CSc31800: Internet Programming, CS-CCNY, Spring 2004 Jinzhong Niu May 9, 2004

Tomcat Server[1]

1 Introduction

Tomcat is a Java servlet container and web server from the Jakarta project of the Apache Software Foundation (http://jakarta.apache.org). A web server is, of course, the program that dishes out web pages in response to requests from a user sitting at a web browser. But web servers aren't limited to serving up static HTML pages; they can also run programs in response to user requests and return the dynamic results to the user's browser. This is an aspect of the web that Apache's Tomcat is very good at because Tomcat provides both Java *servlet* and *Java Server Pages* (*JSP*) technologies (in addition to traditional static pages and external CGI programming). The result is that Tomcat is a good choice for use as a web server for many applications. And it's a very good choice if you want a free, open source (http://opensource.org/) servlet and JSP engine.

Tomcat can be used stand-alone, but it is often used "behind" traditional web servers such as Apache `httpd`, with the traditional server serving static pages and Tomcat serving dynamic servlet and JSP requests.

No matter what we call Tomcat, a Java servlet container or servlet and JSP engine, we mean Tomcat provides an environment in which servlets can run and JSP can be processed. Similarly, we can absolutely say a CGI-enabled Web server is a CGI program container or engine since the server can accommodate CGI programs and communicate with them according to CGI specification. Between Tomcat and the servlets and JSP code residing on it, there is also a standard regulating their interaction, servlet and JSP specification, which is in turn a part of Sun's J2EE (Java 2 Enterprise Edition).

But what are servlets and JSP? Why do we need them? Let's take a look at them in the following subsections before we cover them in much more detail in the future.

[1]This note contains materials from *Tomcat: The Definitive Guide*, O'Reilly.

2 Web Applications of Servlets and JSP

2.1 Advantages

Traditionally, before Java servlets, when we mention web applications, we mean a collection of static HTML pages and a few CGI scripts to generate the dynamic content portions of the web application, which were mostly written in C/C++ or Perl. Those CGI scripts could be written in a platform-independent way, although they didn't need to be (and for that reason often weren't). Also, since CGI was an accepted industry standard across all web server brands and implementations, CGI scripts could be written to be web server implementation-independent. In practice, some are and some aren't. The biggest problem with CGI was that the design made it inherently *slow* and *unscalable*.

For every HTTP request to a CGI script, the OS must fork and execute a new process, and the design mandates this. When the web server is under a high traffic load, too many processes start up and shut down, causing the server machine to dedicate most of its resources to process startups and shutdowns instead of fulfilling HTTP requests.

As for scalability, CGI inherently has nothing to do with it. As we know, whether command line arguments, environment variables or stdin/stdout are used for writing to or reading from CGI programs, all of them are limited to the local machine, not involving networking or distributed mechanisms at all. Contrastingly, Java servlets and their supporting environments are capable of scalability. I will talk about this in future classes.

Another approach to generating dynamic content is web server modules. For instance, the Apache httpd web server allows dynamically loadable modules to run on startup. These modules can answer on pre-configured HTTP request patterns, sending dynamic content to the HTTP client/browser. This high-performance method of generating dynamic web application content has enjoyed some success over the years, but it has its issues as well. Web server modules can be written in a platform-independent way, but there is no web server implementation-independent standard for web server modulesthey're specific to the server you write them for, and probably won't work on any other web server implementation.

Now let us take a look at the Java side. Java brought platform independence to the server, and Sun wanted to leverage that capability as part of the solution toward a fast and platform-independent web application standard. The other part of this solution was Java servlets. The idea behind servlets was to use Java's simple and powerful multithreading to answer requests without starting new processes. You can now write a servlet-based web application, move it from one servlet container to another or from one computer architecture to another, and run it without any change (in fact, without even recompiling any of its code).

2.2 What are Servlets and JSP?

Briefly, a servlet is a Java program designed to run in a servlet container (we hope you didn't catch that circular definition), and a JSP is a web page that can call Java code at request time. If you're a system administrator or web master, you can think of JSPs as just another scripting and templating language for HTML pages; you can learn to write JSPs that call Java objects much as you might have used objects in JavaScript. The difference is that the Java runs on the server side, before the web page is sent to the browser. It's more like PHP, or even ASP. Writing Java classes such as servlets and JSP custom tags, however, is a task probably best left to people trained in Java programming.

More precisely, a servlet is a Java program that uses the `javax.servlet` package, subclasses either the `javax.servlet.http.HttpServlet` or `javax.servlet.GenericServlet` Java class, performs some processing (anything the programmer wants that the servlet container allows) in response to user input (such as clicking on a link or filling in and submitting a web form), and generates some kind of output that might be useful on the Web. A servlet can, of course, generate an HTML page, but servlets can and have been written to generate graphs and charts in GIF, PNG, and JPEG formats; printed documents in PDF; or any format the developer can program.

A Java Server Page is basically an HTML page that can call Java language functionality. The design goal of JSPs is to remove "raw" Java code from the web page markup and to have the Java code isolated into external modules that get loaded into the JSP at runtime.

The following gives a servlet example:

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** A servlet that prints a web page with the date at the top.
 */

public class Hello extends HttpServlet {

  /** Called when the user clicks on a link to this servlet
   * @parameter request Encapsulates the details about the input.
   * @parameter response Encapsulates what you need to get a reply to the
   *               user's browser.
   */

  public void doGet(HttpServletRequest request,
    HttpServletResponse response) throws IOException {

    // Get a writer to generate the reply to user's browser
    PrintWriter out = response.getWriter(  );

    // Generate the HTTP header to say the response is in HTML
    response.setContentType("text/html");

    out.println("<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Transitional//EN\"");
    out.println("\t\"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional. dtd\"");
    out.println(">");
    out.println(  );
    out.println("<html><head><title>Hello from a Servlet</title></head> ");
    out.println("<body>");
    out.println("<p>Time on our server is " + new Date(  ) + "</p>");
    out.println("<h1>Hello from a servlet</h1>");
    out.println("<p>The rest of the actual HTML page would be here...</p> ");
    out.println("</body></html>");
  }
}
```

The result was that the calls to `out.println` often outweighed the actual HTML (and when they didn't, it still felt like it to the developer). So Java Server Pages, or JSPs, were developed. You can think of JSPs mainly as HTML pages containing some Java code, instead of Java code containing some HTML. In other words, a JSP is just a servlet turned inside out! So, the above example could be written as the following JSP, which may be named as `date.jsp`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html><head><title>Hello from a JSP</title></head>
<body>
  <p>Time on our server is <%= new java.util.Date() %></p>
  <h1>Hello from a JSP</h1>
  <p>The rest of the actual HTML page would be here...</p>
</body>
</html>
```

The first time a user views the "page," the JSP engine very cleverly turns it inside out so that it can be run as a servlet. In the simplest case, all it does is put `out.println` calls around each piece of HTML. But then why bother with servlets at all? Excellent question. The answer, of course, is that you can do much more than just print HTML.

More details about servlets and JSP is coming up in the following classes.

3 Installing Tomcat

Refer to Activating Tomcat 5.0 and Servlet Applications for the instructions for installing Tomcat on Solaris platform.

4 Web Applications on Tomcat

Tomcat provides an implementation of both the servlet and JSP specifications. This section discusses what a web application looks like exactly on Tomcat and how we deploy it.

4.1 Layout of a Web Application

As we mentioned above, a web application is a collection of static HTML files, dynamic JSPs, and servlet classes. It is defined as a hierarchy of directories and files in a standard layout. Such a hierarchy can be accessed in its "unpacked" form, where each directory and file exists in the filesystem separately, or in a "packed" form known as a *Web ARchive*, or *WAR* file. The former format is more useful during development, while the latter is used when you distribute your application to be installed.

The top-level directory of your web application hierarchy is also the *document root* of your application. Here, you will place the HTML files and JSP pages that comprise your application's user interface. When the system administrator deploys your application into a particular server, he or she assigns a *context path* to your application. Thus, if the system administrator assigns your application to the context path `/catalog`, then a request URI referring to `/catalog/index.html` will retrieve the `index.html` file from your document root. When you do experiments, you yourself are the administrator. To put your web application working on Tomcat, you create a subdirectory under Tomcat's `webapps` directory, which is the context path where you are supposed to put your web application files. Figure 1 shows the general layout of a web application, where `sample_webapp` is assumed to be the context of your web application.

As you can see, the web pages (whether static HTML, dynamic JSP, or another dynamic templating language's content) can go in the root of a web application directory or in almost any subdirectory that you like. Images often go in a `/images` subdirectory, but this is a convention, not a requirement. The `WEB-INF` directory contains several specific pieces of content. First, the `classes` directory is where you place Java class files, whether they are servlets or other class files used by a servlet, JSP, or other part of your application's code. Second, the `lib` directory is where you put Java Archive (JAR) files containing packages of classes. Finally, the `web.xml` file is known as a *deployment descriptor*, which contains configuration for the web application, a description of the application, and any additional customization.

When you install an application into Tomcat, the classes in the `WEB-INF/classes/` directory, as well as all classes in JAR files found in the `WEB-INF/lib/` directory, are made visible to other classes within your particular web application. Thus, if you include all of the required library classes in one of these places, you will simplify the installation of your web application – no adjustment to the system class path (or installation of global library files in your server) will be necessary.

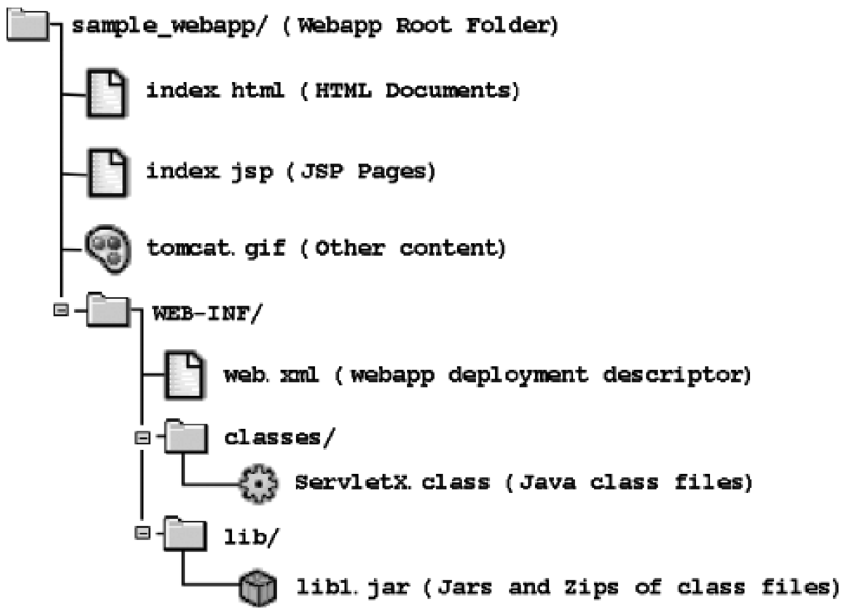
sample_webapp/ (Webapp Root Folder)
index.html (HTML Documents)
index.jsp (JSP Pages)
tomcat.gif (Other content)
WEB-INF/
web.xml (webapp deployment descriptor)
classes/
ServletX.class (Java class files)
lib/
lib1.jar (Jars and Zips of class files)

Figure 1: Servlet Web Application File Layout

4.2 Application Deployment

Once you have gotten all the files of your web application ready, it is time to deploy the application on Tomcat. This step can be done in two ways to be explained respectively in the following parts.

4.2.1 Deploying "Unpacked" Servlets and Java Server Pages

A web application can be deployed in Tomcat by simply copying the unpacked directory hierarchy into a subdirectory in directory `$CATALINA_HOME/webapps/`, where `$CATALINA_HOME` is the directory where Tomcat is installed.

As mentioned above, the `/WEB-INF/web.xml` file contains the Web Application Deployment Descriptor for your application. As the filename extension implies, this file is an XML document, and defines everything about your application that a server needs to know.

For servlets or JSPs to be accessible, you must configure the URI to which a servlet or a JSP is mapped by providing a `servlet-mapping` element in the `WEB-INF/web.xml` file, for example. Listing the servlet in the descriptor is required if you want to provide an alternate mapping, pass any initialization parameters to the servlet, specify loading order on startup, and so on. The `servlet` element is an XML tag that appears near the start of `web.xml`, and it is used for all of these tasks.

Here is an example of a servlet with most of its allowed subelements:

```
<servlet>
  <icon>
    <small-icon>/images/tomcat_tdg16x16.jpg</small-icon>
  </icon>

  <servlet-name>InitParams</servlet-name>

  <display-name>InitParams Demo Servlet</display-name>

  <description>
    A servlet that shows use of both servlet- and
      webapp-specific init-params
  </description>

  <servlet-class>InitParams</servlet-class>

  <init-param>
    <param-name>myParam</param-name>
    <param-value>
      A param for the Servlet:
      Forescore and seven years ago...
    </param-value>
  </init-param>

  <load-on-startup>25</load-on-startup>

</servlet>
```

Once you have your servlets in place, you may also need to add JSPs to your application. JSPs can be installed anywhere in a web application, except under `WEB-INF`; this folder is protected against access from the Web, since it can contain initialization parameters such as database connections, names, and passwords. JSPs can be copied to the root of your web application or placed in any subdirectory other than `WEB-INF`. The same goes for any static content, such as HTML files, data files, and image files.

4.2.2 Deploying Applications in WAR Format

Although you can create directories and copy files using the techniques in the previous section, there are some advantages to using the Web Application Archive packaging format described in the servlet specification. A major benefit with Tomcat is automatic deployment: a single WAR file can be copied directly to Tomcat's `webapps` directory, and it will be automatically available as a context, without requiring any configuration.

Creating WAR files is actually accomplished in the same way you create JAR files: through the `jar` command. So, assuming you have your web application set up correctly and completely in a directory called `testapp`, you can do the following:

```
$ cd ~/testapp
$ jar cvf  ~/testapp.war .
```

The `c` says you want to create an archive. The `v` is optional; it says you want a verbose listing as it creates the archive. The f is required and says that the argument following the letters (`c`, `v`, `f`, ...) is an output filename. The subsequent filename arguments are input names, and they can be files or directories (directories are copied recursively).

That little dot (`.`) at the end of the above command is importantit means "archive the contents of the current directory." Notice also that, although it is a *JAR* file, we called it a *WAR* to indicate that it contains a complete web application; this is recommended in the servlet specification. Once you've issued the command, you should see output similar to the following:

```
bash-2.04$ cd testapp
bash-2.04$ jar cvf testapp.war *
added manifest
adding: WEB-INF/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/Hello.java(in = 1368) (out= 631)(deflated 53%)
adding: WEB-INF/classes/Makefile(in = 580) (out= 360)(deflated 37%)
adding: WEB-INF/classes/Hello.class(in = 1337) (out= 792)(deflated 40%)
adding: WEB-INF/web.xml(in = 1283) (out= 511)(deflated 60%)
adding: index.jsp(in = 367) (out= 253)(deflated 31%)
```

If you are using Tomcat's automatic deployment feature, you can copy the new WAR file into Tomcat's `webapps` directory to deploy it. You may also need to restart Tomcat, depending on your configuration (by default, Tomcat does not need to be restarted when new web applications are deployed). The web application contained in your WAR file should now be ready for use.

When Tomcat is started, it will automatically expand the web application archive file into its unpacked form, and execute the application that way. This approach would typically be used to install an additional application, provided by a third party vendor or by your internal development staff, into an existing Tomcat installation. NOTE that if you use this approach, and wish to update your application later, you must both replace the web application archive file AND delete the expanded directory that Tomcat created, and then restart Tomcat, in order to reflect your changes.

4.2.3 Example

Download `http://www.cs.gc.cuny.edu/~jniu/teachings/csc31800/programs/testapp.war` and put it in `$CATALINA_HOME/webapps/`. After you start Tomcat, the web application presented by `testapp.war` is ready for use.

You may visit `http://[hostname]/testapp/` or `http://[hostname]/testapp/hello` (servlet `hello` visited this time).

Or you can create a subdirectory `"testapp"` in `webapps` and uncompress the WAR file in it. Then you remove the WAR file and restart Tomcat. Now the "unpacked" version is ready for access.

5 Managing Realms, Roles, and Users

The security of a web application's resources can be controlled either by the container or by the web application itself. The J2EE specification calls the former *container-managed security* and the latter *application-managed security.* Tomcat provides several approaches for handling security through built-in mechanisms, which represents container-managed security. On the other hand, if you have a series of servlets and JSPs with their own login mechanism, this would be considered application-managed security. In both types of security, users and passwords are managed in groupings called realms. This section details setting up Tomcat realms and using the built-in security features of Tomcat to handle user authentication.

The combination of a realm configuration in Tomcat's `conf/server.xml` file and a `<security-constraint>` in a web application's `WEB-INF/web.xml` file defines how user and role information will be stored and how users will be authenticated for the web application. There are many ways of configuring each; feel free to mix and match.

5.1 Realms

In order to use Tomcat's container-managed security, you must set up a *realm.* A realm is simply a collection of users, passwords, and roles. Web applications can declare which resources are accessible by which groups of users in their `web.xml` deployment descriptor. Then, a Tomcat administrator can configure Tomcat to retrieve user, password, and role information using one or more of the realm implementations.

Tomcat contains a pluggable framework for realms and comes with several useful realm implementations: `UserDatabaseRealm`, `JDBCRealm`, etc.. Later on, we discuss only `UserDatabaseRealm`. Java developers can also create additional realm implementations to interface with their own user and password stores. To specify which realm should be used, insert a `Realm` element into your `server.xml` file, specify the realm to use through the `className` attribute, and then provide configuration information to the realm through that implementation's custom attributes:

```
<Realm className="some.realm.implementation.className"
       customAttribute1="some custom value"
       customAttribute2="some other custom value"
       <!-- etc... -->
/>
```

5.1.1 UserDatabaseRealm

`UserDatabaseRealm` is loaded into memory from a static file and kept in memory until Tomcat is shut down. In fact, the representation of the users, passwords, and roles that Tomcat uses lives only in memory; in other words, the permissions file is read only once, at startup. The default file for assigning permissions in a `UserDatabaseRealm` is `tomcat-users.xml` in the `$CATALINA_HOME/conf` directory.

The `tomcat-users.xml` file is key to the use of this realm. It contains a list of users who are allowed to access web applications. It is a simple XML file; the root element is `tomcat-users` and the only allowed elements are `role` and `user`. Each `role` element has a single attribute: `rolename`. Each user element has three attributes: username, password, and roles. The `tomcat-users.xml` file that comes with a default Tomcat installation contains the XML listed as follows:

```
<!--
  NOTE:  By default, no user is included in the "manager" role
  required to operate the "/manager" web application.  If you
  wish to use this app, you must define such a user - the
  username and password are arbitrary.
-->
<tomcat-users>
  <user name="tomcat" password="tomcat" roles="tomcat" />
  <user name="role1"  password="tomcat" roles="role1"  />
  <user name="both"   password="tomcat" roles="tomcat,role1" />
</tomcat-users>
```

The meaning of `user` and `password` is fairly obvious, but the interpretation of roles might need some explanation. A *role* is a grouping of users for which web applications may uniformly define a certain set of capabilities. For example, one of the demonstration web applications shipped with Tomcat is the Manager application, which lets you enable, disable, and remove other web applications. In order to use this application, you must create a user belonging to the `manager` role. When you first access the Manager application, the browser prompts for the name and password of such a user and will not allow any access to the directory containing the Manager application until a user belonging to that role logs in.

5.2 Container-Managed Security

Container-managed authentication methods control how a user's credentials are verified when a protected resource is accessed. There are four types of container-managed security that Tomcat supports, and each obtains credentials in a different way:

Basic authentication

The user's password is required via HTTP authentication as base64-encoded text.

When a web application uses basic authentication (BASIC in the web.xml file's auth-method element), Tomcat uses HTTP basic authentication to ask the web browser for a username and password whenever the browser requests a resource of that protected web application. With this authentication method, all passwords are sent across the network in base64-encoded text.

The following shows a web.xml excerpt from a club membership web site with a members-only subdirectory that is protected using basic authentication. Note that this effectively takes the place of the Apache web server's .htaccess files.

```
<!--
  Define the Members-only area, by defining
  a "Security Constraint" on this Application, and
  mapping it to the subdirectory (URL) that we want
  to restrict.
 -->

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>
        Entire Application
      </web-resource-name>
      <url-pattern>/members/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
       <role-name>member</role-name>
    </auth-constraint>
  </security-constraint>

  <!-- Define the Login Configuration for this Application -->

  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>My Club Members-only Area</realm-name>
  </login-config>
```

Digest authentication

The user's password is requested via HTTP authentication as a digest-encoded string.

Form authentication

The user's password is requested on a web page form.

Form authentication displays a web page login form to the user when the user requests a protected resource from a web application. Specify form authentication by setting the auth-method element's value to "FORM". The Java Servlet Specification Versions 2.2 and above standardize container-managed login form submission URIs and parameter names for this type of application. This standardization allows web applications that use form authentication to be portable across servlet container implementations.

To implement form-based authentication, you need a login form page and an authentication failure error page in your web application, a `security-constraint` element similar to those shown above, and a `login-config` element in your `web.xml` file like the one shown as follows:

```
<login-config>
   <auth-method>FORM</auth-method>
   <realm-name>My Club Members-only Area</realm-name>
   <form-login-config>
     <form-login-page>/login.html</form-login-page>
     <form-error-page>/error.html</form-error-page>
   </form-login-config>
 </login-config>
```

The `/login.html` and `/error.html` in the above example refer to files relative to the root of the web application. The `form-login-page` element indicates the page that Tomcat displays to the user when it detects that a user who has not logged in is trying to access a resource that is protected by a `security-constraint`. The `form-error-page` element denotes the page that Tomcat displays when a user's login attempt fails.

Client-cert authentication

The user is verified by a client-side digital certificate.

The client-cert (`CLIENT-CERT` in the `web.xml` file's `auth-method` element) method of authentication is available only when you're serving content over SSL (i.e., HTTPS). It allows clients to authenticate without the use of a passwordinstead, the browser presents a client-side X.509 digital certificate as the login credential. Each user is issued a unique digital certificate that the web server will recognize. Once users import and store their digital certificates in their web browsers, the browsers may present them to the server whenever the server requests them. If you want to know more about this, please refer to relating books or online manual of Tomcat.

6 Configuring Tomcat

After you have Tomcat running, you will soon find a need to customize its configuration. For example, you might want to support virtual hosting.

Configuring Tomcat is done by editing files and restarting Tomcat. The main configuration files provided with Tomcat that reside in the `$CATALINA_HOME/conf` directory are:

`server.xml`

The main Tomcat configuration file.

`web.xml`

A servlet specification standard format configuration file for servlets and other settings that are global to all web applications.

`tomcat-users.xml`

The default list of roles, users, and passwords used by Tomcat's `UserDatabaseRealm` for authentication.

`catalina.policy`

The Java 2 Standard Edition security policy file for Tomcat. We won't cover this file in our classes.

The first three files are well-formed XML documents, and they are parsed by Tomcat at startup.

6.1 `server.xml`

The following gives the major part of `server.xml`, which will be discussed in details in the following sections.

```
<Server port="8005" shutdown="SHUTDOWN" debug="0">
  <Service name="Catalina">
    <Connector port="8080"
               maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
               enableLookups="false" redirectPort="8443" acceptCount="100"
               debug="0" connectionTimeout="20000"
               disableUploadTimeout="true" />
    <Connector port="8009"
               enableLookups="false" redirectPort="8443" debug="0"
               protocol="AJP/1.3" />
    <Engine name="Catalina" defaultHost="localhost" debug="0">
      <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
             debug="0" resourceName="UserDatabase"/>
      <Host name="localhost" debug="0" appBase="webapps"
            unpackWARs="true" autoDeploy="true"
            xmlValidation="false" xmlNamespaceAware="false">
      </Host>
    </Engine>
  </Service>
</Server>
```

6.1.1 `Server`

The `Server` element refers to the entire Tomcat server. It accepts the three attributes listed in Table 1.

Name	**Meaning**	**Default**
`port`	Port number on which to listen for shutdown requests. This port is accessible only from the computer on which you are running Tomcat, to prevent people out on the Internet from shutting down your server.	`8005`
`shutdown`	The string to be sent to stop the server.	`SHUTDOWN`
`debug`	Amount of debugging information to log. Higher numbers mean more debugging detail (and more disk space used).	`0`

Table 1: Server attributes

There can be only one `Server` element in this file because it represents Tomcat itself. If you need two servers, run two Tomcat instances.

The `shutdown` attribute is an arbitrary string that will be sent to the running Tomcat instance when you invoke the `catalina` script with the `stop` argument. Since your `server.xml` file should not be visible outside your local machine, if you change this string from its default, it will be harder for outsiders (system crackers) to shut down your server. Similarly, the `port` attribute is the port number on which `catalina.sh` stop will attempt to contact the running instance. The port number can be changed to any other port that is not in use. Tomcat listens for these connections only on the *localhost* address, meaning that it should be impossible to shut down your machine from elsewhere on the network.

6.1.2 `Service`

A `Service` object represents all of the `Connector`s that feed into an `Engine`. Each `Connector` receives all incoming requests on a given port and protocol, and passes them to the `Engine`, which then processes the requests. As such, the `Service` element must contain one or more `Connector` elements and only one `Engine`. The allowable attributes are shown in Table 2.

You will almost never need to modify this element or provide more than one. The default instance is called `"Tomcat-Standalone"`, representing Tomcat itself with any number of `Connector`s.

Name	Meaning	Default
`className`	Class to implement the service. Must be `org.apache.catalina.core.StandardService`, unless you have some very sophisticated Java developers on staff.	`org.apache.catalina.core.` `StandardService`
`name`	A display name for the service.	`Tomcat-Standalone`

Table 2: Service attributes

6.1.3 `Connector`

A `Connector` is a piece of software that can accept connections (hence the name, derived from the Unix system call `connect()`), either from a web browser (using HTTP) or from another server, such as Apache httpd. All of the `Connector`s provided with Tomcat support the attributes shown in Table 3.

Name	Meaning	Default
`className`	The full Java name of the implementing class, which must implement the `org.apache.catalina.Connector` interface.	None; required
`scheme`	Defines the string value returned by `request.getScheme()` in servlets and JSPs. Should be `https` for an SSL connector.	`http`
...	...	...

Table 3: Connector attributes

Tomcat, in a default installation, is configured to listen on port 8080 rather than the conventional web server port number 80. This is sensible because the default port 80 is often in use, and because opening a network server socket listener on the default port 80 requires special privileges on Unix operating systems. However, there are many applications for which it makes sense to run Tomcat on port 80.

To change the port number, edit the main `Connector` element in the `server.xml` file. Find the XML tag that looks something like this:

```
<!-- Define a non-SSL Coyote HTTP/1.1 Connector on port 8080 -->

<Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
   port="8080" minProcessors="5" maxProcessors="75"
   enableLookups="true" redirectPort="8443"
   acceptCount="100" debug="0" connectionTimeout="20000"
   useURIValidationHack="false" disableUploadTimeout="true" />
```

6.1.4 Engine

An `Engine` element represents the software that receives requests from one of the `Connectors` in its `Service`, hands them off for processing, and returns the results to the `Connector`. The `Engine` element supports the attributes shown in Table 4.

Name	Meaning	Default
`className`	The class implementing the engine. Must be `org.apache.catalina.core.StandardEngine`.	`org.apache.catalina .core.StandardEngine` (default, so you can omit this attribute)
`defaultHost`	The nested host that is the default for requests that do not have an HTTP 1.1 `Host:` header.	`localhost`
`jvmRoute`	A tag for routing requests when load balancing is in effect. Must be unique among all Tomcat instances taking part in load balancing.	
`name`	A display name.	Standalone
...	...	...

Table 4: Engine attributes

6.1.5 Host

A `Host` element represents one host (or virtual host) computer whose requests are processed within a given `Engine`.

To use virtual hosts in Tomcat, you only need to set up the DNS or hosts data for the host. For testing, making an IP alias for `localhost` is sufficient. You then need to add a few lines to the `server.xml` configuration file:

```
<Server port="8005" shutdown="SHUTDOWN" debug="0">
  <Service name="Tomcat-Standalone">
    <Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
               port="8080" minProcessors="5" maxProcessors="75"
               enableLookups="true" redirectPort="8443"/>
    <Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
               port="8443" minProcessors="5" maxProcessors="75"
               acceptCount="10" debug="0" scheme="https" secure="true"/>
      <Factory className="org.apache.coyote.tomcat4.CoyoteServerSocketFactory"
               clientAuth="false" protocol="TLS" />
    </Connector>
    <Engine name="Standalone" defaultHost="localhost" debug="0">
      <Host name="localhost" debug="0" appBase="webapps"
```

```
          unpackWARs="true" autoDeploy="true">
        <Context path="" docBase="ROOT" debug="0"/>
        <Context path="/orders" docBase="/home/ian/orders" debug="0"
                 reloadable="true" crossContext="true">
        </Context>
      </Host>
      <Host name="www.somename.com" appBase="/home/somename/web">
        <Context path="" docBase="."/>
      </Host>
    </Engine>
  </Service>
</Server>
```

6.1.6 Context

A `Context` represents one web application within a Tomcat instance. Your web site is made up of one or more `Contexts`. Table 5 is a list of the key attributes in a `Context`.

Attribute	**Meaning**	**Default**
`crossContext`	Specifies whether `ServletContext.getContext (otherWebApp)` should succeed (true) or return null (false)	false, for generally good security reasons
`docBase`	URL relative to virtual host	None; mandatory
`path`	Absolute path to the directory	None; mandatory
`reloadable`	Specifies whether servlet files on disk will be monitored, and reloaded if their time-stamp changes	false

Table 5: Context attributes

Here are some Context examples:

```
<!-- Tomcat Root Context -->
<Context path="" docBase="/home/ian/webs/daroadweb" debug="0"/>

<!-- buzzinservlet -->
<Context path="/buzzin"
  docBase="/home/ian/javasrc/threads/buzzin"
  debug="0" reloadable="true">
</Context>

<!-- chat server applet -->
<Context path="/chat" docBase="/home/ian/javasrc/network/chat" />

<!-- darian web -->
<Context path="/darian" docBase="/home/ian/webs/darian" />
```

6.1.7 `Realm`

A `Realm` represents a security context, listing users that are authorized to access a given Context and roles (similar to groups) that users are allowed to be in. So a `Realm` is like an administration database of users and groups. Indeed, several of the `Realm` implementations are interfaces to such databases.

The only standard attribute for `Realm` is `classname`, which must be either one of the supported realms listed in Table 6 or a custom `Realm` implementation. `Realm` implementations must be written in Java and must implement the `org.apache.catalina.Realm` interface. The provided `Realm` handlers are listed in Table 6.

Name	**Meaning**
`JAASRealm`	Authenticates users via the Java Authentication and Authorization Service (JAAS)
`JDBCRealm`	Looks users up in a relational database using JDBC
`JNDIRealm`	Uses a Directory Service looked up in JNDI
`MemoryRealm`	Looks users up in the `tomcat-users.xml` file or another file in the same format
UserDatabaseRealm	Uses a `UserDatabase` (which also reads `tomcat-users.xml` or another file in the same format) that is looked up in JNDI; intended to replace `MemoryRealm` in Tomcat 4.1

Table 6: Tomcat's Realm implementations

6.2 `web.xml`

The `web.xml` file format is defined in the Servlet Specification, and will be used in every servlet-conforming Java servlet container. This file format is used in two places in Tomcat: in the `$CATALINA_BASE/conf` directory and in each web application. Each time Tomcat deploys an application (during startup or when the application is reloaded), it reads the global `conf/web.xml`, followed by the `WEB-INF/web.xml` within your web application (if there is one). As you'd expect, then, settings in the `conf/web.xml` file apply to all web applications, whereas settings in a given web application's `WEB-INF/web.xml` apply to only that application.

6.2.1 `web-app`

The root element of this XML deployment descriptor is `web-app`; its top-level elements and the order in which they must appear is shown in Table 7. There are no required elements, but you should always have at least a display-name element for identification.

Element	**Quantity allowed**	**Meaning**
`icon`	0 or 1	A display file, for use in GUI administration tools
`display-name`	0 or 1	Short name, for use in GUI admin tools
`description`	0 or 1	Longer description
`distributable`	0 or 1	Whether the web application can be load-balanced, i.e., distributed to multiple servers
`context-param`	0 or more	Parameters to be made available to all servlets
`servlet`	0 or more	Short name, class name, and options for a servlet
`servlet-mapping`	0 or more	Specifies any non-default URL for a servlet
`mime-mapping`	0 or more	MIME types for files on server
`welcome-file-list`	0 or 1	Alternate default page in directories
`error-page`	0 or more	Alternate error page by HTTP error code
`security-constraint`	0 or more	Requires authentication (e.g., for a protected area of a web site)
`login-config`	0 or 1	Specifies how the login mechanism is to work for a `security-constraint`
`security-role`	0 or more	List name of security role, for use with `security-constraint`

Table 7: Child elements of `web-app`

6.3 `tomcat-users.xml`

This file contains a list of usernames, roles, and passwords, all of which have been explained before. It is a simple XML file; the root element is `tomcat-users`, and the only allowed child elements are `role` and `user`. Each `role` element has one attribute called `rolename`, and each `user` element has three attributes: `name`, `password`, and `roles`.