Java™ Servlet Specification v2.3

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

Public Review Draft 1 - August 15th 2000 Danny Coward (danny.coward@sun.,.com)

Java(TM) Servlet API Specification ("Specification")
Version: 2.3
Status: Pre-FCS
Release: August 15, 2000

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 and General Terms as set forth in Sun's website Legal Terms. 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.

Subject to the terms and conditions of this license, Sun hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without
the right to sublicense) under Sun's intellectual property rights to review the Specification internally for the purposes of evaluation only. Other than this
limited license, you acquire no right, title or interest in or to the Specification or any other Sun intellectual property. The Specification contains the
proprietary and confidential information of Sun and may only be used in accordance with the license terms set forth herein. This license will expire ninety
(90) days from the date of Release listed above and will terminate immediately without notice from Sun if you fail to comply with any provision of this
license. Upon termination, 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" AND IS EXPERIMENTAL AND MAY CONTAIN DEFECTS OR DEFICIENCIES WHICH
CANNOT OR WILL NOT BE CORRECTED BY SUN. 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 based on your use of the Specification for any purposes other than those
of internal evaluation, and from 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
If this Software 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 evaluation 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 sublicense 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.

Contents

Status

This specification is being developed following the Java Community Process. Comments from Experts, Participants, and the Public will be reviewed and incorporated into the specification.

This document is the First Public Review Draft version of the Java Servlet 2.3 Specification. The main goal of this draft is to define the new areas of functionality worked on for this point release of the specification and to solicit.feedback from the public.

Changes in this document since v2.2

- Incorporation of Javadoc API definitions into the specification document
- Application Events
- Servlet Filtering
- Requirement of JDK 1.2 as the underlying platform for web containers
- Dependencies on installed extensions
- Internationalization fixes
- Incorporation of Servlet 2.2 errata and numerous other clarifications

New sections, changes and additions to the Java servlet specification are marked throughout the document with a change bar.

Preface

This document, the Java™ Servlet Specification, v2.3 the Java Servlet API. The reference implementation provides a behavioral benchmark. In the case of a discrepancy, the order of resolution is this specification and then the reference implementation.

Who should read this document

This document is intended for consumption by:

- Web Server and Application Server vendors that want to provide Servlet Engines that conform with this specification.
- Web Authoring Tool developers that want to generate Web Applications that conform to this specification
- Sophisticated Servlet authors who want to understand the underlying mechanisms of Servlet technology.

Please note that this specification is not a User's Guide and is not intended to be used as such.

API Reference

The Java Servlet API Reference, v2.3 provides the complete description of all the interfaces, classes, exceptions, and methods that compose the Servlet API. This document contains the full specification of class, interfaces, method signatures and accompanying javadoc that defines the Servlet API.

Other Java™ Platform Specifications

The following Java API Specifications are referenced throughout this specification:

- Java2 Platform Enterprise Edition, v1.3 (J2EE)

- JavaServer Pages™, v1.2 (JSP)
- Java Naming and Directory Interface (JNDI)

These specifications may be found at the Java2 Enterprise Edition website:

`http://java.sun.com/j2ee/`

Other Important References

The following Internet Specifications provide relevant information to the development and implementation of the Servlet API and engines which support the Servlet API:

- RFC 1630 Uniform Resource Identifiers (URI)
- RFC 1738 Uniform Resource Locators (URL)
- RFC 2396 niform Resource Identifiers (URI): Generic Syntax

TBD Cross check references to URL, URI for consistency

- 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

You can locate the online versions of any of these RFCs at:

`http://www.rfc-editor.org/`

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

The Extensible Markup Language (XML) is utilized by the Deployment Descriptors described in 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/`

`http://www.xml.org/`

Providing Feedback

The success of the Java Community Process depends on your participation in the community. 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.

Acknowledgements

This public draft represents the team work of the JSR053 expert group.

CHAPTER 1

Overview

This chapter provides an overview of the Servlet API.

1.1 What is a Servlet?

A servlet is a web component, managed by a container, that generates dynamic content. Servlets are small, platform independent Java classes compiled to an architecture neutral bytecode that can be loaded dynamically into and run by a web server. Servlets interact with web clients via a request response paradigm implemented by the servlet container. This request-response model is based on the behavior of the Hypertext Transfer Protocol (HTTP).

1.2 What is a Servlet Container?

The servlet container, in conjunction with a web server or application server, provides the network services over which requests and responses are set, decodes MIME based requests, and formats MIME based responses. A servlet container also contains and manages servlets through their lifecycle.

A servlet container can either 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 may also support additional request / response based protocols such as HTTPS (HTTP over SSL). 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 that a servlet executes in. In a Java 2 Platform Standard Edition 1.2 (J2SE) or Java 2 Platform Enterprise Edition 1.3 (J2EE) environment, these restrictions should be placed using the permission architecture defined by Java 2 Platform. For example, high end application servers may limit certain action, such as the creation of a `Thread` object, to insure that other components of the container are not negatively impacted.

1.3 An Example

A client program, such as a web browser, accesses a web server and makes an HTTP request. This request is processed by the web server and is handed off to the servlet container. The servlet container determines which servlet to invoke based on its internal configuration and calls it with objects representing the request and response. The servlet container can run 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.

The servlet uses the request object to find out who the remote user is, what HTML form parameters may have been sent as part of this request, and other relevant data. The servlet can then perform whatever logic it was programmed with and can generate data to send back to the client. It sends this data back to the client via the response object.

Once the servlet is done with the request, the servlet container ensures that the response is properly flushed and returns control back to the host web server.

1.4 Comparing Servlets with Other Technologies

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

Servlets have the following advantages over other server extension mechanisms:

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

1.5 Relationship to Java 2 Platform Enterprise Edition

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

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

CHAPTER 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 API that implement the `Servlet` interface are `GenericServlet` and `HttpServlet`. For most purposes, developers will typically extend `HttpServlet` to implement their servlets.

2.1 Request Handling Methods

The basic `Servlet` interface defines a `service` method for handling client requests. This method is called for each request that the servlet container routes to an instance of a servlet. Multiple request threads may be executing within the service method at any time.

2.1.1 HTTP Specific Request Handling Methods

The `HttpServlet` abstract subclass adds additional methods 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 rest of these methods are considered to be advanced methods for use by programmers very familiar with HTTP programming.

The `doPut` and `doDelete` methods allow Servlet Developers to support HTTP/1.1 clients which support these features. The `doHead` method in `HttpServlet` is a specialized method that will execute the `doGet` method, but only return the headers produced by the `doGet` method to the client. The `doOptions` method automatically determines which HTTP methods are directly supported by the servlet and returns that information to the client. The `doTrace` method causes a response with a message containing all of the headers sent in the TRACE request.

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

2.1.2 Conditional GET Support

The `HttpServlet` interface defines the `getLastModified` method to support conditional get operations. A conditional get operation is one in which the client requests a resource with the HTTP GET method and adds a header that indicates that the content body should only be sent if it has been modified since a specified time.

Servlets that implement the `doGet` method and that provide content that does not necessarily change from request to request should implement this method to aid in efficient utilization of network resources.

2.2 Number of Instances

In the default case of a servlet not implementing SingleThreadModel and not hosted in a distributed environment, the servlet container must use only one instance of a servlet class per servlet definition.

In the case of a servlet that implements the `SingleThreadModel` interface, the servlet container may instantiate multiple instances of that servlet so that it can handle a heavy request load while still serializing requests to a single instance.

In the case where a servlet was deployed as part of an application that is marked in the deployment descriptor as *distributable*, there is one instance of a servlet class per servlet definition per VM in a container. If the servlet implements the `SingleThreadModel` interface as well as is part of a distributable web application, the container may instantiate multiple instances of that servlet in each VM of the container.

2.2.1 Note about SingleThreadModel

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

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, directly or indirectly through the `GenericServlet` or `HttpServlet` abstract classes, implement.

2.3.1 Loading and Instantiation

The servlet container is responsible for loading and instantiating a servlet. The instantiation and loading can occur when the engine is started or it can be delayed until the container determines that it needs the servlet to service a request.

First, a class of the servlet's type must be located by the servlet container. If needed, the servlet container loads a servlet using normal Java class loading facilities from a local file system, a remote file system, or other network services.

After the container has loaded the `Servlet` class, it instantiates an object instance of that class for use.

It is important to note that there can be more than one instance of a given `Servlet` class in the servlet container. For example, this can occur where there was more than one servlet definition that utilized a specific servlet class with different initialization parameters. This can also occur when a servlet implements the `SingleThreadModel` interface and the container creates a pool of servlet instances to use.

2.3.2 Initialization

After the servlet object is loaded and instantiated, the container must initialize the servlet before it can handle requests from clients. Initialization is provided so that a servlet can read any persistent configuration data, initialize costly resources (such as JDBC™ based

connection), and perform any other one-time activities. The container initializes the servlet by calling the `init` method of the `Servlet` interface with a unique (per servlet definition) object implementing the `ServletConfig` interface. This configuration object allows the servlet to access name-value initialization parameters from the servlet container's configuration information. The configuration object also gives the servlet access to an object implementing the `ServletContext` interface which describes the runtime environment that the servlet is running within. See Chapter 3, "Servlet Context" for more information about the `ServletContext` interface.

2.3.2.1 Error Conditions on Initialization

During initialization, the servlet instance can signal that it is not to be placed into active service by throwing an `UnavailableException` or `ServletException`. If a servlet instance throws an exception of this type, it must not be placed into active service and the instance must be immediately released by the servlet container. The `destroy` method is not called in this case as initialization was not considered to be successful.

After the instance of the failed servlet is released, a new instance may be instantiated and initialized by the container at any time. The only exception to this rule is if the `UnavailableException` thrown by the failed servlet which indicates the minimum time of unavailability. In this case, the container must wait for the minimum time of unavailability to pass before creating and initializing a new servlet instance.

2.3.2.2 Tool Considerations

When a tool loads and introspects a web application, it may load and introspect member classes of the web application. This will trigger static initialization methods to be executed. Because of this behavior, a Developer should not assume that a servlet is in an active container runtime unless the `init` method of the `Servlet` interface is called. For example, this means that a servlet should not try to establish connections to databases or Enterprise JavaBeans™ compenent architecture containers when its static (class) initialization methods are invoked.

2.3.3 Request Handling

After the servlet is properly initialized, the servlet container may use it to handle requests. Each request is represented by a request object of type `ServletRequest` and the servlet can create a response to the request by using the 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 container must provide the request and response objects as implementations of `HttpServletRequest` and `HttpServletResponse`.

It is important to note that a servlet instance may be created and placed into service by a servlet container but may handle no requests during its lifetime.

2.3.3.1 Multithreading Issues

During the course of servicing requests from clients, a servlet container may send multiple requests from multiple clients through the `service` method of the servlet at any one time. This means that the Developer must take care to make sure that the servlet is properly programmed for concurrency.

If a Developer wants to prevent this default behavior, he can program the servlet to implement the `SingleThreadModel` interface. Implementing this interface will guarantee that only one request thread at a time will be allowed in the service method. A servlet container may satisfy this guarantee by serializing requests on a servlet or by maintaining a pool of servlet instances. If the servlet is part of an 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.

If a Developer defines a `service` method (or methods such as `doGet` or `doPost` which are dispatched to from the `service` method of the `HttpServlet` abstract class) with the `synchronized` keyword, the servlet container will, by necessity of the underlying Java runtime, serialize requests through it. However, the container must not create an instance pool as it does for servlets that implement the `SingleThreadModel`. It is strongly recommended that developers not synchronize the service method or any of the `HttpServlet` service methods such as `doGet`, `doPost`, etc.

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.

2.3.3.3 Thread Safety

A Developer should note that implementations of the request and response objects are not guaranteed to be thread safe. This means that they should only be used in the scope of the request handling thread. References to the request and response objects should not be given to objects executing in other threads as the behavior may be nondeterministic.

2.3.4 End of Service

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

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

Before the servlet container can call the `destroy` method, it must allow any threads that are currently running in the `service` method of the servlet to either complete, or exceed a server defined time limit, before the container can proceed with calling the `destroy` method.

Once the `destroy` method is called on a servlet instance, the container may not route any more requests to that particular 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 3

Servlet Context

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

A `ServletContext` is rooted at a specific path within a web server. For example a context could be located at `http://www.mycorp.com/catalog`. All requests that start with the `/catalog` request path, which is known as the *context path*, will be routed to this servlet context.

3.1 Scope of a ServletContext

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

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

3.2 Initialization Parameters

A set of context initialization parameters can be associated with a web application and are made available by the following methods of the `ServletContext` interface:

- `getInitParameter`
- `getInitParameterNames`

Initialization parameters can be used by an application developer to convey setup information, such as a webmaster's e-mail address or the name of a system that holds critical data.

3.3 Context Attributes

A servlet can bind an object attribute into the context by name. Any object 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`

3.3.1 Context Attributes in a Distributed Container

Context attributes exist locally to the VM in which they were created and placed. This prevents the `ServletContext` from being used as a distributed shared memory store. If information needs to be shared between servlets running in a distributed environment, that information should be placed into a session (See Chapter 8, "Sessions"), a database or set in an Enterprise JavaBean.

3.4 Resources

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

- `getResource`
- `getResourceAsStream`

Both the `getResource` and `getResourceAsStream` methods take a `String` argument giving the path of the resource relative to the root of the context.

It is important to note that these methods give access to static resources from whatever repository the server uses. This hierarchy of documents may exist in a file system, in a web application archive file, on a remote server, or some other location. These methods are not used to obtain dynamic content. For example, in a container supporting the JavaServer Pages specification[1], a method call of the form `getResource("/index.jsp")` would return the JSP source code and not the processed output. See Chapter 8, “Dispatching Requests” for more information about accessing dynamic content.

3.5 Multiple Hosts and Servlet Contexts

Many web servers support the ability for multiple logical hosts to share the same IP address on a server. This capability is sometimes referred to as "virtual hosting". If a servlet container’s host web server has this capability, each unique logical host must have its own servlet context or set of servlet contexts. A servlet context can not be shared across virtual hosts.

3.6 Reloading Considerations

Many servlet containers support servlet reloading for ease of development. Reloading of servlet classes has been accomplished by previous generations of servlet containers by creating a new class loader to load the servlet which is distinct from class loaders used to load other servlets or the classes that they use in the servlet context. This can have the undesirable side effect of causing object references within a servlet context to point at a different class or object than expected which can cause unexpected behavior.

Therefore, when a Container Provider implements a class reloading scheme for ease of development, they must ensure that all servlets, and classes that they may use, are loaded in the scope of a single class loader guaranteeing that the application will behave as expected by the Developer.

1. The JavaServer Pages specification can be found at `http://java.sun.com/products/jsp`

3.7 Temporary Working Directories

It is often useful for Application Developers to have a temporary working area on the local filesystem. All servlet containers must provide a private temporary directory per servlet context and make it available via the context attribute of `javax.servlet.context.tempdir`. The object associated with the attribute must be of type `java.io.File`.

CHAPTER 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 by the HTTP headers and the message body of the request.

4.1 Parameters

Request parameters are strings sent by the client to a servlet container as part of a request. When the request is a `HttpServletRequest`, the attributes are populated from the URI query string and possibly posted form data. The parameters are stored by the servlet container 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 always equal the first value in the array of `String` objects returned by `getParameterValues`.

All form data from both the query string and the post body are aggregated into the request parameter set. The order of this aggregation is that query string data takes precedence over post body parameter 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)`.

Posted form data is only read from the input stream of the request and used to populate the parameter set when all of the following conditions are met:

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 calls any of the `getParameter` family of methods on the request object.

If any of the `getParameter` family of methods is not called, or not all of the above conditions are met, the post data must remain available for the servlet to read via the request's input stream.

4.2 Attributes

Attributes are objects associated with a request. Attributes may be set by the container to express information that otherwise could not be expressed via the API, or may be set by a servlet to communicate information to another servlet (via `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 package name convention suggested by the Java Programming Language Specification[1] for package naming.

4.3 Headers

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

- `getHeader`
- `getHeaders`

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

- `getHeaderNames`

The `getHeader` method allows access to the value of a header given the name of the header. Multiple headers, such as the `Cache-Control` header, can be present in an HTTP request. If there are multiple headers with the same name in a request, the `getHeader` method returns the first header contained in the request. The `getHeaders` method allow access to all the header values associated with a particular header name returning an `Enumeration` of `String` objects.

Headers may contain data that is better expressed as an `int` or a `Date` object. The following convenience methods of the `HttpServletRequest` interface provide access to header data in a one of these formats:

- `getIntHeader`
- `getDateHeader`

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

4.4 Request Path Elements

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

- **Context Path:** The path prefix associated with the `ServletContext` that this servlet is a part of. If this context is the "default" context rooted at the base of the web server's URL namespace, this path will be an empty string. Otherwise, this 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.
- **PathInfo:** The part of the request path that is not part of the Context Path or the Servlet Path.

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

`ContextPath`	`/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`

4.5 Path Translation Methods

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

- `getRealPath`
- `getPathTranslated`

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

4.6 Cookies

The `HttpServletRequest` interface provides the `getCookies` method to obtain an array of cookies that are present in the request. These cookies are data sent from the client to the server on every request that the client makes. Typically, the only information that the client sends back as part of a cookie is the cookie name and the cookie value. Other cookie attributes that can be set when the cookie is sent to the browser, such as comments, are not typically returned.

4.7 SSL Attributes

If a request has been transmitted over a secure protocol, such as HTTPS, this information must be exposed via the `isSecure` method of the `ServletRequest` interface.

In servlet containers that are running in a Java 2 Standard Edition, v 1.2 or Java 2 Enterprise Edition, v 1.2 environment, if there is an SSL certificate associated with the request, it must be exposed 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`.

For a servlet container that is not running in a Java2 Standard Edition 1.2 environment, vendors may provide vendor specific request attributes to access SSL certificate information.

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.

TBD Containers still have to apply a number of heuristics to convert the Accept-Language into a Locale. Work under weigh to help in this area.

The `getLocales` method will return an `Enumeration` of `Locale` objects indicating, in decreasing order starting with the preferred locale, the locales that are acceptable to the client.

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

4.9 Request data encoding

Currently, many browsers do not send a char encoding qualifier with the Content-Type header. This leaves open the determination of the character encoding for reading Http requests. Many containers default in this case to the JVM default encoding, which causes a breakage when the request data has not been encoded with the same encoding as the platform default.

To aid 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 in this situation if necessary by calling this method. This method 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.

CHAPTER 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.

5.1 Buffering

In order to improve efficiency, a servlet container is allowed, but not required to by default, to buffer output going to the client. The following methods are provided via the `ServletResponse` interface to allow a servlet access to, and the setting of, buffering information:

- `getBufferSize`
- `setBufferSize`
- `isCommitted`
- `reset`
- `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 for this response, this method must return the `int` value of `0 (zero)`.

The servlet can request a preferred buffer size for the response by using the `setBufferSize` method. The actual buffer assigned to this request is not required to be the same size as requested by the servlet, but must be at least as large as the buffer size requested. This allows the container to reuse a set of fixed size buffers, providing a larger

buffer than requested if appropriate. This 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 or not any bytes from the response have yet been returned to the client. The `flushBuffer` method forces any content in the buffer to be written to the client.

The `reset` method clears any data that exists in the buffer as long as the response is not considered to be committed. All headers and the status code set by the servlet previous to the reset called must be cleared as well.

If the response is committed and the `reset` method is called, an `IllegalStateException` must be thrown. In this case, the response and its associated buffer will be unchanged.

When buffering is in use is filled, the container must immediatly flush the contents of the buffer to the client. If this is the first time for this request that data is sent to the client, the response is considered to be committed at this point.

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. If a previous header exists, it is replaced by the new header. In the case where a set of header values exist for the given name, all values are cleared and replaced with the new value.

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

Headers may contain data that is better expressed as 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`

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

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 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 had 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`.

TBD Make it clearer that these mechanisms should not destroy existing header information like Cookies

5.4 Internationalization

In response to a request by a client to obtain a document of a particular language, or perhaps due to preference setting by a client, a servlet can set the language attributes of a response back to a client. This information is communicated via the `Content-Language` header along with other mechanisms described in the HTTP/1.1 specification. The language of a

response can be set with the `setLocale` method of the `ServletResponse` interface. This method must correctly set the appropriate HTTP headers 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 will ensure that the returned `PrintWriter` is configured appropriately for the target `Locale`.

If the `setContentType` method is called after the `setLocale` method and there is a `charset` component to the given content type, the `charset` specified in the content type overrides the value set via the call to `setLocale`.

5.5 Closure of Response Object

A number of events can indicate that the servlet has provided all of the content to satisfy the request and that the response object can be considered to be closed. The events are:

- The termination of the service method of the servlet.
- When 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.

When a response is closed, all content in the response buffer, if any remains, must be immediately flushed to the client.

CHAPTER 6

Servlet Filtering

Filters are a new feature in the Java servlet API for version 2.3. This chapter describes the new API classes and methods that provide a lightweight framework for servlet filtering in the API. It describes the ways that filters can be configured in a web application, and describes some of the conventions and semantics around how they can be implemented..

Filters allow on the fly transformations of the payload and header information both of the request in to a servlet and on the response from a servlet.

API documentation for this model is provided in the API definitions chapters of this document. Configuration syntax for filters is given by the Document Type Definition in Chapter 13. Both should be referenced when reading this chapter.

6.1 What is a filter ?

A filter is a reusable piece of code that transforms either the content of an HTTP request or response and can also modify header information. Filters differ from Servlets in that they do not themselves usually create a response, rather, they are there to modify or adapt the request into a servlet and modify or adapt the response from a Servlet.

The main functionality areas that is available to the Filter author are:-

- They can intercept the invokation of a Servlet before the Servlet is called.
- They can look at the request into a Servlet before the Servlet is called.
- They can modify the request headers and request data by providing customized versions of the request object that wrap the real request.
- They can modify the response headers and response data by providing customized versions of the response object that wrap the real response.
- They can intercept the invokation of a Servlet after the Servlet is called.
- They can be configured to act on a Servlets or on groups of Servlets.

- A Servlet can be configured to be filtered by zero, one or more filters in a specifiable order.

6.1.1 Examples of Filtering Components

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

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 in the Java Servlet API. The implementation class is packaged in the Web Archive along with the rest of the static content and Servlets that make up the web application. Each Filter is declared using the <filter> syntax in the deployment descriptor. A Filter or collection of Filters can be configured to be invoked by defining a number of <filter-mapping> elements in the deployment descriptor. The syntax associates the filter or group of filters with a particular Servlet. This is done by mapping a filter to a particular servlet by the servlet's logical name, or mapping to a group of Servlets by mapping a filter to a url pattern.

6.2.1 Filter Lifecycle

After the time when the web application containing filters is deployed, and before an incoming request for a Servlet in the web application causes an invokation of the servlet's `service()` method, the container must look through the list of filter mappings to locate the list of filters that must be applied to the servlet. How this list is built is described below. The container must ensure at some point in this time that, for each filter that is to be applied, it has instantiated a filter of the appropriate class, and called `setFilterConfig(FilterConfig config)` on each filter instance in the list. The

container ensures that the `javax.servlet.FilterConfig` instance that is passed in to this callhas been initialized with the filter name as declared in the deployment descriptor for that filter, with the collection of remaining filters in the filter list to support the `getFilters()` call and with the set of initialization parameters declared for the filter in the deployment descriptor.

When the container recieves the incoming request, it takes the first filter instance in the list and calls its `doFilter()` method, passing in the `ServletRequest` and `ServletResponse`.

The `doFilter()` method of a Filter will typically be implemented following this or some subset of this pattern:-

1) It will examine the request headers

2) It may wrap the request object passed into its `doFilter()` method with a customized implementation of ServletRequest or HttpServletRequest if it wishes to modify request headers or data.

3) It may wrap the response object passed in to its `doFilter()` method with a cusomized implementation of ServletRequest or HttpServletRequest if it wishes to modify response headers or data.

4) It either obtains a reference to the next filter in the stack from the FilterConfig and calls the `doFilter()` method (passing in the request and response it was called with, or the wrapped versions it may have created), or chooses to block the request by not making the call. In the latter case, the filter is responsible for filling out the response.

5) It may examine response headers after it has invoked the next filter in the chain.

6) Alternatively, the Filter may throw an exception to indicate an error in processing.

7) Optionally, a filter may choose to bypass invokation of particular filters in the list that have yet to be invoked. In order to do this, it can iterate over the remaining Filter objects in the list via the `getFilters()`method on its `FilterConfig` interface.

Note that the last object in the filter list is the Servlet that will ultimately be invoked. The container must supply an implementation of the Filter interface to fill this last entry in the list. This last Filter imeplementation in the list simply delegates the `doFilter()` call to the `service()` method of the Servlet.

Before the container can clean up filter instances throughout the lifetime of a web application, it must call the `setFilterConfig()`method on the Filter passing in null to indicate that the Filter is being taken out of service.

6.2.2 Filter environment

A set of initialization parameters can be associated with a filter using the init-params ekement 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. Additionally, the Filter Config affords access to the ServletContext of the web application for the loading of resources, for logging functionality or for storage of state in the ServletContext's attribute list.

6.2.3 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 - this is used to map the filter to a servlet or URL

filter-class - this is used by the container to identify the filter type

init-params - the initialization parameters for a filter

and optionally can specify icons, a textual description and a display name for tool manipulation.

Once a Filter has been declared in the deployment descriptor, the assembler uses the <filter-mapping> element to define to which Servlets in the web application theFilter is to be applied. Filters can either be associated with a Servlet by using the <servlet-name> style:-

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

In this case the Image Filter is applied to the Servlet with servlet-name 'Image Servlet'.

or by using the <url-pattern> style of filter mapping:-

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

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

In this case, the Logging Filter is applied to all the Servlets 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 12.1.

The order in which the container builds the list of filters to be applied for a particular request URI is

1) The URL pattern matching filter-mappings in the same as the order that those elements appear in the deployment descriptor, and then

2) The servlet-name matching filter-mappings in the same as the order that those elements appear in the deployment descriptor

CHAPTER 7

Sessions

The Hypertext Transfer Protocol (HTTP) is by design a stateless protocol. To build effective web applications, it is imperative that a series of different requests from a particular client can 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 number of approaches to track a user's session without involving the Developer in the nuances of any one approach.

7.1 Session Tracking Mechanisms

There are several strategies to implement session tracking.

7.1.1 URL Rewriting

URL rewriting is the lowest common denominator of session tracking. In cases where a client will not accept a cookie, URL rewriting may be used by the server to establish session tracking. URL rewriting involves adding data to the URL path that can be interpreted by the container on the next request to associate the request with a session.

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

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

7.1.2 Cookies

Session tracking through HTTP cookies is the most used session tracking mechanism and is required to be supported by all servlet containers. The container sends a cookie to the client. The client will then return the cookie on each subsequent request to the server unambiguously associating the request with a session. The name of the session tracking cookie must be `JSESSIONID`.

7.1.3 SSL Sessions

Secure Sockets Layer, the encryption technology which is 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 an accepted session. A servlet container can easily use this data to serve as the mechanism for defining a session.

7.1.4 Session Integrity

Web containers must be able to support the integrity of the HTTP session when servicing HTTP requests from clients that do not support the use of cookies. To fulfil this requirement in this scenario, web containers commonly support the URL rewriting mechanism.

7.2 Creating a Session

Because HTTP is a request-response based protocol, a session is considered to be new until a client “joins” it. A client joins a session when session tracking information has been successfully 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 the 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. This implies that the servlet container has no mechanism by which to associate a request with a previous request.

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

7.3 Session Scope

`HttpSession` objects must be scoped at the application / servlet context level. The underlying mechanism, such as the cookie used to establish the session, can be shared between contexts, but the object exposed, and more importantly the attributes in that object, must not be shared between contexts.

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 that handles a request identified as being a part of the same session.

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

- `valueBound`
- `valueUnbound`

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

7.5 Session Timeouts

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

The default timeout period for sessions is defined by the servlet container and can be obtained via the `getMaxInactiveInterval` method of the `HttpSession` interface. This timeout can be changed by the Developer using the `setMaxInactiveInterval` of the `HttpSession` interface. The timeout periods used by these methods is defined in seconds. If the timeout period for a session is set to `-1`, the session will never expire.

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 handled by the servlet context.

7.7 Important Session Semantics

--- need a line here ---

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 to synchronize access to resources stored in the session as appropriate.

7.7.2 Distributed Environments

Within an application that is marked as distributable, all requests that are part of a session can only be handled on a single VM at any one time. In addition the container must be able to handle all objects placed into instances of the `HttpSession` class using the `setAttribute` or `putValue` methods approriately.

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

The servlet container may throw an `IllegalArgumentException` if a object is placed into the session for which it cannot support the mechanism necessary for migration of the session..

These restrictions mean that the Developer is ensured that there are no additional concurrency issues beyond those encountered in a non-distributed container. In addition, the Container Provider can ensure scalability 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.

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 a 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 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.

8.1 Obtaining a RequestDispatcher

An object implementing the `RequestDispather` 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`. This path is used to look up a servlet, wrap it with a `RequestDispatcher` object, and return it. If no servlet can be resolved based on the given path, a `RequestDispatcher` is provided that simply 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 known to the `ServletContext` by the given name, it is wrapped with a `RequestDispatcher` object and 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, paths which are not relative to the root of the `ServletContext` but instead are relative to the path of the current request, 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`, however it does not require a complete path within the context to be given as part of the argument to operate. The servlet container can use the information in the request object to transform the given relative path to a complete path. For example, in a context rooted at `'/'`, a request to `/garden/tools.html`, a request dispatcher obtained via `ServletRequest.getRequestDispatcher("header.html")` will behave exactly like a call to `ServletContext.getRequestDispatcher("/garden/header.html")`.

8.1.1 Query Strings in Request Dispatcher Paths

In the `ServletContext` and `ServletRequest` methods which allow the creation of a `RequestDispatcher` using path information, optional query string information may be attached 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);
```

The contents of the query string are added to the parameter set that the included servlet has access to. The parameters are ordered so that any parameters specified in the query string used to create the `RequestDispatcher` take precedence. The parameters associated with a RequestDispatcher are only scoped for the duration of the `include` or `forward` call.

8.2 Using a Request Dispatcher

To use a request dispatcher, a developer needs to call either the `include` or `forward` method of the `RequestDispatcher` interface using the `request` and `response` arguments that were passed in via the `service` method of the `Servlet` interface.

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

8.3 Include

The `include` method of the `RequestDispatcher` interface may be called at any time. The target servlet has access to all aspects of the request object, but can only write information to the `ServletOutputStream` or `Writer` of the response object as well as the ability to commit a response by either writing content past the end of the response buffer or explicitly calling the `flush` method of the `ServletResponse` interface. The included servlet cannot set headers or call any method that affects the headers of the response. Any attempt to do so should be ignored.

8.3.1 Included Request Parameters

When a servlet is being used from within an `include`, it is sometimes necessary for that servlet to know the path by which it was invoked and not the original request paths. 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 a `NamedDispatcher`, these attributes are not set.

8.4 Forward

The `forward` method of the `RequestDispatcher` interface may only be called by the calling servlet if no output has been committed to the client. If output data exists in the response buffer that has not been committed, the content must 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 reflect those of the original request.

Before the `forward` method of the `RequestDispatcher` interface returns, the response must be committed and closed by the servlet container.

8.4.1 Query String

The request dispatching mechanism aggregate query string parameters on forwarding or including requests.

8.5 Error Handling

Only runtime exceptions and checked exceptions of type `ServletException` or `IOException` should be propagated to the calling servlet if thrown by the target of a request dispatcher. All other exceptions should be wrapped as a `ServletException` and the root cause of the exception set to the original exception.

CHAPTER 9

Web Applications

A web application is a collection of servlets, html pages, classes, and other resources that can be bundled and run on multiple containers from multiple vendors. 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 also 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 only be run on one VM at any one time. This behavior can be overridden if the application is marked as “distributable” via its the deployment descriptor. When an application is marked as distributable, the Developer must obey a more restrictive set of rules than is expected of a normal web application. These specific rules are called out throughout this specification.

9.1 Relationship to ServletContext

The servlet container must enforce a one to one correspondence between a web application and a `ServletContext`. A `ServletContext` object can be viewed as a Servlet’s view onto its application.

9.2 Elements of a Web Application

A web application may consist of the following items:

- Servlets

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

9.3 Distinction Between Representations

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

9.4 Directory Structure

A web application exists as a structured hierarchy of directories. The root of this hierarchy serves as a document root for serving files that are part of this context. For example, for a web application located at `/catalog` in a web server, the `index.html` file located at the base of the web application hierarchy can be served to satisfy a request to `/catalog/index.html`.

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. It is important to note that 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.

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

- `/WEB-INF/web.xml` deployment descriptor
- `/WEB-INF/classes/*` directory for servlet and utility classes. The classes in this directory are used by the application class loader to load classes from.
- `/WEB-INF/lib/*.jar` area for Java ARchive files which contain servlets, beans, and other utility classes useful to the web application. All such archive files are used by the web application class loader to load classes from.

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

The web application classloader loads classes first from the WEB-INF/classes directory and then from library JARs. For the latter case, the classloader should attempt to load from library JARs in the same order that they appear as WAR archive entries.

9.4.1 Sample Web Application Directory Structure

Illustrated here 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
```

9.5 Web Application Archive File

Web applications can be packaged and signed, using the standard Java Archive tools, into a Web ARchive format (war) file. For example, an application for issue tracking could be distributed in an archive with the filename `issuetrack.war`.

When packaged into such a form, a META-INF directory will be present which contains information useful to the Java Archive tools. If this directory is present, the servlet container must not allow it be served as content to a web client's request.

9.6 Web Application Configuration Descriptor

The following types of configuration and deployment information exist in the web application deployment descriptor:

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

All of these types of information are conveyed in the deployment descriptor (See Chapter 13, “Deployment Descriptor”).

9.6.1 Dependencies on extensions: Library Files

Groups of applications commonly make use of the code or resources contained in a library file or files installed container-wide in current implementations of web containers. The application developer needs to be able to know what extensions are installed on a web container for portability, and in creating a web application that may depend on such libraries, containers need to know what dependencies on such libaries Servlets in a WAR may have.

If web containers have a mechanism by which they can expose to the application classloaders of every web app therein extra JAR files containing resources and code, then it is recommended that they provide a user-friendly way of editing and configuring these library files or extensions, and that they expose information about what extensions are available to web applications deployed on the web container. Application developers that depend on the installation of library JARs installed on a web container should provide a META-INF/MANIFEST.MF entry in the WAR file listing the extensions that the WAR depends upon. The format of the manifest entry follows the 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 must be able to recognise such declared dependencies as expressed in the optional manifest entry in a WAR file, or in the manifest entry of any of the libary JARs under the WEB-INF/lib entry in a WAR. If a web container is not able to satisfy the dependencies that a WAR has on a particular extension declared in this manner, it should reject the application.

9.6.2 Web Application Classloader

The classloader that a container uses to load a servlet in a WAR must not allow the WAR to override JDK or Java Servlet API classes, or allow Servlets in the WAR visibility of the web containers implementation classes.

If a web container has a mechanism for exposing container-wide library JARs to application classloaders, it is recommended that the application classloader be implemented in such a way that classes packaged within the WAR are able to override classes residing in container-wide library JARs.

9.7 Replacing a Web Application

Applications evolve and must occasionally be replaced. In a long running server it is ideal to be able to load a new web application and shut down the old one without restarting the container. When an application is replaced, a container should provide a robust approach to preserving session data within that application.

9.8 Error Handling

A web application may specify that when errors occur, other resources in the application are used. These resources are specified in the deployment descriptor. If the location of the error handler is a servlet or a JSP, the following request attributes can be set:

- `javax.servlet.error.status_code`
- `javax.servlet.error.exception_type`
- `javax.servlet.error.message`

These attributes allow the servlet to generate specialized content depending on the status code, exception type and message of the error. The types of the attribute objects are java.lang.Integer, java.lang.Class and java.lang.String.

The deployment descriptor defines a list of error page descriptions that the container must examine when a Servlet generates an error. The container examines the list in the order that it is defined, and attempts to match the error condition, by status code or by exception class. On the first successful match of the error condition the container serves back the resource defined in the corresponding location.

9.9 Welcome Files

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

The purpose of this mechanism is to allow the deployer to specify an ordered list of partial URIs for the container to appeand to a request for a URI that corresponds to a directory entry in the WAR that is not mapped to a web component. Such a request is known here as a valid partial request. The most common example is to define a welcome file of ‘index.html’ so that a request to a URL like ‘host:port/webapp/directory’ where ‘directory’ is a directoy entry in the WAR that is not mapped to a Servlet or JSP is served back to the client as ‘host:port/webapp/directory/index.html’.

If a web container recieves 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 forward the request to the first resource in the WAR that matches.

If no matching resource is found, the container may handle the request in a manner it finds suitable. For some configurations this may mean serving back a directory listing, for other configurations it may return a 404 status code.

Consider a web application where:-

- The deployment descriptor lists index.html, default.jsp as its welcome files.
- ServletA is exact mapped 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 forwarded to /foo/index.html
- A request URI of /catalog/ will will be forwarded to /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 or register.jsp, or other behavior suitable for the container.

TBD ? Add a flag in the deployment descriptor that allows the developer to control the behavior of whether to return a directory listing or whether to send back a 404 on a per application basis.

9.10 Web Application Environment

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

As servlets are an integral component type of J2EE, 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`
- `resource-ref`

The `env-entry` element contains information to set up basic environment entry names relative to the `java:comp/env` context, the expected Java type of the environment entry value (the type of object returned from the JNDI lookup method), and an optional environment entry value. The `ejb-ref` element contains the information needed to allow a servlet to locate the home interfaces of a enterprise bean. The `resource-ref` element contains the information needed to set up a resource factory.

The requirements of the J2EE environment with regards to setting up the environment are described in Chapter 5 of the Java 2 Platform Enterprise Edition v 1.3 specification[1]. Servlet containers that are not part of a J2EE compliant implementation are encouraged, but not required, to implement the application environment functionality described in the J2EE specification.

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

CHAPTER 10

Application Lifecycle Events

10.1 Introduction

New for the Servlet Specification v2.3 is support for application level events. Application events give the web application developer greater control over interactions with the ServletContext and HttpSession objects, allow for better code factorization and increased efficiency in managing resources that the web application uses.

10.2 Event Listeners

In extending the servlet api to support event notifications for the state changes in the servlet context and http session objects, the developer has a greater ability to manage resources and state at the logical level of the web application. Servlet context listeners can be used to manage resources or state held at a VM level for the application. Http session listeners can be used to manage state or resources associated with a serries of requests made into a web application from the same client or user.

Event listeners are Java classes following the JavaBeans design. They are provided by the developer in the WAR. They implement one or more of the event listener interfaces in the Servlet API v 2.3 and are instantiated and registered in the web container at the time of deployment of the web application. 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.

The events are shown in the following table, together with the 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 just been created, or has been 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 on the API, refer to the API reference at the end of this document.

To illustrate one possible use of the event scheme, consider a simple web application containing a number of servlets that make use of a database. The developer can provide a servlet context listener class that manages the database connection. When the application starts up the listener class is notified and so has the chance to log on to the database and store the connection in the servlet context. Any servlet in the application may access the connection during activity in the web application. When either the web server is shut down, or the application is removed from the web server, the listener class is notified and so the database connection can be at that point closed.

10.3 Configuration of Listener Classes

The developer of the web application writes listener classes to implement one or more of the four listener classes in the Servlet API. Each listener class must provide 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.

Listener classes are declared in the web application deployment descriptor using the <listener> element. The web application deployment descriptor lists the listener classes by classname in the order that it wishes them to be invoked if there are more than one. The web container is responsible for creating an instance of each listener class defined in the deployment descriptor and registering it for event notifications prior to the first request being serviced by the application. The web container checks the interfaces implemented by each listener class and registers the listener instances according to the interfaces they implement in the order that they appear in the deployment descriptor.

Note On application shutdown, all listeners to sessions must be notified of session invalidations prior to context listeners being notified of application shutdown.

Here is an example of 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 wishes for com.acme.MyConnectionManager to be notified of servlet context lifecycle events before com.acme.MyLoggingModule. Here is what the deployment descriptor for this application would look like:-

```
<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
```

```
    ...etc
</web-app>
```

10.4 Listener Instances and Threading

The container is required to instantiate the listener classes in a web application prior to exectuion of the first request into the application. The container must reference each listener instance until the last request is serviced for the web application.

Attribute list changes on both the servlet context and the http session object may occur concurently. The container is not required to synchronize the resulting notifications to attribute listener classes. Listener beans that maintain state hold the responsibility for ensuring integrity of data by handling this case explicitly.

10.5 Distributed Containers

In distributed web containers, Http session instances are scoped to the VM servicing requests within the session, and the servlet context is scoped to one per web container VM. Distributed containers are not required to propogate either servlet context events or Http session events in a distributed manner.

TBD When a session migrates from one VM to another, does the container send out an event notification ?

10.6 Session Events- Invalidation vs Timeout

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 invalidated it using the invalidate() method. There is currently sufficient API with the listeners and API methods on the HTTPSession class to determine this situation indirectly.

CHAPTER **11**

Mapping Requests to Servlets

Previous versions of this specification have allowed servlet containers a great deal of flexibility in mapping client requests to servlets only defining a set a suggested mapping techniques. This specification now requires a set of mapping techniques to be used for web applications which are deployed via the Web Application Deployment mechanism. Just as it is highly recommended that servlet containers use the deployment representations as their runtime representation, it is highly recommended that they use these path mapping rules in their servers for all purposes and not just as part of deploying a web application.

11.1 Use of URL Paths

Servlet containers must use URL paths to map requests to servlets. The container uses the RequestURI from the request, minus the Context Path, as the path to map to a servlet. In determining the part of the URL signifying the context path, the container must choose the longest matching available context path from the list of web applications it hosts. The URL path mapping rules are as follows (where the first match wins and no further rules are attempted):

1. The servlet container will try to match the exact path of the request to a servlet.
2. The container will then try to recursively match the longest path prefix mapping. This process occurs by stepping down the path tree a directory at a time, using the `'/'` character as a path separator, and determining if there is a match with a servlet.
3. If the last node of the url-path contains an extension (`.jsp` for example), the servlet container will try to match a servlet that handles requests for the extension. An extension is defined as the part of the path after the last `'.'` character.
4. If neither of the previous two 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 in this case.

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 as a path mapping.
- A string beginning with a `'*.'` prefix is used as an extension mapping.
- All other strings are used as exact matches only
- A string containing only the `'/'` character indicates that servlet specified by the mapping becomes 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.

11.2.1 Implicit Mappings

If the container has an internal JSP container, the `*.jsp` extension is implicitly mapped to it so that JSP pages may be executed on demand. If the web application defines a `*.jsp` mapping, its mapping takes precedence over this 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 by a container to a server side include functionality.

11.2.2 Example Mapping Set

Consider the following set of mappings:

Table 3: Example Set of Maps

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

The following behavior would result:

Table 4: 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 as it is not an exact match and the rule doesn’t include the ’*’ character.

CHAPTER 12

Security

Web applications are created by a Developer, who then gives, sells, or otherwise transfers the application to the Deployer for installation into a runtime environment. It is useful for the Developer to communicate attributes about how the security should be set up for a deployed application.

As with the web application directory layout and deployment descriptor, the elements of this section are only required as a deployment representation, not a runtime representation. However, it is recommended that containers implement these elements as part of their runtime representation.

12.1 Introduction

A web application contains many resources that can be accessed by many users. Sensitive information often traverses unprotected open networks, such as the Internet. In such an environment, there is a substantial number web applications that have some level of security requirements. Most servlet containers have the specific mechanisms and infrastructure to meet these requirements. Although the quality assurances and implementation details may vary, all of these mechanisms share some of the following characteristics:

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

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 that was derived from the deployment descriptor and configured by the deployer to enforce authentication.

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 that the client authenticated with. The `isUserInRole` queries the underlying security mechanism of the container to determine if a particular user is in a given security role. The `getUserPrinciple` method returns a java.security.Pricipal object.

These APIs allow servlets to make business logic decisions based on the logical role of the remote user. They also allow the servlet to determine the principal name of the current user.

If `getRemoteUser` returns `null` (which means that no user has been authenticated), the `isUserInRole` method will always return `false` and the `getUserPrincipal` will always return `null`.

12.4 Roles

A role is an abstract logical grouping of users that is defined by the Application Developer or Assembler. When the application is deployed, these roles are mapped by a Deployer to security identities, such as 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 that calling principal. For example,

1. When 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. If the principal's user group matches the user group in the operational environment that the security role has been mapped to, the principal is in the security role.

2. When a deployer has mapped a security role to a principal name in a security policy domain, the principal name of the calling principal is retrieved from its security attributes. If the principal is the same as the principal to which the security role was mapped, the calling principal is in the security role.

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

12.5.1 HTTP Basic Authentication

HTTP Basic Authentication is the authentication mechanism defined in the HTTP/1.1 specification. This mechanism is based on a username and password. A web server requests a web client to authenticate the user. As part of the request, the web server passes the string called the *realm* of the request in which the user is to be authenticated. It is important to note that the realm string of the Basic Authentication mechanism does not have to reflect any particular security policy domain (which confusingly, can also be 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 as the user password is transmitted with a simple base64 encoding and the target server is not authenticated. However, additional protection, such as applying a secure transport mechanism (HTTPS) or using security at the network level (such as the IPSEC protocol or VPN strategies) can alleviate some of these concerns.

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. This authentication method is not as secure as any private key scheme such as HTTPS Client Authentication. As Digest Authentication is not currently in widespread use, servlet containers are not required, but are encouraged, to support it.

12.5.3 Form Based Authentication

The look and feel of the “login screen” cannot be controlled with an HTTP browser’s built in authentication mechanisms. Therefore this specification defines a 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 to be used with this mechanism. The login form must contain fields for the user to specify username and 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 if the user has been authenticated. If so, and dependent on the user’s authority to access the resource, the requested web resource is activated and returned. If the user is not authenticated, all of the following steps occur:

1. The login form associated with the security constraint is returned to the client. The URL path which triggered the authentication is stored by the container.
2. The client fills out the form, including the username and password fields.
3. The form is posted back to the server.
4. The container processes the form to authenticate the user. If authentication fails, the error page is returned.
5. The authenticated principal is checked to see if it is in an authorized role for accessing the original web request.
6. The client is redirected to the original resource using the original stored URL path.

If the user is not successfully authenticated, the error page is returned to the client. It is recommended that the error page contains information that allows the user to determine that the authorization failed.

Like Basic Authentication, this is not a secure authentication protocol as the user password is transmitted as plain text and the target server is not authenticated. However, additional protection, such as applying a secure transport mechanism (HTTPS) or using security at the network level (IPSEC or VPN) can alleviate some of these concerns.

12.5.3.1 Login Form Notes

Form based login and URL based session tracking can be problematic to implement. It is strongly recommended that form based login only be used when the session is 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 always work no matter what the resource is that requests it and avoids requiring that the server to process the outbound form to correct the action field.

Here is an HTML sample 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 mechanism is invoked as a result of a http request, all the original request parameters should be preserved when the container redirects the call to the requested resource within the web application on successful login.

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 single-signon from within the browser in an enterprise. Servlet containers that are not J2EE compliant are not required to support the HTTPS protocol.

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 that are deployed in the same container.
3. Require the user to re-authenticate only when crossing a security policy domain.

Therefore, a servlet container is required to track authentication information at the container level and not at the web application level allowing a user who is authenticated against one web application to access any other resource managed by the container which is restricted to the same security identity.

12.7 Propogation of Security Identity

The default mode for security identity propogation of a web user calling in to an EJB container is to propogate the security identity of the web user to the EJB container. Web applications may employ a strategy of programmatic security that allows web users to register themselves during the lifetime of a web application. In other cases, we applications may be configured to allow open access to all web users. In either case, the web users are not known to the web container or the EJB container.

The existence of a runAs element to the ejb-ref element in a web application deployment descriptor is an instruction to the web container that when a Servlet makes calls to an EJB. If present, the container must propogate the security identity of the caller to the EJB layer in terms of the security role name defined in the runAs element. The security role name must one of the security role names defined for the web application.

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 the URL pattern described in the web resource collection is subject to the constraint.

An authorization constraint is a set of roles that users must be a part of to access the resources described by the web resource collection. If the user is not part of a allowed role, the user is denied access to that resource.

A user data constraint indicates that the transport layer of the client server communication process satisfy the requirement of either guaranteeing content integrity (preventing tampering in transit) or guaranteeing confidentiality (preventing reading while in transit).

12.8.1 Default Policies

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

CHAPTER 13

Deployment Descriptor

The Deployment Descriptor conveys the elements and configuration information of a web application between Developers, Assemblers, and Deployers.

13.1 Deployment Descriptor Elements

The following types of configuration and deployment information exist in the web application deployment descriptor:

- ServletContext Init Parameters
- Session Configuration
- Servlet / JSP Definitions
- Servlet / JSP Mappings
- Application Lifecyle Listener classes
- Filter Definitions and Filter Mappings
- Mime Type Mappings
- Welcome File list
- Error Pages
- Security

See the DTD comments for further description of these elements.

13.1.1 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
2.3//EN" "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
```

13.2 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*, filter*, filter-mapping*, listener*,
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*)>


<!--
```

Declares a filter in the web application 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.

```
-->

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

<!--
The logical name of the filter. This name is used to map the filter.
-->
<!ELEMENT filter-name #PCDATA>

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

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

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

<!ELEMENT listener (listener-class)>

<!--
The listener-class element declares a class in the application must
be registered as a web application listener bean.
-->

<!ELEMENT listener-class (#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 authenticated using the configured
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 security-role-ref element defines a mapping between the name of
role called from a Servlet using
isUserInRole(String name) and the name of a security role defined
for the web application. For example,
to map the security role reference "FOO" to the security role with
role-name "manager" the sytax 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.
-->

<!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. If the optional runAs element is used, the security
identity of the call to the EJB must be propogated as the security
role with the same name to the EJB.
-->

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

<!--
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 runAs element must contain the name of a security role defined
for this web application.
-->

<!ELEMENT runAs (#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>
```

13.3 Examples

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

13.3.1 A Basic Example

```
<!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 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>
```

13.3.2 An Example of Security

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 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>
    <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
        <auth-constraint>
          <role-name>manager</role-name>
        </auth-constraint>
     </security-constraint>
   </web-app>
```

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.

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.

1. See RFC 1738

CHAPTER **14**

API Details

The following two chapters define the Java Servlet API in terms of Java classes, interfaces, the accompanying method signatures and javadoc comments.

Package
javax.servlet

Class Summary

Interfaces

Config	This is the super interface for objects in the Servlet API that pass configuration information to Servlets or Filters during initialization.
Filter	A filter is an object than perform filtering tasks on either the request to a servlet, or on the response from a servlet, or both. Filters do their filtering in the DoFilter method.
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.
ServletContextAttributesListener	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.
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

Class Summary	
ServletException	Defines a general exception a servlet can throw when it encounters difficulty.
UnavailableException	Defines an exception that a servlet throws to indicate that it is permanently or temporarily unavailable.

javax.servlet
Config

Syntax
```
public interface Config
```

All Known Subinterfaces: `FilterConfig`, `ServletConfig`

Description
This is the super interface for objects in the Servlet API that pass configuration information to Servlets or Filters during initialization. The configuration information contains initialization parameters, which are a set of name/value pairs, and a `ServletContext` object, which gives the calling object information about the web container.

Since: v 2.3

See Also: `ServletContext`

Member Summary	
Methods	
`getInitParameter(String)`	Returns a `String` containing the value of the named initialization parameter, or `null` if the parameter does not exist.
`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.
`getServletContext()`	Returns a reference to the `ServletContext` in which the caller is executing.

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`

javax.servlet
Filter

Syntax
```
public interface Filter
```

Description
A filter is an object than perform filtering tasks on either the request to a servlet, or on the response from a servlet, or both.

Filters do their 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 and a view into the Filter stack.

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: v 2.3

Member Summary	
Methods	
doFilter(ServletRequest, ServletResponse)	The doFilter method of the Filter is called by the container each time a request/response pair is passed through the stack due to a client request for the Servlet in the stack.
getFilterConfig()	Return the FilterConfig for this Filter.
setFilterConfig(FilterConfig)	The container calls this method when the Filter is instantiated and passes in a FilterConfig object.

Methods

doFilter(ServletRequest, ServletResponse)

```
public void doFilter(ServletRequest request, ServletResponse response)
```

The doFilter method of the Filter is called by the container each time a request/response pair is passed through the stack due to a client request for the Servlet in the stack. 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 stack using the getFilterConfig().getNext() call to obtain the next Filter and calling doFilter(),

4. b) **or** not pass on the request/response pair to the next entity in the filter stack

5. Directly set headers on the response after invokation of the next Filter

Throws: `ServletException`, `IOException`

---

getFilterConfig()

```
public FilterConfig getFilterConfig()
```

Return the FilterConfig for this Filter.

---

setFilterConfig(FilterConfig)

```
public void setFilterConfig(FilterConfig filterConfig)
```

The container calls this method when the Filter is instantiated and passes in a FilterConfig object. When the container is done with the Filter, it calls this method, passing in null.

javax.servlet
FilterConfig

Syntax
```
public interface FilterConfig extends Config
```

All Superinterfaces: Config

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

As well as holding the initialization parameters of a Filter, the FilterConfig provides a view into the next Filter and also of the remaining remaining Filters in the Filter stack of a Servlet. The last object in the Filter stack is always the Servlet that is being filtered. Containers provide a wrapper implementation of the Filter interface to wrap the Servlet so that Filters never know whether the next object in the stack is another Filter or the Servlet that it is filtering.

Since: v 2.3

See Also: Filter

Member Summary	
Methods	
getFilterName()	Returns the filter-name of this filter as defined in the deployment descriptor.
getFilters()	Returns the remaining Filter objects in the Filter stack in the order that they have been configured.
getNext()	Returns the next Filter object in the filter stack.

Inherited Member Summary
Methods inherited from interface Config
getServletContext(), getInitParameter(String), getInitParameterNames()

Methods

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

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

---

getFilters()

```
public java.util.Iterator getFilters()
```

Returns the remaining Filter objects in the Filter stack in the order that they have been configured. The purpose of this method it to allow Filters to decide to skip remaining filters in the stack if they wish. The Iterator returned does not support the optional remove() operation.

---

getNext()

```
public Filter getNext()
```

Returns the next Filter object in the filter stack.

javax.servlet
GenericServlet

Syntax

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

```
java.lang.Object
  |
  +--javax.servlet.GenericServlet
```

Direct Known Subclasses: `HttpServlet`

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

Description

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

Member Summary

Constructors	
`GenericServlet()`	Does nothing.
Methods	
`destroy()`	Called by the servlet container to indicate to a servlet that the servlet is being taken out of service.
`getInitParameter(String)`	Returns a `String` containing the value of the named initialization parameter, or `null` if the parameter does not exist.
`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.
`getServletConfig()`	Returns this servlet's `ServletConfig` object.
`getServletContext()`	Returns a reference to the `ServletContext` in which this servlet is running.
`getServletInfo()`	Returns information about the servlet, such as author, version, and copyright.
`getServletName()`	Returns the name of this servlet instance.
`init()`	A convenience method which can be overridden so that there's no need to call `super.init(config)`.
`init(ServletConfig)`	Called by the servlet container to indicate to a servlet that the servlet is being placed into service.
`log(String)`	Writes the specified message to a servlet log file, prepended by the servlet's name.

Member Summary	
log(String, Throwable) service(ServletRequest, ServletResponse)	Writes an explanatory message and a stack trace for a given `Throwable` exception to the servlet log file, prepended by the servlet's name. Called by the servlet container to allow the servlet to respond to a request.

Inherited Member Summary
Methods inherited from class java.lang.Object `clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructors

GenericServlet()

```
public GenericServlet()
```

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

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 `destroy()` .

Specified By: `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 `getInitParameter(String)` .

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

Specified By: `getInitParameter(String)` in interface `Config`

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 `getInitParameterNames()` .

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

Specified By: `getInitParameterNames()` in interface `Config`

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: `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 `getServletContext()` .

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

Specified By: `getServletContext()` in interface `Config`

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 `getServletInfo()` .

Specified By: `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 `getServletName()` .

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

Returns: the name of this servlet instance

---

init()

```
public void init()
```

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)
```

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

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

Specified By: `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 `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 `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)
```

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

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

Specified By: `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

javax.servlet
RequestDispatcher

Syntax
```
public interface RequestDispatcher
```

Description
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: `getRequestDispatcher(String)`, `getNamedDispatcher(String)`, `getRequestDispatcher(String)`

Member Summary	
Methods	
`forward(ServletRequest, ServletResponse)`	Forwards a request from a servlet to another resource (servlet, JSP file, or HTML file) on the server.
`include(ServletRequest, ServletResponse)`	Includes the content of a resource (servlet, JSP page, HTML file) in the response.

Methods

forward(ServletRequest, ServletResponse)

```
public void forward(ServletRequest request, ServletResponse response)
```

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)
```

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

javax.servlet
Servlet

Syntax
```
public interface Servlet
```

All Known Implementing Classes: `GenericServlet`

Description
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 `getServletConfig` 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`, `HttpServlet`

Member Summary	
Methods	
`destroy()`	Called by the servlet container to indicate to a servlet that the servlet is being taken out of service.
`getServletConfig()`	Returns a `ServletConfig` object, which contains initialization and startup parameters for this servlet.
`getServletInfo()`	Returns information about the servlet, such as author, version, and copyright.
`init(ServletConfig)`	Called by the servlet container to indicate to a servlet that the servlet is being placed into service.
`service(ServletRequest, ServletResponse)`	Called by the servlet container to allow the servlet to respond to a request.

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 `ServletConfig` 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)
```

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)
```

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 multithreaded programming in Java is available in the Java tutorial on multi-threaded programming.

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

javax.servlet
ServletConfig

Syntax
```
public interface ServletConfig extends Config
```

All Superinterfaces: Config

All Known Implementing Classes: GenericServlet

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

Member Summary	
Methods	
getServletName()	Returns the name of this servlet instance.

Inherited Member Summary
Methods inherited from interface Config
getServletContext(), getInitParameter(String), getInitParameterNames()

Methods

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

javax.servlet
ServletContext

Syntax
```
public interface ServletContext
```

Description
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: `getServletConfig()`, `getServletContext()`

Member Summary

Methods

`getAttribute(String)`	Returns the servlet container attribute with the given name, or `null` if there is no attribute by that name.
`getAttributeNames()`	Returns an `Enumeration` containing the attribute names available within this servlet context.
`getContext(String)`	Returns a `ServletContext` object that corresponds to a specified URL on the server.
`getInitParameter(String)`	Returns a `String` containing the value of the named context-wide initialization parameter, or `null` if the parameter does not exist.
`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.
`getMajorVersion()`	Returns the major version of the Java Servlet API that this servlet container supports.
`getMimeType(String)`	Returns the MIME type of the specified file, or `null` if the MIME type is not known.
`getMinorVersion()`	Returns the minor version of the Servlet API that this servlet container supports.
`getNamedDispatcher(String)`	Returns a `RequestDispatcher` object that acts as a wrapper for the named servlet.
`getRealPath(String)`	Returns a `String` containing the real path for a given virtual path.
`getRequestDispatcher(String)`	Returns a `RequestDispatcher` object that acts as a wrapper for the resource located at the given path.
`getResource(String)`	Returns a URL to the resource that is mapped to a specified path.
`getResourceAsStream(String)`	Returns the resource located at the named path as an `InputStream` object.
`getServerInfo()`	Returns the name and version of the servlet container on which the servlet is running.
`getServlet(String)`	

Member Summary	
getServletNames()	
getServlets()	
log(Exception, String)	
log(String)	Writes the specified message to a servlet log file, usually an event log.
log(String, Throwable)	Writes an explanatory message and a stack trace for a given `Throwable` exception to the servlet log file.
removeAttribute(String)	Removes the attribute with the given name from the servlet context.
setAttribute(String, Object)	Binds an object to a given attribute name in this servlet context.

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 absolute (beginning with "/") and is interpreted based on the server's document root.

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

Parameters:
`uripath` - a `String` specifying the absolute URL of a resource on the server

Returns: the `ServletContext` object that corresponds to the named URL

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: `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: `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 2.2 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 2.2 must have this method return the integer 2.

Returns: 2

---

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 `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)`, `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 virtual path "/index.html" has a real path of whatever file on the server's filesystem would be served by a request for "/index.html".

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)
```

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

---

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)
```

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`

---

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`.

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 remove the old attribute and bind the name to the new attribute.

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

javax.servlet
ServletContextAttributeEvent

Syntax

```
public class ServletContextAttributeEvent extends ServletContextEvent
```

```
java.lang.Object
  |
  +--java.util.EventObject
        |
        +--ServletContextEvent
              |
              +--javax.servlet.ServletContextAttributeEvent
```

All Implemented Interfaces: java.io.Serializable

Description

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: ServletContextAttributesListener

Member Summary	
Constructors	
ServletContextAttributeEvent(ServletContext, String, Object)	Construct a ServletContextAttributeEvent from the given context for the given attribute name and attribute value.
Methods	
getName()	Return the name of the attribute that changed on the ServletContext.
getValue()	Returns the value of the attribute being added removed or replaced.

Inherited Member Summary
Fields inherited from class java.util.EventObject
source
Methods inherited from class ServletContextEvent
getServletContext()
Methods inherited from class java.util.EventObject
getSource, toString

Inherited Member Summary
Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

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.

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 being 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.

javax.servlet
ServletContextAttributesListener

Syntax
```
public interface ServletContextAttributesListener extends java.util.EventListener
```

All Superinterfaces: java.util.EventListener

Description
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`

Member Summary	
Methods	
`attributeAdded(ServletContextAttributeEvent)`	Notification that a new attribute was added to the servlet context.
`attributeRemoved(ServletContextAttributeEvent)`	Notification that an existing attribute has been remved from the servlet context.
`attributeReplaced(ServletContextAttributeEvent)`	Notification that an attribute on the servlet context has been replaced.

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 added.

attributeReplaced(ServletContextAttributeEvent)

```
public void attributeReplaced(ServletContextAttributeEvent scab)
```

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

javax.servlet
ServletContextEvent

Syntax

```
public class ServletContextEvent extends java.util.EventObject
```

```
java.lang.Object
  |
  +--java.util.EventObject
        |
        +--javax.servlet.ServletContextEvent
```

Direct Known Subclasses: ServletContextAttributeEvent

All Implemented Interfaces: java.io.Serializable

Description

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

Since: v 2.3

See Also: ServletContextListener

Member Summary	
Constructors	
ServletContextEvent(ServletContext)	Construct a ServletContextEvent from the given context.
Methods	
getServletContext()	Return the ServletContext that changed.

Inherited Member Summary
Fields inherited from class java.util.EventObject
source
Methods inherited from class java.util.EventObject
getSource, toString
Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructors

ServletContextEvent(ServletContext)

```
public ServletContextEvent(ServletContext source)
```

Construct a ServletContextEvent from the given context.

Parameters:

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

Methods

---

getServletContext()

```
public ServletContext getServletContext()
```

Return the ServletContext that changed.

Returns: the ServletContext that sent the event.

javax.servlet
ServletContextListener

Syntax
```
public interface ServletContextListener extends java.util.EventListener
```

All Superinterfaces: java.util.EventListener

Description
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`

Member Summary	
Methods	
`contextDestroyed(ServletContextEvent)`	Notification that the servlet context is about to be shut down.
`contextInitialized(ServletContextEvent)`	Notification that the web application is ready to process requests.

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.

javax.servlet
ServletException

Syntax

```
public class ServletException extends java.lang.Exception
```

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--javax.servlet.ServletException
```

Direct Known Subclasses: UnavailableException

All Implemented Interfaces: java.io.Serializable

Description

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

Member Summary

Constructors	
ServletException()	Constructs a new servlet exception.
ServletException(String)	Constructs a new servlet exception with the specified message.
ServletException(String, Throwable)	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.
ServletException(Throwable)	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.
Methods	
getRootCause()	Returns the exception that caused this servlet exception.

Inherited Member Summary

Methods inherited from class java.lang.Throwable

fillInStackTrace, getLocalizedMessage, getMessage, printStackTrace, printStackTrace, printStackTrace, toString

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

ServletException()

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

Methods

getRootCause()

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

Returns the exception that caused this servlet exception.

Returns: the `Throwable` that caused this servlet exception

javax.servlet
ServletInputStream

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

java.lang.Object
  |
  +--java.io.InputStream
        |
        +--javax.servlet.ServletInputStream
```

Description
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 `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`

Member Summary	
Constructors	
`ServletInputStream()`	Does nothing, because this is an abstract class.
Methods	
`readLine(byte[], int, int)`	Reads the input stream, one line at a time.

Inherited Member Summary
Methods inherited from class java.io.InputStream
`available, close, mark, markSupported, read, read, read, reset, skip`
Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructors

ServletInputStream()

```
protected ServletInputStream()
```

Does nothing, because this is an abstract class.

Methods

readLine(byte[], int, int)

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

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

javax.servlet
ServletOutputStream

Syntax

```
public abstract class ServletOutputStream extends java.io.OutputStream
```

```
java.lang.Object
  |
  +--java.io.OutputStream
        |
        +--javax.servlet.ServletOutputStream
```

Description

Provides an output stream for sending binary data to the client. A `ServletOutputStream` object is normally retrieved via the `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`

Member Summary

Constructors

`ServletOutputStream()`	Does nothing, because this is an abstract class.

Methods

`print(boolean)`	Writes a `boolean` value to the client, with no carriage return-line feed (CRLF) character at the end.
`print(char)`	Writes a character to the client, with no carriage return-line feed (CRLF) at the end.
`print(double)`	Writes a `double` value to the client, with no carriage return-line feed (CRLF) at the end.
`print(float)`	Writes a `float` value to the client, with no carriage return-line feed (CRLF) at the end.
`print(int)`	Writes an int to the client, with no carriage return-line feed (CRLF) at the end.
`print(long)`	Writes a `long` value to the client, with no carriage return-line feed (CRLF) at the end.
`print(String)`	Writes a `String` to the client, without a carriage return-line feed (CRLF) character at the end.
`println()`	Writes a carriage return-line feed (CRLF) to the client.
`println(boolean)`	Writes a `boolean` value to the client, followed by a carriage return-line feed (CRLF).
`println(char)`	Writes a character to the client, followed by a carriage return-line feed (CRLF).
`println(double)`	Writes a `double` value to the client, followed by a carriage return-line feed (CRLF).
`println(float)`	Writes a `float` value to the client, followed by a carriage return-line feed (CRLF).
`println(int)`	Writes an int to the client, followed by a carriage return-line feed (CRLF) character.
`println(long)`	Writes a `long` value to the client, followed by a carriage return-line feed (CRLF).
`println(String)`	Writes a `String` to the client, followed by a carriage return-line feed (CRLF).

Inherited Member Summary

Methods inherited from class java.io.OutputStream
`close, flush, write, write, write`
Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructors

ServletOutputStream()

```
protected ServletOutputStream()
```

Does nothing, because this is an abstract class.

Methods

print(boolean)

```
public void print(boolean b)
```

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)
```

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)
```

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)
```

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)
```

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)
```

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)
```

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()
```

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)
```

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)
```

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)
```

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)
```

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)
```

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)
```

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)
```

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

javax.servlet
ServletRequest

Syntax
```
public interface ServletRequest
```

All Known Subinterfaces: `HttpServletRequest`

All Known Implementing Classes: `ServletRequestWrapper`

Description
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 `HttpServletRequest` .

See Also: `HttpServletRequest`

Member Summary

Methods

`getAttribute(String)`	Returns the value of the named attribute as an `Object`, or `null` if no attribute of the given name exists.
`getAttributeNames()`	Returns an `Enumeration` containing the names of the attributes available to this request.
`getCharacterEncoding()`	Returns the name of the character encoding used in the body of this request.
`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.
`getContentType()`	Returns the MIME type of the body of the request, or `null` if the type is not known.
`getInputStream()`	Retrieves the body of the request as binary data using a `ServletInputStream` .
`getLocale()`	Returns the preferred `Locale` that the client will accept content in, based on the Accept-Language header.
`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.
`getParameter(String)`	Returns the value of a request parameter as a `String`, or `null` if the parameter does not exist.
`getParameterMap()`	Returns a java.util.Map of the parameters of this request.
`getParameterNames()`	Returns an `Enumeration` of `String` objects containing the names of the parameters contained in this request.
`getParameterValues(String)`	Returns an array of `String` objects containing all of the values the given request parameter has, or `null` if the parameter does not exist.
`getProtocol()`	Returns the name and version of the protocol the request uses in the form *protocol/majorVersion.minorVersion*, for example, HTTP/1.1.
`getReader()`	Retrieves the body of the request as character data using a `BufferedReader`.

Member Summary	
getRealPath(String)	
getRemoteAddr()	Returns the Internet Protocol (IP) address of the client that sent the request.
getRemoteHost()	Returns the fully qualified name of the client that sent the request, or the IP address of the client if the name cannot be determined.
getRequestDispatcher(String)	Returns a RequestDispatcher object that acts as a wrapper for the resource located at the given path.
getScheme()	Returns the name of the scheme used to make this request, for example, http, https, or ftp.
getServerName()	Returns the host name of the server that received the request.
getServerPort()	Returns the port number on which this request was received.
isSecure()	Returns a boolean indicating whether this request was made using a secure channel, such as HTTPS.
removeAttribute(String)	Removes an attribute from this request.
setAttribute(String, Object)	Stores an attribute in this request.
setCharacterEncoding(String)	Overrides the name of the character encoding used in the body of this request.

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 -1 if the type is not known

getInputStream()

```
public ServletInputStream getInputStream()
```

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: a java.util.Map container parameter names as keys and parameter values as map values.

---

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()
```

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 `getReader()` 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 `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, or the IP address of the client if the name cannot be determined. 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 `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`, `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.

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)`

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.

javax.servlet
ServletRequestWrapper

Syntax

```
public class ServletRequestWrapper implements ServletRequest

java.lang.Object
  |
  +--javax.servlet.ServletRequestWrapper
```

Direct Known Subclasses: HttpServletRequestWrapper

All Implemented Interfaces: ServletRequest

Description

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

Member Summary

Constructors	
ServletRequestWrapper(ServletRequest)	Creates a ServletRequest adaptor wrapping the given request object.
Methods	
getAttribute(String)	The default behavior of this method is to call getAttribute(String name) on the wrapped request object.
getAttributeNames()	The default behavior of this method is to return getAttributeNames() on the wrapped request object.
getCharacterEncoding()	The default behavior of this method is to return getCharacterEncoding() on the wrapped request object.
getContentLength()	The default behavior of this method is to return getContentLength() on the wrapped request object.
getContentType()	The default behavior of this method is to return getContentType() on the wrapped request object.
getInputStream()	The default behavior of this method is to return getInputStream() on the wrapped request object.
getLocale()	The default behavior of this method is to return getLocale() on the wrapped request object.
getLocales()	The default behavior of this method is to return getLocales() on the wrapped request object.
getParameter(String)	The default behavior of this method is to return getParameter(String name) on the wrapped request object.

Member Summary	
getParameterMap()	The default behavior of this method is to return getParameterMap() on the wrapped request object.
getParameterNames()	The default behavior of this method is to return getParameterNames() on the wrapped request object.
getParameterValues(String)	The default behavior of this method is to return getParameterValues(String name) on the wrapped request object.
getProtocol()	The default behavior of this method is to return getProtocol() on the wrapped request object.
getReader()	The default behavior of this method is to return getReader() on the wrapped request object.
getRealPath(String)	The default behavior of this method is to return getRealPath(String path) on the wrapped request object.
getRemoteAddr()	The default behavior of this method is to return getRemoteAddr() on the wrapped request object.
getRemoteHost()	The default behavior of this method is to return getRemoteHost() on the wrapped request object.
getRequest()	Return the wrapped request object.
getRequestDispatcher(String)	The default behavior of this method is to return getRequestDispatcher(String path) on the wrapped request object.
getScheme()	The default behavior of this method is to return getScheme() on the wrapped request object.
getServerName()	The default behavior of this method is to return getServerName() on the wrapped request object.
getServerPort()	The default behavior of this method is to return getServerPort() on the wrapped request object.
isSecure()	The default behavior of this method is to return isSecure() on the wrapped request object.
removeAttribute(String)	The default behavior of this method is to call removeAttribute(String name) on the wrapped request object.
setAttribute(String, Object)	The default behavior of this method is to return setAttribute(String name, Object o) on the wrapped request object.
setCharacterEncoding(String)	The default behavior of this method is to set the character encoding on the wrapped request object.

Inherited Member Summary
Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

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

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: `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: `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: `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: `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: `getContentType()` in interface `ServletRequest`

---

getInputStream()

```
public ServletInputStream getInputStream()
```

The default behavior of this method is to return getInputStream() on the wrapped request object.

Specified By: `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: `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: `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: `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: `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: `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: `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: `getProtocol()` in interface `ServletRequest`

---

getReader()

```
public java.io.BufferedReader getReader()
```

The default behavior of this method is to return getReader() on the wrapped request object.

Specified By: `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: `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: `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: `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: `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: `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: `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: `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: `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: `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: `setAttribute(String, Object)` in interface `ServletRequest`

---

setCharacterEncoding(String)

```
public void setCharacterEncoding(java.lang.String enc)
```

The default behavior of this method is to set the character encoding on the wrapped request object.

Specified By: `setCharacterEncoding(String)` in interface `ServletRequest`

javax.servlet
ServletResponse

Syntax
```
public interface ServletResponse
```

All Known Subinterfaces: `HttpServletResponse`

All Known Implementing Classes: `ServletResponseWrapper`

Description
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 for more information on MIME. Protocols such as SMTP and HTTP define profiles of MIME, and those standards are still evolving.

See Also: `ServletOutputStream`

Member Summary

Methods

`flushBuffer()`	Forces any content in the buffer to be written to the client.
`getBufferSize()`	Returns the actual buffer size used for the response.
`getCharacterEncoding()`	Returns the name of the charset used for the MIME body sent in this response.
`getLocale()`	Returns the locale assigned to the response.
`getOutputStream()`	Returns a `ServletOutputStream` suitable for writing binary data in the response.
`getWriter()`	Returns a `PrintWriter` object that can send character text to the client.
`isCommitted()`	Returns a boolean indicating if the response has been committed.
`reset()`	Clears any data that exists in the buffer as well as the status code and headers.
`setBufferSize(int)`	Sets the preferred buffer size for the body of the response.
`setContentLength(int)`	Sets the length of the content body in the response In HTTP servlets, this method sets the HTTP Content-Length header.
`setContentType(String)`	Sets the content type of the response being sent to the client.

Member Summary	
setLocale(Locale)	Sets the locale of the response, setting the headers (including the Content-Type's charset) as appropriate.

Methods

flushBuffer()

```
public void flushBuffer()
```

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()
```

Returns a `ServletOutputStream` suitable for writing binary data in the response. The servlet container does not encode the binary data. 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()
```

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.

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()`

---

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()`

javax.servlet
ServletResponseWrapper

Syntax

```
public class ServletResponseWrapper implements ServletResponse

java.lang.Object
  |
  +--javax.servlet.ServletResponseWrapper
```

Direct Known Subclasses: HttpServletResponseWrapper

All Implemented Interfaces: ServletResponse

Description

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

Member Summary

Constructors	
ServletResponseWrapper(ServletResponse)	Creates a ServletResponse adaptor wrapping the given response object.
Methods	
flushBuffer()	The default behavior of this method is to call flushBuffer() on the wrapped response object.
getBufferSize()	The default behavior of this method is to return getBufferSize() on the wrapped response object.
getCharacterEncoding()	The default behavior of this method is to return getCharacterEncoding() on the wrapped response object.
getLocale()	The default behavior of this method is to return getLocale() on the wrapped response object.
getOutputStream()	The default behavior of this method is to return getOutputStream() on the wrapped response object.
getResponse()	Return the wrapped ServletResponse object.
getWriter()	The default behavior of this method is to return getWriter() on the wrapped response object.
isCommitted()	The default behavior of this method is to return isCommitted() on the wrapped response object.
reset()	The default behavior of this method is to call reset() on the wrapped response object.
setBufferSize(int)	The default behavior of this method is to call setBufferSize(int size) on the wrapped response object.

Member Summary	
setContentLength(int)	The default behavior of this method is to call setContentLength(int len) on the wrapped response object.
setContentType(String)	The default behavior of this method is to call setContentType(String type) on the wrapped response object.
setLocale(Locale)	The default behavior of this method is to call setLocale(Locale loc) on the wrapped response object.

Inherited Member Summary
Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

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.

Methods

flushBuffer()

```
public void flushBuffer()
```

The default behavior of this method is to call flushBuffer() on the wrapped response object.

Specified By: `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: `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: `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: `getLocale()` in interface `ServletResponse`

---

getOutputStream()

```
public ServletOutputStream getOutputStream()
```

The default behavior of this method is to return getOutputStream() on the wrapped response object.

Specified By: `getOutputStream()` in interface `ServletResponse`

Throws: `IOException`

---

getResponse()

```
public ServletResponse getResponse()
```

Return the wrapped ServletResponse object.

---

getWriter()

```
public java.io.PrintWriter getWriter()
```

The default behavior of this method is to return getWriter() on the wrapped response object.

Specified By: `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: `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: `reset()` 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: `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: `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: `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: `setLocale(Locale)` in interface `ServletResponse`

javax.servlet
SingleThreadModel

Syntax

```
public interface SingleThreadModel
```

Description

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.

If a servlet implements this interface, the servlet will be thread safe. However, 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.

javax.servlet
UnavailableException

Syntax

```
public class UnavailableException extends ServletException
```

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--ServletException
                    |
                    +--javax.servlet.UnavailableException
```

All Implemented Interfaces: java.io.Serializable

Description

Defines an exception that a servlet throws to indicate that it is permanently or temporarily unavailable.

When a servlet is permanently unavailable, something is wrong with the servlet, and it cannot handle requests until some action is taken. For example, the servlet might be configured incorrectly, or its state may be corrupted. A servlet should log both the error and the corrective action that is needed.

A servlet 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 for a period of time suggested by the servlet, rather than rejecting them until the servlet container restarts.

Member Summary

Constructors	
UnavailableException(int, Servlet, String)	
UnavailableException(Servlet, String)	Constructs a new exception with a descriptive message indicating that the servlet is permanently unavailable.
UnavailableException(String)	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.
UnavailableException(String, int)	
Methods	
getServlet()	
getUnavailableSeconds()	Returns the number of seconds the servlet expects to be temporarily unavailable.

Member Summary	
`isPermanent()`	Returns a `boolean` indicating whether the servlet is permanently unavailable.

Inherited Member Summary

Methods inherited from interface `ServletException`

`getRootCause()`

Methods inherited from class java.lang.Throwable

`fillInStackTrace, getLocalizedMessage, getMessage, printStackTrace, printStackTrace, printStackTrace, toString`

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

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

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()
```

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

Package
javax.servlet.http

Class Summary

Interfaces

HttpServletRequest	Extends the ServletRequest interface to provide request information for HTTP servlets.
HttpServletResponse	Extends the 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.
HttpSessionAttributesListener	This listener interface can be implemented in order to get notifications of changes made to sessions within this web application.
HttpSessionBindingListener	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.
HttpServletRequestWrapper	Provides a convenient implementation of the HttpServletRequest interface that can be subclassed by developers wishing to adapt the request to a Servlet.
HttpServletResponseWrapper	Provides a convenient implementation of the HttpServletResponse interface that can be subclassed by developers wishing to adapt the response from a Servlet.
HttpSessionBindingEvent	Either Sent to an object that implements HttpSessionBindingListener when it is bound or unbound from a session, or to a HttpSessionAttributesListener that has been configured in the deploymewnt 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	

javax.servlet.http
Cookie

Syntax

```
public class Cookie implements java.lang.Cloneable
```

```
java.lang.Object
  |
  +--javax.servlet.http.Cookie
```

All Implemented Interfaces: java.lang.Cloneable

Description

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 `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 `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.

Member Summary

Constructors	
`Cookie(String, String)`	Constructs a cookie with a specified name and value.
Methods	
`clone()`	Overrides the standard `java.lang.Object.clone` method to return a copy of this cookie.
`getComment()`	Returns the comment describing the purpose of this cookie, or `null` if the cookie has no comment.
`getDomain()`	Returns the domain name set for this cookie.
`getMaxAge()`	Returns the maximum age of the cookie, specified in seconds, By default, `-1` indicating the cookie will persist until browser shutdown.
`getName()`	Returns the name of the cookie.
`getPath()`	Returns the path on the server to which the browser returns this cookie.

Member Summary	
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.
getValue()	Returns the value of the cookie.
getVersion()	Returns the version of the protocol this cookie complies with.
setComment(String)	Specifies a comment that describes a cookie's purpose.
setDomain(String)	Specifies the domain within which this cookie should be presented.
setMaxAge(int)	Sets the maximum age of the cookie in seconds.
setPath(String)	Specifies a path for the cookie to which the client should return the cookie.
setSecure(boolean)	Indicates to the browser whether the cookie should only be sent using a secure protocol, such as HTTPS or SSL.
setValue(String)	Assigns a new value to a cookie after the cookie is created.
setVersion(int)	Sets the version of the cookie protocol this cookie complies with.

Inherited Member Summary
Methods inherited from class java.lang.Object
`equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

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)`

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 can use any standard protocol; otherwise, `false`

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()`

javax.servlet.http
HttpServlet

Syntax

```
public abstract class HttpServlet extends GenericServlet implements java.io.Serializable
```

```
java.lang.Object
  |
  +--GenericServlet
        |
        +--javax.servlet.http.HttpServlet
```

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

Description

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 for more information on handling multiple threads in a Java program.

Member Summary

Constructors	
HttpServlet()	Does nothing, because this is an abstract class.
Methods	
doDelete(HttpServletRequest, HttpServletResponse)	Called by the server (via the `service` method) to allow a servlet to handle a DELETE request.
doGet(HttpServletRequest, HttpServletResponse)	Called by the server (via the `service` method) to allow a servlet to handle a GET request.
doOptions(HttpServletRequest, HttpServletResponse)	Called by the server (via the `service` method) to allow a servlet to handle a OPTIONS request.

Member Summary	
doPost(HttpServletRequest, HttpServletResponse)	Called by the server (via the `service` method) to allow a servlet to handle a POST request.
doPut(HttpServletRequest, HttpServletResponse)	Called by the server (via the `service` method) to allow a servlet to handle a PUT request.
doTrace(HttpServletRequest, HttpServletResponse)	Called by the server (via the `service` method) to allow a servlet to handle a TRACE request.
getLastModified(HttpServletRequest)	Returns the time the `HttpServletRequest` object was last modified, in milliseconds since midnight January 1, 1970 GMT.
service(HttpServletRequest, HttpServletResponse)	Receives standard HTTP requests from the public `service` method and dispatches them to the `do`*XXX* methods defined in this class.
service(ServletRequest, ServletResponse)	Dispatches client requests to the protected `service` method.

Inherited Member Summary

Methods inherited from class GenericServlet

destroy(), getInitParameter(String), getInitParameterNames(), getServletConfig(), getServletContext(), getServletInfo(), init(ServletConfig), init(), log(String), log(String, Throwable), getServletName()

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructors

HttpServlet()

```
public HttpServlet()
```

Does nothing, because this is an abstract class.

Methods

doDelete(HttpServletRequest, HttpServletResponse)

```
protected void doDelete(HttpServletRequest req, HttpServletResponse resp)
```

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

`ServletException` - if the request for the DELETE cannot be handled

---

doGet(HttpServletRequest, HttpServletResponse)

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
```

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 `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

`ServletException` - if the request for the GET could not be handled

See Also: `setContentType(String)`

---

doOptions(HttpServletRequest, HttpServletResponse)

```
protected void doOptions(HttpServletRequest req, HttpServletResponse resp)
```

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

`ServletException` - if the request for the OPTIONS cannot be handled

---

doPost(HttpServletRequest, HttpServletResponse)

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
```

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 `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

`ServletException` - if the request for the POST could not be handled

See Also: `ServletOutputStream`, `setContentType(String)`

---

doPut(HttpServletRequest, HttpServletResponse)

```
protected void doPut(HttpServletRequest req, HttpServletResponse resp)
```

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 .

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

`ServletException` - if the request for the PUT cannot be handled

---

doTrace(HttpServletRequest, HttpServletResponse)

```
protected void doTrace(HttpServletRequest req, HttpServletResponse resp)
```

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

`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)
```

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 `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

`ServletException` - if the request for the TRACE cannot be handled

See Also: `service(ServletRequest, ServletResponse)`

---

service(ServletRequest, ServletResponse)

```
public void service(ServletRequest req, ServletResponse res)
```

Dispatches client requests to the protected `service` method. There's no need to override this method.

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

Overrides: `service(ServletRequest, ServletResponse)` in class `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

`ServletException` - if the request for the TRACE cannot be handled

See Also: `service(ServletRequest, ServletResponse)`

javax.servlet.http
HttpServletRequest

Syntax

```
public interface HttpServletRequest extends ServletRequest
```

All Superinterfaces: `ServletRequest`

All Known Implementing Classes: `HttpServletRequestWrapper`

Description

Extends the `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).

Member Summary

Methods

`getAuthType()`	Returns the name of the authentication scheme used to protect the servlet, for example, "BASIC" or "SSL," or `null` if the servlet was not protected.
`getContextPath()`	Returns the portion of the request URI that indicates the context of the request.
`getCookies()`	Returns an array containing all of the `Cookie` objects the client sent with this request.
`getDateHeader(String)`	Returns the value of the specified request header as a `long` value that represents a `Date` object.
`getHeader(String)`	Returns the value of the specified request header as a `String`.
`getHeaderNames()`	Returns an enumeration of all the header names this request contains.
`getHeaders(String)`	Returns all the values of the specified request header as an `Enumeration` of `String` objects.
`getIntHeader(String)`	Returns the value of the specified request header as an `int`.
`getMethod()`	Returns the name of the HTTP method with which this request was made, for example, GET, POST, or PUT.
`getPathInfo()`	Returns any extra path information associated with the URL the client sent when it made this request.
`getPathTranslated()`	Returns any extra path information after the servlet name but before the query string, and translates it to a real path.
`getQueryString()`	Returns the query string that is contained in the request URL after the path.
`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.
`getRequestedSessionId()`	Returns the session ID specified by the client.
`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.
`getRequestURL()`	Reconstructs the URL the client used to make the request.
`getServletPath()`	Returns the part of this request's URL that calls the servlet.
`getSession()`	Returns the current session associated with this request, or if the request does not have a session, creates one.

Member Summary	
getSession(boolean)	Returns the current `HttpSession` associated with this request or, if if there is no current session and `create` is true, returns a new session.
getUserPrincipal()	Returns a `java.security.Principal` object containing the name of the current authenticated user.
isRequestedSessionId-FromCookie()	Checks whether the requested session ID came in as a cookie.
isRequestedSessionId-FromUrl()	
isRequestedSessionId-FromURL()	Checks whether the requested session ID came in as part of the request URL.
isRequestedSessionId-Valid()	Checks whether the requested session ID is still valid.
isUserInRole(String)	Returns a boolean indicating whether the authenticated user is included in the specified logical "role".

Inherited Member Summary
Methods inherited from interface ServletRequest
getAttribute(String), getAttributeNames(), getCharacterEncoding(), setCharacterEncoding(String), getContentLength(), getContentType(), getInputStream(), getParameter(String), getParameterNames(), getParameterValues(String), getParameterMap(), getProtocol(), getScheme(), getServerName(), getServerPort(), getReader(), getRemoteAddr(), getRemoteHost(), setAttribute(String, Object), removeAttribute(String), getLocale(), getLocales(), isSecure(), getRequestDispatcher(String), getRealPath(String)

Methods

getAuthType()

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

Returns the name of the authentication scheme used to protect the servlet, for example, "BASIC" or "SSL," or `null` if the servlet was not protected.

Same as the value of the CGI variable AUTH_TYPE.

Returns: a `String` specifying the name of 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 "".

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: a `Enumeration` containing the values of the requested header, or `null` if the request does not have any headers of that name

---

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 `NumberFormatException`.

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` 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`.

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

---

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. 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 http://foo.bar/a.html HEAD /xyz?a=b HTTP/1.1/xyz

To reconstruct an URL with a scheme and host, use `getRequestURL(HttpServletRequest)` .

Returns: a `String` containing the part of the URL from the protocol name up to the query string

See Also: `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

---

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.

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

javax.servlet.http
HttpServletRequestWrapper

Syntax

```
public class HttpServletRequestWrapper extends ServletRequestWrapper implements
    HttpServletRequest

java.lang.Object
  |
  +--ServletRequestWrapper
        |
        +--javax.servlet.http.HttpServletRequestWrapper
```

All Implemented Interfaces: HttpServletRequest, ServletRequest

Description

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

Member Summary	
Constructors	
HttpServletRequest-Wrapper(HttpServle-tRequest)	Constructs a request object wrapping the given request.
Methods	
getAuthType()	The default behavior of this method is to return getAuthType() on the wrapped request object.
getContextPath()	The default behavior of this method is to return getContextPath() on the wrapped request object.
getCookies()	The default behavior of this method is to return getCookies() on the wrapped request object.
getDateHeader(String)	The default behavior of this method is to return getDateHeader(String name) on the wrapped request object.
getHeader(String)	The default behavior of this method is to return getHeader(String name) on the wrapped request object.
getHeaderNames()	The default behavior of this method is to return getHeaderNames() on the wrapped request object.
getHeaders(String)	The default behavior of this method is to return getHeaders(String name) on the wrapped request object.
getIntHeader(String)	The default behavior of this method is to return getIntHeader(String name) on the wrapped request object.

Member Summary

getMethod()	The default behavior of this method is to return getMethod() on the wrapped request object.
getPathInfo()	The default behavior of this method is to return getPathInfo() on the wrapped request object.
getPathTranslated()	The default behavior of this method is to return getPathTranslated() on the wrapped request object.
getQueryString()	The default behavior of this method is to return getQueryString() on the wrapped request object.
getRemoteUser()	The default behavior of this method is to return getRemoteUser() on the wrapped request object.
getRequestedSessionId()	The default behavior of this method is to return getRequestedSessionId() on the wrapped request object.
getRequestURI()	The default behavior of this method is to return getRequestURI() on the wrapped request object.
getRequestURL()	The default behavior of this method is to return getRequestURL() on the wrapped request object.
getServletPath()	The default behavior of this method is to return getServletPath() on the wrapped request object.
getSession()	The default behavior of this method is to return getSession() on the wrapped request object.
getSession(boolean)	The default behavior of this method is to return getSession(boolean create) on the wrapped request object.
getUserPrincipal()	The default behavior of this method is to return getUserPrincipal() on the wrapped request object.
isRequestedSessionIdFromCookie()	The default behavior of this method is to return isRequestedSessionIdFromCookie() on the wrapped request object.
isRequestedSessionIdFromUrl()	The default behavior of this method is to return isRequestedSessionIdFromUrl() on the wrapped request object.
isRequestedSessionIdFromURL()	The default behavior of this method is to return isRequestedSessionIdFromURL() on the wrapped request object.
isRequestedSessionIdValid()	The default behavior of this method is to return isRequestedSessionIdValid() on the wrapped request object.
isUserInRole(String)	The default behavior of this method is to return isUserInRole(String role) on the wrapped request object.

Inherited Member Summary

Methods inherited from class ServletRequestWrapper

getRequest(), getAttribute(String), getAttributeNames(), getCharacterEncoding(), setCharacterEncoding(String), getContentLength(), getContentType(), getInputStream(), getParameter(String), getParameterMap(), getParameterNames(), getParameterValues(String), getProtocol(), getScheme(), getServerName(), getServerPort(), getReader(), getRemoteAddr(), getRemoteHost(), setAttribute(String, Object), removeAttribute(String), getLocale(), getLocales(), isSecure(), getRequestDispatcher(String), getRealPath(String)

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Inherited Member Summary
Methods inherited from interface `ServletRequest` `getAttribute(String)`, `getAttributeNames()`, `getCharacterEncoding()`, `setCharacterEncoding(String)`, `getContentLength()`, `getContentType()`, `getInputStream()`, `getParameter(String)`, `getParameterNames()`, `getParameterValues(String)`, `getParameterMap()`, `getProtocol()`, `getScheme()`, `getServerName()`, `getServerPort()`, `getReader()`, `getRemoteAddr()`, `getRemoteHost()`, `setAttribute(String, Object)`, `removeAttribute(String)`, `getLocale()`, `getLocales()`, `isSecure()`, `getRequestDispatcher(String)`, `getRealPath(String)`

Constructors

HttpServletRequestWrapper(HttpServletRequest)

```
public HttpServletRequestWrapper(HttpServletRequest request)
```

Constructs a request object wrapping the given request.

Throws: `java.lang.IllegalArgumentException` - if the request is null

Methods

getAuthType()

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

The default behavior of this method is to return getAuthType() on the wrapped request object.

Specified By: `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: `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: `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: `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: `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: `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: `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: `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: `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: `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: `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: `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: `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: `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: `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: `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: `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: `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: `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: `getUserPrincipal()` in interface `HttpServletRequest`

---

isRequestedSessionIdFromCookie()

```
public boolean isRequestedSessionIdFromCookie()
```

The default behavior of this method is to return isRequestedSessionIdFromCookie() on the wrapped request object.

Specified By: `isRequestedSessionIdFromCookie()` in interface `HttpServletRequest`

---

isRequestedSessionIdFromUrl()

```
public boolean isRequestedSessionIdFromUrl()
```

The default behavior of this method is to return isRequestedSessionIdFromUrl() on the wrapped request object.

Specified By: `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: `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: `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: `isUserInRole(String)` in interface `HttpServletRequest`

javax.servlet.http
HttpServletResponse

Syntax
```
public interface HttpServletResponse extends ServletResponse
```

All Superinterfaces: `ServletResponse`

All Known Implementing Classes: `HttpServletResponseWrapper`

Description
Extends the `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: `ServletResponse`

Member Summary

Fields

`SC_ACCEPTED`	Status code (202) indicating that a request was accepted for processing, but was not completed.
`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`	Status code (400) indicating the request sent by the client was syntactically incorrect.
`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`	Status code (100) indicating the client can continue.
`SC_CREATED`	Status code (201) indicating the request succeeded and created a new resource on the server.
`SC_EXPECTATION_FAILED`	Status code (417) indicating that the server could not meet the expectation given in the Expect request header.
`SC_FORBIDDEN`	Status code (403) indicating the server understood the request but refused to fulfill it.
`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`	Status code (410) indicating that the resource is no longer available at the server and no forwarding address is known.
`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`	Status code (500) indicating an error inside the HTTP server which prevented it from fulfilling the request.
`SC_LENGTH_REQUIRED`	Status code (411) indicating that the request cannot be handled without a defined `Content-Length`.
`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`.

Member Summary

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	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	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	Status code (204) indicating that the request succeeded but that there was no new information to return.
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	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	Status code (404) indicating that the requested resource is not available.
SC_NOT_IMPLEMENTED	Status code (501) indicating the HTTP server does not support the functionality needed to fulfill the request.
SC_NOT_MODIFIED	Status code (304) indicating that a conditional GET operation found that the resource was available and not modified.
SC_OK	Status code (200) indicating the request succeeded normally.
SC_PARTIAL_CONTENT	Status code (206) indicating that the server has fulfilled the partial GET request for the resource.
SC_PAYMENT_REQUIRED	Status code (402) reserved for future use.
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	Status code (407) indicating that the client *MUST* first authenticate itself with the proxy.
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	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	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	Status code (416) indicating that the server cannot serve the requested byte range.
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	Status code (303) indicating that the response to the request can be found under a different URI.
SC_SERVICE_UNAVAILABLE	Status code (503) indicating that the HTTP server is temporarily overloaded, and unable to handle the request.
SC_SWITCHING_PROTOCOLS	Status code (101) indicating the server is switching protocols according to Upgrade header.
SC_UNAUTHORIZED	Status code (401) indicating that the request requires HTTP authentication.
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	Status code (305) indicating that the requested resource *MUST* be accessed through the proxy given by the `Location` field.

Methods

addCookie(Cookie)	Adds the specified cookie to the response.
addDateHeader(String, long)	Adds a response header with the given name and date-value.
addHeader(String, String)	Adds a response header with the given name and value.

Member Summary	
addIntHeader(String, int)	Adds a response header with the given name and integer value.
containsHeader(String)	Returns a boolean indicating whether the named response header has already been set.
encodeRedirectUrl(String)	
encodeRedirectURL(String)	Encodes the specified URL for use in the `sendRedirect` method or, if encoding is not needed, returns the URL unchanged.
encodeUrl(String)	
encodeURL(String)	Encodes the specified URL by including the session ID in it, or, if encoding is not needed, returns the URL unchanged.
sendError(int)	Sends an error response to the client using the specified status.
sendError(int, String)	Sends an error response to the client using the specified status code and descriptive message.
sendRedirect(String)	Sends a temporary redirect response to the client using the specified redirect location URL.
setDateHeader(String, long)	Sets a response header with the given name and date-value.
setHeader(String, String)	Sets a response header with the given name and value.
setIntHeader(String, int)	Sets a response header with the given name and integer value.
setStatus(int)	Sets the status code for this response.
setStatus(int, String)	

Inherited Member Summary
Methods inherited from interface ServletResponse
getCharacterEncoding(), getOutputStream(), getWriter(), setContentLength(int), setContentType(String), setBufferSize(int), getBufferSize(), flushBuffer(), isCommitted(), reset(), setLocale(Locale), getLocale()

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_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.

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)
```

Sends an error response to the client using the specified status. The server generally creates the response to look like a normal server error page.

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

---

sendError(int, String)

```
public void sendError(int sc, java.lang.String msg)
```

Sends an error response to the client using the specified status code and descriptive message. The server generally creates the response to look like a normal server error page.

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 before this method call

---

sendRedirect(String)

```
public void sendRedirect(java.lang.String location)
```

Sends a temporary redirect response to the client using the specified redirect location URL. This method can accept relative URLs; the servlet container will convert the relative URL to an absolute URL before sending the response to the client.

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, the `sendError` method should be used instead.

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

javax.servlet.http
HttpServletResponseWrapper

Syntax

```
public class HttpServletResponseWrapper extends ServletResponseWrapper implements
    HttpServletResponse

java.lang.Object
  |
  +--ServletResponseWrapper
        |
        +--javax.servlet.http.HttpServletResponseWrapper
```

All Implemented Interfaces: HttpServletResponse, ServletResponse

Description

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

Member Summary	
Constructors	
HttpServletResponse-Wrapper(HttpServlet-Response)	Constructs a response adaptor wrapping the given response.
Methods	
addCookie(Cookie)	The default behavior of this method is to call addCookie(Cookie cookie) on the wrapped response object.
addDate-Header(String, long)	The default behavior of this method is to call addDateHeader(String name, long date) on the wrapped response object.
addHeader(String, String)	The default behavior of this method is to return addHeader(String name, String value) on the wrapped response object.
addIntHeader(String, int)	The default behavior of this method is to call addIntHeader(String name, int value) on the wrapped response object.
contains-Header(String)	The default behavior of this method is to call containsHeader(String name) on the wrapped response object.
encodeRedirec-tUrl(String)	The default behavior of this method is to return encodeRedirectUrl(String url) on the wrapped response object.
encodeRedirec-tURL(String)	The default behavior of this method is to return encodeRedirectURL(String url) on the wrapped response object.
encodeUrl(String)	The default behavior of this method is to call encodeUrl(String url) on the wrapped response object.

Member Summary

encodeURL(String)	The default behavior of this method is to call encodeURL(String url) on the wrapped response object.
sendError(int)	The default behavior of this method is to call sendError(int sc) on the wrapped response object.
sendError(int, String)	The default behavior of this method is to call sendError(int sc, String msg) on the wrapped response object.
sendRedirect(String)	The default behavior of this method is to return sendRedirect(String location) on the wrapped response object.
setDateHeader(String, long)	The default behavior of this method is to call setDateHeader(String name, long date) on the wrapped response object.
setHeader(String, String)	The default behavior of this method is to return setHeader(String name, String value) on the wrapped response object.
setIntHeader(String, int)	The default behavior of this method is to call setIntHeader(String name, int value) on the wrapped response object.
setStatus(int)	The default behavior of this method is to call setStatus(int sc) on the wrapped response object.
setStatus(int, String)	The default behavior of this method is to call setStatus(int sc, String sm) on the wrapped response object.

Inherited Member Summary

Fields inherited from interface HttpServletResponse

SC_CONTINUE, SC_SWITCHING_PROTOCOLS, SC_OK, SC_CREATED, SC_ACCEPTED, SC_NON_AUTHORITATIVE_INFORMATION, SC_NO_CONTENT, SC_RESET_CONTENT, SC_PARTIAL_CONTENT, SC_MULTIPLE_CHOICES, SC_MOVED_PERMANENTLY, SC_MOVED_TEMPORARILY, SC_SEE_OTHER, SC_NOT_MODIFIED, SC_USE_PROXY, SC_BAD_REQUEST, SC_UNAUTHORIZED, SC_PAYMENT_REQUIRED, SC_FORBIDDEN, SC_NOT_FOUND, SC_METHOD_NOT_ALLOWED, SC_NOT_ACCEPTABLE, SC_PROXY_AUTHENTICATION_REQUIRED, SC_REQUEST_TIMEOUT, SC_CONFLICT, SC_GONE, SC_LENGTH_REQUIRED, SC_PRECONDITION_FAILED, SC_REQUEST_ENTITY_TOO_LARGE, SC_REQUEST_URI_TOO_LONG, SC_UNSUPPORTED_MEDIA_TYPE, SC_REQUESTED_RANGE_NOT_SATISFIABLE, SC_EXPECTATION_FAILED, SC_INTERNAL_SERVER_ERROR, SC_NOT_IMPLEMENTED, SC_BAD_GATEWAY, SC_SERVICE_UNAVAILABLE, SC_GATEWAY_TIMEOUT, SC_HTTP_VERSION_NOT_SUPPORTED

Methods inherited from class ServletResponseWrapper

getResponse(), getCharacterEncoding(), getOutputStream(), getWriter(), setContentLength(int), setContentType(String), setBufferSize(int), getBufferSize(), flushBuffer(), isCommitted(), reset(), setLocale(Locale), getLocale()

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Methods inherited from interface ServletResponse

getCharacterEncoding(), getOutputStream(), getWriter(), setContentLength(int), setContentType(String), setBufferSize(int), getBufferSize(), flushBuffer(), isCommitted(), reset(), setLocale(Locale), getLocale()

Constructors

HttpServletResponseWrapper(HttpServletResponse)

```
public HttpServletResponseWrapper(HttpServletResponse response)
```

Constructs a response adaptor wrapping the given response.

Throws: `java.lang.IllegalArgumentException` - if the response is null

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: `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: `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: `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: `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: `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: `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: `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: `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: `encodeURL(String)` in interface `HttpServletResponse`

---

sendError(int)

```
public void sendError(int sc)
```

The default behavior of this method is to call sendError(int sc) on the wrapped response object.

Specified By: `sendError(int)` in interface `HttpServletResponse`

Throws: `IOException`

---

sendError(int, String)

```
public void sendError(int sc, java.lang.String msg)
```

The default behavior of this method is to call sendError(int sc, String msg) on the wrapped response object.

Specified By: `sendError(int, String)` in interface `HttpServletResponse`

Throws: `IOException`

---

sendRedirect(String)

```
public void sendRedirect(java.lang.String location)
```

The default behavior of this method is to return sendRedirect(String location) on the wrapped response object.

Specified By: `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: `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: `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: `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: `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: `setStatus(int, String)` in interface `HttpServletResponse`

javax.servlet.http
HttpSession

Syntax
```
public interface HttpSession
```

Description
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.

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`

Member Summary

Methods

`getAttribute(String)`	Returns the object bound with the specified name in this session, or `null` if no object is bound under the name.
`getAttributeNames()`	Returns an `Enumeration` of `String` objects containing the names of all the objects bound to this session.
`getCreationTime()`	Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
`getId()`	Returns a string containing the unique identifier assigned to this session.
`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.
`getMaxInactiveInterval()`	Returns the maximum time interval, in seconds, that the servlet container will keep this session open between client accesses.
`getSessionContext()`	

Member Summary	
getValue(String)	
getValueNames()	
invalidate()	Invalidates this session and unbinds any objects bound to it.
isNew()	Returns `true` if the client does not yet know about the session or if the client chooses not to join the session.
putValue(String, Object)	
removeAttribute(String)	Removes the object bound with the specified name from this session.
removeValue(String)	
setAttribute(String, Object)	Binds an object to this session, using the name specified.
setMaxInactiveInterval(int)	Specifies the time, in seconds, between client requests before the servlet container will invalidate this session.

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.

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)`

---

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 and 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 `HttpSessionBindingListener`, the container calls `HttpSessionBindingListener.valueUnbound`.

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 `setAttribute(String, Object)`

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 object implements `HttpSessionBindingListener`, the container calls `HttpSessionBindingListener.valueBound`.

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

---

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

javax.servlet.http

HttpSessionAttributesListener

Syntax

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

All Superinterfaces: java.util.EventListener

Description

This listener interface can be implemented in order to get notifications of changes made to sessions within this web application.

Since: v 2.3

Member Summary	
Methods	
attributeAdded(HttpSessionBindingEvent)	Notification that an attribute has been added to a session.
attributeRemoved(HttpSessionBindingEvent)	Notification that an attribute has been removed from a session.
attributeReplaced(HttpSessionBindingEvent)	Notification that an attribute has been replaced in a session.

Methods

---

attributeAdded(HttpSessionBindingEvent)

```
public void attributeAdded(HttpSessionBindingEvent se)
```

Notification that an attribute has been added to a session.

---

attributeRemoved(HttpSessionBindingEvent)

```
public void attributeRemoved(HttpSessionBindingEvent se)
```

Notification that an attribute has been removed from a session.

---

attributeReplaced(HttpSessionBindingEvent)

```
public void attributeReplaced(HttpSessionBindingEvent se)
```

attributeReplaced(HttpSessionBindingEvent)

Notification that an attribute has been replaced in a session.

javax.servlet.http
HttpSessionBindingEvent

Syntax

```
public class HttpSessionBindingEvent extends HttpSessionEvent
```

```
java.lang.Object
  |
  +--java.util.EventObject
        |
        +--HttpSessionEvent
              |
              +--javax.servlet.http.HttpSessionBindingEvent
```

All Implemented Interfaces: java.io.Serializable

Description

Either Sent to an object that implements `HttpSessionBindingListener` when it is bound or unbound from a session, or to a `HttpSessionAttributesListener` that has been configured in the deploymewnt descriptor when any attribute is bound, unbound or replaced in a session.

Yhe session binds the object by a call to `HttpSession.putValue` and unbinds the object by a call to `HttpSession.removeValue`.

Since: v2.3

See Also: `HttpSession`, `HttpSessionBindingListener`, `HttpSessionAttributesListener`

Member Summary	
Constructors	
`HttpSessionBindingEvent(HttpSession, String)`	Constructs an event that notifies an object that it has been bound to or unbound from a session.
`HttpSessionBindingEvent(HttpSession, String, Object)`	Constructs an event that notifies an object that it has been bound to or unbound from a session.
Methods	
`getName()`	Returns the name with which the object is bound to or unbound from the session.
`getValue()`	Returns the value of the attribute being added, removed or replaced.

Inherited Member Summary
Fields inherited from class java.util.EventObject

Inherited Member Summary
`source`
Methods inherited from class `HttpSessionEvent`
`getSession()`
Methods inherited from class java.util.EventObject
`getSource, toString`
Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

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()`

Methods

getName()

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

Returns the name with which the object 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

---

getValue()

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

Returns the value of the attribute being 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.

javax.servlet.http
HttpSessionBindingListener

Syntax
```
public interface HttpSessionBindingListener extends java.util.EventListener
```

All Superinterfaces: java.util.EventListener

Description
Causes an object to be notified when it is bound to or unbound from a session. The object is notified by an `HttpSessionBindingEvent` object.

See Also: `HttpSession`, `HttpSessionBindingEvent`

Member Summary	
Methods	
`valueBound(HttpSessionBindingEvent)`	Notifies the object that it is being bound to a session and identifies the session.
`valueUnbound(HttpSessionBindingEvent)`	Notifies the object that it is being unbound from a session and identifies the session.

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)`

javax.servlet.http
HttpSessionContext

Syntax
```
public interface HttpSessionContext
```

Description

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`

Member Summary

Methods
`getIds()`
`getSession(String)`

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.

javax.servlet.http
HttpSessionEvent

Syntax

```
public class HttpSessionEvent extends java.util.EventObject
```

```
java.lang.Object
  |
  +--java.util.EventObject
        |
        +--javax.servlet.http.HttpSessionEvent
```

Direct Known Subclasses: HttpSessionBindingEvent

All Implemented Interfaces: java.io.Serializable

Description

This is the class representing event notifications for changes to sessions within a web application.

Since: v 2.3

Member Summary	
Constructors	
HttpSession-Event(HttpSession)	Construct a session event from the given source.
Methods	
getSession()	Return the session that changed.

Inherited Member Summary
Fields inherited from class java.util.EventObject
source
Methods inherited from class java.util.EventObject
getSource, toString
Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructors

HttpSessionEvent(HttpSession)

```
public HttpSessionEvent(HttpSession source)
```

Construct a session event from the given source.

Methods

getSession()

```
public HttpSession getSession()
```

Return the session that changed.

javax.servlet.http
HttpSessionListener

Syntax
```
public interface HttpSessionListener
```

Description
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`

Member Summary	
Methods	
`sessionCreated(HttpSessionEvent)`	Notification that a session was created.
`sessionDestroyed(HttpSessionEvent)`	Notification that a session was invalidated.

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

javax.servlet.http
HttpUtils

Syntax

```
public class HttpUtils

java.lang.Object
  |
  +--javax.servlet.http.HttpUtils
```

Description

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.

Member Summary	
Constructors	
`HttpUtils()`	Constructs an empty `HttpUtils` object.
Methods	
`getRequestURL(HttpServletRequest)`	Reconstructs the URL the client used to make the request, using information in the `HttpServletRequest` object.
`parsePostData(int, ServletInputStream)`	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.
`parseQueryString(String)`	Parses a query string passed from the client to the server and builds a `HashTable` object with key-value pairs.

Inherited Member Summary
Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructors

HttpUtils()

```
public HttpUtils()
```

Constructs an empty `HttpUtils` object.

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, 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

parseQueryString(String)