Global Information Assurance Certification Paper

SANS GIAC Practical for GSEC – version 1.4c

Security in Sun Java System Application Server Platform Edition 8.0

GIAC Security Essentials Certificate (GSEC)
Practical Assignment
Version 1.4c

Sid Ansari
Submitted February 3, 2005

Abstract:

In these competitive times businesses need to secure their services, reduce their costs, and lower the response times of their services to customers, employees, and suppliers. The services provided by a business need to be:

• Secure
• Available
• Reliable and Scalable

In most cases, enterprise services are implemented as multi-tiered applications. The middle tiers integrate existing front end applications with the business functions and the backend data of the service. The presentation logic (or the client) resides on the first tier, the business logic resides on the second-tier, and the backend (databases or other resources) reside on the third-tier. In this paper we look at the business logic when it consists primarily of Enterprise Java Beans as specified by the Java™ 2 Platform, Enterprise Edition (J2EE™). This paper examines security in a J2EE application server on the middle-tier as the enterprise responds to competitive pressures. Sensitive resources that can be accessed by many different users, or that often traverse unprotected open networks (such as the Internet) need to be protected. Although the quality assurances and implementation details can vary, they all share some characteristics. In this paper, we will examine some of these characteristics and see how they are implemented in a J2EE application that runs on a
J2EE server. Our conclusion will be that security is implemented in enterprise java applications using roles.

Introduction:

The architecture that is used to define Enterprise Java Beans (EJB) is a distributed, object-oriented, component, architecture.[1] In what follows, we will examine the various parts of this definition before turning to an examination of how Enterprise Java Beans can be secured. Let us start by looking at components. Enterprise Java Beans can be used to build components that can then be assembled into different applications. More specifically, an Enterprise Java Bean (along with other components) is a J2EE component (A Java 2 Platform Enterprise Edition component), and as such it can be defined as a self-contained functional software unit that is assembled into a J2EE application with its related classes and files and that communicates with other components. The J2EE specification defines the following J2EE components:

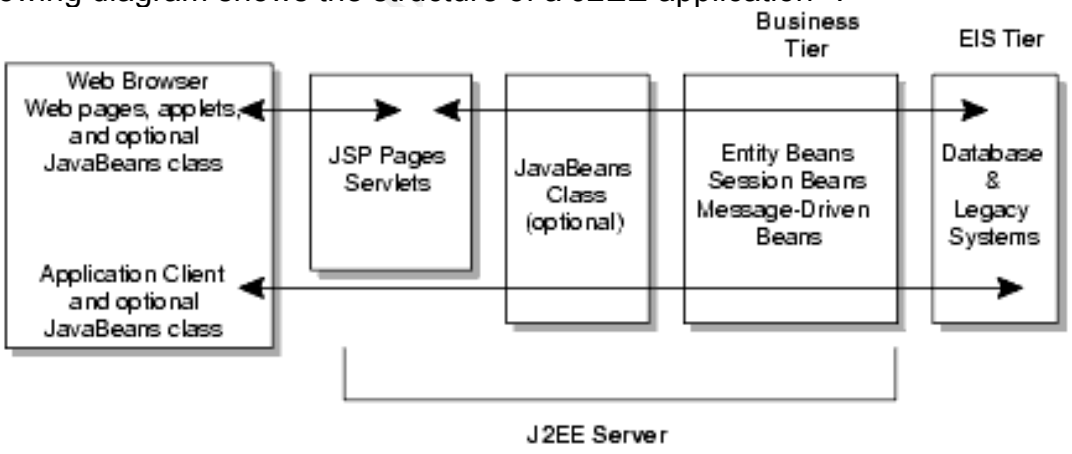
[1] The J2EE Spec. http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf

- Application clients and applets are defined as components that run on the client
- Servlets and JSP technology components are web components that run on the J2EE server
- Enterprise Java Beans are business components that run on the J2EE server

For example, we can build a shopping cart bean that can then be assembled into a customer application or an e-commerce application. This application can then be accessed by a remote client or a local client, and this remote or local client can, in turn, be a web client or an application client. A web client consists of two parts:

- Dynamic web pages containing various types of markup language (HTML, XML, and so on)
- Web browser which renders the pages received from the server

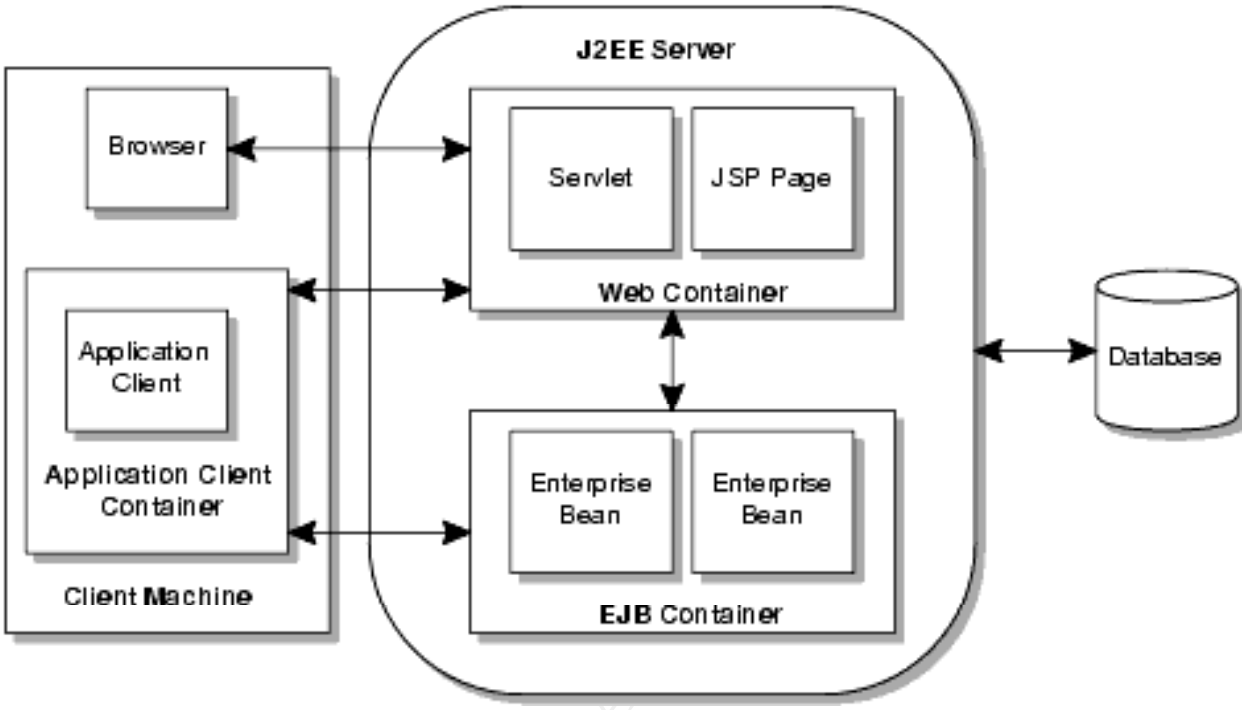
A web client is sometimes called a thin client. Thin clients usually do not do things like query databases, execute complex business rules, or connect to legacy applications. These operations are normally offloaded to enterprise beans executing on the J2EE server where they can leverage the security, reliability, speed, and service of the server side technologies. Thus, a three-tiered architecture extends the standard client-server architecture by placing an application server between the web-based client or the non-web-based client and the backend database. The following diagram shows the structure of a J2EE application[2].



Web components are the preferred API for creating web clients because they enable cleaner and more modular application design and they provide a way to separate programming from web page design. Personnel involved in web page design thus do not need to understand programming in order to do their jobs. A J2EE application client runs on a client machine and provides a way for users to handle tasks that require a richer user interface than can be provided by a markup language. It typically has a GUI created from Swing or Abstract Window Toolkit (AWT) API's,

[2] J2EE 1.4 Tutorial. http://java.sun.com/j2ee/1.4/docs/tutorial/doc/

but a command line interface is possible. Application clients directly access enterprise beans running in the business tier (or the middle tier). It is precisely because enterprise java beans are portable that they can also be deployed in different application servers without having to be rewritten. The application server vendors do not have their own proprietary API (Application Programming Interface). Instead you learn only one API, and the components that you develop will work with any application server. Thus, we can use IBM's proprietary application server (WebSphere), or we can use BEA systems application server (WebLogic), or we can work with Sun's application server. Of course, with each one of these servers we will still have to learn how to deploy the same bean (EJB) in the container of the different servers, but this task is considerably less difficult than learning a new API for each application server vendor. The J2EE (Java 2 Platform Enterprise Edition) compliant server, has to provide the bean container with its own services. Among these services is security and this service will be the focus of our paper. Much work has already been done on security concerning web components and web servers, and so we will confine ourselves to a discussion of security and its application to the EJB component. We will see how we can configure security in an application in such a way that authentication and authorization are performed by the container before access is allowed to the methods of an enterprise java bean. In particular, we will demonstrate how we can do this by modifying the XML code in the deployment descriptor, and we will also demonstrate how the deployment descriptor can itself be generated by using tools that are packaged with the application server. The J2EE server is divided into the web container running the web components (JSP pages, Servlets, and Java Beans), and the bean container running the enterprise beans. The containers and the various components that belong to the containers can be seen more clearly in the next slide[3].



[3] J2EE 1.4 Tutorial. http://java.sun.com/j2ee/1.4/docs/tutorial/doc/

A container is an interface between a component and the low-level platform-specific functionality that supports the component. In other words, a container is responsible for the interface between the bean and the J2EE server. Thus, for example, a container is responsible for creating a new instance of a bean, making sure that these instances are secure, and making sure that they are stored in the proper way. The assembly process of the bean involves specifying container settings for each component in the J2EE application and for the J2EE application itself. Since a container also manages the security concerns of the application, we have to specify the security setting for the container during the assembly and deployment of the bean. After an enterprise bean is created, another company can purchase the bean or component that has been developed and use it in one of their own applications. Java already allows us to develop object-oriented components in the form of classes that can be reused by others and, just like classes, components that are developed as Enterprise Java Beans can be reconfigured and reused without changing any of the code that has been written to develop them. The difference between J2EE components and standard java classes is that J2EE components are assembled into a J2EE application, verified to be well formed and in compliance with the J2EE specification, and deployed to production, where they are run and managed by the J2EE server. These terms require some explanation. The J2EE specification provides for the division of labor in the creation and deployment of enterprise beans. Generally speaking, there are approximately five tasks that can be performed by separate individuals or entities when creating a bean. There is, first of all, the J2EE product vendor who provides a product that implements the J2EE specification. Some well known product vendors are IBM, BEA Systems,

SUN, etc. This product is used by the software component developers to create the programs that are then run in the application servers. These programs include client components as well as web components and enterprise components. We also have the task performed by the application assembler who takes the components created by the application developers and combines them with deployment descriptors to make an application. The deployer puts the application into production. This means that the deployer is responsible for the configuration of the application server. It also means that the assembler and deployer are responsible, among other things, for the security of the application. It is, therefore, the tasks performed by these two that will be the focus of this paper. While it is the assembler who defines one or more roles in the deployment descriptor, it is the deployer who changes the behavior of the bean at runtime by modifying the XML written deployment descriptor. The deployer is also responsible for resolving references to resources in the application (such as a reference to databases), and for configuring the location of components in the distributed environment. Our focus, however, will be on the way security is configured by the assembler and the deployer in the application server. Finally, there is the system administrator who is responsible for the performance of the server and for the day to day operations of the server. It should be noted that the classifications listed above do not necessarily have to be followed by companies. Some companies may decide that the tasks to be performed by the application assembler, the deployer, and the system administrator should be performed by one person. This is especially true in the case of small to medium sized companies which may not have the resources to allocate different tasks to different people. The major advantage, then, in using enterprise java beans is that the developer of the bean can focus on the development process and leave the provision of the underlying services such as security, transaction services, networking, and resource management to the J2EE service vendors, the people who develop the J2EE servers in which the beans will be deployed. Someone else develops the underlying services, and the deployer and the assembler use them by modifying the deployment descriptor without altering the bean code. As mentioned above security for components is provided by their containers in order to achieve the goals for security specified in a J2EE environment. A container provides two kinds of security:

- Declarative security
- Programmatic security

Declarative Security:

Declarative security refers to the way in which an application's security structure (including security roles, access control, and authentication) is defined in a form external to the application. Declarative security is normally defined in an XML file called the deployment descriptor which represents a contract between an application component provider (the person who creates the components) and the application assembler (the person who assembles the application). As we have seen, the primary responsibility of the application component provider is to write the bean code which is then delivered to the application assembler who combines the new and existing beans into a larger application. A deployer maps the deployment descriptor's representation

of the application's security policy to a security structure specific to the particular environment. In other words, the deployer is responsible for resolving any external dependencies that might exist in the application. The deployer knows the security users, the groups, and the roles for this system, and knows what is configured into the server. Before deploying an application, the deployer maps the users and groups to the roles. At runtime, the container uses the security policy security structure derived from the deployment descriptor and configured by the deployer to enforce authorization.

Programmatic Security:

Programmatic security refers to the code that is incorporated in applications. Programmatic security can be used when it is deemed that declarative security is not good enough. The API for programmatic security required by the J2EE specification consists of two methods.

- isCallerInRole
- getCallerPrincipal

These methods allow components to make business logic decisions based on the security role of the caller or remote user. Thus, we can determine whether a client belongs to a given role by invoking the isCallerInRole method (the method isCallerInRole() returns a boolean):

```
boolean result = context.isCallerInRole("approle");
```

Once it is determined that the client belongs to a certain role (that boolean result is true) then the authorization process can kick into gear[4]. Similarly, we can identify the user by invoking the getCallerPrincipal method of the EJBContext interface. This method returns the java.security.Principal object (that identifies the user of the enterprise bean). In the following example, the getUserName() method of an enterprise bean returns the name of the J2EE user that invoked it.

```
public String getUserName() {
        return context.getCallerPrincipal().getName();
}
```

The following code segment illustrates how an application could make use of programmatic security:

```
public double getSalary(String userId) {
   java.security.Principal caller = context.getCallerPrincipal();
   String callerId = caller.getName();
        if ((context. isCallerInRole("approle")) && (callerId == userId)) {
                    getSalary(userId)
                } else {
                    throw new AccessException("Access Denied");
                }

        }
```

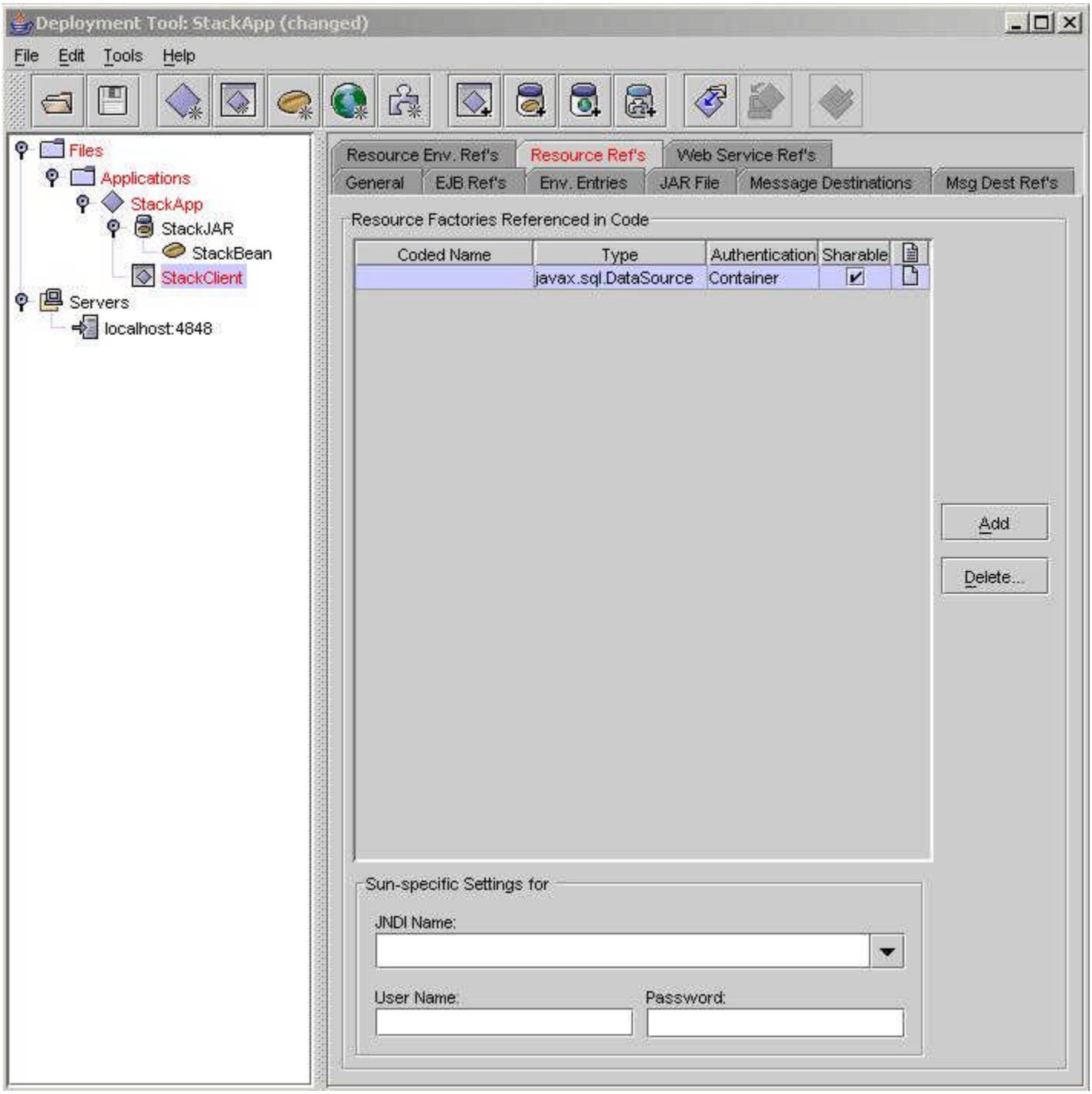
Normally, a client uses the Java Authentication and Authorization Service (JAAS) for authentication. JAAS allows applications to remain independent of the current

authentication technologies. As new authentication technologies emerge, they can be used by an application without making any changes to the code for the application. This is done by using what is called a login module. A typical login module could prompt for and verify a user name and password. Other login modules could read and verify a voice or fingerprint sample. This allows us to plug new authentication technologies under an application without making any modifications to the application itself. Typically, JAAS is used with the custom realm (see the following discussion). However, the discussion of JAAS is beyond the scope of this paper. Interested readers can refer to an excellent discussion of JAAS which can be found in the archives of Sun's worldwide developer conference held in 2003[5].

[4] Eric Jendrock. "Security". http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Security.html

[5] http://java.sun.com/security/javaone/2003/2236-JAASJGSS.pdf

Our focus will be solely on declarative security as it is applied to one of the components of the enterprise application, namely, the enterprise business component. With declarative security, when an application component requests a connection to an enterprise java bean, the container handles the sign-on required to make the connection. The following screen shot displays the selection of container-managed sign- on for the StackApp application.



Types of Enterprise Java Beans:

There are three types of enterprise java beans which can be used by an application as a business component:

1. Entity Beans
2. Message Beans

3. Session Beans

Session Beans, in turn, come in two flavours:

1. Stateless Session Beans
2. Stateful Session Beans

An entity bean is used to represent something in a database. An instance of an entity bean normally represents a row in the database. Since this type of bean represents a persistent store, it can outlive a client specific state or a server crash. A message bean, on the other hand, is used to listen for messages from a JMS messaging service. A client places a call to a message bean through a messaging service, and the messaging service passes the client request to the message bean. A session bean, unlike an entity bean, does not represent a row in the database. It represents a process such as a required calculation or credit card verification or a travel booking. A stateless session bean cannot, therefore, preserve a client conversational state whereas a stateful session bean can do just that. Stateless session beans forget about the specific client once a client call is completed, whereas stateful session beans preserve the client state even after a client call has completed. Calculation of income tax that an individual has to pay can be done using a stateless session bean, whereas a shopping cart application would require a stateful session bean. A crucial part of all enterprise applications is security, and the application server's goal is to provide security for enterprise applications that is in full compliance with the J2EE security model and the EJB security model [6]. Generally speaking, declarative security is handled by the container (web container or enterprise java beans container), and is defined within the application's deployment descriptor. This type of security normally restricts access to web components and enterprise (business) components by forcing the client to authenticate himself/herself to the container. Once the client has been authenticated successfully, the container checks to confirm whether or not the client is authorized to access the enterprise components in question. This authorization can function on a granular level in the sense that the client may only be authorized to access some of the methods belonging to the component in question. On the other hand, the client may have been granted access to the entire component. Either way it is the container which determines (based on access control) whether or not the current request has authorized universal access or restricted access. Furthermore, authentication is a prerequisite for authorization.

[6] "Securing J2EE Applications". http://docs.sun.com/source/817-6087/dgsecure.htm

The techniques for configuring security in sun's application server require that the following tasks have been performed [7]:

1. A realm has been defined. A realm can be thought of as a collection of users and groups that are authenticated in the same way.

2. A user has been defined and a password has been assigned to the user.
3. If there is a group to which the user belongs, then the user has been allocated to that group.
4. A role has been created.
5. The role has been assigned to the user and to the group to which the user belongs.
6. Permissions to access different methods in the enterprise bean have been set up. These method permissions tell us which roles are allowed to access the different methods in the remote interface, or the home interface.

We can compress these tasks into four more generic tasks that have to be performed in order to build and secure a bean of the enterprise type. First of all, we have to code the two interfaces, the bean class, and the client. Although the client could be a web client or a stand alone application client, we have chosen the latter to simplify our bean. To illustrate the use of security in Sun's Application Server we have chosen the example of a stack bean[8]. The code that follows represents the remote interface, the home interface, the bean class and the client for the stack bean. A stack is closely related to an array. There is only one point where the data can enter or leave a stack. The principle that is implemented by a stack is that the element that enters first is going to exit last (First-In-Last-Out), and no element can be removed from the middle of the stack. Our example is a stateless enterprise java bean, and in particular, it is a stateless enterprise java session bean. Next, we have to create users, groups, and passwords, and assign users to groups. Thirdly, an application has to be created. For now we can think of an application as a cabinet for holding the four programs that constitute our enterprise bean and the client. In this application we have to create an XML deployment descriptor that is going to give the application server information about the bean and how the bean should be secured and managed, and we also have to create a JAR file and place the two interfaces, the bean and the deployment descriptor in the JAR file (A JAR file is very much like a zip file). Finally, we have to secure and deploy the bean into the application server by using the tools that have been provided by the vendor, in this case Sun. The tasks involved in creating an application and a JAR file have been covered extensively in the literature, and it is not our aim to cover these topics in this paper. Interested readers can always refer to the J2EE tutorial on this subject (http://java.sun.com/j2ee/1.4/docs/tutorial-update2/doc/index.html).

[7] "Securing J2EE Applications". http://docs.sun.com/source/817-6087/dgsecure.htm

[8] See, for example, the excellent discussion of stacks by Mark Allen Weiss. Weiss, Mark Allen. Data Structures and Problem Solving using Java. Another Excellent source of information is Data Structures Using java by Malik and Nair. Thompson Course Technology (2003).

Step 1: Coding the Bean

Typically, when we write a client to access an enterprise java bean, the client runs in

one java virtual machine and the bean runs in a different java virtual machine. The technology that makes this possible is referred to as Remote Method Invocation (RMI). This technology provides the client with a proxy, referred to as a stub, that acts as a bridge between the client and the remote object. The client calls a method on the stub, and the stub takes care of the low-level communication between the client and the bean (or the remote object). On the bean side there is also a proxy (referred to as a skeleton), which receives the information from the stub and turns it into a local method call on the bean. The whole purpose of this architecture is to provide the client with network transparency. In other words, the client is made to think that remote calls on an object are functioning very much like local calls on an object that is running on the same java virtual machine as the client. Given this architecture, the stub (on the client side) must know which methods to call on the remote object (the stub is, after all, pretending to be the remote object for the client). The stub knows which methods to call because the stub has exactly the same methods as the bean. The client calls the methods on the stub through the component or remote interface, and the remote interface communicates with the skeleton on the machine running the remote object. Thus, we have to code the remote interface as well as coding the bean[9].

The Remote Interface:

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
import java.math.*;
public interface Stack extends EJBObject {
       public void Sample(int maxsize) throws RemoteException;
       public boolean hasNoElements() throws RemoteException;
       public void stuff(int i) throws RemoteException;
       public int drop()throws RemoteException;
}
```

The remote interface contains the methods that will be implemented in the bean class. This interface extends EJBObject and each and every method in the remote interface throws the RemoteException. Apart from the remote interface, an enterprise java bean also has a home interface. Think of the home interface as a tool that provides a way to the client to refer to the stub for the remote interface. Remember, that it is the ultimate objective of the client to get a handle on the remote interface, because once a client gets access to the remote interface, it is a short step to accessing the methods on the bean.

[9] An adequate discussion of the architecture of enterprise java beans requires an entire book. Many such books exist. An excellent book on the subject has been written by Richard Monson-Haefel. Monson-Haefel, Richard. Enterprise Java Beans.

The home interface provides each client with the tools to access the remote interface. Each bean that is created and used in our application will have its own home interface. In the case of session beans, the home interface normally has a create() method which is used to get a handle on the bean's component or remote interface. The return type

of the create() method is, therefore, the remote interface. In our example, the return type of the create() method is the object Stack (the remote interface). Each client gets a reference to his/her own bean (clients never share a bean), but each client will have access to exactly one home interface. Thus, if there are two clients, each client will have access to his/her own bean as well as his/her own remote interface, but they will both share one home interface. This one home interface will enable each client to refer to their very own remote interface.

The Home Interface:

```
import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
public interface StackHome extends EJBHome {
    Stack create() throws RemoteException, CreateException;
}
```

As mentioned previously, the home interface normally contains the create() method which throws CreateException and RemoteException. The home interface extends EJBHome.

The Bean:

```
import java.rmi.RemoteException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import java.math.*;
public class StackBean implements SessionBean {
      protected int stackArray[];
      protected int place_holder;
   public void Sample(int maxsize){
        stackArray = new int[maxsize];
        place_holder = -1;
    }
    public boolean hasNoElements(){
         return place_holder == -1;
      }
    public void stuff(int i){
        if(place_holder+1 < stackArray.length)
            stackArray[++place_holder] = i;
    }
    public int drop(){
        if(hasNoElements())
            return 0;
        return stackArray[place_holder--];
    }

   public StackBean() {}
   public void ejbCreate() {}
   public void ejbRemove() {}
   public void ejbActivate() {}
   public void ejbPassivate() {}
   public void setSessionContext(SessionContext sc) {}
```

```
} //StackBean
```

The bean class implements the methods that have been defined in the remote interface. The bean class also has an empty constructor as well as some “callback methods” that are used by the container to manage the bean. We also need a client file, and here we can choose either an application client or a web client since the implementation of security is very similar in either case. We will use an application client in this case. Here is the client. There are a couple of points that should be noted about the client. First of all, the client has to go through the home interface to get a reference to the remote interface. It does this in the following line:

```
Stack stack = home.create();
```

Secondly, once the client has access to the remote interface, it can call the business methods on the bean. Note, that the client cannot directly get access to the bean. In fact, only the container has direct access to the bean. For the client, the only access to the bean lies through the remote interface, and to get to the remote interface the client has to have access to the create() method of the home interface. Thus, the client is vulnerable on two counts. The client access to the bean can be blocked by blocking access to the create() method of the home interface, or the client access to the bean can be blocked by blocking access to the business methods in the remote interface.

The Client:

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import java.math.BigDecimal;
public class StackClient {
   public static void main(String[] args) {
       int i, j;
       try {
           Context initial = new InitialContext();
           Object objref = initial.lookup("java:comp/env/ejb/SimpleStack");
           StackHome home =
               (StackHome)PortableRemoteObject.narrow(objref,
                                          StackHome.class);
            Stack stack = home.create();
            stack.Sample(10);
            for(i=1;i<11;i++){
            j = (int)(Math.random() * 100);
            stack.stuff(j);
            System.out.println("stuff: " + j);
            }
            while(!stack.hasNoElements()){
              System.out.println("drop: " + stack.drop());
            }

       } catch (Exception ex) {
           System.err.println("Caught an unexpected exception!");
           ex.printStackTrace();
       }
   }
```

```
}
```

A cursory examination of the bean classes shows that we have not put any security related code in these classes. The person developing the bean (the bean developer) does not even have to think about security. Security, as we mentioned earlier, is handled in the deployment descriptor using XML tags. The application assembler defines one or more roles in the deployment descriptor and assigns methods belonging to the beans home and remote interfaces to the security roles. The deployer assigns the users and the groups created by the application server to the roles defined by the application assembler. The client application calls the increment and decrement methods in the bean, so we might wish to restrict access to these methods. We will assume that only payroll administrators can call these methods. The first step, then, in controlling access to the bean class is to create users and a group called “payroll” consisting of payroll administrators.

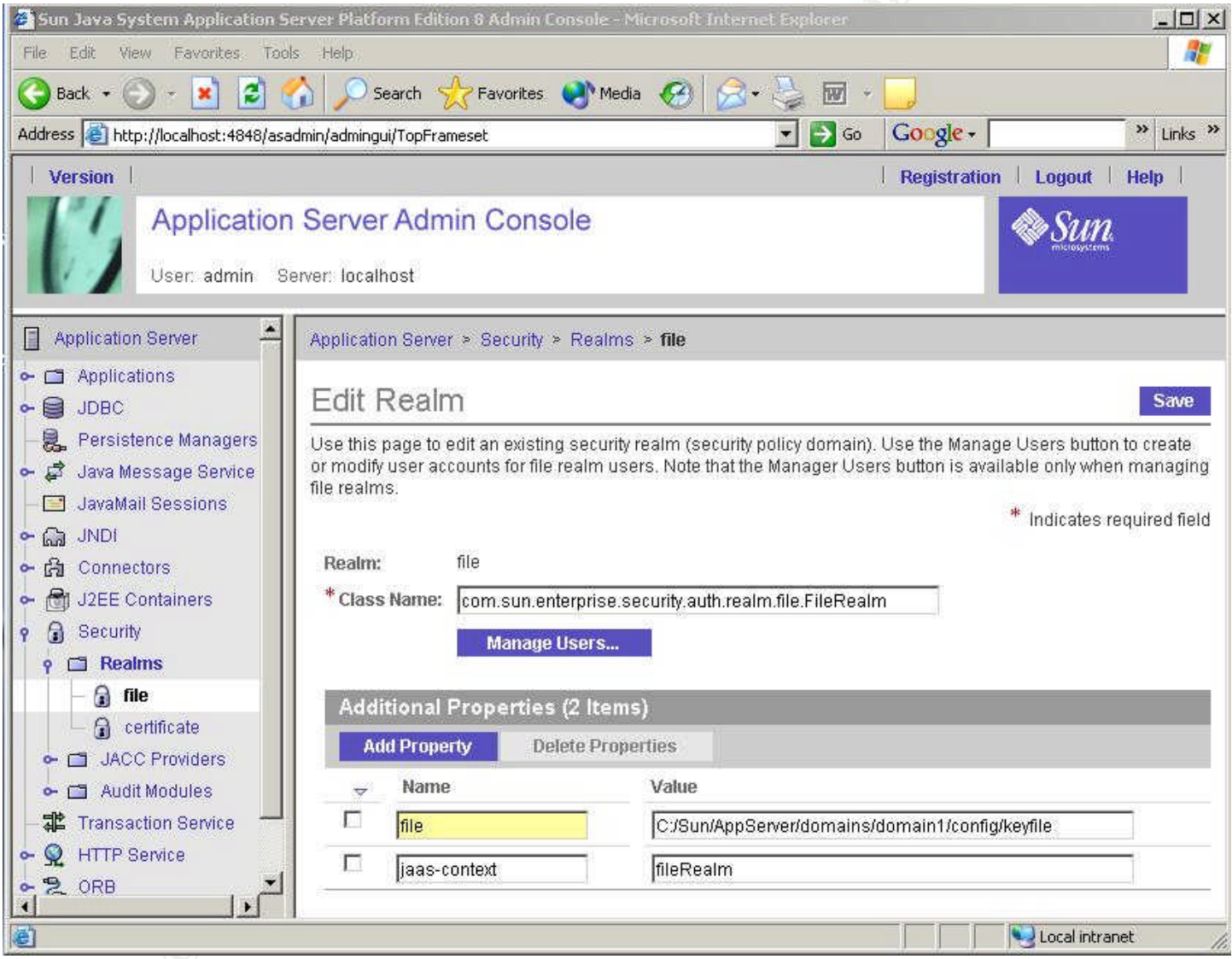
Step 2: Creating Users, Groups, and Passwords

We will create users (with passwords) and assign them to the payroll group. This is accomplished as follows:

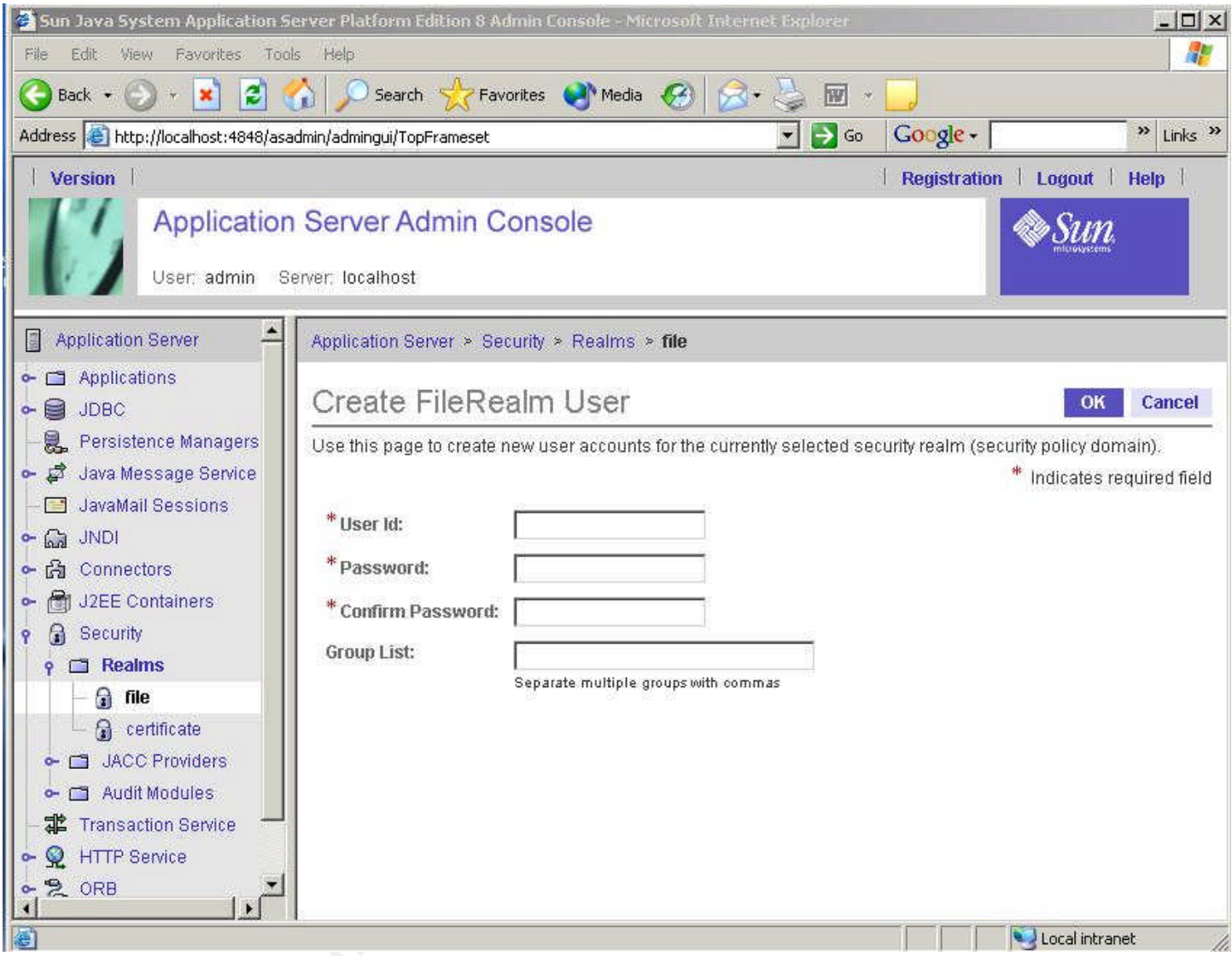
1. Start the server and connect to the Administrative Console by entering the following URL:

   http://localhost:8080/asadmin

2. Enter your username and password, and the screen below displays.
3. Navigate to Application Server => Security => Realms => file
4. Click on Manage Users…



5. Click on New to Create FileRealm User:



6. Enter User Id, Password, Confirm Password, and the Groups to which the user will belong. If the user will belong to more than one group, enter the groups separated by commas. Save your changes and exit from the Administrative Console. The user id and passwords will be stored in a file called keyfile under the config directory of the server instance. The entries will look similar to the following:

```
admin;{SSHA}uzW7cof2SLGbfHIjsz4pGxK60+FVLRjEO1Ovfg==;asadmin
sid;{SSHA}HOQ0ItFAmTzZMHReMHrqOidZxTpNERVeSNaTnA==;payroll,manager
scott;{SSHA}BzEZHtLmmB02eIUpkfeTaVEP4D+xOHnhg4qzfg==;payroll
```

The entries in the file consist of the username, the hashed password, and the name of the group to which the user belongs. Thus, both sid and scott belong to the group payroll, but only sid belongs to the group manager. The admin user belongs to the group asadmin. The asadmin group is created when the software is installed. By definition a file realm is a collection of users and groups that are controlled by the

same authentication policy. Instead of validating the credentials of the user directly, the application server uses a pluggable module. This pluggable module performs the check and returns the results to the server[10]. When using the file realm, the server authentication service verifies user identity by checking the file. This realm is used for the authentication of all clients except for Web browser clients that use HTTP BASIC authentication, FORM authentication, or Secure Socket Layer (SSL) authentication, called CLIENT-CERT in the Servlet Specification[11]. Apart from a file realm, the application server can also use a certificate realm where the user information is stored in a certificate database, an LDAP realm where the user information is stored in an LDAP database, a Solaris realm where the user information is kept in the Solaris operating system user database, and a custom realm where the user information is stored in a database of the applications choice. It is the use and implementation of a custom realm that involves the use of Java Authentication and Authorization Service (JAAS) as the underlying authentication mechanism. It is absolutely essential that when using the file realm, the file used to store user information (keyfile) be stored in a restricted and secure location. This file and the encrypted data that is stored in this file is the weak link in the security chain for this application server. Since we are using a file realm, access to this file is absolutely essential for someone seeking to gain unauthorized permission to execute methods on the beans. The passwords are stored in a hash format using the secure hashing algorithm. This algorithm was developed by the National Institute of Standards and Technology (NIST) in the United States, and it is considered to be more secure than Message Digest 5 (MD5), although MD5 is faster than the secure algorithm. SHA-1 produces a slightly longer digest value (160 bits) than MD5. The secure hash algorithm is considered to be secure because it is "computationally infeasible" to map clear text to a hashed message[12]. It is also difficult to find two messages that will give the same hash. Thus, it is used to generate the hash of a message called a "message digest". This message digest can be thought of as a fingerprint for the message. The message is passed through a digest algorithm in order to create a digest value. This value is unique as far as the message is concerned. The algorithms that generate these unique values are one way functions which simply means that it is not possible to retrieve the text from the hash. It is well known that hashed passwords, or message digests, are vulnerable to dictionary attacks. This attack works in the following way: all entries in a dictionary are hashed and these hashed values are then compared against the hash values that are stored in the file or the database. Once a match is found, the original dictionary value can then be used to log into the server. It may be extremely difficult to crack the passwords that have been hashed using SHA-1, but it is not impossible. It is for this reason that the passwords that are stored in the keyfile are created using "salted" hash. Dictionary attacks don't work very well against a salted hash because before a password is hashed a random value is attached to the password and then the hash is calculated. This random value is referred to as the "salt". Now the attacker has to attach this "salt" value to the dictionary

[10] http://java.sun.com/j2ee/1.4/docs/tutorial-update2/doc/index.htm

[11] http://developers.sun.com/prodtech/appserver/reference/techart/access_control.html

[12] http://www.itl.nist.gov/fipspubs/fip180-1.htm

words before hashing them and comparing them to the values that are stored in the file or the database. This process can take a very long time. To make matters even more difficult for the attacker, numerous functions are available for generating the random value[13] (the functions are available in Win32 API as well as in Java). It should be noted that the secure hash algorithm also pads the values in the message before hashing them, but the padding is not random. It is also important to point out that even salted passwords are open to brute-force attacks, but these can take a long time to succeed. Either way, the system administrator should immediately take action when he/she finds that the file in which the passwords are stored has been compromised.

Before compiling:

Before we can create the deployment descriptor for this bean we must compile the programs. A batch file can be created to set JAVA_HOME, PATH, and CLASSPATH variables, or these variables can be defined as Environment Variables in the Control Panel.

```
set JAVA_HOME=c:\jdk1.5.0
set PATH=.;%PATH%;%JAVA_HOME%\bin
set CLASSPATH=.;%JAVA_HOME%;%JAVA_HOME%\lib;%JAVA_HOME%\lib\tools.jar;
set CLASSPATH=.;c:\Sun\AppServer\libset \j2ee.jar
```

To compile:

```
c:\stack>javac *.java

c:\ stack>
```

A successful compilation will generate four classes, one each for the home interface, the remote interface, the bean, and the client. It is important to note that even though we have used java development kit 1.5 to compile the files, the application server itself uses the earlier versions of java. Installing the application server by referring to jdk1.5.0 will lead to deployment errors.

[13] http://www.aspheute.com/english/20010924.asp

Step 3: Creating the Application

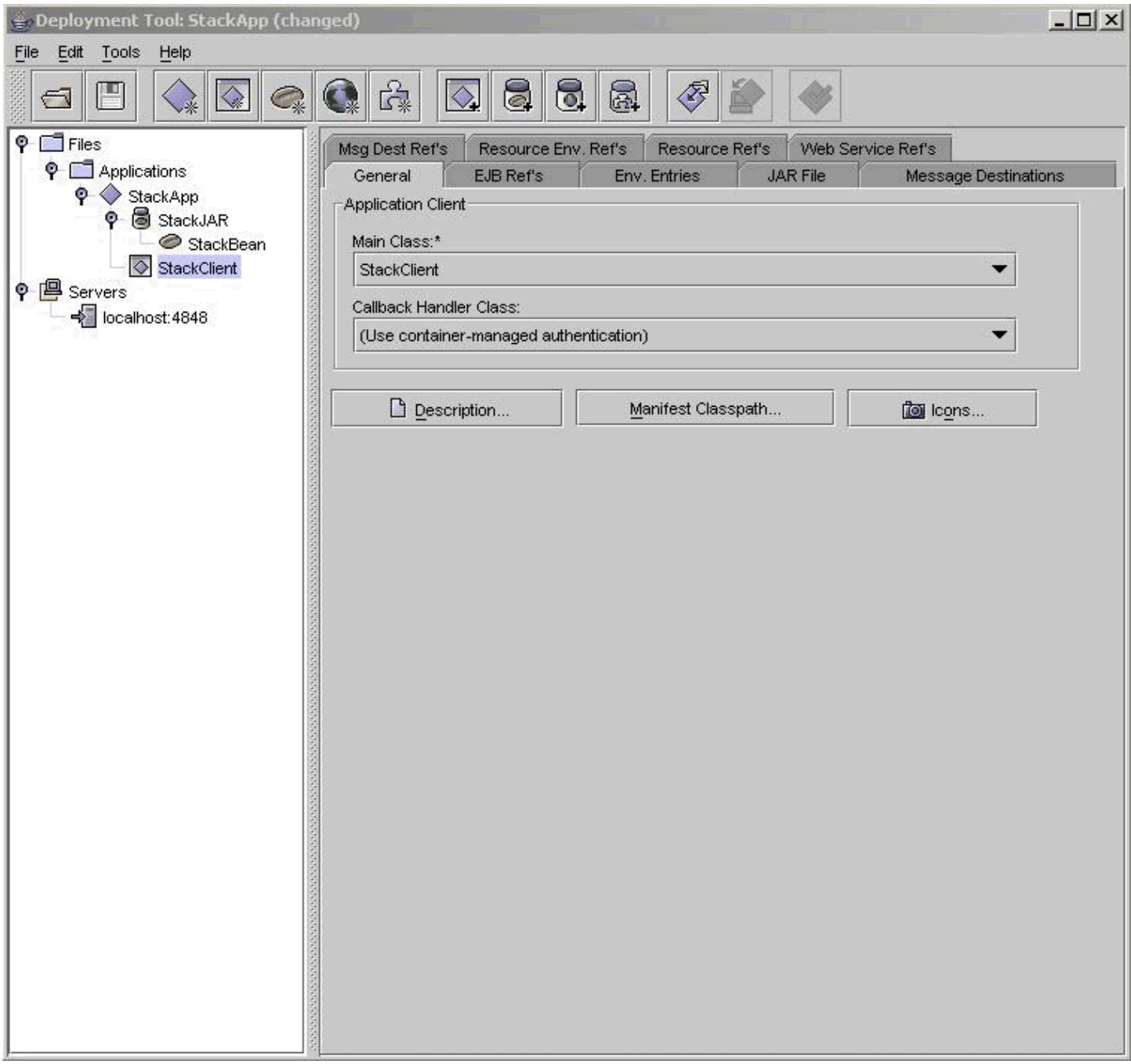
The application is created using a tool that comes with Sun's application server called deploytool. Start deploytool:

File => New => Application

Browse to the folder in which you would like the new application to be created, and type in the new application file name (in our case the file name is StackApp). Click OK. An application is created inside an application directory. We can now create the bean, the deployment descriptor, and the ejb-jar file. The steps for doing this can be found at various online training sites[14]. The next screen shot displays the finished application. There are some points that should be kept in mind when creating the application and creating the beans because they can enhance security in the application:

1. If the application uses entity beans, it becomes even more important to secure access to the application server because the usernames and passwords for the database are now going to be stored in the application server. At this point it becomes important to think of some way of locking down the server and restricting physical access to the server.
2. As much as possible, do not allow direct access to entity beans to the applications. In other words, do not let non-entity bean clients get direct access to entity beans. This can be prevented by creating a layer of session beans which have access to the enterprise beans and which, in turn, can be accessed by the client applications.
3. Remember that if the security in the application server has been set up by the container, and database security has been configured properly in a relational database such as Oracle with its object privileges, system privileges, roles, and profiles then we have an unbreakable system. This is not to say that the systems are not vulnerable to denial of service attacks or to buffer overflow problems, but the confidentiality of the data which can be so important these days can be preserved and maintained by following some basic procedures.
4. As much as possible, the application should try to minimize the number of calls to the remote interface. Rather than call a series of methods in the remote interface in order to initialize values in the application, create an object with the required information, and pass the object to the interface. This improves performance and minimizes the transmission of data over the network.

[14] http://java.sun.com/developer/onlineTraining/J2EE/Intro2/j2ee.html



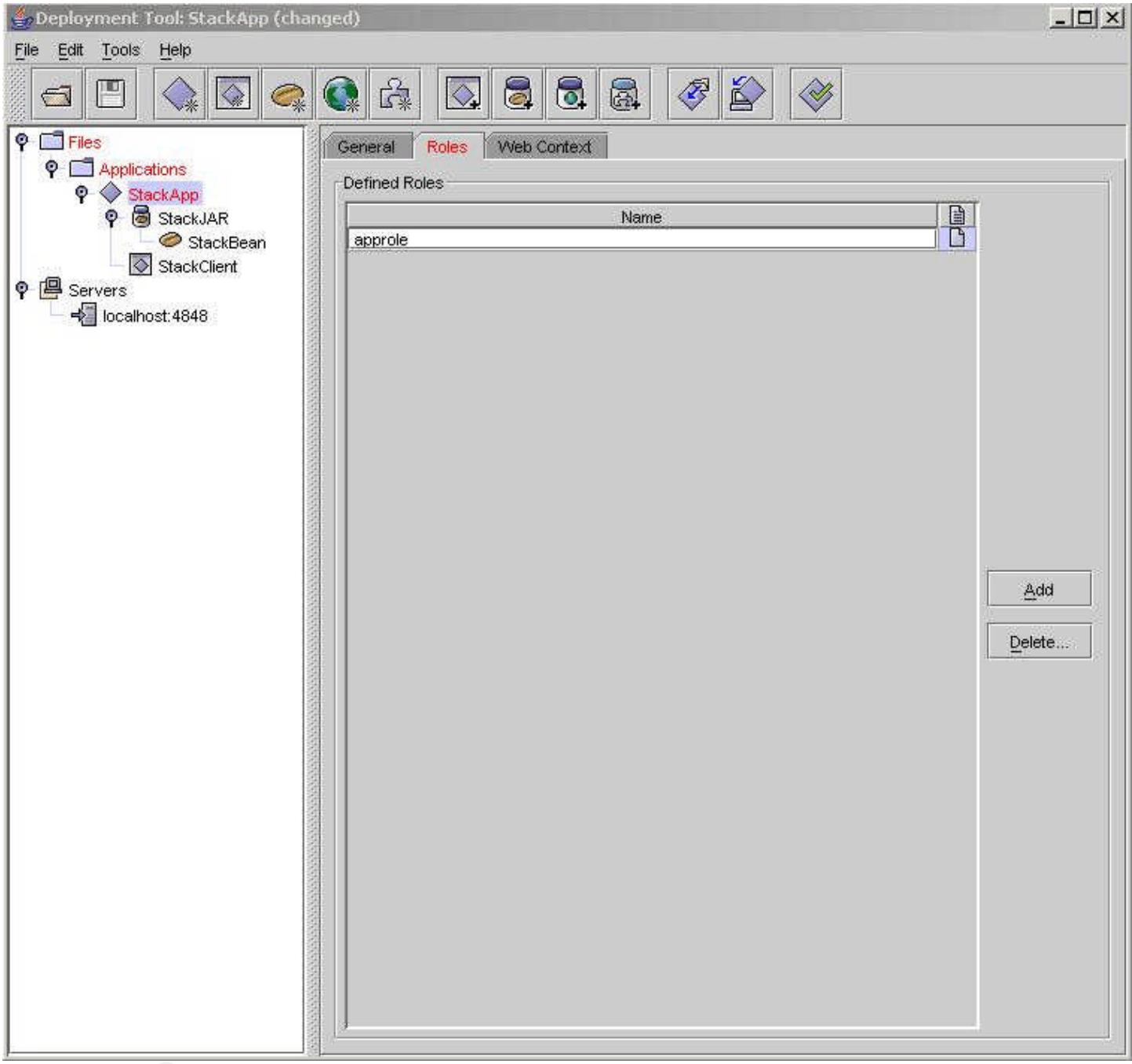
Let us take a look at the deployment descriptor for our JAR file which has been generated by the application server.

```
<?xml version='1.0' encoding='UTF-8'?>
<ejb-jar
        xmlns="http://java.sun.com/xml/ns/j2ee"
        version="2.1"
        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/ejb-jar_2_1.xsd"
        >
    <display-name>StackJAR</display-name>
    <enterprise-beans>
        <session>
            <ejb-name>StackBean</ejb-name>
            <home>StackHome</home>
            <remote>Stack</remote>
            <ejb-class>StackBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Bean</transaction-type>
            <security-identity>
                <use-caller-identity>
                </use-caller-identity>
            </security-identity>
        </session>
    </enterprise-beans>
</ejb-jar>
```

There are a few important points to note. The name of the bean has been saved in the file along with the names of the remote interface and the home interface. The type of the bean is displayed (stateless) along with the type of the transaction (default in this case). No entry has yet been made for security because we have not configured security for our bean class yet.

Step 4: Securing the Bean

The first step in securing our application consists in creating a role for our application. The J2EE security model makes extensive use of roles. To assign a role to a user or a group is to give the user or group access to different components in the application. For example, a user who belongs to the group manager may have the privileges to look up the hire date of an employee. Similarly, a user who belongs to the group payroll may have the privilege to query an employees' salary. As we will see later, it is the roles which are granted access to the methods of an enterprise bean. The roles can be granted access to individual methods in the bean class or to an entire bean. Either way it is role based security that will concern us in J2EE applications. To create a role, Select StackApp and click on the Roles tab. Click on the Add button and create a role by giving it a name.



This security role gets created in the <assembly-descriptor> part of the deployment descriptor as follows:

```
<assembly-descriptor>
```

```
        <security-role>
            <role-name>approle</role-name>
        </security-role>
       …
       …
</assembly-descriptor>
```

If we had created two roles, (approle and managerrole) our deployment descriptor would look like this:

```
<assembly-descriptor>
        <security-role>
            <role-name>approle</role-name>
        </security-role>
        <security-role>
            <role-name>managerrole</role-name>
        </security-role>
       …
       …
</assembly-descriptor>
```

The code that is contained in the remote interface and the home interface contains methods which can be called by the client. When we configure security for the application, the most important task for us is to configure method permissions for the bean. In other words, we have to decide which methods (belonging to the bean) are going to have access to the client. The method permissions are also placed under the `<assembly-descriptor>` tag in the deployment descriptor.

```
    <assembly-descriptor>
        <security-role>
            <role-name>approle</role-name>
        </security-role>
        <method-permission>
            <role-name>approle</role-name>
            <method>
                <ejb-name>StackBean</ejb-name>
                <method-intf>Remote</method-intf>
                <method-name>stuff</method-name>
                <method-params>
                    <method-param>int</method-param>
                </method-params>
            </method>
          ….
    </assembly-descriptor>
```

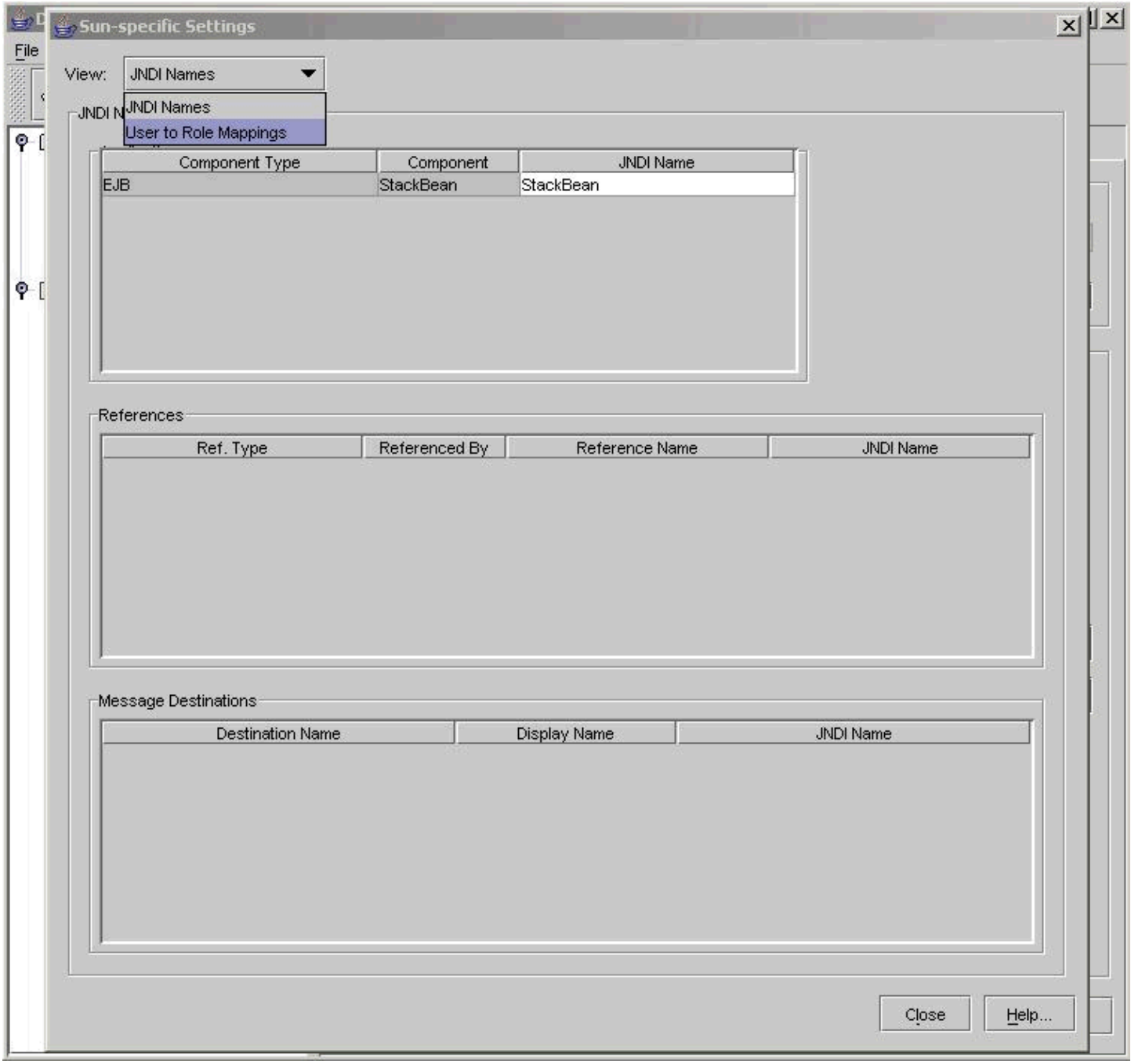
There are, basically, four ways in which method permissions can be defined. We can use the wildcard (*) to indicate that the client has permission to access all the methods in the bean or we can use the empty </unchecked> tag to do the same thing, or we can specify the methods that the client can access by name, or we can specify the methods that are available to the client by name and arguments in order to distinguish between overloaded methods. Finally, we can describe method permissions by name

and interface. This last technique allows us to provide different access controls for methods which have the same name but which are contained in two different interfaces. For example, one of the methods might be in the home interface while the other is in the component (remote) interface. In the following example, the last technique for specifying permissions has been used:

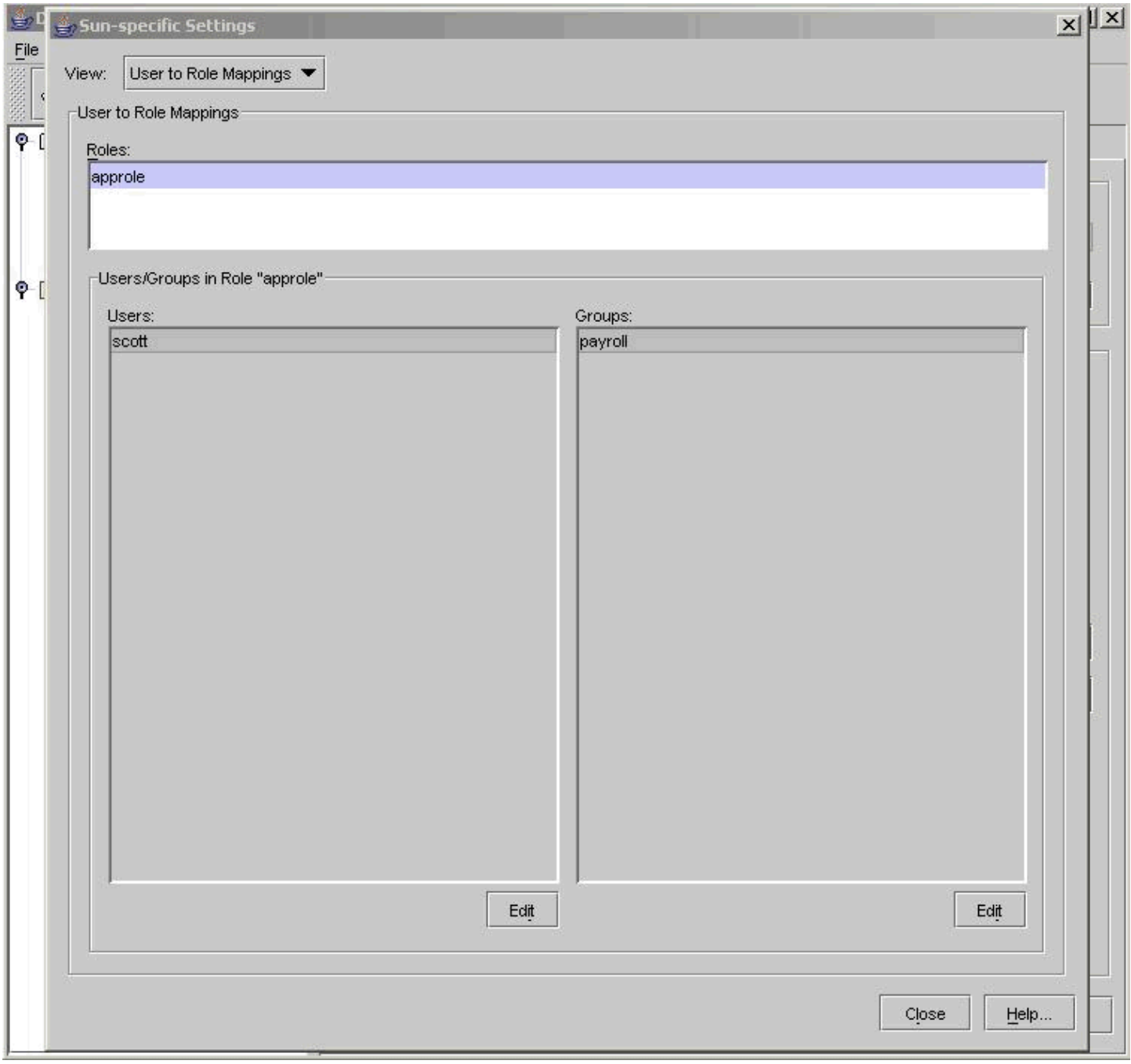
```
<assembly-descriptor>
        <security-role>
            <role-name>approle</role-name>
        </security-role>
        <method-permission>
            <role-name>approle</role-name>
            <method>
                <ejb-name>StackBean</ejb-name>
                <method-intf>Remote</method-intf>
                <method-name>stuff</method-name>
                <method-params>
                    <method-param>int</method-param>
                </method-params>
            </method>
            <method>
                <ejb-name>StackBean</ejb-name>
                <method-intf>Home</method-intf>
                <method-name>remove</method-name>
                <method-params>
                    <method-param>java.lang.Object</method-param>
                </method-params>
            </method>
            …
            …
      </method-permission>
</assembly-descriptor>
```

In the deployment descriptor, we are stating that approle is allowed to call the stuff() method in the remote or component interface of the StackBean (that takes an int parameter), as well as the remove() method in the home interface of the StackBean (which takes an Object parameter). The next step is to map this role to the users that we created in the Administrative Console. Recall that we created two users: sid and scott. Sid belonged to two groups: payroll and manager while scott belongs to only one group: payroll. To map the role (approle) to users do the following:

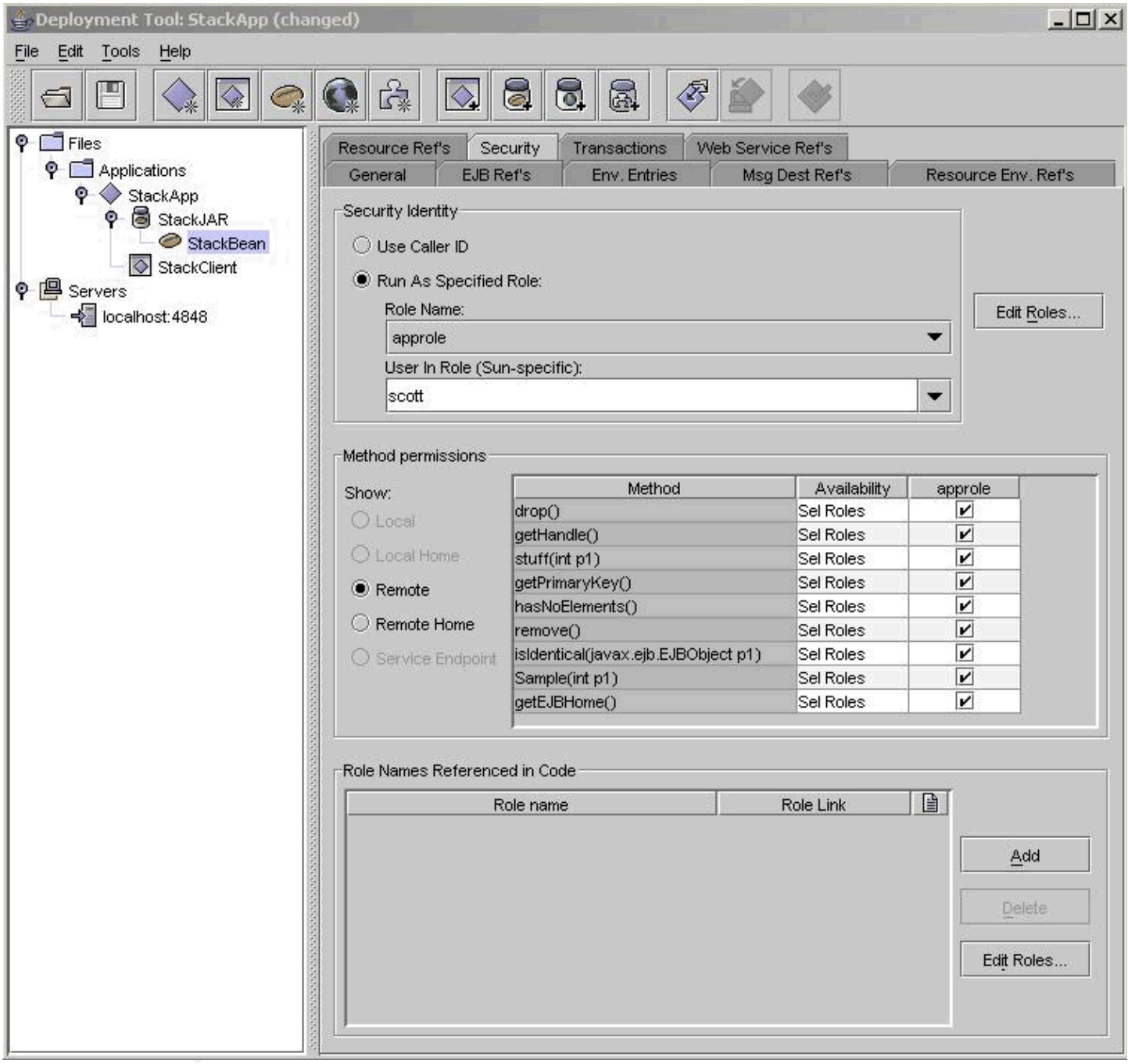
Select StackApp and the General tab. Click on Sun-specific Settings…



Under User to Role Mappings, connect to the deployment manager and add the users and groups to which this role will be mapped.

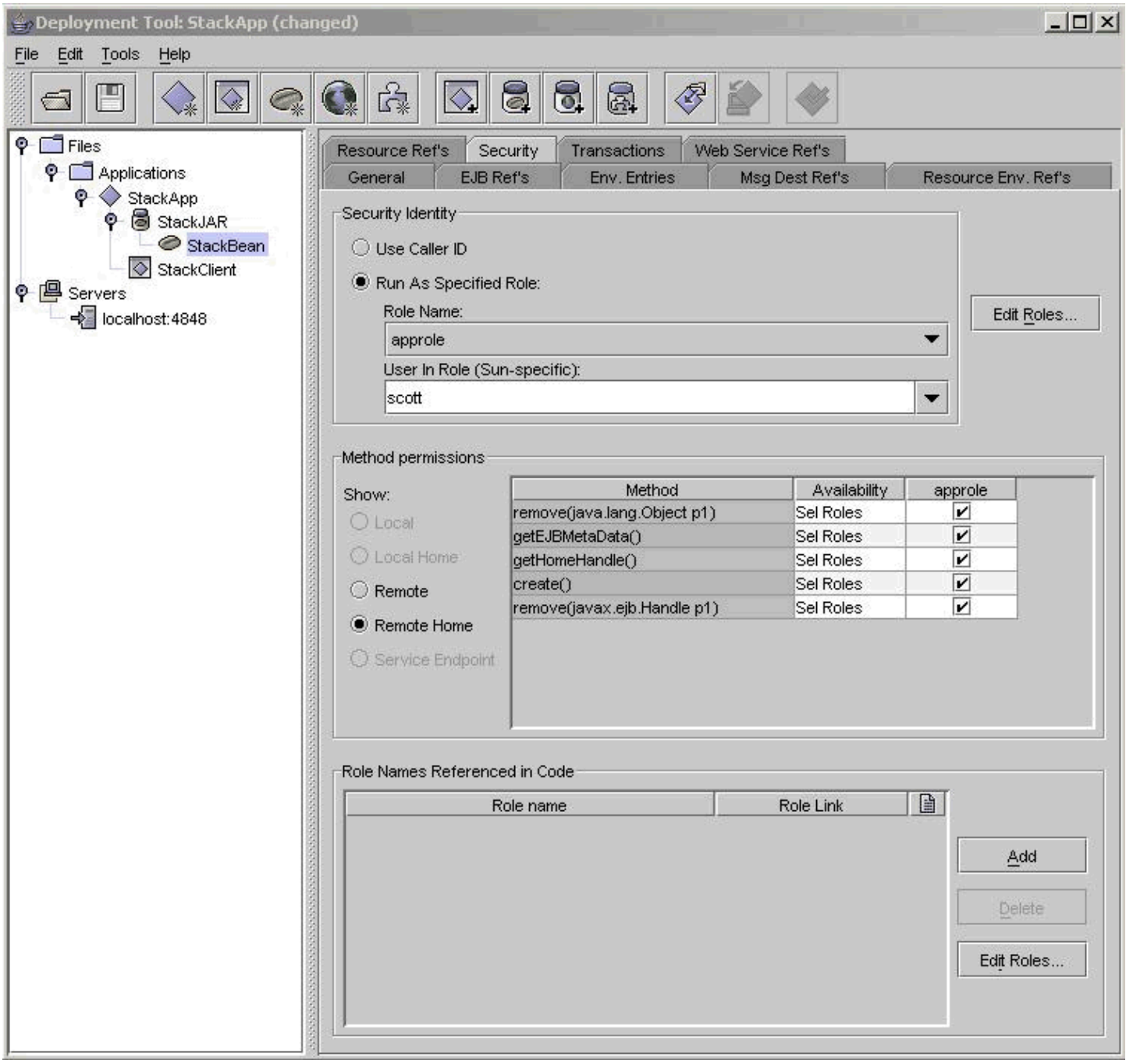


Now we are ready to set up the security for the bean. Select StackBean in the left window, and the Security tab on the right side.



This is the screen where all the security settings will be made for our enterprise bean, and the deployment descriptor will be generated for our application. The first thing that we want to do is to ensure that the identity of the user is determined by the role to which the user belongs by virtue of the fact that this particular user is a member of a given group. The next step is to ensure that the user has permission to call one or more methods on the bean class (through the remote interface). For the remote interface, for example, we may decide to give the user permission to call only some or all of the methods that are available. In our example, we have decided to give the user permission to call all the methods that belong to the bean. The same thing can be

done for the methods that are available in the remote home interface as the following screenshot illustrates:



Recall what we said about the home interface when we were talking about the remote interface. We said that the home interface is used by each client to get a handle on the remote interface so that the client is in a position to call the business methods on the bean by using the stub provided by the remote interface. If we want to deny the client access to a beans remote interface (and deny the client access to the bean), all we have to do is to deny the client access to the create() method in the home interface or

to the getHomeHandle() method in the home interface. If the client cannot use the create() method to return a reference to the remote interface, then the client cannot call any business method on the bean. In addition to the granting of direct access to methods in the remote interface, there are also other options that are available to us. We can decide to grant access to all users, to no user, or to selected roles. If we decide to grant access based on roles, then each one of the roles that we pick will appear on the right side. Again, all this information is gathered by the deploytool utility and saved to the deployment descriptor. If we now look at our previous deployment descriptor it will look something like this:

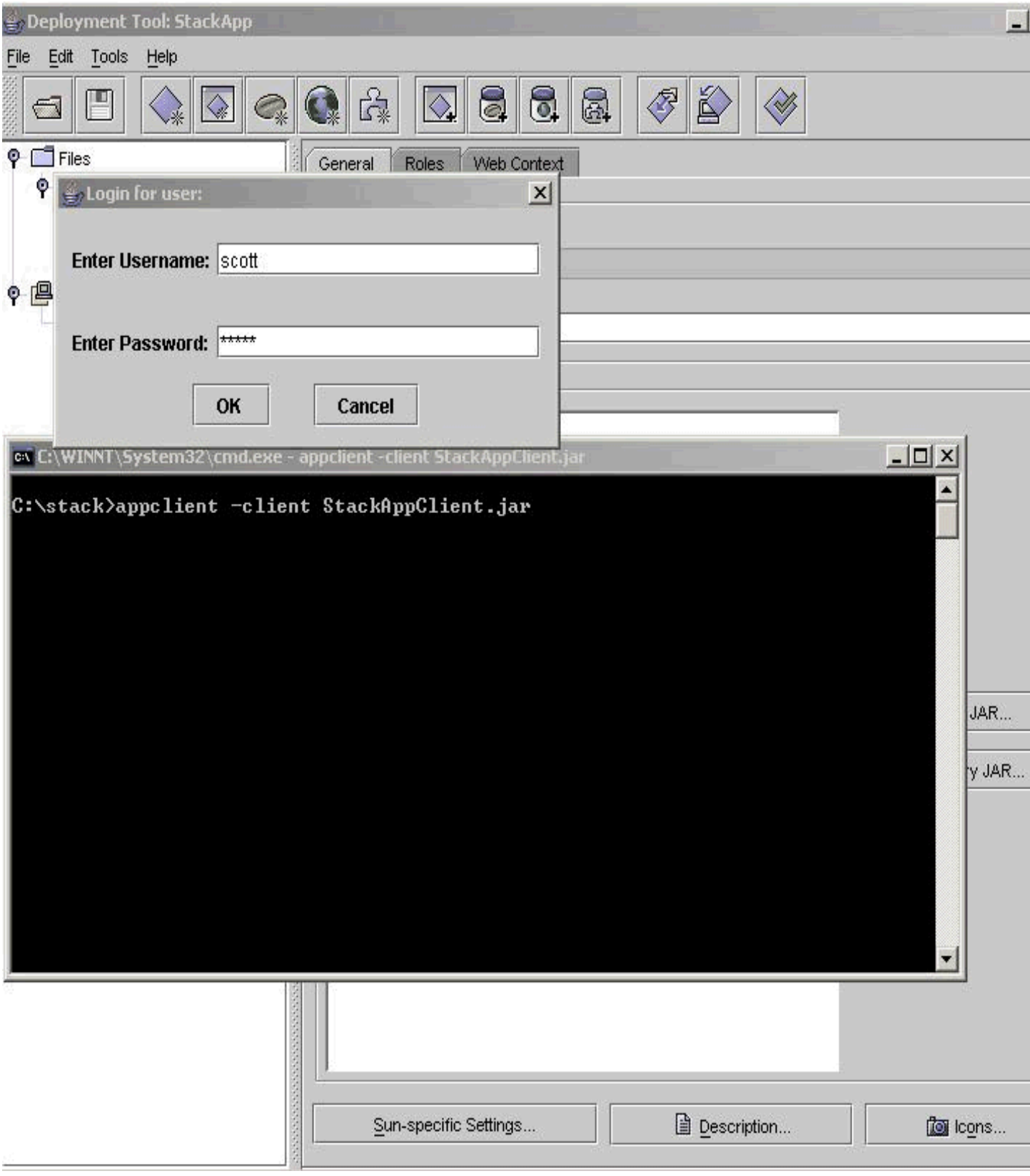
```
<?xml version='1.0' encoding='UTF-8'?>
<ejb-jar
        xmlns="http://java.sun.com/xml/ns/j2ee"
        version="2.1"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd"
        >
    <display-name>StackJAR</display-name>
    <enterprise-beans>
        <session>
            <ejb-name>StackBean</ejb-name>
            <home>StackHome</home>
            <remote>Stack</remote>
            <ejb-class>StackBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Bean</transaction-type>
            <security-identity>
                <run-as>
                    <role-name>approle</role-name>
                </run-as>
            </security-identity>
        </session>
    </enterprise-beans>
    <assembly-descriptor>
        <security-role>
            <role-name>approle</role-name>
        </security-role>
        <method-permission>
            <unchecked>
            </unchecked>
        </method-permission>
        <method-permission>
            <role-name>approle</role-name>
            <method>
                <ejb-name>StackBean</ejb-name>
                <method-intf>Home</method-intf>
                <method-name>remove</method-name>
                <method-params>
                    <method-param>java.lang.Object</method-param>
                </method-params>
            </method>
            <method>
                <ejb-name>StackBean</ejb-name>
                <method-intf>Remote</method-intf>
                <method-name>drop</method-name>
```

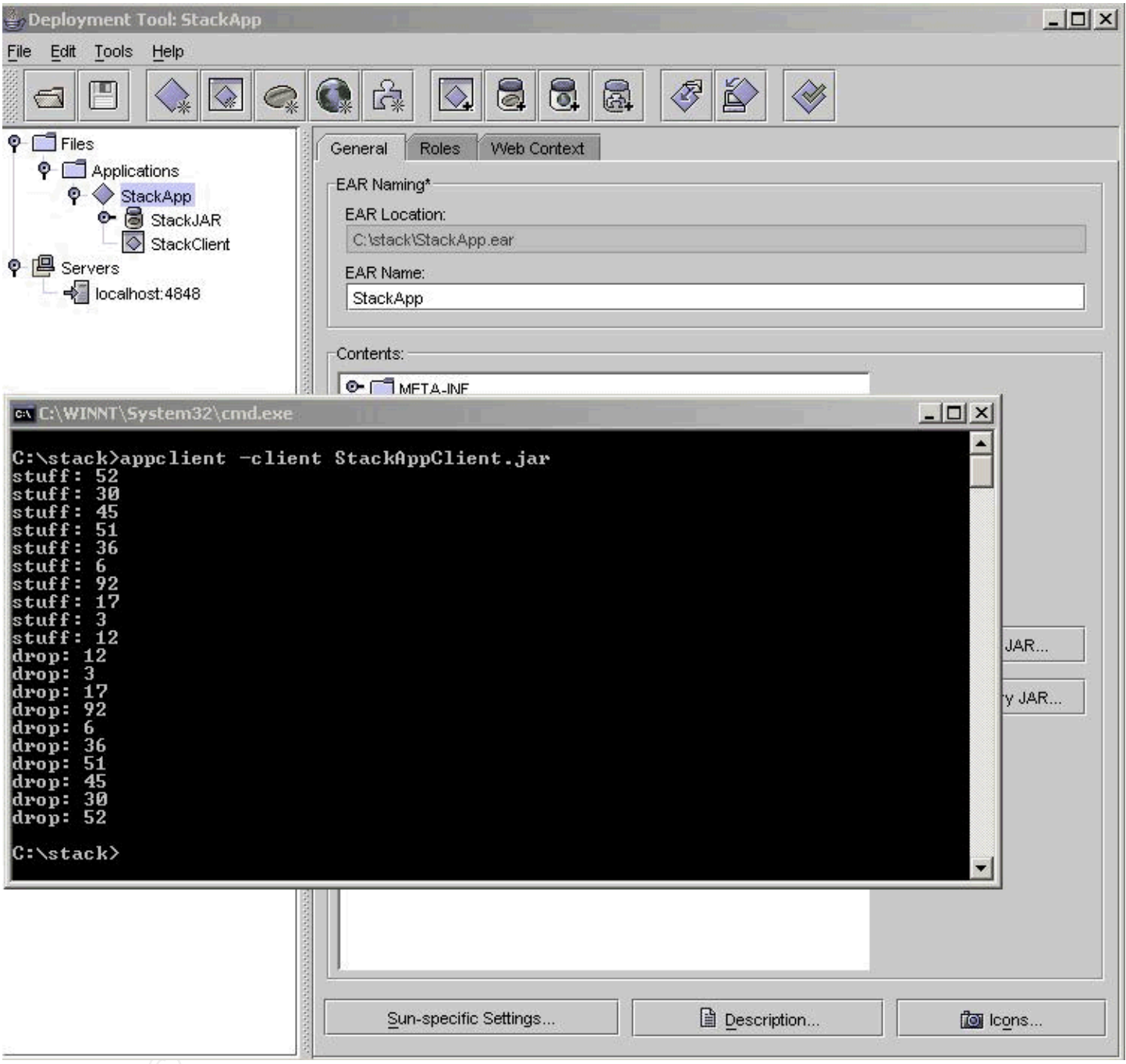
```
            </method>
            …
            …
        </method-permission>
    </assembly-descriptor>
</ejb-jar>
```

The fact that we have given unrestricted access to the methods in the bean is indicated by the </unchecked> tag in the deployment descriptor:

```
        <method-permission>
            <unchecked>
            </unchecked>
        </method-permission>
```

If a deployment descriptor has the </unchecked> tag in it, this tag will override any other permissions that you may have granted or denied to the client application. Thus, it is important to ensure that the <unchecked> tag does not occur in the XML file if you are trying to block method permissions on the bean. We can now deploy the application into the container of the application server and run it. The application is invoked from the command line by referring to the jar file that was created during the deployment process. In this example, the client and the server are on one machine. If we were working within a truly multi-tier environment, the jar file would be exported to the client machine during or after the deployment. Note how the login box displays to allow us to enter the password for the user scott. Once the password has been entered, the container checks for authorization before executing the methods on the bean class.





Summary:

In this paper we have looked at how to configure application security in Sun's java System Application Server 8.0. The requirement that an enterprise application be secure, scalable, and available can only be fulfilled in a multi-tiered environment. Typically, a multi-tiered application consists of three tiers: the presentation tier, the business logic tier, and the data tier. We have shown how security can be configured on the middle tier for enterprise java beans. In particular, we have seen how to configure declarative security (or container managed security) for enterprise java beans. The configuration of container managed security requires the creation of users and groups, as well as the creation of roles and the mapping of users and groups to these roles. It also involves container managed authentication and authorization, through the granting of permissions to roles. Permissions are granted in order that a user may execute certain methods belonging to an enterprise java bean. Restrictions can be placed on the users' ability to execute one or more methods belonging to the bean.

References

The following materials were consulted in the preparation of this document.

[1] Java™ 2 Platform Enterprise Edition Specification, v1.4. 24 Nov. 2003 http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf

[2] Eric Armstrong, Jennifer Ball, Stephanie Bodoff, Debbie Bode Carson, Ian Evans, Dale Green, Kim Haase, Eric Jendrock
The J2EE 1.4 Tutorial. December 16. 2004
< http://java.sun.com/j2ee/1.4/docs/tutorial/doc/ >

[3] Eric Armstrong, Jennifer Ball, Stephanie Bodoff, Debbie Bode Carson, Ian Evans, Dale Green, Kim Haase, Eric Jendrock
The J2EE 1.4 Tutorial. December 16. 2004
< http://java.sun.com/j2ee/1.4/docs/tutorial/doc/ >

[4] Eric Jendrock. "Security". The J2EE™ Tutorial. April 24, 2002 http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Security.html

[5] For more information on JAAS, please refer to the excellent discussion in the following: Charlie lai, Seema Malkani. "Implementing Security using JAAS and Java GSS API". Java One (Sun's 2003 Worldwide Java Developer Conference). < http://java.sun.com/security/javaone/2003/2236-JAASJGSS.pdf>

[6] "Securing J2EE Applications". Sun Java System Application Server Platform Edition 8 Developer's Guide (2004).
< http://docs.sun.com/source/817-6087/dgsecure.html>

[7] "Securing J2EE Applications". Sun Java System Application Server Platform Edition 8 Developer's Guide (2004) .
< http://docs.sun.com/source/817-6087/dgsecure.html>

[8] Weiss, Mark Allen. Data Structures and Problem Solving using Java. Addison Wesley, 2002

[9] Richard Monson-Haefel. Monson-Haefel, Richard. Enterprise Java Beans. O'Reilly (Third Edition), 2001.

[10] Eric Armstrong, Jennifer Ball, Stephanie Bodoff, Debbie Bode Carson, Ian Evans, Dale Green, Kim Haase, Eric Jendrock. The J2EE™ 1.4 Tutorial. June 17, 2004.
<http://java.sun.com/j2ee/1.4/docs/tutorial-update2/doc/index.html>

[11] Jyri Virkki and Marina Sum, Sun Java System Application Server 7 Access Control Guide. Jan 8. 2004.
http://developers.sun.com/prodtech/appserver/reference/techart/access_control.html>

[12] "Secure Hash Standard". Federal Information Processing Standards Publication 180--1
<http://www.itl.nist.gov/fipspubs/fip180-1.htm>

[13] Christoph Willie. Unbreakable Encryption Using One Time Pads. September 24, 2001
<http://www.aspheute.com/english/20010924.asp>

[14] Tutorials & Code Camps: Developing Enterprise Applications Using the J2EE Platform.
http://java.sun.com/developer/onlineTraining/J2EE/Intro2/j2ee.html