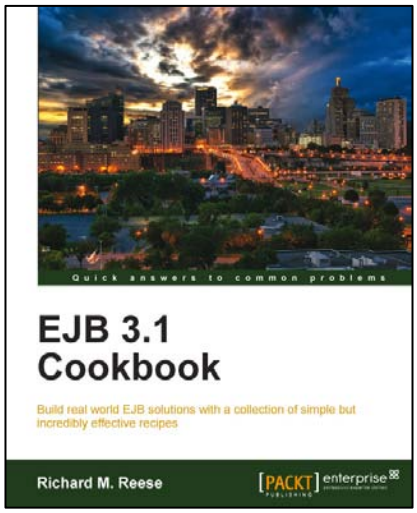
[PACKT] enterprise
professional expertise distilled
PUBLISHING

EJB 3.1 Cookbook

Richard M. Reese



Chapter No.7
"EJB Security"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.7 "EJB Security"

A synopsis of the book's content

Information on where to buy this book

About the Author

Richard Reese is an Associate Professor teaching Computer Science at Tarleton State University in Stephenville, Texas. Previously, he worked in the aerospace and telephony industries for over 16 years. He earned his Ph.D. in Computer Science from Texas A&M University. He also served four years in the Air Force primarily in the field of communication intelligence.

Outside of classroom, he enjoys tending his vegetable garden, maintaining his aquariums, and running with his dog, Zoey. He also enjoys relaxing with an episode of Firefly and is ever hopeful for the return of the epic series.

Dr. Reese has written numerous publications and contributed to *Turbo Pascal: Advanced Applications* .

No book can be written without the help from others. To this end I am thankful for my wife Karla and daughter Jennifer whose patience, support, and reviews have made this effort possible. In addition, I would like to thank the editorial staff of Packt and my reviewers for their input which has resulted in a much better book than it might otherwise have been.

Lastly, I am indebted to my doctorial committee chairman, Dr. Sallie Sheppard, who years ago spent countless hours helping me to learn how to write.

EJB 3.1 Cookbook

Enterprise Java Beans enable rapid and simplified development of secure and portable applications based on Java technology. Creating and using EJBs can be challenging and rewarding. Among the challenges are learning the EJB technology itself, learning how to use the development environment you have chosen for EJB development, and the testing of the EJBs.

EJB 3.1 Cookbook addresses all these challenges and covers the new 3.1 features, along with an explanation of useful features retained from previous versions. It brings the reader quickly up to speed on how to use EJB 3.1 techniques through the use of step-by-step examples without the need to use multiple incompatible resources. The coverage is concise and to the point, and is organized to allow you to quickly find and master those features of interest to you.

The book starts with coverage of EJB clients. The reader can choose the chapters and recipes which best address his or her specific needs. The newer EJB technologies presented include singleton beans which support application-wide needs and interceptors to permit processing before and after a target method is invoked. Asynchronous invocation of methods and enhancements to the timer service are also covered.

EJB 3.1 Cookbook is a very straightforward and rewarding source of techniques used to support Java EE applications.

What This Book Covers

Chapter 1, *Getting Started With EJBs* presents the creation of a few simple EJBs followed by recipes explaining how they can be invoked by a client. Client examples include the use of servlets, JSP, JSF, SE applications, and applets. The use of JNDI and dependency injection is also presented.

Chapter 2, *Session Beans* talks about the stateless, stateful, and the new singleton session bean. The use of single and multiple singletons is illustrated along with how concurrency can be managed. In addition, examples of how to use asynchronous methods are presented. *Chapter 3*, *Message-Driven Beans* explains how these EJBs provide a useful asynchronous approach for supporting an application. The numerous types of messages that can be sent are illustrated along with typical application scenarios. Access to the message queue is also discussed.

Chapter 4, *EJB Persistence* covers the creation and use of entities including the use of a facade class. In addition, numerous validation techniques are presented in support of entities.

Chapter 5, *Querying Entities using JPQL and the Criteria API* covers how to query an underlying data store with emphasis on the use of JPQL and the Criteria API. The use of annotations in support of these queries is illustrated.

Chapter 6, *Transaction Processing*, covers transaction processing which is central to many EJB supported applications. In this chapter, we examine how this support is provided using both container-managed transactions using annotations, and bean-managed transactions using code. Also, the use of timeouts and exception handling in support of transactions is illustrated.

Chapter 7, *EJB Security* covers the process of handling security using annotations and using code. The relationship between the support provided by the server and the roles used by an application is examined.

Chapter 8, *Interceptors*, explains how the interceptors provide a means of moving code that is not central to a business method outside of the method. Here, we learn how to use interceptors to handle a number of different concerns including security and transactions.

Chapter 9, *Timer Services*, explains how the timer services provide a means of periodically executing a method. We will examine the use of declarative and programmatic timers along with the use of persistent and non-persistent timers.

Chapter 10, *Web Services* explores how to create and use EJBs with JAX-RS and JAX-WS web services. Also covered is the use of a message-driven bean with a web service.

Chapter 11, *Packaging the EJB* details the packaging and deployment of EJBs. It covers the class loading process and the use of deployment descriptors for various interceptors such as timers and callbacks. The use of deployment descriptors with transactions and security is also addressed.

Chapter 12, *EJB Techniques*, examines techniques that are applicable to a variety of EJB technologies in this chapter. These include the use of logging and exception handling as they apply to EJBs. Also presented is how to create your own interceptor and effi cient techniques for using strings, time and currency.

7
EJB Security

In this chapter, we will cover:

- Creating the `SecurityApplication`
- Configuring the server to handle security
- Understanding and declaring roles
- Controlling security using declarations
- Propagating identity
- Controlling security programmatically

Introduction

Security is an important aspect of many applications. Central to EJB security is the control of access to classes and methods. There are two approaches to controlling access to EJBs. The first, and the simplest, is through the use of declarative annotations to specify the types of access permitted. The second approach is to use code to control access to the business methods of an EJB. This second approach should not be used unless the declarative approach does not meet the needs of the application. For example, access to a method may be denied during certain times of the day or during certain maintenance periods. Declarative security is not able to handle these types of situations.

In order to incorporate security into an application, it is necessary to understand the Java EE environment and its terminology. The administration of security for the underlying operating system is different from that provided by the EE server. The EE server is concerned with realms, users and groups. The application is largely concerned with roles. The roles need to be mapped to users and groups of a realm for the application to function properly.

A realm is a domain for a server that incorporates security policies. It possesses a set of users and groups which are considered valid users of an application. A user typically corresponds to an individual while a group is a collection of individuals. Group members frequently share a common set of responsibilities. A Java EE server may manage multiple realms.

An application is concerned with roles. Access to EJBs and their methods is determined by the role of a user. Roles are defined in such a manner as to provide a logical way of deciding which users/groups can access which methods. For example, a management type role may have the capability to approve a travel voucher whereas an employee role should not have that capability. By assigning certain users to a role and then specifying which roles can access which methods, we are able to control access to EJBs.

The use of groups makes the process of assigning roles easier. Instead of having to map each individual to a role, the user is assigned to a group and the group is mapped to a role. The business code does not have to check every individual. The Java EE server manages the assignment of users to groups. The application needs only be concerned with controlling a group's access.

A group is a server level concept. Roles are application level. One group can be associated with multiple applications. For example, a student group may use a student club and student registration application while a faculty group might also use the registration application but with more capability.

A role is simply a name for a set of capabilities. For example, an auditor role may be to review and certify a set of accounts. This role would require read access to many, if not all, of the accounts. However, modification privileges may be restricted. Each application has its own set of roles which have been defined to meet the security needs of the application.

The EE server manages realms consisting of users, groups, and resources. The server will authenticate users using Java's underlying security features. The user is then referred to as a principal and has a credential containing the user's security attributes. During the deployment of an application, users and groups are mapped to roles of the application using a deployment descriptor. The configuration of the deployment descriptor is normally the responsibility of the application deployer. During the execution of the application, the **Java Authentication and Authorization Service (JAAS)** API authenticates a user and creates a principal representing the user. The principal is then passed to an EJB.

Security in a Java EE environment can be viewed from different perspectives. When information is passed between clients and servers, transport level security comes into play. Security at this level can include **Secure HTTP (HTTPS)** and **Secure Sockets Layer (SSL)**. Messages can be sent across a network in the form of **Simple Object Access Protocol (SOAP)** messages. These messages can be encrypted. The EE container for EJBs provides application level security which is the focus of the chapter. Most servers provide unified security support between the web container and the EJB container. For example, calls from a servlet in a web container to an EJB are handled automatically resulting in a flexible security mechanism.

Most of the recipes presented in this chapter are interrelated. If your intention is to try out the code examples, then make sure you cover the first two recipes as they provide the framework for the execution of the other recipes. In the first recipe, *Creating the SecurityApplication*, we create the foundation application for the remaining recipes. In the second recipe, *Configuring the server to handle security*, the basic steps needed to configure security for an application are presented.

The use of declarative security is covered in the *Controlling security using declarations* recipe while programmatic security is discussed in the *Controlling security programmatically* recipe. The *Understanding and declaring roles* recipe examines roles in more detail and the *Propagating identity* recipe talks about how the identity of a user is managed in an application.

Creating the SecurityApplication

In this chapter we will create a `SecurityApplication` built around a simple **Voucher** entity to persist travel information. This is a simplified version of an application that allows a user to submit a voucher and for a manager to approve or disapprove it. The voucher entity itself will hold only minimal information.

Getting ready

The illustration of security will be based on a series of classes:

- `Voucher` – An entity holding travel-related information
- `VoucherFacade` – A facade class for the entity
- `AbstractFacade` – The base class of the `VoucherFacade` class as described in *Chapter 4*, *Creating an entity facade* recipe
- `VoucherManager` – A class used to manage vouchers and where most of the security techniques will be demonstrated
- `SecurityServlet` – A servlet used to drive the demonstrations

All of these classes will be members of the `packt` package in the EJB module except for the servlet which will be placed in the `servlet` package of the WAR module.

How to do it...

Create a Java EE application called `SecurityApplication` with an EJB and a WAR module. Add a `packt` package to the EJB module and an entity called `Voucher` to the package.

Add five private instance variables to hold a minimal amount of travel information: `name`, `destination`, `amount`, `approved`, and an `id`. Also, add a default and a three argument constructor to the class to initialize the `name`, `destination`, and `amount` fields. The `approved` field is also set to `false`. The intent of this field is to indicate whether the voucher has been approved or not. Though not shown below, also add getter and setter methods for these fields. You may want to add other methods such as a `toString` method if desired.

```
@Entity
public class Voucher implements Serializable {
  private String name;
  private String destination;
  private BigDecimal amount;
  private boolean approved;
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;

  public Voucher() {

  }

  public Voucher(String name, String destination,
    BigDecimal amount) {
    this.name = name;
    this.destination = destination;
    this.amount = amount;
    this.approved = false;
  }
  ...

}
```

Next, add an `AbstractFacade` class described in *Chapter 4*, *EJB Persistence* and a `VoucherFacade` class derived from it. The `VoucherFacade` class is shown below. As with other facade classes found in previous chapters, the class provides a way of accessing an entity manager and the base class methods of the `AbstractFacade` class.

```
@Stateless
public class VoucherFacade extends AbstractFacade<Voucher> {
  @PersistenceContext(unitName = "SecurityApplication-ejbPU")
  private EntityManager em;

  protected EntityManager getEntityManager() {
    return em;
  }
```

```
  public VoucherFacade() {
    super(Voucher.class);
  }

}
```

Next, add a stateful EJB called `VoucherManager`. Inject an instance of the `VoucherFacade` class using the **@EJB** annotation. Also add an instance variable for a `Voucher`. We need a `createVoucher` method that accepts a name, destination, and amount arguments, and then creates and subsequently persists the `Voucher`. Also, add get methods to return the name, destination, and amount of the voucher.

```
@Stateful
public class VoucherManager {
  @EJB
  VoucherFacade voucherFacade;

  Voucher voucher;

  public void createVoucher(String name, String destination,
    BigDecimal amount) {
    voucher = new Voucher(name, destination, amount);
    voucherFacade.create(voucher);
  }

  public String getName() {
    return voucher.getName();
  }

  public String getDestination() {
    return voucher.getDestination();
  }

  public BigDecimal getAmount() {
    return voucher.getAmount();
  }
  ...
}
```

Next add three methods:

1. `submit` – This method is intended to be used by an employee to submit a voucher for approval by a manager. To help explain the example, display a message showing when the method has been submitted.
2. `approve` – This method is used by a manager to approve a voucher. It should set the `approved` field to `true` and return `true`.

3. `reject` – This method is used by a manager to reject a voucher. It should set the `approved` field to `false` and return `false`.

```
@Stateful
public class VoucherManager {
  ...
  public void submit() {
    System.out.println("Voucher submitted");
  }

  public boolean approve() {
    voucher.setApproved(true);
    return true;
  }

  public boolean reject() {
    voucher.setApproved(false);
    return false;
  }
}
```

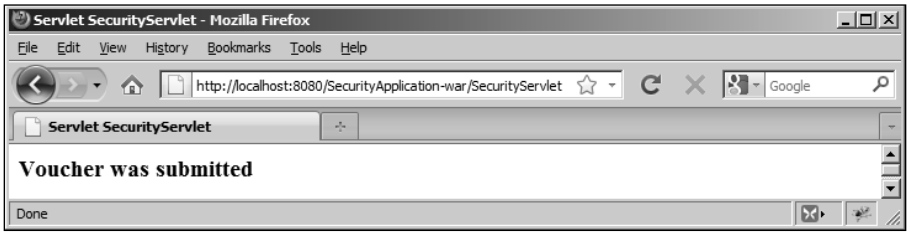
To complete the application framework, add a package called `servlet` to the WAR module and a servlet called `SecurityServlet` to the package. Use the **@EJB** annotation to inject a `VoucherManager` instance field into the servlet.

In the try block of the `processRequest` method, add code to create a new voucher and then use the `submit` method to submit it. Next, display a message indicating the submission of the voucher.

```
public class SecurityServlet extends HttpServlet {

  @EJB
  VoucherManager voucherManager;

  protected void processRequest(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
      response.setContentType("text/html;charset=UTF-8");
      PrintWriter out = response.getWriter();
      try {
        voucherManager.createVoucher("Susan Billings",
          "SanFrancisco", BigDecimal.valueOf(2150.75));
        voucherManager.submit();
        out.println("<html>");
        out.println("<head>");
```

```
            out.println("<title>Servlet SecurityServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h3>Voucher was submitted</h3>");
            out.println("</body>");
            out.println("</html>");

        } finally {
            out.close();
          }
    }
    ...
}
```

Execute the `SecurityServlet`. Its output should appear as shown in the following screenshot:



How it works...

In the `Voucher` entity, notice the use of `BigDecimal` for the `amount` field. This `java.math` package class is a better choice for currency data than `float` or `double`. Its use avoids problems which can occur with rounding as discussed in *Chapter 12*, *How to support currency* recipe. The **@GeneratedValue** annotation, used with the `id` field, is discussed in *Chapter 4*, *Creating an entity facade* recipe.

In the `VoucherManager` class, notice the injection of the stateless `VoucherFacade` session EJB into a stateful `VoucherManager` EJB. Each invocation of a `VoucherFacade` method may result in the method being executed against a different instance of `VoucherManager`. This is the correct use of a stateless session EJB. The injection of a stateful EJB into a stateless EJB is not recommended.

See also

The next recipe, *Configuring the server to handle security*, enables the server to support security for this application.

Configuring the server to handle security

Enabling a server to handle security involves configuring the actual server and configuring the deployment file. In order for the server to handle the application, the application needs to specify certain application security attributes in a deployment descriptor file. This recipe addresses these issues.

Getting ready

Before a Java EE application can use security, the EE server must be configured to handle security. The configuration process involves several steps:

1. Enabling the security manager
2. Selecting a realm
3. Adding users and groups to the realm
4. Optional: Enabling the default principal to role mapping

The actual steps are server-specific. On the application side, this process involves modifying a deployment file. For the `SecurityApplication`, we will modify the `web.xml` file. This process involves:

1. Setting the realm
2. Setting the login configuration
3. Adding the security role
4. Setting the security constraint
5. Enabling the security constraint

How to do it...

The first step is to enable the security manager. Typically, a server administration console is provided as part of the Java EE server and allows the administrator to configure the server. The actual use of the console is server-specific. Here, we will illustrate the process using GlassFish.

For this recipe, you will need to access the administrator console and enable the security manager. This is often simply a checkbox selection. In addition, either use an existing realm or create one if you are comfortable dealing with realms.

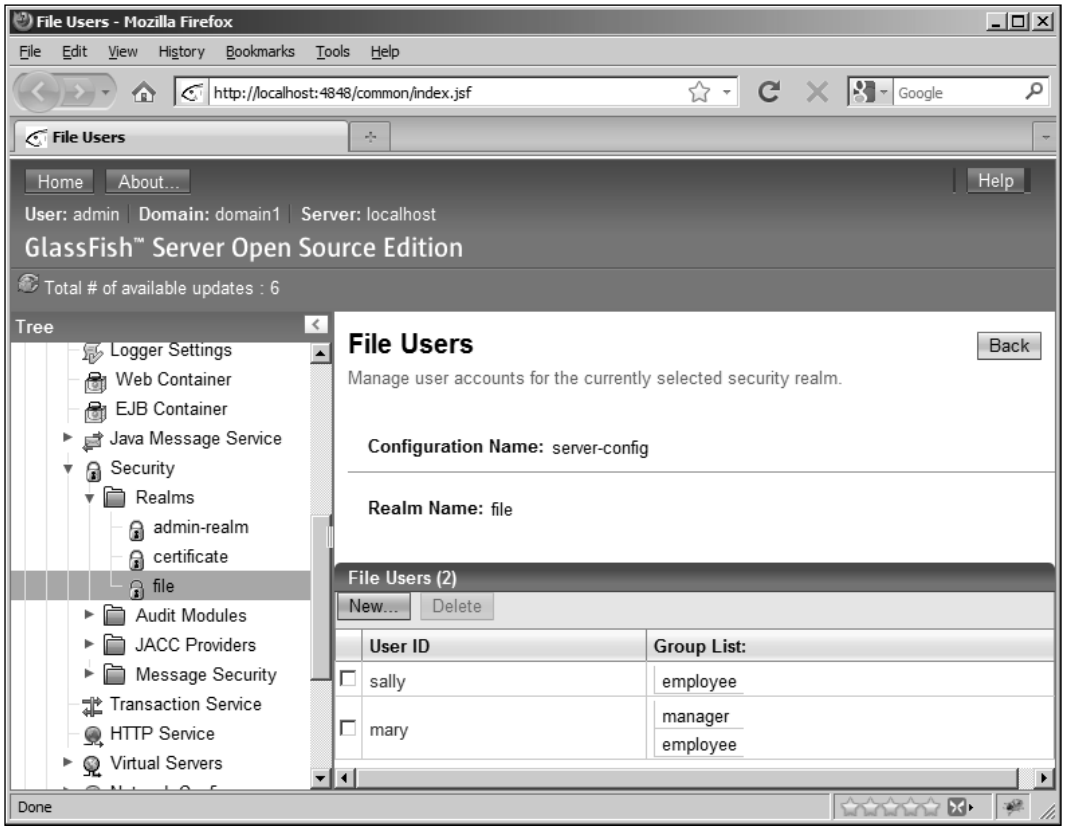
The actual realm to use is dependent on the Java EE server you are using. In Glassfish, the file realm is normally available and easy to use. This example will use the file realm but you can choose a different realm for your implementation.

Enabling the security manager and selecting the file realm in Glassfish is shown in the following screenshot:

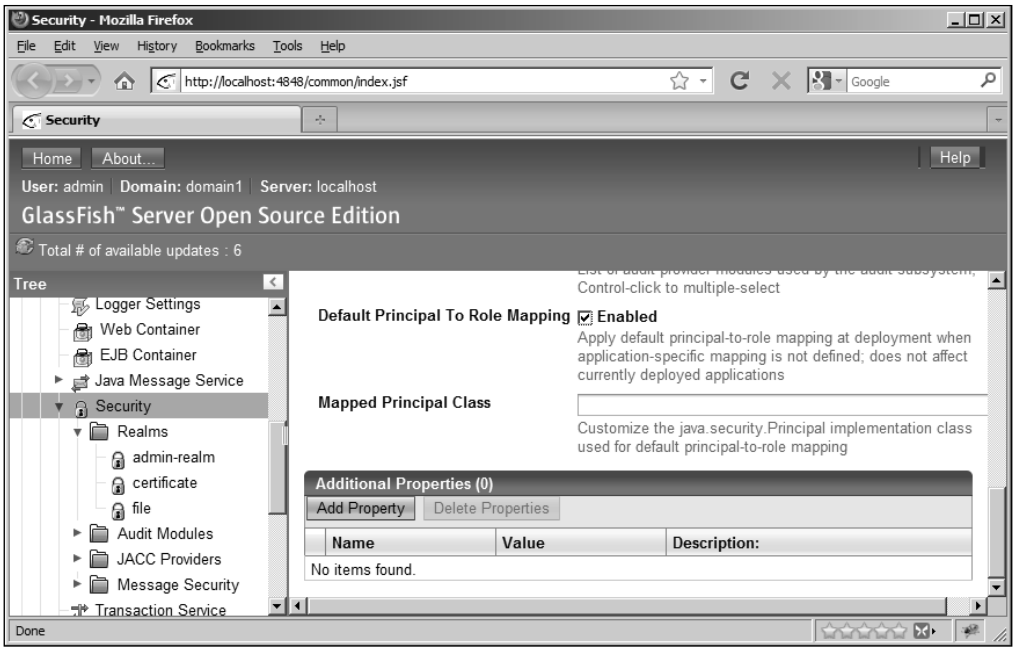
Next, add the following users and associate them with the groups: employee and manager. For each user assign a simple password to use when logging on.

- sally – employee
- mary – manager and employee

From GlassFish, under the Security element select **Realms** and then the **file** realm. Select the **Manage Users** button found at the top of the window. From this window add these two users as shown in the following screenshot:



Some servers have the capability to automatically associate a group name with a role if they are identical. This will avoid having to perform explicit mapping. Should your server support this option, then go ahead and exercise it. In addition, the server may require you to restart the server before some of these configuration actions take effect. The following screenshot shows the mapping enabled in GlassFish.



The next step involves modifying a deployment file. For the `SecurityApplication`, we will modify the `web.xml` file. Your development environment may provide a wizard or similar set of dialog boxes to configure the deployment file. Sometimes it is necessary to manually create and modify the `web.xml`. It all depends on the development environment.

The basic settings we will use include:

- Setting the login configuration to basic authentication
- Setting the realm to file
- Setting the initial security role
- Setting the security constraint to the URL pattern
- Enabling the security constraint
- Optionally mapping groups to roles

When the user makes a request which requires authentication, there are several possible authentication techniques available on most servers including:

- None – No authentication of users will be performed
- Digest – A cryptographic hash of the user ID and password are used
- Client certificate – Authentication is based on the client's public key certificate
- Basic – The server authenticates using a user ID and password
- Form – The developer provides a customized login screen

For simplicity's sake we will use basic authentication. With this technique, the user is prompted for a user ID and a password that the server uses to authenticate the user. This approach may not be the best for many applications as it is not as secure as other approaches. It sends the user information as Base64 encoded which allows the user ID and password to be easily decoded. However, if used in conjunction with a secure transport mechanism such as SSL, the approach becomes more secure.

Modify the `<login-config>` element of the `web.xml` file to specify the basic authentication method and the file realm as shown below. This can be done manually or using a development environment tool. In NetBeans, the edit window for the file supports this task.

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

The roles used in the `SecurityApplication` include employee and manager. Initially, the employee role should be added.

```
<security-role>
  <description/>
  <role-name>employee</role-name>
</security-role>
```

A security constraint also needs to be added to the application. This constraint will specify that when the `SecurityServlet` is accessed, the server needs to prompt the user for a user ID and password. This is accomplished using the `<security-constraint>` XML element.

Nested within this element is a `<web-resource-collection>` element. It includes a `<web-resource-name>` element to hold the name of the resource and the `<url-pattern>` element to specify the resource to authenticate. The name of the resource can be any reasonable descriptive name you choose. The URL pattern should match those pages the security constraint should be applied to. The following code specifies a constraint for our servlet:

```
<security-constraint>
...
  <web-resource-collection>
    <web-resource-name>SecurityApplicationResouces
      </web-resource-name>
    <description/>
    <url-pattern>/SecurityServlet</url-pattern>
  </web-resource-collection>
...
</security-constraint>
```

To enable the security constraint, the `<role-name>` element is added to the `<auth-constraint>` element. Use a role name of employee.

```
<security-constraint>
...
  <auth-constraint>
    <description/>
    <role-name>employee</role-name>
  </auth-constraint>
</security-constraint>
```

Roles need to be mapped to users and groups. As an application developer, it is not necessary to know the names of the users or groups of a realm. In fact, this information may not be known at the time the application is developed. The application may be deployed, and redeployed, to any number of different servers and the group names used by the servers may vary. Mapping the roles to the realm is the responsibility of the deployer. In some circumstances, the deployer and the developer may be one and the same. This is often true during the testing process and within smaller organizations.

If the server has been configured to automatically map groups to roles, it is not necessary to perform this mapping explicitly in a deployment file. If explicit mapping is required, the mapping of roles to groups is performed using the `<security-role-mapping>` element of the runtime deployment descriptor. This descriptor may be in one of several files depending on how the application is deployed (`sun-application.xml`, `sun-web.xml`, or `sun-ejb-jar.xml`). For example, roles can be mapped to either a single user or to a group. Here, the employee role is mapped to a single user: "sally" and the manager group is mapped to the group: "manager".

```
<security-role-mapping>
  <role-name>employee</role-name>
  <principal-name>sally</principal-name>
</security-role-mapping>

<security-role-mapping>
  <role-name>manager</role-name>
  <group-name>manager</group-name>
</security-role-mapping>
```

The use of explicit mapping is not used here as the server was configured to automatically map groups to roles where the name of the role and the group is identical. This allows us to conveniently skip this step. In Glassfish, the `sun-web.xml` contains the `<security-role-mapping elements>`.

This configures the application. Next, execute the application. The results should be the same as in the previous application since no additional security restrictions have been incorporated.

How it works...

We had to configure the server to handle the security requirements of our application. We also needed to modify the `web.xml` file to work with the server. These steps involved enabling security and coordinating the realm, roles, and groups. It was also necessary to specify which web pages were subjected to security constraints using the `<security-constraint>` element.

See also

The next recipe, *Understanding and declaring roles*, adds roles to the application and demonstrates how access to methods is controlled. Subsequent recipes use this framework to demonstrate the implementation of security for EJBs.

Understanding and declaring roles

Roles are defined within an application in one of two ways: using the **@DeclareRoles** annotation and the **@RolesAllowed** annotation. In this recipe we will detail the **@DeclareRoles** annotation while the **@RolesAllowed** annotation will be introduced but developed further in the *Controlling security using declarations* recipe.

Getting ready

The two basic steps used to configure roles involve:

1. Using the **@DeclareRoles** annotation to specify the roles used by the class
2. Adding the **@RolesAllowed** annotation to restrict access to methods

The **@DeclareRoles** annotation, as its name implies, declares the roles used by the application and is applied at the class level. That is, these are the roles to be used with the annotated EJB. The annotation can only be used once per class.

The **@RolesAllowed** annotation is used to specify which methods are accessible by the roles declared within the annotation. If the roles listed in this annotation are not found in the **@DeclareRoles** annotation, the roles are automatically declared.

How to do it...

The **@DeclareRoles** annotation simply declares roles to be used by the EJB. This annotation can only be used at the class level. It does not specify any permission granted for the class or methods within the class.

The annotation takes either a single string argument or an array of string arguments. The following first example declares a single role while the second specifies two roles for a class.

```
@DeclareRoles ("employee")
@DeclareRoles ({"employee", "manager"})
```

Add the second **@DeclareRoles** annotation example to the `VoucherManager` class.

```
@Stateful
@DeclareRoles ({"employee", "manager"})
public class VoucherManager {
  ...
}
```

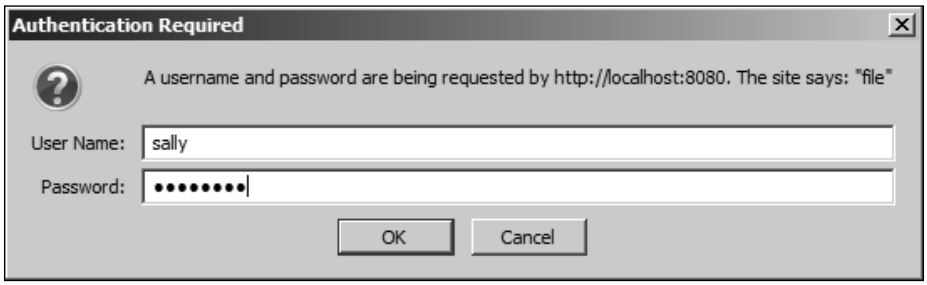
Execute the application. You should not see any difference in its behavior.

Next, add a **@RolesAllowed("employee")** annotation to the `VoucherManager`'s `submit` method.

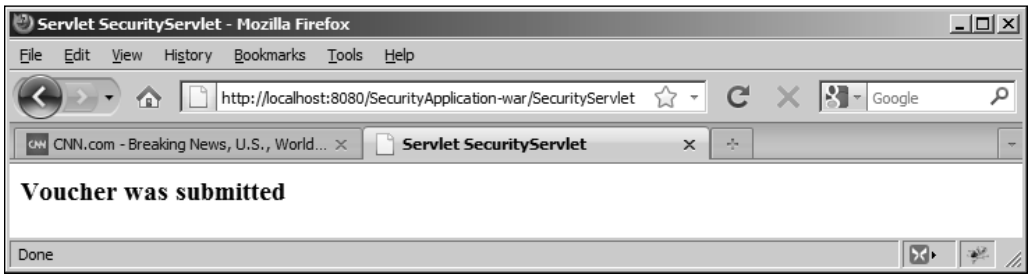
```
@Stateful
@DeclareRoles ({"employee", "manager"})
public class VoucherManager {
  ...
  @RolesAllowed("employee")
  public void submit() {
    System.out.println("Voucher submitted");
  }
  ...
}
```

This annotation will be elaborated upon in the next recipe. However, the use of the annotation restricts use of the method to only those users who are members of the employee role.

Execute the application. Since we are using basic authentication (see *Configuring the server to handle security* recipe) you should be prompted to log on. Use "**sally**" as the username and "**password**" as the password. The following screenshot illustrates the dialog box used by GlassFish.



The output of the application should appear as shown in the following screenshot:



How it works...

The **@RolesAllowed** annotation restricted access to the `submit` method to only those users belonging to the employee role. An exception was thrown otherwise. Since we were using basic authentication, the server-provided login dialog box was used.

Note, if you resubmit the request, the user is not prompted for a name or password. If you want to force the use of the login again, the `HttpSession`'s `invalidate` method or `HttpServletRequest`'s `logout` method can be used to close a session. However, this does not always work in a testing environment and frequently it is necessary to close the browser and open it up again to force the authentication of a user.

See also

The next recipe, *Controlling security using Declarations*, goes into more depth regarding the use of the **@RolesAllowed** annotation and other annotations used to control access to classes and methods.

Controlling security using declarations

Declarative security allows users, defined by roles, to access methods of a class. This is accomplished using a series of annotations to permit either certain roles to use a method, to permit all roles to use a method, or to deny access for all roles.

Getting ready

The application developer needs to determine which users (roles) should be permitted to access which methods. Once this has been determined, the classes and methods are annotated to affect these decisions.

Declarative security can be achieved using any of several annotations including **@RolesAllowed**, **@PermitAll**, and **@DenyAll** annotations. Each of these annotations has restrictions on where they can be used.

Annotation	Use With	Description
@PermitAll	Bean, Method	Allows access by all users
@DenyAll	Method	No roles are permitted access to the method
@RolesAllowed	Bean, Method	A list of roles permitted access

The steps used to control access include:

1. Specifying the role permitted using one of the above annotations
2. Applying the annotation at the desired level

How to do it...

The **@RolesAllowed** is configured with either a single string or an array of strings. These strings are the names of the roles allowed access to the EJB or a method. When applied to a method, the assignment will override any class level assignment. Here the use of both an array and a single role are specified.

```
@RolesAllowed({"bankemployee", "bankcustomer"})
@RolesAllowed("bankemployee")
```

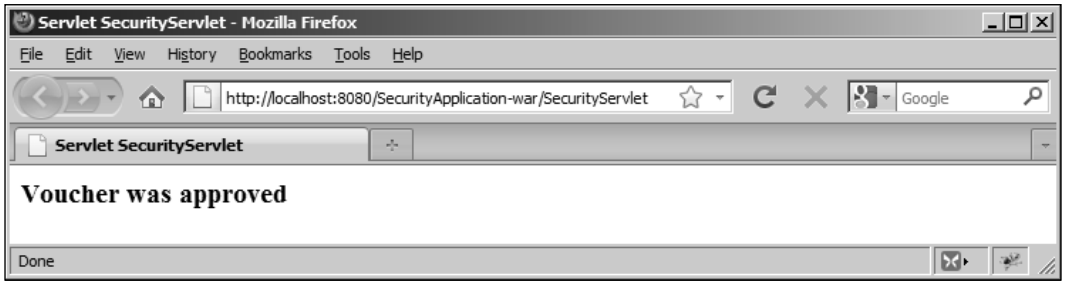
In the `VoucherManager` EJB, add an **@RolesAllowed** annotation to both the `approve` and `reject` methods. Specify a role of manager.

```
@RolesAllowed("manager")
public boolean approve() {
  voucher.setApproved(true);
  return true;
}

@RolesAllowed("manager")
public boolean reject() {
  voucher.setApproved(false);
  return false;
}
```

Next, modify the `SecurityServlet`'s `processRequest` method's try block to create a voucher and approve it using the `approve` method. Also, display a message to indicate whether the voucher was approved or not and add a catch block at the end of the try block to handle any access exceptions. The `javax.ejb.EJBAccessException` is thrown when an access restriction is violated.

```
voucherManager.createVoucher("Susan Billings", "SanFrancisco",
  BigDecimal.valueOf(2150.75));
boolean voucherApproved = voucherManager.approve();
...
if(voucherApproved) {
  out.println("<h3>Voucher was approved</h3>");
} else {
    out.println("<h3>Voucher was not approved</h3>");
  }
  ...
catch(EJBAccessException e) {
  System.out.println("Access exception");
}
```

Execute the servlet and enter "mary" as the user. The application should execute normally with the voucher being approved since "mary" is authorized to use the `approve` method as shown in the following screenshot:



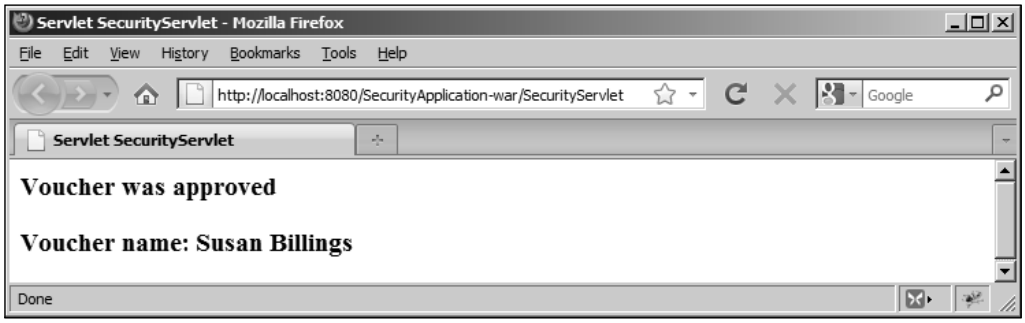
Close the browser and re-execute the servlet. This time, use "sally" as the user. Since "sally" is not authorized to use the approve method, an `EJBAccessException` is thrown.

INFO: Access exception

The **@PermitAll** annotation permits access to the EJB or a specific method by all roles. Anyone is able to access and use such methods. This is the default annotation for methods. For example, the `VoucherManager`'s `getName` method is not annotated. Add the following statement to the `SecurityServlet` immediately after the code which displays the voucher approval message.

```
out.println("<h3>Voucher name: " + voucherManager.getName() +
  "</h3>");
```

Execute the servlet using "mary". The voucher's username is displayed as illustrated in the following screenshot:



If we add the **@PermitAll** annotation to the `getName` method we will see no change in behavior. So what is the use of this annotation?

At the class level, one or more roles may have been declared using the **@DeclareRoles** annotation. The **@PermitAll** annotation can be used to permit a method to be used by some role other than those declared at the class level. For example, use the following annotations for the `VoucherManager` EJB:

```
@Stateful
@DeclareRoles("manager")
@RolesAllowed("manager")
public class VoucherManager {
  ...
}
```

In the `SecurityServlet`, remove the code dealing with the `submit` method.

```
try {
  out.println("<html>");
  out.println("<head>");
  out.println("<title>Servlet SecurityServlet</title>");
  out.println("</head>");
  out.println("<body>");

  voucherManager.createVoucher("Susan Billings", "SanFrancisco",
  BigDecimal.valueOf(2150.75));

  out.println("<h3>Voucher name: " + voucherManager.getName() +
    "</h3>");
```

```
        out.println("</body>");
        out.println("</html>");

      }
      catch(EJBAccessException e) {
        System.out.println("Access exception");
      }
      finally {
        out.close();
      }
```

Next, use the **@RolesAllowed("manager")** annotation only for the `VoucherManager`'s `approve` and `reject` methods. Do not use annotations for any of the other methods. If the servlet is executed using a user in the manager role ("mary"), the application executes cleanly. However, if you log in as "sally", an exception occurs because "sally" is no longer an authorized user. This is to be expected.

Add the **@PermitAll** annotation to the `createVoucher` and `getName` methods. Re-execute the application using "sally". This time the application should execute correctly. While we have only specified the manager role for the class, the **@PermitAll** annotation allows other groups to use those methods.

The **@DenyAll** annotation denies access to all roles and users. However, this annotation has limited utility. If access to the method should be denied to all users, then it can easily be removed from the EJB.

How it works...

The **@PermitAll**, **@DenyAll**, and **@RolesAllowed** annotations provided an easy-to-use mechanism for controlling access to methods based upon the user's role. When an unauthorized user attempted to access a method, an exception was thrown which was caught and dealt with.

However, sometimes it is necessary to allow an unauthorized user access to a method under certain conditions. The need for this access and how to achieve it is detailed in the next recipe.

Propagating identity

In certain situations, the identity of the user may need to be changed to enable a different, possibly more powerful, role. Consider the following analogy:

If you are familiar with the way most operating systems work, a user is not permitted to directly read or write to a file. Low-level access to the file is restricted. When a user needs to read or write to a file, the operating system will verify the individual's access rights to the file and then temporarily grant read/write access privileges to the user. The user assumes a higher level of privilege on a temporary basis.

This is analogous to the use of the **@RunAs** annotation. It allows a new role to be temporarily assigned to the methods of an EJB.

Getting ready

The steps used to propagate an identity include:

1. Executing a method of an EJB using one role which invokes a method of a second EJB
2. Executing the method of the second EJB using a different, more restrictive role as specified by the **@RunAs** annotation

When a method of the second EJB is invoked from the first EJB, the identity (principal) is passed with the invocation as part of the security context. The **@RunAs** annotation is used to temporarily assign a new role to the current principal. When the second class' methods are invoked, the new role is assumed.

This annotation is applied at the class level. The annotation has a single argument, a string containing the name of the role. Only one role can be assumed at a time.

```
@RunAs("manager")
```

When the **@RunAs** annotation is used, it is normally used in conjunction with the **@DeclareRoles** annotation.

How to do it...

We will add another class to the `packt` package to demonstrate the use of the **@RunAs** annotation. Let's assume when a voucher is submitted it should be verified. If it fails verification, then an exception can be thrown. In this example, we will not throw an exception so as to keep it simple. An additional restriction, which we will use to illustrate this technique, requires the methods of this verification EJB to run in the manager role.

Add a class to the `packt` package called `VoucherVerification`. Annotate the class with the **@RunAs** annotation using a value of "manager". In the class, inject a `SessionContext` object using the **@Resource** annotation. Add a `submit` method which passes and returns `void`. In the method, add code to use the `SessionContext` variable to return a principal object. This principal represents the user who invoked the method. The `getCallerPrincipal` method returns a `Principal` object. Follow the call with a `println` method invoking the `getName` method of the principal to display the principal's name.

```
@Stateless
@DeclareRoles("manager")
@RunAs("manager")
public class VoucherVerification {
  @Resource
  private SessionContext sessionContext;

  public void submit() {
    Principal principal = sessionContext.getCallerPrincipal();
    System.out.println("Principal: " + principal.getName());
    // Perform verification checks

  }
}
```

Modify the `VoucherManager` class to inject an instance of the `VerificationManager` class. Modify its `submit` method to call the `VerificationVoucher`'s `submit` method.

```
public class VoucherManager {
  ...
  @EJB
  VoucherVerification voucherVerification;
  ...
  @RolesAllowed("employee")
  public void submit() {
    System.out.println("Voucher submitted");
    voucherVerification.submit();
  }
  ...
}
```

Modify the `SecurityServlet` try block to appear as follows:

```
  out.println("<html>");
  out.println("<head>");
  out.println("<title>Servlet SecurityServlet</title>");
  out.println("</head>");
  out.println("<body>");
```

```
voucherManager.createVoucher("Susan Billings", "SanFrancisco",
  BigDecimal.valueOf(2150.75));
voucherManager.submit();

out.println("<h3>Voucher name: " + voucherManager.getName() +
  "</h3>");

out.println("</body>");
out.println("</html>");
```

Execute the application using the user, "mary", first and then "sally". The output from the `println` method reflects the different users.

INFO: Voucher submitted

INFO: Principal: mary

INFO: Voucher submitted

INFO: Principal: sally

Even though the user, "sally", is not a manager, the `submit` method is still executed.

How it works...

The `VoucherManager`'s `submit` method executed using the role of employee. However, we decided the `VoucherValidation`'s `submit` method needed to run in the manager role. Allowing an employee to temporarily use the manager role was achieved using the **@RunAs** annotation.

The use of the **@RunAs** annotation should be used with care. Sufficient checks should be made to ensure the selected method is executed only when any restrictions placed on its use have been met. In a sense, we are augmenting declarative security with a set of criteria to meet special conditions.

Controlling security programmatically

Programmatic security is based upon the **Java Authentication and Authorization Service (JAAS)** API. It should be used when declarative annotation is not adequate to affect the level of security desired. This can occur when access is time-based. For example, a user may only be allowed to access certain services during normal business hours such as when the stock market is open.

Getting ready

Programmatic security is affected by adding code within methods to determine who the caller is and then allowing certain actions to be performed based on their capabilities. There are two `EJBContext` interface methods available to support this type of security: `getCallerPrincipal` and `isCallerInRole`. The `SessionContext` object implements the `EJBContext` interface. The `SessionContext`'s `getCallerPrincipal` method returns a `Principal` object which can be used to get the name or other attributes of the user. The `isCallerInRole` method takes a string representing a role and returns a Boolean value indicating whether the caller of the method is a member of the role or not.

The steps for controlling security programmatically involve:

1. Injecting a `SessionContext` instance
2. Using either of the above two methods to effect security

How to do it...

To demonstrate these two methods we will modify the `SecurityServlet` to use the `VoucherManager`'s `approve` method and then augment the `approve` method with code using these methods.

First modify the `SecurityServlet` try block to use the following code. We create a voucher as usual and then follow with a call to the `submit` and `approve` methods.

```
out.println("<html>");
out.println("<head>");
out.println("<title>Servlet SecurityServlet</title>");
out.println("</head>");
out.println("<body>");

voucherManager.createVoucher("Susan Billings", "SanFrancisco",
  BigDecimal.valueOf(2150.75));
voucherManager.submit();
boolean voucherApproved = voucherManager.approve();

if(voucherApproved) {
  out.println("<h3>Voucher was approved</h3>");
} else {
    out.println("<h3>Voucher was not approved</h3>");
  }

out.println("<h3>Voucher name: " + voucherManager.getName() +
  "</h3>");

out.println("</body>");
out.println("</html>");
```

Next, modify the `VoucherManager` EJB by injecting a `SessionContext` object using the **@Resource** annotation.

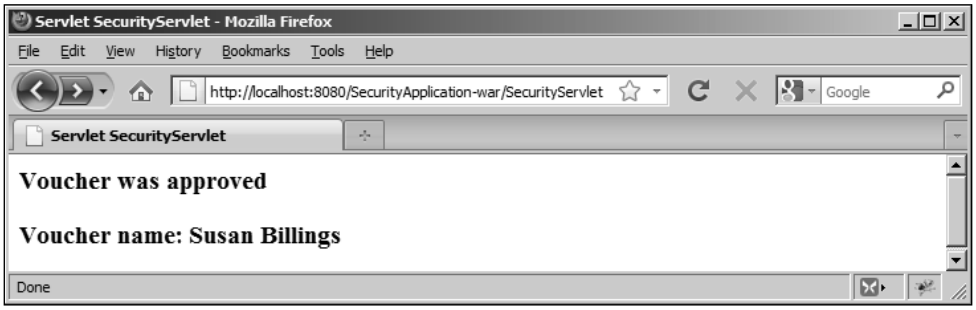
```
public class VoucherManager {
   ...
    @Resource
    private SessionContext sessionContext;
```

Let's look at the `getCallerPrincipal` method first. This method returns a `Principal` object (`java.security.Principal`) which has only one method of immediate interest: `getName`. This method returns the name of the principal.

Modify the `approve` method so it uses the `SessionContext` object to get the `Principal` and then determines if the name of the principal is "mary" or not. If it is, then approve the voucher.

```
public boolean approve() {
  Principal principal = sessionContext.getCallerPrincipal();
  System.out.println("Principal: " + principal.getName());
  if("mary".equals(principal.getName())) {
    voucher.setApproved(true);
    System.out.println("approve method returned true");
    return true;
  } else {
      System.out.println("approve method returned false");
      return false;
    }
}
```

Execute the `SecurityApplication` using "mary" as the user. The application should approve the voucher with the output as shown in the following screenshot:



Execute the application again with a user of "sally". This execution will result in an exception.

INFO: Access exception

The `getCallerPrincipal` method simply returns the principal. This frequently results in the need to explicitly include the name of a user in code. The hard coding of user names is not recommended. Checking against each individual user can be time consuming. It is more efficient to check to see if a user is in a role.

The `isCallerInRole` method allows us to determine whether the user is in a particular role or not. It returns a Boolean value indicating whether the user is in the role specified by the method's string argument. Rewrite the `approve` method to call the `isCallerInRole` method and pass the string "`manager`" to it. If the return value returns `true`, approve the voucher.

```
public boolean approve() {
  if(sessionContext.isCallerInRole("manager")) {
    voucher.setApproved(true);
    System.out.println("approve method returned true");
    return true;
  } else {
      System.out.println("approve method returned false");
      return false;
    }
}
```

Execute the application using both "mary" and "sally". The results of the application should be the same as the previous example where the `getCallerPrincipal` method was used.

How it works...

The `SessionContext` class was used to obtain either a `Principal` object or to determine whether a user was in a particular role or not. This required the injection of a `SessionContext` instance and adding code to determine if the user was permitted to perform certain actions.

This approach resulted in more code than the declarative approach. However, it provided more flexibility in controlling access to the application. These techniques provided the developer with choices as to how to best meet the needs of the application.

There's more...

It is possible to take different actions depending on the user's role using the `isCallerInRole` method. Let's assume we are using programmatic security with multiple roles.

```
@DeclareRoles ({"employee", "manager","auditor"})
```

We can use a `validateAllowance` method to accept a travel allowance amount and determine whether it is appropriate based on the role of the user.

```
public boolean validateAllowance(BigDecimal allowance) {
  if(sessionContext.isCallerInRole("manager")) {
    if(allowance.compareTo(BigDecimal.valueOf(2500)) <= 0) {
      return true;
    } else {
        return false;
    }
  } else  if(sessionContext.isCallerInRole("employee")) {
      if(allowance.compareTo(BigDecimal.valueOf(1500)) <= 0) {
        return true;
      } else {
          return false;
      }
    } else  if(sessionContext.isCallerInRole("auditor")) {
        if(allowance.compareTo(BigDecimal.valueOf(1000)) <= 0) {
          return true;
        } else {
            return false;
        }
      } else {
          return false;
      }
}
```

The `compareTo` method compares two `BigDecimal` values and returns one of three values:

- -1 – If the first number is less than the second number
- 0 – If the first and second numbers are equal
- 1 – If the first number is greater than the second number

The `valueOf` static method converts a number to a `BigDecimal` value. The value is then compared to `allowance`. This data type is discussed in more detail in *Chapter 12*, *How to support currency* recipe.

Where to buy this book

You can buy EJB 3.1 Cookbook from the Packt Publishing website:
`http://www.packtpub.com/ejb-3-1-cookbook/book`.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.PacktPub.com/ejb-3-1-cookbook/book