Java™ Servlet Specification, v2.2

Final Release

microsystems

We're the dot in .com™

December 17th, 1999

James Duncan Davidson
Danny Coward

Please send all comments to *servletapi-feedback@eng.sun.com*.

Java™Servlet Specification ("Specification")
Version: 2.2
Status: Final Release
Release: 12/17/99

Table Of Contents

Preface

This document, the Java™ Servlet Specification, v2.2 the Java Servlet API. In addition to this specification, the Java Servlet API has Javadoc documentation (referred to as the Java Servlet API Reference, v2.2) 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 this specification, then the Java Servlet API Reference, v2.2, and finally the reference implementation.

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 Tool developers that want to generate Web Applications that conform to this specification
- Sophisticated Servlet authors who want to understand the underlying mechanisms of Servlet technology.

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

0.2 API Reference

The Java Servlet API Reference, v2.2 provides the complete description of all the interfaces, classes, exceptions, and methods that compose the Servlet API. Simplified method signatures are provided throughout the spec. Please refer to the API Reference for complete method signatures.

0.3 Other Java™ Platform Specifications

The following Java API Specifications are heavily referenced throughout this specification:

- Java2 Platform Enterprise Edition, v1.2 (J2EE)
- JavaServer Pages™, v1.1 (JSP)
- Java Naming and Directory Interface (JNDI)

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

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

0.4 Other Important References

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

- RFC 1630 Uniform Resource Identifiers (URI)
- RFC 1738 Uniform Resource Locators (URL)
- RFC 1808 Relative Uniform Resource Locators

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

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

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

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

The Extensible Markup Language (XML) is utilized by the Deployment Descriptors described in this specification. More information about XML can be found at the following websites:

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

0.5 Providing Feedback

The success of the Java Community Process depends on your participation in the community. We welcome any and all feedback about this specification. Please e-mail your comments to:

`servletapi-feedback@eng.sun.com`

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

0.6 Acknowledgements

Anselm Baird-Smith, Elias Bayeh, Vince Bonfanti, Larry Cable, Robert Clark, Daniel Coward, Satish Dharmaraj, Jim Driscoll, Shel Finkelstein, Mark Hapner, Jason Hunter, Rod McChesney, Stefano Mazzocchi, Craig McClanahan, Adam Messinger, Ron Monzillo, Vivek Nagar, Kevin Osborn, Bob Pasker, Eduardo Pelegri-Lopart, Harish Prabandham, Bill Shannon, Jon S. 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.

This 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, Clear Ink, IBM, Gefion Software, Live Software, Netscape Communications, New Atlanta Communications, and Oracle.

---

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.

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 into and run by a web server. Servlets interact with web clients via a request response paradigm implemented by the servlet container. This request-response model is based on the behavior of the Hypertext Transfer Protocol (HTTP).

1.2 What is a Servlet Container?

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

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

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

A Servlet Container may place security restrictions on the environment that a servlet executes in. In a Java 2 Platform Standard Edition 1.2 (J2SE) or Java 2 Platform Enterprise Edition 1.2 (J2EE) environment, these restrictions should be placed using the permission architecture defined by Java 2 Platform. For example, high end application servers may limit certain action, such as the creation of a `Thread` object, to insure that other components of the container are not negatively impacted.

1.3 An Example

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

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

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

1.4 Comparing Servlets with Other Technologies

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

Servlets have the following advantages over other server extension mechanisms:

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

1.5 Relationship to Java 2 Platform Enterprise Edition

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

1.6 Distributable Servlet Containers

New in this version of the specification is the ability to mark a web application as *distributable*. This indication allows servlet container vendors to deploy the servlets in a web application across multiple Java Virtual Machines running on the same host or on different hosts. An application marked as distributable must obey a few restrictions so that containers that support distributable applications can implement features such as clustering and failover.

All web applications that may need to run in a high performance environment, one that allows for scalability, clustering, and failover (such as a compliant J2EE implementation), should be written as distributable web applications. This will allow applications to take maximum advantage of servers that provide these features. If a non distributable application is deployed into such a server, then it cannot take full advantage of the features that are given by such servers.

1.7 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 files
- The introduction of response buffering
- The introduction of distributable servlets
- The ability to get a `RequestDispatcher` by name
- The ability to get a `RequestDispather` using a relative path
- 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.

---

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

- 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.
- Replaced the construction methods of `UnavailableException` as existing constructor signatures 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 `isUserInRole` and `getUserPrinciple` 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.

2 Terms Used

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

2.1 Basic Terms

2.1.1 Uniform Resource Locators

A Uniform Resource Locators (URL) is 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 a form of a Uniform Resource Identifier (URI). URLs are typically of the form:

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

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

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

For example:

```
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 for 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. A set of initialization parameters can be associated with a servlet definition.

2.1.3 Servlet Mapping

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

2.1.4 Web Application

A web application is a collection of servlets, JavaServer Pages[2], HTML documents, and other web resources which might include image files, compressed archives, and other data. A web application may be packaged into an archive or exist in an open directory structure.

All compatible servlet containers must accept a web application and perform a deployment of its contents into their runtime. This may mean that a container can run the application directly from a

---

1. See RFC 1738
2. See the JavaServer Pages specification at `http://java.sun.com/products/jsp`

web application archive file or it may mean that it will move the contents of a web application into the appropriate locations for that particular container.

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.

Web application archive files are identified by the `.war` extension. A new extension is used instead of `.jar` because that extension is reserved for files which contain a set of class files and that can be placed in the classpath or double clicked using a GUI to launch an application. As the contents of a web application archive are not suitable for such use, a new extension was in order.

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 Application Developer

The Application 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 (such as images, compressed archive files, etc.) for the web application. The Application Developer is typically an application domain expert. The developer is required to be aware of the servlet environment and its consequences when programming, including concurrency considerations, and create the web application accordingly.

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 Archive or a Web Application in an open directory structure.

2.2.3 Deployer

The Deployer takes one or more web application archive files or other directory structures provided by an Application Developer and deploys the application into a specific operational environment. The operational environment includes a specific servlet container and web server. The Deployer must resolve all the external dependencies declared by the developer. To perform his role, the deployer uses tools provided by the Servlet Container.

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

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 Provider 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 Container Provider to split the implementation of the required functionality between the container and the server.

2.3 Security Terms

2.3.1 Principal

A principal is an entity that can be authenticated by an authentication protocol. A principal is identified by a *principal name* and authenticated by using *authentication data*. The content and format of the principal name and the authentication data depend on the authentication protocol.

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.

2.3.4 Role

A role is an abstract notion used by a developer in an application that can be mapped by the deployer to a user, or group of users, in a security policy 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 in 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 servlet 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 for use by programmers very familiar with HTTP programming.

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

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

3.1.2 Conditional GET Support

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

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

3.2 Number of Instances

By default, there must be only one instance of a servlet class per servlet definition in a 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 while still serializing requests to a single instance.

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

3.2.1 Note about SingleThreadModel

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

3.3 Servlet Life Cycle

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

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 it can be delayed until the container determines that it needs the servlet to service a request.

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

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

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

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. Initialization is provided so that a servlet can read any persistent configuration data, initialize costly resources (such as JDBC™ based connection), and perform any other one-time activities. The container initializes the servlet by calling the `init` method of the `Servlet` interface with a unique (per servlet definition) object implementing the

`ServletConfig` interface. This configuration object allows the servlet to access name-value initialization parameters from the servlet container's configuration information. The configuration object also gives the servlet access to an object implementing the `ServletContext` interface which describes the runtime environment that the servlet is running within. See section 4 titled "Servlet Context" on page 23 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` or `ServletException`. If a servlet instance throws an exception of this type, it must not be placed into active service and the instance must be immediately released by the servlet container. The `destroy` method is not called in this case as initialization was not considered to be successful.

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

3.3.2.2 Tool Considerations

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

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 its lifetime.

3.3.3.1 Multithreading Issues

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

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

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

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 servlet container must remove the servlet from service, call its `destroy` method, and release the servlet instance.

If temporary unavailability is indicated by the `UnavailableException`, then the container may choose to not route any requests through the servlet during the time period of the temporary unavailability. Any requests refused by the container during this period must be returned with a `SERVICE_UNAVAILABLE` (503) response status along with a `Retry-After` header indicating when the unavailability will terminate. The container may choose to ignore the distinction between a permanent and temporary unavailability and treat all `UnavailableExceptions` as permanent, thereby removing a servlet that throws any `UnavailableException` from service.

3.3.3.3 Thread Safety

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

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 servlet container for a period of only milliseconds, for the lifetime of the servlet container (which could be measured in days, months, or years), or any amount of time in between.

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

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

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

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

4 Servlet Context

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

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

Only one instance of a `ServletContext` may be available to the servlets in a web application. In cases where the web application indicates that it is distributable, there must only be one instance of the `ServletContext` object in use per application per Java Virtual Machine.

4.1 Scope of a ServletContext

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

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

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:

- `getInitParameter`
- `getInitParameterNames`

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

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` interface allow access to this functionality:

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

4.3.1 Context Attributes in a Distributed Container

Context attributes exist locally to the VM in which they were created and placed. This prevents the `ServletContext` from being used as a distributed shared memory store. If information needs to be shared between servlets running in a distributed environment, that information should be placed into a session (See section 7 titled “Sessions” on page 35), a database or set in an Enterprise JavaBean.

4.4 Resources

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

- `getResource`
- `getResourceAsStream`

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

It is important to note that these methods give access to static resources from whatever repository the server uses. This hierarchy of documents may exist in a file system, in a web application archive file, on a remote server, or some other location. These methods are not used to obtain dynamic content. For example, in a container supporting the JavaServer Pages specification[1], a method call of the form `getResource("/index.jsp")` would return the JSP source code and not the processed output. See section 8 titled “Dispatching Requests” on page 39 for more information about accessing dynamic content.

4.5 Multiple Hosts and Servlet Contexts

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

4.6 Reloading Considerations

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

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

---

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

4.7 Temporary Working Directories

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

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

5.1 Parameters

Request parameters are strings sent by the client to a servlet container as part of a request. When the request is a `HttpServletRequest`, the attributes are populated from the URI query string and possibly posted form data. The parameters are stored by the servlet container as a set of name-value pairs. Multiple parameter values can exist for any given parameter name. The following methods of the `ServletRequest` interface are available to access parameters:

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

The `getParameterValues` method returns an array of `String` objects containing all the parameter values associated with a parameter name. The value returned from the `getParameter` method must always equal the first value in the array of `String` objects returned by `getParameterValues`.

All form data from both the query string and the post body are aggregated into the request parameter set. The order of this aggregation is that query string data takes precedence over post body parameter data. For example, if a request is made with a query string of `a=hello` and a post body of `a=goodbye&a=world`, the resulting parameter set would be ordered `a=(hello, goodbye, world)`.

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

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

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

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 of the `ServletRequest` interface:

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

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

Attribute names beginning with the prefixes of "`java.`" and "`javax.`" are reserved for definition by this specification. Similarly attribute names beginning with the prefixes of "`sun.`", and "`com.sun.`" are reserved for definition by Sun Microsystems. It is suggested that all attributes placed into the attribute set be named in accordance with the reverse package name convention suggested by the Java Programming Language Specification[1] for package naming.

5.3 Headers

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

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

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

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

- `getIntHeader`
- `getDateHeader`

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

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 `ServletContext` that this servlet is a part of. If this context is the "default" context rooted at the base of the web server's URL namespace, this path will be an empty string. Otherwise, this path starts with a`'/'` character but does not end with a`'/'` character.
- **Servlet Path:** The path section that directly corresponds to the mapping which activated this request. This path starts with a`'/'` character.
- **PathInfo:** The part of the request path that is not part of the Context Path or the Servlet Path.

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

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

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

---

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

`requestURI = contextPath + servletPath + pathInfo`

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

Table 1: Example Context Set Up

`ContextPath`	`/catalog`
`Servlet Mapping`	`Pattern: /lawn` `Servlet: LawnServlet`
`Servlet Mapping`	`Pattern: /garden` `Servlet: GardenServlet`
`Servlet Mapping`	`Pattern: *.jsp` `Servlet: JSPServlet`

The following behavior is observed:

Table 2: Observed Path Element Behavior

Request Path	Path Elements

`/catalog/lawn/index.html`	`ContextPath: /catalog`
	`ServletPath: /lawn`
	`PathInfo: /index.html`

`/catalog/garden/implements/`	`ContextPath: /catalog`
	`ServletPath: /garden`
	`PathInfo: /implements/`

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

5.5 Path Translation Methods

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

- `getRealPath`
- `getPathTranslated`

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

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

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 `isSecure` method of the `ServletRequest` interface.

In servlet containers that are running in a Java 2 Standard Edition, v 1.2 or Java 2 Enterprise Edition, v 1.2 environment, if there is an SSL certificate associated with the request, it must be exposed to the servlet programmer as an array of objects of type `java.security.cert.X509Certificate` and accessible via a `ServletRequest` attribute of `javax.servlet.request.X509Certificate`.

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

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:

- `getLocale`
- `getLocales`

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

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

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

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

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

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

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

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

The `isCommitted` method returns a boolean value indicating whether or not any bytes from the response have yet been returned to the client. The `flushBuffer` method forces any content in the buffer to be written to the client.

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

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

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

6.2 Headers

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

- `setHeader`

- `addHeader`

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

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

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

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

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

6.3 Convenience Methods

The following convenience methods exist in the `HttpServletResponse` interface:

- `sendRedirect`
- `sendError`

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

The `sendError` method will set the appropriate headers and content body to return to the client. An optional `String` argument can be provided to the `sendError` method which can be used in the content body of the error.

These methods will have the side effect of committing the response, if it had not already been committed, and terminating it. No further output to the client should be made by the servlet after these methods are called. If data is written to the response after these methods are called, the data is ignored.

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

6.4 Internationalization

In response to a request by a client to obtain a document of a particular language, or perhaps due to preference setting by a client, a servlet can set the language attributes of a response back to a client. This information is communicated via the `Content-Language` header along with other mechanisms described in the HTTP/1.1 specification. The language of a response can be set with the `setLocale` method of the `ServletResponse` interface. This method must correctly set the appropriate HTTP headers to accurately communicate the Locale to the client.

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

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

6.5 Closure of Response Object

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

- The termination of the service method of the servlet.
- When the amount of content specified in the `setContentLength` method of the response has been written to the response.
- The `sendError` method is called.
- The `sendRedirect` method is called.

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

7 Sessions

The Hypertext Transfer Protocol (HTTP) is by design a stateless protocol. To build effective web applications, it is imperative that a series of different requests from a particular client can be associated with each other. Many strategies for session tracking have evolved over time, but all are difficult or troublesome for the programmer to use directly.

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

7.1 Session Tracking Mechanisms

7.1.1 URL Rewriting

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

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

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

7.1.2 Cookies

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

7.1.3 SSL Sessions

Secure Sockets Layer, the encryption technology which is used in the HTTPS protocol, has a mechanism built into it allowing multiple requests from a client to be unambiguously identified as being part of an accepted session. A servlet container can easily use this data to serve as the mechanism for defining a session.

7.2 Creating a Session

Because HTTP is a request-response based protocol, a session is considered to be new until a client "joins" it. A client joins a session when session tracking information has been successfully returned to the server indicating that a session has been established. Until the client joins a session, it cannot be assumed that the next request from the client will be recognized as part of the session.

The session is considered to be "new" if either of the following is true:

- The client does not yet know about the session

- The client chooses not to join a session. This implies that the servlet container has no mechanism by which to associate a request with a previous request.

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

7.3 Session Scope

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

7.4 Binding Attributes into a Session

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

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

- `valueBound`
- `valueUnbound`

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

7.5 Session Timeouts

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

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

7.6 Last Accessed Times

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

7.7 Important Session Semantics

7.7.1 Threading Issues

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

7.7.2 Distributed Environments

Within an application that is marked as distributable, all requests that are part of a session can only be handled on a single VM at any one time. In addition all objects placed into instances of the `HttpSession` class using the `setAttribute` or `putValue` methods must implement the `Serializable` interface. The servlet container may throw an `IllegalArgumentException` if a non serializable object is placed into the session.

These restrictions mean that the Developer is ensured that there are no additional concurrency issues beyond those encountered in a non-distributed container. In addition, the Container Provider can ensure scalability by having the ability to move a session object, and its contents, from any active node of the distributed system to a different node of the system.

7.7.3 Client Semantics

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

8 Dispatching Requests

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

8.1 Obtaining a RequestDispatcher

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

- `getRequestDispatcher`
- `getNamedDispatcher`

The `getRequestDispatcher` method takes a `String` argument describing a path within the scope of the `ServletContext`. This path must be relative to the root of the `ServletContext`. This path is used to look up a servlet, wrap it with a `RequestDispatcher` object, and return it. If no servlet can be resolved based on the given path, a `RequestDispatcher` is provided that simply returns the content for that path.

The `getNamedDispatcher` method takes a `String` argument indicating the name of a servlet known to the `ServletContext`. If a servlet is known to the `ServletContext` by the given name, it is wrapped with a `RequestDispatcher` object and returned. If no servlet is associated with the given name, the method must return `null`.

To allow `RequestDispatcher` objects to be obtained using relative paths, paths which are not relative to the root of the `ServletContext` but instead are relative to the path of the current request, the following method is provided in the `ServletRequest` interface:

- `getRequestDispatcher`

The behavior of this method is similar to the method of the same name in the `ServletContext`, however it does not require a complete path within the context to be given as part of the argument to operate. The servlet container can use the information in the request object to transform the given relative path to a complete path. For example, in a context rooted at `'/'`, a request to `/garden/tools.html`, a request dispatcher obtained via `ServletRequest.getRequestDispatcher("header.html")` will behave exactly like a call to `ServletContext.getRequestDispatcher("/garden/header.html")`.

8.1.1 Query Strings in Request Dispatcher Paths

In the `ServletContext` and `ServletRequest` methods which allow the creation of a `RequestDispatcher` using path information, optional query string information may be attached to the path. For example, a Developer may obtain a `RequestDispatcher` by using the following code:

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

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

8.2 Using a Request Dispatcher

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

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

8.3 Include

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

8.3.1 Included Request Parameters

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

```
javax.servlet.include.request_uri
javax.servlet.include.context_path
javax.servlet.include.servlet_path
javax.servlet.include.path_info
javax.servlet.include.query_string
```

These attributes are accessible from the included servlet via the `getAttribute` method on the `request` object.

If the included servlet was obtained by using a `NamedDispatcher`, these attributes are not set.

8.4 Forward

The `forward` method of the `RequestDispatcher` interface may only be called by the calling servlet if no output has been committed to the client. If output exists in the response buffer that has not been committed, it must be reset (clearing the buffer) before the target servlet's `service` method is called. If the response has been committed, an `IllegalStateException` must be thrown.

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

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

8.5 Error Handling

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

9 Web Applications

A web application is a collection of servlets, html pages, classes, and other resources that can be bundled and run on multiple containers from multiple vendors. A web application is rooted at a specific path within a web server. For example, a catalog application could be located at `http://www.mycorp.com/catalog`. All requests that start with this prefix will be routed to the `ServletContext` which represents the catalog application.

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

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

9.1 Relationship to ServletContext

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

9.2 Elements of a Web Application

A web application may consist of the following items:

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

9.3 Distinction Between Representations

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

9.4 Directory Structure

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

---

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

9.4.1 Sample Web Application Directory Structure

Illustrated here is a listing of all the files in a sample web application:

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

9.5 Web Application Archive File

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

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

9.6 Web Application Configuration Descriptor

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

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

All of these types of information are conveyed in the deployment descriptor (See section 13 titled "Deployment Descriptor" on page 63).

9.7 Replacing a Web Application

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

9.8 Error Handling

A web application may specify that when errors occur, other resources in the application are used. These resources are specified in the deployment descriptor (See section 13 titled "Deployment Descriptor" on page 63). If the location of the error handler is a servlet or a JSP, the following request attributes can be set:

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

These attributes allow the servlet to generate specialized content depending on the status code, exception type and message of the error.

9.9 Web Application Environment

Java 2 Platform Enterprise Edition, v 1.2 defines a naming environment that allows applications to easily access resources and external information without the explicit knowledge of how the external information is named or organized.

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

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

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

The requirements of the J2EE environment with regards to setting up the environment are described in Chapter 5 of the Java 2 Platform Enterprise Edition v 1.2 specification[1]. Servlet containers that are not part of a J2EE compliant implementation are encouraged, but not required, to implement the application environment functionality described in the J2EE specification.

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

10 Mapping Requests to Servlets

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

10.1 Use of URL Paths

Servlet containers must use URL paths to map requests to servlets. The container uses the RequestURI from the request, minus the Context Path, as the path to map to a servlet. The URL path mapping rules are as follows (where the first match wins and no further rules are attempted):

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

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
- A string containing only the `'/'` character indicates that servlet specified by the mapping becomes the "default" servlet of the application.

10.2.1 Implicit Mappings

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

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

10.2.2 Example Mapping Set

Consider the following set of mappings:

Table 3: Example Set of Maps

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

The following behavior would result:

Table 4: Incoming Paths applied to Example Maps

incoming path	servlet handling request
`/foo/bar/index.html`	`servlet1`
`/foo/bar/index.bop`	`servlet1`
`/baz`	`servlet2`
`/baz/index.html`	`servlet2`
`/catalog`	`servlet3`
`/catalog/index.html`	`“default” servlet`
`/catalog/racecar.bop`	`servlet4`
`/index.bop`	`servlet4`

Note that in the case of `/catalog/index.html` and `/catalog/racecar.bop`, the servlet mapped to “`/catalog`” is not used as it is not an exact match and the rule doesn’t include the ’*’ character.

11 Security

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

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

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

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

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 the following methods of the `HttpServletRequest` interface:

- `getRemoteUser`
- `isUserInRole`

- `getUserPrincipal`

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

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

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

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, these roles are mapped by a Deployer to security identities, such as principals or groups, in the runtime environment.

A servlet container enforces declarative or programmatic security for the principal associated with an incoming request based on the security attributes of that calling principal. For example,

1. When a deployer has mapped a security role to a user group in the operational environment. The user group to which the calling principal belongs is retrieved from its security attributes. If the principal's user group matches the user group in the operational environment that the security role has been mapped to, the principal is in the security role.
2. When a deployer has mapped a security role to a principal name in a security policy domain, the principal name of the calling principal is retrieved from its security attributes. If the principal is the same as the principal to which the security role was mapped, the calling principal is in the security role.

11.5 Authentication

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

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

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 string called the *realm* of the request in which the user is to be authenticated. It is important to note that the realm string of the Basic Authentication mechanism does not have to reflect any particular security policy domain (which confusingly, can also be referred to as a realm). The web client obtains the username and the password from the user 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 (such as the IPSEC protocol or VPN strategies) can alleviate some of these concerns.

11.5.2 HTTP Digest Authentication

Like HTTP Basic Authentication, HTTP Digest Authentication authenticates a user based on a username and a password. However the authentication is performed by transmitting the password in an encrypted form which is much more secure than the simple base64 encoding used by Basic Authentication. This authentication method is not as secure as any private key scheme such as HTTPS Client Authentication. As Digest Authentication is not currently in widespread use, servlet containers are not required, but are encouraged, to support it.

11.5.3 Form Based Authentication

The look and feel of the "login screen" cannot be controlled with an HTTP browser's built in authentication mechanisms. Therefore this specification defines a form based authentication mechanism which allows a Developer to control the look and feel of the login screens.

The web application deployment descriptor contains entries for a login form and error page to be used with this mechanism. The login form must contain fields for the user to specify username and password. These fields must be named `'j_username'` and `'j_password'`, respectively.

When a user attempts to access a protected web resource, the container checks if the user has been authenticated. If so, and dependent on the user's authority to access the resource, the requested web resource is activated and returned. If the user is not authenticated, all of the following steps occur:

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

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

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

11.5.3.1 Login Form Notes

Form based login and URL based session tracking can be problematic to implement. It is strongly recommended that form based login only be used when the session is being maintained by cookies or by SSL session information.

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

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

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

11.5.4 HTTPS Client Authentication

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

11.6 Server Tracking of Authentication Information

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

1. Make login mechanisms and policies a property of the environment the web application is deployed in.
2. Be able to use the same authentication information to represent a principal to all applications that are deployed in the same container.
3. Require the user to re-authenticate only when crossing a security policy domain.

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

11.7 Specifying Security Constraints

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

- web resource collection
- authorization constraint
- user data constraint

A web resource collection is a set of URL patterns and HTTP methods that describe a set of resources to be protected. All requests that contain a request path that matches the URL pattern described in the web resource collection is subject to the constraint.

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

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

11.7.1 Default Policies

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

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 Java Servlet API Reference, v2.2.

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

Table 5: 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 getServletContext();
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 InputStream getResourceAsStream(String path);
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 void removeAttribute(String name);
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 void setBufferSize(int size);
public int getBufferSize();
public void reset();
public boolean isCommitted();
public void flushBuffer() throws IOException;
public void setLocale(Locale locale);
public Locale getLocale();
```

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 isUserInRole(String role);
public java.security.Principal getUserPrincipal();
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 HttpServletResponse extends ServletResponse
<<< STATUS CODES 416 AND 417 REPORTED MISSING>>>

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_REQUESTED_RANGE_NOT_SATISFIABLE;
public static final int SC_EXPECTATION_FAILED;
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(int interval);
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 String[] 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

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

13.1 Deployment Descriptor Elements

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

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

See the DTD comments for further description of these elements.

13.1.1 Deployment Descriptor DOCTYPE

All valid web application deployment descriptors must contain the following DOCTYPE declaration:

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

13.2 DTD

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

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

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

<!--
The icon element contains a small-icon and a large-icon element
which specify the location within the web application for a small and
large image used to represent the web application in a GUI tool. At a
minimum, tools must accept GIF and JPEG format images.
-->

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

```
<!--
The small-icon element contains the location within the web
application of a file containing a small (16x16 pixel) icon image.
-->

<!ELEMENT small-icon (#PCDATA)>

<!--
The large-icon element contains the location within the web
application of a file containing a large (32x32 pixel) icon image.
-->

<!ELEMENT large-icon (#PCDATA)>

<!--
The display-name element contains a short name that is intended
to be displayed by GUI tools
-->

<!ELEMENT display-name (#PCDATA)>

<!--
The description element is used to provide descriptive text about
the parent element.
-->

<!ELEMENT description (#PCDATA)>

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

<!ELEMENT distributable EMPTY>

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

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

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

<!ELEMENT param-name (#PCDATA)>

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

<!ELEMENT param-value (#PCDATA)>
```

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

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

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

<!ELEMENT servlet-name (#PCDATA)>

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

<!ELEMENT servlet-class (#PCDATA)>

<!--
The jsp-file element contains the full path to a JSP file within
the web application.
-->

<!ELEMENT jsp-file (#PCDATA)>

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

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

<!--
The load-on-startup element indicates that this servlet should be
loaded on the startup of the web application. The optional contents of
these element must be a positive integer indicating the order in which
the servlet should be loaded. Lower integers are loaded before higher
integers. If no value is specified, or if the value specified is not a
positive integer, the container is free to load it at any time in the
startup sequence.
-->

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

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

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

```
<!--
The url-pattern element contains the url pattern of the
mapping. Must follow the rules specified in Section 10 of the Servlet
API Specification.
-->

<!ELEMENT url-pattern (#PCDATA)>

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

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

<!--
The session-timeout element defines the default session timeout
interval for all sessions created in this web application. The
specified timeout must be expressed in a whole number of minutes.
-->

<!ELEMENT session-timeout (#PCDATA)>

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

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

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

<!ELEMENT extension (#PCDATA)>

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

<!ELEMENT mime-type (#PCDATA)>

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

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

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

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

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

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

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

<!ELEMENT taglib-uri (#PCDATA)>

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

<!ELEMENT taglib-location (#PCDATA)>

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

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

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

<!ELEMENT error-code (#PCDATA)>

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

<!ELEMENT exception-type (#PCDATA)>

<!--
The location element contains the location of the resource in the
web application
-->

<!ELEMENT location (#PCDATA)>

<!--
The resource-ref element contains a declaration of a Web
Application's reference to an external resource.
-->

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

```
<!--
The res-ref-name element specifies the name of the resource
factory reference name.
-->

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

<!--
The res-type element specifies the (Java class) type of the data
source.
-->

<!ELEMENT res-type (#PCDATA)>

<!--
The res-auth element indicates whether the application component
code performs resource signon programmatically or whether the
container signs onto the resource based on the principle mapping
information supplied by the deployer. Must be CONTAINER or SERVLET
-->

<!ELEMENT res-auth (#PCDATA)>

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

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

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

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

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

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

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

<!ELEMENT http-method (#PCDATA)>

<!--
The user-data-constraint element is used to indicate how data
```

```
communicated between the client and container should be protected
-->

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

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

<!ELEMENT transport-guarantee (#PCDATA)>

<!--
The auth-constraint element indicates the user roles that should
be permitted access to this resource collection. The role used here
must appear in a security-role-ref element.
-->

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

<!--
The role-name element contains the name of a security role.
-->

<!ELEMENT role-name (#PCDATA)>

<!--
The login-config element is used to configure the authentication
method that should be used, the realm name that should be used for
this application, and the attributes that are needed by the form login
mechanism.
-->

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

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

<!ELEMENT realm-name (#PCDATA)>

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

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

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

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

<!--
The form-error-page element defines the location in the web app
where the error page that is displayed when login is not successful
can be found
-->

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

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

<!ELEMENT auth-method (#PCDATA)>

<!--
The security-role element contains the declaration of a security
role which is used in the security-constraints placed on the web
application.
-->

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

<!--
The role-name element contains the name of a role. This element
must contain a non-empty string.
-->

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

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

<!ELEMENT role-link (#PCDATA)>

<!--
The env-entry element contains the declaration of an
application's environment entry. This element is required to be
honored on in J2EE compliant servlet containers.
-->
```

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

<!--
The env-entry-name contains the name of an application's
environment entry
-->

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

<!--
The env-entry-value element contains the value of an
application's environment entry
-->

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

<!--
The env-entry-type element contains the fully qualified Java type
of the environment entry value that is expected by the application
code. The following are the legal values of env-entry-type:
java.lang.Boolean, java.lang.String, java.lang.Integer,
java.lang.Double, java.lang.Float.
-->

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

<!--
The ejb-ref element is used to declare a reference to an
enterprise bean.
-->

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

<!--
The ejb-ref-name element contains the name of an EJB
reference. This is the JNDI name that the servlet code uses to get a
reference to the enterprise bean.
-->

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

<!--
The ejb-ref-type element contains the expected java class type of
the referenced EJB.
-->

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

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

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

```
<!--
The ejb-remote element contains the fully qualified name of the
EJB's remote interface
-->

<!ELEMENT remote (#PCDATA)>

<!--
The ejb-link element is used in the ejb-ref element to specify
that an EJB reference is linked to an EJB in an encompassing Java2
Enterprise Edition (J2EE) application package. The value of the
ejb-link element must be the ejb-name of and EJB in the J2EE
application package.
-->

<!ELEMENT ejb-link (#PCDATA)>

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

<!ATTLIST web-app id ID #IMPLIED>
<!ATTLIST icon id ID #IMPLIED>
<!ATTLIST small-icon id ID #IMPLIED>
<!ATTLIST large-icon id ID #IMPLIED>
<!ATTLIST display-name id ID #IMPLIED>
<!ATTLIST description id ID #IMPLIED>
<!ATTLIST distributable id ID #IMPLIED>
<!ATTLIST context-param id ID #IMPLIED>
<!ATTLIST param-name id ID #IMPLIED>
<!ATTLIST param-value id ID #IMPLIED>
<!ATTLIST servlet id ID #IMPLIED>
<!ATTLIST servlet-name id ID #IMPLIED>
<!ATTLIST servlet-class id ID #IMPLIED>
<!ATTLIST jsp-file id ID #IMPLIED>
<!ATTLIST init-param id ID #IMPLIED>
<!ATTLIST load-on-startup id ID #IMPLIED>
<!ATTLIST servlet-mapping id ID #IMPLIED>
<!ATTLIST url-pattern id ID #IMPLIED>
<!ATTLIST session-config id ID #IMPLIED>
<!ATTLIST session-timeout id ID #IMPLIED>
<!ATTLIST mime-mapping id ID #IMPLIED>
<!ATTLIST extension id ID #IMPLIED>
<!ATTLIST mime-type id ID #IMPLIED>
<!ATTLIST welcome-file-list id ID #IMPLIED>
<!ATTLIST welcome-file id ID #IMPLIED>
<!ATTLIST taglib id ID #IMPLIED>
<!ATTLIST taglib-uri id ID #IMPLIED>
<!ATTLIST taglib-location id ID #IMPLIED>
<!ATTLIST error-page id ID #IMPLIED>
```

```
<!ATTLIST error-code id ID #IMPLIED>
<!ATTLIST exception-type id ID #IMPLIED>
<!ATTLIST location id ID #IMPLIED>
<!ATTLIST resource-ref id ID #IMPLIED>
<!ATTLIST res-ref-name id ID #IMPLIED>
<!ATTLIST res-type id ID #IMPLIED>
<!ATTLIST res-auth id ID #IMPLIED>
<!ATTLIST security-constraint id ID #IMPLIED>
<!ATTLIST web-resource-collection id ID #IMPLIED>
<!ATTLIST web-resource-name id ID #IMPLIED>
<!ATTLIST http-method id ID #IMPLIED>
<!ATTLIST user-data-constraint id ID #IMPLIED>
<!ATTLIST transport-guarantee id ID #IMPLIED>
<!ATTLIST auth-constraint id ID #IMPLIED>
<!ATTLIST role-name id ID #IMPLIED>
<!ATTLIST login-config id ID #IMPLIED>
<!ATTLIST realm-name id ID #IMPLIED>
<!ATTLIST form-login-config id ID #IMPLIED>
<!ATTLIST form-login-page id ID #IMPLIED>
<!ATTLIST form-error-page id ID #IMPLIED>
<!ATTLIST auth-method id ID #IMPLIED>
<!ATTLIST security-role id ID #IMPLIED>
<!ATTLIST security-role-ref id ID #IMPLIED>
<!ATTLIST role-link id ID #IMPLIED>
<!ATTLIST env-entry id ID #IMPLIED>
<!ATTLIST env-entry-name id ID #IMPLIED>
<!ATTLIST env-entry-value id ID #IMPLIED>
<!ATTLIST env-entry-type id ID #IMPLIED>
<!ATTLIST ejb-ref id ID #IMPLIED>
<!ATTLIST ejb-ref-name id ID #IMPLIED>
<!ATTLIST ejb-ref-type id ID #IMPLIED>
<!ATTLIST home id ID #IMPLIED>
<!ATTLIST remote id ID #IMPLIED>
<!ATTLIST ejb-link id ID #IMPLIED>
```

13.3 Examples

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

13.3.1 A Basic Example

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.2//EN" "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
  <display-name>A Simple Application</display-name>
  <context-param>
    <param-name>Webmaster</param-name>
    <param-value>webmaster@mycorp.com</param-value>
  </context-param>
  <servlet>
    <servlet-name>catalog</servlet-name>
    <servlet-class>com.mycorp.CatalogServlet</servlet-class>
    <init-param>
      <param-name>catalog</param-name>
      <param-value>Spring</param-value>
```

```
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>catalog</servlet-name>
    <url-pattern>/catalog/*</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>30</session-timeout>
  </session-config>
  <mime-mapping>
    <extension>pdf</extension>
    <mime-type>application/pdf</mime-type>
  </mime-mapping>
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
  <welcome-file-list>
  <error-page>
    <error-code>404</error-code>
    <location>/404.html</location>
  </error-page>
</web-app>
```

13.3.2 An Example of Security

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.2//EN" "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
  <display-name>A Secure Application</display-name>
  <security-role>
    <role-name>manager</role-name>
  </security-role>
  <servlet>
    <servlet-name>catalog</servlet-name>
    <servlet-class>com.mycorp.CatalogServlet</servlet-class>
    <init-param>
      <param-name>catalog</param-name>
      <param-value>Spring</param-value>
    </init-param>
    <security-role-ref>
      <role-name>MGR</role-name> <!-- role name used in code -->
      <role-link>manager</role-link>
    </security-role-ref>
  </servlet>
  <servlet-mapping>
    <servlet-name>catalog</servlet-name>
    <url-pattern>/catalog/*</url-pattern>
  </servlet-mapping>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>SalesInfo</web-resource-name>
      <url-pattern>/salesinfo/*</url-pattern>
```

```
      <http-method>GET</http-method>
      <http-method>POST</http-method>
      <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
      </user-data-constraint>
      <auth-constraint>
        <role-name>manager</role-name>
      </auth-constraint>
    </web-resource-collection>
  </security-constraint>
</web-app>
```

14 Futures

Many excellent suggestions for additions to this specification have been made by contributors from both our partners, and the general public. As time to market considerations constrain the amount of work that can be done for any particular revision of the specification, some of these suggestions cannot be incorporated in this version of the specification. However, by including these items as future directions, we indicate that we will be considering them for inclusion into a future revision of the specification.

The following are items under consideration:

- Filtering of response content
- Allowing easy access to internationalized content via the getResource method of the `ServletContext` interface.
- Internationalization of web application content
- WebDAV Integration
- Application event handlers
- HTTP Extensions Framework integration

Please note that inclusion of an item on this list is not a commitment for inclusion into a future revision of this specification, only that the item is under serious consideration and may be included into a future revision.