Java™ Servlet API

Version 2.2 Public Review Draft

James Duncan Davidson

We're the dot in .com™

June 28, 1999

Please send all comments to *servletapi-feedback@eng.sun.com*.
Comments must be submitted by July 28th, 1999

Java™ Servlet API Specification (“Specification”)

Version: 2.2
Status: Public Review Draft
Release: June 28, 1999

NOTICE
This Specification is protected by copyright and the information described herein 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 this Specification may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Any use of this Specification and the information described herein will be governed by these terms and conditions and the Export Control and General Terms as set forth in Sun’s website Legal Terms. By viewing, downloading or otherwise copying this Specification, you agree that you have read, understood, and will comply with all the terms and conditions set forth herein.

Subject to the terms and conditions of this license, Sun Microsystems, Inc. (“Sun”) hereby grants to you a fully-paid, nonexclusive, 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 this Specification or any other Sun intellectual property. This Specification contains the proprietary and confidential information of Sun and may only be used in accordance with the license terms set forth therein. 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 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 licensor is granted hereunder.

Sun, Sun Microsystems, the Sun logo, Java, JavaBeans, Enterprise JavaBeans, JavaServer, JavaServer Pages, JDK, JDBC, Java 2, The Network is the Computer, and Write Once, Run Anywhere are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

DISCLAIMER OF WARRANTIES
THIS 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; NOR 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 this Specification in any product(s).

THIS SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW VERSIONS OF THE SPECIFICATION, IF ANY. SUN MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then current terms and conditions 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 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
Use, duplication, or disclosure by the U.S. Government is subject to the restrictions set forth in this license and as provided in DFARS 227.7202-1(a) and 227.7202-3(a) (1995), DFARS 252.227-7013(c)(1)(ii)(Oct 1988), FAR 12.212(a) (1995), FAR 52.227-19 (June 1987), or FAR 52.227-14(ALT III) (June 1987), as applicable.

REPORT
As an Evaluation Posting of this Specification, 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 to Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license to incorporate, disclose, and use without limitation the Feedback for any purpose relating to the Specification and future versions, implementations, and test suites thereof.

Table Of Contents

Preface

This document, the Java™ Servlet Specification, describes Version 2.2 of the Java Servlet API. In addition to this specification, the Java Servlet API has Javadoc documentation and a reference implementation available for public download at the following location:

`http://java.sun.com/products/servlet/index.html`

The reference implementation provides a behavioral benchmark. In the case of a discrepancy, the order of resolution is the specification (this document), then the Javadoc documentation, and finally the reference implementation.

Reviewer Note: The javadoc and reference implementation will not be available until the next public release of the specification.

0.1 Who Should Read This Specification

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 Tools that want to generate Web Applications that conform to this specification
- Sophisticated Servlet authors.

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

0.2 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 1738 Uniform Resource Locators (URL)
- 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]

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

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

---

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.

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 following documents are located at the W3C website:

- HTTP/1.1 Draft 6
- Basic and Digest Authentication Draft 3

Both of these drafts have been accepted by the Internet Engineering Task Force (IETF) as Draft Standards.

0.3 Providing Feedback

The success of the Java Community Process depends on the participation of 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 and evaluated by the specification team.

0.4 Acknowledgements

Anselm Baird-Smith, Elias Bayeh, Vince Bonfanti, Robert Clark, Daniel Coward, Satish Dharmaraj, Jim Driscoll, Shel Finkelstein, Mark Hapner, Jason Hunter, Rod McChesney, Stefano Mazzocchi, Craig McClanahan, Adam Messinger, Vivek Nagar, Bob Pasker, Eduardo Pelegri-Lopart, Bill Shannon, Jon Stevens, James Todd, Spike Washburn, and Alan Williamson have all (in no particular order other than alphabetic) provided invaluable input into the evolution of this specification. Connie Weiss, Jeff Jackson, and Mala Chandra have provided incredible management assistance in supporting and furthering the Servlet effort at Sun.

The Servlet Specification is an ongoing and broad effort that includes contributions from numerous groups at Sun and at partner companies. Most notably the following companies and groups have gone the extra mile to help the Servlet Specification process: The Apache Developer Community, Art Technology Group, BEA Weblogic, IBM, Gefion Software, Live Software, Netscape Communications, New Atlanta Communications, and Oracle.

And of course, the ongoing specification review process has been extremely valuable. The many comments that we have received from both our partners and the general public have helped us to define and evolve the specification. Many thanks to all who have contributed feedback.

1 Overview

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. 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.

1.3 Client-Server Interaction

A client program, which could be a Web Browser or some other program that can make connections across a network such as the Internet or a corporate intranet, accesses a Web Server and makes an HTTP request. This request is processed by the servlet container that runs inside (or in conjunction with) the host Web Server which calls a specific servlet. This servlet in turn sends a response back to the client.

1.4 An Example

To help clarify the above points, here is an example:

A client program, accesses a web server and makes a 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 uses the request object to find out who the remote user is, what form parameters were sent as part of this request, and other relevant data. The servlet can then perform whatever logic is 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.

1.5 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 language, including ease of development and platform independence.
- They can access the large set of APIs available for the Java Platform.

1.6 API Status

1.6.1 Changes Since Version 2.1

The following major changes have been made to the specification since version 2.1:

- The introduction of the Web Application concept
- The introduction of the Web Application Archive
- Internationalization improvements
- Many clarifications of distributed servlet engine semantics

The following changes have been made to the API:

- Added the `getServletName` method to the `ServletConfig` interface to allow a servlet to obtain the name by which it is known to the system, if any.
- Added the localization methods `getResource(String name, Locale locale)` and `getResourceAsStream(String name, Locale locale)` to the `ServletContext` interface so that localized content can be acquired from the underlying server if available.
- Added the `getInitParameter` and `getInitParameterNames` method to the `ServletContext` interface so that initialization parameters can be set at the application level to be shared by all servlets that are part of that application.
- Added the `getLocale` method to the `ServletRequest` interface to aid in determining what locale the client is in.
- Added the `isSecure` method to the `ServletRequest` interface to indicate whether or not the request was transmitted via a secure transport such as HTTPS.
- Deprecated the construction methods of `UnavailableException` as the information is redundant to the container and this has caused some amount of developer confusion. These constructors have been replaced by simpler signatures.
- Added the `getHeaders` method to the `HttpServletRequest` interface to allow all the headers associated with a particular name to be retrieved from the request.
- Added the `getContextPath` method to the `HttpServletRequest` interface so that the part of the request path associated with a web application can be obtained.
- Added the `isRemoteUserInRole` and `getRemoteUserPrinciple` methods to the `HttpServletRequest` method to allow servlets to use an abstract role based authentication.
- Added the `addHeader`, `addIntHeader`, `and addDateHeader` methods to the `HttpServletResponse` interface to allow multiple headers to be created with the same header name.
- Added the `getAttribute`, `getAttributeNames`, `setAttribute`, and `removeAttribute` methods to the `HttpSession` interface to improve the naming conventions of the API. The `getValue`, `getValueNames`, `setValue`, and `removeValue` methods are deprecated as part of this change.

In addition, a large number of clarifications have been made to spec.

1.6.2 Changes from Version 2.0 to Version 2.1

Reviewer Note: Upcoming...

1.6.3 Changes from Version 1.0 to Version 2.0

Reviewer Note: Upcoming...

2 Terms Used

These terms are widely used throughout the rest of this specification.

2.1 Basic Terms

2.1.1 Uniform Resource Locators

Uniform Resource Locators (URLs) are 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] URLs are typically of the form:

```
<protocol>//<servername>/<resource>
```

For the purposes of this specification, we are primarily interested in HTTP based URLs which are of the form:

```
http[s]://<servername>[:port]/<url-path>[?<query-string>]
```

For example:

```
http://java.sun.com/products/servlet/index.html
https://javashop.sun.com/purchase
```

In HTTP based URLs, the `//` character is reserved for use to separate a hierarchical path structure in the url-path portion of the URL. The server is responsible to determining the meaning of the hierarchical structure. There is no correspondence between a url-path and a given file system path.

2.1.2 Servlet Definition

A Servlet Definition is a unique name associated with a fully qualified class name of a class implementing the `Servlet` interface as well as a set of initialization parameters.

2.1.3 Servlet Mapping

A Servlet Mapping is a Servlet Definition that is associated by a container with a URL path pattern. All requests to that path pattern are handled by the servlet associated with the Servlet Definition.

2.1.4 Web Application

A Web Application is a collection of servlets, JSP pages, HTML documents, and other resources. 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 “best effort” deployment of its contents into their runtime.

This specification does not mandate a specific runtime representation of all of the components of a web application, however it does recommend that container providers allow web applications to exist in the structure defined for deployment.

---

1. See RFC 1738

2.1.5 Web Application Archive

A Web Application Archive is a single file which 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.

2.2 Roles

The following roles are defined to aid in identifying the actions and responsibilities taken by various parties during the development, deployment, and running of a Servlet based application. In some scenarios, a single party may perform several roles, in others each role may be performed by a different party.

2.2.1 Developer

The Developer is the producer of a web based application. His or her output is a set of servlet classes, JSP pages, HTML pages, and supporting libraries and files for the web application. The 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.

The product of a developer may be a Web Application in archive form or in an open directory form.

2.2.2 Application Assembler

The application assembler takes the work done by the developer and ensures that it is a deployable unit. 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 in archive file form or as a set of files in a directory structure with all assembly information located in a deployment descriptor.

2.2.3 Deployer

The deployer takes one or more web application archive files, or other directory structures, provided by a 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.

The deployer is an expert in a specific operational environment. For example, the deployer is responsible for mapping the security roles defined by the developer to the user groups and accounts that exist in the operational environment where the web application is deployed.

2.2.4 System Administrator

The System Administrator is 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 and server vendors to accomplish these tasks.

2.2.5 Servlet Container Provider

The Servlet Container Provider is responsible for providing 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 vendor to split the implementation of the required functionality between the container and the server.

2.3 Security Terms

2.3.1 Principal

A principle 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 principle name and the authentication data depend on the authentication protocol.

2.3.2 Security Policy Domain

A security policy domain is a 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.*

2.3.3 Security Technology Domain

A security technology domain is 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.

3 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 delivered as part of the API that implement the `Servlet` interface are `GenericServlet` and `HttpServlet`. For most purposes, developers will typically extend `HttpServlet` to implement their servlets.

3.1 Request Handling Methods

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

3.1.1 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 and should not be used unless the programmer is very familiar with HTTP programming.

The `doPut` and `doDelete` methods allow developers to support HTTP/1.1 clients which support these features. The `doHead` method in `HttpServlet` is 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` and `doPost` methods will be used as HTTP/1.0 does not define the PUT, DELETE, HEAD, OPTIONS, or TRACE methods.

3.1.2 Conditional GET Support

The `HttpServlet` interface defines the `getLastModified` method to support conditional get operations. A conditional get operation is one which the client requests a resource with the 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.

3.2 Number of Instances

By default, there must be only one instance of a servlet class per servlet definition in a container. This default behavior recognizes common use of the Servlet API as well as allows for maximum efficiency and minimal memory usage by the host container.

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 without serializing requests to a single instance.

In the case where a servlet was deployed as part of an application that is marked 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, the container may instantiate multiple instances of that servlet in each VM.

3.2.1 Note about SingleThreadModel

The use of the `SingleThreadModel` interface only guarantees that one thread at a time will execute through a given servlet instance's `service` method. Use of this interface does not mean that all objects which are accessed in the `service` method, such as instances of `HttpSession` and `ServletContext` attributes, do not require possible synchronization by the application programmer when necessary.

3.3 Servlet Life Cycle

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

3.3.1 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 delayed until whenever 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. A given servlet can be instantiated more than once at a time -- i.e. there can be more than once instance of a given servlet in the servlet container. For example, this could occur where there was more than one servlet mapping that utilized a specific servlet class with different initialization parameters.

3.3.2 Initialization

After the servlet object is loaded and instantiated, the container must initialize the servlet before it can handle requests from clients. 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 section 4 titled “Servlet Context” on page 21 for more information about the `ServletContext` interface.

3.3.2.1 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`. 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 engine. The `destroy` method is not called in this case as initialization was not considered to be successful.

If the servlet throws an `UnavailableException` that indicates that it is unavailable for a limited amount of time, the servlet instance must be immediately released and the Servlet Engine may try to initialize a new instance of the servlet after the time period defined in the exception.

3.3.2.2 Why Initialize?

Initialization is provided so that a servlet can read any persistent configuration data, initialize costly resources (such as JDBC connections), and perform other one-time activities.

3.3.2.3 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. Servlet authors should not assume that a servlet has been loaded into an execution runtime environment due to the class being loaded and any static initialization methods being triggered.

3.3.3 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 the lifetime of the servlet instance.

3.3.3.1 Multithreading Issues

During the course of servicing requests from clients, a 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, they can program their 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 at a time. A 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 one of its `HttpServlet` descendants such as `doGet` or `doPost`) with the synchronized keyword, the 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 recom-

mended that servlet developers not synchronize the service method or any of its `HttpServlet` descendant methods such as `doGet`, `doPost`, etc.

3.3.3.2 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 container must remove the servlet from service, call its `destroy` method, and release the servlet instance. All further requests that the container maps to the servlet should be returned with a `SERVICE_UNAVAILABLE` (503) response status.

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.

3.3.4 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 server for a period of only milliseconds, for the lifetime of the server (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 if 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 dependent 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.

4 Servlet Context

The `ServletContext` defines a servlet's view of the Web Application within which the servlet is running. As well, it serves as the glue between a set of related servlets that have been grouped by a developer and deployed into a web server.

The servlet container is responsible for providing an implementation of the `ServletContext` interface. It is used to provide various items of information to a servlet and allow the servlet access to resources available to it through its Web Application. 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.

In cases where the servlet container is running in a clustered or distributed configuration, there must only be one instance of the `ServletContext` object per Application per Java Virtual Machine. It is important to note that a WebApplication may not be distributed across Virtual Machines unless the developer has indicated, through the deployment descriptor or some other action, that the application has been developed to run in a distributed environment.

4.1 Scope of a ServletContext

A `ServletContext` is associated directly (one-to-one) with a Web Application.

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.

4.2 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:

- `ServletContext.getInitParameter`
- `ServletContext.getInitParameterNames`

Initialization parameters can be used by an application developer to convey setup information, such as a webmaster's e-mail, a database reference, or even a reference to an Enterprise Java Bean (EJB)[1].

4.2.1 Initialization Parameters as JNDI References

When configured appropriately, such as through a web application's deployment descriptor or by other administrative intervention, the value of a context initialization parameter may be used by a servlet as the JNDI name to a resource factory or to an EJB. This allows an application developer to reference database resources, EJBs, and other resources in an abstract fashion.

This association is made in the deployment descriptor by creating an `ejb-ref` or `resource-ref` element entry with the same name as a `context-param` entry. For example, a deployment descriptor could contain the following information:

1. More information about Enterprise Java Beans can be found at http://java.sun.com/products/ejb

```
<context-param>
<param-name>primaryEJB</param-name>
<param-value>ejb/primary</param-value>
</context-param>
...
<ejb-ref>
<ejb-name>primaryEJB</ejb-name>
<ejb-type>com.widgets.ejb.PrimaryEjb</ejb-type>
<ejb-home>com.widgets.ejb.PrimaryEjbHome</ejb-home>
<ejb-remote>com.widgets.ejb.PrimaryEjbRemote</ejb-remote>
</ejb-ref>
```

See section 13 titled “Deployment Descriptor” on page 55 for more information about deployment descriptors.

4.3 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` allow access to this functionality:

- `ServletContext.setAttribute`
- `ServletContext.getAttribute`
- `ServletContext.getAttributeNames`
- `ServletContext.removeAttribute`

4.3.1 Context Attributes in a Distributed Container

If a Web Application is marked as distributed, attributes placed into a ServletContext must be made available to servlets in all Virtual Machines. When a developer marks a Web Application as distributable, they must ensure that all objects placed as attributes into the context must be either implement the `Serializable` or `Remote` interfaces. A container may throw an `IllegalArgumentException` if this contract is broken.

4.4 Resources

The `ServletContext` interface allows access to the document hierarchy of content documents, such as HTML, GIF, and JPEG files, via the following methods:

- `ServletContext.getResource`
- `ServletContext.getResourceAsStream`

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

4.4.1 Localization of Resources

Both the `getResource` and `getResourceAsStream` methods accept an optional `java.util.Locale` argument which will return a reference to the localized resource if available. If the localized resource is not available, these methods will return `null`.

4.5 Virtual Hosts and Servlet Contexts

If a servlet container supports the concept of virtual hosts, each virtual host must have its own Servlet Context or set of Servlet Contexts. A Servlet Context cannot be shared across virtual hosts.

4.6 Reloading Considerations

Many servlet container allow for servlet reloading for ease of development. Servlet reloading has typically been accomplished by creating new class loaders to load the servlet. This can have the undesirable side effect of causing references to not point at the active class or cause class casting exceptions.

All servlets (and code accessed from a servlet) in a servlet context must exist in such a way that there are no problems with references or class casting exceptions. A container can ensure this guarantee however the container developer sees fit.

5 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 either by HTTP headers or the message body of the request.

5.1 Parameters

Request parameters are Strings sent by the client to the web server 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 container as a set of name-value pairs. Multiple parameter values can exist for any given parameter name. The following methods are available to access parameters:

- `ServletRequest.getParameter`
- `ServletRequest.getParameterNames`
- `ServletRequest.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.

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.

5.2 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:

- `ServletRequest.getAttribute`
- `ServletRequest.getAttributeNames`
- `ServletRequest.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 Language Specification for package naming.

5.3 Headers

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

- HttpServletRequest.getHeader
- HttpServletRequest.getHeaders
- HttpServletRequest.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 provide access to header data in a one of these formats:

- HttpServletRequest.getIntHeader
- HttpServletRequest.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 IllegalArgumnetException is thrown.

5.4 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 Web Application that this servlet is a part of. If this application is the "default" application 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:** An 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:

- HttpServletRequest.getContextPath
- HttpServletRequest.getServletPath
- HttpServletRequest.getPathInfo

It is important to note that the following equation is always true:

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

In all cases a servlet developer must be able to access these portions of the path, add them together as shown, and obtain the same result as returned by ServletRequest.getRequestURI.

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:`

5.5 Path Translation Methods

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

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

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

In situations where the container cannot give a valid file path as a return to these methods, such as the resource being contained in a web application archive, these methods must return null.

5.6 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.

5.7 SSL Attributes

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

In containers that are running in a Java2 (JDK 1.2) environment, the SSL certificate must be exposed to the servlet programmer as an object implementing the `java.security.cert.X509Certificate` interface and accessible via a `ServletRequest` attribute of `javax.servlet.request.X509Certificate`.

For a container that is not running in a Java2 environment, vendors may provide vendor specific request attributes to access SSLcertificate information.

5.8 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:

- `ServletRequest.getLocale`
- `ServletRequest.getLocales`

The `getLocale` method will return the preferred locale that the client will accept content in. As clients will typically send a weighted list of acceptable languages, the `getLocales` method is provided which will return an `Enumeration` of `Locale` objects indicating, in decreasing order, the locales that are acceptable to the client.

If no preferred locale can be determined from the client, the locale returned by the `getLocale` method must be the default locale that the container is running in.

6 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.

6.1 Buffering

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

- `ServletResponse.getBufferSize`
- `ServletResponse.setBufferSize`
- `ServletResponse.isCommitted`
- `ServletResponse.clearBuffer`
- `ServletResponse.flushBuffer`

These methods are provided on the `ServletRequest` interface to allow buffering operations to be performed irregardless if 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'`.

The servlet can request a preferred buffer size for the response by using the `setBufferSize` method. The return value of this method is the buffer size assigned to this request by the container. 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 an 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 `clearBuffer` method clears any data that exists in the buffer as long as the response is not considered to be committed. If the response is committed and the `clearBuffer` method is called, an `IllegalStateException` must be thrown.

When buffering is being used, and the servlet fills up the buffer via the `ServletOutputStream` or a `Writer`, the buffer is immediately flushed to the client. If this is the first time for the request that data is sent to the client, the response is considered to be committed at this point.

6.2 Headers

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

- `HttpServletResponse.setHeader`
- `HttpServletResponse.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 associate 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 allow a servlet to set a header using the correct formatting for the appropriate data type:

- `HttpServletResponse.getIntHeader`
- `HttpServletResponse.getDateHeader`
- `HttpServletResponse.addIntHeader`
- `HttpServletResponse.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.

6.3 Convenience Methods

The following convenience methods exist in the HttpServletResponse interface:

- `HttpServletResponse.sendRedirect`
- `HttpServletResponse.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 partial URL path, however the underlying container must translate the partial path to a fully qualified URL for transmission back to the client.

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 normally have the side effect of committing and terminating the response to the client. No further output to the client should be made by the servlet after these methods are called. 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`.

6.4 Internationalization

In response to an 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.

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

7 Sessions

The Hypertext Transfer Protocol (HTTP) is by design a stateless protocol. To build effective web server applications, it is imperative that a series of unique requests from a remote 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.

The Servlet API provides a simple `HttpSession` interface that allows a servlet container to use any number of approaches to track a user's session without involving the application programmer in the nuances of any one approach.

7.1 Session Tracking Mechanisms

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.

7.1.2 Cookies

Session tracking through HTTP Cookies is the most used session tracking mechanism. 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.

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 to associate Session objects with.

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 thereby implying 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 are 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 the session by name. Any object bound into a session is available to any other servlet that handles a request from 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 methods that will signal an object being bound into, or being unbound from, a session.

7.5 Session Timeouts

Due to the fact that when an HTTP client stops making requests on a server, the server has no indication of the end of a session other than timeout.

All servlet containers must ensure that there is a default session timeout. A Web Application may specify a default timeout value as part of their deployment information.

7.6 Last Accessed Times

The session object has a method that allows a servlet to determine the last time it 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 server.

7.7 Important Session Semantics

7.7.1 Threading Issues

Multiple request threads may have active access to a session object at the same time. The servlet implementor has the responsibility to synchronize access to resources stored in the session as appropriate.

7.7.2 Distributed Environments

Unless a Web Application is marked as being “distributable”, all requests that are part of a session can only be executing on a single VM an any one time. In a distributed environment, all objects placed into the session must implement either the `Serializable` or `Remote` interfaces. The container may throw an `IllegalArgumentException` if this contract is broken.

7.7.3 Client Semantics

When session tracking is established via cookies or SSL certificates, requests from all windows of a client application on a container will be part of the same session. This has been a source of confusion in the past.

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 a servlet in the current 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:

- `ServletContext.getRequestDispatcher`
- `ServletContext.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. If a query string

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:

- `ServletRequest.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 operate.

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 `RequestDispatcher` may be obtained 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 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. It cannot set headers or call any method that affects the output of the response.

8.2.1 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.

8.3 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 exists in the response buffer that has not been committed, it is cleared before the target servlet is executed.

8.4 Error Handling

Only 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 be the original exception.

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 catalog application.

Rules can also be established by the container for automatic generation of web application based on whatever criteria. For example `~user/` mapping to a web app based at `/home/user/public_html`.

By default an instance of a Web Application must only be run 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 servlet programmer must obey a more restrictive set of rules than is expected of a normal servlet application. These rules are called out throughout this specification.

9.1 Relationship to ServletContext

There is 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 Members of a Web Application

A web application may consist of the following items:

- Servlets
- JavaServer Pages[1]
- JSP Beans and Tags
- 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 Web Servers 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`.

---

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

A special directory exists within the application hierarchy named "WEB-INF". This directory contains all things related to the application that aren't in the document root of the application. 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
- /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 usable to load classes from by the class loader of the web application.

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). 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 may be present which contains information useful to the Java Archive tools. If this directory is present, it must not be served by the container to web clients.

9.6 Internationalization of Web Application Content

Content within a web application can be localized in two different ways. The first way that web content can be localized is that the localized files are placed in the same location as the primary file and are named with additional information (following the pattern established by java.util.PropertyResourceBundle).

Table 3: Localized Filename Convention Example

Filename	
index.html	Default content
index_fr.html	Content Localized for France
index_de.html	Content Localized for Germany

The second mechanism that can be used is that secondary web application directories or archive files can be used.

For example, if there were French localized content provided for our issue tracking application above, this content could be provided in a web application archive named issuetrack_fr.war. If this archive file is located alongside the `issuetrack.war` file, localized content can be obtained from it.

If the archive files or directory structures cannot be located parallel to the master archive or directory, the location of the localized content hierarchy can be specified in the deployment descriptor using the `localized-content` element. See section 13 titled “Deployment Descriptor” on page 55 for more information about the deployment descriptor and its elements.

These secondary web application structures do not contain a `WEB-INF` directory. If a `WEB-INF` directory exists in a localized web application, it is ignored by the container.

9.7 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 section 13 titled “Deployment Descriptor” on page 55).

9.8 Replacing a Web Application

Applications evolve and must occasionally be replaced. In a long runner 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.

10 Mapping Requests to Servlets

Servlet containers in the past have had a great deal of flexibility in mapping client requests to servlets. In the previous version of this specification, we defined a set of suggested mapping techniques. In this version of the specification, we are now requiring a set of mapping techniques be used for web applications which are deployed via the Web Application Deployment mechanism. Just as it is highly recommended that containers use the deployment representations as their runtime representation, it is highly recommended that containers use these path mapping rules in their servers for all purposes and not just as party of deploying a web application.

10.1 Use of URL Paths

Servlet engines must use URL paths to map requests to servlets. The container uses the Request-URI from the request, minus the Context Path, as the path to map to a servlet. 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 engine will try to match a servlet that handles requests for the extension.
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.

10.2 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

10.2.1 Implicit Mappings

If the container has an internal JSP engine, 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 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.

10.2.2 Example Mapping Set

Consider the following set of mappings:

Table 4: Example Set of Maps

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

The following behavior would result:

Table 5: 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 meaning that the pattern can map to paths that start with `/catalog`.

11 Security

As web applications are created by a application 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.

11.1 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 methods by which communicating entities prove to one another that they are acting on behalf of specific identities.
- **Access control for resources:** The methods 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 methods used to prove that information could not have been modified by a third party while in transit.
- **Confidentiality or Data Privacy:** The methods used to ensure that the information is only made available to users who are authorized to access it and is not compromised during transmission.

11.2 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 application's logical security requirements are mapped by a deployer to a representation of the security policy that is specific to the runtime environment. At runtime, the container users the security policy that was derived from the deployment descriptor and configured by the deployer to enforce authentication.

11.3 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 two servlet security APIs:

- `HttpServletRequest.getRemoteUser`
- `HttpServletRequest.isRemoteUserInRole`
- `HttpServletRequest.getRemoteUserPrinciple`

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 principle name of the current user.

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

11.4 Roles

A role is an abstract logical grouping of users that is defined by the application developer or assembler. When the application is deployed, is mapped by a deployer to security identities, such as principals or groups) in the runtime environment.

Enforcement of either declarative or programmatic security depends on determining if the principal associated with an incoming request is in a given security role or not. A container makes this determination based on the security attributes of the calling principal. For example,

1. A deployer could have mapped a security role to a user group in the operational environment. In this case, 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. A deployer could have mapped a security role to a principal name in a security policy domain. In this case, the principal name of the calling principal is retrieved from its security attributes. If the principal is the same as the principal to which the security role was mapped, the calling principal is in the security role.

11.5 Authentication

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

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

11.5.1 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 *realm* in which the user is to be authenticated. The web client obtains the username and the password from the user an 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 (IPSEC or VPN) can alleviate some of these concerns.

11.5.2 Form Based Authentication

The look and feel of the "login screen" cannot be controlled with a HTTP browser's built in authentication mechanisms. Therefore a "Form Based Authentication" mechanism is defined by this specification for authentication of a user controlling to 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.

The mechanism depends on sessions.

When a user attempts to access a protected web resource, the container checks if the user has been authenticated. If so, and dependant 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.

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.

11.5.2.1 Login Form Notes

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 to avoid having to put the requirement that the server processes the outbound form to correct the action field.

```
<form method="POST" action="j_security_check">
<input type="text" name="J_USERNAME">
<input type="password" name="J_PASSWORD">
</form>
```

11.5.3 HTTPS Client Authentication

End user authentication using HTTPS (HTTP over SSL) is a strong authentication mechanism. This mechanism requires the user to posses a Public Key Certificate (PKC). Currently, PKCs are rarely used by end users on the internet. however, it is useful in e-commerce applications and also for single-signon from within the browser in the enterprise.

11.6 Web Single Signon

As the underlying security identities (such as users and groups) to which roles are mapped to in a runtime environment are not application specific, but environment 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.

User authentication information is therefore to be held at the container level and not the web application level. This allows a user to authenticate himself against one application and be able to access protected resources in another application in the same container as long as his authentication information is valid without having to re-authenticate.

11.7 Specifying Security Constraints

Security constraints are defined in the web application's deployment descriptor. Each constraint consists of the following:

- **Web Resource Collection:** A set of URL patterns and HTTP methods that describe a set of resources to be protected.
- **Authorization Constraints:** A set of roles that users accessing the resources described by the Web Resource Collection must be in.
- **User Data Constraints:** An optional parameter which can specify that the communication between client and server be guaranteed to be either integral or secure.

11.7.1 Default Policies

When specifying security constraints in the deployment descriptor, there are a few default policies that are implied.

By default, authentication is not needed to access resources via the HTTP `GET`, `POST`, `HEAD`, `OPTIONS`, or `TRACE` methods. Authentication is only needed when specified by the deployment descriptor. Authentication is always needed to access or modify resources using the HTTP `PUT` and `DELETE` methods.

12 Application Programming Interface

This is a listing of the interfaces, classes, and exceptions that compose the servlet api. For detailed descriptions of these members and their methods, please see the API documentation.

Items in bold face are new in this version of the specification.

Table 6: Servlet API Package Summary

Package javax.servlet	Package javax.serlvet.http
`RequestDispatcher`	`HttpServletRequest`
`Servlet`	`HttpServletResponse`
`ServletConfig`	`HttpSession`
`ServletContext`	`HttpSessionBindingListener`
`ServletRequest`	`HttpSessionContext`
`ServletResponse`	`Cookie`
`SingleThreadModel`	`HttpServlet`
`GenericServlet`	`HttpSessionBindingEvent`
`ServletInputStream`	`HttpUtils`
`ServletOutputStream`	
`ServletException`	
`UnavailableException`	

12.1 Package javax.servlet

12.1.1 RequestDispatcher

```
public interface RequestDispatcher

public void forward(ServletRequest req, ServletResponse res);
public void include(ServletRequest req, ServletResponse res);
```

12.1.2 Servlet

```
public interface Servlet

public void init(ServletConfig config) throws ServletException;
public ServletConfig getServletConfig();
public void service(ServletRequest req, ServletResponse res)
    throws IOException, ServletException;
public String getServletInfo();
public void destroy();
```

12.1.3 ServletConfig

```
public interface ServletConfig

public ServletContext getContext();
public String getInitParameter(String name);
public Enumeration getInitParameterNames();
public String getServletName();
```

12.1.4 ServletContext

```
public interface ServletContext

public String getMimeType(String filename);
public URL getResource(String path) throws MalformedURLException;
public URL getResource(String path, Locale locale)
    throws MalformedURLException;
public InputStream getResourceAsStream(String path);
public InputStream getResourceAsStream(String path, Locale loc);
public RequestDispatcher getRequestDispatcher(String path);
public RequestDispatcher getNamedDispatcher(String name);
public String getRealPath(String path);
public ServletContext getContext(String uripath);
public String getServerInfo();
public String getInitParameter(String name);
public Enumeration getInitParameterNames();
public Object getAttribute(String name);
public Enumeration getAttributeNames();
public void setAttribute(String name, Object attribute);
public void removeAttribute(String name);
public int getMajorVersion();
public int getMinorVersion();
public void log(String message);
public void log(String message, Throwable cause);

// deprecated methods
public Servlet getServlet(String name) throws ServletException;
public Enumeration getServlets();
public Enumeration getServletNames();
public void log(Exception exception, String message);
```

12.1.5 ServletRequest

```
public interface ServletRequest

public Object getAttribute(String name);
public Object setAttribute(String name, Object attribute);
public Enumeration getAttributeNames();
public Locale getLocale();
public Enumeration getLocales();
public String getCharacterEncoding();
public int getContentLength();
public String getContentType();
public ServletInputStream getInputStream() throws IOException;
```

```
public String getParameter(String name);
public String getParameterNames();
public String getParameterValues();
public String getProtocol();
public String getScheme();
public String getServerName();
public int getServerPort();
public BufferedReader getReader() throws IOException;
public String getRemoteAddr();
public String getRemoteHost();
public boolean isSecure();
public RequestDispatcher getRequestDispatcher(String path);

// deprecated methods
public String getRealPath();
```

12.1.6 ServletResponse

```
public interface ServletResponse

public String getCharacterEncoding();
public ServletOutputStream getOutputStream() throws IOException
public PrintWriter getWriter throws IOException
public void setContentLength(int length);
public void setContentType(String type);
public int setBufferSize(int size);
public int getBufferSize();
public void clearBuffer();
public boolean isCommitted();
public void flushBuffer();
public void setLocale();
```

12.1.7 SingleThreadModel

```
public interface SingleThreadModel

// no methods
```

12.1.8 GenericServlet

```
public abstract class GenericServlet implements Servlet

public GenericServlet();

public String getInitParameter();
public Enumeration getInitParameterNames();
public ServletConfig getServletConfig();
public ServletContext getServletContext();
public String getServletInfo();
public void init();
public void init(ServletConfig config) throws ServletException;
public void log(String message);
public void log(String message, Throwable cause);
public abstract void service(ServletRequest req,
```

```
    ServletResponse res) throws ServletException, IOException.;
public void destroy();
```

12.1.9 ServletInputStream

```
public abstract class ServletInputStream extends InputStream

public ServletInputStream();

public int readLine(byte[] buffer, int offset, int length)
    throws IOException;
```

12.1.10 ServletOutputStream

```
public abstract class ServletOutputStream extends OutputStream

public ServletOutputStream();

public void print(String s) throws IOException;
public void print(boolean b) throws IOException;
public void print(char c) throws IOException;
public void print(int i) throws IOException;
public void print(long l) throws IOException;
public void print(float f) throws IOException;
public void print(double d) throws IOException;
public void println() throws IOException;
public void println(String s) throws IOException;
public void println(boolean b) throwsIOException;
public void println(char c) throws IOException;
public void println(int i) throws IOException;
public void println(long l) throws IOException;
public void println(float f) throws IOException;
public void println(double d) throws IOException;
```

12.1.11 ServletException

```
public class ServletException extends Exception;

public ServletException();
public ServletException(String message);
public ServletException(String message, Throwable cause);
public ServletException(Throwable cause);

public Throwable getRootCause();
```

12.1.12 UnavailableException

```
public class UnavailableException extends ServletException

public UnavailableException(String message);
public UnavailableException(String message, int sec);

public int getUnavailableException();
public boolean isPermanent();
```

```
// newly deprecated methods
public UnavailableException(Servlet servlet, String message);
public UnavailableException(int sec, Servlet servlet, String msg);

public Servlet getServlet();
```

12.2 Package javax.servlet.http

```
interface HttpServletRequest
interface HttpServletResponse
interface HttpSession
interface HttpSessionBindingListener
interface HttpSessionContext

class Cookie
class HttpServlet
class HttpSessionBindingEvent
class HttpUtils
```

12.2.1 HttpServletRequest

```
public interface HttpServletRequest extends ServletRequest;

public String getAuthType();
public Cookie[] getCookies();
public long getDateHeader(String name);
public String getHeader(String name);
public Enumeration getHeaders(String name);
public Enumeration getHeaderNames();
public int getIntHeader(String name);
public String getMethod();
public String getContextPath();
public String getPathInfo();
public String getPathTranslated();
public String getQueryString();
public String getRemoteUser();
public boolean isRemoteUserInRole(String role);
public Principal getRemoteUserPrincipal();
public String getRequestedSessionId();
public boolean isRequestedSessionIdValid();
public boolean isRequestedSessionIdFromCookie();
public boolean isRequestedSessionIdFromURL();
public String getRequestURI();
public String getServletPath();
public HttpSession getSession();
public HttpSession getSession(boolean create);

// deprecated methods

public boolean isRequestSessionIdFromUrl();
```

12.2.2 HttpServletResponse

```
public interface HttpServletReponse extends ServletResponse

public static final int SC_CONTINUE;
public static final int SC_SWITCHING_PROTOCOLS;
public static final int SC_OK;
public static final int SC_CREATED;
public static final int SC_ACCEPTED;
public static final int SC_NON_AUTHORITATIVE_INFORMATION;
public static final int SC_NO_CONTENT;
public static final int SC_RESET_CONTENT;
public static final int SC_PARTIAL_CONTENT;
public static final int SC_MULTIPLE_CHOICES;
public static final int SC_MOVED_PERMANENTLY;
public static final int SC_MOVED_TEMPORARILY;
public static final int SC_SEE_OTHER;
public static final int SC_NOT_MODIFIED;
public static final int SC_USE_PROXY;
public static final int SC_BAD_REQUEST;
public static final int SC_UNAUTHORIZED;
public static final int SC_PAYMENT_REQUIRED;
public static final int SC_FORBIDDEN;
public static final int SC_NOT_FOUND;
public static final int SC_METH0D_NOT_ALLOWED;
public static final int SC_NOT_ACCEPTABLE;
public static final int SC_PROXY_AUTHENTICATION_REQUIRED;
public static final int SC_REQUEST_TIMEOUT;
public static final int SC_CONFLICT;
public static final int SC_GONE;
public static final int SC_LENGTH_REQUIRED;
public static final int SC_PRECONDITION_FAILED;
public static final int SC_REQUEST_ENTITY_TOO_LARGE;
public static final int SC_REQUEST_URI_TOO_LONG;
public static final int SC_UNSUPPORTED_MEDIA_TYPE;
public static final int SC_INTERNAL_SERVER_ERROR;
public static final int SC_NOT_IMPLEMENTED;
public static final int SC_BAD_GATEWAY;
public static final int SC_SERVICE_UNAVAILABLE;
public static final int SC_GATEWAY_TIMEOUT;
public static final int SC_VERSION_NOT_SUPPORTED;

public void addCookie(Cookie cookie);
public boolean containsHeader(String name);
public String encodeURL(String url);
public String encodeRedirectURL(String url);
public void sendError(int status) throws IOException;
public void sendError(int status, String message)
    throws IOException;
public void sendRedirect(String location) throws IOException;
public void setDateHeader(String headername, long date);
public void setHeader(String headername, String value);
```

```
public void addHeader(String headername, String value);
public void addDateHeader(String headername, long date);
public void addIntHeader(String headername, int value);
public void setIntHeader(String headername, int value);
public void setStatus(int statuscode);

// deprecated methods
public void setStatus(int statuscode, String message);
public String encodeUrl(String url);
public String encodeRedirectUrl(String url);
```

12.2.3 HttpSession

```
public interface HttpSession

public long getCreationTime();
public String getId();
public long getLastAccessedTime();
public boolean isNew();
public int getMaxInactiveInterval();
public void setMaxInactiveInterval();
public Object getAttribute(String name);
public Enumeration getAttributeNames();
public void setAttribute(String name, Object attribute);
public void removeAttribute(String name);
public void invalidate();

// deprecated methods
public Object getValue(String name);
public Enumeration getValueNames();
public void putValue(String name, Object value);
public void removeValue(String name);
public HttpSessionContext getSessionContext();
```

12.2.4 HttpSessionBindingListener

```
public interface HttpSessionBindingListener extends EventListener

public void valueBound(HttpSessionBindingEvent event);
public void valueUnbound(HttpSessionBindingEvent event);
```

12.2.5 HttpSessionContext

```
// deprecated
public abstract interface HttpSessionContext

// deprecated methods
public void Enumeration getIds();
public HttpSession getSession(String id);
```

12.2.6 Cookie

```
public class Cookie implements Cloneable
```

```
public Cookie(String name, String value);
public void setComment(String comment);
public String getComment();
public void setDomain(String domain);
public String getDomain();
public void setMaxAge(int expiry);
public int getMaxAge();
public void setPath(String uriPath);
public String getPath();
public void setSecure();
public boolean getSecure();
public String getName();
public void setValue(String value);
public String getValue();
public int getVersion();
public void setVersion(int version);
public Object clone();
```

12.2.7 HttpServlet

```
public abstract class HttpServlet extends GenericServlet
    implements Serializable

public HttpServlet();

protected void doGet(HttpServletRequest req,
    HttpServletResponse res) throws ServletException, IOException;
protected void doPost(HttpServletRequest req,
    HttpServletResponse res) throws ServletException, IOException;
protected void doPut(HttpServletRequest req,
    HttpServletResponse res) throws ServletException, IOException;
protected void doDelete(HttpServletRequest req,
    HttpServletResponse res) throws ServletException, IOException;
protected void doOptions(HttpServletRequest req,
    HttpServletResponse res) throws ServletException, IOException;
protected void doTrace(HttpServletRequest req,
    HttpServletResponse res) throws ServletException, IOException;
protected void service(HttpServletRequest req,
    HttpServletResponse res) throws ServletException, IOException;
public void service(ServletRequest req, ServletResponse res)
    throws ServletException, IOException;
protected long getLastModifed(HttpServletRequest req);
```

12.2.8 HttpSessionBindingEvent

```
public class HttpSessionBindingEvent extends EventObject

public HttpSessionBindingEvent(HttpSession session, String name);

public String getName();
public HttpSession getSession();
```

12.2.9 HttpUtils

```
public class HttpUtils

public HttpUtils();

public static Hashtable parseQueryString(String queryString);
public static Hashtable parsePostData(int length,
    ServletInputStream in);
public static StringBuffer getRequestURL(HttpServletRequest req);
```

13 Deployment Descriptor

13.1 Deployment Descriptor Elements

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

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

See the DTD comments for further description of these elements.

13.2 DTD

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

<!ELEMENT web-app (icon?, display-name, description?, distributable,
context-param*, servlet*, servlet-mapping*, session-config, localized-
content*, mime-mapping*, welcome-file*, error-page*, resource-ref*, web-
resource-collection*, security-role*, security-constraint*, ejb-ref*)>


<!-- The icon element contains a small-icon and a large-icon element
which specify the location within the web application for a small and
large image used to represent the web application in a GUI tool -->

<!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. The
image must be either GIF or JPEG format and the filename must end with
the extension of ".gif" or ".jpg" -->

<!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. The
image must be either GIF or JPEG format and the filename must end with
the extension of ".gif" or ".jpg" -->

<!ELEMENT large-icon (#PCDATA)>


<!-- The display-name element contains a short than 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. There are two
special kinds of initialization parameters for the optional cases where
the parameter is used to obtain the JNDI name of an EJB or a data source
for a subsequent look up. These are indicated by an context-param element
together with an accompanying ejb-ref or resource-ref element with
matching name -->

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


<!-- the name of a configuration parameter -->

<!ELEMENT param-name (#PCDATA)>


<!-- the value of a configuration parameter -->

<!ELEMENT param-value (#PCDATA>


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

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


<!-- The canonical name of the servlet. -->

<!ELEMENT servlet-name (#PCDATA)>


<!-- The class name of a servlet -->

<!ELEMENT servlet-class (#PCDATA)>


<!-- The path to a JSP file within the web application. -->

<!ELEMENT jsp-file (#PCDATA)>
```

```
<!-- The path to a serialized object file containing an object instance
of the servlet. -->

<!ELEMENT ser-file (#PCDATA)>


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

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


<!-- indicates that this servlet should be loaded on startup. The
optional contents of these element must be an integer indicating the
order in which the servlet should be loaded. -->

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


<!-- defines a mapping between a servlet and a url pattern -->

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


<!-- the url pattern of the mapping. Must follow the rules specified in
section 10 -->

<!ELEMENT url-pattern (#PCDATA)>


<!-- defines the session parameters for this web application. -->

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


<!-- defines the default session timeout interval for all sessions
created in this web application. The default units are measured in
minutes. -->

<!ELEMENT session-timeout (#PCDATA)>


<!-- defines where to find document bases for localized content -->

<!ELEMENT localized-content (locale, url)>


<!-- defines a locale -->

<!ELEMENT locale (#PCDATA)>


<!-- defines a url location to a resource external to this web
application -->

<!ELEMENT url (#PCDATA)>


<!-- defines a mapping between and extension and a mime type -->

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

```
<!-- an extension. example: ".txt" -->
<!ELEMENT extension (#PCDATA)>

<!-- a defined mime type. example: "text/plain" -->
<!ELEMENT mime-type (#PCDATA)>

<!-- And ordered list of welcome files. Both spaces and commas (or the
combination of them) serve as delimiters -->
<!ELEMENT welcome-files (#PCDATA)>

<!-- 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 HTTP error code, ex: 404 -->
<!ELEMENT error-code (#PCDATA)>

<!-- a Java exception type -->
<!ELEMENT exception-type (#PCDATA)>

<!-- The location of the resource in the web application -->
<!ELEMENT location (#PCDATA)>

<!-- The optional resource-ref element contains a declaration of a Web
Application's reference to an external resource. An context-param with
the same name must be present whose value will be the JNDI data source.
The context-param's value may be specified at deployment time. -->
<!ELEMENT resource-ref (description?, res-ref-name, res-type)>

<!-- Specifies the name of the resource factory reference name. This
value is the name of the context-param whose value contains the JNDI name
of the data source. -->
<!ELEMENT resource-ref-name (#PCDATA)>

<!-- Specifies the (Java class) type of the data source. -->
<!ELEMENT resource-type (#PCDATA)>

<!-- Used to associate security constraints with one or more web resource
collections -->
<!ELEMENT security-constraint (web-resource-collection+, auth-
constraint*, userdata-constraint*)>
```

```
<!-- Used to identify a subset of the resources and HTTP methods on those
resources within a web application to which a security constraint
applies. -->

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


<!-- The name of this web resource collection -->

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


<!-- An HTTP method (GET | POST |...) -->

<!ELEMENT http-method (#PCDATA)>


<!-- Used to indicate how data communicated between the client and
container should be protected -->

<!ELEMENT userdata-constraint (description?, transport-guarantee*)>


<!-- Specifies that the communication between client and server should be
SECURE or INTEGRAL -->

<!ELEMENT transport-guarantee (#PCDATA)>


<!-- indicates the user roles that should be permitted access to this
resource collection as well as the authorization method to be used. -->

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


<!-- defines a role name that the developer can use. the deployer maps a
set of users to the role name -->

<!ELEMENT role-name (#PCDATA)>


<!-- defines the authorization method to be used for the authorization
contstraint -->

<!ELEMENT auth-method (basic-auth|form-auth|mutual-auth)>


<!-- presence in the auth-method indicates that HTTP Basic authorization
should be used. The data is used to define the Realm string presented to
the use -->

<!ELEMENT basic-auth (#PCDATA)> // description of realm


<!-- presence in the auth-method indicates that Form based
authentications should be used -->

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

```
<!-- defines the location in the web app where the page that is used for
login can be found -->

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


<!-- 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)>


<!-- defines that SSL client certificate authentication should be used --
>

<!ELEMENT mutual-auth EMPTY>


<!-- used in conjunction with a matching context-param of the same name
to indicate that this parameter will be used to obtain the JNDI name of
an EJB. -->

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


<!-- contains the name of an EJB reference. This name must match an
context-param name -->

<!ELEMENT ejb-name (#PCDATA)>


<!-- contains the expected java class type of the referenced EJB. -->

<!ELEMENT ejb-type (#PCDATA)>


<!-- contains the fully qualified name of the EJB's home interface -->

<!ELEMENT ejb-home (#PCDATA)>


<!-- contains the fully qualified name of the EJB's remote interface -->

<!ELEMENT ejb-remote (#PCDATA)>


<!-- used in the ejb-ref element to specify that an EJB reference is
linked to an EJB in an encompassing 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)>
```

13.3 Examples

The following examples illustrate the usage of the above DTD.

13.3.1 A Basic Example

```
<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>
  <session-timeout>30</session-timeout>
  <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>
    <urlpattern>/catalog/*</urlpattern>
  </servlet-mapping>
  <mime-mapping>
    <extension>pdf</extension>
    <mime-type>application/pdf</mime-type>
  </mime-mapping>
  <welcome-files>index.jsp, index.html, index.htm</welcome-files>
  <error-page>
    <error-code>404</error-code>
    <location>/404.html</location>
  </error-page>
</web-app>
```

13.3.2 An Example of Security

```
<web-app>
  <display-name>A Security Example</display-name>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>salesinfo</web-resource-name>
      <urlpattern>/salesinfo/*</urlpattern>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
    </web-resource-collection>
    <userdata-constraint>
      <transport-guarantee>SECURE</transport-guarantee>
    </userdata-constraint>
    <auth-constraint>
      <role-name>manager</role-name>
      <role-name>executive</role-name>
      <role-name>sales</role-name>
    </auth-constraint>
  </security-constraint>
</web-app>
```

Please
Recycle

We're the d●t in .com™