**MODULE 9**

**Security**

**Securing the Unix File System**

You’ll now see how to secure your server’s file system against malicious intruders who may read and execute sensitive files. You may already know these techniques if you’re a seasoned administrator, so feel free to move on to the Tomcat-specific sections.

Security has always been inherent in the Unix file system. The Unix permissions are the same for directories and files, because a directory is considered to be a special kind of file. Table 9-1 describes these permissions.

**Table 9-1.** *File Permissions for Unix*

|  |  |
| --- | --- |
| **Permission** | **Actions Permitted** |
| *Read* | View the contents of the file or directory. |
| *Write* | Modify or delete a file, or create files in a directory. |
| *Execute* | Execute a file, or access a directory. |

**Setting Permissions in Unix**

To view permissions in Unix, all you need is the ls utility. The following command will display the contents of a directory complete with permissions information:

*$ ls -l*

The -l parameter tells the ls command to display the long file directory format, which includes the permissions. The output of this command looks like the following:

*$ ls -l*

*drwxr-xr-x 2 tomcat tomcat 4096 Oct 22 18:38 bin*

You’ll see a series of columns that correspond to each file or directory in the current direc- tory. You’re concerned only with the first, third, fourth, and last columns. Let’s define each of those and drop out the other columns that are irrelevant to the discussion.

*Permissions Owner Group Filename*

*=======================================*

*drwxr-xr-x tomcat tomcat bin*

Now, let’s break down the values of each entry in the permissions column. The permissions column itself can be viewed as four separate columns: file type, owner permissions, group permissions, and permissions for other users. Let’s take the first and last files from the previous list and break down the permissions column for each.

*File Type Owner Group Other P. Filename*

*=====================================================*

*d rwx r-x r-x bin*

The first subcolumn of the permissions column dictates the file type, such as d for directory, l for link, or - for a normal file. All of the remaining columns display whether the owner, group, or other

users have read (r), write (w), or executable (x) access to that file (remember that for directories, the executable property indicates whether the user has access to the direc- tory). In the previous case, all three of these groups have read and execute rights, but only the owner of the bin directory can write to the directory.

**Changing Permissions**

You use the chmod utility to change the permissions of a file. For example, you can change the permissions of a file so that every user can read and write to it by running chmod with the fol- lowing parameters:

*$ chmod u=rw,g=rw,o=rw file01*

The u parameter sets the permissions for the owner of the file; the g parameter sets the permissions for the group associated with the file, and o sets the permissions for everyone else. You can use one, two, or all three of these parameters. The following is the result of the operation:

*$ chmod u=rw,g=rw,o=rw file01*

*$ ls -l | grep file01*

*-rw-rw-rw- ... file01*

**Changing Ownership**

You can use the chown command to change the owner of a file and the group with which the file is associated.

*$ chown user[:group] filename*

So, if you want to change the owner of the LICENSE file from tomcat to bobcat, you’d issue this command:

*$ chown bobcat LICENSE*

Unix actually stores two more pieces of metadata with every file that relate to security. These are the SUID and SGUI bits. If a file has the SUID bit set, it indicates that users who execute the file will execute it as though they are the owners of the file. For example, if a file named program was owned by root and had the SUID bit set, and if another user executed program, the operating system would execute program as though the user were root.

The SGID bit is similar. Any file with the SGID bit will make the user who executes that file a member of the group associated with the file for that file’s execution.

You can set the SUID and SGID bits with the chmod utility. The syntax is as follows:

*$ chmod u+s [filename] (sets the SUID bit)*

*$ chmod g+s [filename] (sets the SGID bit)*

The SUID and SGID bits show up in the executable column of the permissions of each file as an s, as follows:

*-rwsr-sr-x 2 tomcat tomcat 4096 Aug 25 01:28 program01*

Of course, you should use the SUID and SGID bits with great caution.

**Planning Security Permissions**

Now it’s time to talk about how to secure your system by using wise permissions configurations.

**Separate Tomcat Account**

Some users run Tomcat with their normal user account or with the superuser account, both of which are bad ideas. If Tomcat ever becomes compromised, it could use the permissions granted to the account that started it (such as your own account or the all-powerful superuser account) to wreak havoc. Therefore, you vastly improve the security of your file system by creating a special user account just for running Tomcat. This distinct account should be assigned only those permissions necessary to run Tomcat and nothing more.

**Suggested Account Settings for Unix**

Create an account for running Tomcat called tomcat. You should include tomcat in only one group, also named tomcat. Because you’ll want to run Tomcat as a daemon, you shouldn’t let console logins use this account. Disabling login ability is often achieved by starring the account’s password.

Here are two examples of this technique. The first example is from a BSD-like system that doesn’t use a shadow password file.

*/etc/passwd: tomcat:\*:23102:100:Tomcat:/:/bin/csh*

The second example is from a Linux system that does use a shadow password file.

*/etc/passwd: tomcat:x:502:502:Tomcat:/:/bin/bash*

*/etc/shadow: tomcat:\*:12040:0:99999:7:::*

Note how the password column has been given an asterisk (\*). This means you can’t log into this account. The various Unix operating systems have several mechanisms for creating and configuring daemons. Chapter 2 discussed this procedure.

**Configuring File Permissions in Unix**

Up until now, you’ve created a special tomcat account and instructed your operating system to launch the service with your tomcat account. You now need to configure your file system’s permissions.

Table 9-2 shows the recommended directory, owner/group, and file permission combinations.

**Table 9-2.** *Assigning Permissions to Tomcat’s Directories*

|  |  |  |  |
| --- | --- | --- | --- |
| **Directory/File** | **Owner/Group** |  | **Permissions** |
| $CATALINA\_HOME | root/tomcat |  | rwxr-x--- |
| $CATALINA\_HOME/bin | root/tomcat |  | rwxr-x--- |
| $CATALINA\_HOME/bin/\*.sh | root/tomcat |  | rwxr-x--- |
| $CATALINA\_HOME/common | root/tomcat |  | rwxr-x--- |
| $CATALINA\_HOME/conf | root/tomcat |  | rwxrwx--- (only if using the admin application or a user database) or rwxr-x--- (otherwise) |
| $CATALINA\_HOME/logs | root/tomcat |  | rwxrwx--- |
| $CATALINA\_HOME/logs/\*.\* | root/tomcat | r | w-rw---- |
| $CATALINA\_HOME/server | root/tomcat |  | rwxr-x--- |
| $CATALINA\_HOME/shared | root/tomcat |  | rwxr-x--- |
| $CATALINA\_HOME/temp | root/tomcat |  | rwxrwx--- |
| $CATALINA\_HOME/webapps | root/tomcat |  | rwxr-x--- |
| $CATALINA\_HOME/work | root/tomcat |  | rwxrwx--- |

If not otherwise indicated, all files in the listed directories should have the same ownership as their parent directory and have rw-r----- permissions.

**Examining General Tomcat Security Principles**

Now that you’ve secured your file system against attack, you should consider a few other security issues before tackling Tomcat’s configuration files. All but one of these applies to Windows and Unix.

**Retaining Tomcat’s Administration Tools**

Because tomcat-users.xml stores your username and password for Tomcat’s management applications, you may want to switch to a different realm. Otherwise, a hacker may view this file and gain access to your Tomcat management applications (though you can also digest the passwords). Because the tomcat account has only read access to the webapps directory, the hacker couldn’t modify your web applications, but the hacker could stop or reload your web applications on demand. Thus, if you use the Tomcat realms that use tomcat-users.xml, you should consider removing the admin and manager web applications.

If you use the admin application to change server.xml or users in tomcat-users.xml, then you must have write access to the directory. This also means a hacker can write to this directory and carry out any of the commands that the admin tool can and so may be able to add context XML files for malicious code. This also applies to user databases because they need write access to tomcat-users.xml.

If this bothers you, remove the admin tool, should it be installed, and use a JDBC or JNDI realm (or even the memory realm if you want; it doesn’t require write access to any files).

**Read-Only webapps Directory**

You’ll note that you’ve set read-only permissions on the webapps directory. This is to prevent hackers from modifying your web applications. This also means, however, that you may not be able to use WAR files when running Tomcat with the tomcat account, since Tomcat won’t be able to expand the WAR into the file system. Thus, you’ll need to unzip the WAR file yourself when you deploy it, run Tomcat as root from the console when you deploy new WAR files for the first time, or set the containing host’s unpackWARs attribute to false.

This also means that web applications can’t write to their directories. This is recommended in the Servlet specification because it’s assumed that some servlet containers will run unexpanded WAR files and would thus not be able to write changes to the web application’s file system (it doesn’t have one). This is the case if unpackWARs is set to false.

**Securing Your Files**

You should review your operating system for any additional files stored on the server that should be secured. You should consider either moving these files to another server or ensuring that the tomcat account doesn’t have any permissions for them. On Windows, add the tomcat user to the Security tab for the file or directory, and explicitly click the Deny checkboxes. In Unix, set the permissions for others to nothing (-rwx).

**Knowing If Your Security Has Been Violated**

Despite your best efforts, it’s possible that a hacker may exploit Tomcat (or another service) and modify your file system in some unimagined way. Intrusion detection systems can help you detect when your file system has been tampered with. Tripwire is one of these programs, and Red Hat includes instructions for installing and configuring Tripwire in its Official Red Hat Linux Reference Guide.

If you’re not using Red Hat Linux, see [www.tripwire.com](http://www.tripwire.com/) for more information.

**Read-Only File Systems**

Some operating systems support read-only file systems. In this concept, available only to Unix-like operating systems, you configure two separate file systems. The first file system, a root file system, contains your operating system configured just the way you need it, with a Tomcat daemon. You then make this file system read-only (or for ultimate security, transfer it to a read-only medium, such as a CD or a hard drive that you can make read-only). The sec- ond file system is contained on a read-write medium and contains your Tomcat installation and other files that must be modified.

Should you want to take this (highly secure) step, you’ll need to find documentation for your specific operating system. No standard Unix way exists to achieve this functionality. You must exercise caution if you attempt this route; once you mark your root file system as read- only, you’ll need a boot disk to make any changes.

**Securing Tomcat’s Default Configuration**

In this section, you’ll tighten up the default installation by editing the configuration files and managing the web applications that come with Tomcat. This will remove some of the most vulnerable entry points for attacks against the server.

As detailed earlier, the admin application and user databases are the main causes of concern. If you don’t use either of these, then you may keep and use the manager application, if you have considered the previous options. Without access to the manager password or write access to server.xml, a hacker can’t access the manager application.

If you don’t take these steps, it’s safer to place the manager folder and its contents outside of Tomcat’s directory structure (you should consider doing this anyway). This means an intruder can’t enable it by just adding an entry to *tomcat-users.xml*. However, it’s still possible to enable the application by modifying the server.xml file and modifying the manager context’s *docBase* attribute. As long as the manager folder is on the same machine as the server installation, it’s possible to set up the manager again (though it’s more difficult if you have appropriate per- missions set).

You should always remove the example web applications (jsp-examples and servlets-examples) on a development server. They aren’t necessary for Tomcat to run and take up disk space, if nothing

else. Likewise, unless you’re using *WebDAV*, you should remove the webdav web application.

The Tomcat documentation is now provided as a web application named tomcat-docs, which is an entirely static web application with no JSP pages or servlets. Whether or not you leave this in place is up to you, as it may be useful for developers to have a local copy of the documentation, whether to save network traffic or in case of problems connecting to the outside world.

It may also be worth disabling the default ROOT web application if you don’t have one of your own. If your applications will be accessed by a web application context name, then it may be worth replacing the contents of the ROOT folder with an empty index.html file. You can then supply an empty web application that would show access restriction error messages to clients who attempt to access the directory.

Alternatively, you can also disable unauthorized access to the web application. Thus, it’s possible to restrict access to the ROOT application to internal clients, such as the developer group, using valves or filters.

**Securing Tomcat’s Permissions**

Configuring your file system for maximum security is an important part of securing your Tomcat deployment, but it’s only half of the picture. By using Java’s security manager architecture, you can restrict those features of the Java language that Tomcat is able to access.

**The Java Security Manager**

The Java security manager architecture allows you to impose fine-grained security restrictions to all Java applications. This security architecture is turned off by default, but you can turn it on at any time. In the following sections, you’ll see the security manager architecture in general terms and then look at how this architecture specifically applies to Tomcat.

**Overview of the Security Manager**

The security manager architecture works on the notion of permissions (just as the file system does). Once the security manager is turned on, applications must have explicit permission to perform certain security-sensitive tasks, such as creating a custom class loader or opening a socket to servers.

Therefore, to use the security manager effectively, it’s necessary to know how applications can be given permissions and what the possible permissions are.

**Granting Permissions to Applications**

Policy files are the mechanism that the security manager uses to grant permissions to applications. Policy files are nothing more than simple text files composed of individual actions that applications can perform.

A policy file is composed of grant entries, as shown in Listing 9-1.

**Listing 9-1.** *A Policy File*

*// first grant entry grant {*

*permission java.lang.RuntimePermission "stopThread";*

*}*

*// second grant entry*

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

*};*

The first grant entry demonstrates the simplicity of the syntax. It grants all pplications the ability to access the deprecated *Thread.stop*() method.

The second grant entry illustrates that code in specific locations can also be granted permissions. This is useful when you want to extend permissions to certain trusted code while denying permissions to all other code. In this case, all code in the JAVA\_HOME/lib/ext directory is granted all permissions, which disables the security manager architecture for that code.

**Writing Grant Entries**

Each grant entry must be composed of the following syntax:

*grant codeBase "URL" {*

*// this is a comment*

*permission permission\_class\_name "target\_name", "action";*

*...*

*};*

Note that comments in policy files must begin with // on each line. As you saw in the first grant entry, the *codeBase* attribute is optional. codeBase specifies a URL to which all the permissions should apply. Table 9-3 describes the syntax.

**Table 9-3.** *The* codeBase *Attribute’s Syntax*

###### codeBase Example Description

file:/c:/myapp/ This assigns the permissions in the grant block to the c:\myapp

directory. Note that the slash (/) indicates that only class files in the

directory will receive the permissions, not any JAR files or subdirectories.

[http://java.sun.com/\*](http://java.sun.com/) All code from the specified URL will be granted the permissions. In this case, the /\* at the end of the URL indicates that all class files and JAR files will be assigned the permissions but not any subdirectories.

file:/matthewm/- All code in the /matthewm directory will be granted the permissions in the grant block. The /- indicates that all class files and JAR files in the directory and its subdirectories will be assigned the permissions.

Within the grant block, one or more permissions can be assigned. A permission consists of a permission class name and, in some cases, an additional target that identifies a specific permission within the permission class. Some permission targets can additionally take parameters, called actions. Listing 9-2 shows examples of permissions.

**Listing 9-2.** *Example Permissions*

*grant {*

*// allows applications to listen on all ports*

*permission java.net.SocketPermission "localhost", "listen";*

*// allows applications to read the "java.version" property permission java.util.PropertyPermission "java.version", "read";*

*}*

Special classes that ultimately inherit from the abstract class java.security.Permission define permissions. Most permission classes define special targets that represent a security permission that can be turned on and off.

Nineteen different permission classes offer control over various permissions. You can view the complete list of permission classes and their targets at [http://java.sun.com/j2se/1.4/docs /guide/security/permissions.html.](http://java.sun.com/j2se/1.4/docs%20/guide/security/permissions.html.)

**Enabling the Security Manager System**

You can enable the security manager system by passing the -Djava.security.manager parameter to the JVM at startup, as follows:

*> java -Djava.security.manager MyClass*

By default, Java looks for JAVA\_HOME/lib/security/java.policy to determine what per- missions to grant when the security manager is turned on. For more information on enabling the security manager and using your own policy files, see <http://java.sun.com/j2se/1.4/> docs/guide/security/PolicyFiles.html.

**Using the Security Manager with Tomcat**

Now that I’ve covered the basics of the security manager system, it’s time to talk about how to use it with Tomcat.

**Enabling Tomcat’s Security Manager**

The preferred way to start Tomcat with the security manager enabled on Unix systems follows:

*$CATALINA\_HOME/bin/catalina.sh start -security*

**Tomcat’s Policy File**

Tomcat uses the *CATALINA\_HOME/conf/catalina.policy* file to determine its own permissions and those of its web applications.

Tomcat’s policy file grants all permissions to javac, which compiles JSP pages into servlets, and it also grants all permissions to any Java standard extensions. Four grant lines are used instead of two to deal with multiple path possibilities. Note that you may need to add additional grants to this section if your JVM uses different paths for its standard extensions (Mac OS X needs additional grants, for example) and you’re actually putting JARs or classes in those paths.

**Recommended Security Manager Practices**

Now that you know how to turn on the security manager with Tomcat and where Tomcat stores its policy file, you can look at recommended practices for granting permissions to your applications.

**Use the Security Manager**

If you don’t turn on Tomcat’s security manager, any JSP page or class file is free to perform any action it likes. This includes opening unauthorized connections to other network hosts, destroying your file system, or even abnormally terminating Tomcat by issuing the *System.exit*() command.

To maintain a secure Tomcat installation, you should assume that at some point a hacker will be able to deploy malicious code into one of Tomcat’s web applications. By turning the security manager on, you gain explicit control over what web applications are allowed to do.

**Regulating Common Code**

Placing code into Tomcat’s common class loader directories (CATALINA\_HOME/lib) is a good way to share common libraries among web applications. However, because of Tomcat’s liberal permission grants for this class loader (all permissions are granted), you may want to think twice before you make a habit of placing code in this class loader.

You must do either of the following:

* Ensure that all code placed in this class loader is trusted.
* Place the code in the shared class loader. This class loader isn’t covered by the security manager by default and is thus restricted in its actions.

**Example Grants**

As mentioned, turning the security manager on gives you complete control over what web applications are allowed to do. The flip side of this security coin is that web applications will find themselves unable to do some things that they may have taken for granted before. Consider the following tasks that are unauthorized with Tomcat’s default policy configuration:

* Creating a class loader
* Accessing a database via a socket (for example, the MySQL JDBC driver establishing a connection with a MySQL database)
* Sending an e-mail via the JavaMail API
* Reading or writing to files outside a web application’s directory

**Creating a Class Loader**

Listing 9-5 shows how to give a specific web application the ability to create a class loader.

**Listing 9-5.** *Allowing Class Loader Creation*

*grant codeBase "file:${catalina.home}/webapps/tomcatBook/WEB-INF/-" { permission java.lang.RuntimePermission "createClassLoader";*

*};*

This is an extremely dangerous permission to grant. Applications that can instantiate their own class loaders can, by definition, load their own classes. As mentioned earlier, mali- cious classes could then be used to compromise your system in a number of ways.

**Opening Socket Connections to Databases**

Listing 9-6 shows how to allow all web applications access to a specific database running on the host db.server.com on port 54321.

**Listing 9-6.** *Allowing a Database Connection*

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

*permission java.net.SocketPermission "db.server.com:54321", "connect";*

*};*

This example allows all code in all web applications to connect to db.server.com:54321. If this is too much of a security risk for you, you have a few alternative options.

First, explicitly assign permission to each web application’s JDBC driver individually, as shown in Listing 9-7.

**Listing 9-7.** *Enabling a Web Application to Make a Database Connection*

*grant codeBase "file:${catalina.home}/webapps/tomcatBook/WEB-INF/lib/JDBC.jar" { permission java.net.SocketPermission "db.server.com:54321", "connect";*

*};*

Second, place the JDBC driver into the common class loader, which has all permissions granted to it. This means the driver can access the database, but the web application can’t.

**Sending an E-mail with JavaMail**

To send e-mail, web applications need access to port 25 on an SMTP server. Listing 12-8 shows how to grant this permission to all classes in a web application.

**Listing 9-8.** *Allowing Access to an SMTP Server*

*grant codeBase "file:${catalina.home}/webapps/myWebApp/WEB-INF/classes/-" { permission java.net.SocketPermission "mail.server.com:25", "connect";*

*};*

**Reading or Writing to Files Outside a Web Application’s Directory**

If you want to use your operating system, rather than Java’s permissions, to control file access, you can give your web applications free rein once again, as in Listing 9-9.

**Listing 9-9.** *Allowing Access to All Files*

*grant {*

*java.io.FilePermission "<<ALL FILES>>", "read,write,execute,delete";*

*};*

If you don’t grant at least some file permissions to your web application, your web appli- cations will be shut out from accessing your file system. You should still secure your system with your operating system’s file permissions, because, even though your web applications may be shut out, Tomcat itself has full permissions, and should a malicious hacker modify Tomcat somehow, they could still access your file system.

**Security Realms**

A security realm is a Tomcat’s mechanism for protecting web application resources. It gives you the ability to protect a resource with a defined security constraint, and then define the user roles that can access the protected resource. More specifically, Tomcat’s realm can be defined as a collection of usernames, corresponding passwords, and the user roles associated with each user integrated with Tomcat server.

Tomcat’s abstraction of a security realm is defined by the *org.apache.catalina.Realm* interface. This interface provides a mechanism by which a collection of usernames, passwords, and their associated roles can be integrated into Tomcat. If you downloaded the Tomcat source, you can find this interface in the following location:

*<CATALINA\_HOME>/src/catalina/src/share/org/apache/catalina/Realm.java*

Tomcat 7 provides several implementations of Realm interface. Table 9-1 lists the implementations and their usage.

***Table 9-1.*** *Realm Implementation Available in Tomcat*

|  |  |
| --- | --- |
| **Concrete Realm class** | **Description** |
| *MemoryRealm* | Simple implementation that uses an xml file as a user store (typically tomcat-users.xml). |
| *JDBCRealm* | Realm that support storing usernames, passwords, and roles information in the SQL database. |
| *JNDIRealm* | Implementation backed by the Java Naming Directory Interface (JNDI). |
| *DataSourceRealm* | Realm backed by a JNDI-configured JDBC data source. |
| *UserDatabaseRealm* | Realm backed by a custom UserDatabase configured via JNDI. |
| *JaasRealm* | Security authentication using Java Authentication and Authorization Service (JAAS). |
| *CombinedRealm* | Realm implementation that allows usage of multiple realms at the same time. |
| *LockOutRealm* | Extends CombinedRealm, to lock out users if too many incorrect login tries are detected – prevents brute force server attack. |

In addition to the Realm implementations available out of the box with Tomcat, you can provide your own custom implementation. All you have to do is implement Realm interface, package your implementation with all required classes into jar file, and copy file to *CATALINA\_HOME/lib* directory. As discussed in previous chapters, all libraries in the lib directory will be available to Tomcat on startup, so you can start using your implementation straight away.

In this module, we will cover *MemoryRealm* and *JDBCRealm* in more detail. With *MemoryRealm*, we will cover its more advanced version, *UserDatabaseRealm*, and similarly *JDBCRealm* will be further explained with JNDI version *DataSourceRealm*. Finally, we will introduce *JNDIRealm*, which is used for authentication with LDAP directory server. For more information about other Realm implementations available with Tomcat 7, take a look at Tomcat’s online resources (<http://tomcat.apache.org/tomcat-> 7.0-doc/realm-howto.html).

**MemoryRealm**

Let’s take a look at the simplest realm available in Tomcat: *MemoryRealm*. The *MemoryRealm* (contained in package *org.apache.catalina.realm.MemoryRealm*) uses an in-memory user database, which is read from an XML file on server startup. The root of the XML structure is the *<tomcat-users>* element, which holds the <user> elements. Each <user> element contains the required attributes for each user – name (username for authentication), password (plain text password), and roles (comma-separated list of roles that the user belongs to, used for authorization).

Listing 9-10 shows an example of the XML file containing four users.

***Listing 9-10.*** *XML Defining Four Users, with Usernames, Passwords, and User Roles*

*<tomcat-users>*

*<user name="tomcat" password="tomcat" roles="tomcat" /> #1*

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

*<user name="both" password="tomcat" roles="tomcat,role1" /> #2*

*<user name="bob" password="password" roles="apressuser" />*

*</tomcat-users>*

Most user definitions are simple, with username, password, and a single role (#1), but one common scenario involves users that have two or more roles, which can be specified as comma-separated values in XML (#2).

* NOTE The default location of the *MemoryRealm’s* XML file is the *<CATALINA\_HOME>/conf/tomcat-users.xml*. You can change the location of this file in the Tomcat’s XML configuration, as we will show later in this section.

Now that we have discussed how user information is stored for usage with *MemoryRealm*, we can protect access to our sample web application.

**Protecting a Resource with a MemoryRealm**

To actually see how a *MemoryRealm* works, let’s create a realm that protects our sample web application /apress, which we introduced in previous modules.

The first step is to enable *MemoryRealm* in the Tomcat configuration file CATALINA\_HOME/conf/server.xml. All we need to do to achieve this is to add the following line under the

<Engine>, <Host>, or <Context> element of server.xml. We will configure it for our /apress web application under <Context> element, as Listing 9-11 shows.

***Listing 9-11.*** *MemoryRealm Configuration in server.xml file*

*<Context path="/chapter9" docBase="chapter9">*

*<Realm className="org.apache.catalina.realm.MemoryRealm" />*

*</Context>*

If the *<Realm>* element is added under the *<Engine>* element of server.xml, all web applications served by the engine will have the realm enabled. That usually means that all web applications deployed on the Tomcat instance can use the configured realm. If the *<Realm>* element is within the *<Host>* element, all web applications within that host will have the realm enabled, but the web applications belonging to other hosts defined won’t see the realm at all. Finally, if the *<Realm>* element is within the *<Context>* XML element, only web applications that are defined for that context can use the configured realm.

Only one realm is active for each web application at any given time. Therefore, the realm defined in *<Engine>* will apply to all web applications deployed on the server, unless it’s overridden by the *<Realm>* element under *<Host>.* Similarly, a realm defined under the <Engine> or <Host> elements can be overridden by a single web application-configured realm, under <Context> element. To avoid confusion and to keep configuration as robust as possible, it is a recommended approach to configure a realm on the web application level, within <Context> element, as our example shows.

However, just configuring the realm in this way won’t make the web application secure; adding the

<Realm> element to server.xml simply enables the realm for the web application. The web application must also be configured to use the realm’s security features to complete the configuration. You can test this by starting the Tomcat server after this change is made and trying to access /examples, /apress, or any other application deployed – all of them will still be accessible as before, without any username or password required.

The next step is to configure the /apress sample web application to use the configured *MemoryRealm* implementation by editing web.xml file for our web application (located in CATALINA\_HOME/webapps/apress/WEB-INF/web.xml). We need to add the <security-constraint> element to the web.xml, and define the protected resource. Listing 9-12 shows the web.xml configuration.

***Listing 9-12,*** *Definition of a Secured Resource in web.xml*

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

[*<web-app version="2.4" 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)

*<display-name>Module 9</display-name>*

*<description>Security Realms Demo</description>*

*<security-constraint> #1*

*<web-resource-collection> #2*

*<web-resource-name>Memory Realm Sample</web-resource-name> #3*

*<url-pattern>/\*</url-pattern> #4*

*</web-resource-collection>*

*<auth-constraint> #5*

*<role-name>apressuser</role-name> #6*

*</auth-constraint>*

*</security-constraint>*

*</web-app>*

* NOTE The order of XML elements in web.xml is important. If your application does not start, check the error messages in the log files, and make sure that the order of XML elements is correct.

First, we define the *<security-constraint>* element, which acts as a holder for all security configurations for this web application. This element has two required child elements: <web-resource- collection>, which defines the URL path of the web applications that are to be protected (#2), and <auth-constraint>, which defines authorization roles that a user accessing the protected URLs must have (#5). Within the *<web-resource-collection>* element we define its name (#3) and the URL pattern that matches all web application URLs that we want to protect (#4). It this example, we want to protect all pages of our sample application, so we use the wildcard character to match all URLs (/\*).

As for the authorization part, we set the role name that is required for the access pages specified (#6) – the role name is “apressuser.” If you take a look at Listing 6-1, you will notice that we have configured user “bob” to have role “apressuser.”

If you now restart Tomcat and navigate to our web application’s home page (http://localhost:8080/chapter6/jsps/index.jsp), you will see an error page with HTML status code

403 Forbidden – which means that access to the page is restricted successfully. Figure 6-1 shows the page in the browser.



***Figure 9-1.*** *Accessing a protected page in the browser without the required role*

So we have secured our web application, but when we tried to access it, we got the error page straight away. We should be offered a chance to enter a username and password and, if we log in successfully and have the required role, we should see the actual page.

The configuration so far only protected the page – we need a few more lines of XML code in order to tell Tomcat to render a login page if a user is not logged in.

For that, we are going to add the *<login-config>* element to web.xml file, after the *<security- constraint>* element. Listing 9-13 shows the additional configuration we need to add to the web.xml file of our web application.

***Listing 9-13.*** *Login Configuration for Secured Resources in web.xml*

*<login-config>*

*<auth-method>BASIC</auth-method> #1*

*<realm-name>Apress</realm-name> #2*

*</login-config>*

For login configuration, we need to set the authentication method to use. For this example, we will use basic authentication supported by Tomcat (#1). With basic authentication, the server will display a popup window when you try to access a secure resource. In this window, you can type a username and password. The username and password are then transported (in plain text) to the Tomcat’s built-in authentication mechanism, which will check your credentials against a configured user database (in our case, *tomcat-users.xml* file, as defined by *MemoryRealm*). We also configure the authentication realm name, which will be displayed in the login window (#2).

In addition to *BASIC* authentication, Tomcat supports other types of authentication: *FORM*, *DIGEST*, and *CLIENT\_CERT. FORM* authentication will be covered with the *JDBCRealm* sample in the next section. For more information about other authentication methods, consult the Tomcat

online manual. Table 9-2 lists authentication types supported by Tomcat.

***Table 6-2.*** *Authentication Types Supported by Apache Tomcat.*

|  |  |
| --- | --- |
| **Authentication** | **Description** |
| *BASIC* | Client authenticates by entering a username and password to a built-in browser login window. The browser sends the username and password in HTTP header in plain text, Base64 encoded. The username and password are then decoded on the server. |
| *FORM* | This is the most common authentication type. Client authenticates using HTML form by entering a username and password. The sending mechanism, password encoding, and the style of the HTML form are customizable by the user. The form attributes (input field names and form action attributes) are defined as part of Java Servlet specification. |
| *DIGEST* | Similarly to BASIC authentication, this uses the browser’s login window to collect credential details and send them to the server as HTTP header. Unlike with BASIC authentication, usernames and passwords are digested – encoded using MD5 algorithm, which is a one-way hash function, so the username and password cannot be decoded – making authentication more secure. Some older browsers do not support DIGEST authentication. |
| *CLIENT\_CERT* | This authentication type uses Secure Socket Layer (SSL), where both the client (user) and the server have their own SSL certificates, which are used for mutual authentication. This type of authentication does not require a username and password, and is the most secure of all the types mentioned.. |

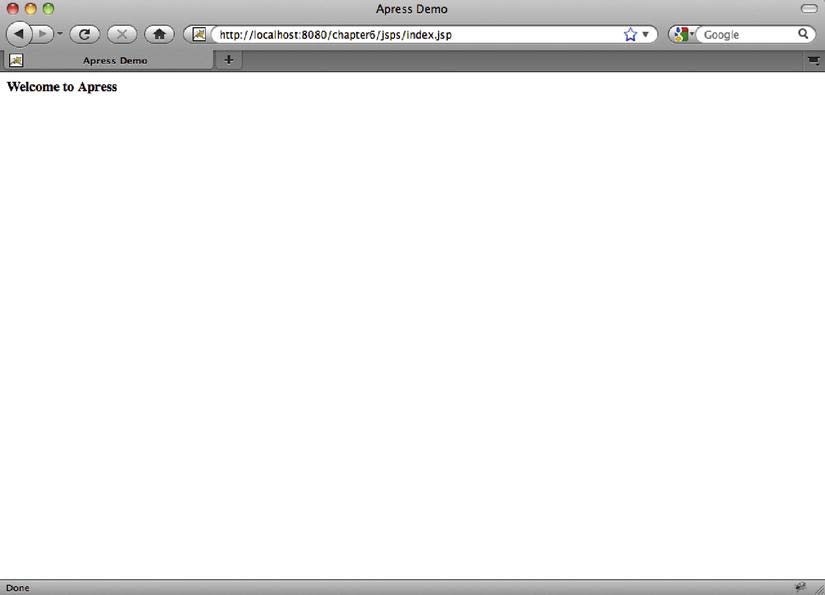
After finishing with the configuration, you’ll have to restart Tomcat in order to apply the changes.

After you restart, navigate the browser to the same URL as before (<http://localhost:8080/> module9/jsps /index.jsp), and you will see a login window displayed, asking you to enter your username and password.

If you enter a username or password that is not contained in the tomcat-users.xml file, you will see the login dialog displayed again. This will repeat until you enter a valid username and password. Tomcat handles the flow of BASIC authentication process, and displays the login prompt until the correct details are entered; there is no special configuration for this functionality.

Now enter the username and password of the user that has required role “apressuser” – which is our user “bob” with password “password” – and you will be presented with our sample web application home page. Figure 9-3 shows the page rendered in the browser after a successful authentication and authorization.

125



***Figure 9-3.*** *After a successful login, the home page of the application is displayed*

If you entered credentials for the existing user, but without required role, you will see the same page as shown in Figure 9-1: the 403 Forbidden error page.

No matter what authentication method you use, you are always at risk of malicious users discovering your usernames and passwords and being able to access secured pages. That is the reason the tomcat-users.xml file does not have any users entered by default – imagine if an inexperienced user downloaded Tomcat and deployed it to a public server – anyone who knows the default usernames and passwords would be able to access all the pages deployed on the server!

Even with all standard protection methods, malicious users can still attempt brute force attacks on your server. In the next section we will cover some options available within Tomcat to avoid such attacks.

**Protection Against Brute Force Attacks**

Brute force attacks are usually performed by other programs, which try huge numbers of usernames and passwords to attempt to gain access to the server. These malicious programs are usually written so they try words that are known to be used as usernames and often used as passwords (names, dates of birth, names followed with year of birth, and the like). Although there are millions of combinations available, it is still possible – given time – for the usernames and passwords to be guessed, especially on web sites with millions of users. A common approach for preventing this is to lock out users for a given period of time if they enter an incorrect password too many times.

Tomcat’s solution that is available out of the box is *LockOutRealm*, which we mentioned before as one of the realms available out of the box in Table 6-1. The *LockOutRealm* is a simple *CombinedRealm*, which encapsulates any other realms we can use. It needs to be configured in as a default Realm, replacing the realm we defined in listing 9-2. Listing 9-5 shows the *LockOutRealm*

configuration.

***Listing 9-5.*** *Configuring LockOutRealm for Brute Force Attack Protection*

*<Context path="/chapter6" docBase="chapter6">*

*<Realm className="org.apache.catalina.realm.LockOutRealm"*

*failureCount="3 lockoutTime="3600"> #1*

*<Realm className="org.apache.catalina.realm.MemoryRealm" /> #2*

*</Realm>*

*</Context>*

The LockOutRealm is configured as any other realm (#1). The MemoryRealm we want to use is configured within the LockOutRealm (#2). LockOutRealm has two important properties:

* f*ailureCount*: Defines how many failed attempts a user can have before being locked out. The default value is 5.
* *lockoutTime*: Defines how long (in seconds) the user will be locked out if the failureCount has been exceeded. The default is 300.

Although not the prefect solution, the user lock out strategy is simple and usually good enough to protect against brute force attacks to your server.

**UserDatabaseRealm**

UserDatabaseRealm is an advanced version of the *MemoryRealm*. It has the same capabilities as *MemoryRealm*, and the users database is backed with the same XML file (which defaults to *CATALINA\_HOME/conf/tomcat-users.xml* by default). The advantage of the UserDatabaseRealm is that it’s configurable via the Java Naming Directory Interface – the standard Java API that allows clients to look up objects by a known name. Don’t worry if you don’t understand every bit about JNDI configuration at this point; it will be covered in much more detail in Chapter 11. The configuration for the *UserDatabaseRealm* is very similar to that of the *MemoryRealm*, with the addition of JNDI resource configuration. Listing 9-6 shows the *UserDatabaseRealm* configuration within *<Context>* element in a server.xml file.

***Listing 9-6.*** *Configuring UserDatabaseRealm in the Tomcat’s server.xml file*

*<GlobalNamingResources>" #1*

*<Resource name="UserDatabase" auth="Container" #2 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" /> #3*

*</GlobalNamingResources>*

*<Context path="/chapter9" docBase="chapter9">*

*<Realm className="org.apache.catalina.realm.UserDatabaseRealm" resourceName="UserDatabase"/> #4*

*</Context>*

The JNDI resource is configured within the *<GlobalNamingResources>* element in server.xml (#1). All JNDI resources for the server instance will be listed here. Next, we configure our *UserDatabase* resource (#2) with some basic properties. The configuration and properties of the JNDI resource configuration will be covered in more detail in Module 8. The one attribute to notice is the pathname attribute, which points to the XML file, which contains the user database (#3). The XML file used is the same *tomcat-users.xml* file we used for our *MemoryRealm* example. You can configure the custom path for your user database file, but it has to have same XML format as Listing 9-1 shows. Finally, we configure the actual Realm implementation, referencing the configured JNDI resource via *resourceName* attribute (#4).

Using a JNDI-backed resource allows us to access it easily – either programmatically (using JNDI API) or using JMX, and potentially change the user’s details without needing to change the XML file on the disk. We can change *UserDatabase* resource instead.

*MemoryRealm* and *UserDatabaseRealm* security are easy to set up and configure; therefore, they are good to use for prototyping. Also, if your web application uses another authentication mechanism and does not use realms at all, *MemoryRealm* (or *UserDatabaseRealm*) can be handy for occasional Tomcat administration. However, user information is still stored in unsecure file, with passwords in plain text, making it potentially dangerous. In addition, the XML file can become difficult to maintain if you have hundreds or thousands of users. Because information about users in large systems is usually stored in some kind of relational database, in the next section we will demonstrate how to configure *JDBCRealm*, which will authenticate users against SQL database.

**JDBC Realms**

The second Realm implementation provided with Tomcat that we are going to cover is a JDBC realm, which is implemented by the *org.apache.catalina.realm.JDBCRealm* class. This class is much like the *MemoryRealm* discussed in the previous section, with the exception that it stores its collection of users. A *JDBCRealm* stores all of its users in a user-defined and JDBC-compliant database. Setting up a JDBC realm involves several steps, but it is really simple to manage once it is configured.

**Creating the Users Database**

Before you begin configuring Tomcat to use a *JDBCRealm*, you must first create a database to hold your collection of users. For this example, we will be configuring a MySQL database.

* NOTE If you already have a database of users, you can substitute the values we are using here with the appropriate values relating to your database. If you do not have an existing database, you can find the SQL scripts to create the MySQL database with the rest of the source code for this book.

In order to store user information in the SQL database and use JDBCRealm with the data, you will need to have two tables in the database schema.

The first table will store the usernames and passwords of all users, and must contain at least two text columns: one for usernames and one for passwords. For the sake of clarity, let’s call this table the users table. The names of the database, tables, and columns are not important, as they can be configured as part of *JDBCRealm* configuration. Table 9-3 shows the users table definition, as we will use it for our examples.

***Table 9-3.*** *Users Table Definition*

|  |  |
| --- | --- |
| **COLUMN** | **DESCRIPTION** |
| *user\_name* | Contains a string representing the username that will be used in the login form. It has a type of varchar(12). |
| *user\_pass* | Contains a string representing the user’s password. It also has a type of varchar(12). |

The second table will store the roles for every user; let’s call it the user\_roles table. It must have two text columns. One column will contain usernames, and it will be a foreign key column referencing usernames in the first table (with usernames and passwords). The other column will contain single role for the user. The user\_roles table will have a many-to-one relationship to the users table. Again, the name of the table or columns is not important. You can call these tables and columns whatever you want, as long as you configure the relevant names with the *<Realm>* element configuration in server.xml file. Table 9-4 shows the column definitions for the user\_roles table.

***Table 9-4.*** *The user\_roles Table Definition*

|  |  |
| --- | --- |
| **COLUMN** | **DESCRIPTION** |
| *user\_name* | Contains a string referring to a user in the users table. It has a type of varchar(12). |
| *role\_name* | Contains a string referring to a role in the roles table. It also has a type of varchar(12). |

* NOTE the columns specification discussed is the minimal requirement. Tables can contain other columns as well, as required for the application. For example, most users tables will contain information like last login, full name, email address, and other. Other columns in your tables will not be read or used by JDBCRealm.

Listing 9-7 shows the SQL script for the creation of the MySQL database with tables required for the JDBCRealm example.

***Listing 9-7.*** *Create Database SQL Script*

*create database tomcatusers; use tomcatusers;*

*create table users (*

*user\_name varchar(12) not null primary key,*

*user\_pass varchar(12) not null*

*);*

*create table users\_roles (*

*user\_name varchar(12) not null, role\_name varchar(12) not null,*

*primary key(user\_name, role\_name)*

*);*

Before you can create the users database in MySQL, you need to have downloaded and installed the MySQL server, which can be found at [www.mysql.com.](http://www.mysql.com/) You should also download the latest JDBC driver for MySQL, which can also be found at the same Web site. After you have MySQL installed, you need to complete the following steps to create and configure a MySQL Users database:

1. Start the MySQL client found in the <MYSQL\_HOME>/bin/ directory.
2. Create the tomcatusers database, and the users and user\_roles tables, by executing the SQL script from Listing 9-7.
3. Create the user that we will use to connect to the database from Tomcat, and grant this user permission to access the tomcatusers database. To keep things simple, we will create a user with username “test” and the password “test.” You will need to execute the SQL script from the following code snippet to create the “test” user and grant the required permissions:

*create user 'test'@'localhost' identified by test; grant all privileges on tomcatusers.\* to test@localhost*

1. Finally, we’ll need to insert few users to the database so that we can run our sample application. Listing 9-8 shows the insert SQL script, which populates the database with the same usernames, passwords, and roles that we used for the *MemoryRealm* example in the previous section (see Listing 9-1).

***Listing 9-8.*** *SQL Script Used to Populate the tomcatusers Database*

insert into users values("tomcat", "tomcat"); insert into users values("role1", "tomcat"); insert into users values("both", "tomcat"); insert into users values("bob", "password");

insert into user\_roles values("tomcat", "tomcat"); insert into user\_roles values("role1", "role1"); insert into user\_roles values("both", "tomcat"); insert into user\_roles values("both", "role1");

insert into user\_roles values("bob", "apressuser");

Now we have a MySQL database of users ready and we can proceed with JDBCRealm configuration.

**Configuring Tomcat to Use a JDBCRealm**

Now that we have a collection of users stored in a relational database (MySQL), let’s configure Tomcat to use the JDBC container instead of the previously configured *MemoryRealm*. Here are the steps involved in configuring a *JDBCRealm*:

1. Open the <CATALINA\_HOME>/conf/server.xml and place a comment around the previously uncommented <Realm> element.

*<!-- <Realm className="org.apache.catalina.realm.MemoryRealm" /> -->*

We mentioned before that only one realm can be active for each web application, so we are removing the old *MemoryRealm* configuration in order to introduce the new *JDBCRealm*.

1. Replace the commented out *<Realm>* element with the *JDBCRealm* configuration, as Listing 6-9 illustrates.

Listing 6-9. JDBCRealm Configuration in server.xml

*<Realm className="org.apache.catalina.realm.JDBCRealm" #1*

*driverName="org.gjt.mm.mysql.Driver" #2 connectionURL="jdbc:mysql://*

*localhost/tomcatusers?user=test;password=test" #3*

*userTable="users" #4*

*userNameCol="user\_name" userCredCol="user\_pass" #5*

*userRoleTable="user\_roles" #6*

*roleNameCol="role\_name" #7*

*/>*

To enable *JDBCRealm*, the className attribute of <Realm> element must specify its class, *org.apache.catalina.realm.JDBCRealm* (#1). The next required attribute is driverName, which specifies the JDBC driver class name for the selected database – in our case, MySQL driver (#2). The *connectionURL* attribute specifies the database connection URL (#3). It starts with jdbc:, and contains the host name or IP of the server where the database is running (in our case, localhost), as well as the database name, and username and password of the database user (these details are configured outside Tomcat, within the selected database administration console).

These properties conclude the general database configuration. The next step is to tell *JDBCRealm* which tables and columns to use in the configured database to load users, their passwords, and roles. Attribute *userTable* specifies the table, which contains usernames and passwords, and in our case it has the value “users” (#4). The columns of the table specified with userTable attribute are defined with the attributes *userNameCol* (username column) and *userCredColumn* (password column) (#5). Finally, we specify the table, which contains user roles using attribute userRoleTable (#6) and its column for storing user role (#7). The column of userRoleTable that contains the username and is referencing userTable will be automatically detected by Tomcat, and doesn’t need to be configured.

* NOTE Make sure that the JAR file containing the JDBC driver referenced by the driverName attribute is placed in Tomcat’s CLASSPATH. To do so, download the driver jar file from the MySQL web site, and copy it to CATALINA\_HOME/lib directory.

To complete this configuration change, stop and restart the Tomcat server.

A JDBCRealm is now configured so it can secure resources your Web applications. At this point, you should be able to log in to the /apress Web application by selecting from the users table the user who has a role of apressuser. The login process and screens in the browser will be exactly the same as with the MemoryRealm examples from the previous section.

BASIC authentication is just what it name implies: basic. It uses the built-in browser form, which you cannot customize using CSS. The password is sent in plain text, encoded using the Base64 algorithm, which is easy to decode. Because of this, BASIC authentication is not often used on production systems. FORM-based authentication is probably the most common type of authentication used on the Internet, including by global web giants, like Google, Facebook, or Yahoo. In the next section, we will take a look at a sample using FORM-based authentication with JDBCRealm configured earlier.

**Configuring FORM-Based Authentication with JDBCRealm**

Let’s first describe how FORM-based authentication works. A user requests a protected resource – a web page that requires login. If the user is not already logged in, the login form is displayed where the user can enter a username and password. After entering these credentials, the container (in our case, Tomcat) checks the entered details against configured database of users (in our case, JDBCRealm checks the database columns for username and password). If the username and password do not match the stored records, an error page is displayed.

If the entered user credentials are matched in the database, user is authenticated. Now that the container knows who the user is, it checks the user’s roles to see if the user can access the requested resource. This is called authorization. If the authenticated user has the required role, the requested page is displayed in the browser. If user does not have the role, an error page is displayed, with HTML status code 403 Forbidden.

Now that we know how the FORM-based authentication works, we can configure our web application to use it. The first step is to implement a login page. Listing 9-10 shows the JSP page with the login form.

***Listing 9-10.*** *Login Form for FORM-Based Authentication*

*<html>*

*<head>*

*<title>Apress Demo</title>*

*<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">*

*</head>*

*<body>*

*<div class="content">*

*<b>Please login to continue</b>*

*<form action="j\_security\_check" method="POST"> #1*

*<table>*

*<tr>*

*<td><label for="username">Username:</label></td>*

*<td><input id="username" type="text" name="j\_username"*

*value=""/><td> #2*

*</tr>*

*<tr>*

*<td><label for="password">Password:</label></td>*

*<td><input id="password" type="password" name="j\_password"*

*/><td> #3*

*<td><input type="submit" value="Login" /></td> #4*

*</tr>*

*</table>*

*</form>*

*</div>*

*</body>*

*</html>*

We are rendering the form using the HTML *<form>* tag (#1). The action attribute is set to *j\_security\_check*, which is the Java Servlet specification naming convention for FORM-based authentication action. Next, we add a text input field for the username (#2). The name of the field must be *j\_username* – again, Java Servlet specification naming convention for the username field. Then we add password field, input with type password (#3) – and named *j\_password* – another convention. These conventions are agreed as part of Java Servlet specification, so all servlet containers (like Tomcat, Jetty, JBoss, and so on) that conform to the specification have well defined configuration for FORM-based authentication. Finally, we need a submit field (#4).

Apart from form action and the names of input fields, a user can customize all other parts of this form. Labels, HTML elements and attributes, JavasSript handlers, and CSS styles all can be edited to match the look and feel requirements of your web application. This is something we couldn’t do with BASIC authentication.

* NOTE FORM-based authentication is not more secure by itself then the BASIC authentication, but it’s much more customizable. You can configure SSL certificates and set up the authentication rules so the username and password are sent encrypted using more secure encoding algorithms.

Let’s save the login page in the separate directory, on path /WEB-INF/secure/login.jsp. Remember that our index page is stored in the /WEB-INF/jsps/index.jsp. We’ll discuss these paths shortly.

Now that we have the login form, we need to implement a page that will be rendered when a user cannot be authenticated – a login error page. Listing 9-11 shows the simple login error JSP page.

***Listing 9-11.*** *Login Error JSP page*

*<html>*

*<head>*

*<title>Apress Demo</title>*

*<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">*

*</head>*

*<body>*

*<div class="content">*

*<b>Login error, please go back to <a href="login.jsp">login page</a>.</b>*

*</div>*

*</body>*

*</html>*

Let’s save the error page in the same directory as the login page, so its path looks like this: /WEB- INF/secure/login-error.jsp.

We have the views that will be rendered in the browser. The last step we have to complete is the configuration of the FORM-based authentication in the web.xml file. As you can see in Listing 9-12, the changed web.xml has a few more lines of XML configuration than the BASIC configuration example.

***Listing 9-12.*** *Configuration of FORM-Based Authentication in the web.xml file*

*<security-constraint>*

*<web-resource-collection>*

*<web-resource-name>JDBC Realm Sample</web-resource-name>*

*<url-pattern>/jsps/\*</url-pattern> #1*

*</web-resource-collection>*

*<auth-constraint>*

*<role-name>apressrole</role-name> #2*

*</auth-constraint>*

*</security-constraint>*

*<login-config>*

*<auth-method>FORM</auth-method> #3*

*<realm-name>Apress</realm-name>*

*<form-login-config>*

*<form-login-page>/secure/login.jsp</form-login-page> #4*

*<form-error-page>/secure/login-error.jsp</form-error-page> #5*

*</form-login-config>*

*</login-config>*

Similarly, as with the BASIC authentication example from the previous section, we first need to configure security-constraint, which consists of the URL pattern (#1) and the role required to access it (#2). Note that we configured the URL pattern using the wild character (\*) to match all pages under the /jsps/ directory.

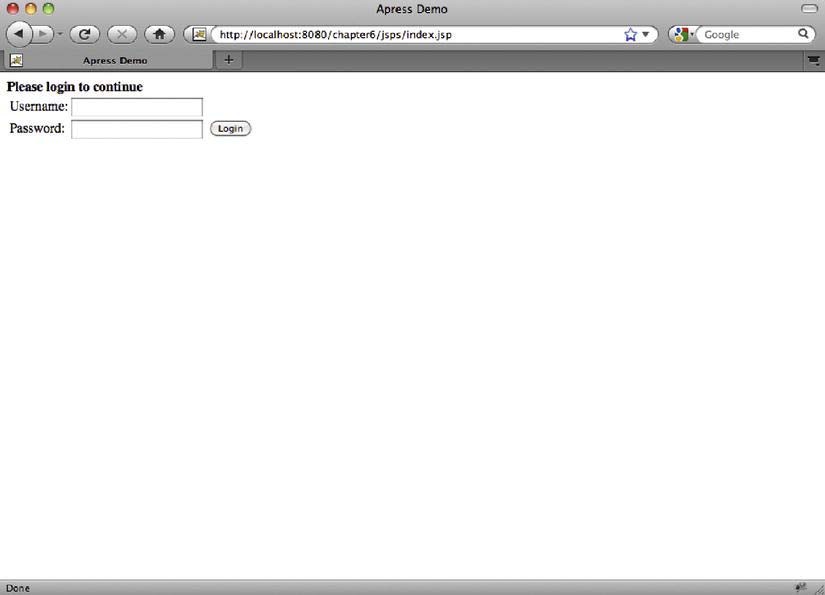
Next, we configure the login-config element. This time we are configuring an authentication method to be FORM based (#3). For FORM-based authentication, two new elements must be set, which haven’t been used in the BASIC authentication example. Those elements are:

* <form-login-page>: Defines the page to be rendered when a user requests a secure resource, but is not authenticated – our login page (#4).
* <form-error-page>: Defines the page to be rendered when user credentials are not recognized – our login error page (#5).

One thing to point out here is how we configured the URL pattern for the secured pages. The URL pattern we configured security for (#1 in code Listing 9-11) should match all URLs that we want to protect. On the other hand, the actual login page (and login error page) must not be protected – so un- authenticated user can access those pages to log in.

Now you understand why we saved the login pages (login.jsp and login-error.jsp) and content pages (index.jsp) in the separate directories, /WEB-INF/secure/ and /WEB-INF/jsps, respectively. It is a common mistake to have a login page that matches the secure URL pattern, so you end up with a login page that requires a user to log in, creating an infinite loop that can never be resolved, and resulting in a server crash. However, all modern browsers can detect infinite redirect loop, and display the error page before the server crashes.

Let’s now see how FORM-based authentication with JDBC realm works in practice. Build the web application into WAR archive, and deploy it to the server using any of the methods we covered in this book so far. For the purpose of this example, we will name our WAR file chapter6.war. Once you have Tomcat started up, navigate to your application in the browser by entering the following URL: http://localhost:8080/chapter6/jsps/index.jsp. Because of our security setup, instead of the welcome page you will be redirected to the login page. Figure 6-4 shows the login page rendered in the browser.



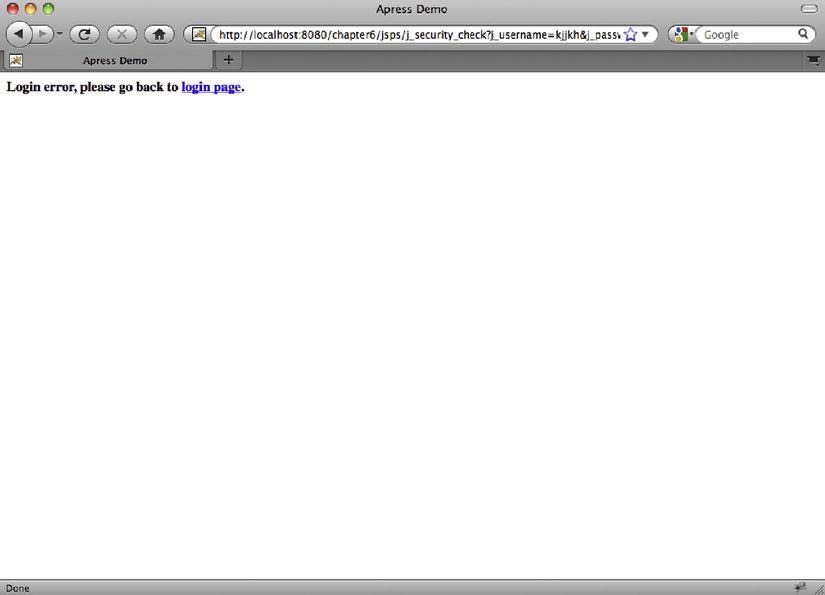
***Figure 9-4.*** *Login page in the browser for FORM-based authentication*

The form in our example looks pretty basic – but there is nothing stopping you from adding CSS styles to the HTML form in Listing 9-10 to make it look however you want.

The next step is to actually log in using username and password. Enter the credential that we inserted in the database for the JDBCRealm; we have one user with required role “apressuser”, with

username “bob” and password “password” (as Listing 9-8 shows). After submitting the form, you will be authenticated and authorized to see the requested page, and the welcome page shown in Figure 9-3 will be displayed in the browser.

In case you enter invalid credentials in the form from Figure 6-4 (an unknown user or username and password that do not match), you will be redirected to the login-error page. Figure 6-5 illustrates the login error page rendered in the browser.



***Figure 6-5.*** *Login error page displayed in the browser after unsuccessful login*

If you log in as a user that does not have required appressuser role, you will see Tomcat’s standard error page with HTML status 403 forbidden, as Figure 9-1 shows.

In the previous section, we discussed that, to logout using BASIC authentication, a user has to close the browser, or to clear the browser history – which can be inconvenient. When using FORM-based authentication, a user can be logged out by invalidating the user’s session. You can use that from the Tomcat’s Manager web application as we saw in Module 4, or you can do it programmatically, from your servlet code. For more information about sessions and how to access them.

**DataSourceRealm**

*DataSourceRealm* is the upgraded version of the *JDBCRealm*, which allows configuration of the database connection as the JNDI resource. JNDI configuration offers more flexibility and easier management – either programmatically (using JNDI API) or via JMX access. Similar to *UserDatabaseRealm*, which is a JNDI configurable version of *MemoryRealm*, *DataSourceRealm* configuration is very similar to *JDBCRealm* configuration, with the addition of JNDI resource configuration. The functionality it offers is the same as that of JDBCRealm, with database tables and columns defined in the same way. To configure *DataSourceRealm*, you only need to configure JNDI data source resource, and use that resource with the security realm configured in the usual way. Listing 9-13 shows the server.xml file with the configured *DataSourceRealm*.

***Listing 9-13.*** *Configuring DataSourceRealm in Tomcat’s server.xml File*

*<GlobalNamingResources>*

*<Resource name="jdbc/tomcatusersdb" auth="Container" #1 type="javax.sql.DataSource"*

*maxActive="100" maxIdle="30" maxWait="10000" username="test" password="test"*

*driverClassName="com.mysql.jdbc.Driver" url="jdbc:mysql://localhost:3306/tomcatusers"/>*

*</GlobalNamingResources>*

*<Context path="/chapter6" docBase="chapter6">*

*<Realm className="org.apache.catalina.realm.DataSourceRealm" #2*

*dataSourceName="jdbc/tomcatusersdb" #3*

*userTable="users"*

*userNameCol="user\_name" userCredCol="user\_pass"* *userRoleTable="user\_roles" roleNameCol="role\_name"*

*/>*

*</Context>*

Similarly to the configuration of the *UserDatabase* JNDI resource in Listing 6-6, first we need to configure the JDBC data source as a global JNDI resource (#1). The attributes of the resources are similar to the database connection parameters we used for the *JDBCRealm* configuration, including database URL, driver class, and username and password credentials. Next we configure the *DataSourceRealm* for the chapter6 *<Context>* element (#2). We set the same attributes as we did for the *JDBCRealm* – namely names of the database tables in which we store the users’ details, and the relevant column names in those tables. The only new attribute we need to add is the *dataSourceName*, defining the name of the JNDI resource we want to use (#3). The value of the *dataSourceName* attribute is set to *jdbc/tomcatusersdb*, same as name attribute of the *<Resource>* element (#1).

And that’s it – we have fully configured *DataSourceRealm*. The examples we used with *JDBCRealm*earlier will all work without any changes.

**The Benefits of Using a JDBCRealm**

*JDBCRealm* and *DataSourceRealm* provide advantages over simple *MemoryRealm* (and *UserDatabaseRealm*) for managing Tomcat users. Using a *JDBCRealm* makes it possible for you to leverage your application’s database as users storage, whereas, in most previously existing web applications, the container of users exists in some proprietary web server database. The management of a user database with more than a dozen users quickly becomes impossible when using *MemoryRealm* and store users in the file. On the other hand, every relational database ships with an administration console with which user information can be easily managed. You can even implement the user management functionalities within your own web application deployed in the same Tomcat instance.

In addition, you can make changes to the SQL user database and have the changes take effect without restarting the Tomcat server. The changes you make to the user data will be visible to the user when he or she logs in. For BASIC authentication, that is when the browser is restart. If using FORM- based authentication, the changes will take effect once a user’s current session expires. When using a MemoryRealm, you must restart the Tomcat server after adding new users, or modifying user information (passwords or roles) in any way.

As a note, the production systems usually don’t use *MemoryRealm* and *JDBCRealm* directly. The common practice is to use one of the JNDI versions of realms – *UserDatabaseRealm* instead of

*MemoryRealm* and *DataSourceRealm* instead of *JDBCRealm*. These two realms provide exactly the same functionality as the two we discussed earlier in this chapter, with the addition of managing the user database as a JNDI resource, treating is as abstract, reusable resource, managed by the Tomcat itself.

Finally, the production systems will usually use some of the advanced realms available in Tomcat, such as *CombinedRealm* (if you have multiple database of users, or want to use both *MemoryRealm* and *JDBCRealm* at the same time); or *LockOutRealm*, which we demonstrated earlier – allowing the built-in protection against brute force attacks from malicious software or users.

**JNDIRealm**

When discussing large production systems, you will often find LDAP server is used for user information storage and management. LDAP stands for Lightweight Directory Access Protocol, and it’s a protocol for accessing directory-based resources over IP networks. The resource, in LDAP terminology, can be anything that is organized as a set of hierarchical records (students at a university, organized across departments, for example; or a list of IP addresses across networks). LDAP is most often used as a user data storage and authentication system within large organizations. LDAP is usually accessed via a JNDI provider, where a JNDI API is used to access LDAP information, analogous to how relational databases are accessed via JDBC.

Tomcat supports authentication using LDAP server via its *JNDIRealm* implementation.

*JNDIRealm* is configured to access an LDAP directory via a JNDI provider and provide data for the authentication and authorization of users in Tomcat. Detailed discussion about LDAP directory server and its configuration is beyond the scope of this book, but we will demonstrate a simple *JNDIRealm* configuration without going into depth of LDAP settings.

Similarly to other security realm configurations, all we need to do it add a *<Realm>* element in the *CATALINA\_HOME/conf/server.xml* file. Listing 6-14 illustrates the *JNDIRealm* configuration in the server.xml file.

***Listing 6-14.*** *Sample JNDIRealm Configuration for LDAP Authentication*

*<Realm className="org.apache.catalina.realm.JNDIRealm #1*

*connectionURL="ldap://apress.com:389 #2*

*userPattern="uid={0},ou=users,dc=apress,dc=com" #3*

*roleBase="ou=groups,dc=apress,dc=com" #4*

*roleSearch="(uniqueMember={0})" /> #5*

Firstly, we configure the *<Realm>* element with the *JNDIRealm* full class name (#1). Next, we set the connectionURL attribute, which sets the URL where the LDAP server is running. Then, we set the userPattern attribute, which contains the pattern for the distinguished name directory entry for locating the records (#3). In the pattern, {0} will be replaced by the actual username. Attribute roleBase specifies the LDAP directory entry where information about user roles can be retrieved (#4). Finally, the roleSearch attribute defines the LDAP filter for searching the user roles (#5).

There are numerous LDAP servers implementations available, such as Microsoft Active Directory and Apple Directory. If you are looking to to explore LDAP in more detail, we suggest looking at the most popular open source implementation *OpenLDAP* (www.openldap.org). With its large user base and open source support, you can find a lot of online resources and configuration examples for Apache Tomcat. In addition, Apache Tomcat’s online documentation has a good example of the LDAP setting for *OpenLDAP* server and corresponding *JNDIRealm* configuration (<http://tomcat.apache.org/tomcat-7.0-> doc/realm-howto.html#JNDIRealm).

**Accessing an Authenticated User**

Once a user has been authenticated, it is very easy to access the user’s information using the *HttpServletRequest* interface. Because the user’s information is stored in the *HttpServletRequest* object, it is available to all JSPs and servlets existing in the same request. To see how this information is accessed, we are going to edit the index.jsp page we used in throughout this chapter. The modified index.jsp can be found in Listing 9-15.

***Listing 9-15.*** *The Modified index.jsp Page*

*<html>*

*<head>*

*<title>Apress Demo</title>*

*<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">*

*</head>*

*<body>*

*<div class="content">*

*<b>Welcome to Apress, <%= request.getRemoteUser() %></b> #1*

*</div>*

*</body>*

*</html>*

To illustrate the access of the authentication information on the JSP page, we have embedded a bit of Java code in the JSP page to print the logged in user on the screen (#1). The request variable used is a reference to the current HttpServletRequest object. It’s implicitly available to all JSP pages (by Tomcat), so we can access it, and all of its properties within the JSP page.

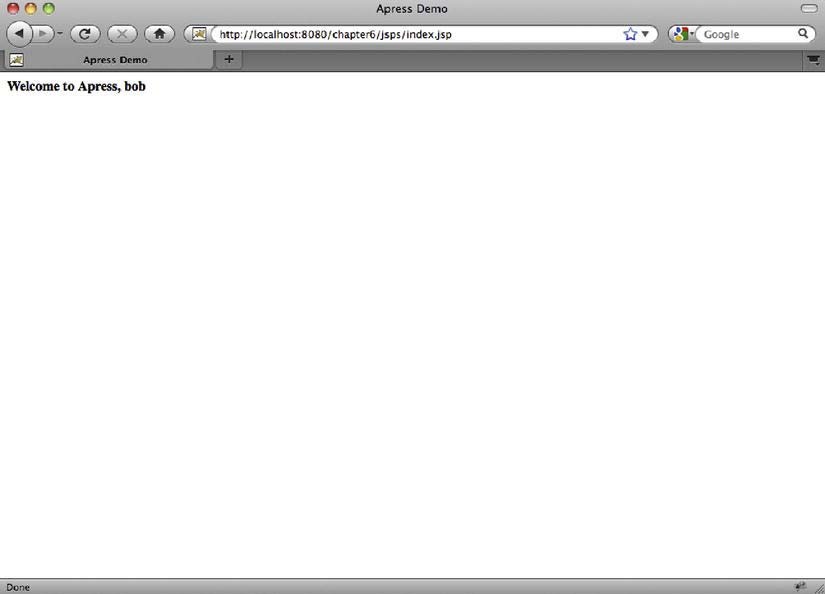
This code uses the request.getRemoteUser() method to retrieve the authenticated user’s username.

It then outputs the returned username. After you have made the changes to this JSP, copy it to the

*<CATALINA\_HOME>/webapps/apress/* directory and point your browser to the following URL:

*http://localhost:8080/apress/index.jsp.*

If you have already been authenticated, you should see the screen shown in Figure 6-6 with the username with which you were authenticated following the Welcome to Apress text. If you have not been authenticated in the /apress Web application, enter bob and password in the BASIC authentication dialog box.



***Figure 6-6.*** *The Modified welcome.jsp page shows the effect of retrieving the username from a security realm.*

In case you call this method when there is no authenticated user, it will return null. In addition to username of the authenticated user, you can also check if the user is in the required role programmatically, by querying *HttpServletRequest* object. The method *HttpServletRequest.isUserInRole*(String roleName) returns true if the authenticated user had the role specifies with argument, and false otherwise.

**Introduction to SSL**

Secure Socket Layer (SSL) is a secure transfer protocol used for communication on the Internet using cryptographic methods. SSL was first developed by Netscape to make its own products secure. After its public release, it was quickly adopted by a number of big players in the IT industry, including Microsoft, and it became the de-facto standard for Internet traffic encryption.

* Note More recently, the Internet Engineering Task Force (IETF), the organization that develops and promotes Internet standards, developed Transport Level Security (TLS) based on SSL. TLS is a community-standardized protocol and will probably eventually replace SSL. Because almost all current server and client applications support both TLS and SSL, you can decide which one to use. Although there is little difference between the SSL and TLS protocols, TLS is backward compatible with SSL, and thus a safer option. However, due to its widespread usage, SSL is still the more commonly used secure protocol for web applications. For that reason, and to avoid confusion, we’re using the term SSL to refer to the secure communication protocol throughout this chapter.

**What SSL Does**

The main purpose of the SSL protocol is to guarantee that no one can eavesdrop on or tamper with the communication between a browser and the server where the web application is deployed. When accessing the web sites secured by SSL, a user can be sure that no one can intercept and read the information passed to the remote server; for example, usernames and passwords or credit card information when using e-commerce web sites. In addition, the user is safe to send and receive any sensitive information to and from web server, knowing that no one could have tampered with the information during transport, or can change the transported content in any way.

Another purpose of secure communication is the ability to authenticate the server and its owner based on the SSL information – so that a user can be certain that the server that it’s accessing is the one that it’s saying it is. This has become very important in today’s Internet-dependent society, so that we are sure that we are accessing our bank’s web site, for example, and not some malicious web site representing itself as our bank. The term for a site that is masquerading itself as another, tricking a user to pass sensitive information to it, is phishing.

**How SSL works**

SSL is a cryptographic protocol, using symmetric pair of keys to encrypt and decrypt traffic sent over the Internet.

In a common SSL scenario, when the user accesses the web server for the first time, the server sends its SSL certificate, or public key, to the client. The SSL certificate contains the information about the server, its owner, company, and its validity period. A user can reject a certificate if it does not trust its authenticity, effectively terminating the connection.

If the user accepts the certificate, the certificate itself is stored in the browser, and is used to initiate a secure connection with the issuing server. This key is public, as the server sends it to anyone that asks for it.

The information encoded using the public key can only be decoded using the symmetric private key. The private key is, as its name suggests, private, and kept safely on the server. A client generates the symmetric key, with which both client and server will encrypt all traffic sent to the other side. With the symmetric key, content is encoded and decoded using the same key known to both parties in the communication, which is less secure than the asymmetric public/private key communication. That’s why it’s important that the exchange of the symmetric keys is secure. Therefore, a client encodes the symmetric key using the server’s public key (received in the certificate), and the server encodes it using its private key.

When a server receives the symmetric key, the connection between server and client is secure, and all traffic sent from the browser to the web application, and response content from the web server to the browser, will be encrypted using the generated symmetric key known to client and server only. The exchanged symmetric key is valid only for the duration of the current session, and needs to be re- generated every time a user initiates a new session.

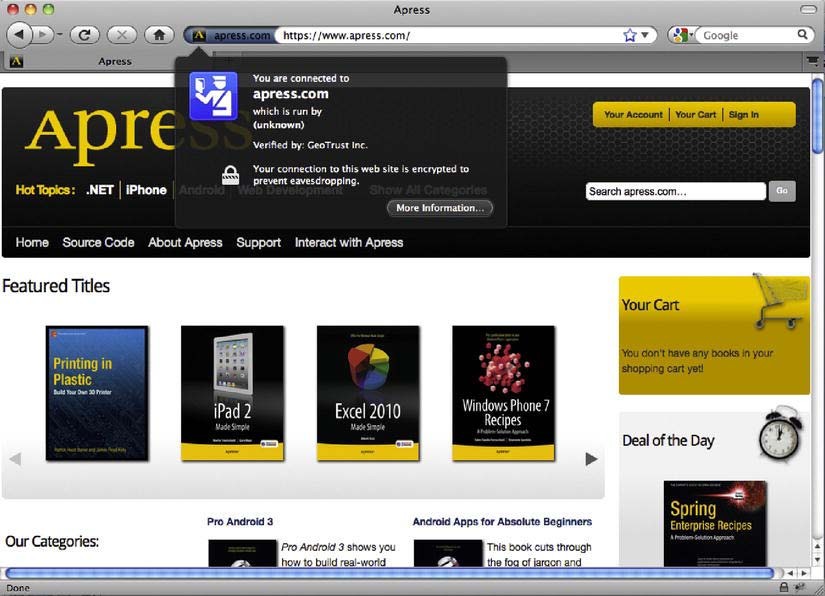
The security of SSL depends on how secure the key used to encrypt data is. SSL is not unbreakable in theory, but with the amount of time required to break standard 128-bit key, in practice it is often assumed as safe from brute-force attacks. For the industry-standard 128-bit key, there are 2128 combinations possible to generate the key. The value of 2128 calculates to 340,282,366,920,938,463,463,374,607,431,768,211,456, or roughly 240 trillion trillion trillions! To put things into perspective, today’s super computers can break the DES encryption algorithm, which uses 56-bit long keys, in around one day. If, in the future, man develops a machine that can perform the same task of breaking a 56-bit key in one second, it would still take 149.7 trillion years to brute force a 128-bit key!

SSL protocol communication over HTTP protocol is referred to as HTTPS (secure HTTP). The web sites that are using SSL encrypted connections display https as the protocol name in the browser’s address bar, for example http[s://www.mybank.com](http://www.mybank.com/).

Ensuring that the connection is secure from eavesdropping and tampering, using asymmetric and symmetric cryptography methods is the main purpose of using SSL. However, there is another useful purpose of SSL-certificate protected web sites: the ability to authenticate that the site is what it says it is. SSL certificates, sent by the secure server as public keys to the client, contain basic information about the site to which they belong, such as the domain name, owner name, and company name.

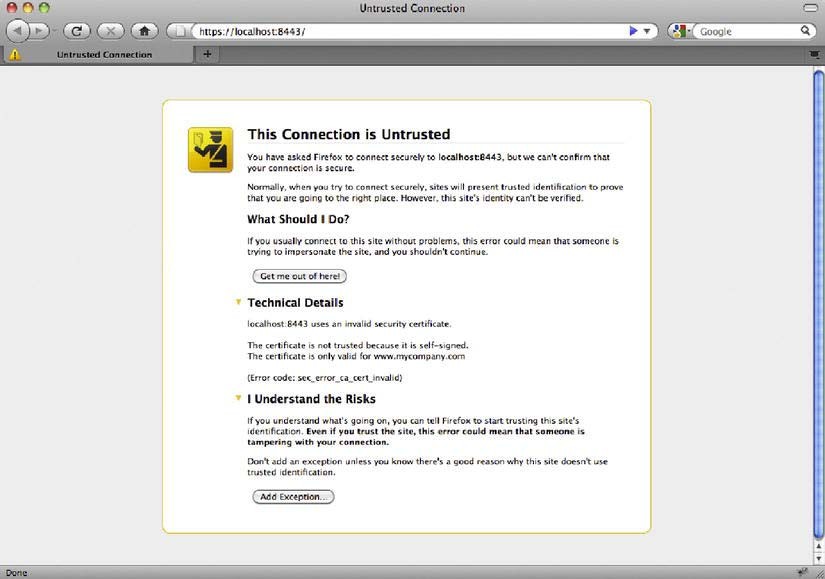
Organizations called Certificate Authorities (CA) can authenticate the details of the SSL certificate, so if the user trusts the CA, they can be sure that the secure web site is certified, and its details are correct. There is a number of CAs that can issue a certified SSL certificate. Modern browsers automatically recognize the largest and best-known CAs, and allow connections to the sites providing SSL certificates certified by these organizations automatically.

The secure icon and the registered domain name are displayed in the browser’s address bar if the SSL connection is active. Figure 7-1 shows an example in the Firefox web browser. (Note that the company name is not a mandatory field, as indicated by the “unknown” reference in Figure 7-1.)



***Figure 7-1.*** *The secure connection icon and the registered domain name from the certificate are displayed in the browser’s address bar.*

If the SSL certificate is not certified by a CA, or is certified by the CA but not recognized by the user’s browser, the user will be presented with a warning screen, where he or she can decide whether to trust the certificate. Figure 7-2 shows the warning received if the certificate is not automatically accepted by the browser.



***Figure 7-2.*** *If the SSL certificate cannot be accepted automatically, a warning is displayed to the user.*

**Configuring Tomcat with SSL**

In order to use SSL for server access, we need to create our own public key, which we will use as a certificate. The certificate we are going to create is usually called a self-signed certificate, as it will not be verified with an independent CA, so we cannot authorize as web site owners with it. But, having self- signed certificate is sufficient enough to have SSL encrypted communication with our server. For development and testing purposes, or for internal applications not accessible outside the company’s network, this is often enough to provide a usable, secure environment.

**Creating Keystore with SSL Certificate**

All publicly known certificates are stored in a repository called the keystore. The keystore is a file that contains the public and private key data required to encode and decode information using SSL or TLS. Applications that are using SSL must be able to read the keystore file, and use its data as keys to encrypt and decrypt information. The key data in the keystore file can be stored in a number of formats that depend on the tool used to create keystore. Apache Tomcat 7 can read keystores in one of the following formats: PKCS11, PKCS12, and JSK. PKCS11 and PKCS12 formats belong to the standards called Public- Key Cryptography Standards, developed by the RSA Laboratories, which is one of the leading network security companies in the world. JKS format stands for Java KeyStore, which is a Java-specific keystore format. JKS keystore can be created and manipulated using the keytool utility application, distributed as part of Java SDK from version 1.4. Because JKS is simple enough to use and easily accessible as part of Java SDK, we will be using its keytool application to create JKS keystores, which we will use to configure SSL on Tomcat.

Keytool, which we will use to create a self-signed SSL certificate, is located in the JAVA\_HOME/bin/ directory. The file name is keytool.exe (on Windows systems), or just keytool, with no extension (on Unix-based systems). Because JAVA\_HOME/bin is on the systems path by default after Java installation, you can execute it from any directory on the file system. In order to check if the keytool is available on your system, just type keytool on the command line in the terminal window. You should see the list of all required and optional command line arguments you can use to invoke the application.

Example 9-1 shows the command with the required arguments that you should execute for Windows and Unix-based systems, respectively.

***Example 9-1.*** *The Commands to Generate a New, Self-Signed Certificate in a JKS Keystore*

**OS Command**

*keytool -genkey -alias tomcat -keyalg RSA -keystore /home/aleksav/mykeystore*

We are using the keytool to generate a new SSL certificate by specifying the -genkeypair argument. The genkeypair argument was named genkey in earlier versions of the Java SDK (earlier than version 1.6), and it is still supported for backward compatibility. Because one keystore can contain multiple public/private key pairs, we will be using the alias “tomcat” to identify our new certificate within the keystore (by specifying the command argument -alias tomcat). We can also specify the algorithm used to generate the key pair, using the *–keyalg* argument. We will be using the RSA algorithm, a commonly used cryptography algorithm for public and private key generation. Finally, we will specify the file where our keystore will be saved using the *-keystore* argument. If you omit this argument, new file named keystore, will be created in the home directory of the user running the command.

When you press Enter, you will first be prompted for the keystore password. This password will be used to access any certificate stored in the created keystore, so use the usual precaution when deciding on the keystore password. For the purpose of this example, we will set the keystore password to abc123.

After that, you will be prompted to enter some information about the certificate, the owner’s name, the company, and location (country). You are first asked to enter your first and last name. The information you enter is stored in the keystore as the common name (CN) of the certificate. Although it looks confusing, you should enter the host name for the server for which you are generating a certificate. The reason for this is that, according to HTTPS specification, browsers should match the CN of the provided certificate to the domain name of the requested URL. If they do not match, the user will be presented with a warning in the browser that states that the certificate does not match the domain name of the site they are accessing. Because this decreases the amount of trust users have in the web site, the warning is not something that is desired for the public web site. To avoid this, make sure that any certificate you generate has a valid domain name as the CN, confusingly set as the first and last name value in the keytool interface (see the first question in Figure 9-3).

* Note You can only have one SSL certificate for one IP address. If you host multiple domains on the same IP, only one of these host names can have a valid SSL certificate that matches its domain name. If you try to use SSL for any other domain name on the same IP, the browser will display a warning that the domain name does not match the certificate. This is a known limitation of SSL, because an SSL protocol handshake must happen before the hostname is extracted from the HTTP request.

Once you enter all the required details, you will be asked for another password, this time the password for this particular certificate within the keystore. For this example, we will set this password to tomcat123. Both the keystore password we set earlier and the certificate password are used for Tomcat configuration later, so remember them or write them down.

Once completed, you will have the chance to confirm all details entered. Once these are confirmed, you will have your certificate generated and stored in the file you specified as the –keystore argument.

Now that we have the SSL certificate generated, we can configure Tomcat to use it for SSL server access.

**Configuring Tomcat’s SSL Connector**

The first step in configuring SSL with Tomcat is to configure Tomcat’s Connector, which will handle SSL connections over HTTPS. To configure the <Connector> element for SSL, all you have to do is set the SSLEnabled attribute to true, and set the scheme and secure attributes in order that the SSL information is correctly interpreted by the servlets deployed in Tomcat. Along with these basic attributes, you can configure additional details for your SSL configuration relating to the keystore and certificate we generated in the previous section. Listing 9-1 shows typical Tomcat SSL Connector configuration in the server.xml file.

***Listing 9-1.*** *Tomcat’s SSL Connector Configuration Using Non-Blocking Java SSL Implementation*

*<Connector*

*SSLEnabled="true" #1*

*scheme="https" secure="true" #2*

*port="8443" #3*

*protocol="HTTP/1.1" #4*

*clientAuth="false" #5*

*keystoreFile="/var/mykeystore" keystorePass="abc123" #6*

*keyAlias="tomcat" keyPass="tomcat123" #7*

*/>*

We first set the SSLEnabled attribute to true, telling Tomcat that this connector handles SSL connections (#1). In addition, we set the scheme attribute to value https, so that all SSL-secure URLs are recognized by the standard https:// protocol identifier; and we need to set secure attribute to true, if we wish to check for an SSL connection within Java servlet code, by calling *HttpServletRequest.isSecure*() method (#2). Next, we set the port on which the SSL connector will accept incoming connections (#3). Because 443 is the common port number for HTTPS connections, Tomcat itself uses the slight variation 8443 (using the same convention for port 8080 for standard HTTP request, as opposed to the “common” port 80, adding 8000 for the standardized port).

The next bit is the protocol configuration. We used standard HTTP/1.1 protocol, which is implemented in Tomcat’s class *org.apache.coyote.http11.Http11Protocol* (#4). Http11Protocol is the standard blocking Java HTTP protocol. It is blocking because it uses a single thread to process the incoming requests and dispatch the response back to the browser. You have a choice to use a non- blocking version of Java Tomcat protocol instead, or even an APR Tomcat native protocol. Non-blocking Java protocol must reference its implementing class as the protocol attribute value:

*org.apache.coyote.http11.Http11NioProtocol*. To use Tomcat’s native APR protocol, you must make sure you have Tomcat’s native library on the Tomcat classpath, and reference the APR HTTP connector class name as protocol attribute value: org.apache.coyote.http11.Http11AprProtocol.

* **Note** In order to use the Tomcat native library, you must compile it from the source code for your specific operating system. The native library compilation is beyond the scope of this book, but you can consult Tomcat’s native documentation about the required steps at [http://tomcat.apache.org/native-doc/.](http://tomcat.apache.org/native-doc/)

The next attribute we configure is the clientAuth attribute (#5). This attribute specifies whether we are using one-way secure certificate authentication or if the client needs to present the valid SSL certificate to the server as well. When using SSL for web applications publicly accessible on the Internet, we don’t require users to authenticate to the server using SSL certificate; to establish a secure connection, it is enough that the client accepts the certificate from the server. However, sometimes the server requires the client to be authenticated as well, usually in a business-to-business scenario, where both client and server are actually computer programs, without any human influence. For this example, we will configure standard public web site SSL connectivity by setting the clientAuth attribute to false.

Finally, we need to set the attributes relating to the keystore and the certificate our Tomcat server will use to encrypt and decrypt SSL traffic. We configure the keystore file path by specifying the keystoreFile attribute, setting it to the absolute path of the keystore file we created in the previous section (#6). At the same time, we specify the password we used for the keystore. And, as the last step, we specify the alias of the actual SSL certificate that can be used to identify it within the specified keystore (#7). We must use the same alias used when creating the certificate, as well as the password for the certificate itself. As you can see, we used the same values as specified when running the keytool application in the previous section.

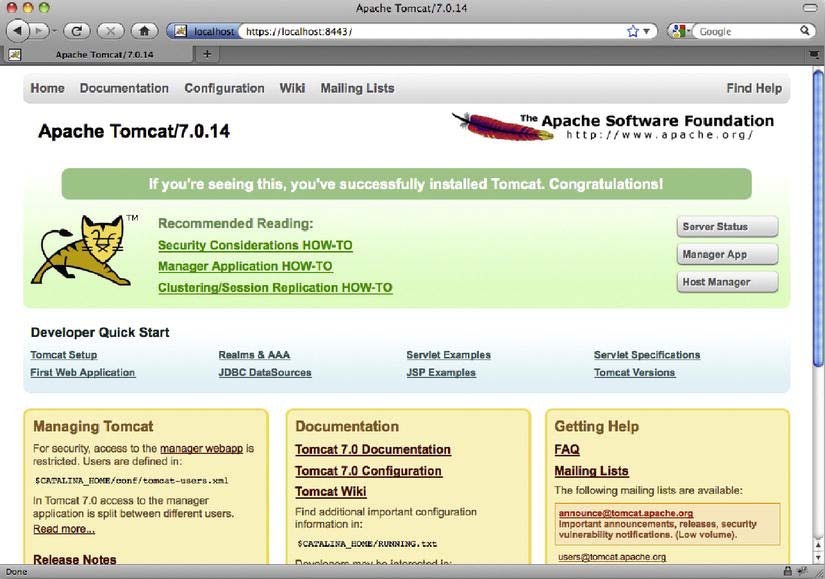
Now we need to start Tomcat (or restart it if it’s already running). After the startup is completed, you will be able to access Tomcat’s home page by typing https://localhost:8443 URL to your browser’s address bar.

* Note Make sure you start the web address with https for the protocol name to access the application correctly. If by mistake you use http (for example http://localhost:8443), you will end up with the error page in the browser, because Tomcat expects any URL with HTTP protocol to use port 8080 (and at the same time, any URL with HTTPS protocol must use port 8443). Any other combination will not be recognized by Tomcat as a valid URL.

When you do that for the first time, you will be presented with a warning, exactly the same as the warning shown in Figure 7-2. This is expected, for two reasons. First, we are using a self-signed certificate that has not been validated by any CA, so the browser will warn the user about that. Second, we created the certificate by specifying CN record as [www.mycompany.com,](http://www.mycompany.com/) and now we access it using the localhost domain name. As expected, the browser is displaying a warning that the CN domain name and the domain name used to access the site do not match.

In order to proceed, you will need to add exception for this certificate by confirming to the browser that you indeed trust this site. After adding the SSL exception, you will be presented with the familiar Tomcat home page, but this time using HTTPS secure protocol on the configured port 8443.

Figure 9-4 shows the browser windows when you access the Tomcat’s home page using the HTTPS secure protocol.



***Figure 7-4.*** *Tomcat’s home page access over SSL on port 8443*

If you change the port name to 8080 and the protocol to HTTP, you will notice that you can still access Tomcat’s home page using the standard HTTP protocol as before on http://localhost:8080.

With the configuration so far, we have enabled the SSL on Tomcat, and users can choose whether to access the site over HTTP or HTTPS. However, for real-world secure applications, we would like to force users to use the SSL-secured protocol when accessing pages with security-critical information. For example, when users are logging in or when users are typing credit card details in the HTML form. In the next section, we are going to configure our sample web application to do precisely that: force users to use the secure protocol for required pages.

**Configuring Secure Resources in the Web Application**

To demonstrate forcing the use of the HTTPS protocol, we are going to use the sample web application from the previous chapter, the FORM-based authentication example. We will improve the FORM-based login example by forcing Tomcat to load and submit a login page, with user username and password, over SSL. Our login page is located at /WEB-INF/secure/login.jsp file. All we need to do is to specify a new security constraint in the web deployment descriptor (web.xml). Listing 7-2 shows the updated web.xml file from the FORM-based login example, with the added SSL configuration in bold.

***Listing 7-2.*** *Updated Web Configuration to Load Login Pages Using Secure Channel*

*<security-constraint>*

*<web-resource-collection>*

*<web-resource-name>JDBC Realm Sample with SSL</web-resource-name>*

*<url-pattern>/jsps/\*</url-pattern> #1*

*</web-resource-collection>*

*<auth-constraint>*

*<role-name>apressuser</role-name>*

*</auth-constraint>*

*<user-data-constraint> #2*

*<transport-guarantee>CONFIDENTIAL</transport-guarantee> #3*

*</user-data-constraint>*

*</security-constraint>*

*<login-config>*

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

*<realm-name>Apress</realm-name>*

*<form-login-config>*

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

*<form-error-page>/secure/login-error.jsp</form-error-page>*

*</form-login-config>*

*</login-config>*

We are adding SSL configuration to the existing web resource collection, matching all JSP pages in the /jsps directory (#1). For all resources in the configured collection, we apply a security constraint using <user-data-constraint> element (#2). This element requires that transport guarantee be configured, which we set to value *CONFIDENTAL* (#3). By setting transport guarantee to *CONFIDENTAL*, we are telling Tomcat to force secure channel (HTTPS) for all resources that match the <web-resource- collection> configuration. So, if a user tries to load the login page over HTTP on port 8080 (by typing the address http://localhost:8080/chapter7/jsps/index.jsp), Tomcat will recognize that this URL matches the pattern set in the web.xml, and automatically redirect user to the secure URL https://localhost:8443/chapter7/jsps/index.jsp. The configured realm will kick in and the user will be presented with the login screen. After login, all password-protected pages will be loaded over the SSL protocol.

We used value *CONFIDENTAL* to specify the transport guarantee for the configured secure pages, which is the most restrictive setting, guaranteeing that the data hasn’t been tampered with in transit, and that no unauthorized person has read the content while it was in transit. If your security policy allows more relaxed settings, you can use value INTEGRAL, which only guarantees that the content hasn’t been changed while being transported between client and server. The default value is NONE, which disables any security checks.

Although we generated the keystore and the SSL certificate system wide, and configured for the entire Tomcat server, we specified the required security constraints on the web application level, giving developers the freedom to configure web application SSL security as required. SSL encryption and decryption is not a trivial operation, and it adds performance overhead to all requests over HTTPS protocol. By configuring SSL within the web application only for pages that should be secure, leaving access to ordinary public pages without SSL, you can benefit by having secure access to the pages that required such protection, while not affecting the performance of the public pages that have no security- critical content. Common examples of such configuration are using SSL for the password-protected parts of the site, or accessing checkout pages over HTTPS only, so that a user can enter his or her credit card details without fear of someone unauthorized accessing it during transport to the web server.

**Installing a Certificate from the Certificate Authority**

Using a self-signed SSL certificate will enable us to use the full power of SSL to secure our web application, for development and testing purposes, or for internal web applications used within a private network. However, if we were using self-signed certificate to protect a public web site, we would probably end up with users trusting us less than ever. This is because their browsers will

display a warning whenever they access our site, letting them know that the browser cannot determine the authenticity of the certificate. To avoid this, we need to use the certificate signed (therefore authenticated) by the known CA.

**Obtaining a CA-Signed Certificate**

The first step in the process is to generate the self-signed certificate using the keytool, as we just did in the previous section.

Then, we need to generate a certificate-signing request (CSR) for the generated self-signed certificate, and submit it to the chosen CA. We will use the same keytool to generate CSR. The only difference when generating this self-signed certificate is the different command specified: certreq instead of genkey. The following code snipped shows the example CSR command:

*keytool -certreq -keyalg RSA -alias tomcat -file mycompany\_csr.csr –keystore /var/mykeystore.*

It is important to specify the same keystore where you stored the certificate generated in the previous step. In addition, you have to use the alias with which you stored your self-signed certificate. The CSR will be saved to the file specified with the –file argument, in our case *mycompany\_csr.csr*.

Now you need to submit the generated CSR to your CA, and pay for the service, of course. When the transaction is complete, you will receive your new certificate, signed by the well-known CA.

**Importing the CA-Signed Certificate**

Before we can import the new certificate to the keystore, we need to import the so-called chained certificate, or root certificate, for the CA we used. The chain certificate is publicly available to download from your CA’s web site, and it’s used to verify that the CA that will, in turn, authenticate our certificate to the browser. Let’s say we downloaded the chain certificate to the /var/chaincert.crt file; you can import this to the keystore using the following command:

*#keytool -import -alias root -keystore /var/mykeystore -trustcacerts –file /var.chaincert.crt*

You can pick the alias to import the chain certificate to the keystore, but it is important th=99999 at it doesn’t already exist in the keystore. By specifying the argument *–trustcacerts*, we are setting this certificate as the “chain trust,” so it can be used to certify other certificates.

Finally, we can import the certificate received from the CA using the required alias. The following code snippet demonstrates the command used to import valid certificate:

*keytool -import -alias tomcat -keystore /var/mykeystore -file /var/myca-cert.crt*

Now you can restart Tomcat and, assuming the keystore and the certificate alias and passwords are correctly set, your new, signed SSL certificate will be used for HTTPS connections. Because the trusted CA signed the certificate, no warning will be displayed in the browser when users access your web application using HTTPS.

## Secure Session Tracking with Tomcat

When we discussed HTTP sessions in Chapter, we described the session-tracking mechanism available in Tomcat, namely cookie session tracking and URL rewriting. When using cookies,

which is the default mechanism, the session identifier is stored in the JSESSIONID cookie in the browser, and its value is sent to the Tomcat on every request. With URL rewriting, the session identifier is passed as request parameter with every URL pointing to a web application deployed in Tomcat; it requires that every link displayed on the web page has the session request parameter encoded in the URL.

From version 7, Tomcat supports a third mechanism: using SSL protocol for session tracking. SSL session tracking is actually part of Java Servlet specification 3.0, which Tomcat 7 supports, and can be used for session tracking on web sites that are using SSL protocol to encrypt/decrypt information sent over HTTP. Therefore, if your web application is not using SSL exclusively, you will not be able to use the SSL session-tracking mechanism.

When tracking sessions using SSL, the session identifier actually is the id of the established SSL session, generated by the server at the start of the SSL handshake, when the client (browser) requests the SSL certificate from the server. The SSL session identifier is passed over an encrypted SSL connection along with all other traffic; therefore, this is the most secure way to track a session between server and client.

* Note You can only use SSL session tracking with standard Java connectors, both blocking (BIO) and non- blocking (NIO). The APR Tomcat native connector does not yet support the SSL session tracking.

However, there is a bit of configuration required to enable Tomcat to use the SSL session tracking mechanism. By default, Tomcat checks all available session-tracking modes before deciding which strategy to use. In addition, cookie strategy and URL rewriting mechanisms take precedence over SSL session tracking. So, to use SSL session tracking with your web application, you must explicitly set this mode as the only one available using custom ServletContextListener. Listing 7-3 illustrates the implementation of this listener, and its configuration in the web.xml file.

***Listing 7-3.*** *Configuring ServletContextListener for SSL Session Tracking*

public class SSLSessionTrackingContextListener implements ServletContextListener { public void contextInitialized(ServletContextEvent event) {

ServletContext context = event.getServletContext();

EnumSet<SessionTrackingMode> modes = EnumSet.of(SessionTrackingMode.SSL); context.setSessionTrackingModes(modes);

}

public void contextDestroyed(ServletContextEvent event) {

// NOOP

}

}

…WEB.XML:

<listener>

<listener-class>

com.appress.apachetomcat7.chapter7.SSLSessionTrackingContextListener

</listener-class>

</listener>

As per Java Server specification, the configured listener will be instantiated, and its

contextInitialized() will be executed before the rest of the web application is configured. This will ensure that Tomcat uses SSL session identifier to track session in our web application.

And that’s it. When you deploy the web application to Tomcat, the session tracking will be performed using the SSL session identifier.

To demonstrate this, we will display the session id in our JSP page. The SSL session id is available as request attribute javax.servlet.request.ssl\_session. All we have to do is to add the following code

snippet to our JSP page:

SSL Session ID: <%= request.getAttribute("javax.servlet.request.ssl\_session") %>

Figure 7-5 shows the SSL session id displayed in the browser.



***Figure 7-5.*** *SSL session id accessed as request attribute in the JSP file*

When using SSL session tracking, you should pay special attention to session invalidation. You have to make sure you invalidate both standard HTTP session and Tomcat’s SSL session. At the moment, this can only be done using the Tomcat-specific API. In addition, you will need to explicitly close the connection to the client by sending the Connection=close header, as the low-level SSL session will be active until the connection is closed. Listing 7-4 illustrates the code that you will need to write in order to invalidate SSL session correctly.

***Listing 7-4.*** *Invalidating an SSL Session Using Tomcat’s API*

*HttpSession session = request.getSession(true);*

*// invalidate standard HTTP session session.invalidate();*

*// Invalidate the SSL session SSLSessionManager mgr = (SSLSessionManager)*

*request.getAttribute("javax.servlet.request.ssl\_session\_mgr"); mgr.invalidateSession();*

*// Close the conection response.setHeader("Connection", "close");*

We use Tomcat’s *SSLSessionManager* class to invalidate the underlying SSL session. You need to have Tomcat’s jar library *tomcat-coyote.jar* on the project classpath in order to compile this code.

SSL session tracking has a limitation when used in session replication scenarios, when multiple Tomcat instances are deployed in a cluster. This is because, when a Tomcat instance crashes, session replication will not work when another node takes over users from the failed instance. Because of the way SSL protocol works, SSL session id will be different on each server node, so the existing session id from the client cannot be associated with the replicated session.