**MODULE 4**

**Configuring Tomcat 7**

In this module, you’ll focus on basic Tomcat configuration by examining the files found in *CATALINA\_HOME/conf*. The default installation of Tomcat uses these files to configure the server when it starts up; therefore, it’s of the utmost importance that you understand what the default configuration will do and how you can modify it.

You’ll see the main configuration file, server.xml, and Tomcat’s other configuration files.

As you saw in Module 1, Tomcat uses a component-based, hierarchical architecture. This model greatly simplifies the configuration of the complex server.

You’ll see the top-level components and the hierarchy of containers below them, as well as the nested components, all of which should be familiar to you from Module 1. By the end of this chapter, you’ll be comfortable with the default configuration of Tomcat, and you’ll also be able to modify this configuration for your own needs.

**Examining Tomcat’s Configuration Files**

Tomcat’s configuration directory is CATALINA*\_HOME/conf*. It contains the following files:

* catalina.policy
* catalina.properties
* context.xml
* server.xml
* logging.properties
* tomcat-users.xml
* web.xml

**Using *catalina.policy* for Access Control**

Because you’ll see more on Tomcat security in Module 12, in this chapter you’ll take only a quick look through catalina.policy to learn how it provides fine-grained access control to a Tomcat server administrator through the built-in security model of Java.

Any access to system resources that isn’t explicitly allowed is prohibited; therefore, you must anticipate all the resources that Tomcat will need and explicitly grant permission for it to do so. By default, Tomcat starts without security. To start it with security, use the *-security* switch.

*[root@tomcat: ~]# $CATALINA\_HOME/bin/startup -security*

Tomcat only reads, processes, and enforces the catalina.policy file when started in the security manager in this manner. The general policy entry is in the following form:

grant <security principal> { permission list... };

Here, <security principal> is typically a body of trusted code.

Looking at the catalina.policy file, the first set of permissions grants access to all resources for code

from the Java compiler directories; this is essentially the Java compiler and runtime system code. (See

<http://java.sun.com/j2se/1.5.0/docs/guide/security/permissions.html> for details about permissions.)

*// These permissions apply to javac*

*grant codeBase "file:${java.home}/lib/-" { permission java.security.AllPermission;*

*};*

*// These permissions apply to all shared system extensions grant codeBase "file:${java.home}/jre/lib/ext/-" {*

*permission java.security.AllPermission;*

*};*

*// These permissions apply to javac when ${java.home] points at $JAVA\_HOME/jre grant codeBase "file:${java.home}/../lib/-" {*

*permission java.security.AllPermission;*

*};*

*// These permissions apply to all shared system extensions when*

*// ${java.home} points at $JAVA\_HOME/jre*

*grant codeBase "file:${java.home}/lib/ext/-" { permission java.security.AllPermission;*

*};*

As these directories have access to the entire system, it’s vital that you protect them using your operating system file protection features (see Module 12 for details). Without this precaution, malicious code could run unchecked on your system.

The next section of *catalina.policy*  grants the Catalina server and API libraries access to all resources.

*/ These permissions apply to the daemon code*

*grant codeBase "file:${catalina.home}/bin/commons-daemon.jar"*

*{ permission java.security.AllPermission;};*

*// These permissions apply to the logging API*

*grant codeBase "file:${catalina.home}/bin/tomcat-juli.jar" { permission java.security.AllPermission;};*

*// These permissions apply to the server startup code grant codeBase "file:${catalina.home}/bin/bootstrap.jar" {*

*permission java.security.AllPermission;};*

*// These permissions apply to the servlet API classes*

*// and those that are shared across all class loaders*

*// located in the "lib" directory*

*grant codeBase "file:${catalina.home}/lib/-" { permission java.security.AllPermission;};*

Again, you must secure the previous directories on the file system, thus avoiding the possibility of

an attacker adding malicious code to them. Any class files you place in these directories will be granted access to all system resources. The final set of permissions in catalina.policy contains the default web application permissions. They’re significantly more restrictive than those shown previously. In other words, they’re never granted the *java.security.AllPermission super* permission.

The first section enables access to system properties that enable Java Naming and Directory Interface (JNDI) and JDBC access.

*grant {*

*// Required for JNDI lookup of named JDBC DataSource's and*

*// javamail named MimePart DataSource used to send mail permission java.util.PropertyPermission "java.home", "read"; permission java.util.PropertyPermission "java.naming.\*", "read"; permission java.util.PropertyPermission "javax.sql.\*", "read";*

The next section enables read-only access to some operating system description properties: the type of operating system Tomcat is running under and what this operating system uses to separate file extensions in a filename.

*// OS-specific properties to allow read access*

*permission java.util.PropertyPermission "os.name", "read"; permission java.util.PropertyPermission "os.version", "read"; permission java.util.PropertyPermission "os.arch", "read"; permission java.util.PropertyPermission "file.separator", "read"; permission java.util.PropertyPermission "path.separator", "read"; permission java.util.PropertyPermission "line.separator", "read";*

The third section enables read-only access to some JVM-specific properties that are often used in application programming.

*// JVM properties to allow read access*

*permission java.util.PropertyPermission "java.version", "read"; permission java.util.PropertyPermission "java.vendor", "read"; permission java.util.PropertyPermission "java.vendor.url", "read"; permission java.util.PropertyPermission "java.class.version", "read";*

*permission java.util.PropertyPermission "java.specification.version", "read"; permission java.util.PropertyPermission "java.specification.vendor", "read"; permission java.util.PropertyPermission "java.specification.name", "read";*

*permission java.util.PropertyPermission "java.vm.specification.version", "read"; permission java.util.PropertyPermission "java.vm.specification.vendor", "read"; permission java.util.PropertyPermission "java.vm.specification.name", "read"; permission java.util.PropertyPermission "java.vm.version", "read";*

*permission java.util.PropertyPermission "java.vm.vendor", "read"; permission java.util.PropertyPermission "java.vm.name", "read";*

The next two sections provide access for JavaBean getAttribute methods and the XML parser debugger, frequently required during code development (see the JavaBean and Java API for XML Processing [JAXP] specifications for more details on these properties).

// Required for OpenJMX

permission java.lang.RuntimePermission "getAttribute";

// Allow read of JAXP-compliant XML parser debug

permission java.util.PropertyPermission "jaxp.debug", "read";

The final section gives permission to the Jasper runtime classes for precompiled JSP pages.

Internal Tomcat classes aren’t available by default, but they can be made available in the *catalina.properties* file, which is described next.

*// Precompiled JSPs need access to this package. permission java.lang.RuntimePermission*

*"accessClassInPackage.org.apache.jasper.runtime"; permission java.lang.RuntimePermission*

*"accessClassInPackage.org.apache.jasper.runtime.\*";};*

The following permissions are the minimal ones that are granted by default to web applications. Your secured production configuration may require additional access to a JDBC server or network access to an external authentication system. You can find examples of these at the end of catalina.policy.

// The permissions granted to the context root directory apply to JSP pages.

// grant codeBase "file:${catalina.home}/webapps/examples/-" {

// permission java.net.SocketPermission "dbhost.mycompany.com:5432", "connect";

// permission java.net.SocketPermission "\*.noaa.gov:80", "connect";

// };

// The permissions granted to the context WEB-INF/classes directory

// grant codeBase "file:${catalina.home}/webapps/examples/WEB-INF/classes/-" {

// };

// The permission granted to your JDBC driver

// grant codeBase "jar:file:${catalina.home}

/webapps/examples/WEB-INF/lib/driver.jar!/-" {

// permission java.net.SocketPermission "dbhost.mycompany.com:5432", "connect";

// };

// The permission granted to the scrape taglib

// grant codeBase "jar:file:${catalina.home}

/webapps/examples/WEB-INF/lib/scrape.jar!/-" {

// permission java.net.SocketPermission "\*.noaa.gov:80", "connect";

// };

**Using catalina.properties to Configure Tomcat’s Class Loaders**

You use the *catalina.properties* file to configure Tomcat’s class loaders. These determine which classes are available to different parts of the server. In the previous chapter, you saw the three directories—common, server, and shared—that are set as the path for the class loaders by default. You can change these directories in *catalina.properties*.

Another setting you can alter here is the classes that are available to web applications running on the server. When a class loader tries to load a forbidden class, a *java.security.AccessControlException* is thrown. This setting applies only if you start Tomcat in security mode.

The first section lists the forbidden packages. By default, Tomcat won’t allow web applications to load any of Tomcat’s internal classes.

*# List of comma-separated packages that start with or equal this string*

*# will cause a security exception to be thrown when*

*# passed to checkPackageAccess unless the*

*# corresponding RuntimePermission ("accessClassInPackage."+package) has been granted.*

*package.access=sun.,org.apache.catalina.,org.apache.coyote., org.apache.tomcat.,org.apache.jasper.,sun.beans.*

If a web application on your server wants to have access to an internal class, say org.apache.tomcat.util.IntrospectionUtils, and you’re happy to let it, then you would add the following to catalina.policy:

*// Permission for org.apache.tomcat. package permission java.lang.RuntimePermission*

*"accessClassInPackage.org.apache.tomcat.util";*

The next section disallows users from defining classes in certain restricted packages.

*# List of comma-separated packages that start with or equal this string*

*# will cause a security exception to be thrown when*

*# passed to checkPackageDefinition unless the*

*# corresponding RuntimePermission ("defineClassInPackage."+package) has*

*# been granted.*

*# by default, no packages are restricted for definition, and none of*

*# the class loaders supplied with the JDK call checkPackageDefinition.*

*package.definition=sun.,java.,org.apache.catalina.,org.apache.coyote., org.apache.tomcat.,org.apache.jasper.*

This section is here for completeness only and doesn’t affect Tomcat’s operation. It defines Tomcat’s common class loader, which in this case corresponds to the common directory and its subdirectories.

*# List of comma-separated paths defining the contents of the "common"*

*# class loader. Prefixes should be used to define what is the repository type.*

*# Path may be relative to the CATALINA\_HOME path or absolute. If left as blank,*

*# the JVM system loader will be used as Catalina's "common" loader.*

*# Examples:*

*# "foo": Add this folder as a class repository*

*# "foo/\*.jar": Add all the JARs of the specified folder as class*

*# repositories*

*# "foo/bar.jar": Add bar.jar as a class repository common.loader=${catalina.home}/lib,${catalina.home}/lib/\*.jar*

Any classes placed in these directories will be available to Tomcat’s internal classes as well as all web applications. The next section defines Tomcat’s server class loader, which in this case corresponds to the server directory and its subdirectories.

*# List of comma-separated paths defining the contents of the "server"*

*# class loader. Prefixes should be used to define what is the repository type.*

*# Path may be relative to the CATALINA\_HOME path or absolute. If left as blank,*

*# the "common" loader will be used as Catalina's "server" loader.*

*# Examples:*

*# "foo": Add this folder as a class repository*

*# "foo/\*.jar": Add all the JARs of the specified folder as class*

*# repositories*

*# "foo/bar.jar": Add bar.jar as a class repository server.loader=*

Any classes placed in these directories will be available to Tomcat’s internal classes only. The final section of this file defines Tomcat’s shared class loader, which in this case corresponds to the shared directory and its subdirectories. If this definition is left out, the common class loader will be used.

*# List of comma-separated paths defining the contents of the "shared"*

*# class loader. Prefixes should be used to define what is the repository type.*

*# Path may be relative to the CATALINA\_BASE path or absolute. If left as blank,*

*# the "common" loader will be used as Catalina's "shared" loader.*

*# Examples:*

*# "foo": Add this folder as a class repository*

*# "foo/\*.jar": Add all the JARs of the specified folder as class*

*# repositories*

*# "foo/bar.jar": Add bar.jar as a class repository shared.loader*

Any classes placed in these directories will be available to web applications but not Tomcat’s internal classes.

**Using server.xml to Configure Tomcat**

Tomcat’s component-based architecture considerably simplifies configuration. Any properties that are set on the outer components are inherited by the inner components. For example, a listener that’s configured in an engine will be used by a nested host component.

However, if you need a lower-level component to have its own setup, you can override the outer configuration by adding components to the inner component. For example, you could set a realm on a context component to override the realm configured at the engine level. This means that the web applications running within this context will use the newly defined realm instead of the outer one.

This component-based model lends itself to XML configuration because of XML’s hierarchical nature. Each component is represented by an element in an XML file, which makes it easy to insert and remove components from a server as appropriate. The name of the file that does this job in Tomcat is server.xml, which Tomcat reads at startup.

■Note No DTD or schema exists for server.xml because the attributes of certain elements depend on the class implementing the component that the element represents.

Tomcat comes bundled with a default server.xml file ready to run on your machine. It defines a Catalina service, a Catalina engine, and a localhost host.

**Configuring a Server**

Let’s take a closer look at how to configure a server by going through the server.xml file that comes with Tomcat. As you’ll recall from Module 1, a server component is a top-level component, and any Tomcat instances can have only one server component. This means that the *<Server>* element in server.xml is the root element.

*<Server port="8005" shutdown="SHUTDOWN">*

The *<Server>* element represents the JVM and listens to port 8005 for a shutdown command, which will contain the text *SHUTDOWN*. This provides a graceful way for an administrator (or management console software) to shut down this Tomcat server instance. Table 4-1 lists the possible attributes of the *<Server>* element.

**Table 4-1.** *The Attributes of the* <Server> *Element*

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required?** |
| *className* | The Java class for the server to use. This class must implement the org.apache.catalina.Server interface. The standard implementation is used by default. | No |
| *port* | The TCP/IP port to listen to for the command specified by the shutdown attribute before shutting down gracefully. This command must come from the same physical server machine on which Tomcat is running. This provides a certain level of security when used in combination with the shutdown attribute. | Yes |
| *shutdown* | The command string that must be sent to the port number specified by the port attribute. | Yes |

Table 4-2 lists the subelements of the *<Server>* element.

**Table 4-2.** *The Subelements of the* <Server> *Element*

|  |  |  |
| --- | --- | --- |
| **Subelement** | **Description** | **Number** |
| *<GlobalNamingResources>* | The global JNDI resources for this server | 1 |
| *<Service>* | A grouping of connectors associated with an engine | 1 or more |

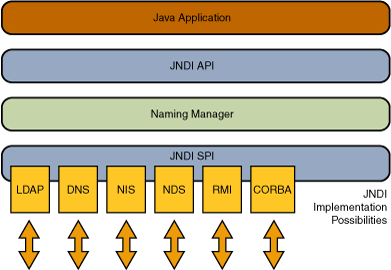
**Configuring Global Naming Resources**

JNDI is an API used for looking up information via a naming and directory service. It’s a platform- independent API, much like JDBC, and it’s designed to work with any compatible naming and directory service—regardless of its native interface API. Some common information you can store and retrieve through JNDI includes the following:

* Usernames and passwords
* An access control policy, such as the Tomcat user and role mechanism
* Organizational directories
* Servers (databases and so on)
* Printers
* Java objects, such as EJBs

JNDI allows you to avoid the problem of programming for the native interfaces of specific platforms and thus simplifies the process immeasurably. JNDI acts as a layer on top of the native interfaces and translates between the Java classes and the naming servers on the server platform, presenting Tomcat with a uniform view of the naming and directory service no matter what the underlying system is.

Additionally, many Java applications use JNDI to locate resources without the need for an underlying naming service. This means a Java application can access resources without knowing their underlying setup or location. For example, a database reference is looked up using its JNDI name only, so it doesn’t matter what the underlying database is or what the driver is. This allows programmers to decouple their applications from hard-coded system resources. Figure 4-1 shows JNDI as a directory service and as a Java lookup mechanism.



***Figure 4-1****. JNDI*

Once the application has the database reference, it can connect to the database directly using JDBC. A constant stored as a JNDI resource can be used across all the web applications running on a server, as well as by any other Java applications that require it. Tomcat and the web applications that run on it use the JNDI resource lookup mechanism extensively.

You configure the server’s global JNDI resources with the *<GlobalNamingResources>* element.

Table 4-3 shows the subelements of <*GlobalNamingResources>*.

**Table 4-3.** *The Subelements of the* <GlobalNamingResources> *Element*

|  |  |  |
| --- | --- | --- |
| **Subelement** | **Description** | **Number** |
| *<Environment>* | A global variable | 0 or more |
| *<Resource>* | A global JNDI resource | 0 or more |

**Configuring Environment Entries**

The first type of resource is a serverwide variable. This variable must be of one of the primitive wrapper types that are specified for environment entries in the Servlet specification. You use an <Environment> entry to specify this kind of resource.

*<Environment name="simpleValue" type="java.lang.Integer" value="30"/>*

This environment entry is called *simpleValue*, is of type *java.lang.Integer*, and has the value 30. It’s looked up using the *java:comp/env/simpleValue* string. Table 4-4 specifies the attributes that *<Environment>* can take.

**Table 4-4.** *The Attributes of the* <Environment> *Element*

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required?** |
| *description* | A description of this environment entry. | No |
| *name* | The name of the environment entry, relative to the java: comp/env context. | Yes |
| *override* | Set this to false if you don’t want a web application deployment descriptor to override this value. The default is true. | No |
| *type* | The fully qualified Java class type of this entry. It must be one of the legal values specified in the Servlet specification for web application deployment descriptor environment entries:  java.lang.Boolean, java.lang.Byte, java.lang.Character, java.lang.Double, java.lang.Float, java.lang.Integer, java.lang.Long, java.lang.Short, and java.lang.String. | Yes |
| *value* | The value of this entry. | Yes |

**Configuring a Global Resource**

Global resources can include JDBC data sources, Enterprise JavaBean (EJB) references, and user authentication databases. You define them with a *<Resource>* element, and you must also define a set of resource parameters to configure the object factory for this resource type. You’ll see how this is done next.

*<Resource name="UserDatabase" auth="Container" type="org.apache.catalina.UserDatabase"*

*description="User database that can be updated and saved">*

*</Resource>*

This is a user database for authenticating users and is set as the default for the Catalina engine further down in server.xml. Table 4-5 describes the attributes that a *<Resource>* element can take.

**Table 4-5.** *The Attributes of the* <Resource> *Element*

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required?** |
| *auth* | Specifies whether the web application signs onto the corresponding resource manager programmatically or whether the container will sign onto the resource manager on behalf of the application. The value of this attribute must be *Application* or *Container*. This attribute is required if the web application uses a *<resource-ref>* element in the web application deployment descriptor but is optional if the application uses a *<resource-env-ref>* instead. | No |
| *description* | A description of this resource. | No |
| *name* | The name of the resource to be created. | Yes |
| *scope* | Specifies whether connections obtained through this resource manager can be shared. The value of this attribute must be  or *Unshareable*. The default is Shareable. | No |
| *type* | The fully qualified Java class name of this resource. | Yes |

**Configuring Resource Parameters in Tomcat 7**

Tomcat 7 no longer uses the *<ResourceParams>* element that was used in Tomcat 5 and earlier. Instead, you provide the information with the <Resource> element’s attributes.

*<Resource name="UserDatabase" auth="Container" type="org.apache.catalina.UserDatabase"*

*description="User database that can be updated and saved" factory="org.apache.catalina.users.MemoryUserDatabaseFactory" pathname="conf/tomcat-users.xml" />*

**Configuring a JDBC Data Source**

Web applications running on your server may use common databases and thus will benefit from a JDBC data source. In fact, even if an application is the only one that uses a database, it will benefit from making it into a data source. This makes it easy for you, as the administrator, to change the underlying database without disturbing web applications.

Another advantage of data sources is that Tomcat can use connection pooling with them, which means database connections can be recycled once they’ve finished executing. This, in turn, leads to improved performance. Tomcat uses the Jakarta Commons Database Connection Pool mechanism, which supports JDBC2.0 and JDBC 3.0. You can find the classes repackaged as part of the Tomcat distribution, which you can find in *CATALINA\_HOME/lib/tomcat-dbcp.jar*.

The first step to data source configuration is to place the required JDBC driver in *CATALINA\_ HOME/lib* or some other directory in the classpath. This will allow Tomcat to find and access this driver.

As you saw earlier, you can configure the JNDI resource factory using the *<Resource>* element. Listing 4-1 shows you a MySQL data source defined for the whole server. This instance of the database will be shared among all the web applications running on the server.

**Listing 4-1.** *Configuring a JDBC Data Source*

*<Resource name="jdbc/TestDB" auth="Container" type="javax.sql.DataSource" maxActive="100" maxIdle="30" maxWait="10000" username="javauser"*

*password="javadude" driverClassName="com.mysql.jdbc.Driver"*

*url= "jdbc:mysql://localhost:3306/javatest?autoReconnect=true"/>*

This listing defines a data source called jdbc/CatalogDB and sets its drivers and connection URL. It illustrates how you could change the underlying database without affecting web applications. In this case, the parameters shown in Table 4-6 are allowed.

**Table 4-6.** *The Parameters for Use with a JDBC Data Source*

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Description** | **Required?** |
| *driverClassName* | Java class name of the JDBC driver. This driver should be placed in *CATALINA\_HOME/common/lib*. | Yes |
| *maxActive* | The maximum number of active connections in this pool. | No |
| *maxIdle* | The maximum number of idle connections in this pool. | No |
| *maxWait* | The time in milliseconds that the driver should wait for a  connection before throwing an exception. | No |
| *username* | The user ID used to log onto the database. | No |
| *password* | The password used to log onto the database. | No |
| *url* | The URL of the database server to be used. | Yes |
| *validationQuery* | A SQL query used to validate a connection. The factory will  perform this query and ensure that rows are returned before  considering the connection valid. | No |

In addition to the previous configuration, the developer must declare the use of the resource in the application’s web.xml file using a <resource-ref> element, as shown in Listing 4-2.

**Listing 4-2.** *Configuring a Reference to a JDBC Data Source in* web.xml

*<?xml version="1.0" encoding="ISO-8859-1"?>*

[*<web-app xmlns="http://java.sun.com/xml/ns/j2ee"*](http://java.sun.com/xml/ns/j2ee)[*xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"*](http://www.w3.org/2001/XMLSchema-instance)[*xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee*](http://java.sun.com/xml/ns/j2ee)[*http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd"*](http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd) *version="2.4">*

*<!-- Describe a DataSource -->*

*<resource-ref>*

*<description>*

*Resource reference to a factory for java.sql.Connection instances that may be used for talking to a particular database that is configured in the tomcatBook.xml file.*

*</description>*

*<res-ref-name>*

*jdbc/CatalogDB*

*</res-ref-name>*

*<res-type> javax.sql.DataSource*

*</res-type>*

*<res-auth> SERVLET*

*</res-auth>*

*</resource-ref>*

*<!-- Define a Security Constraint on this Application -->*

*<security-constraint>*

*<web-resource-collection>*

*<web-resource-name>Tomcat Book Application</web-resource-name>*

*<url-pattern>/\*</url-pattern>*

*</web-resource-collection>*

*<auth-constraint>*

*<role-name>tomcat</role-name>*

*</auth-constraint>*

*<user-data-constraint>*

*<description>*

*Constrain the user data transport for the whole application*

*</description>*

*<transport-guarantee>CONFIDENTIAL</transport-guarantee>*

*</user-data-constraint>*

*</security-constraint>*

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

*<login-config>*

*<auth-method>FORM</auth-method>*

*<realm-name>Tomcat Book Application</realm-name>*

*<form-login-config>*

*<form-login-page>/ch12/login.jsp</form-login-page>*

*<form-error-page>/ch12/error.jsp</form-error-page>*

*</form-login-config>*

*</login-config>*

*<!-- Security roles referenced by this web application -->*

*<security-role>*

*<description>*

*The role that is required to log in to the Tomcat Book Application*

*</description>*

*<role-name>tomcat</role-name>*

*</security-role>*

*</web-app>*

**Configuring Mail Sessions**

*JavaMail* is a standard programming API that can be used to create and send e-mails. Tomcat supports *JavaMail* by allowing you to configure a *JavaMail* session as a JNDI resource. Web applications can then use JNDI to look up and use this session.

You can configure *JavaMail* sessions that web applications can use much in the same way as you can configure JDBC data sources. The theory and practice in both configurations are similar. In the case of *JavaMail* sessions, the web application obtains a reference to the mail session without needing to know about the underlying implementation. Again, this allows you to change the underlying mail server without compromising any web applications.

As already mentioned, the process of setting up a *JavaMail* session is analogous to setting up a JDBC data source. First, you must place the JavaMail API in CATALINA\_HOME/ /lib so that Tomcat and web applications can use its classes. It’s available from <http://java.sun.com/> products/javamail/downloads/index.html. Second, configure the mail session in server.xml as shown in Listing 4-3.

**Listing 4-3.** *Configuring a Mail Session*

*<Resource name="mail/Session" auth="Container" type="javax.mail.Session" mail.smtp.host="localhost"/>*

By convention, you configure mail sessions to resolve to the mail subcontext. The snippet in Listing 4-3 configures the mail/Session context, which refers to an SMTP server running on localhost. You

can modify the SMTP port (if it isn’t at the standard port 25) by setting the *mail.smtp.port parameter*. Finally, set the JNDI resource in web.xml. Listing 4-4 shows the mail/Session reference.

**Listing 4-4.** *Configuring a Reference to a JavaMail Session in* web.xml

*<resource-ref>*

*<res-ref-name>mail/Session</res-ref-name>*

*<res-type>javax.mail.Session</res-type>*

*<res-auth>Container</res-auth>*

*</resource-ref>*

**Configuring a Service**

A service component groups together all the connectors that may be used with an engine.

*<Service name="Catalina">*

This service is called Catalina. This name will be visible in logs and error messages, clearly identifying the component. Service management software can also use it to identify the service instance. Table 4-7 describes the attributes of the *<Service>* element.

**Table 4-7.** *The Attributes of the* <Service> *Element*

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required?** |
| *className* | The Java class name for the service class to use. The default is *org.apache.catalina.core.StandardService*. | No |
| *name* | The service name, used in logging and management. If more than one *<Service>* element appears inside the *<Server>* element, you must make sure their name attributes are different. | Yes |

Table 4-8 describes the subelements that a *<Service>* element can have.

**Table 4-8.** *The Subelements of the* <Service> *Element*

|  |  |  |
| --- | --- | --- |
| **Subelement** | **Description** | **Number** |
| *Connector* | Connects Tomcat, either from users or from another web server. | 1 or more |
| *Engine* | This is Tomcat’s request-processing machinery. | 1 |

**Configuring a Connector**

The following are the two connection points where a request enters Tomcat:

* From a front-end web server, which could be Apache, IIS, or any other web server
* From a web browser

One way to handle these connection requirements is to create a customized version of Tomcat for each situation. This is inefficient and hard to maintain. This is where connectors come in: a connector adapts an engine to the outside world by passing requests into the engine and passing responses out to the user. The connector handles the protocol, connection conventions, and so on, so that the engine doesn’t have to handle them.

You can associate more than one connector with a single engine. For example, you may want to provide an HTTP service and an HTTPS service to your users from the same server. In this case, you configure an HTTP connector and an SSL connector in the same engine. You’ll see more on this in Module 9. In the meantime, let’s look through server.xml and see the default settings. A number of different options are available to you when you configure a connector.

server.xml shows four of the most common.

* An HTTP connector
* An SSL connector
* An AJP 1.3 connector for connecting to another web server
* A proxy connector

The default connector for the Catalina engine is an HTTP/1.1 connector.

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

*<Connector port="8080"*

*maxThreads="150" minSpareThreads="25" maxSpareThreads="75" enableLookups="false" redirectPort="8443" acceptCount="100" debug="0" connectionTimeout="20000" disableUploadTimeout="true" />*

This sets a connector to listen on port 8080 for HTTP requests. Table 4-9 describes the attributes that are common to all connectors, and Table 4-10 describes the HTTP connector’s attributes after the descriptions of the other <Connector> elements in server.xml.

*<!-- Define a SSL Coyote HTTP/1.1 Connector on port 8443 -->*

*<!--*

*<Connector port="8443"*

*maxThreads="150" minSpareThreads="25" maxSpareThreads="75" enableLookups="false" disableUploadTimeout="true" acceptCount="100" debug="0" scheme="https" secure="true" clientAuth="false" sslProtocol="TLS" />*

*-->*

This sets a secure SSL connector to listen on port 8443 for HTTPS requests. It shares all the attributes of an ordinary HTTP connector but has some unique SSL attributes all its own. Table 4-11 describes these.

*<!-- Define a Coyote/JK2 AJP 1.3 Connector on port 8009 -->*

*<Connector port="8009"*

*enableLookups="false" redirectPort="8443" debug="0" protocol="AJP/1.3" />*

This sets up an AJP 1.3 connector listening on port 8009. This type of connector allows Tomcat to connect to an Apache web server to provide JSP pages and servlets while Apache provides HTML pages and important user-management functionality. Table 4-12 describes the AJP connector’s attributes.

*<!-- Define a Proxied HTTP/1.1 Connector on port 8082 -->*

*<!-- See proxy documentation for more information about using this. -->*

*<!--*

*<Connector port="8082"*

*maxThreads="150" minSpareThreads="25" maxSpareThreads="75" enableLookups="false"*

*acceptCount="100" debug="0" connectionTimeout="20000"* *proxyPort="80" disableUploadTimeout="true" />*

-->

The final connector setup in server.xml configures a connector to work with a proxy server. This allows

the proxy to provide a firewall. All these connectors are configured to automatically send errors and logging information to the logger associated with their engine.

As promised, Table 4-9 describes the common attributes that are shared by all the connectors described previously.

**Table 4-9.** *The Common Attributes of the* <Connector> *Element*

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required?** |
| *address* | For servers with more than one IP address, this attribute specifies which address will be used for listening on the specified port. By default, this port will be used on all IP addresses associated with the server. | No |
| *allowTrace* | A Boolean value that enables or disables the TRACE HTTP method (which prompts the server to return a copy of the request back to the client for inspection). The default is false. | No |
| *enableLookups* | Sets whether the DNS host name of the client can be looked up. false skips the DNS lookup and returns the IP address as a string (thereby improving performance). The default is true. | No |
| *maxPostSize* | The maximum size in bytes of a POST request. You can disable this by setting this attribute to a value less than or equal to zero. The default is 2097152 (2 megabytes). | No |
| *redirectPort* | If this connector supports non-SSL requests and a request is received that requires SSL transport, Tomcat will automatically redirect the request to the port number specified here. | No |
| *scheme* | The name of the protocol you want to use for this connector. For example, you’d set this attribute to https for an SSL connector. The default is http. | No |
| *secure* | If you want to have calls to request.isSecure() return true (which is the case for an SSL connector), set this to true. The default is false. | No |
| *URIEncoding* | The character encoding to use to decode the URI bytes. The default is ISO-8859-1. | No |
| *useBodyEncodingForURI* | This specifies if the encoding specified in contentType should be used for URI query parameters, instead of using the URIEncoding. This setting is for compatibility with Tomcat 4.1.x, where the encoding specified in contentType was also used for the parameters from the URL. The default is false. | No |

The default HTTP connector has the attributes described in Table 4-10.

**Table 4-10.** *The Attributes of the HTTP* <Connector> *Element*

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required?** |
| *acceptCount* | The maximum queue length for incoming connection requests when all possible request-processing threads are in use. Any requests received when the queue is full will be refused. The default is 10. | No |
| *bufferSize* | The size (in bytes) of the buffer to be provided for input streams created by this connector. The default is 2048. | No |
| *compressableMimeTypes* | The value is a comma-separated list of MIME types for which HTTP compression may be used. The default is text/html, text/xml, text/plain. | No |
| *compression* | The connector may use HTTP/1.1 GZIP compression in an attempt to save server bandwidth. The acceptable values for the parameter are off (disables compression), on (allows compression, which causes text data to be compressed), force (forces compression in all cases), or an integer (which is equivalent to on but specifies the minimum amount of data before the output is compressed). If the content length isn’t known and compression is set to on (or a more aggressive setting), the output will also be compressed. The default is off. | No |
| *connectionLinger* | The number of milliseconds during which the sockets used by this connector will linger when they are closed. The default is -1 (socket linger is disabled). | No |
| *connectionTimeout* | The number of milliseconds this connector will wait after accepting a connection for the request URI line to be presented. The default is 60000 (that is, 60 seconds). | No |
| *disableUploadTimeout* | Used to set a connection timeout while a servlet is being executed. This gives the servlet longer to complete its execution or allows a longer timeout during data upload. The default is false. | No |
| *maxHttpHeaderSize* | The maximum size of the request and response HTTP header, specified in bytes. The default is 4096. | No |
| *maxKeepAliveRequests* | The maximum number of HTTP requests that can be maintained until Tomcat closes the connection. A setting of 1 will disable HTTP/1.0 and HTTP/1.1 keep-alive and pipelining. A setting of -1 will allow an unlimited amount of pipelined or keep-alive HTTP requests. The default is 100. | No |
| *maxSpareThreads* | The maximum number of unused request-processing threads that will be allowed to exist until the thread pool stops the unnecessary threads. The default is 50. | No |
| *maxThreads* | The maximum number of request-processing threads to be created, which therefore determines the maximum number of simultaneous requests that can be handled. The default is 200. | No |

**Table 4-10.** *The Attributes of the HTTP* <Connector> *Element Cont.*

|  |  |  |
| --- | --- | --- |
| *minSpareThreads* | The number of request-processing threads created when this connector starts. The connector will also make sure it has the specified number of idle processing threads available. This attribute should be set to a value smaller than maxThreads. The default is 4. | No |
| *noCompressionUserAgents* | A comma-separated list of regular expressions matching HTTP user-agents for which compression should not be used. The default is an empty string. | No |
| *port* | The port on which this connector will create a server socket and await incoming connections. Only one application may listen to a particular port number on a particular IP address. | Yes |
| *protocol* | Must be HTTP/1.1 to use the HTTP handler, which is the default. | No |
| *proxyName* | If this connector is being used in a proxy configuration, configure this attribute to specify the server name to be returned for calls to  request.getServerName(). | No |
| *proxyPort* | If this connector is being used in a proxy configuration, configure this attribute to specify the server port to be returned for calls to request.getServerPort(). | No |
| *restrictedUserAgents* | A comma-separated list of regular expressions matching HTTP user-agents for which HTTP/1.1 or HTTP/1.0 keep-alive should not be used, even if they advertise support for these features. The default is an empty string. | No |
| *server* | The Server header for the HTTP response (Tomcat 5.5 only). | No |
| *strategy* | The thread-pooling strategy to be used (Tomcat 5.5 only). The default strategy doesn’t use a master thread. However, you can use a more conventional strategy with a master listener thread by setting this attribute’s value to ms. The master strategy will work significantly better if you also use the threadPriority attribute,  which will apply only to the thread that listens on the server socket. The default is lf. | No |
| *socketBuffer* | The size (in bytes) of the buffer to be provided for socket output buffering. A setting of -1 disables the use of a buffer. The default is 9000. | No |
| *tcpNoDelay* | If set to true, the TCP\_NO\_DELAY option will be set on the server socket, which improves performance under most circumstances. The default is true. | No |
| *threadPriority* | | The priority of the request-processing threads within the JVM. The default is java.lang.Thread#NORM\_ PRIORITY. See the documentation for the java.lang.Thread class for more details on what this priority means. | No |

An SSL connector has a number of unique attributes, as described in Table 4-11.

**Table 4-11.** *The Attributes of an SSL-Enabled* <Connector> *Element*

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required?** |
| *algorithm* | The certificate algorithm to be used. The default is SunX509. | No |
| *clientAuth* | Set to true if you require a valid certificate chain from the client before accepting a connection. false won’t require a certificate chain unless the client requests a resource protected by a security constraint that uses CLIENT-CERT authentication. The default is false. | No |
| *keystoreFile* | The path to the keystore file where you have stored the server certificate to be loaded. The default is .keystore in the home directory of the user that’s running Tomcat. | No |
| *keystorePass* | The password used to access the server certificate from the specified keystore file. The default is changeit. | No |
| *keystoreType* | The type of keystore file to be used for the server certificate. The default is JKS. | No |
| *sslProtocol* | The version of the SSL protocol to use. The default is TLS. | No |
| *ciphers* | A comma-separated list of the encryption ciphers that may  be used. Any available cipher may be used by default. | No |

The final set of attributes belongs to the AJP connector and is described in Table 4-12.

Remember that this connector also has the common attributes described in Table 4-9.

**Table 4-12.** *The Attributes of an AJP1.3* <Connector> *Element*

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required?** |
| *protocol* | Must be AJP/1.3 to use the AJP handler | Yes |

**Configuring an Engine**

You can have as many connectors as you need in a service to handle the different connection requirements for a server, but you can have only one engine. An engine executes web applications when processing incoming requests and generating outgoing responses.

*<!-- Define the top level container in our container hierarchy -->*

*<Engine name="Catalina" defaultHost="localhost" debug="0">*

An engine represents a running instance of the servlet processor; in other words, it’s the servlet engine. The default engine in server.xml is called Catalina. The *defaultHost* is the host component to which this engine will direct a request if it’s not for a known host on this server. The debug attribute here specifies that there will be no debug messages for this engine written by the logger to the log.

Table 4-13 describes the attributes of the *<Engine>* element.

**Table 4-13.** *The Attributes of the* <Engine> *Element*

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required?** |
| *backgroundProcessorDelay* | Represents the delay in seconds between the invocation of the *backgroundProcess*() method on this engine and its child containers, including all hosts and contexts. Child containers will be invoked if their delay value is negative (which would mean they’re using their own processing thread). Setting this to a positive value will cause a thread to be spawned. After waiting the specified amount of time, the thread will invoke the *backgroundProcess*() method on this engine and all its child containers. The default is 10. | No |
| *className* | Class name of the implementation to use. The default is *org.apache.catalina.core. StandardEngine*. | No |
| *defaultHost* | The default hostname. This host will process requests directed to host names on this server that aren’t configured in server.xml. This name must match the name of one of the host elements nested in this engine. | Yes |
| *jvmRoute* | The identifier that must be used in load balancing to enable session affinity. This value must be unique across all Tomcat 5 servers that participate in the cluster. It will be appended to the generated session identifier, therefore allowing the front-end proxy to forward a particular session to the same Tomcat 5 instance. | No |
| *name* | Name of this engine, used in log and error messages. | Yes |

The *<Engine>* element has the subelements described in Table 4-14.

**Table 4-14.** *The Subelements of the* <Engine> *Element*

|  |  |  |
| --- | --- | --- |
| **Subelement** | **Description** | **Number** |
| *Realm* | The user-authentication realm used by Tomcat’s declarative security support. | 0 or 1 |
| *Host* | Each *<Host>* element is a virtual host handled by this engine. Tomcat can handle multiple virtual hosts per engine instance. | 1 or more |
| *Listener* | Life cycle listeners monitor the starting and stopping of the engine. You’ll see one use for listeners in Module 9. | 0 or more |
| *Valve* | Valves add processing logic into the request- and response- handling pipeline at the engine level. Standard valves are used to perform access logging, request filtering, implementing single sign-on, and so on. | 0 or more |

**Configuring a Realm**

The next entry in server.xml is a realm, which is used for user authentication.

*<!-- Because this Realm is here, an instance will be shared globally -->*

*<!-- This Realm uses the UserDatabase configured in the global JNDI resources under the key "UserDatabase". Any edits*

*that are performed against this UserDatabase are immediately available for use by the Realm. -->*

*<Realm className="org.apache.catalina.realm.UserDatabaseRealm" debug="0" resourceName="UserDatabase"/>*

Note that this realm uses the global resource you looked at earlier in the module 3 section. This realm is, therefore, attaching the global resource to the engine in which it resides.

Tomcat uses realms to perform authentication and implement container-managed security. They map usernames to passwords (for authentication) and usernames to user roles (for container-managed security). This means that Tomcat can then determine that users are who they say they are, using authentication, and determine which areas of the server are available to them, using container-managed security.

The user database is only one implementation of a realm. Others are data source realm, JDBC realm, JNDI realm, and JAAS realm. Tomcat 5 supports the memory realm for backward compatibility with Tomcat 4, but it’s inefficient and insecure, so you shouldn’t use it.

**Configuring a User Database Realm**

The user database realm is an upgraded version of the memory realm and is backward compatible with the memory realm. It comes with the same caveats as the memory realm.

**Configuring a Data Source Realm**

Data source realms use JDBC data sources stored with JNDI names to authenticate users. This allows you to change the underlying storage mechanism without having to change the settings on your realms.

**Configuring a JDBC Realm**

JDBC realms access relational databases to obtain authentication information. You can use any source of data that can be accessed with JDBC. This includes ODBC sources, such as Excel or comma-separated files, accessed with the JDBC-ODBC bridge. server.xml

has a number of example JDBC realms commented out. The following uses a MySQL database instead of a text file:

*<!--*

*<Realm className="org.apache.catalina.realm.JDBCRealm" debug="99" driverName="org.gjt.mm.mysql.Driver" connectionURL="jdbc:mysql://localhost/authority" connectionName="test" connectionPassword="test" userTable="users" userNameCol="user\_name" userCredCol="user\_pass"*

*userRoleTable="user\_roles" roleNameCol="role\_name" />*

*-->*

**Configuring a JNDI Realm**

You can configure a JNDI LDAP service provider to provide user information from an existing directory service. This would allow you to use employee information that’s already available.

**Configuring a JAAS Realm**

You can use the JAAS realm to authenticate users using the Java Authentication and Authorization Service (JAAS). This allows you to use any authentication mechanism you choose, but you have to write your own authentication module to implement it.

**Configuring a Host**

A host component represents a single virtual host running on this server.

*<Host name="localhost" debug="0" appBase="webapps" unpackWARs="true" autoDeploy="true" xmlValidation="false" xmlNamespaceAware="false">*

This virtual host is localhost. The applications running in this host are located in the *CATALINA\_HOME/webapps* directory.

The *unpackWARs* attribute tells Tomcat to unpack any packaged WAR files found in the *appBase* directory if it’s set to true. A value of false means that Tomcat will execute the web applications without unpacking them, which saves space but increases response time.

A <Host> element is a container and has the attributes described in Table 4-16. These are the attributes of all hosts, and custom implementations of hosts are possible.

**Table 4-16.** *The Common Attributes of the* <Host> *Element*

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required** |
| appBase | The base directory for this virtual host. This is a directory that may contain web applications to be deployed on this virtual host. You may specify an absolute pathname or a pathname relative to the CATALINA\_HOME directory. | Yes |
| autoDeploy | This flag value indicates whether new web applications added to the appBase directory while Tomcat is running should be deployed automatically. The default is true. | No |
| *background-ProcessorDelay* | The delay in seconds between the invocation of the *backgroundProcess*() method on this host and itschild containers, including all contexts. Child  containers will be invoked if their delay values are negative (which would mean they’re using their own processing threads). A positive value will cause a thread to be spawned. After waiting the specified amount of time, the thread will invoke the *backgroundProcess*() method on this host and all its child containers. A host will use background processing to perform live web application deployment-related tasks. The default is -1, which means the host will rely on the background-processing thread of its parent engine. | Yes |
| *className* | Class name of the implementation to use. The default is org.apache.catalina.core. StandardHost. | No |
| *deployOnStartup* | Set to true to automatically deploy web applications on startup. The default is true. | No |
| *name* | The name of this virtual host, as configured with DNS. One of the hosts nested within an engine must have a name that matches the defaultHost  attribute of that engine. | Yes |

In addition, the standard host has the attributes defined in Table 4-17.

**Table 4-17.** *The Attributes of the Standard* <Host> *Element*

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Description** | **Required** |
| *deployXML* | false disables the ability to deploy applications using an XML configuration file. This also prohibits the manager application from deploying web application directories or WAR files that aren’t located in CATALINA\_HOME/conf/ [Engine\_name]/[Host\_name].  XML-configured applications are deployed with Tomcat’s security permissions, so you should set this to false if untrustworthy users can manage web applications. *The default is true*. | No |
| *errorReportValveClass* | Class name of the error-reporting valve that will be used by this host. You can use this property to customize the look of the error pages that will be generated by Tomcat. The class must implement the *org.apache.catalina.Valve* interface. The default is *org.apache.catalina.valves.ErrorReportValve*. | No |
| *unpackWARs* | Set to true if you want to unpack WAR files in the appBase directory into a corresponding directory structure. False tells Tomcat to run such web applications directly from their WAR file. The default is true. | No |
| *workDir* | A scratch directory to be used by applications running in this host. Each application will have its own directory with temporary read/write use. Configuring a working directory for a context will override this value. This directory is visible to servlets in the web application as a servlet context attribute (of type java.io.File) named javax.servlet.context. tempdir, as described in the Servlet specification. The default is a suitable directory underneath CATALINA\_HOME/work. | No |

Table 4-18 describes the subelements that can be placed inside a *<Host>* element.

**Table 4-18.** *The Subelements of the* <Host> *Element*

|  |  |  |
| --- | --- | --- |
| **Subelement** | **Description** | **Number** |
| *Context* | A context defines a web application deployed within this host.  When using Tomcat 5, you shouldn’t place any context entries in  server.xml because server.xml isn’t reloaded after changes are  made. Use XML configuration files or deployment tools, such as the manager application instead. | 0 or more |
| *Realm* | A realm that can be accessed across all the web applications  running within this host—unless a lower-level component specifies its own realm. | 0 or 1 |
| *Valve* | You can add a valve to monitor access, filter requests, and  implement single sign-on. | 0 or more |
| *Listener* | You can add a listener to monitor life cycle events, such as this host  starting or stopping, and to implement user web applications. | 0 or more |
| *Alias* | Defines an alias for this host if two or more network names need to  apply to it. | 0 or more |

**Configuring a Valve**

A valve is a Tomcat-specific interception mechanism for catching requests and responses. Any requests destined for the localhost host would be passed through the valve defined here, if it were to be uncommented.

*<!--*

*<Valve className="org.apache.catalina.valves.AccessLogValve" directory="logs" prefix="localhost\_access\_log." suffix=".txt"*

*pattern="common" resolveHosts="false"/>*

*-->*

The *org.apache.catalina.valves.AccessLogValve* valve creates access log files in the same format as Apache’s log file. The previous configuration will create log files, in the common format, in *CATALINA\_HOME/logs*. They will be named in the form localhost*\_access\_log.DATE*.

You can also install valves at the engine level. Any valve that’s installed at the engine level will have access to every request handled by the engine, regardless of which connector the request comes through. Therefore, you must test the valve thoroughly and make sure it doesn’t require a lot of processor time to complete its operation. The standard valves that come with Tomcat have been designed and tested for efficiency.

■Note Valves are specific to Tomcat and not part of the Servlet specification. Application programmers can use filters as a similar interception mechanism. They are part of the Servlet specification and reside within a web application.

You’ll see how to configure and use the standard valves in Module 5.

**Configuring a Listener**

If you have an object that needs to know about server life cycle events, then you need to implement a listener. The basic listener configuration is as follows:

*<Listener className="com.acme.listeners.Listener" />*

The className attribute is required. You can add other attributes according to the properties of the class. They’re matched with the standard JavaBean naming mechanism.

**Configuring an Alias**

If you need to map more than one network name to a single virtual host, then you need to configure an alias. For example, say you want to map [www.company.com](http://www.company.com/) and [www.company.org](http://www.company.org/) to the same host; you’d do the following:

[*<Host name="www.company.com"*](http://www.company.com/) *...>*

[*<Alias>www.company.org</Alias>*](http://www.company.org/)

*</Host>*

**Understanding Authentication and the tomcat-users. xml File**

Tomcat’s user database realm uses the tomcat-users.xml file by default and reads the entire file into memory. Once the realm has loaded the file into memory, no modification to the *tomcat-users.xml* file will be reflected until the next server restart. Here’s tomcat-users.xml:

*<?xml version='1.0' encoding='utf-8'?>*

*<tomcat-users>*

*<role rolename="tomcat"/>*

*<role rolename="role1"/>*

*<user username="tomcat" password="tomcat" roles="tomcat"/>*

*<user username="both" password="tomcat" roles="tomcat,role1"/>*

*<user username="role1" password="tomcat" roles="role1"/>*

*</tomcat-users>*

Each role that a user can play is defined with a <role> element, and each user has a *<user>* entry. Note that a user can have more than one role by adding a comma-separated list to the roles attribute.

**Configuring Web Application Defaults with web.xml**

Every Servlet 2.4 web application must contain a web.xml deployment descriptor. This file must be placed in the WEB-INF directory of the web application.

However, Tomcat comes with a default web.xml in CATALINA\_HOME/conf. This file is similar to a web application’s web.xml file but is used to specify the default properties for all web applications that are running within this server instance.

To gain an understanding of what you can do with this file, let’s look at it. The file starts with the standard XML header and a reference to a DTD. Unlike server.xml, web.xml can be formally validated against a corresponding DTD.

*<?xml version="1.0" encoding="ISO-8859-1"?>*

[*<Web-app xmlns="http://java.sun.com/xml/ns/j2ee"*](http://java.sun.com/xml/ns/j2ee)[*xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"*](http://www.w3.org/2001/XMLSchema-instance)

[*xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee*](http://java.sun.com/xml/ns/j2ee)

[*http://java.sun.com/xml/ns/j2ee/Web-app\_2\_4.xsd"*](http://java.sun.com/xml/ns/j2ee/Web-app_2_4.xsd) *version="2.4">*

The most noteworthy thing about this is that the default web application version is 2.4.

**Default Servlet Definitions**

The default servlet that invokes any resources not mapped to any other servlet, either in this web.xml file or in an application’s web.xml file, is defined in the first *<servlet>* definition. This includes all static resources. You’ll see the mappings later.

*<web-app>*

*<servlet>*

*<servlet-name>default</servlet-name>*

*<servlet-class> org.apache.catalina.servlets.DefaultServlet*

*</servlet-class>*

*<init-param>*

*<param-name>debug</param-name>*

*<param-value>0</param-value>*

*</init-param>*

*<init-param>*

*<param-name>listings</param-name>*

*<param-value>true</param-value>*

*</init-param>*

*<load-on-startup>1</load-on-startup>*

*</servlet>*

Next comes the invoker servlet, which loads and executes anonymous servlets directly using the servlet’s filename. This mechanism is inherently unsafe because any class that exists in Tomcat’s classpath can be invoked in this way, so the invoker servlet has been commented out of recent versions of Tomcat.

*<!--*

*<servlet>*

*<servlet-name>invoker</servlet-name>*

*<servlet-class> org.apache.catalina.servlets.InvokerServlet*

*</servlet-class>*

*<init-param>*

*<param-name>debug</param-name>*

*<param-value>0</param-value>*

*</init-param>*

*<load-on-startup>2</load-on-startup>*

*</servlet>*

*-->*

Just as servlets have their default servlet, JSP pages have a servlet that compiles them into servlets and executes them.

*<servlet>*

*<servlet-name>jsp</servlet-name>*

*<servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>*

*<init-param>*

*<param-name>fork</param-name>*

*<param-value>false</param-value>*

*</init-param>*

*<init-param>*

*<param-name>xpoweredBy</param-name>*

*<param-value>false</param-value>*

*</init-param>*

*<load-on-startup>3</load-on-startup>*

*</servlet>*

The next set of servlets is commented out by default. You should uncomment them if you plan to add Apache-style Server Side Include (SSI) features to the stand-alone Tomcat server or process CGI. You’ll see more of this in later chapters.

**Matching URLs: Servlet Mappings**

Servlet mappings specify which servlets are to process incoming requests, as defined by the request URL.

*<!-- The mapping for the default servlet -->*

*<servlet-mapping>*

*<servlet-name>default</servlet-name>*

*<url-pattern>/</url-pattern>*

*</servlet-mapping>*

The previous *<servlet-mapping>* element maps the pattern / to the default servlet defined earlier in web.xml. So, [www.apress.com/tomcat/](http://www.apress.com/tomcat/) will map to the default servlet, which will process the request.

The second <servlet-mapping> maps all requests that end in /servlet/\* to the invoker servlet defined earlier in web.xml.

*<!-- The mapping for the invoker servlet -->*

*<!--*

*<servlet-mapping>*

*<servlet-name>invoker</servlet-name>*

*<url-pattern>/servlet/\*</url-pattern>*

*</servlet-mapping>*

*-->*

The next <servlet-mapping> specifies that all URLs containing \*.jsp and \*.jspx should be passed to the servlet named jsp for processing:

<!-- The mapping for the JSP servlet -->

<servlet-mapping>

<servlet-name>jsp</servlet-name>

<url-pattern>\*.jsp</url-pattern>

</servlet-mapping>

<servlet-mapping>

<servlet-name>jsp</servlet-name>

<url-pattern>\*.jspx</url-pattern>

</servlet-mapping>

**Configuring Session Timeout**

The *<session-config>* element configures how long Tomcat will maintain a session on the server side on behalf of a client. For example, if the user leaves a service registration transaction in the middle and doesn’t return to the cart for 30 minutes, all of that user’s information will be lost.

You must be careful to balance the <session-timeout> value with the potential of overloading the server with too many stale sessions.

*<session-config>*

*<session-timeout>30</session-timeout>*

*</session-config>*

**Configuring MIME Mappings**

The *<mime-mapping>* elements that make up a large chunk of web.xml help Tomcat serve static files with specific extensions to the client. It will generate an HTTP Content-Type header when transmitting the file to the client. Most browsers will use a helper application to process the file being transmitted if it recognizes the Content-Type specified. For example, a browser may start Adobe Acrobat when it detects the application/pdf content type.

*<mime-mapping>*

*<extension>abs</extension>*

*<mime-type>audio/x-mpeg</mime-type>*

*</mime-mapping>*

*... and so on ...*

**Configuring Welcome Files**

To be compatible with the default behavior of most modern web servers, including Apache, the default servlet will display a welcome file if the incoming URI is terminated in /—for example, [www.apress.com/.](http://www.apress.com/)

The default servlet will examine the root directory of the named virtual host and look for index.html, index.htm, or index.jsp in turn to be displayed. Each web application may override this list in its own deployment descriptor file.

<welcome-file-list>

<welcome-file>index.html</welcome-file>

<welcome-file>index.htm</welcome-file>

<welcome-file>index.jsp</welcome-file>

</welcome-file-list>

</web-app>

**Deploying Servlet and JSP Web Applications in Tomcat**

Now that you’ve got Tomcat installed, you will invariably need to deploy web applications. This chapter shows you web applications composed of servlets, JSPs, and other files, and several approaches for deploying them. It ends with a discussion of the Manager web application, which can handle some deployment operations for you.

Before Java servlets, web applications were mostly written in C/C++ or Perl. Usually they were made up mainly of static HTML pages and a few CGI scripts\* to generate the dynamic content portions of the web application. Those CGI scripts could be written in a platform-independent way, although they didn’t need to be (and for that reason the scripts often weren’t). Also, because 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.

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 preconfigured 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 modules—they’re specific to the server you write them for and probably won’t work on any other web server implementation.

Common Gateway Interface (CGI), an older standard for hooking up web servers to custom web application code, was meant for scripting dynamic content. Thus, it’s commonly referred to as “CGI scripting,” even though it’s possible to write a CGI program in C (which we don’t usually call a script).

Every HTTP request to a CGI script means that 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 to fulfilling HTTP requests.

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 JSP or servlet-basedweb application and move it from one servlet container to another, from one computer archi- tecture to another, and run it without any change (in fact, without even recompiling any of its code in almost all cases).

Servlet web applications are also designed to be relocateable. That is, you can write your webapps so that you can remap their content to a different URI on a host with- out rewriting anything inside the application itself, including the dynamic content. For example, Tomcat 7 comes with some example webapps including a couple called examples and docs. The default configuration maps these webapps to <http://yourhost/examples> and <http://yourhost/tomcat-docs>, respectively. By changing nothing other than some configuration lines in Tomcat’s server.xml file, you could remap these webapps to different URIs on the same host. This makes it easy to create modular portions of web sites that are easily moved around and also reused across multiple web sites (and potentially even within the same web site).

Tomcat’s configuration has always referredto webapps as “contexts.” Tomcat’s main configuration file, server.xml, has an XML element named Context that represents a webapp’s configuration. For each explicitly configured webapp, there should be one context element either in server.xml or in a separate context XML fragment file.

Which webapp deployment options you should use depends mainly on your use case(s). Are you a developer, running your own Tomcat instance on your own machine, and you want to deploy to it repeatedly as you develop? Are you a system administrator deploying only to a production Tomcat instance on another machine? The good news is that Tomcat supports so many methods of deployment that your use case is supported.

The Manager web application (detailed later in this chapter) offers the most flexible set of features for deployment in Tomcat. It allows for local and remote deployment, does not require any Tomcat restarts, and integrates nicely with the Apache Ant build tool. We suggest using the Manager for deployment, over copying WAR files or webapp directories “by hand.” But, if you are a more advanced Tomcat user, deploying to a Tomcat instance on the same machine, and you’re trying to use only the bare essentials, the Manager webapp is not necessary. It may be best to copy your webapp directory’s content or WAR file to where Tomcat can automatically find and deploy it.

Be careful not to confuse deployment with webapp reloading. Deployment is when you are installing and configuring your web application in Tomcat for the first time. Redeployment is when you have already deployed your webapp, and you want to stop the webapp so that it is not running anymore and redeploy its files (which could be completely different from the first set of files). But “reloading,” in Tomcat lingo, is something else entirely. Reloading is when the Tomcat servlet container is watching your webapp’s web.xml file, Java .jar files, and class files (and any other WatchedResource), and reloads them, picking up any changes. You may specify a list of any set of files in your webapp that will trigger a webapp reload if Tomcat detects that one of them has been modified.

In development, it is best to configure your context to be reloadable so that when you modify a class file, Tomcat will notice the change right away and reload the class. Depending on the size of your webapp, this will usually be faster than stop- ping the entire webapp, redeploying it, and restarting it. In production, however, we suggest turning off context reloading; your webapp will run faster as Tomcat does not need to continually check to see if any of the watched resources changed.

**Hosts**

In order to deploy any webapps into Tomcat, you must deploy them under a Host. A host represents a fully qualified domain name or IP address, such as groovywigs.com, for example. The stock Tomcat server.xml configuration file has a default host named localhost. The fact that this Host is the default Host as well as the only Host means that all HTTP requests entering Tomcat will be mapped to this Host, regardless of what host name is specified in the HTTP requests. For example, if the Host header in an incoming HTTP request says groovywigs.com as the host that the request is destined for, it won’t be a match for the only Host name that Tomcat knows about (localhost), so Tomcat will instead map it to the default Host: the same one named localhost.

For example, if you create a webapp for a web site named groovywigs.com,\* the webapp itself will probably be the root webapp of that site. There are at least a few ways of deploying the

webapp, but let’s say that you want to deploy it as an unpacked webapp directory (all of the

webapp’s content resides within one outer- most directory) named ROOT. You could deploy that as webapps/ROOT. In this case, the host’s name is groovywigs.com.

To deploy the webapp into the groovywigs.com host, you must already have configured Tomcat for the groovywigs.com host. This is pretty easy to do. Edit your server.xml file and find the spot where the first <Host> XML element is defined. Then, add a new <Host> element above it, like this:

*<Host name="groovywigs.com" appBase="webapps" unpackWARs="true" autoDeploy="true" xmlValidation="false" xmlNamespaceAware="false">*

*<!-- Context elements for the groovywigs.com host go here. -->*

*</Host>*

*<!--* *Define the default virtual host*

*Note: XML Schema validation will not work with Xerces 2.2.*

*-->*

*<Host name="localhost" appBase="webapps" unpackWARs="true" autoDeploy="true" xmlValidation="false"* *xmlNamespaceAware="false">*

And, if your Tomcat primarily serves requests for the groovywigs.com host, you should also change your <Engine>’s default host, also in server.xml:

*<!-- Define the top level container in our container hierarchy -->*

*<Engine name="Catalina" defaultHost="groovywigs.com">*

If the groovywigs.com host is but one of many hosts that your Tomcat will serve requests for, you should keep the default setting of *defaultHost="localhost".*

You should configure your hosts and your default host in server.xml before deploying webapps so that you can deploy your webapp(s) into the right host. This is the way we recommend adding a new host.

Tomcat also supports deploying and undeploying webapps while Tomcat is running, without requiring a restart of Tomcat, which is known in the industry as “hot deployment.” You can deploy and/or undeploy any number of webapps into Tomcat without restarting it as long as you turn on the feature Tomcat calls *autoDeploy* on one or more Hosts. In this method of deployment, Tomcat looks to see if you are configuring a Context for the webapp being hot deployed. If so, it will use the Context you supplied, and if not, Tomcat will create a default one for you. Locally, Tomcat allows you to do this by setting *autoDeploy="true"* on your Host. If you want hot deployment, you probably don’t want Tomcat to deploy your webapp at Tomcat star- tup time in addition to hot deploying it, so you should set *deployOnStartup="false"* on the Host as well. If you don’t explicitly set deployOnStartup to false, your webapps will each be deployed twice: once for the “on startup” deployment and a second time for the hot deployment. Setting these attributes requires one Tomcat restart if you are editing server.xml but does not require a restart if you set them only in memory using the Host Manager webapp, which is detailed in the next section. Then, just copy the webapp’s unpacked directory or WAR file into the Host’s *appBase* directory, while Tomcat is running, and the webapp will be deployed.

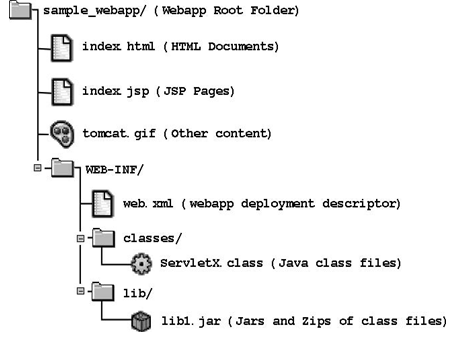
Your web application may be one of two forms when you deploy it into Tomcat: an unpacked webapp directory or a WAR file. We suggest deploying it as an unpacked webapp directory in most use cases because if the webapp is deployed unpacked so that the class files, JSP files, XML files, etc. are all individual files, it is easier to diag- nose any problems with the webapp. You can get on the server and inspect individual resources, and also move individual resources around and/or modify them in place if you want to, without needing to restart the web application in many cases. You can also watch modification timestamps of each file of your webapp individually. For those who work on webapps where local shell user security is a big issue, it may be best to deploy your webapp as a WAR file so that you have only one file to watch for malicious modification. A malicious user could modify the files in a WAR file as well, though, and an administrator of the machine would probably only be able to detect that if the administrator is routinely checking the checksum of the WAR file (and the checksum program isn’t maliciously modified as well). In the most common use cases, it ends up not being helpful to deploy a webapp as a WAR file.

**Layout of a Web Applica****tion**

Tomcat provides an implementation of both the servlet and JSP specifications. These specifications are in turn part of Sun’s Java Enterprise Edition. Java EE is designed to let application developers move their applications from one compliant application server (a program that implements the Java EE specification) to another, without significant rewriting or revising. To accomplish this, applications are packaged in very specific, portable ways; for example, as web application archives or enterprise application archives.

The Java Servlet Specification defines the Web Application aRchive (WAR) file format and its file structure for this very purpose. For your webapp to be application-server- implementation-independent, your files must follow certain conventions, such as the directory layout for storing web

pages, configuration files, and so on.



*Figure 4-19. Servlet/JSP web application file layout*

As a concrete example, Acme Widgets’ site might look like [Example 4-20.](#_bookmark532)

*Example 4-20. Example* *web application file layout*

*/*

*/index.jsp*

*/products.jsp*

*/widgets/index.html*

*/widgets/pricing.jsp*

*/images/logo.png*

*/WEB-INF/web.xml*

*/WEB-INF/classes/com/acme/PriceServlet.class*

*/WEB-INF/classes/DataHelper.class*

*/WEB-INF/lib/acme-util.jar*

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, except the WEB-INF or META-INF directory trees. Images often go in a /images subdirectory, though this is a convention, not a requirement. The WEB-INF directory has several specific pieces of con- tent. First, the classes directory is where you place any Java class files that are not in a JAR file, whether they are servlets or other class files used by a servlet, JSP, or other part of your webapp’s code. Second, the lib directory is where you put any JAR files containing packages of classes. And finally, the web.xml file is known as a deployment descriptor, which contains configuration for the web application, a description of the webapp, and any additional customization.

One of the nice things about the notion of putting per-site customizations into an XML file (the deployment descriptor) in a site’s directory, compared with the way other web servers tend to do things, is that the customizations for each site are stored with that site’s deployment. This makes it easier for maintenance and also makes it easy to package up the files from one site to move them to another server or even to a different ISP. Additionally, the contents of the WEB-INF and META-INF directories are automatically protected from access by client web browsers, so this configuration information (which may contain account names and passwords) is safe from client view.

**Deploying Servlets and JavaServer Pages**

You can configure the URI to which a servlet 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 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-speicific init-params*

*</description>*

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

*<init-param>*

*<param-name>myParm</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>*

You may also want to add JSPs to your webapp. JSPs can be installed anywhere in a web application (except under WEB-INF; this folder is protected against access from the web because it may contain initialization parameters, such as database connections, names, and passwords). JSPs can simply 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.

**Deploying an Unpacked Webapp Directory**

One way of deploying a web application into Tomcat is to deploy it as a directory of webapp content that is not packed into a WAR file. If you deploy your webapp as an unpacked directory, you won’t need to pack it into a WAR file at all—you may go straight to deploying it once you have your webapp’s content organized, as shown earlier.

There are two ways to configure Tomcat to recognize and start your web application, when you first deploy it as an unpacked webapp directory:

*server.xml context deployment*

Add a Context element to the server.xml file and restart Tomcat.

*Context XML fragment file deployment*

Add a new context XML fragment file in Tomcat’s CATALINA\_HOME/conf/ [EngineName]/[Hostname] directory tree, or create it as your web application’s WEB-INF/context.xml file relative to the root directory of your web application, and then restart Tomcat.

For any given webapp, you should choose just one of the deployment configuration methods. In the next sections, we show you details about each deployment method.

**Deploying a WAR File**

Another major way of deploying a web application into Tomcat is to deploy the application packed into a WAR file. WAR files are described in detail in the Java Servlet Specification.

With Tomcat, when you deploy your WAR file, you must decide whether to serve your webapp after unpacking the WAR file or while it is still packed into a WAR file. Both ways are supported. By default, when Tomcat deploys a WAR file, the first thing it does is unpack the contents of the WAR file into a directory of the same name minus the .war extension, and then serves the files from the unpacked directory.

For example, if your WAR file is named *suitcase.war*, Tomcat would unpack the contents of suitcase.war into a directory named suitcase, and then the files that are served as part of the webapp will be read from the suitcase directory on disk, not from the WAR file. You may turn off the automatic unpacking behavior by setting *unpackWARs="false"* on your Host element in server.xml. With it set to false, Tomcat will serve your webapp’s files right from the packed WAR file itself.

There are two ways to configure Tomcat to recognize andstart your web application when you first deploy it as a WAR file:

*server.xml context deployment*

Add a *<Context>* element to the server.xml file and restart Tomcat.

*Context XML fragment file deployment*

Add a new context XML fragment file in Tomcat’s CATALINA\_HOME/conf/ [EngineName]/[Hostname] directory tree, or create it as your web application’s WEB-INF/context.xml file relative to the root directory of your web application, then restart Tomcat.

For any given webapp, you should choose just one of these deployment configuration methods. In the next sections, we show you details about each deployment method.