class does not support the cache control defined with HTTP 1.1.

This class supports both the Version 0 (by Netscape) and Version 1 (by RFC 2109) cookie specifications. By default, cookies are created using Version 0 to ensure the best interoperability.

SRV.15.1.1.1 Constructors

Cookie(String, String)

`public **Cookie**(java.lang.String name, java.lang.String value)`

Constructs a cookie with a specified name and value.

The name must conform to RFC 2109. That means it can contain only ASCII alphanumeric characters and cannot contain commas, semicolons, or white space or begin with a $ character. The cookie's name cannot be changed after creation.

The value can be anything the server chooses to send. Its value is probably of interest only to the server. The cookie's value can be changed after creation with the `setValue` method.

By default, cookies are created according to the Netscape cookie specification. The version can be changed with the `setVersion` method.

Parameters:
`name` - a `String` specifying the name of the cookie

`value` - a `String` specifying the value of the cookie

Throws:
`IllegalArgumentException` - if the cookie name contains illegal characters (for example, a comma, space, or semicolon) or it is one of the tokens reserved for use by the cookie protocol

See Also: `setValue(String)`, `setVersion(int)`

SRV.15.1.1.2 Methods

clone()

`public java.lang.Object **clone**()`

Overrides the standard `java.lang.Object.clone` method to return a copy of this cookie.

Overrides: java.lang.Object.clone() in class java.lang.Object

getComment()

`public java.lang.String **getComment**()`

Returns the comment describing the purpose of this cookie, or `null` if the cookie has no comment.

Returns: a `String` containing the comment, or `null` if none

See Also: `setComment(String)`

getDomain()

`public java.lang.String getDomain()`

Returns the domain name set for this cookie. The form of the domain name is set by RFC 2109.

Returns: a `String` containing the domain name

See Also: `setDomain(String)`

getMaxAge()

`public int getMaxAge()`

Returns the maximum age of the cookie, specified in seconds, By default, `-1` indicating the cookie will persist until browser shutdown.

Returns: an integer specifying the maximum age of the cookie in seconds; if negative, means the cookie persists until browser shutdown

See Also: `setMaxAge(int)`

getName()

`public java.lang.String getName()`

Returns the name of the cookie. The name cannot be changed after creation.

Returns: a `String` specifying the cookie's name

getPath()

`public java.lang.String getPath()`

Returns the path on the server to which the browser returns this cookie. The cookie is visible to all subpaths on the server.

Returns: a `String` specifying a path that contains a servlet name, for example, */catalog*

See Also: `setPath(String)`

getSecure()

`public boolean getSecure()`

Returns `true` if the browser is sending cookies only over a secure protocol, or `false` if the browser can send cookies using any protocol.

Returns: `true` if the browser uses a secure protocol; otherwise, `true`

See Also: `setSecure(boolean)`

getValue()

`public java.lang.String **getValue**()`

Returns the value of the cookie.

Returns: a `String` containing the cookie's present value

See Also: `setValue(String)`, `Cookie`

getVersion()

`public int **getVersion**()`

Returns the version of the protocol this cookie complies with. Version 1 complies with RFC 2109, and version 0 complies with the original cookie specification drafted by Netscape. Cookies provided by a browser use and identify the browser's cookie version.

Returns: 0 if the cookie complies with the original Netscape specification; 1 if the cookie complies with RFC 2109

See Also: `setVersion(int)`

setComment(String)

`public void **setComment**(java.lang.String purpose)`

Specifies a comment that describes a cookie's purpose. The comment is useful if the browser presents the cookie to the user. Comments are not supported by Netscape Version 0 cookies.

Parameters:
`purpose` - a `String` specifying the comment to display to the user

See Also: `getComment()`

setDomain(String)

`public void **setDomain**(java.lang.String pattern)`

Specifies the domain within which this cookie should be presented.

The form of the domain name is specified by RFC 2109. A domain name begins with a dot (`.foo.com`) and means that the cookie is visible to servers in a specified Domain Name System (DNS) zone (for example, `www.foo.com`, but not `a.b.foo.com`). By default, cookies are only returned to the server that sent them.

Parameters:

`pattern` - a `String` containing the domain name within which this cookie is visible; form is according to RFC 2109

See Also: `getDomain()`

setMaxAge(int)

`public void `**`setMaxAge`**`(int expiry)`

Sets the maximum age of the cookie in seconds.

A positive value indicates that the cookie will expire after that many seconds have passed. Note that the value is the *maximum* age when the cookie will expire, not the cookie's current age.

A negative value means that the cookie is not stored persistently and will be deleted when the Web browser exits. A zero value causes the cookie to be deleted.

Parameters:
`expiry` - an integer specifying the maximum age of the cookie in seconds; if negative, means the cookie is not stored; if zero, deletes the cookie

See Also: `getMaxAge()`

setPath(String)

`public void `**`setPath`**`(java.lang.String uri)`

Specifies a path for the cookie to which the client should return the cookie.

The cookie is visible to all the pages in the directory you specify, and all the pages in that directory's subdirectories. A cookie's path must include the servlet that set the cookie, for example, */catalog*, which makes the cookie visible to all directories on the server under */catalog*.

Consult RFC 2109 (available on the Internet) for more information on setting path names for cookies.

Parameters:
`uri` - a `String` specifying a path

See Also: `getPath()`

setSecure(boolean)

`public void `**`setSecure`**`(boolean flag)`

Indicates to the browser whether the cookie should only be sent using a secure protocol, such as HTTPS or SSL.

The default value is `false`.

Parameters:

`flag` - if `true`, sends the cookie from the browser to the server using only when using a secure protocol; if `false`, sent on any protocol

See Also: `getSecure()`

setValue(String)

`public void **setValue**(java.lang.String newValue)`

Assigns a new value to a cookie after the cookie is created. If you use a binary value, you may want to use BASE64 encoding.

With Version 0 cookies, values should not contain white space, brackets, parentheses, equals signs, commas, double quotes, slashes, question marks, at signs, colons, and semicolons. Empty values may not behave the same way on all browsers.

Parameters:
`newValue` - a `String` specifying the new value

See Also: `getValue()`, `Cookie`

setVersion(int)

`public void **setVersion**(int v)`

Sets the version of the cookie protocol this cookie complies with. Version 0 complies with the original Netscape cookie specification. Version 1 complies with RFC 2109.

Since RFC 2109 is still somewhat new, consider version 1 as experimental; do not use it yet on production sites.

Parameters:
`v` - 0 if the cookie should comply with the original Netscape specification; 1 if the cookie should comply with RFC 2109

See Also: `getVersion()`

SRV.15.1.2 HttpServlet

```
public abstract class HttpServlet extends
javax.servlet.GenericServlet implements java.io.Serializable
```

All Implemented Interfaces: `java.io.Serializable`, `javax.servlet.Servlet`, `javax.servlet.ServletConfig`

Provides an abstract class to be subclassed to create an HTTP servlet suitable for a Web site. A subclass of `HttpServlet` must override at least one method, usually one of these:

- `doGet`, if the servlet supports HTTP GET requests
- `doPost`, for HTTP POST requests

- `doPut`, for HTTP PUT requests
- `doDelete`, for HTTP DELETE requests
- `init` and `destroy`, to manage resources that are held for the life of the servlet
- `getServletInfo`, which the servlet uses to provide information about itself

There's almost no reason to override the `service` method. `service` handles standard HTTP requests by dispatching them to the handler methods for each HTTP request type (the `do`*XXX* methods listed above).

Likewise, there's almost no reason to override the `doOptions` and `doTrace` methods.

Servlets typically run on multithreaded servers, so be aware that a servlet must handle concurrent requests and be careful to synchronize access to shared resources. Shared resources include in-memory data such as instance or class variables and external objects such as files, database connections, and network connections. See the Java Tutorial on Multithreaded Programming (http://java.sun.com/Series/Tutorial/java/threads/multithreaded.html) for more information on handling multiple threads in a Java program.

SRV.15.1.2.1 Constructors

HttpServlet()

```
public HttpServlet()
```

Does nothing, because this is an abstract class.

SRV.15.1.2.2 Methods

doDelete(HttpServletRequest, HttpServletResponse)

```
protected void doDelete(HttpServletRequest req,
    HttpServletResponse resp)
    throws ServletException, IOException
```

Called by the server (via the `service` method) to allow a servlet to handle a DELETE request. The DELETE operation allows a client to remove a document or Web page from the server.

This method does not need to be either safe or idempotent. Operations requested through DELETE can have side effects for which users can be held accountable. When using this method, it may be useful to save a copy of the affected URL in temporary storage.

If the HTTP DELETE request is incorrectly formatted, `doDelete` returns an HTTP "Bad Request" message.

Parameters:
`req` - the `HttpServletRequest` object that contains the request the client made of the servlet

`resp` - the `HttpServletResponse` object that contains the response the servlet returns to the client

Throws:
`IOException` - if an input or output error occurs while the servlet is handling the DELETE request

`javax.servlet.ServletException` - if the request for the DELETE cannot be handled

doGet(HttpServletRequest, HttpServletResponse)

```
protected void doGet(HttpServletRequest req,
    HttpServletResponse resp)
    throws ServletException, IOException
```

Called by the server (via the `service` method) to allow a servlet to handle a GET request.

Overriding this method to support a GET request also automatically supports an HTTP HEAD request. A HEAD request is a GET request that returns no body in the response, only the request header fields.

When overriding this method, read the request data, write the response headers, get the response's writer or output stream object, and finally, write the response data. It's best to include content type and encoding. When using a `PrintWriter` object to return the response, set the content type before accessing the `PrintWriter` object.

The servlet container must write the headers before committing the response, because in HTTP the headers must be sent before the response body.

Where possible, set the Content-Length header (with the `javax.servlet.ServletResponse.setContentLength(int)` method), to allow the servlet container to use a persistent connection to return its response to the client, improving performance. The content length is automatically set if the entire response fits inside the response buffer.

The GET method should be safe, that is, without any side effects for which users are held responsible. For example, most form queries have no side effects. If a client request is intended to change stored data, the request should use some other HTTP method.

The GET method should also be idempotent, meaning that it can be safely repeated. Sometimes making a method safe also makes it idempotent. For example, repeating queries is both safe and idempotent, but buying a product online or modifying data is neither safe nor idempotent.

If the request is incorrectly formatted, `doGet` returns an HTTP “Bad Request” message.

Parameters:
`req` - an `HttpServletRequest` object that contains the request the client has made of the servlet

`resp` - an `HttpServletResponse` object that contains the response the servlet sends to the client

Throws:
`IOException` - if an input or output error is detected when the servlet handles the GET request

`javax.servlet.ServletException` - if the request for the GET could not be handled

See Also: `javax.servlet.ServletResponse.setContentType(String)`

doHead(HttpServletRequest, HttpServletResponse)

```
protected void doHead(HttpServletRequest req,
    HttpServletResponse resp)
    throws ServletException, IOException
```

Receives an HTTP HEAD request from the protected `service` method and handles the request. The client sends a HEAD request when it wants to see only the headers of a response, such as Content-Type or Content-Length. The HTTP HEAD method counts the output bytes in the response to set the Content-Length header accurately.

If you override this method, you can avoid computing the response body and just set the response headers directly to improve performance. Make sure that the `doHead` method you write is both safe and idempotent (that is, protects itself from being called multiple times for one HTTP HEAD request).

If the HTTP HEAD request is incorrectly formatted, `doHead` returns an HTTP “Bad Request” message.

Parameters:
`req` - the request object that is passed to the servlet

`resp` - the response object that the servlet uses to return the headers to the clien

Throws:
`IOException` - if an input or output error occurs

`javax.servlet.ServletException` - if the request for the HEAD could not be handled

doOptions(HttpServletRequest, HttpServletResponse)

```
protected void doOptions(HttpServletRequest req,
    HttpServletResponse resp)
    throws ServletException, IOException
```

Called by the server (via the `service` method) to allow a servlet to handle a OPTIONS request. The OPTIONS request determines which HTTP methods the server supports and returns an appropriate header. For example, if a servlet overrides `doGet`, this method returns the following header:

```
Allow: GET, HEAD, TRACE, OPTIONS
```

There's no need to override this method unless the servlet implements new HTTP methods, beyond those implemented by HTTP 1.1.

Parameters:
`req` - the `HttpServletRequest` object that contains the request the client made of the servlet

`resp` - the `HttpServletResponse` object that contains the response the servlet returns to the client

Throws:
`IOException` - if an input or output error occurs while the servlet is handling the OPTIONS request

`javax.servlet.ServletException` - if the request for the OPTIONS cannot be handled

doPost(HttpServletRequest, HttpServletResponse)

```
protected void doPost(HttpServletRequest req,
    HttpServletResponse resp)
    throws ServletException, IOException
```

Called by the server (via the `service` method) to allow a servlet to handle a POST request. The HTTP POST method allows the client to send data of unlimited length to the Web server a single time and is useful when posting information such as credit card numbers.

When overriding this method, read the request data, write the response headers, get the response's writer or output stream object, and finally, write the response data. It's best to include content type and encoding. When using a `PrintWriter` object to return the response, set the content type before accessing the `PrintWriter` object.

The servlet container must write the headers before committing the response, because in HTTP the headers must be sent before the response body.

Where possible, set the Content-Length header (with the `javax.servlet.ServletResponse.setContentLength(int)` method), to allow the servlet container to use a persistent connection to return its response to the client, improving performance. The content length is automatically set if the entire response fits inside the response buffer.

When using HTTP 1.1 chunked encoding (which means that the response has a Transfer-Encoding header), do not set the Content-Length header.

This method does not need to be either safe or idempotent. Operations requested through POST can have side effects for which the user can be held accountable, for example, updating stored data or buying items online.

If the HTTP POST request is incorrectly formatted, `doPost` returns an HTTP “Bad Request” message.

Parameters:
`req` - an `HttpServletRequest` object that contains the request the client has made of the servlet

`resp` - an `HttpServletResponse` object that contains the response the servlet sends to the client

Throws:
`IOException` - if an input or output error is detected when the servlet handles the request

`javax.servlet.ServletException` - if the request for the POST could not be handled

See Also: `javax.servlet.ServletOutputStream`, `javax.servlet.ServletResponse.setContentType(String)`

doPut(HttpServletRequest, HttpServletResponse)

```
protected void doPut(HttpServletRequest req,
    HttpServletResponse resp)
    throws ServletException, IOException
```

Called by the server (via the `service` method) to allow a servlet to handle a PUT request. The PUT operation allows a client to place a file on the server and is similar to sending a file by FTP.

When overriding this method, leave intact any content headers sent with the request (including Content-Length, Content-Type, Content-Transfer-Encoding, Content-Encoding, Content-Base, Content-Language, Content-Location, Content-MD5, and Content-Range). If your method cannot handle a content header, it must issue an error message (HTTP 501 - Not Implemented) and discard the request. For more information on HTTP 1.1, see RFC 2068 (http://info.internet.isi.edu:80/in-notes/rfc/files/rfc2068.txt).

This method does not need to be either safe or idempotent. Operations that `doPut` performs can have side effects for which the user can be held accountable. When using this method, it may be useful to save a copy of the affected URL in temporary storage.

If the HTTP PUT request is incorrectly formatted, `doPut` returns an HTTP “Bad Request” message.

Parameters:
`req` - the `HttpServletRequest` object that contains the request the client made of the servlet

`resp` - the `HttpServletResponse` object that contains the response the servlet returns to the client

Throws:
`IOException` - if an input or output error occurs while the servlet is handling the PUT request

`javax.servlet.ServletException` - if the request for the PUT cannot be handled

doTrace(HttpServletRequest, HttpServletResponse)

```
protected void doTrace(HttpServletRequest req,
    HttpServletResponse resp)
    throws ServletException, IOException
```

Called by the server (via the `service` method) to allow a servlet to handle a TRACE request. A TRACE returns the headers sent with the TRACE request to the client, so that they can be used in debugging. There’s no need to override this method.

Parameters:
`req` - the `HttpServletRequest` object that contains the request the client made of the servlet

`resp` - the `HttpServletResponse` object that contains the response the servlet returns to the client

Throws:
`IOException` - if an input or output error occurs while the servlet is handling the TRACE request

`javax.servlet.ServletException` - if the request for the TRACE cannot be handled

getLastModified(HttpServletRequest)

```
protected long getLastModified(HttpServletRequest req)
```

Returns the time the `HttpServletRequest` object was last modified, in milliseconds since midnight January 1, 1970 GMT. If the time is unknown, this method returns a negative number (the default).

Servlets that support HTTP GET requests and can quickly determine their last modification time should override this method. This makes browser and proxy caches work more effectively, reducing the load on server and network resources.

Parameters:
`req` - the `HttpServletRequest` object that is sent to the servlet

Returns: a `long` integer specifying the time the `HttpServletRequest` object was last modified, in milliseconds since midnight, January 1, 1970 GMT, or -1 if the time is not known

service(HttpServletRequest, HttpServletResponse)

```
protected void service(HttpServletRequest req,
    HttpServletResponse resp)
    throws ServletException, IOException
```

Receives standard HTTP requests from the public `service` method and dispatches them to the `do`*XXX* methods defined in this class. This method is an HTTP-specific version of the `javax.servlet.Servlet.service(ServletRequest, ServletResponse)` method. There's no need to override this method.

Parameters:
`req` - the `HttpServletRequest` object that contains the request the client made of the servlet

`resp` - the `HttpServletResponse` object that contains the response the servlet returns to the client

Throws:
`IOException` - if an input or output error occurs while the servlet is handling the TRACE request

`javax.servlet.ServletException` - if the request for the TRACE cannot be handled

See Also: `javax.servlet.Servlet.service(ServletRequest, ServletResponse)`

service(ServletRequest, ServletResponse)

```
public void service(javax.servlet.ServletRequest req,
    javax.servlet.ServletResponse res)
    throws ServletException, IOException
```

Dispatches client requests to the protected `service` method. There's no need to override this method.

Specified By: `javax.servlet.Servlet.service(ServletRequest, ServletResponse)` in interface `javax.servlet.Servlet`

Overrides: `javax.servlet.GenericServlet.service(ServletRequest, ServletResponse)` in class `javax.servlet.GenericServlet`

Parameters:
`req` - the `HttpServletRequest` object that contains the request the client made of the servlet

`resp` - the `HttpServletResponse` object that contains the response the servlet returns to the client

Throws:
`IOException` - if an input or output error occurs while the servlet is handling the TRACE request

`javax.servlet.ServletException` - if the request for the TRACE cannot be handled

See Also: `javax.servlet.Servlet.service(ServletRequest, ServletResponse)`

SRV.15.1.3 HttpServletRequest

`public interface` **`HttpServletRequest extends`**
`javax.servlet.ServletRequest`

All Superinterfaces: `javax.servlet.ServletRequest`

All Known Implementing Classes: `HttpServletRequestWrapper`

Extends the `javax.servlet.ServletRequest` interface to provide request information for HTTP servlets.

The servlet container creates an `HttpServletRequest` object and passes it as an argument to the servlet's service methods (`doGet`, `doPost`, etc).

SRV.15.1.3.1 Fields

BASIC_AUTH

`public static final java.lang.String` **`BASIC_AUTH`**

String identifier for Basic authentication. Value "BASIC"

CLIENT_CERT_AUTH

`public static final java.lang.String` **`CLIENT_CERT_AUTH`**

String identifier for Basic authentication. Value “CLIENT_CERT”

DIGEST_AUTH

`public static final java.lang.String` **`DIGEST_AUTH`**

String identifier for Basic authentication. Value “DIGEST”

FORM_AUTH

`public static final java.lang.String` **`FORM_AUTH`**

String identifier for Basic authentication. Value “FORM”

SRV.15.1.3.2 Methods

getAuthType()

`public java.lang.String` **`getAuthType`**`()`

Returns the name of the authentication scheme used to protect the servlet. All servlet containers support basic, form and client certificate authentication, and may additionally support digest authentication. If the servlet is not authenticated `null` is returned.

Same as the value of the CGI variable AUTH_TYPE.

Returns: one of the static members BASIC_AUTH, FORM_AUTH, CLIENT_CERT_AUTH, DIGEST_AUTH (suitable for == comparison) indicating the authentication scheme, or `null` if the request was not authenticated.

getContextPath()

`public java.lang.String` **`getContextPath`**`()`

Returns the portion of the request URI that indicates the context of the request. The context path always comes first in a request URI. The path starts with a “/” character but does not end with a “/” character. For servlets in the default (root) context, this method returns “”. The container does not decode this string.

Returns: a `String` specifying the portion of the request URI that indicates the context of the request

getCookies()

`public Cookie[]` **`getCookies`**`()`

Returns an array containing all of the `Cookie` objects the client sent with this request. This method returns `null` if no cookies were sent.

Returns: an array of all the `Cookies` included with this request, or `null` if the request has no cookies

getDateHeader(String)

`public long getDateHeader(java.lang.String name)`

Returns the value of the specified request header as a `long` value that represents a `Date` object. Use this method with headers that contain dates, such as `If-Modified-Since`.

The date is returned as the number of milliseconds since January 1, 1970 GMT. The header name is case insensitive.

If the request did not have a header of the specified name, this method returns -1. If the header can't be converted to a date, the method throws an `IllegalArgumentException`.

Parameters:
`name` - a `String` specifying the name of the header

Returns: a `long` value representing the date specified in the header expressed as the number of milliseconds since January 1, 1970 GMT, or -1 if the named header was not included with the reqest

Throws:
`IllegalArgumentException` - If the header value can't be converted to a date

getHeader(String)

`public java.lang.String getHeader(java.lang.String name)`

Returns the value of the specified request header as a `String`. If the request did not include a header of the specified name, this method returns `null`. The header name is case insensitive. You can use this method with any request header.

Parameters:
`name` - a `String` specifying the header name

Returns: a `String` containing the value of the requested header, or `null` if the request does not have a header of that name

getHeaderNames()

`public java.util.Enumeration getHeaderNames()`

Returns an enumeration of all the header names this request contains. If the request has no headers, this method returns an empty enumeration.

Some servlet containers do not allow do not allow servlets to access headers using this method, in which case this method returns `null`

Returns: an enumeration of all the header names sent with this request; if the request has no headers, an empty enumeration; if the servlet container does not allow servlets to use this method, `null`

getHeaders(String)

```
public java.util.Enumeration getHeaders(java.lang.String name)
```

Returns all the values of the specified request header as an `Enumeration` of `String` objects.

Some headers, such as `Accept-Language` can be sent by clients as several headers each with a different value rather than sending the header as a comma separated list.

If the request did not include any headers of the specified name, this method returns an empty `Enumeration`. The header name is case insensitive. You can use this method with any request header.

Parameters:
`name` - a `String` specifying the header name

Returns: an `Enumeration` containing the values of the requested header. If the request does not have any headers of that name return an empty enumeration. If the container does not allow access to header information, return null

getIntHeader(String)

```
public int getIntHeader(java.lang.String name)
```

Returns the value of the specified request header as an `int`. If the request does not have a header of the specified name, this method returns -1. If the header cannot be converted to an integer, this method throws a `Number-FormatException`.

The header name is case insensitive.

Parameters:
`name` - a `String` specifying the name of a request header

Returns: an integer expressing the value of the request header or -1 if the request doesn't have a header of this name

Throws:
`NumberFormatException` - If the header value can't be converted to an `int`

getMethod()

```
public java.lang.String getMethod()
```

Returns the name of the HTTP method with which this request was made, for example, GET, POST, or PUT. Same as the value of the CGI variable REQUEST_METHOD.

Returns: a `String` specifying the name of the method with which this request was made

getPathInfo()

`public java.lang.String` **`getPathInfo`**`()`

Returns any extra path information associated with the URL the client sent when it made this request. The extra path information follows the servlet path but precedes the query string. This method returns `null` if there was no extra path information.

Same as the value of the CGI variable PATH_INFO.

Returns: a `String`, decoded by the web container, specifying extra path information that comes after the servlet path but before the query string in the request URL; or `null` if the URL does not have any extra path information

getPathTranslated()

`public java.lang.String` **`getPathTranslated`**`()`

Returns any extra path information after the servlet name but before the query string, and translates it to a real path. Same as the value of the CGI variable PATH_TRANSLATED.

If the URL does not have any extra path information, this method returns `null`. The web container does not decode thins string.

Returns: a `String` specifying the real path, or `null` if the URL does not have any extra path information

getQueryString()

`public java.lang.String` **`getQueryString`**`()`

Returns the query string that is contained in the request URL after the path. This method returns `null` if the URL does not have a query string. Same as the value of the CGI variable QUERY_STRING.

Returns: a `String` containing the query string or `null` if the URL contains no query string. The value is not decoded by the container.

getRemoteUser()

`public java.lang.String` **`getRemoteUser`**`()`

Returns the login of the user making this request, if the user has been authenticated, or `null` if the user has not been authenticated. Whether the user name is sent with each subsequent request depends on the browser and type of authentication. Same as the value of the CGI variable REMOTE_USER.

Returns: a `String` specifying the login of the user making this request, or `null</code if the user login is not known`

getRequestedSessionId()

`public java.lang.String` **`getRequestedSessionId`**`()`

Returns the session ID specified by the client. This may not be the same as the ID of the actual session in use. For example, if the request specified an old (expired) session ID and the server has started a new session, this method gets a new session with a new ID. If the request did not specify a session ID, this method returns `null`.

Returns: a `String` specifying the session ID, or `null` if the request did not specify a session ID

See Also: `isRequestedSessionIdValid()`

getRequestURI()

`public java.lang.String` **`getRequestURI`**`()`

Returns the part of this request's URL from the protocol name up to the query string in the first line of the HTTP request. The web container does not decode this String. For example:

First line of HTTP request Returned Value	
POST /some/path.html HTTP/1.1	/some/path.html
GET http://foo.bar/a.html HTTP/1.0	/a.html
HEAD /xyz?a=b HTTP/1.1	/xyz

To reconstruct an URL with a scheme and host, use `HttpUtils.getRequestURL(HttpServletRequest)` .

Returns: a `String` containing the part of the URL from the protocol name up to the query string

See Also: `HttpUtils.getRequestURL(HttpServletRequest)`

getRequestURL()

`public java.lang.StringBuffer` **`getRequestURL`**`()`

Reconstructs the URL the client used to make the request. The returned URL contains a protocol, server name, port number, and server path, but it does not include query string parameters.
Because this method returns a `StringBuffer`, not a string, you can modify the URL easily, for example, to append query parameters.
This method is useful for creating redirect messages and for reporting errors.

Returns: a `StringBuffer` object containing the reconstructed URL

getServletPath()

`public java.lang.String **getServletPath**()`

Returns the part of this request's URL that calls the servlet. This includes either the servlet name or a path to the servlet, but does not include any extra path information or a query string. Same as the value of the CGI variable SCRIPT_NAME.

Returns: a `String` containing the name or path of the servlet being called, as specified in the request URL, decoded.

getSession()

`public HttpSession **getSession**()`

Returns the current session associated with this request, or if the request does not have a session, creates one.

Returns: the `HttpSession` associated with this request

See Also: `getSession(boolean)`

getSession(boolean)

`public HttpSession **getSession**(boolean create)`

Returns the current `HttpSession` associated with this request or, if if there is no current session and `create` is true, returns a new session.
If `create` is `false` and the request has no valid `HttpSession`, this method returns `null`.
To make sure the session is properly maintained, you must call this method before the response is committed. If the container is using cookies to maintain session integrity and is asked to create a new session when the response is committed, an IllegalStateException is thrown.

Parameters:
<code>true</code> - to create a new session for this request if necessary; `false` to return `null` if there's no current session

Returns: the `HttpSession` associated with this request or `null` if `create` is `false` and the request has no valid session

See Also: `getSession()`

getUserPrincipal()

`public java.security.Principal` **`getUserPrincipal`**`()`

Returns a `java.security.Principal` object containing the name of the current authenticated user. If the user has not been authenticated, the method returns `null`.

Returns: a `java.security.Principal` containing the name of the user making this request; `null` if the user has not been authenticated

isRequestedSessionIdFromCookie()

`public boolean` **`isRequestedSessionIdFromCookie`**`()`

Checks whether the requested session ID came in as a cookie.

Returns: `true` if the session ID came in as a cookie; otherwise, `false`

See Also: `getSession(boolean)`

isRequestedSessionIdFromUrl()

`public boolean` **`isRequestedSessionIdFromUrl`**`()`

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

isRequestedSessionIdFromURL()

`public boolean` **`isRequestedSessionIdFromURL`**`()`

Checks whether the requested session ID came in as part of the request URL.

Returns: `true` if the session ID came in as part of a URL; otherwise, `false`

See Also: `getSession(boolean)`

isRequestedSessionIdValid()

`public boolean` **`isRequestedSessionIdValid`**`()`

Checks whether the requested session ID is still valid.

Returns: `true` if this request has an id for a valid session in the current session context; `false` otherwise

See Also: `getRequestedSessionId()`, `getSession(boolean)`, `HttpSessionContext`

isUserInRole(String)

`public boolean` **`isUserInRole`**`(java.lang.String role)`

Returns a boolean indicating whether the authenticated user is included in the specified logical “role”. Roles and role membership can be defined using deployment descriptors. If the user has not been authenticated, the method returns `false`.

Parameters:
`role` - a `String` specifying the name of the role

Returns: a `boolean` indicating whether the user making this request belongs to a given role; `false` if the user has not been authenticated

SRV.15.1.4 HttpServletRequestWrapper

`public class` **`HttpServletRequestWrapper`** `extends` `javax.servlet.ServletRequestWrapper` `implements` `javax.servlet.http.HttpServletRequest`

All Implemented Interfaces: `HttpServletRequest`, `javax.servlet.ServletRequest`

Provides a convenient implementation of the HttpServletRequest interface that can be subclassed by developers wishing to adapt the request to a Servlet. This class implements the Wrapper or Decorator pattern. Methods default to calling through to the wrapped request object.

Since: v 2.3

See Also: `HttpServletRequest`

SRV.15.1.4.1 Constructors

HttpServletRequestWrapper(HttpServletRequest)

`public` **`HttpServletRequestWrapper`**`(HttpServletRequest request)`

Constructs a request object wrapping the given request.

Throws:
`java.lang.IllegalArgumentException` - if the request is null

SRV.15.1.4.2 Methods

getAuthType()

`public java.lang.String` **`getAuthType`**`()`

The default behavior of this method is to return getAuthType() on the wrapped request object.

Specified By: `HttpServletRequest.getAuthType()` in interface `HttpServletRequest`

getContextPath()

`public java.lang.String getContextPath()`

The default behavior of this method is to return getContextPath() on the wrapped request object.

Specified By: `HttpServletRequest.getContextPath()` in interface `HttpServletRequest`

getCookies()

`public Cookie[] getCookies()`

The default behavior of this method is to return getCookies() on the wrapped request object.

Specified By: `HttpServletRequest.getCookies()` in interface `HttpServletRequest`

getDateHeader(String)

`public long getDateHeader(java.lang.String name)`

The default behavior of this method is to return getDateHeader(String name) on the wrapped request object.

Specified By: `HttpServletRequest.getDateHeader(String)` in interface `HttpServletRequest`

getHeader(String)

`public java.lang.String getHeader(java.lang.String name)`

The default behavior of this method is to return getHeader(String name) on the wrapped request object.

Specified By: `HttpServletRequest.getHeader(String)` in interface `HttpServletRequest`

getHeaderNames()

`public java.util.Enumeration getHeaderNames()`

The default behavior of this method is to return getHeaderNames() on the wrapped request object.

Specified By: `HttpServletRequest.getHeaderNames()` in interface `HttpServletRequest`

getHeaders(String)

`public java.util.Enumeration getHeaders(java.lang.String name)`

The default behavior of this method is to return getHeaders(String name) on the wrapped request object.

Specified By: `HttpServletRequest.getHeaders(String)` in interface `HttpServletRequest`

getIntHeader(String)

```
public int getIntHeader(java.lang.String name)
```

The default behavior of this method is to return getIntHeader(String name) on the wrapped request object.

Specified By: `HttpServletRequest.getIntHeader(String)` in interface `HttpServletRequest`

getMethod()

```
public java.lang.String getMethod()
```

The default behavior of this method is to return getMethod() on the wrapped request object.

Specified By: `HttpServletRequest.getMethod()` in interface `HttpServletRequest`

getPathInfo()

```
public java.lang.String getPathInfo()
```

The default behavior of this method is to return getPathInfo() on the wrapped request object.

Specified By: `HttpServletRequest.getPathInfo()` in interface `HttpServletRequest`

getPathTranslated()

```
public java.lang.String getPathTranslated()
```

The default behavior of this method is to return getPathTranslated() on the wrapped request object.

Specified By: `HttpServletRequest.getPathTranslated()` in interface `HttpServletRequest`

getQueryString()

```
public java.lang.String getQueryString()
```

The default behavior of this method is to return getQueryString() on the wrapped request object.

Specified By: `HttpServletRequest.getQueryString()` in interface `HttpServletRequest`

getRemoteUser()

`public java.lang.String` **`getRemoteUser`**`()`

The default behavior of this method is to return getRemoteUser() on the wrapped request object.

Specified By: `HttpServletRequest.getRemoteUser()` in interface `HttpServletRequest`

getRequestedSessionId()

`public java.lang.String` **`getRequestedSessionId`**`()`

The default behavior of this method is to return getRequestedSessionId() on the wrapped request object.

Specified By: `HttpServletRequest.getRequestedSessionId()` in interface `HttpServletRequest`

getRequestURI()

`public java.lang.String` **`getRequestURI`**`()`

The default behavior of this method is to return getRequestURI() on the wrapped request object.

Specified By: `HttpServletRequest.getRequestURI()` in interface `HttpServletRequest`

getRequestURL()

`public java.lang.StringBuffer` **`getRequestURL`**`()`

The default behavior of this method is to return getRequestURL() on the wrapped request object.

Specified By: `HttpServletRequest.getRequestURL()` in interface `HttpServletRequest`

getServletPath()

`public java.lang.String` **`getServletPath`**`()`

The default behavior of this method is to return getServletPath() on the wrapped request object.

Specified By: `HttpServletRequest.getServletPath()` in interface `HttpServletRequest`

getSession()

`public HttpSession getSession()`

The default behavior of this method is to return getSession() on the wrapped request object.

Specified By: `HttpServletRequest.getSession()` in interface `HttpServletRequest`

getSession(boolean)

`public HttpSession getSession(boolean create)`

The default behavior of this method is to return getSession(boolean create) on the wrapped request object.

Specified By: `HttpServletRequest.getSession(boolean)` in interface `HttpServletRequest`

getUserPrincipal()

`public java.security.Principal getUserPrincipal()`

The default behavior of this method is to return getUserPrincipal() on the wrapped request object.

Specified By: `HttpServletRequest.getUserPrincipal()` in interface `HttpServletRequest`

isRequestedSessionIdFromCookie()

`public boolean isRequestedSessionIdFromCookie()`

The default behavior of this method is to return isRequestedSessionIdFrom-Cookie() on the wrapped request object.

Specified By:
`HttpServletRequest.isRequestedSessionIdFromCookie()` in interface `HttpServletRequest`

isRequestedSessionIdFromUrl()

`public boolean isRequestedSessionIdFromUrl()`

The default behavior of this method is to return isRequestedSessionIdFrom-Url() on the wrapped request object.

Specified By: `HttpServletRequest.isRequestedSessionIdFromUrl()` in interface `HttpServletRequest`

isRequestedSessionIdFromURL()

`public boolean isRequestedSessionIdFromURL()`

The default behavior of this method is to return isRequestedSessionIdFromURL() on the wrapped request object.

Specified By: `HttpServletRequest.isRequestedSessionIdFromURL()` in interface `HttpServletRequest`

isRequestedSessionIdValid()

`public boolean` **`isRequestedSessionIdValid`**`()`

The default behavior of this method is to return isRequestedSessionIdValid() on the wrapped request object.

Specified By: `HttpServletRequest.isRequestedSessionIdValid()` in interface `HttpServletRequest`

isUserInRole(String)

`public boolean` **`isUserInRole`**`(java.lang.String role)`

The default behavior of this method is to return isUserInRole(String role) on the wrapped request object.

Specified By: `HttpServletRequest.isUserInRole(String)` in interface `HttpServletRequest`

SRV.15.1.5 HttpServletResponse

`public interface` **`HttpServletResponse extends`**
`javax.servlet.ServletResponse`

All Superinterfaces: `javax.servlet.ServletResponse`

All Known Implementing Classes: `HttpServletResponseWrapper`

Extends the `javax.servlet.ServletResponse` interface to provide HTTP-specific functionality in sending a response. For example, it has methods to access HTTP headers and cookies.

The servlet container creates an `HttpServletRequest` object and passes it as an argument to the servlet's service methods (`doGet`, `doPost`, etc).

See Also: `javax.servlet.ServletResponse`

SRV.15.1.5.1 Fields

SC_ACCEPTED

`public static final int` **`SC_ACCEPTED`**

Status code (202) indicating that a request was accepted for processing, but was not completed.

SC_BAD_GATEWAY

```
public static final int SC_BAD_GATEWAY
```

Status code (502) indicating that the HTTP server received an invalid response from a server it consulted when acting as a proxy or gateway.

SC_BAD_REQUEST

```
public static final int SC_BAD_REQUEST
```

Status code (400) indicating the request sent by the client was syntactically incorrect.

SC_CONFLICT

```
public static final int SC_CONFLICT
```

Status code (409) indicating that the request could not be completed due to a conflict with the current state of the resource.

SC_CONTINUE

```
public static final int SC_CONTINUE
```

Status code (100) indicating the client can continue.

SC_CREATED

```
public static final int SC_CREATED
```

Status code (201) indicating the request succeeded and created a new resource on the server.

SC_EXPECTATION_FAILED

```
public static final int SC_EXPECTATION_FAILED
```

Status code (417) indicating that the server could not meet the expectation given in the Expect request header.

SC_FORBIDDEN

```
public static final int SC_FORBIDDEN
```

Status code (403) indicating the server understood the request but refused to fulfill it.

SC_GATEWAY_TIMEOUT

```
public static final int SC_GATEWAY_TIMEOUT
```

Status code (504) indicating that the server did not receive a timely response from the upstream server while acting as a gateway or proxy.

SC_GONE

```
public static final int SC_GONE
```

Status code (410) indicating that the resource is no longer available at the server and no forwarding address is known. This condition *SHOULD* be considered permanent.

SC_HTTP_VERSION_NOT_SUPPORTED

```
public static final int SC_HTTP_VERSION_NOT_SUPPORTED
```

Status code (505) indicating that the server does not support or refuses to support the HTTP protocol version that was used in the request message.

SC_INTERNAL_SERVER_ERROR

```
public static final int SC_INTERNAL_SERVER_ERROR
```

Status code (500) indicating an error inside the HTTP server which prevented it from fulfilling the request.

SC_LENGTH_REQUIRED

```
public static final int SC_LENGTH_REQUIRED
```

Status code (411) indicating that the request cannot be handled without a defined `Content-Length`.

SC_METHOD_NOT_ALLOWED

```
public static final int SC_METHOD_NOT_ALLOWED
```

Status code (405) indicating that the method specified in the `Request-Line` is not allowed for the resource identified by the `Request-URI`.

SC_MOVED_PERMANENTLY

```
public static final int SC_MOVED_PERMANENTLY
```

Status code (301) indicating that the resource has permanently moved to a new location, and that future references should use a new URI with their requests.

SC_MOVED_TEMPORARILY

```
public static final int SC_MOVED_TEMPORARILY
```

Status code (302) indicating that the resource has temporarily moved to another location, but that future references should still use the original URI to access the resource.

SC_MULTIPLE_CHOICES

`public static final int` **`SC_MULTIPLE_CHOICES`**

Status code (300) indicating that the requested resource corresponds to any one of a set of representations, each with its own specific location.

SC_NO_CONTENT

`public static final int` **`SC_NO_CONTENT`**

Status code (204) indicating that the request succeeded but that there was no new information to return.

SC_NON_AUTHORITATIVE_INFORMATION

`public static final int` **`SC_NON_AUTHORITATIVE_INFORMATION`**

Status code (203) indicating that the meta information presented by the client did not originate from the server.

SC_NOT_ACCEPTABLE

`public static final int` **`SC_NOT_ACCEPTABLE`**

Status code (406) indicating that the resource identified by the request is only capable of generating response entities which have content characteristics not acceptable according to the accept headerssent in the request.

SC_NOT_FOUND

`public static final int` **`SC_NOT_FOUND`**

Status code (404) indicating that the requested resource is not available.

SC_NOT_IMPLEMENTED

`public static final int` **`SC_NOT_IMPLEMENTED`**

Status code (501) indicating the HTTP server does not support the functionality needed to fulfill the request.

SC_NOT_MODIFIED

`public static final int` **`SC_NOT_MODIFIED`**

Status code (304) indicating that a conditional GET operation found that the resource was available and not modified.

SC_OK

`public static final int` **`SC_OK`**

Status code (200) indicating the request succeeded normally.

SC_PARTIAL_CONTENT

`public static final int` **`SC_PARTIAL_CONTENT`**

Status code (206) indicating that the server has fulfilled the partial GET request for the resource.

SC_PAYMENT_REQUIRED

`public static final int` **`SC_PAYMENT_REQUIRED`**

Status code (402) reserved for future use.

SC_PRECONDITION_FAILED

`public static final int` **`SC_PRECONDITION_FAILED`**

Status code (412) indicating that the precondition given in one or more of the request-header fields evaluated to false when it was tested on the server.

SC_PROXY_AUTHENTICATION_REQUIRED

`public static final int` **`SC_PROXY_AUTHENTICATION_REQUIRED`**

Status code (407) indicating that the client *MUST* first authenticate itself with the proxy.

SC_REQUEST_ENTITY_TOO_LARGE

`public static final int` **`SC_REQUEST_ENTITY_TOO_LARGE`**

Status code (413) indicating that the server is refusing to process the request because the request entity is larger than the server is willing or able to process.

SC_REQUEST_TIMEOUT

`public static final int` **`SC_REQUEST_TIMEOUT`**

Status code (408) indicating that the client did not produce a requestwithin the time that the server was prepared to wait.

SC_REQUEST_URI_TOO_LONG

`public static final int` **`SC_REQUEST_URI_TOO_LONG`**

Status code (414) indicating that the server is refusing to service the request because the `Request-URI` is longer than the server is willing to interpret.

SC_REQUESTED_RANGE_NOT_SATISFIABLE

`public static final int` **`SC_REQUESTED_RANGE_NOT_SATISFIABLE`**

Status code (416) indicating that the server cannot serve the requested byte range.

SC_RESET_CONTENT

```
public static final int SC_RESET_CONTENT
```

Status code (205) indicating that the agent *SHOULD* reset the document view which caused the request to be sent.

SC_SEE_OTHER

```
public static final int SC_SEE_OTHER
```

Status code (303) indicating that the response to the request can be found under a different URI.

SC_SERVICE_UNAVAILABLE

```
public static final int SC_SERVICE_UNAVAILABLE
```

Status code (503) indicating that the HTTP server is temporarily overloaded, and unable to handle the request.

SC_SWITCHING_PROTOCOLS

```
public static final int SC_SWITCHING_PROTOCOLS
```

Status code (101) indicating the server is switching protocols according to Upgrade header.

SC_TEMPORARY_REDIRECT

```
public static final int SC_TEMPORARY_REDIRECT
```

Status code (307) indicating that the requested resource resides temporarily under a different URI. The temporary URI *SHOULD* be given by the `Location` field in the response.

SC_UNAUTHORIZED

```
public static final int SC_UNAUTHORIZED
```

Status code (401) indicating that the request requires HTTP authentication.

SC_UNSUPPORTED_MEDIA_TYPE

```
public static final int SC_UNSUPPORTED_MEDIA_TYPE
```

Status code (415) indicating that the server is refusing to service the request because the entity of the request is in a format not supported by the requested resource for the requested method.

SC_USE_PROXY

```
public static final int SC_USE_PROXY
```

Status code (305) indicating that the requested resource *MUST* be accessed through the proxy given by the `Location` field.

SRV.15.1.5.2 Methods

addCookie(Cookie)

`public void **addCookie**(Cookie cookie)`

Adds the specified cookie to the response. This method can be called multiple times to set more than one cookie.

Parameters:
`cookie` - the Cookie to return to the client

addDateHeader(String, long)

`public void **addDateHeader**(java.lang.String name, long date)`

Adds a response header with the given name and date-value. The date is specified in terms of milliseconds since the epoch. This method allows response headers to have multiple values.

Parameters:
`name` - the name of the header to set

`value` - the additional date value

See Also: `setDateHeader(String, long)`

addHeader(String, String)

```
public void addHeader(java.lang.String name,
    java.lang.String value)
```

Adds a response header with the given name and value. This method allows response headers to have multiple values.

Parameters:
`name` - the name of the header

`value` - the additional header value

See Also: `setHeader(String, String)`

addIntHeader(String, int)

`public void **addIntHeader**(java.lang.String name, int value)`

Adds a response header with the given name and integer value. This method allows response headers to have multiple values.

Parameters:
`name` - the name of the header

`value` - the assigned integer value

See Also: `setIntHeader(String, int)`

containsHeader(String)

`public boolean containsHeader(java.lang.String name)`

Returns a boolean indicating whether the named response header has already been set.

Parameters:
`name` - the header name

Returns: `true` if the named response header has already been set; `false` otherwise

encodeRedirectUrl(String)

`public java.lang.String encodeRedirectUrl(java.lang.String url)`

Deprecated. As of version 2.1, use encodeRedirectURL(String url) instead

Parameters:
`url` - the url to be encoded.

Returns: the encoded URL if encoding is needed; the unchanged URL otherwise.

encodeRedirectURL(String)

`public java.lang.String encodeRedirectURL(java.lang.String url)`

Encodes the specified URL for use in the `sendRedirect` method or, if encoding is not needed, returns the URL unchanged. The implementation of this method includes the logic to determine whether the session ID needs to be encoded in the URL. Because the rules for making this determination can differ from those used to decide whether to encode a normal link, this method is seperate from the `encodeURL` method.

All URLs sent to the `HttpServletResponse.sendRedirect` method should be run through this method. Otherwise, URL rewriting cannot be used with browsers which do not support cookies.

Parameters:
`url` - the url to be encoded.

Returns: the encoded URL if encoding is needed; the unchanged URL otherwise.

See Also: `sendRedirect(String)`, `encodeUrl(String)`

encodeUrl(String)

```
public java.lang.String encodeUrl(java.lang.String url)
```

Deprecated. As of version 2.1, use encodeURL(String url) instead

Parameters:
`url` - the url to be encoded.

Returns: the encoded URL if encoding is needed; the unchanged URL otherwise.

encodeURL(String)

```
public java.lang.String encodeURL(java.lang.String url)
```

Encodes the specified URL by including the session ID in it, or, if encoding is not needed, returns the URL unchanged. The implementation of this method includes the logic to determine whether the session ID needs to be encoded in the URL. For example, if the browser supports cookies, or session tracking is turned off, URL encoding is unnecessary.

For robust session tracking, all URLs emitted by a servlet should be run through this method. Otherwise, URL rewriting cannot be used with browsers which do not support cookies.

Parameters:
`url` - the url to be encoded.

Returns: the encoded URL if encoding is needed; the unchanged URL otherwise.

sendError(int)

```
public void sendError(int sc)
    throws IOException
```

Sends an error response to the client using the specified status code and clearing the buffer.

If the response has already been committed, this method throws an IllegalStateException. After using this method, the response should be considered to be committed and should not be written to.

Parameters:
`sc` - the error status code

Throws:
`IOException` - If an input or output exception occurs

`IllegalStateException` - If the response was committed before this method call

sendError(int, String)

```
public void sendError(int sc, java.lang.String msg)
    throws IOException
```

Sends an error response to the client using the specified status clearing the buffer. The server defaults to creating the response to look like an HTML-formatted server error page containing the specified message, setting the content type to "text/html", leaving cookies and other headers unmodified. If an error-page declaration has been made for the web application corresponding to the status code passed in, it will be served back in preference to the suggested msg parameter.

If the response has already been committed, this method throws an IllegalStateException. After using this method, the response should be considered to be committed and should not be written to.

Parameters:
`sc` - the error status code

`msg` - the descriptive message

Throws:
`IOException` - If an input or output exception occurs

`IllegalStateException` - If the response was committed

sendRedirect(String)

```
public void sendRedirect(java.lang.String location)
    throws IOException
```

Sends a temporary redirect response to the client using the specified redirect location URL. This method can accept relative URLs; the servlet container must convert the relative URL to an absolute URL before sending the response to the client. If the location is relative without a leading '/' the container interprets it as relative to the current request URI. If the location is relative with a leading '/' the container interprets it as relative to the servlet container root.

If the response has already been committed, this method throws an IllegalStateException. After using this method, the response should be considered to be committed and should not be written to.

Parameters:
`location` - the redirect location URL

Throws:
`IOException` - If an input or output exception occurs

`IllegalStateException` - If the response was committed

setDateHeader(String, long)

```
public void setDateHeader(java.lang.String name, long date)
```

Sets a response header with the given name and date-value. The date is specified in terms of milliseconds since the epoch. If the header had already been set, the new value overwrites the previous one. The `containsHeader` method can be used to test for the presence of a header before setting its value.

Parameters:
`name` - the name of the header to set

`value` - the assigned date value

See Also: `containsHeader(String)`, `addDateHeader(String, long)`

setHeader(String, String)

```
public void setHeader(java.lang.String name,
    java.lang.String value)
```

Sets a response header with the given name and value. If the header had already been set, the new value overwrites the previous one. The `containsHeader` method can be used to test for the presence of a header before setting its value.

Parameters:
`name` - the name of the header

`value` - the header value

See Also: `containsHeader(String)`, `addHeader(String, String)`

setIntHeader(String, int)

```
public void setIntHeader(java.lang.String name, int value)
```

Sets a response header with the given name and integer value. If the header had already been set, the new value overwrites the previous one. The `containsHeader` method can be used to test for the presence of a header before setting its value.

Parameters:
`name` - the name of the header

`value` - the assigned integer value

See Also: `containsHeader(String)`, `addIntHeader(String, int)`

setStatus(int)

```
public void setStatus(int sc)
```

Sets the status code for this response. This method is used to set the return status code when there is no error (for example, for the status codes SC_OK or SC_MOVED_TEMPORARILY). If there is an error, and the caller wishes to invoke an defined in the web applicaion, the `sendError` method should be used instead.

The container clears the buffer and sets the Location header, preserving cookies and other headers.

Parameters:
`sc` - the status code

See Also: `sendError(int, String)`

setStatus(int, String)

`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.

Parameters:
`sc` - the status code

`sm` - the status message

SRV.15.1.6 HttpServletResponseWrapper

`public class` **`HttpServletResponseWrapper`** `extends`
`javax.servlet.ServletResponseWrapper` `implements`
`javax.servlet.http.HttpServletResponse`

All Implemented Interfaces: `HttpServletResponse`, `javax.servlet.ServletResponse`

Provides a convenient implementation of the HttpServletResponse interface that can be subclassed by developers wishing to adapt the response from a Servlet. This class implements the Wrapper or Decorator pattern. Methods default to calling through to the wrapped response object.

Since: v 2.3

See Also: `HttpServletResponse`

SRV.15.1.6.1 Constructors

HttpServletResponseWrapper(HttpServletResponse)

`public` **`HttpServletResponseWrapper`**`(HttpServletResponse response)`

Constructs a response adaptor wrapping the given response.

Throws:
`java.lang.IllegalArgumentException` - if the response is null

SRV.15.1.6.2 Methods

addCookie(Cookie)

`public void addCookie(Cookie cookie)`

The default behavior of this method is to call addCookie(Cookie cookie) on the wrapped response object.

Specified By: `HttpServletResponse.addCookie(Cookie)` in interface `HttpServletResponse`

addDateHeader(String, long)

`public void addDateHeader(java.lang.String name, long date)`

The default behavior of this method is to call addDateHeader(String name, long date) on the wrapped response object.

Specified By: `HttpServletResponse.addDateHeader(String, long)` in interface `HttpServletResponse`

addHeader(String, String)

```
public void addHeader(java.lang.String name,
    java.lang.String value)
```

The default behavior of this method is to return addHeader(String name, String value) on the wrapped response object.

Specified By: `HttpServletResponse.addHeader(String, String)` in interface `HttpServletResponse`

addIntHeader(String, int)

`public void addIntHeader(java.lang.String name, int value)`

The default behavior of this method is to call addIntHeader(String name, int value) on the wrapped response object.

Specified By: `HttpServletResponse.addIntHeader(String, int)` in interface `HttpServletResponse`

containsHeader(String)

`public boolean containsHeader(java.lang.String name)`

The default behavior of this method is to call containsHeader(String name) on the wrapped response object.

Specified By: `HttpServletResponse.containsHeader(String)` in interface `HttpServletResponse`

encodeRedirectUrl(String)

```
public java.lang.String encodeRedirectUrl(java.lang.String url)
```

The default behavior of this method is to return encodeRedirectUrl(String url) on the wrapped response object.

Specified By: `HttpServletResponse.encodeRedirectUrl(String)` in interface `HttpServletResponse`

encodeRedirectURL(String)

```
public java.lang.String encodeRedirectURL(java.lang.String url)
```

The default behavior of this method is to return encodeRedirectURL(String url) on the wrapped response object.

Specified By: `HttpServletResponse.encodeRedirectURL(String)` in interface `HttpServletResponse`

encodeUrl(String)

```
public java.lang.String encodeUrl(java.lang.String url)
```

The default behavior of this method is to call encodeUrl(String url) on the wrapped response object.

Specified By: `HttpServletResponse.encodeUrl(String)` in interface `HttpServletResponse`

encodeURL(String)

```
public java.lang.String encodeURL(java.lang.String url)
```

The default behavior of this method is to call encodeURL(String url) on the wrapped response object.

Specified By: `HttpServletResponse.encodeURL(String)` in interface `HttpServletResponse`

sendError(int)

```
public void sendError(int sc)
    throws IOException
```

The default behavior of this method is to call sendError(int sc) on the wrapped response object.

Specified By: `HttpServletResponse.sendError(int)` in interface `HttpServletResponse`

Throws:
`IOException`

sendError(int, String)

```
public void sendError(int sc, java.lang.String msg)
    throws IOException
```

The default behavior of this method is to call sendError(int sc, String msg) on the wrapped response object.

Specified By: `HttpServletResponse.sendError(int, String)` in interface `HttpServletResponse`

Throws:
`IOException`

sendRedirect(String)

```
public void sendRedirect(java.lang.String location)
    throws IOException
```

The default behavior of this method is to return sendRedirect(String location) on the wrapped response object.

Specified By: `HttpServletResponse.sendRedirect(String)` in interface `HttpServletResponse`

Throws:
`IOException`

setDateHeader(String, long)

```
public void setDateHeader(java.lang.String name, long date)
```

The default behavior of this method is to call setDateHeader(String name, long date) on the wrapped response object.

Specified By: `HttpServletResponse.setDateHeader(String, long)` in interface `HttpServletResponse`

setHeader(String, String)

```
public void setHeader(java.lang.String name,
    java.lang.String value)
```

The default behavior of this method is to return setHeader(String name, String value) on the wrapped response object.

Specified By: `HttpServletResponse.setHeader(String, String)` in interface `HttpServletResponse`

setIntHeader(String, int)

```
public void setIntHeader(java.lang.String name, int value)
```

The default behavior of this method is to call setIntHeader(String name, int value) on the wrapped response object.

Specified By: `HttpServletResponse.setIntHeader(String, int)` in interface `HttpServletResponse`

setStatus(int)

`public void **setStatus**(int sc)`

The default behavior of this method is to call setStatus(int sc) on the wrapped response object.

Specified By: `HttpServletResponse.setStatus(int)` in interface `HttpServletResponse`

setStatus(int, String)

`public void **setStatus**(int sc, java.lang.String sm)`

The default behavior of this method is to call setStatus(int sc, String sm) on the wrapped response object.

Specified By: `HttpServletResponse.setStatus(int, String)` in interface `HttpServletResponse`

SRV.15.1.7 HttpSession

`public interface **HttpSession**`

Provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.

The servlet container uses this interface to create a session between an HTTP client and an HTTP server. The session persists for a specified time period, across more than one connection or page request from the user. A session usually corresponds to one user, who may visit a site many times. The server can maintain a session in many ways such as using cookies or rewriting URLs.

This interface allows servlets to

- •View and manipulate information about a session, such as the session identifier, creation time, and last accessed time
- •Bind objects to sessions, allowing user information to persist across multiple user connections

When an application stores an object in or removes an object from a session, the session checks whether the object implements `HttpSessionBindingListener`. If it does, the servlet notifies the object that it has been bound to or unbound from the session. Notifications are sent after the binding methods complete. For session that are invalidated or expire, notifications are sent after the session has been invalidatd or expired.

When container migrates a session between VMs in a distributed container setting, all session atributes implementing the `HttpSessionActivationListener` interface are notified.

A servlet should be able to handle cases in which the client does not choose to join a session, such as when cookies are intentionally turned off. Until the client joins the session, `isNew` returns `true`. If the client chooses not to join the session, `getSession` will return a different session on each request, and `isNew` will always return `true`.

Session information is scoped only to the current web application (`ServletContext`), so information stored in one context will not be directly visible in another.

See Also: `HttpSessionBindingListener`, `HttpSessionContext`

SRV.15.1.7.1 Methods

getAttribute(String)

`public java.lang.Object` **`getAttribute`**`(java.lang.String name)`

Returns the object bound with the specified name in this session, or `null` if no object is bound under the name.

Parameters:
`name` - a string specifying the name of the object

Returns: the object with the specified name

Throws:
`IllegalStateException` - if this method is called on an invalidated session

getAttributeNames()

`public java.util.Enumeration` **`getAttributeNames`**`()`

Returns an `Enumeration` of `String` objects containing the names of all the objects bound to this session.

Returns: an `Enumeration` of `String` objects specifying the names of all the objects bound to this session

Throws:
`IllegalStateException` - if this method is called on an invalidated session

getCreationTime()

`public long` **`getCreationTime`**`()`

Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.

Returns: a `long` specifying when this session was created, expressed in milliseconds since 1/1/1970 GMT

Throws:
`IllegalStateException` - if this method is called on an invalidated session

getId()

`public java.lang.String` **`getId`**`()`

Returns a string containing the unique identifier assigned to this session. The identifier is assigned by the servlet container and is implementation dependent.

Returns: a string specifying the identifier assigned to this session

getLastAccessedTime()

`public long` **`getLastAccessedTime`**`()`

Returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT, and marked by the time the container recieved the request.

Actions that your application takes, such as getting or setting a value associated with the session, do not affect the access time.

Returns: a `long` representing the last time the client sent a request associated with this session, expressed in milliseconds since 1/1/1970 GMT

getMaxInactiveInterval()

`public int` **`getMaxInactiveInterval`**`()`

Returns the maximum time interval, in seconds, that the servlet container will keep this session open between client accesses. After this interval, the servlet container will invalidate the session. The maximum time interval can be set with the `setMaxInactiveInterval` method. A negative time indicates the session should never timeout.

Returns: an integer specifying the number of seconds this session remains open between client requests

See Also: `setMaxInactiveInterval(int)`

getServletContext()

`public javax.servlet.ServletContext` **`getServletContext`**`()`

Returns the ServletContext to which this session belongs.

Returns: The ServletContext object for the web application

Since: 2.3

getSessionContext()

`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.

getValue(String)

`public java.lang.Object getValue(java.lang.String name)`

Deprecated. As of Version 2.2, this method is replaced by `getAttribute(String)` .

Parameters:
`name` - a string specifying the name of the object

Returns: the object with the specified name

Throws:
`IllegalStateException` - if this method is called on an invalidated session

getValueNames()

`public java.lang.String[] getValueNames()`

Deprecated. As of Version 2.2, this method is replaced by `getAttributeNames()`

Returns: an array of `String` objects specifying the names of all the objects bound to this session

Throws:
`IllegalStateException` - if this method is called on an invalidated session

invalidate()

`public void invalidate()`

Invalidates this session then unbinds any objects bound to it.

Throws:
`IllegalStateException` - if this method is called on an already invalidated session

isNew()

`public boolean isNew()`

Returns `true` if the client does not yet know about the session or if the client chooses not to join the session. For example, if the server used only cookie-based sessions, and the client had disabled the use of cookies, then a session would be new on each request.

Returns: `true` if the server has created a session, but the client has not yet joined

Throws:
`IllegalStateException` - if this method is called on an already invalidated session

putValue(String, Object)

`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)`

Parameters:
`name` - the name to which the object is bound; cannot be null

`value` - the object to be bound; cannot be null

Throws:
`IllegalStateException` - if this method is called on an invalidated session

removeAttribute(String)

`public void removeAttribute(java.lang.String name)`

Removes the object bound with the specified name from this session. If the session does not have an object bound with the specified name, this method does nothing.

After this method executes, and if the object implements `HttpSession-BindingListener`, the container calls `HttpSessionBinding-Listener.valueUnbound`. The container then notifies any `HttpSessionAttributeListeners` in the web application.

Parameters:
`name` - the name of the object to remove from this session

Throws:
`IllegalStateException` - if this method is called on an invalidated session

removeValue(String)

`public void removeValue(java.lang.String name)`

Deprecated. As of Version 2.2, this method is replaced by `removeAttribute(String)`

Parameters:
`name` - the name of the object to remove from this session

Throws:
`IllegalStateException` - if this method is called on an invalidated session

setAttribute(String, Object)

```
public void setAttribute(java.lang.String name,
    java.lang.Object value)
```

Binds an object to this session, using the name specified. If an object of the same name is already bound to the session, the object is replaced.

After this method executes, and if the new object implements `HttpSession-BindingListener`, the container calls `HttpSessionBinding-Listener.valueBound`. The container then notifies any `HttpSessionAttributeListeners` in the web application.

If an object was already bound to this session of this name that implements `HttpSessionBindingListener`, its `HttpSessionBindingListener.value-Unbound` method is called.

If the value passed in is null, this has the same effect as calling `remove-Attribute().`

Parameters:
`name` - the name to which the object is bound; cannot be null

`value` - the object to be bound

Throws:
`IllegalStateException` - if this method is called on an invalidated session

setMaxInactiveInterval(int)

```
public void setMaxInactiveInterval(int interval)
```

Specifies the time, in seconds, between client requests before the servlet container will invalidate this session. A negative time indicates the session should never timeout.

Parameters:
`interval` - An integer specifying the number of seconds

SRV.15.1.8 HttpSessionActivationListener

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

All Superinterfaces: `java.util.EventListener`

Objects that are bound to a session may listen to container events notifying them that sessions will be passivated 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.

Since: 2.3

SRV.15.1.8.1 Methods

sessionDidActivate(HttpSessionEvent)

`public void sessionDidActivate(HttpSessionEvent se)`

Notification that the session has just been activated.

sessionWillPassivate(HttpSessionEvent)

`public void sessionWillPassivate(HttpSessionEvent se)`

Notification that the session is about to be passivated.

SRV.15.1.9 HttpSessionAttributeListener

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

All Superinterfaces: `java.util.EventListener`

This listener interface can be implemented in order to get notifications of changes to the attribute lists of sessions within this web application.

Since: v 2.3

SRV.15.1.9.1 Methods

attributeAdded(HttpSessionBindingEvent)

`public void attributeAdded(HttpSessionBindingEvent se)`

Notification that an attribute has been added to a session. Called after the attribute is added.

attributeRemoved(HttpSessionBindingEvent)

`public void attributeRemoved(HttpSessionBindingEvent se)`

Notification that an attribute has been removed from a session. Called after the attribute is removed.

attributeReplaced(HttpSessionBindingEvent)

`public void attributeReplaced(HttpSessionBindingEvent se)`

Notification that an attribute has been replaced in a session. Called after the attribute is replaced.

SRV.15.1.10 HttpSessionBindingEvent

```
public class HttpSessionBindingEvent extends
```

javax.servlet.http.HttpSessionEvent

All Implemented Interfaces: java.io.Serializable

Events of this type are either sent to an object that implements HttpSessionBindingListener when it is bound or unbound from a session, or to a HttpSessionAttributeListener that has been configured in the deployment descriptor when any attribute is bound, unbound or replaced in a session.

The session binds the object by a call to HttpSession.setAttribute and unbinds the object by a call to HttpSession.removeAttribute.

See Also: HttpSession, HttpSessionBindingListener, HttpSessionAttributeListener

SRV.15.1.10.1 Constructors

HttpSessionBindingEvent(HttpSession, String)

```
public HttpSessionBindingEvent(HttpSession session,
    java.lang.String name)
```

Constructs an event that notifies an object that it has been bound to or unbound from a session. To receive the event, the object must implement HttpSessionBindingListener.

Parameters:
session - the session to which the object is bound or unbound

name - the name with which the object is bound or unbound

See Also: getName(), getSession()

HttpSessionBindingEvent(HttpSession, String, Object)

```
public HttpSessionBindingEvent(HttpSession session,
    java.lang.String name, java.lang.Object value)
```

Constructs an event that notifies an object that it has been bound to or unbound from a session. To receive the event, the object must implement HttpSessionBindingListener.

Parameters:
session - the session to which the object is bound or unbound

name - the name with which the object is bound or unbound

See Also: getName(), getSession()

SRV.15.1.10.2 Methods

getName()

`public java.lang.String` **`getName`**`()`

Returns the name with which the attribute is bound to or unbound from the session.

Returns: a string specifying the name with which the object is bound to or unbound from the session

getSession()

`public HttpSession` **`getSession`**`()`

Return the session that changed.

Overrides: `HttpSessionEvent.getSession()` in class `HttpSessionEvent`

getValue()

`public java.lang.Object` **`getValue`**`()`

Returns the value of the attribute that has been added, removed or replaced. If the attribute was added (or bound), this is the value of the attribute. If the attrubute was removed (or unbound), this is the value of the removed attribute. If the attribute was replaced, this is the old value of the attribute.

Since: 2.3

SRV.15.1.11 HttpSessionBindingListener

`public interface` **`HttpSessionBindingListener extends java.util.EventListener`**

All Superinterfaces: `java.util.EventListener`

Causes an object to be notified when it is bound to or unbound from a session. The object is notified by an `HttpSessionBindingEvent` object. This may be as a result of a servlet programmer explicitly unbinding an attribute from a session, due to a session being invalidated, or due to a session timing out.

See Also: `HttpSession`, `HttpSessionBindingEvent`

SRV.15.1.11.1 Methods

valueBound(HttpSessionBindingEvent)

`public void` **`valueBound`**`(HttpSessionBindingEvent event)`

Notifies the object that it is being bound to a session and identifies the session.

Parameters:
`event` - the event that identifies the session

See Also: `valueUnbound(HttpSessionBindingEvent)`

valueUnbound(HttpSessionBindingEvent)

`public void valueUnbound(HttpSessionBindingEvent event)`

Notifies the object that it is being unbound from a session and identifies the session.

Parameters:
`event` - the event that identifies the session

See Also: `valueBound(HttpSessionBindingEvent)`

SRV.15.1.12 HttpSessionContext

`public interface HttpSessionContext`

Deprecated. As of Java(tm) Servlet API 2.1 for security reasons, with no replacement. This interface will be removed in a future version of this API.

See Also: `HttpSession`, `HttpSessionBindingEvent`, `HttpSessionBindingListener`

SRV.15.1.12.1 Methods

getIds()

`public java.util.Enumeration getIds()`

Deprecated. As of Java Servlet API 2.1 with no replacement. This method must return an empty `Enumeration` and will be removed in a future version of this API.

getSession(String)

`public HttpSession getSession(java.lang.String sessionId)`

Deprecated. As of Java Servlet API 2.1 with no replacement. This method must return null and will be removed in a future version of this API.

SRV.15.1.13 HttpSessionEvent

`public class HttpSessionEvent extends java.util.EventObject`

All Implemented Interfaces: `java.io.Serializable`

Direct Known Subclasses: `HttpSessionBindingEvent`

This is the class representing event notifications for changes to sessions within a web application.

Since: v 2.3

SRV.15.1.13.1 Constructors

HttpSessionEvent(HttpSession)

`public HttpSessionEvent(HttpSession source)`

Construct a session event from the given source.

SRV.15.1.13.2 Methods

getSession()

`public HttpSession getSession()`

Return the session that changed.

SRV.15.1.14 HttpSessionListener

`public interface HttpSessionListener extends java.util.EventListener`

All Superinterfaces: `java.util.EventListener`

Implementations of this interface may are notified of changes to the list of active sessions in a web application. To recieve notification events, the implementation class must be configured in the deployment descriptor for the web application.

Since: v 2.3

See Also: `HttpSessionEvent`

SRV.15.1.14.1 Methods

sessionCreated(HttpSessionEvent)

`public void sessionCreated(HttpSessionEvent se)`

Notification that a session was created.

Parameters:
`se` - the notification event

sessionDestroyed(HttpSessionEvent)

`public void sessionDestroyed(HttpSessionEvent se)`

Notification that a session was invalidated.

Parameters:
`se` - the notification event

SRV.15.1.15 HttpUtils

```
public class HttpUtils
```

Deprecated. As of Java(tm) Servlet API 2.3. These methods were only useful with the default encoding and have been moved to the request interfaces.

SRV.15.1.15.1 Constructors

HttpUtils()

```
public HttpUtils()
```

Constructs an empty `HttpUtils` object.

SRV.15.1.15.2 Methods

getRequestURL(HttpServletRequest)

```
public static java.lang.StringBuffer
    getRequestURL(HttpServletRequest req)
```

Reconstructs the URL the client used to make the request, using information in the `HttpServletRequest` object. The returned URL contains a protocol, server name, port number, and server path, but it does not include query string parameters.

Because this method returns a `StringBuffer`, not a string, you can modify the URL easily, for example, to append query parameters.

This method is useful for creating redirect messages and for reporting errors.

Parameters:
`req` - a `HttpServletRequest` object containing the client's request

Returns: a `StringBuffer` object containing the reconstructed URL

parsePostData(int, ServletInputStream)

```
public static java.util.Hashtable parsePostData(int len,
    javax.servlet.ServletInputStream in)
```

Parses data from an HTML form that the client sends to the server using the HTTP POST method and the *application/x-www-form-urlencoded* MIME type.

The data sent by the POST method contains key-value pairs. A key can appear more than once in the POST data with different values. However, the key appears only once in the hashtable, with its value being an array of strings containing the multiple values sent by the POST method.

The keys and values in the hashtable are stored in their decoded form, so any + characters are converted to spaces, and characters sent in hexadecimal notation (like *%xx*) are converted to ASCII characters.

Parameters:
`len` - an integer specifying the length, in characters, of the `ServletInputStream` object that is also passed to this method

`in` - the `ServletInputStream` object that contains the data sent from the client

Returns: a `HashTable` object built from the parsed key-value pairs

Throws:
`IllegalArgumentException` - if the data sent by the POST method is invalid

parseQueryString(String)

```
public static java.util.Hashtable parseQueryString(java.lang.String
    s)
```

Parses a query string passed from the client to the server and builds a `HashTable` object with key-value pairs. The query string should be in the form of a string packaged by the GET or POST method, that is, it should have key-value pairs in the form *key=value*, with each pair separated from the next by a & character.

A key can appear more than once in the query string with different values. However, the key appears only once in the hashtable, with its value being an array of strings containing the multiple values sent by the query string.

The keys and values in the hashtable are stored in their decoded form, so any + characters are converted to spaces, and characters sent in hexadecimal notation (like *%xx*) are converted to ASCII characters.

Parameters:
`s` - a string containing the query to be parsed

Returns: a `HashTable` object built from the parsed key-value pairs

Throws:
`IllegalArgumentException` - if the query string is invalid

Changes since version 2.2

This document is the Proposed Final Draft version of the Java Servlet 2.3 Specification developed under the Java Commuity Process^{SM} (JCP).

SRV.S.16 Changes in this document since version 2.2

The Java Servlet 2.2 Specification was the last released version of the servlet specification. The following changes have been made since version 2.2:

- Incorporation of Javadoc^{TM} API definitions into the specification document
- Application Events
- Servlet Filtering
- Requirement of J2SE 1.2 or newer as the underlying platform for web containers
- Dependencies on installed extensions
- Internationalization fixes
- Incorporation of Servlet 2.2 errata and numerous other clarifications

SRV.S.17 Changes since Public Draft

Responding to a large amount of feedback to the public draft, the following changes were made:

SRV.S.17.1 Specification document changes

- Added 2.2 deployment descriptor as appendix
- Added the API documentation as part of the specfication
- Many editorial changes
- Added change list

SRV.S.17.2 Servlets - Chapter 2

- Added `doHead()` method back to `HttpServlet` (see API)

SRV.S.17.3 ServletContexts - Chapter 3

- added `getServletContextName()` (see API)
- added `getResourcePaths()` (see API)

SRV.S.17.4 Request - Chapter 4

- Add attributes for error processing
- Added `UnsupportedCharacterEncoding` to throws clause of `setCharacterEncoding()` (see API)
- `getQueryString()` - specify value is not decoded (see API)
- `getParameterMap()` - return value is immutable (see API)
- clarify `getAuthType()` API documentation, added statics for authentication types (see API)
- clarify default character encoding
- clarify behavior of `getRealPath()` (see API)
- clarification of `HttpServletRequest.getHeaders()` when name not found (see API)

SRV.S.17.5 Response - Chapter 5

- clarify status code on response when errors occur (see API)
- added `resetBuffer()` method to `ServletResponse` (see API)
- `sendError` clarifications (see API))
- disallow container defaulting the content type of a response
- clarify behavior of `flush()` on `PrintWriter` and `ServletOutputStream` (see API)
- clarify default character encoding of response
- clarify what container does with headers on `setStatus()` (see API)
- `sendRedirect()` clarification for non-absolute URLs (API doc)
- `sendError()` clarifications (API doc)

SRV.S.17.6 Filters - Chapter 6

- Scoping of filter instances
- Clarification of filters acting on static resources
- Added `FilterChain` interface and minor refactoring
- Removed `Config` interface
- Added `set{Response,Request}` methods to filter wrapper classes

SRV.S.17.7 Sessions - Chapter 7

- Addition of `HttpSessionActivationListener` interface used in distributed containers (also see API)
- Clarification of semantics for persisting & migrating sessions in distributed containers
- many clarifications of session expiry and notification, order of notification (see API)

SRV.S.17.8 Application Event Listeners - Chapter 10

- Clarifying notifications on shutdown and ordering thereof

SRV.S.17.9 RequestMappings - Chapter 11

- clarified servlet mapped to `/foo/*` is called by a request for `/foo`
- Request matching is done by case-sensitive string match

SRV.S.17.10 Security - Chapter 12

- Specify a default behavior for `isUserInRole()` in absernce of `role-refs`
- Clarify interaction between `RequestDispatcher` and security model
- Clarify policy for processing multiple security constraints
- Added security attributes for SSL algorithm
- Specify status code for failed form login
- Specify allowed methods of return for form login error page

SRV.S.17.11 Deployment Descriptor - Chapter 13

- corrected bad comment for `ejb-ref-type`
- clarifying web container policy for whitespace in the deployment descriptor
- clarifying paths in deployment descriptor are assumed decoded
- recommend validation of deployment descriptor documents and some semantic checking by web containers as aid to developers
- policy for paths refering to resources in the `WAR:` must start with ’`/`’
- clarify policy for relativizing paths in `web.xml`
- added display name to security-constraint for tool manipulation
- fixed security example
- Use of "`*`" to mean ’all roles’ in the security-constraint element
- syntax for specifying sharing scope for connection factory connections
- syntax for declaring dependencies on administered objects in J2EE

- clarify `<error-page>` path usage
- clarify `<jsp-file>` path usage
- snyc with EJB and EE specs on allowed strings in `res-auth` element
- clarify 2.2 dtd must be supported for backwards compatibility

SRV.S.18 Changes since Proposed Final Draft 1

- Minor changes to Filter API
- Renaming listener classes
- added getServletContext() to HttpSession
- added ServletContext.getResourcePaths() directory argument
- expanded section on error pages
- many typos and clarification of text
- many javadoc and DTD clarifications
- many small clarifications of behaviors in the document text

SRV.S.19 Changes since Proposed Final Draft 2

- editorial changes
- added trademarks
- added clarification that containers can recycle container objects SRV.4.10 and SRV.5.6
- clarification of wrapper behavior SRV.6.2.2
- clarification of number of instances of filters SRV.6.2.3
- clarification of filter mappings SRV.6.2.4
- removed requirement of ordering of JARs within a WAR SRV.9.5
- clarified requirements around JNDI/lookups & object invokations on application threads SRV.9.11

- clarified function of session invalidation on form login SRV.12.5.3
- added status code 307 (temporary redirect) to HttpServletResponse

APPENDIX SRV.A Deployment Descriptor Version 2.2

This appendix defines the deployment descriptor for version 2.2. All web containers are required to support web applications using the 2.2 deployment descriptor.

SRV.A.1 Deployment Descriptor DOCTYPE

All valid web application deployment descriptors must contain the following DOCTYPE declaration:

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Appli-
cation 2.2//EN" "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

SRV.A.2 DTD

The DTD that follows defines the XML grammar for a web application deployment descriptor.

```
<!--
The web-app element is the root of the deployment descriptor for a
web application
-->
```

```
<!ELEMENT web-app (icon?, display-name?, description?,
   distributable?, context-param*, servlet*, servlet-mapping*,
   session-config?, mime-mapping*, welcome-file-list?, error-page*,
   taglib*, resource-ref*, security-constraint*, login-config?,
   security-role*, env-entry*, ejb-ref*)>

<!--
The icon element contains a small-icon and a large-icon element
which specify the location within the web application for a small and
large image used to represent the web application in a GUI tool. At a
minimum, tools must accept GIF and JPEG format images.
-->

<!ELEMENT icon (small-icon?, large-icon?)>

<!--
The small-icon element contains the location within the web
application of a file containing a small (16x16 pixel) icon image.
-->

<!ELEMENT small-icon (#PCDATA)>

<!--
The large-icon element contains the location within the web
application of a file containing a large (32x32 pixel) icon image.
-->

<!ELEMENT large-icon (#PCDATA)>

<!--
The display-name element contains a short name that is intended
to be displayed by GUI tools
-->

<!ELEMENT display-name (#PCDATA)>

<!--
The description element is used to provide descriptive text about
the parent element.
-->

<!ELEMENT description (#PCDATA)>

<!--
The distributable element, by its presence in a web application
deployment descriptor, indicates that this web application is
```

```
programmed appropriately to be deployed into a distributed servlet
container
-->

<!ELEMENT distributable EMPTY>

<!--
The context-param element contains the declaration of a web
application's servlet context initialization parameters.
-->

<!ELEMENT context-param (param-name, param-value, description?)>

<!--
The param-name element contains the name of a parameter.
-->

<!ELEMENT param-name (#PCDATA)>

<!--
The param-value element contains the value of a parameter.
-->

<!ELEMENT param-value (#PCDATA)>

<!--
The servlet element contains the declarative data of a
servlet.
If a jsp-file is specified and the load-on-startup element is
present, then the JSP should be precompiled and loaded.
-->

<!ELEMENT servlet (icon?, servlet-name, display-name?, description?,
   (servlet-class|jsp-file), init-param*, load-on-startup?,
   security-role-ref*)>

<!--
The servlet-name element contains the canonical name of the
servlet.
-->

<!ELEMENT servlet-name (#PCDATA)>

<!--
The servlet-class element contains the fully qualified class name
```

```
of the servlet.
-->

<!ELEMENT servlet-class (#PCDATA)>

<!--
The jsp-file element contains the full path to a JSP file within
the web application.
-->

<!ELEMENT jsp-file (#PCDATA)>

<!--
The init-param element contains a name/value pair as an
initialization param of the servlet
-->

<!ELEMENT init-param (param-name, param-value, description?)>

<!--
The load-on-startup element indicates that this servlet should be
loaded on the startup of the web application.
The optional contents of these element must be a positive integer
indicating the order in which the servlet should be loaded.
Lower integers are loaded before higher integers.
If no value is specified, or if the value specified is not a positive
integer, the container is free to load it at any time in the startup
sequence.
-->

<!ELEMENT load-on-startup (#PCDATA)>

<!--
The servlet-mapping element defines a mapping between a servlet and
a url pattern
-->

<!ELEMENT servlet-mapping (servlet-name, url-pattern)>

<!--
The url-pattern element contains the url pattern of the
mapping. Must follow the rules specified in Section 10 of the Servlet
API Specification.
-->

<!ELEMENT url-pattern (#PCDATA)>
```

```
<!--
The session-config element defines the session parameters for this
web application.
-->

<!ELEMENT session-config (session-timeout?)>

<!--
The session-timeout element 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.
-->

<!ELEMENT session-timeout (#PCDATA)>

<!--
The mime-mapping element defines a mapping between an extension and
a mime type.
-->

<!ELEMENT mime-mapping (extension, mime-type)>

<!--
The extension element contains a string describing an
extension. example: "txt"
-->

<!ELEMENT extension (#PCDATA)>

<!--
The mime-type element contains a defined mime type. example: "text/
plain"
-->

<!ELEMENT mime-type (#PCDATA)>

<!--
The welcome-file-list contains an ordered list of welcome files
elements.
-->

<!ELEMENT welcome-file-list (welcome-file+)>
```

```
<!--
The welcome-file element contains file name to use as a default
welcome file, such as index.html
-->

<!ELEMENT welcome-file (#PCDATA)>

<!--
The taglib element is used to describe a JSP tag library.
-->

<!ELEMENT taglib (taglib-uri, taglib-location)>

<!--
The taglib-uri element describes a URI, relative to the location of
the web.xml document, identifying a Tag Library used in the Web
Application.
-->

<!ELEMENT taglib-uri (#PCDATA)>

<!--
the taglib-location element contains the location (as a resource
relative to the root of the web application) where to find the Tag
Libary Description file for the tag library.
-->

<!ELEMENT taglib-location (#PCDATA)>

<!--
The error-page element contains a mapping between an error code or
exception type to the path of a resource in the web application
-->

<!ELEMENT error-page ((error-code | exception-type), location)>

<!--
The error-code contains an HTTP error code, ex: 404
-->

<!ELEMENT error-code (#PCDATA)>

<!--
The exception type contains a fully qualified class name of a Java
exception type.
-->
```

```
<!ELEMENT exception-type (#PCDATA)>

<!--
The location element contains the location of the resource in the
web application
-->

<!ELEMENT location (#PCDATA)>

<!--
The resource-ref element contains a declaration of a Web
Application’s reference to an external resource.
-->

<!ELEMENT resource-ref (description?, res-ref-name, res-type, res-
   auth)>

<!--
The res-ref-name element specifies the name of the resource factory
reference name.
-->

<!ELEMENT res-ref-name (#PCDATA)>

<!--
The res-type element specifies the (Java class) type of the data
source.
-->

<!ELEMENT res-type (#PCDATA)>

<!--
The res-auth element indicates whether the application component
code performs resource signon programmatically or whether the
container signs onto the resource based on the principle mapping
information supplied by the deployer.

Must be CONTAINER or SERVLET
-->

<!ELEMENT res-auth (#PCDATA)>

<!--
The security-constraint element is used to associate security
constraints with one or more web resource collections
-->
```

```
<!ELEMENT security-constraint (web-resource-collection+, auth-
   constraint?, user-data-constraint?)>

<!--
The web-resource-collection element is used to identify a subset of
the resources and HTTP methods on those resources within a web
application to which a security constraint applies.
If no HTTP methods are specified, then the security constraint
applies to all HTTP methods.
-->

<!ELEMENT web-resource-collection (web-resource-name, description?,
   url-pattern*, http-method*)>

<!--
The web-resource-name contains the name of this web resource
collection
-->

<!ELEMENT web-resource-name (#PCDATA)>

<!--
The http-method contains an HTTP method (GET | POST |...)
-->

<!ELEMENT http-method (#PCDATA)>

<!--
The user-data-constraint element is used to indicate how data
communicated between the client and container should be protected
-->

<!ELEMENT user-data-constraint (description?, transport-guarantee)>

<!--
The transport-guarantee element specifies that the communication
between client and server should be NONE, INTEGRAL, or CONFIDENTIAL.
NONE means that the application does not require any transport
guarantees.
A value of INTEGRAL means that the application requires that the data
sent between the client and server be sent in such a way that it
can’t be changed in transit.
CONFIDENTIAL means that the application requires that the data be
transmitted in a fashion that prevents other entities from observing
the contents of the transmission.
```

```
In most cases, the presence of the INTEGRAL or CONFIDENTIAL flag will
indicate that the use of SSL is required.
-->

<!ELEMENT transport-guarantee (#PCDATA)>

<!--
The auth-constraint element indicates the user roles that should be
permitted access to this resource collection.
The role used here must appear in a security-role-ref element.
-->

<!ELEMENT auth-constraint (description?, role-name*)>

<!--
The role-name element contains the name of a security role.
-->

<!ELEMENT role-name (#PCDATA)>

<!--
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.
-->

<!ELEMENT login-config (auth-method?, realm-name?, form-login-
   config?)>

<!--
The realm name element specifies the realm name to use in HTTP Basic
authorization
-->

<!ELEMENT realm-name (#PCDATA)>

<!--
The form-login-config element specifies the login and error pages
that should be used in form based login.
If form based authentication is not used, these elements are ignored.
-->

<!ELEMENT form-login-config (form-login-page, form-error-page)>
```

```
<!--
The form-login-page element defines the location in the web app where
the page that can be used for login can be found
-->

<!ELEMENT form-login-page (#PCDATA)>

<!--
The form-error-page element defines the location in the web app where
the error page that is displayed when login is not successful can be
found
-->

<!ELEMENT form-error-page (#PCDATA)>

<!--
The auth-method element is used to configure the authentication
mechanism for the web application.
As a prerequisite to gaining access to any web resources which are
protected by an authorization constraint, a user must have
mechanism.
Legal values for this element are "BASIC", "DIGEST", "FORM", or
"CLIENT-CERT".
-->

<!ELEMENT auth-method (#PCDATA)>

<!--
The security-role element contains the declaration of a security role
which is used in the security-constraints placed on the web
application.
-->

<!ELEMENT security-role (description?, role-name)>

<!--
The role-name element contains the name of a role. This element must
contain a non-empty string.
-->

<!ELEMENT security-role-ref (description?, role-name, role-link)>

<!--
The role-link element is used to link a security role reference to
a defined security role.
```

```
The role-link element must contain the name of one of the security
roles defined in the security-role elements.
-->

<!ELEMENT role-link (#PCDATA)>

<!--
The env-entry element contains the declaration of an application’s
environment entry.
This element is required to be honored on in J2EE compliant servlet
containers.
-->

<!ELEMENT env-entry (description?, env-entry-name, env-entry-
   value?, env-entry-type)>

<!--
The env-entry-name contains the name of an application’s environment
entry
-->

<!ELEMENT env-entry-name (#PCDATA)>

<!--
The env-entry-value element contains the value of an application’s
environment entry
-->

<!ELEMENT env-entry-value (#PCDATA)>

<!--
The env-entry-type element contains the fully qualified Java type of
the environment entry value that is expected by the application
code.
The following are the legal values of env-entry-type:
java.lang.Boolean, java.lang.String, java.lang.Integer,
java.lang.Double, java.lang.Float.
-->

<!ELEMENT env-entry-type (#PCDATA)>

<!--
The ejb-ref element is used to declare a reference to an enterprise
bean.
-->
```

```
<!ELEMENT ejb-ref (description?, ejb-ref-name, ejb-ref-type, home,
   remote, ejb-link?)>

<!--
The ejb-ref-name element contains the name of an EJB
reference. This is the JNDI name that the servlet code uses to get a
reference to the enterprise bean.
-->

<!ELEMENT ejb-ref-name (#PCDATA)>

<!--
The ejb-ref-type element contains the expected java class type of
the referenced EJB.
-->

<!ELEMENT ejb-ref-type (#PCDATA)>

<!--
The ejb-home element contains the fully qualified name of the EJB’s
home interface
-->

<!ELEMENT home (#PCDATA)>

<!--
The ejb-remote element contains the fully qualified name of the EJB’s
remote interface
-->

<!ELEMENT remote (#PCDATA)>

<!--
The ejb-link element is used in the ejb-ref element to specify that
an EJB reference is linked to an EJB in an encompassing Java2
Enterprise Edition (J2EE) application package.
The value of the ejb-link element must be the ejb-name of and EJB in
the J2EE application package.
-->

<!ELEMENT ejb-link (#PCDATA)>

<!--
The ID mechanism is to allow tools to easily make tool-specific
references to the elements of the deployment descriptor.
```

```
This allows tools that produce additional deployment information
(i.e information beyond the standard deployment descriptor
information) to store the non-standard information in a separate
file, and easily refer from these tools-specific files to the
information in the standard web-app deployment descriptor.
-->

<!ATTLIST web-app id ID #IMPLIED>
<!ATTLIST icon id ID #IMPLIED>
<!ATTLIST small-icon id ID #IMPLIED>
<!ATTLIST large-icon id ID #IMPLIED>
<!ATTLIST display-name id ID #IMPLIED>
<!ATTLIST description id ID #IMPLIED>
<!ATTLIST distributable id ID #IMPLIED>
<!ATTLIST context-param id ID #IMPLIED>
<!ATTLIST param-name id ID #IMPLIED>
<!ATTLIST param-value id ID #IMPLIED>
<!ATTLIST servlet id ID #IMPLIED>
<!ATTLIST servlet-name id ID #IMPLIED>
<!ATTLIST servlet-class id ID #IMPLIED>
<!ATTLIST jsp-file id ID #IMPLIED>
<!ATTLIST init-param id ID #IMPLIED>
<!ATTLIST load-on-startup id ID #IMPLIED>
<!ATTLIST servlet-mapping id ID #IMPLIED>
<!ATTLIST url-pattern id ID #IMPLIED>
<!ATTLIST session-config id ID #IMPLIED>
<!ATTLIST session-timeout id ID #IMPLIED>
<!ATTLIST mime-mapping id ID #IMPLIED>
<!ATTLIST extension id ID #IMPLIED>
<!ATTLIST mime-type id ID #IMPLIED>
<!ATTLIST welcome-file-list id ID #IMPLIED>
<!ATTLIST welcome-file id ID #IMPLIED>
<!ATTLIST taglib id ID #IMPLIED>
<!ATTLIST taglib-uri id ID #IMPLIED>
<!ATTLIST taglib-location id ID #IMPLIED>
<!ATTLIST error-page id ID #IMPLIED>
<!ATTLIST error-code id ID #IMPLIED>
<!ATTLIST exception-type id ID #IMPLIED>
<!ATTLIST location id ID #IMPLIED>
<!ATTLIST resource-ref id ID #IMPLIED>
<!ATTLIST res-ref-name id ID #IMPLIED>
<!ATTLIST res-type id ID #IMPLIED>
<!ATTLIST res-auth id ID #IMPLIED>
<!ATTLIST security-constraint id ID #IMPLIED>
<!ATTLIST web-resource-collection id ID #IMPLIED>
<!ATTLIST web-resource-name id ID #IMPLIED>
<!ATTLIST http-method id ID #IMPLIED>
<!ATTLIST user-data-constraint id ID #IMPLIED>
```

```
<!ATTLIST transport-guarantee id ID #IMPLIED>
<!ATTLIST auth-constraint id ID #IMPLIED>
<!ATTLIST role-name id ID #IMPLIED>
<!ATTLIST login-config id ID #IMPLIED>
<!ATTLIST realm-name id ID #IMPLIED>
<!ATTLIST form-login-config id ID #IMPLIED>
<!ATTLIST form-login-page id ID #IMPLIED>
<!ATTLIST form-error-page id ID #IMPLIED>
<!ATTLIST auth-method id ID #IMPLIED>
<!ATTLIST security-role id ID #IMPLIED>
<!ATTLIST security-role-ref id ID #IMPLIED>
<!ATTLIST role-link id ID #IMPLIED>
<!ATTLIST env-entry id ID #IMPLIED>
<!ATTLIST env-entry-name id ID #IMPLIED>
<!ATTLIST env-entry-value id ID #IMPLIED>
<!ATTLIST env-entry-type id ID #IMPLIED>
<!ATTLIST ejb-ref id ID #IMPLIED>
<!ATTLIST ejb-ref-name id ID #IMPLIED>
<!ATTLIST ejb-ref-type id ID #IMPLIED>
<!ATTLIST home id ID #IMPLIED>
<!ATTLIST remote id ID #IMPLIED>
<!ATTLIST ejb-link id ID #IMPLIED>
```

APPENDIX SRV.B Glossary

Application Developer The producer of a web application. The output of an Application Developer is a set of servlet classes, JSP pages, HTML pages, and supporting libraries and files (such as images, compressed archive files, etc.) for the web application. The Application Developer is typically an application domain expert. The developer is required to be aware of the servlet environment and its consequences when programming, including concurrency considerations, and create the web application accordingly.

Application Assembler Takes the output of the Application Developer and ensures that it is a deployable unit. Thus, the input of the Application Assembler is the servlet classes, JSP pages, HTML pages, and other supporting libraries and files for the web application. The output of the Application Assembler is a web application archive or a web application in an open directory structure.

Deployer The Deployer takes one or more web application archive files or other directory structures provided by an Application Developer and deploys the application into a specific operational environment. The operational environment includes a specific servlet container and web server. The Deployer must resolve all the external dependencies declared by the developer. To perform his role, the deployer uses tools provided by the Servlet Container Provider.

The Deployer is an expert in a specific operational environment. For example, the Deployer is responsible for mapping the security roles defined by the Application Developer to the user groups and accounts that exist in the operational environment where the web application is deployed.

principal A principal is an entity that can be authenticated by an authentication protocol. A principal is identified by a *principal name* and authenticated by using *authentication data*. The content and format of the principal name and the authentication data depend on the authentication protocol.

role (development) The actions and responsibilities taken by various parties during the development, deployment, and running of a web application. In some scenarios, a single party may perform several roles; in others, each role may be performed by a different party.

role (security) An abstract notion used by an Application Developer in an application that can be mapped by the Deployer to a user, or group of users, in a security policy domain.

security policy domain The scope over which security policies are defined and enforced by a security administrator of the security service. A security policy domain is also sometimes referred to as a *realm*.

security technology domain The scope over which the same security mechanism, such as Kerberos, is used to enforce a security policy. Multiple security policy domains can exist within a single technology domain.

Servlet Container Provider A vendor that provides the runtime environment, namely the servlet container and possibly the web server, in which a web application runs as well as the tools necessary to deploy web applications.

The expertise of the Container Provider is in HTTP-level programming. Since this specification does not specify the interface between the web server and the servlet container, it is left to the Container Provider to split the implementation of the required functionality between the container and the server.

servlet definition A unique name associated with a fully qualified class name of a class implementing the `Servlet` interface. A set of initialization parameters can be associated with a servlet definition.

servlet mapping A servlet definition that is associated by a servlet container with a URL path pattern. All requests to that path pattern are handled by the servlet associated with the servlet definition.

System Administrator The person responsible for the configuration and administration of the servlet container and web server. The administrator is also responsible for overseeing the well-being of the deployed web applications at run time.

This specification does not define the contracts for system management and administration. The administrator typically uses runtime monitoring and management tools provided by the Container Provider and server vendors to accomplish these tasks.

uniform resource locator (URL) A compact string representation of resources available via the network. Once the resource represented by a URL has been accessed, various operations may be performed on that resource.[1] A URL is a type of uniform resource identifier (URI). URLs are typically of the form:

```
<protocol>//<servername>/<resource>
```

For the purposes of this specification, we are primarily interested in HTT-based URLs which are of the form:

```
http[s]://<servername>[:port]/<url-path>[?<query-string>]
```

For example:

```
http://java.sun.com/products/servlet/index.html
https://javashop.sun.com/purchase
```

In HTTP-based URLs, the '/' character is reserved to separate a hierarchical path structure in the URL-path portion of the URL. The server is responsible for determining the meaning of the hierarchical structure. There is no correspondence between a URL-path and a given file system path.

web application A collection of servlets, JSP pages , HTML documents, and other web resources which might include image files, compressed archives, and other data. A web application may be packaged into an archive or exist in an open directory structure.

All compatible servlet containers must accept a web application and perform a deployment of its contents into their runtime. This may mean that a container can run the application directly from a web application archive file or it may mean that it will move the contents of a web application into the appropriate locations for that particular container.

1. See RFC 1738

web application archive A single file that contains all of the components of a web application. This archive file is created by using standard JAR tools which allow any or all of the web components to be signed.

Web application archive files are identified by the `.war` extension. A new extension is used instead of `.jar` because that extension is reserved for files which contain a set of class files and that can be placed in the classpath or double clicked using a GUI to launch an application. As the contents of a web application archive are not suitable for such use, a new extension was in order.

web application, distributable A web application that is written so that it can be deployed in a web container distributed across multiple Java virtual machines running on the same host or different hosts. The deployment descriptor for such an application uses the `distributable` element.