Using the Embedded Enterprise Bean Container

This chapter demonstrates how to use the embedded enterprise bean container to run enterprise bean applications in the Java SE environment, outside of a Java EE server.

The following topics are addressed here:

- Overview of the Embedded Enterprise Bean Container
- Developing Embeddable Enterprise Bean Applications
- The `standalone` Example Application

Overview of the Embedded Enterprise Bean Container

The embedded enterprise bean container is used to access enterprise bean components from client code executed in a Java SE environment.

The container and the client code are executed within the same virtual machine.

The embedded enterprise bean container is typically used for testing enterprise beans without having to deploy them to a server.

Most of the services present in the enterprise bean container in a Java EE server are available in the embedded enterprise bean container, including injection, container-managed transactions, and security.

Enterprise bean components execute similarly in both embedded and Java EE environments, and therefore the same enterprise bean can be easily reused in both standalone and networked applications.

Developing Embeddable Enterprise Bean Applications

All embeddable enterprise bean containers support the features listed in Table 26-1.

Table 26-1 Required Enterprise Bean Features in the Embeddable Container

Enterprise Bean Feature	Description
Local session beans	Local and no-interface view stateless, stateful, and singleton session beans. All method access is synchronous. Session beans must not be web service endpoints.
Transactions	Container-managed and bean-managed transactions.
Security	Declarative and programmatic security.
Interceptors	Class-level and method-level interceptors for session beans.

Deployment descriptor	The optional `ejb-jar.xml` deployment descriptor, with the same overriding rules for the enterprise bean container in Java EE servers.

Container providers are allowed to support the full set of features in enterprise beans, but applications that use the embedded container will not be portable if they use enterprise bean features not listed in Table 26-1, such as the timer service, session beans as web service endpoints, or remote business interfaces.

Running Embedded Applications

The embedded container, the enterprise bean components, and the client all are executed in the same virtual machine using the same classpath.

As a result, developers can run an application that uses the embedded container just like a typical Java SE application as follows:

```
java -classpath
mySessionBean.jar:containerProvider
Runtime.jar:myClient.jar
com.example.ejb.client.Main
```

In the above example, `mySessionBean.jar` is an EJB JAR containing a local stateless session bean,

`containerProviderRuntime.jar` is a JAR file supplied by the enterprise bean provider that contains the needed runtime classes for the embedded container, and `myClient.jar` is a JAR file containing a Java SE application that calls the business methods in the session bean through the embedded container.

Creating the Enterprise Bean Container

The `javax.ejb.embedded.EJBContainer` abstract class represents an instance of the enterprise bean container and includes factory methods for creating a container instance.

The `EJBContainer.createEJBContainer` method is used to create and initialize an embedded container instance.

The following code snippet shows how to create an embedded container that is initialized with the container provider's default settings:

```
EJBContainer ec =
EJBContainer.createEJBContainer();
```

By default, the embedded container will search the virtual machine classpath for enterprise bean modules:

directories containing a `META-INF/ejb-jar.xml` deployment descriptor, directories containing a class file with one of the enterprise bean component annotations (such as `@Stateless`), or JAR files containing an `ejb-jar.xml` deployment descriptor or class file with an enterprise bean annotation.

Any matching entries are considered enterprise bean modules within the same application.

Once all the valid enterprise bean modules have been found in the classpath, the container will begin initializing the modules.

When the `createEJBContainer` method successfully returns, the client application can obtain references to the client view of any enterprise bean module found by the embedded container.

An alternate version of the `EJBContainer.createEJBContainer` method takes a `Map` of properties and settings for customizing the embeddable container instance:

```
Properties props =
new Properties();
props.setProperty(...);
...
EJBContainer ec = EJBContainer.
```

```
createEJBContainer(props);
```

Explicitly Specifying

Enterprise Bean Modules to be Initialized

Developers can specify exactly which enterprise bean modules the embedded container will initialize.

To explicitly specify the enterprise bean modules initialized by the embedded container, set the `EJBContainer.MODULES` property.

The modules may be located either in the virtual machine classpath in which the embedded container and client code run, or alternately outside the virtual machine classpath.

To specify modules in the virtual machine classpath, set `EJBContainer.MODULES` to a `String` to specify a single module name, or a `String` array containing the module names.

The embedded container searches the virtual machine classpath for enterprise bean modules matching the specified names.

```
Properties props =
new Properties();
props.setProperty
(EJBContainer.MODULES,
"mySessionBean");
EJBContainer ec = EJBContainer.
createEJBContainer(props);
```

To specify enterprise bean modules outside the virtual machine classpath, set `EJBContainer.MODULES` to a `java.io.File` object or an array of `File` objects.

Each `File` object refers to an EJB JAR file, or a directory containing an expanded EJB JAR.

```
Properties props =
new Properties();
File ejbJarFile = new File(...);
props.setProperty
 (EJBContainer.MODULES, ejbJarFile);
EJBContainer ec = EJBContainer.
createEJBContainer(props);
```

Looking Up Session Bean References

To look up session bean references in an application using the embedded container, use an instance of `EJBContainer` to retrieve a `javax.naming.Context` object.

Call the `EJBContainer.getContext` method to retrieve the `Context` object.

```
EJBContainer ec =
EJBContainer.createEJBContainer();
Context ctx = ec.getContext();
```

References to session beans can then be obtained using the portable JNDI syntax detailed in Portable JNDI Syntax.

For example, to obtain a reference to MySessionBean, a local session bean with a no-interface view, use the following code:

```
MySessionBean msb = (MySessionBean)
ctx.lookup("java:global/
mySessionBean/MySessionBean");
```

Shutting Down the Enterprise Bean Container

From the client, call the `close` method of the instance of `EJBContainer` to shut down the embedded container:

```
EJBContainer ec =
EJBContainer.createEJBContainer();
...
ec.close();
```

While clients are not required to shut down `EJBContainer` instances, doing so frees resources consumed by the embedded container.

This is particularly important when the virtual machine under which the client application is running has a longer lifetime than the client application.

The `standalone` Example Application

The `standalone` example application demonstrates how to create an instance of the embedded enterprise bean container in a JUnit test class and call a session bean business method.

Testing the business methods of an enterprise bean in a unit test allows developers to exercise the business logic of an application separately from the other application layers, such as the presentation layer, and without having to deploy the application to a Java EE server.

The `standalone` example has two main components: `StandaloneBean`, a stateless session bean, and `StandaloneBeanTest`, a JUnit test class that acts as a client to `StandaloneBean` using the embedded container.

StandaloneBean is a simple session bean exposing a local, no-interface view with a single business method, returnMessage, which returns "Greetings!" as a String.

```
@Stateless
public class StandaloneBean {
private static final String message
= "Greetings!";
```

```
public String returnMessage()
{ return message; }
}
```

`StandaloneBeanTest` calls `StandaloneBean.returnMessage` and tests that the returned message is correct.

First, an embedded container instance and initial context are created within the `setUp` method, which is annotated with `org.junit.Before` to indicate that the method should be executed before the test methods.

```
@Before
public void setUp() {
ec =
EJBContainer.createEJBContainer();
ctx = ec.getContext();
```

```
}
```

The `testReturnMessage` method, annotated with `org.junit.Test` to indicate that the method includes a unit test, obtains a reference to `StandaloneBean` through the `Context` instance, and calls `StandaloneBean.returnMessage`.

The result is compared with the expected result

using a JUnit assertion, `assertEquals`.

If the string returned from `StandaloneBean.returnMessage` is equal to "Greetings!" the test passes.

```
@Test
public void testReturnMessage()
throws Exception {
```

```
logger.info("Testing
standalone.ejb.StandalonBean.return
Message()");
StandaloneBean instance =
(StandaloneBean)
ctx.lookup("java:global/classes/
StandaloneBean");
String expResult = "Greetings!";
String result =
instance.returnMessage();
assertEquals(expResult, result); }
```

Finally, the `tearDown` method, annotated with `org.junit.After` to indicate that the method should be executed after all the unit tests have run, closes the embedded container instance.

```
@After
public void tearDown() {
 if (ec != null) {
 ec.close();
 }
}
```

Running the `standalone` Example Application

Before You Begin

You must run the `standalone` example application within NetBeans IDE.

1. From the File menu, choose Open Project.

2. In the Open Project dialog, navigate to:
   `tut-install/examples/ejb/`

3. Select the `standalone` folder and click Open Project.

4. In the Projects tab, right-click `standalone` and select Test.

This will execute the JUnit test class `StandaloneBeanTest`.

The Output tab shows the progress of the test and the output log.