# Running the Enterprise Bean Examples

Session beans provide a simple but powerful way to encapsulate business logic within an application.

They can be accessed from remote Java clients, web service clients, and components running in the same server.

In Chapter 23, Getting Started with Enterprise Beans, you built a stateless session bean named `ConverterBean`.

This chapter examines the source code of four more session beans:

- `CartBean`: a stateful session bean that is accessed by a remote client
- `CounterBean`: a singleton session bean

- `HelloServiceBean`: a stateless session bean that implements a web service
- `TimerSessionBean`: a stateless session bean that sets a timer

The following topics are addressed here:

- The `cart` Example
- A Singleton Session Bean Example: `counter`
- A Web Service Example: `helloservice`
- Using the Timer Service
- Handling Exceptions

## The `cart` Example

The `cart` example represents a shopping cart in an online bookstore and uses a stateful session bean to manage the operations of the shopping cart.

The bean's client can add a book to the cart, remove a book, or retrieve the cart's contents.

## To assemble `cart`, you need the following code:

- Session bean class (`CartBean`)
- Remote business interface (`Cart`)

All session beans require a session bean class.

All enterprise beans that permit remote access must have a remote business interface.

To meet the needs of a specific application, an enterprise bean may also need some helper classes.

The `CartBean` session bean uses two helper classes, `BookException` and `IdVerifier`, which are discussed in the section Helper Classes.

The source code for this example is in the *tut-install*`/examples/ejb/cart/` directory.

## The Business Interface

The `Cart` business interface is a plain Java interface that defines all the business methods implemented in the bean class.

If the bean class implements a single interface, that interface is assumed to the business interface.

The business interface is a local interface unless it is annotated with the `javax.ejb.Remote` annotation; the `javax.ejb.Local` annotation is optional in this case.

The bean class may implement more than one interface.

In that case, the business interfaces must either be explicitly annotated `@Local` or `@Remote` or be specified by decorating the bean class with `@Local` or `@Remote`.

However, the following interfaces are excluded when determining whether the bean class implements more than one interface:

- `java.io.Serializable`
- `java.io.Externalizable`
- Any of the interfaces defined by the `javax.ejb` package

## The source code for the Cart business interface follows:

```
package
com.sun.tutorial.javaee.ejb;
import java.util.List;
import javax.ejb.Remote;
@Remote public interface Cart {
public void initialize
(String person)
throws BookException;
public void initialize
(String person, String id)
throws BookException;
public void addBook(String title);
public void removeBook
(String title)
throws BookException;
public List<String> getContents();
public void remove();}
```

## Session Bean Class

The session bean class for this example is called `CartBean`.

Like any stateful session bean, the `CartBean` class must meet the following requirements.

- The class is annotated `@Stateful`.
- The class implements the business methods defined in the business interface.

## Stateful session beans also may

- Implement the business interface, a plain Java interface.

  It is good practice to implement the bean's business interface.

- Implement any optional lifecycle callback methods, annotated `@PostConstruct`, `@PreDestroy`, `@PostActivate`, and `@PrePassivate`.

- Implement any optional business methods annotated `@Remove`.

## The source code for the CartBean class follows:

```
package
com.sun.tutorial.javaee.ejb;
import java.util.ArrayList;
import java.util.List;
import javax.ejb.Remove;
import javax.ejb.Stateful;
@Stateful
public class CartBean implements
Cart {
```

```
String customerName;
String customerId;
List<String> contents;
public void initialize
(String person)
throws BookException {
if (person == null) {
throw new BookException
("Null person not allowed.");
} else { customerName = person; }
customerId = "0";
```

```
contents = new ArrayList<String>();
}
public void initialize
(String person, String id)
throws BookException {
if (person == null) {
throw new BookException
("Null person not allowed.");
} else {
customerName = person;
}
```

```
IdVerifier idChecker =
new IdVerifier();
if (idChecker.validate(id)) {
customerId = id;
} else {
throw new BookException
("Invalid id: " + id);
}
contents = new ArrayList<String>();
}
```

```
public void addBook(String title)
{ contents.add(title); }
public void removeBook
(String title)throws BookException{
boolean result =
contents.remove(title);
if (result == false) {
throw new BookException
(title + " not in cart.");
}
}
```

```
public List<String> getContents()
{ return contents; }
@Remove
public void remove()
{ contents = null; }
}
```

## *Lifecycle Callback Methods*

A method in the bean class may be declared as a lifecycle callback method by annotating the method with the following annotations:

- `javax.annotation.PostConstruct`: Methods annotated with `@PostConstruct` are invoked by the container on newly constructed bean instances after all dependency injection has completed and before the first business method is invoked on the enterprise bean.

- `javax.annotation.PreDestroy`: Methods annotated with `@PreDestroy` are invoked after any method annotated `@Remove` has completed and before the container removes the enterprise bean instance.

- **`javax.ejb.PostActivate`: Methods annotated with `@PostActivate` are invoked by the container after the container moves the bean from secondary storage to active status.**

- `javax.ejb.PrePassivate`: Methods annotated with `@PrePassivate` are invoked by the container before it passivates the enterprise bean, meaning that the container temporarily removes the bean from the environment and saves it to secondary storage.

Lifecycle callback methods must return `void` and have no parameters.

## *Business Methods*

**The primary purpose of a session bean is to run business tasks for the client.**

**The client invokes business methods on the object reference it gets from dependency injection or JNDI lookup.**

From the client's perspective, the business methods appear to run locally, although they run remotely in the session bean.

The following code snippet shows how the `CartClient` program invokes the business methods:

```
cart.create("Duke DeEarl", "123");
...
cart.addBook("Bel Canto");
```

```
...
List<String> bookList =
cart.getContents();
...
cart.removeBook("Gravity's
Rainbow");
```

The `CartBean` class implements the business methods in the following code:

```
public void addBook(String title)
{ contents.addElement(title); }
public void removeBook
(String title)
throws BookException {
boolean result =
contents.remove(title);
if (result == false){
throw new BookException
(title + "not in cart.");
} }
```

```
public List<String> getContents()
{ return contents; }
```

The signature of a business method must conform to these rules.

- The method name must not begin with `ejb`, to avoid conflicts with callback methods defined by the EJB architecture.

For example, you cannot call a business method `ejbCreate` or `ejbActivate`.

- The access control modifier must be `public`.

- If the bean allows remote access through a remote business interface, the arguments and return types must be legal types for the Java Remote Method Invocation (RMI) API.

- If the bean is a web service endpoint, the arguments and return types for the methods annotated `@WebMethod` must be legal types for JAX-WS.

- The modifier must not be `static` or `final`.

The `throws` clause can include exceptions that you define for your application.

The `removeBook` method, for example, throws a `BookException` if the book is not in the cart.

To indicate a system-level problem, such as the inability to connect to a database, a business method should throw a `javax.ejb.EJBException`.

The container will not wrap application exceptions, such as `BookException`.

Because `EJBException` is a subclass of `RuntimeException`, you do not need to include it in the `throws` clause of the business method.

## The `@Remove` Method

Business methods annotated with `javax.ejb.Remove` in the stateful session bean class can be invoked by enterprise bean clients to remove the bean instance.

The container will remove the enterprise bean after a `@Remove` method completes, either normally or abnormally.

## In `CartBean`, the `remove` method is a `@Remove` method:

```
@Remove
public void remove()
{ contents = null; }
```

## Helper Classes

The `CartBean` session bean has two helper classes: `BookException` and `IdVerifier`.

The `BookException` is thrown by the `removeBook` method, and the `IdVerifier` validates the `customerId` in one of the `create` methods.

Helper classes may reside in an EJB JAR file that contains the enterprise bean class, a WAR file if the enterprise bean is packaged within a WAR, or in an EAR that contains an EJB JAR or a WAR file that contains an enterprise bean.

## Building, Packaging, Deploying, and Running the `cart` Example

Now you are ready to compile the remote interface (`Cart.java`), the home interface (`CartHome.java`), the enterprise bean class (`CartBean.java`), the client class (`CartClient.java`), and the helper classes (`BookException.java` and `IdVerifier.java`).

Follow these steps.

You can build, package, deploy, and run the `cart` application using either NetBeans IDE or the Ant tool.

## *To Build, Package, Deploy, and Run the `cart` Example Using 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 `cart` folder.

4. Select the Open as Main Project and Open Required Projects check boxes.

5. Click Open Project.

6. In the Projects tab, right-click the `cart` project and select Deploy.

This builds and packages the application into `cart.ear`, located in

*tut-install*`/examples/ejb/cart/dist/`,

and deploys this EAR file to your GlassFish Server instance.

7. From the Run menu, choose Run Main Project.

## You will see the output of the `cart` application client in the Output pane:

```
...
Retrieving book title from cart: Infinite Jest
Retrieving book title from cart: Bel Canto
Retrieving book title from cart: Kafka on the Shore
Removing "Gravity's Rainbow" from cart.
Caught a BookException: "Gravity's Rainbow" not in
cart.
Java Result: 1
run-cart-app-client:
run-nb:
BUILD SUCCESSFUL (total time: 14 seconds)
```

## *To Build, Package, Deploy, and Run the `cart` Example Using Ant*

1. In a terminal window, go to:

```
tut-install/examples/ejb/cart/
```

2. Type the following command:

```
ant
```

This command calls the `default` target, which builds and packages the application into an EAR file, `cart.ear`, located in the `dist` directory.

3. Type the following command:

```
ant deploy
```

The `cart.ear` file is deployed to the GlassFish Server.

4. Type the following command:

```
ant run
```

This task retrieves the application client JAR, `cartClient.jar`, and runs the application client.

The client JAR, `cartClient.jar`, contains the application client class, the helper class `BookException`, and the `Cart` business interface.

This task is equivalent to running the following command:

```
appclient -client cartClient.jar
```

When you run the client, the application client container injects any component references declared in the application client class, in this case the reference to the `Cart` enterprise bean.

## *The `all` Task*

**As a convenience, the `all` task will build, package, deploy, and run the application.**

**To do this, enter the following command:**

```
ant all
```

## A Singleton Session Bean Example: `counter`

The `counter` example demonstrates how to create a singleton session bean.

## Creating a Singleton Session Bean

The `javax.ejb.Singleton` annotation is used to specify that the enterprise bean implementation class is a singleton session bean:

```
@Singleton
public class SingletonBean { ... }
```

## *Initializing Singleton Session Beans*

**The EJB container is responsible for determining when to initialize a singleton session bean instance unless the singleton session bean implementation class is annotated with the `javax.ejb.Startup` annotation.**

In this case, sometimes called eager initialization, the EJB container must initialize the singleton session bean upon application startup.

The singleton session bean is initialized before the EJB container delivers client requests to any enterprise beans in the application.

This allows the singleton session bean to perform, for example, application startup tasks.

## The following singleton session bean stores the status of an application and is eagerly initialized:

```
@Startup
@Singleton
public class StatusBean {
private String status;
@PostConstruct
void init { status = "Ready"; }
...
}
```

Sometimes multiple singleton session beans are used to initialize data for an application and therefore must be initialized in a specific order.

In these cases, use the `javax.ejb.DependsOn` annotation to declare the startup dependencies of the singleton session bean.

The `@DependsOn` annotation's value attribute is one or more strings that specify the name of the target singleton session bean.

If more than one dependent singleton bean is specified in `@DependsOn`, the order in which they are listed is not necessarily the order in which the EJB container will initialize the target singleton session beans.

The following singleton session bean, `PrimaryBean`, should be started up first:

```
@Singleton
public class PrimaryBean { ... }
```

## SecondaryBean depends on PrimaryBean:

```
@Singleton
@DependsOn("PrimaryBean")
public class SecondaryBean { ... }
```

This guarantees that the EJB container will initialize PrimaryBean before SecondaryBean.

## The following singleton session bean, `TertiaryBean`, depends on `PrimaryBean` and `SecondaryBean`:

```
@Singleton
@DependsOn
("PrimaryBean", "SecondaryBean")
public class TertiaryBean { ... }
```

`SecondaryBean` explicitly requires `PrimaryBean` to be initialized before it is initialized, through its own `@DependsOn` annotation.

In this case, the EJB container will first initialize `PrimaryBean`, then `SecondaryBean`, and finally `TertiaryBean`.

If, however, `SecondaryBean` did not explicitly depend on `PrimaryBean`, the EJB container may initialize either `PrimaryBean` or `SecondaryBean` first.

That is, the EJB container could initialize the singletons in the following order: `SecondaryBean`, `PrimaryBean`, `TertiaryBean`.

## *Managing Concurrent Access in a Singleton Session Bean*

**Singleton session beans are designed for concurrent access, situations in which many clients need to access a single instance of a session bean at the same time.**

A singleton’s client needs only a reference to a singleton in order to invoke any business methods exposed by the singleton and doesn’t need to worry about any other clients that may be simultaneously invoking business methods on the same singleton.

When creating a singleton session bean, concurrent access to the singleton’s business methods can be controlled in two ways:

**container-managed concurrency and**

**bean-managed concurrency.**

**The `javax.ejb.ConcurrencyManagement` annotation is used to specify container-managed or bean-managed concurrency for the singleton.**

With `@ConcurrencyManagement`, a type attribute must be set to either `javax.ejb.ConcurrencyManagementType.CONTAINER` or `javax.ejb.ConcurrencyManagementType.BEAN`.

If no `@ConcurrencyManagement` annotation is present on the singleton implementation class, the EJB container default of container-managed concurrency is used.

## *Container-Managed Concurrency*

**If a singleton uses container-managed concurrency, the EJB container controls client access to the business methods of the singleton.**

**The `javax.ejb.Lock` annotation and a `javax.ejb.LockType` type are used to specify the access level of the singleton's business methods or `@Timeout` methods.**

Annotate a singleton's business or timeout method with `@Lock(READ)` if the method can be concurrently accessed, or shared, with many clients.

Annotate the business or timeout method with `@Lock(WRITE)` if the singleton session bean should be locked to other clients while a client is calling that method.

Typically, the `@Lock(WRITE)` annotation is used when clients are modifying the state of the singleton.

Annotating a singleton class with `@Lock` specifies that all the business methods and any timeout methods of the singleton will use the specified lock type unless they explicitly set the lock type with a method-level `@Lock` annotation.

If no `@Lock` annotation is present on the singleton class, the default lock type, `@Lock(WRITE)`, is applied to all business and timeout methods.

The following example shows how to use the `@ConcurrencyManagement`, `@Lock(READ)`, and `@Lock(WRITE)` annotations for a singleton that uses container-managed concurrency.

Although by default, singletons use container-managed concurrency, the `@ConcurrencyManagement(CONTAINER)` annotation may be added at the class level of the singleton to explicitly set the concurrency management type:

```
@ConcurrencyManagement(CONTAINER)
@Singleton
public class ExampleSingletonBean{
```

```
private String state;
@Lock(READ)
public String getState()
{return state;}
@Lock(WRITE)
public void setState
(String newState)
{ state = newState; }
}
```

The `getState` method can be accessed by many clients at the same time because it is annotated with `@Lock(READ)`.

When the `setState` method is called, however, all the methods in `ExampleSingletonBean` will be locked to other clients because `setState` is annotated with `@Lock(WRITE)`.

This prevents two clients from attempting to simultaneously change the `state` variable of `ExampleSingletonBean`.

The `getData` and `getStatus` methods in the following singleton are of type `READ`, and the `setStatus` method is of type `WRITE`:

```
@Singleton
@Lock(READ)
public class SharedSingletonBean {
private String data;
private String status;
public String getData()
{ return data; }
public String getStatus()
{ return status; }
@Lock(WRITE)
public void setStatus
 (String newStatus)
{ status = newStatus; }
}
```

**If a method is of locking type `WRITE`, client access to all the singleton's methods is blocked until the current client finishes its method call or an access timeout occurs.**

When an access timeout occurs, the EJB container throws a `javax.ejb.ConcurrentAccessTimeoutException`.

The `javax.ejb.AccessTimeout` annotation is used to specify the number of milliseconds before an access timeout occurs.

If added at the class level of a singleton, `@AccessTimeout` specifies the access timeout value for all methods in the singleton unless a method explicitly overrides the default with its own `@AccessTimeout` annotation.

The `@AccessTimeout` annotation can be applied to both `@Lock(READ)` and `@Lock(WRITE)` methods.

The `@AccessTimeout` annotation has one required element, `value`, and one optional element, `unit`.

By default, the `value` is specified in milliseconds.

To change the `value` unit, set `unit` to one of the `java.util.concurrent.TimeUnit` constants: `NANOSECONDS`, `MICROSECONDS`, `MILLISECONDS`, or `SECONDS`.

The following singleton has a default access timeout value of 120,000 milliseconds, or 2 minutes.

## The `doTediousOperation` method overrides the default access timeout and sets the value to 360,000 milliseconds, or 6 minutes.

```
@Singleton
@AccessTimeout(value=120000)
public class StatusSingletonBean {
private String status;
@Lock(WRITE)
public void setStatus
 (String new Status)
{ status = newStatus; }
@Lock(WRITE)
@AccessTimeout(value=360000)
public void doTediousOperation
{...}
}
```

The following singleton has a default access timeout value of 60 seconds, specified using the `TimeUnit.SECONDS` constant:

```
@Singleton
@AccessTimeout(value=60,
timeUnit=SECONDS)
public class StatusSingletonBean
{ ... }
```

## *Bean-Managed Concurrency*

Singletons that use bean-managed concurrency allow full concurrent access to all the business and timeout methods in the singleton.

The developer of the singleton is responsible for ensuring that the state of the singleton is synchronized across all clients.

Developers who create singletons with bean-managed concurrency are allowed to use the Java programming language synchronization primitives, such as `synchronization` and `volatile`, to prevent errors during concurrent access.

Add a `@ConcurrencyManagement` annotation at the class level of the singleton to specify bean-managed concurrency:

```
@ConcurrencyManagement(BEAN)
@Singleton
public class AnotherSingletonBean
{ ... }
```

## *Handling Errors in a Singleton Session Bean*

**If a singleton session bean encounters an error when initialized by the EJB container, that singleton instance will be destroyed.**

Unlike other enterprise beans, once a singleton session bean instance is initialized, it is not destroyed if the singleton's business or lifecycle methods cause system exceptions.

This ensures that the same singleton instance is used throughout the application lifecycle.

## The Architecture of the `counter` Example

The `counter` example consists of a singleton session bean, `CounterBean`, and a JavaServer Faces Facelets web front end.

`CounterBean` is a simple singleton with one method, `getHits`, that returns an integer representing the number of times a web page has been accessed.

Here is the code of `CounterBean`:

```
package counter.ejb;
import javax.ejb.Singleton;
```

```java
/**
 * CounterBean is a simple
 * singleton session bean
 * that records the number
 * of hits to a web page.
 */
@Singleton
public class CounterBean {
private int hits = 1;
// Increment and return
// the number of hits
public int getHits()
{return hits++;}
}
```

The `@Singleton` annotation marks `CounterBean` as a singleton session bean.

`CounterBean` uses a local, no-interface view.

`CounterBean` uses the EJB container's default metadata values for singletons to simplify the coding of the singleton implementation class.

There is no `@ConcurrencyManagement` annotation on the class, so the default of container-managed concurrency access is applied.

There is no `@Lock` annotation on the class or business method, so the default of `@Lock(WRITE)` is applied to the only business method, `getHits`.

The following version of `CounterBean` is functionally equivalent to the preceding version:

```
package counter.ejb;
import javax.ejb.Singleton;
import
javax.ejb.ConcurrencyManagement;
import static javax.ejb.
ConcurrencyManagementType.CONTAINER
;
import javax.ejb.Lock;
import javax.ejb.LockType.WRITE;
```

```
/**
 * CounterBean is a simple
 * singleton session bean
 * that records the number
 * of hits to a web page.
 */
@Singleton
@ConcurrencyManagement(CONTAINER)
public class CounterBean
{ private int hits = 1;
// Increment and return
// the number of hits
@Lock(WRITE)
public int getHits()
{ return hits++; }
}
```

The web front end of `counter` consists of a JavaServer Faces managed bean, `Count.java`, that is used by the Facelets XHTML files `template.xhtml` and `template-client.xhtml`.

The `Count` JavaServer Faces managed bean obtains a reference to `CounterBean` through dependency injection.

`Count` defines a `hitCount` JavaBeans property.

When the `getHitCount` getter method is called from the XHTML files, `CounterBean`'s `getHits` method is called to return the current number of page hits.

Here's the `Count` managed bean class:

```
@ManagedBean
@SessionScoped
public class Count {
@EJB
private CounterBean counterBean;
private int hitCount;
public Count()
{this.hitCount = 0;}
public int getHitCount() {
hitCount = counterBean.getHits();
return hitCount;
}
public void setHitCount
 (int newHits)
{ this.hitCount = newHits; }
}
```

The `template.xhtml` and `template-client.xhtml` files are used to render a Facelets view that displays the number of hits to that view.

The `template-client.xhtml` file uses an expression language statement, `#{count.hitCount}`, to access the `hitCount` property of the `Count` managed bean.

Here is the content of `template-client.xhtml`:

```
<?xml version='1.0'
encoding='UTF-8' ?>
```

```
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0
Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html
xmlns=
"http://www.w3.org/1999/xhtml"
xmlns:ui=
"http://java.sun.com/jsf/facelets"
```

```
xmlns:h=
"http://java.sun.com/jsf/html">
<body>
This text above will not be
displayed.
<ui:composition
template="/template.xhtml">
This text will not be displayed.
<ui:define name="title">
This page has been accessed
#{count.hitCount} time(s).
```

```
</ui:define>
This text will also not be
displayed.
<ui:define name="body">
Hooray!
</ui:define>
This text will not be displayed.
</ui:composition>
This text below will also not be
displayed.
</body> </html>
```

## Building, Packaging, Deploying, and Running the `counter` Example

The `counter` example application can be built, deployed, and run using NetBeans IDE or Ant.

## *To Build, Package, Deploy, and Run the `counter` Example Using 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 `counter` folder.

4. Select the Open as Main Project check box.

5. Click Open Project.

6. In the Projects tab, right-click the `counter` project and select Run.

A web browser will open the URL `http://localhost:8080/counter`, which displays the number of hits.

7. Click the browser's Refresh button to see the hit count increment.

## *To Build, Package, Deploy, and Run the `counter` Example Using Ant*

1. In a terminal window, go to:

```
tut-install/examples/ejb/counter
```

2. Type the following command:

```
ant all
```

This will build and deploy `counter` to your GlassFish Server instance.

3. In a web browser, type the following URL:

   `http://localhost:8080/counter`

4. Click the browser's Refresh button to see the hit count increment.

## A Web Service Example: `helloservice`

This example demonstrates a simple web service that generates a response based on information received from the client.

`HelloServiceBean` is a stateless session bean that implements a single method: `sayHello`.

This method matches the `sayHello` method invoked by the client described in A Simple JAX-WS Application Client.

## The Web Service Endpoint Implementation Class

`HelloServiceBean` is the endpoint implementation class, typically the primary programming artifact for enterprise bean web service endpoints.

## The web service endpoint implementation class has the following requirements.

- The class must be annotated with either the `javax.jws.WebService` or the `javax.jws.WebServiceProvider` annotation.

- The implementing class may explicitly reference an SEI through the `endpointInterface` element of the `@WebService` annotation but is not required to do so.

  If no `endpointInterface` is specified in `@WebService`, an SEI is implicitly defined for the implementing class.

- The business methods of the implementing class must be public and must not be declared `static` or `final`.

- Business methods that are exposed to web service clients must be annotated with `javax.jws.WebMethod`.

- **Business methods that are exposed to web service clients must have JAXB-compatible parameters and return types.**

  **See the list of JAXB default data type bindings at Types Supported by JAX-WS.**

- **The implementing class must not be declared `final` and must not be `abstract`.**

- **The implementing class must have a default public constructor.**

- **The endpoint class must be annotated `@Stateless`.**

- **The implementing class must not define the `finalize` method.**

- The implementing class may use the `javax.annotation.PostConstruct` or `javax.annotation.PreDestroy` annotations on its methods for lifecycle event callbacks.

  The `@PostConstruct` method is called by the container before the implementing class begins responding to web service clients.

The `@PreDestroy` method is called by the container before the endpoint is removed from operation.

## Stateless Session Bean Implementation Class

The `HelloServiceBean` class implements the `sayHello` method, which is annotated `@WebMethod`.

The source code for the `HelloServiceBean` class follows:

```
package
com.sun.tutorial.javaee.ejb;
import javax.ejb.Stateless;
import javax.jws.WebMethod;
import javax.jws.WebService;
@Stateless
@WebService
public class HelloServiceBean {
private String message = "Hello, ";
public void HelloServiceBean() {}
@WebMethod
public String sayHello(String name)
{ return message + name + "."; }
}
```

## Building, Packaging, Deploying, and Testing the `helloservice` Example

You can build, package, and deploy the `helloservice` example using either NetBeans IDE or Ant.

You can then use the Administration Console to test the web service endpoint methods.

## *To Build, Package, and Deploy the `helloservice` Example Using 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 `helloservice` folder.

4. Select the Open as Main Project and Open Required Projects check boxes.

5. Click Open Project.

6. In the Projects tab, right-click the `helloservice` project and select Deploy.

This builds and packages the application into `helloservice.ear`, located in

*tut-install*`/examples/ejb/`

`helloservice/dist`, and deploys this EAR file to the GlassFish Server.

## *To Build, Package, and Deploy the `helloservice` Example Using Ant*

1. In a terminal window, go to:

```
tut-install/examples/ejb/
helloservice/
```

2. Type the following command:

```
ant
```

**This runs the `default` task, which compiles the source files and packages the application into a JAR file located at**

***tut-install*/`examples/ejb/`**

**`helloservice/dist/`**

**`helloservice.jar`.**

3. To deploy `helloservice`, type the following command:

```
ant deploy
```

Upon deployment, the GlassFish Server generates additional artifacts required for web service invocation, including the WSDL file.

## *To Test the Service without a Client*

The GlassFish Server Administration Console allows you to test the methods of a web service endpoint.

To test the `sayHello` method of `HelloServiceBean`, follow these steps.

1. **Open the Administration Console by opening the following URL in a web browser:**

   `http://localhost:4848/`

2. **In the left pane of the Administration Console, select the Applications node.**

3. **In the Applications table, click `helloservice`.**

4. In the Modules and Components table, click View Endpoint.

5. On the Web Service Endpoint Information page, click the Tester link:

```
/HelloServiceBeanService/
HelloServiceBean?Tester
```

A Web Service Test Links page opens.

6. On the Web Service Test Links page, click the non-secure link (the one that specifies port 8080).

A HelloServiceBeanService Web Service Tester page opens.

7. Under Methods, type a name as the parameter to the `sayHello` method.

8. Click the `sayHello` button.

The `sayHello` Method invocation page opens.

Under Method returned, you'll see the response from the endpoint.

## Using the Timer Service

Applications that model business work flows often rely on timed notifications.

The timer service of the enterprise bean container enables you to schedule timed notifications for all types of enterprise beans except for stateful session beans.

You can schedule a timed notification to occur according to a calendar schedule, at a specific time, after a duration of time, or at timed intervals.

For example, you could set timers to go off at 10:30 a.m.

on May 23, in 30 days, or every 12 hours.

Enterprise bean timers are either programmatic timers or automatic timers.

Programmatic timers are set by explicitly calling one of the timer creation methods of the `TimerService` interface.

Automatic timers are created upon the successful deployment of an enterprise bean that contains a method annotated with the `java.ejb.Schedule` or `java.ejb.Schedules` annotations.

## Creating Calendar-Based Timer Expressions

Timers can be set according to a calendar-based schedule, expressed using a syntax similar to the UNIX `cron` utility.

Both programmatic and automatic timers can use calendar-based timer expressions.

Table 24-1 shows the calendar-based timer attributes.

## Table 24-1 Calendar-Based Timer Attributes

| Attribute | Description | Allowable Values | Default Value | Examples |
|---|---|---|---|---|
| `second` | One or more seconds within a minute | 0 to 59 | 0 | `second="30"` |

| | | | | |
|---|---|---|---|---|
| minute | One or more minutes within an hour | 0 to 59 | 0 | minute="15" |
| hour | One or more hours within a day | 0 to 23 | 0 | hour="13" |
| dayOfWeek | One or more days within a week | 0 to 7 (both 0 and 7 refer to Sunday) | * | dayOfWeek="3"<br>dayOfWeek="Mon" |

| | | | | |
|---|---|---|---|---|
| | | Sun, Mon, Tue, Wed, Thu, Fri, Sat | | |
| dayOfMonth | One or more days within a month | 1 to 31<br><br>−7 to −1 (a negative number means the *n*th day or days before the end of the month) | * | dayOfMonth="15"<br><br>dayOfMonth="-3"<br><br>dayOfMonth="Last"<br><br>dayOfMonth= "2nd Fri" |

| | | | | |
|---|---|---|---|---|
| | | Last [1st, 2nd, 3rd, 4th, 5th, Last] [Sun, Mon, Tue, Wed, Thu, Fri, Sat] | | |

| | | | | |
|---|---|---|---|---|
| `month` | One or more months within a year | `1` to `12`<br><br>`Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, `Dec` | `*` | `month="7"`<br><br>`month="July"` |
| `year` | A particular calendar year | A four–digit calendar year | `*` | `year="2010"` |

## *Specifying Multiple Values in Calendar Expressions*

**You can specify multiple values in calendar expressions, as described in the following sections.**

## *Using Wildcards in Calendar Expressions*

**Setting an attribute to an asterisk symbol (*) represents all allowable values for the attribute.**

**The following expression represents every minute:**

```
minute="*"
```

The following expression represents every day of the week:

```
dayOfWeek="*"
```

## *Specifying a List of Values*

To specify two or more values for an attribute, use a comma ( , ) to separate the values.

A range of values is allowed as part of a list.

Wildcards and intervals, however, are not allowed.

## Duplicates within a list are ignored.

The following expression sets the day of the week to Tuesday and Thursday:

```
dayOfWeek="Tue, Thu"
```

The following expression represents 4:00 a.m., every hour from 9:00 a.m. to 5:00 p.m. using a range, and 10:00 p.m.:

```
hour="4,9-17,22"
```

## *Specifying a Range of Values*

Use a dash character (–) to specify an inclusive range of values for an attribute.

Members of a range cannot be wildcards, lists, or intervals.

A range of the form `x-x`, is equivalent to the single-valued expression `x`.

A range of the form `x-y` where `x` is greater than `y` is equivalent to the expression `x-`*`maximum value`*`,` *`minimum value`*`-y`.

That is, the expression begins at `x`, rolls over to the beginning of the allowable values, and continues up to `y`.

The following expression represents 9:00 a.m. to 5:00 p.m.: hour="9–17"

The following expression represents Friday through Monday: dayOfWeek="5–1"

The following expression represents the twenty-fifth day of the month to the end of the month, and the beginning of the month to the fifth day of the month: dayOfMonth="25–5"

It is equivalent to the following expression:

```
dayOfMonth="25-Last,1-5"
```

## *Specifying Intervals*

The forward slash (`/`) constrains an attribute to a starting point and an interval and is used to specify every `N` seconds, minutes, or hours within the minute, hour, or day.

For an expression of the form `x/y`, `x` represents the starting point and `y` represents the interval.

The wildcard character may be used in the `x` position of an interval and is equivalent to setting `x` to `0`.

Intervals may be set only for `second`, `minute`, and `hour` attributes.

The following expression represents every 10 minutes within the hour: minute="*/10"

It is equivalent to:

```
minute="0,10,20,30,40,50"
```

The following expression represents every 2 hours starting at noon:

```
hour="12/2"
```

## Programmatic Timers

When a programmatic timer expires (goes off), the container calls the method annotated `@Timeout` in the bean's implementation class.

The `@Timeout` method contains the business logic that handles the timed event.

## *The `@Timeout` Method*

Methods annotated `@Timeout` in the enterprise bean class must return `void` and optionally take a `javax.ejb.Timer` object as the only parameter.

They may not throw application exceptions.

```
@Timeout
public void timeout(Timer timer) {
System.out.println
("TimerBean: timeout occurred");
}
```

## *Creating Programmatic Timers*

To create a timer, the bean invokes one of the `create` methods of the `TimerService` interface.

These methods allow single-action, interval, or calendar-based timers to be created.

For single-action or interval timers, the expiration of the timer can be expressed as either a duration or an absolute time.

The duration is expressed as a the number of milliseconds before a timeout event is triggered.

To specify an absolute time, create a `java.util.Date` object and pass it to the `TimerService.`

`createSingleActionTimer` or the `TimerService.createTimer` method.

The following code sets a programmatic timer that will expire in 1 minute (6,000 milliseconds):

```
long duration = 6000;
Timer timer =
timerService.
createSingleActionTimer
(duration, new TimerConfig());
```

**The following code sets a programmatic timer that will expire at 12:05 p.m. on May 1, 2010, specified as a `java.util.Date`:**

```
SimpleDateFormatter formatter =
new SimpleDateFormatter
("MM/dd/yyyy 'at' HH:mm");
Date date = formatter.parse
("05/01/2010 at 12:05");
Timer timer = timerService.
createSingleActionTimer
(date, new TimerConfig());
```

For calendar-based timers, the expiration of the timer is expressed as a `javax.ejb.ScheduleExpression` object, passed as a parameter to the `TimerService.createCalendarTimer` method.

The `ScheduleExpression` class represents calendar-based timer expressions and has methods that correspond to the attributes described in Creating Calendar-Based Timer Expressions.

The following code creates a programmatic timer using the `ScheduleExpression` helper class:

```
ScheduleExpression schedule =
new ScheduleExpression();
schedule.dayOfWeek("Mon");
schedule.hour("12-17, 23");
Timer timer = timerService.
createCalendarTimer(schedule);
```

For details on the method signatures, see the `TimerService` API documentation at http://download.oracle.com/javaee/6/api/javax/ejb/TimerService.html.

The bean described in The `timersession` Example creates a timer as follows:

```
Timer timer =
timerService.createTimer
(intervalDuration,
"Created new programmatic timer");
```

In the `timersession` example, `createTimer` is invoked in a business method, which is called by a client.

Timers are persistent by default.

If the server is shut down or crashes, persistent timers are saved and will become active again when the server is restarted.

If a persistent timer expires while the server is down, the container will call the `@Timeout` method when the server is restarted.

Nonpersistent programmatic timers are created by calling `TimerConfig.setPersistent(false)` and passing the `TimerConfig` object to one of the timer-creation methods.

The `Date` and `long` parameters of the `createTimer` methods represent time with the resolution of milliseconds.

However, because the timer service is not intended for real-time applications, a callback to the `@Timeout` method might not occur with millisecond precision.

The timer service is for business applications, which typically measure time in hours, days, or longer durations.

## Automatic Timers

Automatic timers are created by the EJB container when an enterprise bean that contains methods annotated with the `@Schedule` or `@Schedules` annotations is deployed.

An enterprise bean can have multiple automatic timeout methods, unlike a programmatic timer, which allows only one method annotated with the `@Timeout` annotation in the enterprise bean class.

Automatic timers can be configured through annotations or through the `ejb-jar.xml` deployment descriptor.

Adding a `@Schedule` annotation on an enterprise bean marks that method as a timeout method according to the calendar schedule specified in the attributes of `@Schedule`.

The `@Schedule` annotation has elements that correspond to the calendar expressions detailed in Creating Calendar-Based Timer Expressions and the `persistent`, `info`, and `timezone` elements.

The optional `persistent` element takes a Boolean value and is used to specify whether the automatic timer should survive a server restart or crash.

By default, all automatic timers are persistent.

The optional `timezone` element is used to specify that the automatic timer is associated with a particular time zone.

If set, this element will evaluate all timer expressions in relation to the specified time zone, regardless of the time zone in which the EJB container is running.

By default, all automatic timers set are in relation to the default time zone of the server.

The optional `info` element is used to set an informational description of the timer.

A timer's information can be retrieved later by using `Timer.getInfo`.

The following timeout method uses `@Schedule` to set a timer that will expire every Sunday at midnight:

```
@Schedule
(dayOfWeek="Sun", hour="0")
public void cleanupWeekData(){...}
```

The `@Schedules` annotation is used to specify multiple calendar-based timer expressions for a given timeout method.

The following timeout method uses the `@Schedules` annotation to set multiple calendar-based timer expressions.

The first expression sets a timer to expire on the last day of every month.

## The second expression sets a timer to expire every Friday at 11:00 p.m.

```
@Schedules ({
@Schedule(dayOfMonth="Last"),
@Schedule
(dayOfWeek="Fri", hour="23")
})
public void doPeriodicCleanup(){..}
```

## Canceling and Saving Timers

Timers can be canceled by the following events.

- When a single-event timer expires, the EJB container calls the associated timeout method and then cancels the timer.

- When the bean invokes the `cancel` method of the `Timer` interface, the container cancels the timer.

If a method is invoked on a canceled timer, the container throws the `javax.ejb.`

`NoSuchObjectLocalException`.

To save a `Timer` object for future reference, invoke its `getHandle` method and store the `TimerHandle` object in a database.

(A `TimerHandle` object is serializable.)

To reinstantiate the `Timer` object, retrieve the handle from the database and invoke `getTimer` on the handle.

A `TimerHandle` object cannot be passed as an argument of a method defined in a remote or web service interface.

In other words, remote clients and web service clients cannot access a bean's `TimerHandle` object.

Local clients, however, do not have this restriction.

## Getting Timer Information

In addition to defining the `cancel` and `getHandle` methods, the `Timer` interface defines methods for obtaining information about timers:

```
public long getTimeRemaining();
public java.util.Date
getNextTimeout();
public java.io.Serializable
getInfo();
```

The `getInfo` method returns the **object** that was the last parameter of the `createTimer` invocation.

For example, in the `createTimer` code snippet of the preceding section, this information parameter is a `String` object with the value `created timer`.

To retrieve all of a bean's active timers, call the `getTimers` method of the `TimerService` interface.

The `getTimers` method returns a collection of `Timer` objects.

## Transactions and Timers

An enterprise bean usually creates a timer within a transaction.

If this transaction is rolled back, the timer creation also is rolled back.

**Similarly, if a bean cancels a timer within a transaction that gets rolled back, the timer cancellation is rolled back.**

**In this case, the timer's duration is reset as if the cancellation had never occurred.**

**In beans that use container-managed transactions, the `@Timeout` method usually has the `Required` or `RequiresNew` transaction attribute to preserve transaction integrity.**

**With these attributes, the EJB container begins the new transaction before calling the `@Timeout` method.**

**If the transaction is rolled back, the container will call the `@Timeout` method at least one more time.**

## The `timersession` Example

The source code for this example is in the *tut-install*`/examples/ejb/timersession/src/java/` directory.

`TimerSessionBean` is a singleton session bean that shows how to set both an automatic timer and a programmatic timer.

In the source code listing of `TimerSessionBean` that follows, the `setTimer` and `@Timeout` methods are used to set a programmatic timer.

A `TimerService` instance is injected by the container when the bean is created.

Because it's a business method, `setTimer` is exposed to the local, no-interface view of `TimerSessionBean` and can be invoked by the client.

In this example, the client invokes `setTimer` with an interval duration of 30,000 milliseconds.

The `setTimer` method creates a new timer by invoking the `createTimer` method of `TimerService`.

Now that the timer is set, the EJB container will invoke the `programmaticTimeout` method of `TimerSessionBean` when the timer expires, in about 30 seconds.

```
...
public void setTimer
(long intervalDuration) {
logger.info("Setting a programmatic
timeout for " + intervalDuration
+ " milliseconds from now.");
```

```
Timer timer =
timerService.createTimer
(intervalDuration,
"Created new programmatic timer");
}
@Timeout
public void programmaticTimeout
(Timer timer){
this.setLastProgrammaticTimeout
(new Date());
```

```
logger.info
("Programmatic timeout occurred.");
}
...
```

`TimerSessionBean` also has an automatic timer and timeout method, `automaticTimeout`.

## The automatic timer is set to expire every 3 minutes and is set by using a calendar-based timer expression in the @Schedule annotation:

```
...
@Schedule(minute="*/3", hour="*")
public void automaticTimeout(){
this.setLastAutomaticTimeout
(new Date());
logger.info
("Automatic timeout occured");}...
```

`TimerSessionBean` also has two business methods: `getLastProgrammaticTimeout` and `getLastAutomaticTimeout`.

Clients call these methods to get the date and time of the last timeout for the programmatic timer and automatic timer, respectively.

Here's the source code for the `TimerSessionBean` class:

```
package timersession.ejb;
import java.util.Date;
import java.util.logging.Logger;
import javax.annotation.Resource;
import javax.ejb.Schedule;
import javax.ejb.Stateless;
import javax.ejb.Timeout;
import javax.ejb.Timer;
import javax.ejb.TimerService;
@Singleton
public class TimerSessionBean{
```

```
@Resource
TimerService timerService;
private Date
lastProgrammaticTimeout;
private Date lastAutomaticTimeout;
private Logger logger =
Logger.getLogger(
"com.sun.tutorial.javaee.ejb.
timersession.TimerSessionBean");
 public void setTimer
(long intervalDuration){
```

```
logger.info(
"Setting a programmatic timeout for "
+ intervalDuration +
" milliseconds from now.");
Timer timer =
timerService.createTimer
(intervalDuration,
"Created new programmatic timer");
}
```

```
@Timeout
public void programmaticTimeout
(Timer timer){
this.setLastProgrammaticTimeout
(new Date());
logger.info
("Programmatic timeout occurred.");
}
@Schedule(minute="*/3", hour="*")
public void automaticTimeout() {
```

```
this.setLastAutomaticTimeout
(new Date());
logger.info
("Automatic timeout occured");
}
public String
getLastProgrammaticTimeout(){
if
(lastProgrammaticTimeout != null) {
return
lastProgrammaticTimeout.toString();
```

```
} else { return "never"; }
}
public void
setLastProgrammaticTimeout
(Date lastTimeout) {
this.lastProgrammaticTimeout =
lastTimeout;
}
public String
getLastAutomaticTimeout() {
if (lastAutomaticTimeout != null) {
```

```
return
lastAutomaticTimeout.toString();
} else { return "never"; }
}
public void setLastAutomaticTimeout
(Date lastAutomaticTimeout) {
this.lastAutomaticTimeout =
lastAutomaticTimeout;
}
}
```

**Note - GlassFish Server has a default minimum timeout value of 1,000 milliseconds, or 1 second.**

**If you need to set the timeout value lower than 1,000 milliseconds, change the value of the `minimum-delivery-interval-in-millis` element in *domain-dir*`/config/domain.xml`.**

The lowest practical value for `minimum-delivery-interval-in-millis` is around 10 milliseconds, owing to virtual machine constraints.

## Building, Packaging, Deploying, and Running the `timersession` Example

You can build, package, deploy, and run the `timersession` example by using either NetBeans IDE or Ant.

## *To Build, Package, Deploy, and Run the `timersession` Example Using 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 `timersession` folder.**

4. Select the Open as Main Project check box.

5. Click Open Project.

6. From the Run menu, choose Run Main Project.

**This builds and packages the application into `timersession.war`, located in**

***tut-install*`/examples/ejb/`**

**`timersession/dist/`, deploys this WAR file to your GlassFish Server instance, and then runs the web client.**

## *To Build, Package, and Deploy the* `timersession` *Example Using Ant*

1. In a terminal window, go to:

```
tut-install/examples/ejb/
timersession/
```

2. Type the following command:

```
ant
```

This runs the `default` task, which compiles the source files and packages the application into a WAR file located at

*tut-install*`/examples/ejb/timersession/dist/timersession.war`.

3. To deploy the application, type the following command:

```
ant deploy
```

## *To Run the Web Client*

1. Open a web browser to `http://localhost:8080/timersession`.

2. Click the Set Timer button to set a programmatic timer.

3. Wait for a while and click the browser's Refresh button.

   You will see the date and time of the last programmatic and automatic timeouts.

   To see the messages that are logged when a timeout occurs, open the `server.log` file located in *domain-dir*`/server/logs/`.

## Handling Exceptions

The exceptions thrown by enterprise beans fall into two categories: system and application.

A system exception indicates a problem with the services that support an application.

**For example, a connection to an external resource cannot be obtained, or an injected resource cannot be found.**

**If it encounters a system-level problem, your enterprise bean should throw a `javax.ejb.EJBException`.**

Because the `EJBException` is a subclass of the `RuntimeException`, you do not have to specify it in the `throws` clause of the method declaration.

If a system exception is thrown, the EJB container might destroy the bean instance.

Therefore, a system exception cannot be handled by the bean's client program, but instead requires intervention by a system administrator.

An application exception signals an error in the business logic of an enterprise bean.

Application exceptions are typically exceptions that you've coded yourself, such as the `BookException` thrown by the business methods of the `CartBean` example.

When an enterprise bean throws an application exception, the container does not wrap it in another exception.

The client should be able to handle any application exception it receives.

If a system exception occurs within a transaction, the EJB container rolls back the transaction.

However, if an application exception is thrown within a transaction, the container does not roll back the transaction.