Pro Apache Geronimo

Kishore Kumar

Apress®

Pro Apache Geronimo

Copyright © 2006 by Kishore Kumar

ISBN-13 (pbk): 978-1-59059-642-5

ISBN-10 (pbk): 1-59059-642-0

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewer: Harshad Oak, Dilip Thomas
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Manager: Beth Christmas
Copy Edit Manager: Nicole LeClerc
Copy Editor: Elizabeth Cate
Assistant Production Director: Kari Brooks-Copony
Production Editor: Kelly Gunther
Compositor: Linda Weidemann, Wolf Creek Press
Proofreader: Elizabeth Berry
Indexer: Rebecca Plunkett
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Source Code section.

CHAPTER 3

Web Application Development with Geronimo

Using Apache Geronimo, you can deploy and run various application components, like web applications, EJB components, J2EE connectors, J2EE client application components, and J2EE enterprise application components. Geronimo provides a web application container to host your web applications. It also enables your web application to connect to any EIS, including relational databases and messaging middleware systems. Additionally, your applications can utilize the web security infrastructure Geronimo provides.

In this chapter, we will discuss how to use Geronimo to deploy and run your web applications. You'll consider the steps involved by looking at the development of a sample application. Next, you'll learn how to configure database connectivity from a web application. Finally, we'll discuss connection pools and web application security.

J2EE Web Applications

J2EE web applications consist of one or more servlets, JSPs, and resource files (including static HTML pages, images, and JavaScript pages). In addition, a J2EE web application must provide a standard deployment descriptor file to describe the application parameters to the hosting container. Optionally, it can provide a container-specific deployment descriptor to define container-specific application parameters.

Servlets are Java classes that provide dynamic extension capabilities to the web server. JSPs are template files that provide a convenient mechanism for creating dynamic content. They are ultimately run as servlets by the container.

Geronimo supports the Servlet 2.4 and JSP 2.0 specifications. The open source web containers Jetty and Apache Tomcat provide Geronimo's web application support. As of this writing, Geronimo is preconfigured to work with the Jetty and Tomcat web containers, and each is available in a separate download of Geronimo.

Developing and Running a Sample Application

The general steps involved in creating a J2EE web application are as follows:

1. Develop the web application (servlets, JSPs, filters, Java helper classes, and resource files).
2. Develop the web application standard deployment descriptor (web.xml).
3. Develop the optional Geronimo-specific deployment descriptor (geronimo-web.xml).
4. Compile the web application (including helper classes).
5. Package the application to a standard deployable unit (a WAR file).
6. Deploy the application by using the Geronimo deployer tool.
7. Access the application by using its URL.

Let's consider these steps in more detail by creating a sample Hello World application.

Develop the Web Application

Let's say our sample application has the following components and features:

- It has a front controller servlet that accepts the user requests, executes a command, and forwards to the next view to be displayed back to the user.
- Dynamic logic that needs to be executed is implemented as a command class (in this case, a WelcomeCommand class).
- When the command class is executed, it creates and returns a model object (in this case, a String message).
- The welcome.jsp view implements the presentation logic.

The controller servlet is shown here:

```
public class FrontController extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse ➥
response) throws ServletException, IOException{
    String commandName=request.getParameter("command");
    Command command=null;
    if("welcome".equals(commandName)){
      command=new WelcomeCommand();
    } else {
      // handle this case - maybe use a default command
    }
    Object msg=command.execute(request);
    request.setAttribute("message",msg);
    String nextView=command.getNextView(request,msg);
    RequestDispatcher rd=getServletContext().➥
getRequestDispatcher(nextView);
    rd.forward(request,response);
  }
}
```

The controller servlet instantiates an appropriate command instance, depending on the request parameter, and executes it. The WelcomeCommand returns the welcome message as the execution result. The controller uses the command to find the next view and then dispatches the control to this view.

Create the WelcomeCommand class as shown here:

```
public class WelcomeCommand implements Command {
  public Object execute(HttpServletRequest request){
    return "Hello World !!!";
  }
  public String getNextView(HttpServletRequest request, Object ➥
result){
    return "/welcome.jsp";
  }
}
```

The WelcomeCommand class implements the Command interface, which defines the execute method.

You implement the view (welcome.jsp) as a JSP page, as shown here:

```
<html> <head><title>Hello World</title></head><body>
<h1>This is a sample application.</h1>
Message is: <%=request.getAttribute("message")%>
</body></html>
```

Develop the Standard Deployment Descriptor

A J2EE web application is required to have a standard deployment descriptor that defines the web application details to the web container. This deployment descriptor is an XML file named web.xml. The deployment descriptor for our sample application is given here:

```
<!DOCTYPE web-app PUBLIC   "-//Sun Microsystems, Inc.//DTD Web ➥
Application 2.3//EN"  "http://java.sun.com/dtd/web-app_2_4.dtd">
  <web-app>
    <display-name>Hello World</display-name>
    <description>Hello World Web Application</description>
    <servlet>
      <servlet-name>frontControllerServlet</servlet-name>
      <servlet-class>FrontController</servlet-class>
    </servlet>

    <servlet-mapping>
      <servlet-name>frontControllerServlet</servlet-name>
      <url-pattern>*.htm</url-pattern>
    </servlet-mapping>

  </web-app>
```

This web deployment descriptor defines the servlets (the front controller servlet) and specifies a servlet mapping to invoke the front controller servlet for all requests having a URL that ends in an .htm extension.

Develop the Geronimo-Specific Deployment Descriptor

A J2EE web application can have an optional vendor-specific deployment descriptor to define vendor-specific application parameters. Geronimo uses Jetty as its web container and requires a deployment descriptor file geronimo-web.xml. This file is defined by the geronimo-web.xsd schema file in the schema subdirectory of the Geronimo installation. In our example, since we do not need any vendor-specific configurations, this file is trivial and simple.

```
<?xml version="1.0" encoding="UTF-8"?>
  <web-app xmlns="http://geronimo.apache.org/xml/ns/web" ➥
configId="welcome" parentId="geronimo/j2ee-server/car/1.0">
    <context-root>welcome</context-root>
    < context-priority-classloader>true</context-priority-classloader>
  </web-app>
```

The configId attribute is a unique name that identifies this module (also called a configuration in Geronimo). You can use this name to identify (or refer to) this module wherever required (especially when you use the deploy tool to manage this module). The parentId attribute refers to a parent configuration. Every configuration in Geronimo can have a parent configuration. If we had an EJB module that our web module depended on, we could have it as the parent configuration of the web configuration. This way, the web application could see all the classes from the EJB application. This is because their classloaders would follow the same structure as their respective configurations, and hence the EJB module's classloader would be the parent classloader for the web module's classloader. By default, the parent configuration for all J2EE components should be geronimo/j2ee-server/car/1.0, which is the configId for the Geronimo J2EE server configuration.

The context-root element specifies the context root of the application. Every web application is deployed in the web container in a different context root, and the container makes the application available at the URL `http://[host-name][:port]/[context-Root]`. The context-priority-classloader element defines whether the web application classloader loads classes from the web application before loading classes from its parent classloader.

Compile and Package the Web Application

Next, you need to compile the servlets and the helper classes.

J2EE web applications have a standard structure. The top-level directory of the web application (web module) is called the document root. This is where the static HTML pages, resource files (like images and JavaScript pages), JSP pages, and so on are stored. The document root directory contains a WEB-INF subdirectory, which contains the following files and subdirectories:

- *web.xml*: The web application standard deployment descriptor
- *Geronimo-specific deployment descriptor*: In our case, geronimo-web.xml
- *Compiled classes*: A subdirectory (WEB-INF/classes) where all the Java classes (including servlet and helper classes) are placed
- *lib*: A subdirectory (WEB-INF/lib) where the external JARs on which this application depends are placed

Figure 3-1 shows the web application directory structure.

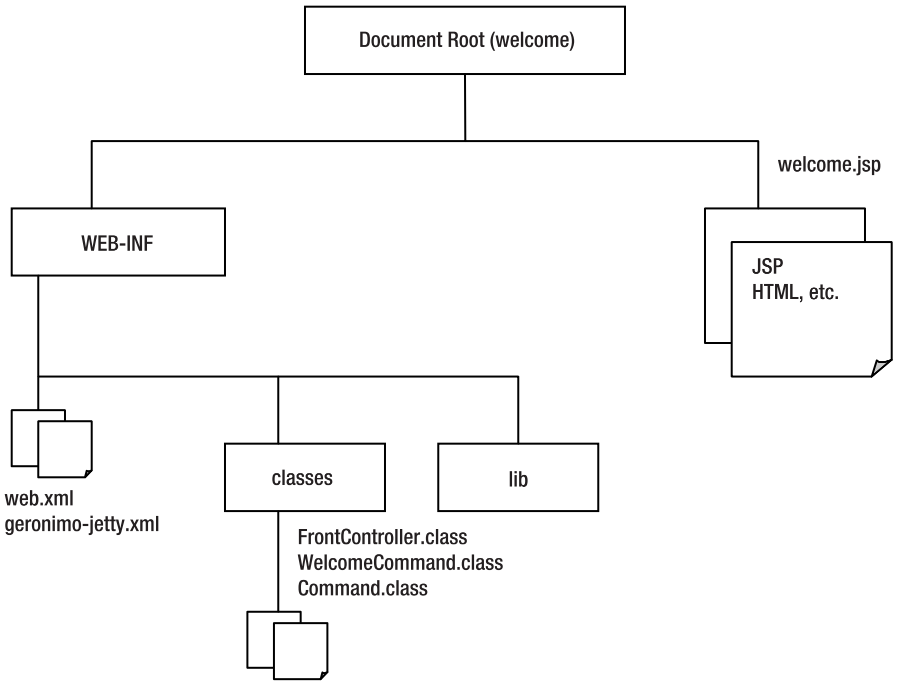

Figure 3-1. *Web application standard structure*

You can deploy web applications as an unpacked file structure (exploded directory format) or as a packaged JAR known as a Web Archive, or WAR, file. WAR files have a .war file extension.

From the document root directory (welcome) JAR, you create a WAR file by issuing the command shown here:

```
jar -cvf welcome.war
```

You can also use an IDE-like NetBean to create the WAR file and to deploy the application by clicking a single button.

Deploy the Application

You can deploy applications (and services) into Geronimo by using the Geronimo deployer JAR. To deploy a WAR file, use the following command:

```
java -jar bin/deployer.jar deploy welcome.war
```

This command requires the Geronimo server to be running. You can start the server by issuing the following command:

```
java -jar bin/server.jar
```

The application will be deployed with the context root as welcome (specified in the geronimo-web.xml file) and will then be started. You can also deploy applications into Geronimo by using any JSR 88–compliant IDE.

Access the Application

You can access the welcome application by using this URL:

```
http://localhost/welcome/welcome.htm?command=welcome
```

The context root, welcome, tells the container which web application should handle this request. This sample web application has a servlet mapping specified in its web.xml file that tells the container to execute the FrontController servlet for all URLs ending with the .htm extension. This servlet will be executed, and finally, the welcome.jsp displays a response page back to the user.

Configuring Database Connectivity

Geronimo comes with an embedded Derby database (`http://db.apache.org/derby/`) for its internal use. Derby is a pure Java Apache project. We will use this database as an example to explore database connectivity from a web application.

The following three steps are required to configure database connectivity from a web application:

1. Add a resource reference entry in the web deployment descriptor (web.xml).
2. Add a resource reference entry in the Geronimo web deployment descriptor (geronimo-web.xml).
3. Use JNDI to look up a data source and to create a JDBC connection.

Add a Resource Reference Entry in the Web Deployment Descriptor

Add the following resource reference entry in the web.xml web application deployment descriptor:

```
<web-app>
  <!-- other elements -->

  <resource-ref>
    <res-ref-name>jdbc/sampleResource </res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
  </resource-ref>

</web-app>
```

This declares a resource reference named jdbc/sampleResource and of the type javax.sql.DataSource. The web application needs to use this resource reference name when querying the JNDI server for the data source instance that can be used to create JDBC connection instances, as shown in the following code fragment:

```
Context ic =  new InitialContext();
Context envContext = (Context)ic.lookup("java:comp/env");

DataSource ds = envContext.lookup("jdbc/sampleResource");
Connection con = ds.getConnection();
```

The res-auth element specifies that the container, not the user application, will perform the database authentication. The res-sharing-scope element specifies that the connection instances this resource obtains are shareable among components of the same transaction.

Add a Resource Reference Entry in the Geronimo Web Deployment Descriptor

The resource reference entry in the web deployment descriptor declares a resource that can be referenced in a container implementation–independent manner. This allows the application code to be totally independent of the container. You need to specify all container-specific details through corresponding vendor-specific deployment descriptor entries, as shown here:

```
<web-app xmlns="http://geronimo.apache.org/xml/ns/web" ➥
 xmlns:naming="http://geronimo.apache.org/xml/ns/naming" ➥
configId="welcome" parentId="geronimo/j2ee-server/car/1.0">

  <naming:resource-ref>
    <naming:ref-name>jdbc/sampleResource</naming:ref-name>
    <naming:resource-link>SystemDatasource</naming:resource-link>
  </naming:resource-ref>

</web-app>
```

The resource-ref entry maps the resource reference name used in the application (and as specified in the corresponding resource-ref/res-ref-name element of web.xml) to the actual data source. The resource-link element should match the name of the Derby system database deployment plan.

To see whether the Derby system database configuration is available and started, list all the configIds on the current (running) server by using the following command:

```
java –jar bin/deployer.jar --user userName --password password ➥
list-modules --started
```

Use JNDI to Look Up the Data Source

The last step in configuring database connectivity is to use JNDI to look up the data source and to create JDBC connections to the database. From your web application, you can access the data source as shown here:

```
Context ic =  new InitialContext();
Context envContext = (Context)ic.lookup("java:comp/env");

DataSource ds = envContext.lookup("jdbc/ sampleResource");
Connection con = ds.getConnection();
```

Here, note that we use the resource reference name to look up the JNDI tree under the component environment context to obtain the reference to the data source.

Configuring a New Connection Pool

With Geronimo, you can configure a connection pool at the server level, the application level, or the module level. Resources you configure at the server level are visible and available to all applications and services. Resources you configure at the application level are available only to all the modules in a particular application. Module-level resources can be used only by a particular module.

To configure and use a new connection pool, you need to do the following:

1. Add the JDBC driver to the Geronimo repository.
2. Configure the connection pool.
3. Deploy the connection pool.

Add the JDBC Driver to the Repository

Common libraries that need to be shared across applications are generally stored in the Geronimo repository. Geronimo's architecture supports pluggable repository implementations, and the default repository implementation provided is a read-only local repository (SERVER_HOME/repository). Figure 3-2 shows the repository's directory structure.

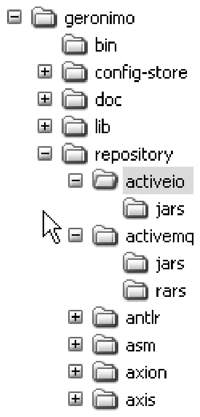

Figure 3-2. *Repository directory structure*

The libraries are stored under the repository folder in the repository/library-name/jars folder. So, for example, if you want to add the postgreSQL JDBC driver to the repository, you need to add it to the repository/postgreSQL/jars folder.

Configure the Database Pool

You configure database pools and other resources as J2EE connectors that provide the connection pool functionality for a particular database. The TranQL RAR (repository/tranql/rars/tranql-connector-1.0xxx.rar) provides the resource adapter module that implements the J2EE connector functionality. You can configure and deploy different instances of the RAR module to access different resources, like database pools and external systems.

To configure the connector RAR, we need to create a connector deployment plan. The code that follows shows a sample connector deployment plan:

```
<connector xmlns="http://geronimo.apache.org/xml/ns/j2ee/connector" ➥
 version="1.5"  configId="MyPostgreSQLDbPool" ➥
parentId="geronimo/j2ee-server/car/1.0">

  <dependency>
    <uri>postgresql/jars/ postgresql.jar </uri>
  </dependency>

  <resourceadapter>
    <outbound-resourceadapter>

      <connection-definition>
        <connectionfactory-interface>
          javax.sql.DataSource
        </connectionfactory-interface>

          <connectiondefinition-instance>
            <name>PostgreSQLDataSource</name>
            <config-property-setting name="UserName">
              Dbuser
            </config-property-setting>
            <config-property-setting name="Password">
              Dbpw
            </config-property-setting>
            <config-property-setting name="Driver">
              org.postgresql.Driver
            </config-property-setting>
            <config-property-setting name="ConnectionURL">
              jdbc:postgresql://localhost/mydb
            </config-property-setting>
            <config-property-setting name="CommitBeforeAutocommit">
              true
            </config-property-setting>
            <config-property-setting name="ExceptionSorterClass">
              org.tranql.connector.NoExceptionsAreFatalSorter
            </config-property-setting>
            <connectionmanager>
              <local-transaction/>
              <single-pool>
                <max-size>10</max-size>
                <min-size>0</min-size>
                <blocking-timeout-milliseconds>
                  5000
                </blocking-timeout-milliseconds>
                <idle-timeout-minutes>
                  30
                </idle-timeout-minutes>
                <match-one/>
              </single-pool>
            </connectionmanager>

            <global-jndi-name>
              jdbc/MyPostgresDatabase
            </global-jndi-name>
          </connectiondefinition-instance>

        </connection-definition>
      </outbound-resourceadapter>
    </resourceadapter>
</connector>
```

The key elements of this connector deployment plan are described in Table 3-1. (I will cover resource adapters in detail in a later chapter.)

Table 3-1. *Connector Deployment Plan Elements*

Element	Description
configId	This is the configuration identifier for the connection pool.
parentId	This is the parent configuration. The default value is `geronimo/j2ee-server/car/1.0`.
dependency	One or more dependency elements denote the external JARs that the resource adapter requires. These libraries should be available in the repository. In our case, the resource adapter is dependent only on the PostgreSQL JDBC driver JAR.
connectionfactory-interface	For JDBC connection pools, this needs to be javax.sql.DataSource. The connection factory interface should specify a fully qualified name of the connection factory interface supported by the resource adapter.
name	This is the name used for all references to this connection pool. This value is used in the resource-link element of the geronimo-web deployment descriptor to reference this connection pool from a web application.
config-property-setting	This specifies a value for one of the configuration settings for the connector.

Element	Description
max-size	This specifies the maximum number of simultaneous connection instances in the pool.
min-size	This indicates the minimum number of connection instances in the pool. If the pool size falls below this limit, the pool will be refilled to the size specified here. The default value is `0`.
blocking-timeout-milliseconds	The caller will wait this long for a connection to be available before throwing an exception back. If all the available connections (as set by max-size) are currently in use, the caller needs to wait until a connection is freed. This value sets a timeout for this waiting period.
idle-timeout-minutes	The connection pool periodically checks for idle connections and removes them. This entry specifies the interval between these checks.
global-jndi-name	This is the JNDI name a J2EE client application uses to look up this connection pool. It should not be used by any server components. This name should be unique for every resource deployed in Geronimo.

Deploy the Connection Pool

Once the deployment plan is available, you can deploy the connection pool with one of three visibility options: server, application, or module. To deploy a connection pool, you need both the deployment plan (connection pool configuration) and the resource adapter TranQL RAR.

Deploying a Server-Scoped Connection Pool

As mentioned, a server-scoped connection pool is available to all applications and services in Geronimo. To deploy a server-scoped connection pool, create a deployment plan file (let's call it postgreSQLPlan.xml) and use the following command with the server running:

```
java  -jar    bin/deployer.jar    deploy    postgreSQLPlan.xml ➥
repository/tranql/rars/ tranql-connector-1.0-xxx.rar
```

The general syntax is shown here:

```
java –jar bin/deployer.jar [command-name] [plan-name] ➥
[tranql-module-rar-file-name]
```

If the server is not running, you can use the `distribute` command instead of the `deploy` command. Then, you need to explicitly start the connection pool configuration after the server is started, as shown here:

```
java –jar bin/deployer.jar start [configId]
```

This configId value should match the configId value in the connection pool deployment plan.

Deploying an Application-Scoped Connection Pool

An application-scoped connection pool is visible only to the application that deployed it. To deploy an application-scoped connection pool, you need to follow these steps:

1. Specify the connector module in the application deployment descriptor.
2. Specify the connector deployment plan in the Geronimo-specific application deployment descriptor.
3. Package the application EAR.

Specify the Connector Module in the Application Deployment Descriptor

The application deployment descriptor (META-INF/application.xml) should define the TranQL connector module, as shown here:

```
<application xmlns="http://java.sun.com/xml/ns/j2ee" ➥
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ➥
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee ➥
http://java.sun.com/xml/ns/j2ee/application_1_4.xsd" version="1.4">
  <module>
    <ejb>ejbs.jar</ejb>
  </module>
  <module>
    <web>
      <web-uri>my-web-app.war</web-uri>
      <context-root>/my-web-app</context-root>
    </web>
  </module>
  <module>
    <connector> tranql-connector-1.0-xxx.rar </connector>
  </module>
</application>
```

You should package the connector RAR file along with the application EAR file.

Specify the Connector Deployment Plan in the Geronimo Application Deployment Descriptor

Specify the connector deployment plan file in the Geronimo application deployment descriptor (META-INF/geronimo-application.xml), as shown here:

```
<application ➥
xmlns="http://geronimo.apache.org/xml/ns/j2ee/application" ➥
configId="MyApplication">
  <module>
   <connector> tranql-connector-1.0-xxx.rar </connector>
   <alt-dd> postgreSQLPlan.xml</alt-dd>
  </module>
</application>
```

The alt-dd element specifies which deployment plan to use for deploying the connector module. This is a convenient way to override the Geronimo deployment plan. If the alt-dd element is not present, the file META-INF/geronimo-ra.xml, which is packaged within the RAR file, is used as the Geronimo deployment plan.

Package the Application EAR

In this case, you need to package the TranQL RAR and the connection pool deployment plan within an application EAR. Figure 3-3 depicts this arrangement.

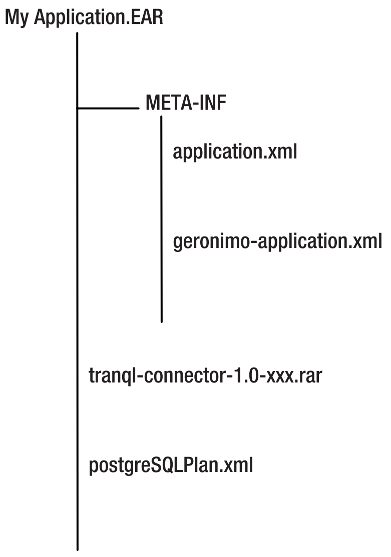

Figure 3-3. *Application EAR packaged with a connector RAR*

Deploying a Module-Scoped Connection Pool

Module-scoped connection pools can be used from within only the module that defined the connection pool. To deploy a module-scoped connection pool, specify the connection deployment plan inline with the Geronimo module-specific deployment descriptor. For a web application, you need to specify this in the geronimo-web.xml file, as shown here:

```
<web-app xmlns="http://geronimo.apache.org/xml/ns/web" ➥
xmlns:naming="http://geronimo.apache.org/xml/ns/naming" ...>
...
  <resource>

    <external-rar>
      tranql/rars/ tranql-connector-1.0-xxx.rar
    </external-rar>
    <connector ➥
xmlns="http://geronimo.apache.org/xml/ns/j2ee/connector" ➥
version="1.5"configId=" MyPostgreSQLDbPool ">
      <!-- Same as the connector deployment plan (postgreSQLPlan.xml) ➥
contents -->
    </connector>

  </resource>

</web-app>
```

The external-rar element should point to the common RAR file in the Geronimo repository.

Securing Your Web Application

Now that you've seen how to use Geronimo to develop and run a web application, we'll discuss providing security for your application. You configure web application security in the web deployment descriptor. J2EE containers are required to provide security services to J2EE components. Containers provide both declarative security and programmatic security. With declarative security, application components specify access control security needs in an external file such as the deployment descriptor. Using programmatic security, the components can programmatically use security features to implement custom security requirements.

A J2EE application can consist of both protected and unprotected resources. Application security generally involves two different processes: authentication and authorization. Authentication is the process of identifying the user (usually through a user login) to the system, and authorization is the process by which the system restricts access to protected resources to authorized users.

Let's examine the authentication process in more detail. Then, you'll learn how to configure web application security in Geronimo.

J2EE Authentication Methods

J2EE web containers are required to support the following authentication schemes:

- HTTP basic authentication
- Form-based authentication
- Digest authentication
- Client-side/server-side certification–based authentication

HTTP Basic Authentication

Figure 3-4 shows how the HTTP basic authentication mechanism works.

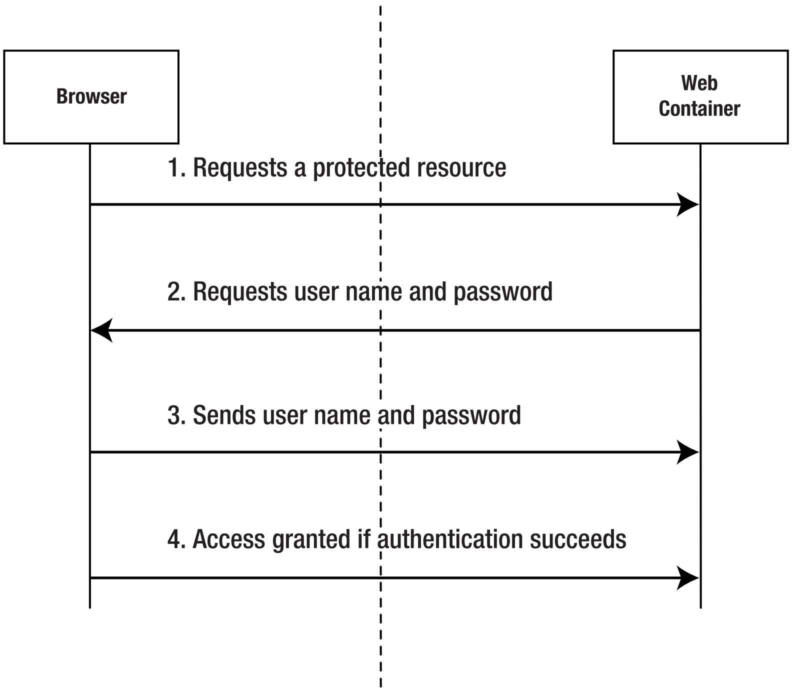
Browser
Web Container
1. Requests a protected resource
2. Requests user name and password
3. Sends user name and password
4. Access granted if authentication succeeds

Figure 3-4. *HTTP basic authentication*

When the client requests any of a web application's protected resources, the container intercepts the request and presents the client with a login dialog box. The container allows access only if the user name and password are valid. The main disadvantage of HTTP basic authentication is that it transmits user names and passwords as clear text data (base64-encoded form) through the network. However, you can use HTTPS transport to secure the data transported between the client and the server.

Form-Based Authentication

Form-based authentication allows you to use custom login pages and login error pages. With the basic authentication method, when the server challenges the browser for authentication information, it shows the user a login dialog box. With form-based authentication, the container displays a custom login page to the user. Figure 3-5 depicts the form-based authentication mechanism.

Here, also, the authentication information is sent as clear text data unless it is using the HTTPS protocol.

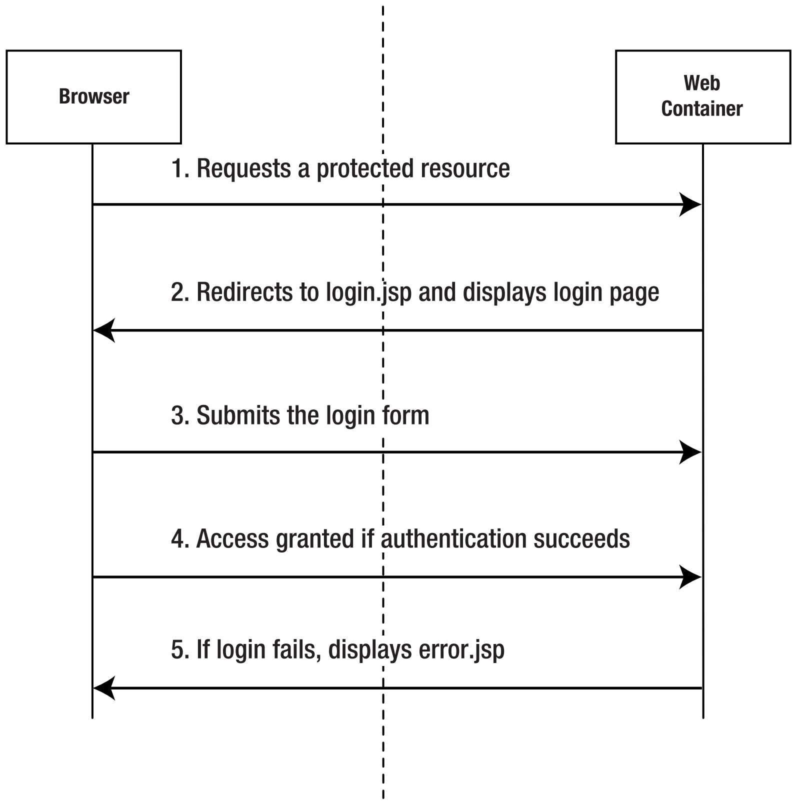
Browser
Web Container
1. Requests a protected resource
2. Redirects to login.jsp and displays login page
3. Submits the login form
4. Access granted if authentication succeeds
5. If login fails, displays error.jsp

Figure 3-5. *Form-based authentication*

Digest Authentication

This method is more secure than the basic- and form-based mechanisms. When the user tries to access a protected resource, the browser shows a login dialog box similar to the one shown for basic authentication. It sends the authentication details in an encrypted form that is more secure than the base64 encoding in the basic authentication scheme. Also, with this approach, the integrity of the URL data is certified. This method is not widely used.

Client-Side/Server-Side Certification–Based Authentication

This is the most secure authentication mechanism. Here, the client and the server use HTTPS protocol to transfer data to each other. When the client requests access to a protected resource, the server sends a digital certificate to the client. The client verifies the certificate, and if it is valid, the client uses the server-supplied public key to encrypt the data (login information) and sends it to the server. By verifying the server certificate, the client ensures it is securely sending data to the right server. You can increase security by having the client present a certificate to the server to verify the authenticity of the client.

■Note We will discuss authentication methods in more detail, including how to set up Geronimo for HTTPS connections, in Chapter 10.

Web Application Security Configuration

In this section, you will explore how to configure web application security with Geronimo by using an available security realm. In a later chapter, we will discuss how to create a custom security realm in Geronimo. The steps required to configure web application security in Geronimo are as follows:

1. Specify the protected resource list (web.xml).
2. Specify the login configuration (web.xml).
3. Specify security roles (web.xml).
4. Define security role mapping (geronimo-web.xml).

Specify the Protected Resource List

To specify the list of resources in the web application that need to be protected, you can use a security constraint element in the web deployment descriptor, as shown here:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>My Protected Resources</web-resource-name>
      <url-pattern>/admin/*</url-pattern>
    </web-resource-collection>

    <auth-constraint>
      <role-name>appUser</role-name>
    </auth-constraint>
</security-constraint>
```

This configuration specifies that only users belonging to the appUser role can access resources with the URL pattern `/admin/*`. With this configuration, the URL `http://hostname:port/webAppContext/admin/index.jsp` will be protected, but the resource `http://hostname:port/webAppContext/index.jsp` will not be.

Specify the Login Configuration

Next, you need to specify the authentication method and the security realm (in web.xml) that should be used to authenticate a user. The authentication method can be any one of the methods described earlier (indicate the mechanism as follows: `BASIC`, `FORM`, `DIGEST`, or `CLIENT_CERT`). A security realm is a collection of user and group definitions that are controlled by the same authentication policy.

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>geronimo-properties-realm</realm-name>
</login-config>
```

In the case of form-based authentication, you need to specify the login and login error pages.

```
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>geronimo-properties-realm</realm-name>
  <form-login-config>
    <form-login-page>/logon.jsp</form-login-page>
    <form-error-page>/logonError.jsp</form-error-page>
  </form-login-config>
</login-config>
```

The realm name specified here should be a valid realm on the Geronimo server.

Specify Security Roles

Specify the security role used in the security constraint configuration in the web deployment descriptor, as shown here:

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

Map Security Roles to Physical Roles in the Geronimo Deployment Descriptor

The final step is to map security roles to physical roles in a security realm defined on the Geronimo server. (We will discuss the details of defining and configuring a security realm in the chapter about J2EE security.) To complete this step, use the code shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://geronimo.apache.org/xml/ns/web" ➥
xmlns:naming="http://geronimo.apache.org/xml/ns/naming"  ➥
xmlns:sec="http://geronimo.apache.org/xml/ns/security"  ➥
configId="HelloWorldWebApp" parentId="geronimo/j2ee-security/car/1.0">

  <context-root>welcome</context-root>
  <context-priority-classloader>true</context-priority-classloader>

  <security-realm-name>geronimo-properties-realm</security-realm-name>
  <sec:security>
    <sec:default-principal realm-name="geronimo-properties-realm">
      <sec:principal ➥
class="org.apache.geronimo.security.realm.providers. ➥
GeronimoUserPrincipal"  name="metro"/>
    </sec:default-principal>
    <sec:role-mappings>
      <sec:role role-name="appUser">
        <sec:realm realm-name="geronimo-properties-realm">
          <sec:principal ➥
class="org.apache.geronimo.security.realm.providers. ➥
GeronimoGroupPrincipal" name="it" designated-run-as="true"/>
          <sec:principal ➥
class="org.apache.geronimo.security.realm.providers. ➥
GeronimoUserPrincipal" name="metro"/>
        </sec:realm>
      </sec:role>
    </sec:role-mappings>
  </sec:security>

</web-app>
```

The Geronimo web deployment descriptor defines the geronimo/j2ee-security/car/1.0 configuration as the parent configuration. This configuration defines the realm geronimo-properties-realm. I've included this realm in our example only for demonstration purposes—you need to create a new realm for use in your applications. The geronimo-properties-realm uses two property files (var/security/ users.properties and var/security/ groups.properties) for configuring users and user groups. The security element defines a default principal and a role-to-principal mapping for the logical role appUser defined in the web application. The role appUser is mapped to the group named "it" and the user named "metro" in the geronimo-properties-realm. To configure a new user to the realm, add the line `username=password` to the users.properties file, and to configure a new group, add the line `groupName=userName1, userName2` to the groups.properties file.

You can access the application by using the URL `http://localhost:8080/welcome`. When you are prompted to enter a user name and password, enter the user name and password as configured in your users.properties file.

Summary

In this chapter, we discussed how to use Geronimo to deploy and run web applications. We also looked at configuring, deploying, and using a database connection pool, and you saw how to configure a web application to utilize Geronimo's security features. In the next chapter, you will explore developing and using EJBs with Geronimo.