CHAPTER 2

Configuring Tomcat

Once Tomcat is up and running, you will want to keep an eye on it to help it along occasionally. Troubleshooting application servers can be intimidating. In this chapter, we show you the various places to look for information about your server, how to find out why things aren't working, and give some examples of common mistakes in setting up and configuring Tomcat. We round out this chapter with some ideas on performance tuning the underlying Java runtime environment and the Tomcat server itself. Finally, we discuss the Tomcat administration web application, a tool for helping you with the task of keeping Tomcat running.

Using the Apache Web Server

You can use Tomcat as a standalone web server and servlet container, or you can use it as an add-on servlet container for a separate web server. Both are common, and each is appropriate in certain situations.

The Tomcat authors have spent quite a bit of time and effort to make Tomcat run efficiently as a standalone web server; as a result, it is easy to set up and run a web site without worrying about connecting Tomcat to a third-party web server. Tomcat's built-in web server is a highly efficient HTTP 1.1 server that is quite fast at serving static content once it is configured correctly for the computer on which it runs. They've also added features to Tomcat that one would expect from full-featured web servers, such as Common Gateway Interface (CGI) scripting, a Server-Side Includes (SSI) dynamic page interpreter, a home directory mapper, and more.

The Tomcat authors also realized that many companies and other organizations already run the Apache *httpd* web server and might not want to switch from that server to Tomcat's built-in web server. The Apache web server is the number one web server on the Internet, and it is arguably the most flexible, fully featured, and supported web server ever written. Even if someone running Apache *httpd* wanted to switch web servers, it might be difficult for them to do so because their web sites are often already too integrated with Apache's features. Also, it's difficult for other web servers to keep up with Apache efficiency, since it's been tuned for performance in so many different ways over the years.

With these issues in mind, if you're still considering using Apache *httpd* and Tomcat together, refer to Chapter 5 for an in-depth look at how to hook these two programs together.

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.

In this and future sections, the term *context* is used interchangeably with web application. A context is the technical term used within Tomcat for a web application, and it has a corresponding set of XML elements and attributes that define it in Tomcat's *server.xml* file.

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`, `JNDIRealm`, and `JAASRealm`. Java developers can also create additional realm implementations to interface with

[*] See "server.xml" in Chapter 7 for a detailed explanation of Tomcat's main configuration file's contents.

[†] See "security-constraint" in Chapter 7 for a description of this element.

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... -->
/>
```

`Realm` configurations can be overridden by subsequent realm configurations. Suppose one `Realm` is configured in a top-level element of your *server.xml* file. Also suppose a second `Realm` is defined within one of your `Host` elements. The second `Realm` configuration is the one that is used for the `Host` that contains it, but all other `Hosts` use the first `Realm`.

No part of Tomcat's `Realm` API is used for adding or removing users—it's just not part of the `Realm` interface. To add users to or remove users from a realm, you're on your own unless the realm implementation you decide to use happens to implement those features.

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.

If you change the *tomcat-users.xml* file without restarting Tomcat, Tomcat will *not* reread the file until the server is restarted.

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 in Example 2-1.

Example 2-1. Distribution version of tomcat-users.xml

```
<!--
  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.
-->
```

Example 2-1. Distribution version of tomcat-users.xml (continued)

```
<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.

`UserDatabaseRealms` are not really intended for serious production work, as the only way to update them is to write a custom servlet that accesses the realm via JNDI. The servlet would then need to modify the user database in memory or the *tomcat-users.xml* file on disk. Finally, Tomcat would have to be restarted to utilize these changes. Additionally, storing passwords in clear text (such as in the *tomcat-users.xml* file), even on a secure filesystem, is frowned upon by the security-conscious.

JDBCRealm

The `JDBCRealm` provides substantially more flexibility than a `UserDatabaseRealm`, as well as dynamic access to data. It is essentially a realm backed by a relational database; users, passwords, and roles are stored in that database, and `JDBCRealm` accesses them as often as needed. If your existing administrative software adds an account to a relational database table, for example, the `JDBCRealm` will be able to access it immediately. You need to specify the JDBC connection parameters as attributes for the realm in your *server.xml* file. Example 2-2 is a simple example of a `JDBCRealm` for a news portal site named JabaDot.

Example 2-2. JDBCRealm example

```
<!-- Set up a JDBC Real for JabaDot user database -->
<Realm className="org.apache.catalina.realm.JDBCRealm"
        <!-- How to find the jabadot database (jabadb) -->
        driverName="org.postgresql.Driver"
        connectionURL="jdbc:postgresql:jabadot"
        connectionName="system"
        connectionPassword="something top secret"
        <!-- userdb and its fields -->
        userTable="users" userCredCol="passwd"
        <!-- roles table and its fields -->
```

Example 2-2. JDBCRealm example (continued)

```
        userRoleTable="controls" roleNameCol="roles"
        <!-- Name of the username column in both tables -->
        userNameCol="nick"
/>
```

Table 2-1 lists the allowed attributes for a `Realm` element using the `JDBCRealm` implementation.

Table 2-1. JDBCRealm attributes

Attribute	Meaning
`className`	The Java class name of this realm implementation; must be `org.apache.catalina.realm.JDBCRealm` for `JDBCRealms`.
`connectionName`	The database username used to establish a JDBC connection.
`connectionPassword`	The database password used to establish a JDBC connection.
`connectionURL`	The database URL used to establish a JDBC connection.
`debug`	Debugging level, where `0` is none, and positive numbers result in increasing detail. The default is `0`.
`digest`	Digest algorithm (SHA, MD2, or MD5 only). The default is `"cleartext"`.
`driverName`	The Java class name of the JDBC driver.
`roleNameCol`	The name of the column in the roles table that has role names (for assigning to users).
`userNameCol`	The name of the column in the users and roles tables listing usernames.
`userCredCol`	The name of the column in the users table listing users' passwords.
`userRoleTable`	The name of the table for mapping roles to users.
`userTable`	The name of the table listing users and passwords.

JNDIRealm

If you need Tomcat to retrieve usernames, passwords, and roles from an LDAP directory, `JNDIRealm` is for you. `JNDIRealm` is a very flexible realm implementation—it allows you to authenticate users against your LDAP directory of usernames, passwords, and roles, while allowing many different schema layouts for that data. `JNDIRealm` can recursively search an LDAP hierarchy of entries until it finds the information it needs, or you can configure it to look in a specific location in the directory server for the information. You can store your passwords as clear text and use the basic authentication method, or you can store them in digest-encoded form and use the digest authentication method (both authentication methods are discussed in the following section).

Here's an example of a `JNDIRealm` configured to use an LDAP server:

```
<Realm className="org.apache.catalina.realm.JNDIRealm" debug="99"
  connectionURL="ldap://ldap.groovywigs.com:389"
  userPattern="uid={0},ou=people,dc=groovywigs,dc=com"
```

```
    roleBase="ou=groups,dc=groovywigs,dc=com"
    roleName="cn"
    roleSearch="(uniqueMember={0})"
  />
```

Table 2-2 lists `JNDIRealm`'s allowed attributes for its `Realm` element in a *server.xml* file.

Table 2-2. JNDIRealm attributes

Attribute	Meaning
`className`	The Java class name of this realm implementation; must be `org.apache.catalina.realm.JNDIRealm` for `JNDIRealm`s.
`connectionName`	The username used to authenticate a read-only LDAP connection. If left unset, an anonymous connection will be made.
`connectionPassword`	The password used to establish a read-only LDAP connection.
`connectionURL`	The directory URL used to establish an LDAP connection.
`contextFactory`	The fully qualified Java class name of the JNDI context factory to be used for this connection. If left unset, the default JNDI LDAP provider class is used.
`debug`	Debugging level, where `0` is none, and positive numbers result in increasing detail. The default is `0`.
`digest`	Digest algorithm (SHA, MD2, or MD5 only). The default is `"cleartext"`.
`roleBase`	The base LDAP directory entry for looking up role information. If left unspecified, the default is to use the top-level element in the directory context.
`roleName`	The attribute name that the realm should search for role names. You may use this in conjunction with the `userRoleName` attribute. If left unspecified, roles are taken only from the user's directory entry.
`roleSearch`	The LDAP filter expression used for performing role searches. Conforms to the syntax supported by `java.text.MessageFormat`. Use `{0}` to substitute the distinguished name (DN) of the user, and/or `{1}` to substitute the username. If left unspecified, roles are taken only from the attribute in the user's directory entry specified by `userRoleName`.
`roleSubtree`	Set to `true` if you want to recursively search the subtree of the element specified in the `roleBase` attribute for roles associated with a user. If left unspecified, the default value of `false` causes only the top level to be searched (a nonrecursive search).
`userBase`	Specifies the base element for user searches performed using the `userSearch` expression. If left unspecified, the top-level element in the directory context will be used. This attribute is ignored if you are using the `userPattern` expression.
`userPassword`	The name of the attribute in the user's directory entry containing the user's password. If you specify this value, the `JNDIRealm` will bind to the directory using the values specified by the `connectionName` and `connectionPassword` attributes, and retrieve the corresponding password attribute from the directory server for comparison to the value specified by the user being authenticated. If the `digest` attribute is set, the specified digest algorithm is applied to the password offered by the user before comparing it with the value retrieved from the directory server. If left unset, `JNDIRealm` will attempt a simple bind to the directory using the DN of the user's directory entry and password specified by the user, with a successful bind being interpreted as a successful user authentication.
`userPattern`	A pattern for the distinguished name (DN) of the user's directory entry, conforming to the syntax of `java.text.MessageFormat`, with `{0}` marking where the actual username will be inserted.

Table 2-2. JNDIRealm attributes (continued)

Attribute	Meaning
`userRoleName`	The name of an attribute in the user's directory entry containing values for the names of roles associated with this user. You may use this in conjunction with the `roleName` attribute. If left unspecified, all roles for a user derive from the role search.
`userSearch`	The LDAP filter expression to use when searching for a user's directory entry, with `{0}` marking where the actual username will be inserted. Use this attribute (along with the `userBase` and `userSubtree` attributes) instead of `userPattern` to search the directory for the user's directory entry.
`userSubtree`	Set this value to `true` if you want to recursively search the subtree of the element specified by the `userBase` attribute for the user's directory entry. The default value of `false` causes only the top level to be searched (a nonrecursive search). This is ignored if you are using the `userPattern` expression.

JAASRealm

`JAASRealm` is an experimental realm implementation that authenticates users via the Java Authentication and Authorization Service (JAAS). JAAS implements a version of the standard Pluggable Authentication Module (PAM) framework, which allows applications to remain independent from the authentication implementation. New or updated authentication implementations can be plugged into an application (Tomcat, in this case) without requiring modifications to the application itself—it requires only a small configuration change. For example, you could use `JAASRealm` configured to authenticate users against your Unix users/passwords/groups database, and then reconfigure it to authenticate against Kerberos by simply changing the configuration, rather than the entire realm implementation. Additionally, JAAS allows stacking authentication modules, so that two or more authentication modules can be used in conjunction with each other in an authentication stack. Stacking the pluggable authentication modules allows for highly customized authentication logic that Tomcat doesn't implement on its own.

JAAS can be used with either Java 1.3 or Java 1.4. If you use JDK/JRE 1.4, JAAS is built into the JVM. If you need to use Java 1.3, you must download JAAS as a Java Standard Extension package (from *http://java.sun.com/products/jaas/index-10.html*) and install it into your JVM. Doing that, though, means that you get an older version of JAAS that doesn't come with any native code libraries for your operating system (unlike JDK/JRE 1.4, which does include some native code libraries). This is similar to the version of JAAS that is included with the Tomcat classes, within the *jaas.jar* archive.

If you download the standard extension without the native libraries (or use Tomcat's version of JAAS), you may run into some limitations. For example, one major problem with using JAAS for authentication through the Unix users and groups database is that JAAS relies on a native code library to validate a user's password. The *jaas.jar* file that comes with Tomcat contains the pure Java classes that make up JAAS itself, but no native libraries—you must download those separately from Sun or from another source.

We were only able to find optional JAAS native login modules for SPARC Solaris, x86 Solaris, and Windows, and even then there was almost no documentation about what functionality was included. Even with the version of JAAS included in JDK 1.4, we were unable to get `JAASRealm` to validate Unix user passwords on Solaris or Linux. At the time of this writing, this realm implementation does not seem to work, but it could be fixed by the time you read this. In the interests of completeness, Table 2-3 lists the supported `Realm` attributes for a `JAASRealm` implementation. These should remain the same when `JAASRealm` is working as documented.

Table 2-3. JAASRealm attributes

Attribute	Meaning
`className`	The Java class name of this realm implementation; must be `org.apache.catalina.realm.JAASRealm` for `JAASRealm`s.
`debug`	Debugging level, where `0` is none, and positive numbers result in increasing detail. The default is `0`.
`digest`	Digest algorithm (SHA, MD2, or MD5 only). The default is `"cleartext"`. This attribute is inherited from the base `Realm` implementation and may not work with `JAASRealm`, depending on the underlying authentication method being used.
`appName`	Identifies the application name that is passed to the JAAS `LoginContext` constructor (and therefore picks the relevant set of login methods based on your JAAS configuration). This defaults to `"Tomcat"`, but you can set it to anything you like as long as you change the corresponding name in your JAAS *.java.login.config* file.
`userClassNames`	Comma-delimited list of `javax.security.Principal` classes that represent individual users. For the `UnixLoginModule`, this should be set to include the `UnixPrincipal` class.
`roleClassNames`	Comma-delimited list of `javax.security.Principal` classes that represent security roles. For the `UnixLoginModule`, this should be set to include the `UnixNumericGroupPrincipal` class.

To try using `JAASRealm` configured for the `UnixLoginModule` on your box, install the `Realm` element as shown in Example 2-3 in your *server.xml* file, use the configuration from Example 2-4 in your web appliation's *web.xml* file, and add a *.java.login.conf* file with the contents shown in Example 2-5 in the root of your home directory. Depending on your JVM and JAAS setup, you might need to set the following environment variable before starting Tomcat so that JAAS finds its login configuration file:

```
# export JAVA_OPTS=\
'-Djava.security.auth.login.config=/root/.java.login.config'
```

The *.java.login.config* file can be stored anywhere, as long as you point to it with the above environment variable.

If your JVM isn't running as the `root` user, it will not be able to access user passwords (at least on typically configured machines). As the JVM running Tomcat is often configured to run as a `web` or `tomcat` user, this can cause a lot of trouble. You may find that running Tomcat under the `root` account is more trouble than the help that `JAASRealm` provides.

Once you start up Tomcat and make the first request to a protected resource, `JAASRealm` should read your */etc/passwd* and */etc/group* files, as well as interface with your OS to compare passwords, and be able to authenticate using that data.

So What Really Happens?

In our tests, we could get the pure Java `UnixLoginModule` and `JAASRealm` to see Unix usernames and numeric group IDs, but not to compare passwords. We also found the best supported authentication method (`auth-method` in the *web.xml* file) seems to be form authentication (FORM).

Even if Sun's JAAS `UnixLoginModule` and associated code doesn't work on your system, it may still be possible to write your own `LoginModule`, `Principal`, and associated JAAS implementations that do work. Doing so could yield you a stackable, pluggable authentication module system for use with Tomcat.

Example 2-3. A Realm configuration that uses JAASRealm to authenticate against the Unix users and groups database

```
<Realm className="org.apache.catalina.realm.JAASRealm"
       userClassNames="com.sun.security.auth.UnixPrincipal"
       roleClassNames="com.sun.security.auth.UnixNumericGroupPrincipal"
       debug="3"/>
```

Example 2-4. A web.xml snippet showing security-constraint, login-config, and security-role elements configured for JAASRealm

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Entire Application</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
     <role-name>0</role-name>
   </auth-constraint>
</security-constraint>

<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>

<security-role>
  <role-name>0</role-name>
</security-role>
```

Example 2-5. The complete contents of a JAAS .java.login.conf file that is stored in the home directory of the user who runs Tomcat

```
Tomcat {
    com.sun.security.auth.module.UnixLoginModule required debug=true;
};
```

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.

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.

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

Basic authentication

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.

While Tomcat's basic authentication is reliant upon HTTP basic authentication, the two are not synonymous. In this book, *basic authentication* refers to Tomcat's container-managed security scheme; references to HTTP basic authentication are specifically noted.

With this authentication method, all passwords are sent across the network in base64-encoded text.

Using basic authentication is generally considered a security flaw, unless the site also uses HTTPS or some other form of encryption between the client and the server (for instance, a virtual private network). Without this extra encryption, network monitors can intercept (and misuse) users' passwords.

Example 2-6 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.

Example 2-6. Club site with members-only subdirectory

```
<!--
  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>
```

For a complete listing of the elements in the *web.xml* descriptor and their meanings, refer to Chapter 7.

Digest authentication

Digest authentication (indicated by a value of `DIGEST` in the *web.xml* file's `auth-method` element) is a nice alternative to basic authentication because it sends passwords across the network in a more strongly encoded form and stores passwords on disk that way as well. The main disadvantage to using digest authentication is that some web browser versions do not support it. We could find no definitive list of web browser versions that are known to support or not support digest authentication on the Web. Therefore, before you decide to use only digest authentication for your web application, we highly suggest that you test each browser brand and version that you intend to support.

To use the container-managed digest authentication, use a `security-constraint` element along with a `login-config` element like that shown in Example 2-7. Then, modify the `Realm` setting in your *server.xml* file to ensure that your passwords are stored in an encoded form.

At the time of this writing, container-managed digest authentication is broken in almost all of Tomcat 4.1's realm implementations. See Remy Maucherat's comments in bug number 9852 at *http://nagoya.apache.org/bugzilla/show_bug.cgi?id=9852* for more details.

Example 2-7. DIGEST authentication settings in the web.xml file

```
<!--
  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>

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

In your *server.xml*, add a `digest` attribute to your `Realm` element, as shown in Example 2-8. Give this attribute the value `"MD5"`. This tells Tomcat which encoding algorithm you wish to use to encode the passwords before they are written to disk. Possible values for the `digest` attribute include SHA, MD2, and MD5, but you should stick with MD5; that option is much better supported in the Tomcat codebase.

If you want Tomcat to log information about whether each digest authentication attempt succeeded, set the `debug` attribute of the realm to 2 (or to a higher number) in your *server.xml*.

Example 2-8. A UserDatabaseRealm configured to use the MD5 digest algorithm

```
<Realm className="org.apache.catalina.realm.UserDatabaseRealm"
       debug="0" resourceName="UserDatabase" digest="MD5"/>
```

In addition to telling Tomcat how the passwords will be stored, you need to manually encode each user's password in the specified format (in this case, MD5). This involves a two-step process that you must repeat with each user's password. At least in Tomcat 4, these steps are not automated.

First, run the following commands to encode the password with the MD5 algorithm:

```
jasonb$ cd $CATALINA_HOME
jasonb$ bin/digest.sh -a MD5 user-password

user-password:9a3729201fdd376c76ded01f986481b1
```

Substitute *user-password* with the password you're encoding. The output from the program is shown in the last line; it will echo back the supplied password and a colon, followed by the MD5-encoded password. It is this lengthy hexadecimal value that you are interested in.

Second, store the encoded password in your realm's password field for the appropriate user. For the `UserDatabaseRealm`, for example, just add a `user` element line in the *tomcat-users.xml* file, like this:

```
<?xml version='1.0'?>
<tomcat-users>
  <role rolename="tomcat"/>
  <role rolename="role1"/>
  <role rolename="member"/>
  <user username="jasonb"
        password="9a3729201fdd376c76ded01f986481b1"
        roles="member"/>
</tomcat-users>
```

When you're done encoding and storing the passwords, you must restart Tomcat.

Form authentication

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 in Examples 2-6 and 2-7, and a `login-config` element in your *web.xml* file like the one shown in Example 2-9.

Example 2-9. FORM authentication settings in the web.xml file

```
<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 Example 2-9 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. Example 2-10 shows a working example of the login page for a form-authentication setup.

Example 2-10. A sample HTML form login page to use with FORM logins

```
<html>
  <body>
    <center>

      <!-- Begin login form -->
      <form method="POST" action="j_security_check" name="loginForm">
        <table border="0" cellspacing="5">
          <tr>
            <td height="50">
              Please log in.
            </td>
          </tr>

          <!-- username password prompts fields layout -->
          <tr>
            <td>
              <table width="100%" border="0"
                     cellspacing="2" cellpadding="5">
                <tr>
                  <th align="right">
                    Username
                  </th>
                  <td align="left">
                    <input type="text" name="j_username" size="16"
                           maxlength="16"/>
                  </td>
                </tr>
                <p>
                <tr>
                  <th align="right">
                    Password
                  </th>
                  <td align="left">
                    <input type="password" name="j_password" size="16"
                           maxlength="16"/>
                  </td>
                </tr>

                <tr>
                  <td width="50%" valign="top"><div align="right" /></td>
                  <td width="55%" valign="top">&nbsp;</td>
                </tr>
```

Example 2-10. A sample HTML form login page to use with FORM logins (continued)

```
            <!-- login reset buttons layout -->
            <tr>
              <td width="50%" valign="top">
                <div align="right">
                  <input type="submit" value='Login'>&nbsp;&nbsp;
                </div>
              </td>
              <td width="55%" valign="top">
                &nbsp;&nbsp;<input type="reset" value='Reset'>
              </td>
            </tr>
          </table>
        </td>
      </tr>
    </table>
  </form>

  <!-- End login form -->
</center>

<script language="JavaScript" type="text/javascript">
  <!--
  // Focus the username field when the page loads in the browser.
  document.forms["loginForm"].elements["j_username"].focus( )
  // -->
</script>

  </body>
</html>
```

Example 2-11 is a simple error page for notifying the user of a failed login attempt.

Example 2-11. A sample HTML login error page to use with FORM logins

```
<html>
  <body>
    <center>

      <h2>
        Login failed.
        <br>
        Please try <a href="/">logging in again.</a>
      </h2>

    </center>
  </body>
</html>
```

Client-cert authentication

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 password—instead, 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. How the certificates are generated and stored is up to the administrators of the web site, but it's usually a manual process. 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. Modern web browsers can store any number of client certificates and can prompt the user for which certificates to send to the server. As this is a rather advanced and lengthy topic, we deal with the subject in full in Chapter 6 and show examples of how to use client-cert authentication with HTTPS.

Single Sign-On

Once you've set up your realm and method of authentication, you'll need to deal with the actual process of logging the user in. More often than not, logging into an application is a nuisance to an end user, and you will need to minimize the number of times they must authenticate. By default, each web application will ask the user to log in the first time the user requests a protected resource. This can seem like a hassle to your users if you run multiple web applications and each application asks the user to authenticate. Users cannot tell how many separate applications make up any single web site, so they won't know when they're making a request that crosses a context boundary and will wonder why they're being repeatedly asked to log in.

The "single sign-on" feature of Tomcat 4 allows a user to authenticate only once to access all of the web applications loaded under a virtual host (virtual hosts are described in detail in Chapter 7). To use this feature, you need only add a `SingleSignOn` valve element at the host level. This looks like the following:

```
<Valve
    className="org.apache.catalina.authenticator.SingleSignOn"
    debug="0"
/>
```

The Tomcat distribution's default *server.xml* contains a commented-out single sign-on `Valve` configuration example that you can uncomment and use. Then, any user who is considered valid in a context within the configured virtual host will be considered valid in all other contexts for that same host.

There are several important restrictions for using the single sign-on valve:

- The valve must be configured and nested within the same `Host` element that the web applications (represented by `Context` elements) are nested within.
- The `Realm` that contains the shared user information must be configured either at the level of the same `Host` or in an outer nesting.
- The `Realm` cannot be overridden at the `Context` level.
- The web applications that use single sign-on must use one of Tomcat's built-in authenticators (in the `auth-method` element of *web.xml*), rather than a custom

authenticator. The built-in methods are basic, digest, form, and client-cert authentication.

- If you're using single sign-on and wish to integrate another third-party web application into your web site, and the new web application uses only its own authentication code that doesn't use container-managed security, you're basically stuck. Your users will have to log in once for all of the web applications that use single sign-on, and then once again if they make a request to the new third-party web application. Of course, if you get the source and you're a developer, you could fix it, but that's probably not so easy to do.
- The single sign-on valve requires the use of HTTP cookies.

The Servlet Specification 2.3 standardizes the name `JSESSIONID` as the cookie name that stores a user's session ID. This session ID value is unique for each web application, even if the single sign-on valve is in use. The valve adds its own cookie named `JSESSIONIDSSO`, which is not part of the Servlet Specification 2.3 but must be present in order for Tomcat's single sign-on feature to work.

Controlling Sessions

An HTTP session is a series of interactions between a single browser instance and a web server. The servlet specification defines an `HttpSession` object that temporarily stores information about a user, including a unique session identifier and references to Java objects that the web application stores as attributes of the session. Typical uses of sessions include shopping carts and sites that require users to sign in. Usually, sessions are set to time out after a configurable period of user inactivity. Once a session has timed out, it is said to be an *invalid* session; if the user makes a new HTTP request to the site, a new, valid session must be created, usually through a relogin.

Tomcat 4 has pluggable session `Managers`[*] that control the logic of how sessions are handled, and it has session `Stores` to save and load sessions. Not every `Manager` uses a `Store` to persist sessions; it is an implementation option to use the `Store` interface to provide pluggable session store capabilities. Robust session `Managers` will implement some kind of persistent storage for their sessions, regardless of whether they use the `Store` interface. Specifying a `Manager` implementation works in a similar fashion to specifying a `Realm`:

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

[*] This `Manager` is an HTTP session manager. Do not confuse it with the Manager web application described in Chapter 3.

Almost all of the control over sessions is vested in the `Manager` and `Store` objects, but some options are set in *web.xml* (that is, at the context level). These options are described in detail in Chapter 7, under the "listener" and "session-config" element headings.

Session Persistence

Session persistence is the saving (persisting) to disk of HTTP sessions when the server is shut down and the corresponding reloading when the server is restarted. Without session persistence, a server restart will result in all active user sessions being lost. To users this means they will be asked to log in again (if you're using container-managed security), and that they may lose the web page they were on, along with any shopping cart information or other web page state information that was stored in the session. Persisting that information helps to ensure that it won't be lost.

Keep in mind that long-term servlet session persistence (longer than an hour or more) should never be a desirable goal, because it's not the place to put permanent user information. It's a temporary cache, not a storage location. Some reasons for this include:

- A user might change web browsers, and sessions are almost always tied to either a cookie that is stored in a browser or an SSL session that is open on only one web browser.
- Users who are actively using a site will likely make a request more often than once an hour; if their session is missing, they'll likely just consider the original one lost and create a new one.
- Sessions can and do time out eventually, invalidating the persisted session. Once reloaded, timed-out sessions are unusable and are simply garbage collected.

If you need a permanent place to store user information, you should store it in a relational database, LDAP directory, or in your own custom file store on disk.

For more detail about session persistence, see the book *Java Enterprise Best Practices* (O'Reilly).

StandardManager

`StandardManager` is the default `Manager` when none is explicitly configured in the *server.xml* file. `StandardManager` does not use any `Stores`. It has its own built-in code to persist all sessions to the filesystem, and it does so only when Tomcat is gracefully shut down. It serializes sessions to a file called *SESSIONS.ser*, located in the web application's work directory (look in the *$CATALINA_HOME/work/Standalone/<hostname>/<webapp name>/* directory). `StandardManager` reloads these sessions from the file when Tomcat restarts, and then deletes the file, so you won't find it on disk once Tomcat has completed its startup. Of course, if you terminate the server abruptly (e.g., *kill -9* on Unix, system crash, etc.), the sessions will all be lost because `StandardManager` won't get a chance to save them to disk. Table 2-4 shows the attributes of `StandardManager`.

Table 2-4. StandardManager attributes

Attribute	Meaning
`className`	The name of the `Manager` implementation to use. Must be set to `org.apache.catalina.session.StandardManager` for `StandardManagers`.
`checkInterval`	The session timeout check interval (in seconds) for this `Manager`. The default is `60`.
`maxActiveSessions`	The maximum number of active sessions allowed or `-1` for no limit, which is the default.
`maxInactiveInterval`	The default maximum inactive interval (in seconds) for sessions created by this `Manager`. The default is `60`.
`pathname`	The path or filename of the file to which this `Manager` saves active sessions when Tomcat stops, and from which these sessions are loaded when Tomcat starts. If left unset, this value defaults to *SESSIONS.ser*. Set it to an empty value to indicate that you do not desire persistence. If this pathname is relative, it will be resolved against the temporary working directory provided by the context, available via the `javax.servlet.context.tempdir` context attribute.
`debug`	Debugging level, where `0` is none, and positive numbers result in increasing detail. The default is `0`.
`algorithm`	The message digest algorithm that this `Manager` uses to generate session identifiers. Valid values include `SHA`, `MD2`, or `MD5`. The default is `MD5`.
`entropy`	You can set this to any string you want, and it will be used numerically to create a random number generator seed. The random number generator is used in conjunction with the digest algorithm to generate secure random session identifiers. The default is to use the string representation of the `Manager` class name.
`distributable`	If this flag is set to `true`, all user data objects added to sessions associated with this manager must implement `java.io.Serializable` because they may be serialized and sent to other computers running other Tomcat JVMs. The default is `false`, but this attribute is unused in `StandardManager`.
`randomClass`	The random number generator class name. The default is `java.security.SecureRandom`.

Here's an example of how to configure a `StandardManager` that times out sessions after two hours of inactivity:

```
<Manager className="org.apache.catalina.session.StandardManager"
         debug="0"  maxInactiveInterval="7200">
</Manager>
```

PersistentManager

Another `Manager` you can use is `PersistentManager`, which stores sessions to a session `Store`, and in doing so provides persistence in the event of unexpected crashes. `PersistentManager` is considered experimental, and Tomcat does not use it by default.

The class `org.apache.catalina.session.PersistentManager` implements full persistence management. It must be accompanied by a `Store` element telling where to save the sessions; supported locations include files and JDBC databases.

```
<Manager className="org.apache.catalina.session.PersistentManager"
        debug="0"  saveOnRestart="true">
  <Store className="org.apache.catalina.session.FileStore"/>
</Manager>
```

Table 2-5 shows the attributes of the `PersistentManager`.

Table 2-5. PersistentManager attributes

Attribute	Meaning
`className`	The name of the `Manager` class to use; must be set to `org.apache.catalina.session.PersistentManager` for `PersistentManagers`.
`checkInterval`	The session timeout check interval (in seconds) for this `Manager`. The default is `60`.
`maxActiveSessions`	The maximum number of active sessions allowed or `-1` for no limit, which is the default.
`maxIdleBackup`	How long (in seconds) a session must be idle before it should be backed up. `-1` means sessions won't be backed up (the default).
`maxIdleSwap`	The maximum time a session may be idle before it should be swapped to file. Setting this to `-1` means sessions should not be forced out (the default).
`minIdleSwap`	The minimum time a session must be idle before it is swapped to disk. This overrides `maxActiveSessions`, to prevent thrashing if there are lots of active sessions. Setting this to `-1` (the default) means to ignore this parameter.
`maxActiveSessions`	The maximum number of active sessions allowed or `-1` for no limit. If the configured maximum is exceeded, no more sessions can be created until one or more sessions are invalidated. The default is `-1`.
`saveOnRestart`	Whether to save and reload sessions when Tomcat is gracefully stopped and restarted. The default is `true`.
`maxInactiveInterval`	The default maximum inactive interval (in seconds) for sessions created by this `Manager`. The default is `60`.
`debug`	Debugging level, where `0` is none, and positive numbers result in increasing detail. The default is `0`.
`algorithm`	The message digest algorithm that this `Manager` uses to generate session identifiers. Valid values include `SHA`, `MD2`, or `MD5`. The default is `MD5`.
`entropy`	You can set this to any string you want, and it will be used numerically to create a random number generator seed. The random number generator is used in conjunction with the digest algorithm to generate secure random session identifiers. The default is to use the string representation of the `Manager` class name.
`distributable`	If this flag is set to `true`, all user data objects added to sessions associated with this manager must implement `java.io.Serializable` because they may be serialized and sent to other computers running other Tomcat JVMs. The default value is `false`.
`randomClass`	The random number generator class name. The default is `java.security.SecureRandom`.

As of this writing, Tomcat comes with only two Store implementations: FileStore and JDBCStore. They store session information to and retrieve session information from the filesystem and a relational database, respectively. Since StandardManager doesn't use Stores, the only Manager you can use with FileStore or JDBCStore that comes with Tomcat is PersistentManager.

Using FileStore for storing sessions

Here's an example of how you can configure PersistentManager to use FileStore in your *server.xml* file:

```
<Manager className="org.apache.catalina.session.PersistentManager"
         debug="0"  saveOnRestart="true">
  <Store className="org.apache.catalina.session.FileStore"
         directory="/home/jasonb/tomcat-sessions"/>
</Manager>
```

If you decide to set the directory attribute to a custom value, be sure to set it to a directory that exists and to which the user who runs Tomcat has read/write file permissions. Table 2-6 shows the allowed attributes for FileStore.[*]

Table 2-6. FileStore attributes

Attribute	Meaning
className	The name of the Store class to use; must be set to org.apache.catalina.session.FileStore for FileStores.
directory	The filesystem pathname of the directory in which sessions are stored. This can be an absolute pathname or a path that is relative to the temporary work directory for this web application.
checkInterval	The interval (in seconds) at which FileStore's background Thread checks for expired sessions. The default is 60.
debug	Debugging level, where 0 is none, and positive numbers result in increasing detail. The default is 0.

FileStore saves each user's session (including all session attribute objects) to the filesystem. Each session is saved in a file named *<session ID>.session*; for example, *4FF8890ED8A53D6B163A27382602B0EB.session*. FileStore will load and save these sessions whenever the PersistentManager asks it to. If a session is saved to disk when Tomcat is shut down and times out in the meantime (while Tomcat isn't running), FileStore will invalidate and remove it once Tomcat is running again.

Do not try to delete these sessions by hand while Tomcat is running—FileStore may subsequently try to load a session file you've deleted. This will result in a "No persisted data file found" message in the log file.

[*] As you probably have guessed, the Store element works exactly as the Realm and Manager elements do.

Using JDBCStore for storing sessions

Here's an example of how you can configure `PersistentManager` to use `JDBCStore` in your *server.xml* file:

```
<Manager className="org.apache.catalina.session.PersistentManager"
         debug="0"  saveOnRestart="true">
  <Store className="org.apache.catalina.session.JDBCStore"
         driverName="org.gjt.mm.mysql.Driver"
        connectionURL="jdbc:mysql://localhost:3306/mydb?user=jb&password=pw"
  />
</Manager>
```

`JDBCStore` must be able to log into the database as well as read and write to a session table, which you must set up in the database before you start Tomcat. A typical representative table setup is shown here:

```
create table tomcat$sessions (
  id           varchar(64) not null primary key,
  data         blob
  valid        char(1) not null,
  maxinactive  int not null,
  lastaccess   bigint not null,
);
```

You can assign the table and columns different names, but the above example reflects the defaults that `JDBCStore` will use if you don't specify them. Table 2-7 shows the attributes for `JDBCStore`.

Table 2-7. JDBCStore attributes

Attribute	Meaning
`className`	The name of the `Store` class to use; must be set to `org.apache.catalina.session.JDBCStore` for `JDBCStores`.
`driverName`	The fully qualified Java class name of the JDBC driver to use.
`connectionURL`	The JDBC connection URL to use.
`sessionTable`	The name of the session table in the database. The default is `tomcat$sessions`.
`sessionIdCol`	The name of the session ID column in the session table. The default is `id`.
`sessionDataCol`	The name of the session data column in the session table. The default is `data`.
`sessionValidCol`	The name of the column in the session table that stores the validity of sessions. The default is `valid`.
`sessionMaxInactiveCol`	The name of the column in the session table that stores the maximum inactive interval for sessions. The default is `maxinactive`.
`sessionLastAccessedCol`	The name of the column in the session table that stores the last accessed time for sessions. The default is `lastaccess`.
`checkInterval`	The interval (in seconds) at which `JDBCStore`'s background `Thread` checks for expired sessions. The default is `60`.
`debug`	Debugging level, where `0` is none, and positive numbers result in increasing detail. The default is `0`.

Accessing JNDI and JDBC Resources

Many web applications will need access to a relational database. To make web applications portable, the J2EE specification requires a database-independent description in the web applications's *WEB-INF/web.xml* file. It also allows the container developer to supply a means for providing the database-dependent details; Tomcat developers naturally chose to put this in the *server.xml* file. Then, the Java Naming and Directory Interface (JNDI) is used to locate database sources and other resources. Each of these resources, when accessed through JNDI, is referred to as a *context*.

Watch out! A JNDI context is completely different than a Tomcat context (which represents a web application). In fact, the two are completely unrelated.

JDBC DataSources

You probably know whether your web application requires a JDBC `DataSource`. If you're not sure, look in the *web.xml* file for the application and search for something like this:

```
<resource-ref>
  <description>
    The database DataSource for the Acme web application.
  </description>
  <res-ref-name>
    jdbc/JabaDotDB
  </res-ref-name>
  <res-type>
    javax.sql.DataSource
  </res-type>
  <res-auth>
    Container
  </res-auth>
</resource-ref>
```

As an alternative, if you have the Java source code available, you can look for something like this:

```
Context ctx = new InitialContext( );
DataSource ds = (DataSource)
  ctx.lookup("java:comp/env/jdbc/JabaDotDB");

Connection conn = ds.getConnection( );
... Java code that accesses the database ...
conn.close( );
```

If you're not familiar with JNDI usage from Java, a `DataSource` is an object that can hand out JDBC `Connection` objects on demand, usually from a pool of preallocated connections.

Tomcat 4 uses the Apache Database Connection Pool (DBCP) by default.

In either of the previous code snippets, the resource string `jdbc/JabaDotDB` tells you what you need to configure a reference for in the *server.xml* file. Find the `Context` element for your web application, and insert `Resource` and `ResourceParam` elements similar to those shown in Example 2-12.

Example 2-12. DataSource: Resource and ResourceParam in server.xml

```
<!-- Configure a JDBC DataSource for the user database. -->
<Resource name="jdbc/JabaDotDB"
          type="javax.sql.DataSource"
          auth="Container"
/>

<ResourceParams name="jdbc/JabaDotDB">
    <parameter>
        <name>user</name>
        <value>ian</value>
    </parameter>
    <parameter>
        <name>password</name>
        <value>top_secret_stuff</value>
    </parameter>
    <parameter>
        <name>driverClassName</name>
        <value>org.postgresql.Driver</value>
    </parameter>
    <parameter>
        <name>url</name>
        <value>jdbc:postgresql:jabadot</value>
    </parameter>
    <parameter>
        <name>maxActive</name>
        <value>8</value>
    </parameter>
    <parameter>
        <name>maxIdle</name>
        <value>4</value>
    </parameter>
</ResourceParams>
```

If this same `DataSource` will be used by other web applications, then the `Resource` and `ResourceParam` elements can instead be placed in a `GlobalNamingResources` element for the appropriate `Host` or `Engine`. See Chapter 7 for details on the `GlobalNamingResources` element.

You also need to install the JAR file for the database driver (we used PostgreSQL, so the driver is in *pgjdbc2.jar*). Since the driver is now being used both by the server and the web application, it should be copied from the application's *WEB-INF/lib* into *$CATALINA_HOME/common/lib*.

For more detailed information about using JDBC with servlets, see Chapter 9 in the book *Java Servlet Programming*, by Jason Hunter (O'Reilly).

Other JNDI Resources

Tomcat allows you to use its established JNDI context to look up any kind of resource available through JNDI. If the Java class being looked up fits the standard "JavaBeans conventions" (at the least, it must be a public class with a public no-argument constructor and must use the `setXXX( )`/`getXXX( )` pattern), you can use a Tomcat-provided `BeanFactory` class. Otherwise, you must write some Java code to create a factory class.

Here we configure `BeanFactory` to return instances of a `java.util.Calendar` object. First, add these lines in *web.xml*:

```
<!--
 How to get a Calendar on demand (real code would just
 call Calendar.getInstance; we just pick on Calendar as
 a handy Bean.
-->
<resource-env-ref>
    <description>
        Fake up a Factory for Calendar objects
    </description>
    <resource-env-ref-name>
        bean/CalendarFactory
    </resource-env-ref-name>
    <resource-env-ref-type>
        java.util.GregorianCalendar
    </resource-env-ref-type>
</resource-env-ref>
```

And in *server.xml*, make the following additions:

```
<!-- Set up factory for Calendar objects -->
<Resource name="bean/CalendarFactory"
         type="java.util.GregorianCalendar"
         auth="Container" />
 <ResourceParams name="bean/CalendarFactory">
     <parameter>
         <name>factory</name>
         <value>org.apache.naming.factory.BeanFactory</value>
     </parameter>
</ResourceParams>
```

Because this book is not aimed primarily at Java developers, we are not including a custom factory class. An example appears in the Tomcat documentation file at *http://jakarta.apache.org/tomcat/tomcat-4.0-doc/jndi-resources-howto.html*.

For more detailed information about using JNDI with servlets, see Chapter 12 in the book *Java Servlet Programming* (O'Reilly).

Servlet Auto-Reloading

By default, Tomcat will automatically reload a servlet when it notices that the servlet's class file has been modified. This is certainly a great convenience when debugging servlets. However, bear in mind that Tomcat must periodically check the modification time on every servlet in order to implement this functionality. This entails a lot of filesystem activity that is unnecessary when the servlets have been debugged and are not changing.

To turn this feature off, you need only set the `reloadable` attribute in the web application's `Context` element (in *web.xml*) and restart Tomcat. Once you've done this, you can still reload the servlet classes in a given `Context` by using the `Manager` application (detailed earlier in this chapter).

Relocating the Web Applications Directory

Depending on how you install and use Tomcat, you may not want to store your web application's files in the Tomcat distribution's directory tree. For example, if you installed Tomcat as a Linux RPM, you probably shouldn't modify Tomcat's files—for instance, *conf/server.xml*, which you will likely need or want to modify in order to configure Tomcat for your site*—because RPM is supposed to have control over the contents of the files it installs. This is likely to be the case with other native package managers as well. Upgrading the Tomcat package means that the native package manager might replace your configuration files with stock versions from the new package. Usually, package managers save the file they're replacing, but even then it's a pain to get your site back in running order. Regardless of how you installed Tomcat, though, it may be a good idea to keep your web site's files clearly separate from the Tomcat distribution files.

Another scenario when you may not want to store your web application files in the Tomcat distribution's directory tree is if you install one copy of the Tomcat distribution, but you wish to run more than one instance of Tomcat on your server computer. There are plenty of reasons why you may want to run more than one Tomcat instance, such as having each one serving content on different TCP ports. In this

* See "server.xml" in Chapter 7 for detailed information about configuring the XML elements in the *server.xml* file.

case, you don't want files from one instance clashing with files from another instance.

In order to have one Tomcat distribution installed while running two or more Tomcat instances of the Tomcat installation, each with different configuration and writeable data directories, you must keep each instance's files separate. During normal usage of Tomcat, the server reads configuration files from the *conf* and *webapps* directories, and writes files to the *logs*, *temp*, and *work* directories. This means that for multiple instances to work, each Tomcat instance has to have its own set of these directories—they cannot be shared. To make this work, just set the `CATALINA_BASE` environment variable to point to a new directory on disk where the Tomcat instance can find its set of configuration and other writeable directories.

First, change to the directory in which you'd like to put an instance's files. This can be anywhere on your system, but we suggest you locate this directory somewhere convenient that can hold a large amount of data:

```
# cd /usr/local
# mkdir tomcat-instance
# cd tomcat-instance
```

Next, create a directory for the new Tomcat instance (it should probably be named after the site that will be stored within it):

```
# mkdir groovywigs.com
# cd groovywigs.com
```

If you don't like the dot in the filename, you can change it to an underscore, or make a directory called *com* and add subdirectories named after the domain, such as *groovywigs*. You'll end up with a structure like most Java environments: *com/groovywigs*, *com/moocows*, *org/bigcats*, and so forth.

Now copy the Tomcat distribution's *config* directory to this new directory, and then create all of the other Tomcat instance directories:

```
# cp -r $CATALINA_HOME/conf .
# mkdir logs temp webapps work
```

When you create these directories and files, make sure that the user you run Tomcat as has read/write permissions to all of them.

Finally, place the web application content for this instance in the *webapps* subdirectory, just as you would in any other configuration of Tomcat. Edit the *conf/server.xml* file to be specific to this instance, and remove all unnecessary configuration elements. You need to do this so that your Tomcat instance doesn't try to open the same host and ports as someone else's Tomcat instance on the same server computer, and so that it doesn't try to load the example web applications that come with Tomcat. Change the shutdown port to a different port number:

```
<Server port="8007" shutdown="SHUTDOWN" debug="0">
```

and change the ports of any connectors:

```
<!-- Define a non-SSL Coyote HTTP/1.1 Connector on port 8080 -->
<Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
           port="8081" minProcessors="5" maxProcessors="75"
           enableLookups="true" redirectPort="8444"
           acceptCount="100" debug="0" connectionTimeout="20000"
           useURIValidationHack="false" disableUploadTimeout="true" />
```

Remove all of the example `Context` elements and anything nested within them, and add a context for your own web application (see "Context" in Chapter 7 for more information about how to configure a `Context`).

Repeat these steps to create additional instance directories as necessary.

To start up an instance, set `CATALINA_BASE` to the full path of the instance directory, set `CATALINA_HOME` to the full path of the Tomcat distribution directory, and then start Tomcat normally:

```
# set CATALINA_BASE="/usr/local/tomcat-instance/groovywigs.com"
# set CATALINA_HOME="/usr/local/jakarta-tomcat-4.0"
# export CATALINA_BASE CATALINA_HOME
# /etc/rc.d/init.d/tomcat4 start
```

You can stop these instances similarly:

```
# set CATALINA_BASE="/usr/local/tomcat-instance/groovywigs.com"
# set CATALINA_HOME="/usr/local/jakarta-tomcat-4.0"
# export CATALINA_BASE CATALINA_HOME
# /etc/rc.d/init.d/tomcat4 stop
```

You can also create small *start* and *stop* scripts so that you can start and stop instances easily. Perform the following steps:

```
# cd /usr/local/tomcat-instance/groovywigs.com
# mkdir bin
# cd bin
```

Now edit a file named *start* and put the following contents in it:

```
#!/bin/sh
set CATALINA_BASE="/usr/local/tomcat-instance/groovywigs.com"
set CATALINA_HOME="/usr/local/jakarta-tomcat-4.0"
export CATALINA_BASE CATALINA_HOME
/etc/rc.d/init.d/tomcat4 start
```

Be sure to make this script executable:

```
# chmod 700 start
```

Again, make sure that the Tomcat process owner has read and execute permissions to the *bin* directory and the new *start* script.

Then, to start up an instance, you can simply use this script:

```
# /usr/local/tomcat-instance/groovywigs.com/bin/start
```

Follow the same steps to create a *stop* script.

Once you organize your own files separately from the Tomcat distribution, upgrading Tomcat is easy because you can replace your entire Tomcat distribution directory with a new one without worrying about disturbing any of your own files. The only exception to this would be upgrading to a new Tomcat that is not compatible with your last Tomcat's instance files (something that very rarely happens). Once you start up a web application on a new Tomcat version, be sure to check the log files for any problems.

Customized User Directories

Some sites like to allow individual users to publish a directory of web pages on the server. For example, a university department might want to give each student a public area, or an ISP might make some web space available on one of its servers to customers that don't have a virtually hosted web server. In such cases, it is typical to use the tilde character (~) plus the user's name as the virtual path of that user's web site:

```
http://www.cs.myuniversity.edu/~ian
http://members.mybigisp.com/~ian
```

The notion of using ~ to mean a user's home directory originated at the University of Berkeley during the development of Berkeley Unix, when the C-shell command interpreter was being developed in the late 1970s. This usage has been expanded in the web world to refer to a particular directory inside a user's home directory or, more generally, a particular user's web directory, typically a directory named *public_html*.

Tomcat gives you two ways to map this on a per-host basis, using a couple of special `Listener` elements. The `Listener`'s `className` attribute should be `org.apache.catalina.startup.UserConfig`, with the `userClass` attribute specifying one of several mapping classes. If your system runs Unix, has a standard */etc/passwd* file that is readable by the account running Tomcat, and that file specifies users' home directories, use the `PasswdUserDatabase` mapping class:

```
<Listener className="org.apache.catalina.startup.UserConfig"
  directoryName="public_html"
  userClass="org.apache.catalina.startup.PasswdUserDatabase"
/>
```

Web files would need to be in directories like */home/users/ian/public_html* and */users/jbrittain/public_html*. Of course, you can change *public_html* to be whatever subdirectory your users put their personal web pages into.

In fact, the directories don't have to be inside a user's home directory at all. If you don't have a password file but want to map from a user name to a subdirectory of a common parent directory such as */home*, use the `HomesUserDatabase` class:

```
<Listener className="org.apache.catalina.startup.UserConfig"
   directoryName="public_html"
   homeBase="/home"
   userClass="org.apache.catalina.startup.HomesUserDatabase"
/>
```

In this case, web files would be in directories like */home/ian/public_html* and */home/jbrittain/public_html*.

This format is more useful on Windows, where you'd likely use a directory such as *C:\home*.

These `Listener` elements, if present, must be inside a `Host` element, but not inside a `Context` element, as they apply to the `Host` itself. For example, if you have a host named `localhost`, a `UserConfig` listener, and a `Context` named "tomcatbook", then the URL *http://localhost/~ian* will be valid (if it can be mapped to a directory), but the URL *http://localhost/tomcatbook/~ian* will be invalid and will return a 404 error. That is, the `UserConfig` mapping applies to the overall host, not to its contexts.

Tomcat Example Applications

When installed out of the box, Tomcat includes a variety of sample applications, all within the context */examples*. These are actually quite useful to people learning how to write JavaServer Pages and servlets. (The two relevant subdirectories under *examples* are *jsp* and *servlets*, each containing several examples, with source code viewable online.) Since these examples are so helpful, you may wish to make them available; on the other hand, you may not want somebody else's examples showing up on your production web server. In that case, you should remove the deployment `Context` element from the file *server.xml* in the Tomcat *conf* directory. Look for the section beginning like this:

```
<!-- Tomcat Examples Context -->
<Context path="/examples" docBase="examples" debug="0"
        reloadable="true" crossContext="true">
  <!-- Context content -->
</Context>
```

Comment out all of these lines (and the content in between them), and restart Tomcat. But be careful—it's about 100 lines long, containing examples of many XML deployment elements. And you can't just put the XML comment delimiters `<!--` and `-->` around the element, because comments in XML do not nest! If you don't want to deal with this issue and haven't made any other changes to your *server.xml* file, you can use the provided *server-noexamples.xml*. This file is almost identical to the default *server.xml*, except that it does not contain the *examples* `Context`.

> Alas, it is maintained by hand, so there are often minor differences between the two files, in addition to the absence of the *examples* `Context`.

Just rename the old *server.xml* to *server-examples.xml*, and rename *server-noexamples.xml* to *server.xml*. Finally, restart Tomcat to put these changes into effect.

Server-Side Includes

Tomcat is primarily meant to be a servlet/JSP engine, but it has a surprising number of abilities that rival those of a traditional web server. One of these is its Server-Side Include (SSI) capability. Traditional web servers provide a means for including common page elements—such as header and footer sections, JavaScript code, and other reusable items—into a web page. This idea is actually patterned on the C-language preprocessor `#include` mechanism, but it is slightly enhanced for use in web pages. For example, in one of our web sites we have a table navigator that is meant to appear at the left of each page. Instead of copying the table cells into each page, we store them in their own small file. Then, each page that needs the navigator accesses it using code like this:

```
<!--#include virtual=/"tablenav.html" -->
```

Files with these directives are generally named with the extension *shtml*, rather than *html*, to indicate to the web server that they should be processed differently than standard HTML code.

While JSP experts would argue that it is probably simpler to rename the files to JSP and use a `jsp:include` element, there might be cases when a large number of legacy pages must be served, and you prefer to keep them as SHTML files, perhaps for compatibility with other servers. It's probably not worthwhile to convert SHTML files to JSP just so you can process them as JSPs. In fact, Tomcat provides an SSI servlet for this very purpose. To enable it, you need only do the following:

1. Rename the file *servlets-ssi.renametojar* (found in *CATALINA_HOME/server/lib/*) to *servlets-ssi.jar*, so that the servlet processing SSI files will be on Tomcat's `CLASSPATH`. (This file is normally not installed with the extension *jar*, which reduces overhead by keeping it out of the `CLASSPATH` if you're not using it.)
2. Copy and uncomment the definition of the servlet named `ssi` from Tomcat's *web.xml* file to your application's *web.xml* (this is around line 180 in the distribution). Alternatively, if you want all contexts to have SSI processing, simply uncomment this servlet's definition in Tomcat's global *web.xml* file.

3. Copy and uncomment the `servlet-mapping` element for the HTML servlet from Tomcat's *web.xml* to your application's *web.xml* (around line 285 in the distributed file). Again, you can simply uncomment this in the main file.

Restart Tomcat or your web application (depending on which set of files you modified), and your SSI should now be operational.

The SSI servlet accepts a few `init-param` elements to control its behavior (see Chapter 7 for more detail on servlet parameters). The allowed parameters are listed in Table 2-8, along with sample and default values.

Table 2-8. SSI servlet initialization parameters

Parameter name	Meaning	Example	Default
`buffered`	`0` means to use unbuffered output, and `1` means to use buffered output.	`1`	`0`
`debug`	Debugging level.	`1`	`0`
`expires`	Seconds before expiry.	`300`	No expiry
`ignoreUnsupportedDirective`	Specifies how to react to unsupported SSI directives. `0` means don't ignore and report an error, and `1` means ignore silently.	`0`	`1`
`isVirtualWebappRelative`	Specifies how to react to included virtual paths. `0` maps them to the server root, and `1` maps them to the context root.	`1`	`0`

Buffering should normally be left on for efficiency; turn it off only for debugging, and remember to turn it back on for production. The expiry header is used by browsers to decide when to reload the page when the user revisits it. If there are no includes of dynamic content, set it high or leave it out; otherwise, set it to a low value, such as `60` (seconds). Setting the `ignoreUnspportedDirective` directive to `0` (that is, telling the servlet not to ignore errors) allows you to be notified if the SSI web page attempts to use an undefined SSI directive. The `isVirtualWebappRelative` directive controls whether `#include` paths are to be interpreted relative to the server root (that is, the directory specified in the `appBase` attribute of the `Host` element in *server.xml*) or relative to the context root (the `docBase` attribute in the `Context` element). The appropriate choice here depends on how your files are organized; for our site, we use the default.

For compatibility with other web servers that implement SSI, Tomcat's SSI servlet accepts the commands listed in Table 2-9.

Table 2-9. SSI commands supported by Tomcat

Name	Meaning	Example
`config`	Configures error message and various formats	`<!--#config errmsg="You goofed" -->`
`echo`	Prints an SSI server variable (set by set)	`<!--#echo var="myvar" -->`

Table 2-9. SSI commands supported by Tomcat (continued)

Name	Meaning	Example
`exec`	Executes a program and captures its output	`<p>Here is a list of Tomcat files:<br>`<br>`<!--#exec cmd="/bin/ls" -->`
`include`	Includes a file	`<!--#include virtual="x.html" -->`

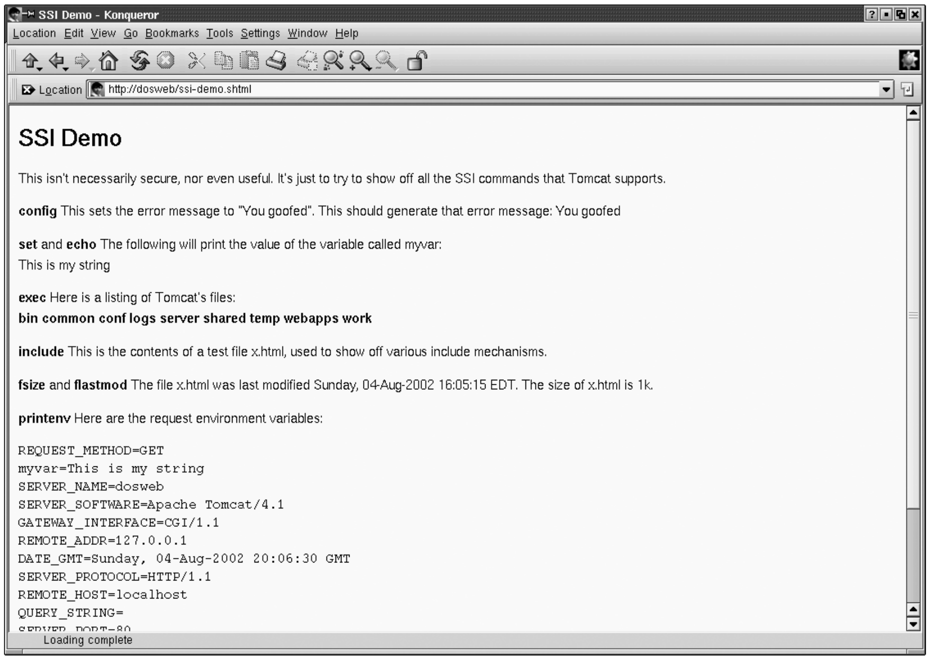
`flastmod`	File last modified time	`The file x.html was last modified`
		`<!--#flastmod file="x.html"-->.`

`fsize`	File size on disk	`The size of file x.html is`
		`<!--#fsize file="x.html"-->.`

`printenv`	Prints all SSI request variables; takes no arguments	`<p>Here are the request variables:`
		`<pre>`
		`<!--#printenv -->`
		`</pre>`

`set`	Sets an SSI server variable	`<!--#set var="myvar"`
		`  value="This is my string"-->`

The output of running the *ssi-demo.shtml* file on Tomcat with the SSI servlet enabled appears in Figure 2-1.


SSI Demo - Konqueror
Location Edit View Go Bookmarks Tools Settings Window Help
Location http://dosweb/ssi-demo.shtml
Loading complete

SSI Demo

This isn't necessarily secure, nor even useful. It's just to try to show off all the SSI commands that Tomcat supports.

config This sets the error message to "You goofed". This should generate that error message: You goofed

set and **echo** The following will print the value of the variable called myvar:
This is my string

exec Here is a listing of Tomcat's files:
bin common conf logs server shared temp webapps work

include This is the contents of a test file x.html, used to show off various include mechanisms.

fsize and **flastmod** The file x.html was last modified Sunday, 04-Aug-2002 16:05:15 EDT. The size of x.html is 1k.

printenv Here are the request environment variables:

```
REQUEST_METHOD=GET
myvar=This is my string
SERVER_NAME=dosweb
SERVER_SOFTWARE=Apache Tomcat/4.1
GATEWAY_INTERFACE=CGI/1.1
REMOTE_ADDR=127.0.0.1
DATE_GMT=Sunday, 04-Aug-2002 20:06:30 GMT
SERVER_PROTOCOL=HTTP/1.1
REMOTE_HOST=localhost
QUERY_STRING=
SERVER_PORT=80
```

Figure 2-1. SSI output from ssi-demo.shtml

Common Gateway Interface (CGI)

As mentioned in the previous section, Tomcat is primarily meant to be a servlet/JSP engine, but it has many capabilities rivalling a traditional web server. One of these is support for the Common Gateway Interface (CGI), which provides a means for running an external program in response to a browser request, typically to process a web-based form. CGI is called common because it can invoke programs in almost any programming or scripting language: Perl, Python, awk, Unix shell scripting, and even Java are all supported options. However, you probably wouldn't run Java applications as a CGI due to the start-up overhead; elimination of this overhead was what led to the original design of the servlet specification. Servlets are almost always more efficient than CGIs because you're not starting up a new operating system–level process every time somebody clicks on a link or button. You can consult a good book on web site management for details on writing CGI scripts.

Tomcat includes an optional CGI servlet that allows you to run legacy CGI scripts; the assumption is that most new backend processing will be done by user-defined servlets and JSPs. A simple CGI is shown in Example 2-13.

Example 2-13. CGI demonstration

```
#! /usr/local/bin/python

# Trivial CGI demo

print "content-type: text/html"
print ""

print "<html><head>Welcome</head>"
print "<body><h1>Welcome to the output of a CGI under Tomcat</h1>"
print "<p>The subject says all.</p>"
print "</body></html>"
```

As already mentioned, scripts can be written in almost any language. For the example we chose Python, and the first line is a bit of Unix that tells the system which interpreter to use for the script. On Windows, the filename would have to match some pattern, such as **.py*, to produce the same effect. The first few statements print the content type (useful to the browser, of course) and a blank line to separate the HTTP headers from the body. The remaining lines print the HTML content. This is typical of CGI scripts. Of course, most CGI scripts also handle some kind of forms processing, but that is left as an exercise for the reader. Presumably your CGI scripts are already working in whatever language you regularly use for this purpose.

To enable Tomcat's CGI servlet, you must do the following:

1. Rename the file *servlets-cgi.renametojar* (found in *CATALINA_HOME/server/lib/*) to *servlets-cgi.jar*, so that the servlet that processes CGI scripts will be on Tomcat's `CLASSPATH`.

2. In Tomcat's *web.xml* file, uncomment the definition of the servlet named `cgi` (this is around line 235 in the distribution).
3. Also in Tomcat's *web.xml*, uncomment the servlet mapping for the `cgi` servlet (around line 290 in the distributed file). Remember, this specifies the HTML links to the CGI script.
4. Either place the CGI scripts under the *WEB-INF/cgi* directory (remember that *WEB-INF* is a safe place to hide things that you don't want the user to be able to view for security reasons), or place them in some other directory within your context and adjust the `cgiPathPrefix` parameter (see Table 2-10) to identify the directory containing the files. This specifies the actual location of the CGI scripts, which typically will not be the same as the URL in the previous step.
5. Restart Tomcat, and your CGI processing should now be operational.

The CGI servlet accepts a few `init-param` elements to control its behavior. These are listed in Table 2-10.

Table 2-10. CGI servlet initialization parameters

Parameter Name	Meaning	Example	Default
`cgiPathPrefix`	Directory for the script files	*/cgi-bin*	*WEB-INF/cgi*
`clientInputTimeout`	How long to wait (in milliseconds) before giving up reading user input	`1000`	`100`
`debug`	Debugging level	`1`	`0`

The default directory for the servlet to locate the actual scripts is *WEB-INF/cgi*. As has been noted, the *WEB-INF* directory is protected against casual snooping from browsers, so this is a good place to put CGI scripts, which may contain passwords or other sensitive information. For compatibility with other servers, though, you may prefer to keep the scripts in the traditional directory, */cgi-bin*, but be aware that files in this directory may be viewable by the curious web surfer.

On Unix, be sure that the CGI script files are executable by the user under which you are running Tomcat.

The Tomcat Admin Application

Most of the work in this chapter has been figuring out what needs changing in a configuration file; knowing which XML to edit, and then editing that file; and restarting either Tomcat or the affected web application. We end this chapter with a look at an alternative way of making changes to Tomcat, one that will eventually become at least as important as editing XML files (among those who aren't dedicated to hand-editing XML, at any rate).

Most commercial J2EE servers provide a fully functional administrative interface, and many of these are accessible as web applications. The Tomcat Admin application is on its way to become a full-blown Tomcat administration tool rivaling these commercial offerings. First included in Tomcat 4.1, it already provides control over contexts, data sources, and users and groups. You can also control resources such as initialization parameters, as well as users, groups, and roles in a variety of user databases. The list of capabilities will be expanded upon in future releases, but the present implementation has proven itself to be quite useful.

The Admin web application is defined in the auto-deployment *CATALINA_BASE/webapps/admin.xml*.

If you do not have this file, you may not be running Tomcat 4.1. You will need to upgrade to take advantage of the Admin application.

You must edit this file to ensure that the path specified in the `docBase` attribute of the `Context` element is absolute, i.e., the absolute path of *CATALINA_HOME/server/webapps/manager*. Alternatively, you could just remove the auto-deployment file and specify the Admin context manually in your *server.xml* file. On machines that will not be managed by this application, you should probably disable it altogether by simply removing *CATALINA_BASE/webapps/admin.xml*.

You must also have a user who is assigned the admin role. Once you've performed these steps and restarted Tomcat, visit the URL *http://<yourhost>/admin* or *http://<yourhost>:8080/admin*, and you should see a login screen. The Admin application is built using container-managed security and the Jakarta Struts framework. Once you have logged in as a user assigned the admin role, you will see a screen like Figure 2-2.

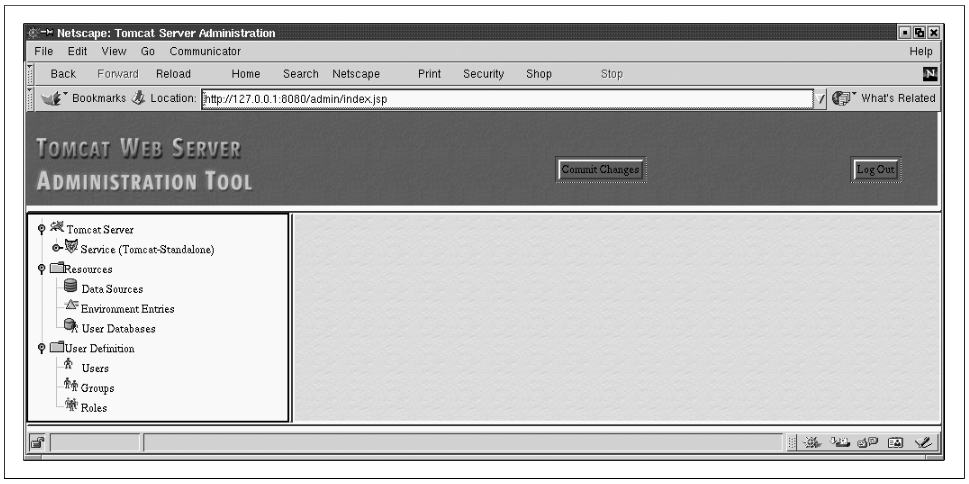
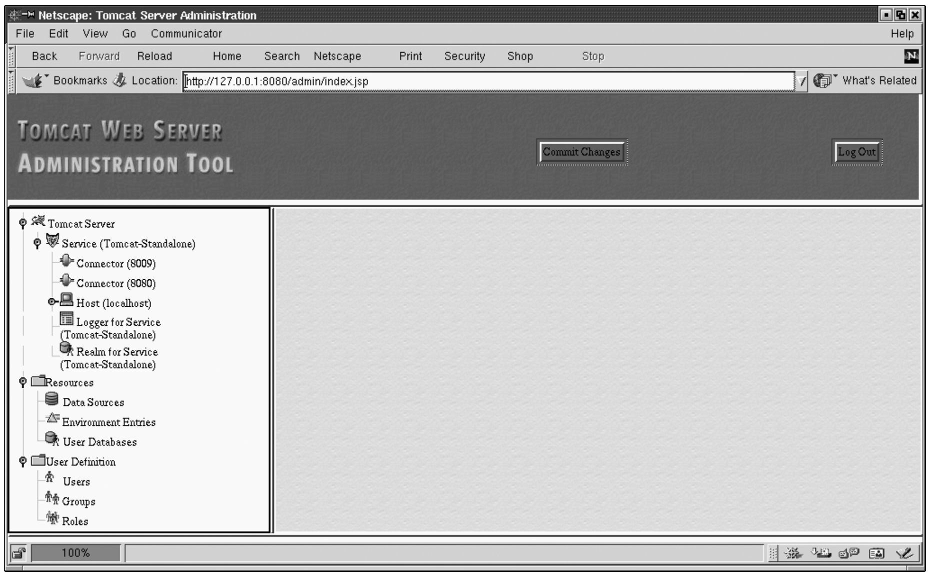

Netscape: Tomcat Server Administration
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://127.0.0.1:8080/admin/index.jsp
What's Related
TOMCAT WEB SERVER
ADMINISTRATION TOOL
Commit Changes
Log Out
Tomcat Server
Service (Tomcat-Standalone)
Resources
Data Sources
Environment Entries
User Databases
User Definition
Users
Groups
Roles

Figure 2-2. The Admin web application initial screen

As you can see, the application provides for controlling the Tomcat `Server`, `Host`, and `Context` elements; accessing resources such as JDBC `DataSources`, environment entries for web applications, and user databases; and performing user administration tasks such as editing users, groups, and roles. You can make any change to your web applications through this web interface.

Figure 2-3 shows the expanded server tree.



Netscape: Tomcat Server Administration
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://127.0.0.1:8080/admin/index.jsp
What's Related
Tomcat Web Server Administration Tool
Commit Changes
Log Out
Tomcat Server
Service (Tomcat-Standalone)
Connector (8009)
Connector (8080)
Host (localhost)
Logger for Service (Tomcat-Standalone)
Realm for Service (Tomcat-Standalone)
Resources
Data Sources
Environment Entries
User Databases
User Definition
Users
Groups
Roles
100%

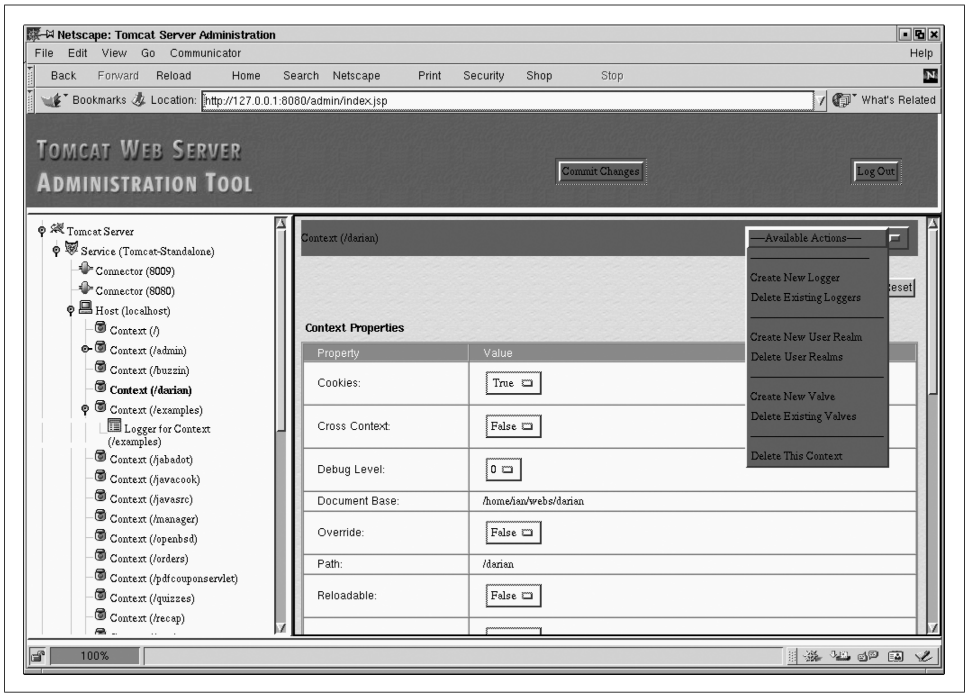
Figure 2-3. The Admin application with the server tree expanded

Figure 2-4 shows one context selected and some of the actions that can be performed on a context.

> Note that any changes you make will not take effect unless you press the "Commit Changes" button before leaving the panel.
>
> Additionally, changes made to a `Context` are only the changes you could make by editing *server.xml*; this version of the Admin application does not change the contents of the context's *web.xml* file.

The Admin web application is new in Tomcat 4.1, and there might be some changes, so we are not going to document all of Admin's capabilities in this edition. We'd like to close with the following points:

- The Admin web application is a frontend for editing XML. You still need to know what you're doing when you fill in the forms, so the Admin application is no substitute for poor XML or for the rest of this book.



Context Properties

Property	Value
Cookies:	True
Cross Context:	False
Debug Level:	0
Document Base:	/home/ian/webs/darian
Override:	False
Path:	/darian
Reloadable:	False

Figure 2-4. The Admin application and actions that it can perform on a context

- When you commit your changes, all of the comments and extra spacing that make the XML human-readable are discarded. The application also specifically adds attributes with default values, adding a lot of verbosity to the XML configuration files.
- Clicking on the wrong button can remove any part of your XML structure, so be careful (it does keep a backup of your *server.xml* file).
- You (or the developers of relevant web applications) still need to edit the *web.xml* file within the web application.

Having said all that, we encourage you to explore the Admin web application and see if it is more useful than editing the XML directly.