Martin Keen
Rafael Coutinho
Sylvi Lippmann
Salvatore Sollami
Sundaragopal Venkatraman
Steve Baber
Henry Cui
Craig Fleming

Developing Enterprise JavaBeans Applications

This IBM® Redpaper™ publication introduces Enterprise JavaBeans (EJB) and demonstrates by example how to create, maintain, and test EJB components. It explains how to develop session beans and describes the relationships between the session beans and the Java Persistence API (JPA) entity beans. Then, it integrates the EJB with a front-end web application for the sample application. It includes examples for creating, developing, and testing the EJB by using IBM Rational® Application Developer.

The paper is organized into the following sections:

► Introduction to Enterprise JavaBeans
► Developing an EJB module
► Testing the session EJB and the JPA entities
► Invoking EJB from web applications
► More information

The sample code for this paper is in the `4885code\ejb` folder.

This paper is intended for developers interested in creating, maintaining, and testing EJB components using IBM Rational Application Developer.

This paper was originally published as a chapter in the IBM Redbooks® publication, *Rational Application Developer for WebSphere Software V8 Programming Guide*, SG24-7835. The full publication includes working examples that show how to develop applications and achieve the benefits of visual and rapid application development. The publication is available at the following website:

http://www.redbooks.ibm.com/abstracts/sg247835.html?Open

Introduction to Enterprise JavaBeans

EJB is an architecture for server-side, component-based distributed applications written in Java. Details of the EJB 3.1 specification, EJB components and services, and new features in Rational Application Developer are described in the following sections:

- ► EJB 3.1 specification
- ► EJB component types
- ► EJB services and annotations
- ► EJB 3.1 application packaging
- ► EJB 3.1 Lite
- ► EJB 3.1 features in Rational Application Developer

EJB 3.1 specification

The EJB 3.1 specification is defined in *Java Specification Request (JSR) 318: Enterprise JavaBeans 3.1.* The JPA 2.0 specification and EJB 3.1 specification are separate. The specification of JPA 1.*x* was included in the EJB 3.0 specification. In this paper we describe the usage of the EJB 3.*x* specification, with a focus on EJB 3.1.

EJB 3.1 simplified model

Many publications discuss the complexities and differences between the old EJB 2.*x* programming model and the new EJB 3.*x*. For this reason, in this paper we focus on the new programming model. To overcome the limitations of the EJB 2.x, the new specification introduces a new simplified model with the following features:

- ► Entity EJB are now JPA entities, plain old Java objects (POJO) that show regular business interfaces, as plain old Java interface (POJI), and there is no requirement for home interfaces.

- ► The requirement for specific interfaces and deployment descriptors is removed (deployment descriptor information can be replaced by annotations).

- ► A new persistence model, which is based on the JPA standard, replaces EJB 2.*x* entity beans.

- ► An interceptor facility is used to start user methods at the invocation of business methods or at lifecycle events.

- ► Default values are used whenever possible ("configuration by exception" approach).

- ► Requirements for the use of checked exceptions are reduced.

- ► EJB 3.1 Lite, as a minimal subset of the full EJB 3.1 API, offers the major functions of EJB 3.1 API.

Figure 1 shows how the model of Java 2 Platform, Enterprise Edition (J2EE) 1.4 has been reworked with the introduction of the EJB 3.*x* specification. With the EJB 3.1 specification, this model was updated with new features. The new features are summarized in "EJB 3.1 features in Rational Application Developer" on page 20.
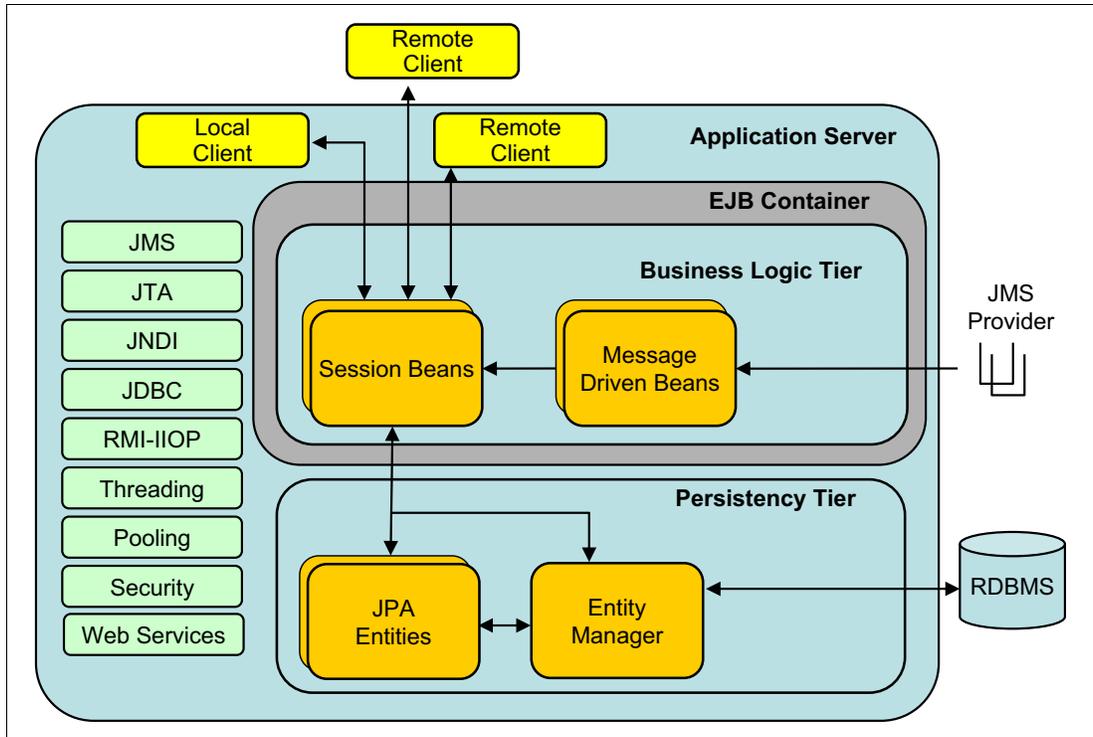

Figure 1   EJB 3.1 architecture

EJB component types

EJB 3.1 has the following component types of EJB:

► Session beans: stateless
► Session beans: stateful
► Session beans: Singleton bean
► Message-driven EJB (MDB)

This section defines several EJB.

Session beans

There are several kinds of session beans, the stateless and stateful EJB, and as a new feature, the definition of Singleton session beans. In this section, we describe these tasks:

► Defining a stateless session bean in EJB 3.1
► Defining a stateful session bean in EJB 3.1
► Defining a Singleton session bean in EJB 3.1

Additionally, we show the lifecycle events and leading practices for developing session beans.

Defining a stateless session bean in EJB 3.1

Stateless session EJB are always used to model a task that is performed for client code that invokes it. They implement the business logic or rules of a system, and provide the

coordination of those activities between beans, such as a banking service that allows for a transfer between accounts.

A stateless session bean is generally used for business logic that spans a single request and therefore cannot retain the client-specific state among calls.

Because a stateless session bean does not maintain a conversational state, all the data exchanged between the client and the EJB is passed either as input parameters, or as a return value, declared on the business method interface.

To declare a session stateless bean, add the @Stateless annotation to a POJO, as shown in Example 1.

Example 1   Definition of a stateless session bean

```
@Stateless
public class MyFirstSessionBean implements MyBusinessInterface {

    // business methods according to MyBusinessInterface
    .....
}
```

Note the following points in Example 1:

▶ MyFirstSessionBean is a POJO that shows a POJI, in this case, MyBusinessInterface. This interface is available to clients to invoke the EJB business methods. For EJB 3.1, a business interface is not required.

▶ The @Stateless annotation indicates to the container that the bean is a stateless session bean so that the appropriate lifecycle and runtime semantics can be enforced.

▶ By default, this session bean is accessed through a local interface.

This information is all that you need to set up a session EJB. There are no special classes to extend and no interfaces to implement.

Figure 2 shows the simple model of EJB 3.1.
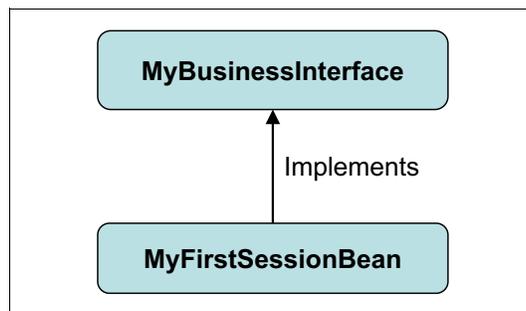
Figure 2   EJB is a POJO exposing a POJI

If we want to show the same bean on the remote interface, we use the @Remote annotation, as shown in Example 2 on page 5.

Example 2   Defining a remote interface for stateless session bean

```
@Remote(MyRemoteBusinessInterface.class)
@Stateless
public class MyBean implements MyRemoteBusinessInterface {
    // ejb methods
    .....
}
```

> **Tip:** If the session bean implements only one interface, you can use the `@Remote` annotation without a class name.

Defining a stateful session bean in EJB 3.1

Stateful session EJB are typically used to model a task or business process that spans multiple client requests. Therefore, a stateful session bean retains its state on behalf of an individual client. The client stores the handling of the stateful EJB, so that it always accesses the same EJB instance. By using the same approach that we adopted before, to define a stateful session EJB, you must declare a POJO with the annotation `@Stateful`, as shown in Example 3.

Example 3   Defining a stateful session bean

```
@Stateful
public class SecondSessionBean implements MyBusinessStatefulInterface {
    // ejb methods
    .....
}
```

The `@Stateful` annotation indicates to the container that the bean is a stateful session bean so that the appropriate lifecycle and runtime semantics can be enforced.

Defining a Singleton session bean in EJB 3.1

The definition of a Singleton session bean is a new feature of EJB 3.1. The definition of the Singleton pattern is associated with the defined design pattern in software engineering. You define a session bean as a Singleton to restrict the instantiation of this class to only one object. For example, the object, which coordinates actions across the system, can be defined as a Singleton, because only this object is responsible for the coordination. Therefore, you define the annotation `@Singleton` in front of the class declaration, as shown in Example 4. The new annotation `@LocalBean` is described in "Business interfaces" on page 6.

Example 4   Defining a Singleton session bean

```
@Startup
@Singleton
@LocalBean
public class MySingletonBean{

    // ejb methods
    .....
}
```

A Singleton session bean offers you the opportunity to initialize objects at application start-up. This function can replace proprietary IBM WebSphere® Application Server start-up beans. Therefore, you also define the new annotation `@Startup` in front of the class declaration, as

shown in Example 4 on page 5. As result, you have the initialization at the application start-up instead of the first invocation by the client code.

The concurrency in Singleton session beans is either defined as container-managed concurrency (CMC) or bean-managed concurrency (BMC). In case no annotation for the concurrency is specified in front of the class, the default value is CMC. The further default value for CMC is `@Lock(WRITE)`. If you want to define a class or method associated with a shared lock, use the annotation `@Lock(READ)`.

To define BMC, use the annotation `@ConcurrencyManagement` `(ConcurrencyManagementType.BEAN)`. After it is defined as BMC, the container allows full concurrent access to the Singleton session bean. Furthermore, you can define that concurrency is not allowed. Therefore, use the annotation `@ConcurrencyManagement` `(ConcurrencyManagementType.CONCURRENCY_NOT_ALLOWED)`.

For detailed information about Singleton session beans, see section 3.4.7.3 "Singleton Session Beans" in *JSR 318: Enterprise JavaBeans 3.1.*

Business interfaces

EJB can show various business interfaces, because the EJB can be accessed from either a local or remote client. Therefore, place common behavior to both local and remote interfaces in a superinterface, as shown in Figure 3.

You must ensure the following aspects:

► A business interface cannot be a local and a remote business interface of the bean.

► If a bean class implements a single interface, that interface is assumed to be the business interface of the bean. This business interface is a *local* interface, unless the interface is designated as a remote business interface by use of the `@Remote` annotation or with the deployment descriptor.

This approach provides flexibility during the design phase, because you can decide which methods are visible to local and remote clients.
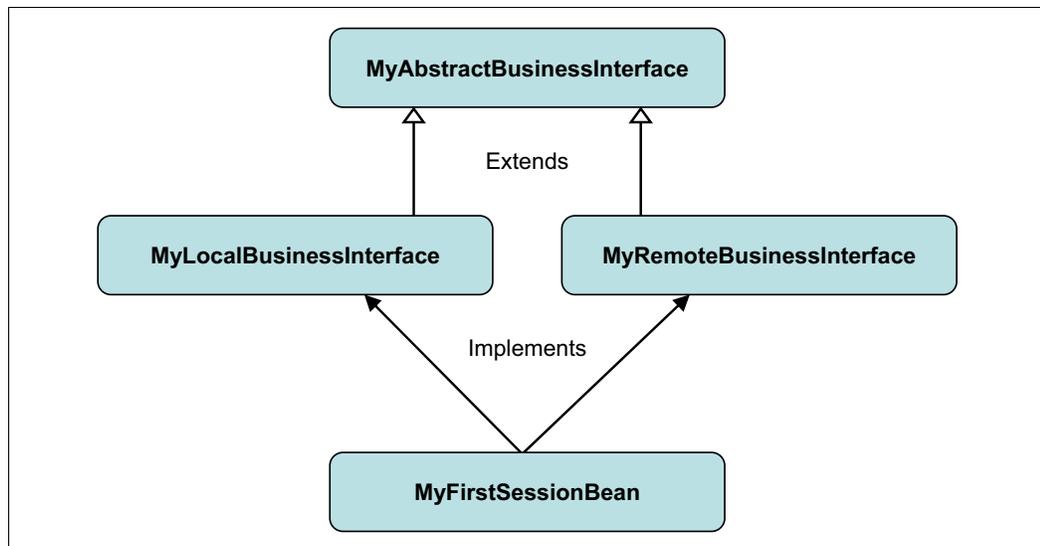

Figure 3   How to organize the EJB component interfaces

By using these guidelines, the first EJB is refactored, as shown in Example 5.

Example 5   Implementing local and remote interface

```
@Stateless
public class MyFirstSessionBean
      implements MyLocalBusinessInterface, MyRemoteBusinessInterface {

   // implementation of methods declared in MyLocalBusinessInterface
   ....

   // implementation of methods declared in MyRemoteBusinessInterface
   ....
}
```

The MyLocalBusinessInterface is declared as an interface with an @Local annotation, and the MyRemoteBusinessInterface is declared as an interface with the @Remote annotation, as shown in Example 6.

Example 6   Defining local and remote interface

```
@Local
public interface MyLocalBusinessInterface
                              extends MyAbstractBusinessInterface {

   // methods declared in MyLocalBusinessInterface
   ......
}


======================================================================


@Remote
public interface MyRemoteBusinessInterface
                              extends MyAbstractBusinessInterface {

   // methods declared in MyRemoteBusinessInterface
   ......
}
```

Another technique to define the business interfaces shown either as local or remote is to specify @Local or @Remote annotations with the full class name that implements these interfaces, as shown in Example 7 on page 8.

Example 7   Defining full class interfaces

```
@Stateless
@Local(MyLocalBusinessInterface.class)
@Remote(MyRemoteBusinessInterface.class)
public class MyFirstSessionBean implements MyLocalBusinessInterface,
                                           MyRemoteBusinessInterface {

   // implementation of methods declared in MyLocalBusinessInterface
   ....
   // implementation of methods declared in MyRemoteBusinessInterface
   ....
   }
```

You can declare any exceptions on the business interface, but be aware of the following rules:

► Do not use RemoteException.
► Any runtime exception thrown by the container is wrapped into an EJBException.

As a new feature of EJB 3.1, you can define a session bean without a local business interface. Therefore, the local view of a session bean can be accessed without the definition of a local business interface.

As shown in Figure 4, there is a new check box available that, when selected, dictates that no interface is created.
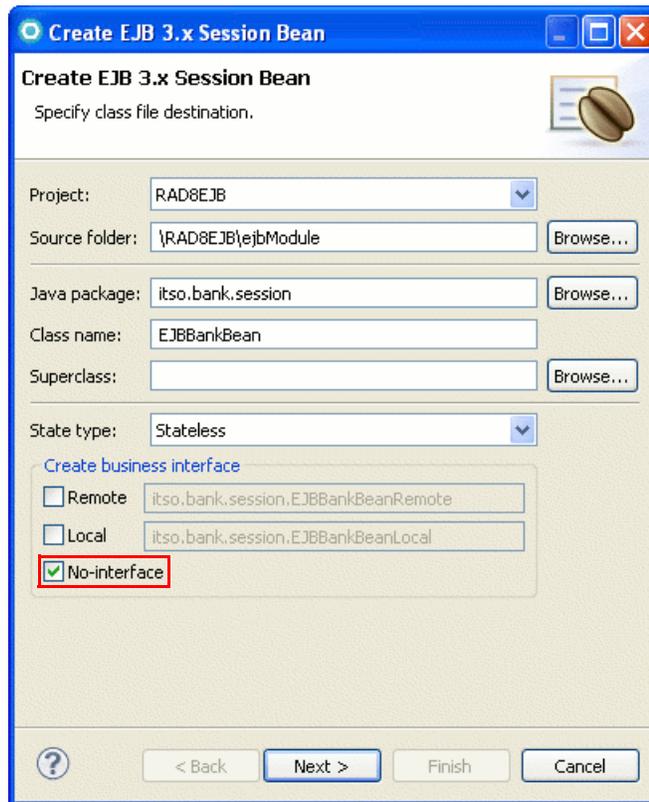
Figure 4   Creating a session bean without an interface

If you select No-interface in the Create EJB 3.x Session Bean wizard, the `@LocalBean` annotation is generated for your session bean, as shown in Example 8.

Example 8   Session bean with No-interface view

```
package itso.bank.session;
import javax.ejb.LocalBean;
import javax.ejb.Stateless;

/**
* Session Bean implementation class EJBBankBean
*/
@Stateless
@LocalBean
public class EJBBankBean{
/**
* Default constructor.
*/
   public EJBBankBean() {
   // TODO Auto-generated constructor stub
   }
...
}
```

For detailed information, see the *JSR 318: Enterprise JavaBeans 3.1* specification.

Lifecycle events

Another use of annotations is to *mark* callback methods for session bean lifecycle events.

EJB 2.1 and prior releases required the implementation of several lifecycle methods, such as `ejbPassivate`, `ejbActivate`, `ejbLoad`, and `ejbStore`, for every EJB. These methods were required even if you did not need them.

Lifecycle methods: As we use POJO in EJB 3.*x*, the implementation of these lifecycle methods is optional. The container invokes any callback method if you implement it in the EJB.

The lifecycle of a session bean can be categorized into several phases or events. The most obvious two events of a bean lifecycle are the creation of and destruction of stateless session beans.

After the container creates an instance of a session bean, the container performs any *dependency injection* (described in the following section) and then invokes the method annotated with `@PostConstruct`, if there is one.

The client obtains a reference to a session bean and invokes a business method.

Lifecycle of a stateless session bean: The lifecycle of a stateless session bean is independent of when a client obtains a reference to it. For example, the container might give a reference to the client, but not create the bean instance until later, when a method is invoked on the reference. In another example, the container might create several instances at start-up and match them with references later.

At the end of the lifecycle, the EJB container calls the method annotated with `@PreDestroy`, if there is one. The bean instance is ready for garbage collection.

Example 9 shows a stateless session bean with the two callback methods.

Example 9   Stateless session bean with two callback methods

```
@Stateless
public class MyStatelessBean implements MyBusinessLogic {
// .. bean business method

    @PostConstruct
    public void initialize() {
        // initialize the resources uses by the bean
    }

    @PreDestroy
    public void cleanup() {
        // deallocates the resources uses by the bean
    }
}
```

All stateless and stateful session EJB go through these two phases.

In addition, stateful session beans go through the passivation and activation cycle. An instance of a stateful bean is bound to a specific client; therefore, it cannot be reused among various requests. The EJB container manages the amount of available physical resources, and might decide to deactivate, or *passivate*, the bean by moving it from memory to secondary storage.

With this more complex lifecycle, we have further callback methods specific to stateful session beans:

► The EJB container invokes the method annotated with `@PrePassivate`, immediately before passivating it.

► If a client invokes a business method on the bean when it is in the passive stage, the EJB container activates the bean by calling the method annotated with `@PostActivate` and then moves it to the ready stage.

At the end of the lifecycle, the client explicitly invokes a method annotated with `@Remove`, and the EJB container calls the callback method annotated `@PreDestroy`. Developers can explicitly invoke only the lifecycle method annotated with `@Remove`. The other methods are invoked automatically by the EJB container.

Stateful session beans: Because a stateful bean is bound to a particular client, it is a leading practice to correctly design stateful session beans to minimize their footprints inside the EJB container. Also, it is a leading practice to correctly deallocate it at the end of its lifecycle by invoking the method annotated with `@Remove`.

Stateful session beans have a *timeout* value. If the stateful session bean is not used in the timeout period, it is marked inactive and is eligible for automatic deletion by the EJB container. It is still a leading practice for applications to remove the bean when the client is finished with it, rather than relying on the timeout mechanism.

Leading practices for developing session EJB

As a leading practice, EJB 3.*x* developers follow these guidelines:

► Each session bean must be a POJO, the class must be concrete, and it must have a no-argument constructor. If the no-argument constructor is not present, the compiler inserts a default constructor.

► If the business interface is annotated as @Remote, all the values passed through the interface must implement java.io.Serializable. Typically, the declared parameters are defined as serializable, but this definition is not required if the actual values passed are serializable.

► A session EJB can subclass a POJO, but cannot subclass another session EJB.

Message-driven EJB

MDBs are used for the processing of asynchronous Java Message Service (JMS) messages within JEE-based applications. MDBs are invoked by the container on the arrival of a message.

In this way, MDBs can be thought of as another interaction mechanism for invoking EJB. However, unlike session beans, the container is responsible for invoking them when a message is received, not a client or another bean.

To define an MDB in EJB 3.*x*, you must declare a POJO with the @MessageDriven annotation, as shown in Example 10.

Example 10   Declaring a POJO to define an MDB in EJB

```
@MessageDriven(activationConfig = {
        @ActivationConfigProperty(propertyName="destinationType",
                                propertyValue="javax.jms.Queue"),
        @ActivationConfigProperty(propertyName="destination",
                                propertyValue="queue/myQueue")
})
public class MyMessageBean implements javax.jms.MessageListener {

    public void onMessage(javax.msg.Message inMsg) {
        //implement the onMessage method to handle the incoming message
        ....
    }
}
```

Note the following features of Example 10:

► In EJB 3.*x*, the MDB class is annotated with the @MessageDriven annotation, which specifies a set of activation configuration parameters. These parameters are unique to the particular Java EE Connector Architecture (JCA) 1.5 adapter that is used to drive the MDB. Certain adapters have configuration parameters with which you can specify the destination queue of the MDB. If not, the destination name must be specified by using a *<message-destination>* entry in the XML binding file.

► The bean class implements the javax.jms.MessageListener interface, which defines only one method, onMessage. When a message arrives in the queue monitored by this MDB, the container calls the onMessage method of the bean class and passes the incoming message as the parameter.

► Furthermore, the activationConfig property of the @MessageDriven annotation provides messaging system-specific configuration information.

EJB services and annotations

The use of annotations is important to define EJB services:

► Interceptors
► Dependency injection
► Asynchronous invocations
► EJB timer service
► Web services

We describe the definitions of these services, when using annotations, in this section. Additionally, we provide the description of using deployment descriptors and the description of the new features of Portable JNDI name and Embedded Container API in this section.

Interceptors

The EJB 3.*x* specification defines the ability to apply custom-made interceptors to the business methods of session and MDB beans. Interceptors take the form of methods annotated with the `@AroundInvoke` annotation, as shown in Example 11.

Example 11   Applying an interceptor

```
@Stateless
public class MySessionBean implements MyBusinessInterface {

   @Interceptors(LoggerInterceptor.class)
   public Customer getCustomer(String ssn) {
      ...
   }
    ...
}

public class LoggerInterceptor {
   @AroundInvoke
   public Object logMethodEntry(InvocationContext invocationContext)
                                              throws Exception {
      System.out.println("Entering method: "
                          + invocationContext.getMethod().getName());
      Object result = invocationContext.proceed();
      // could have more logic here
      return result;
   }
}
```

Note the following points for Example 11:

► The `@Interceptors` annotation is used to identify the session bean method where the interceptor is applied.

► The `LoggerInterceptor` interceptor class defines a method (`logMethodEntry`) annotated with `@AroundInvoke`.

► The `logMethodEntry` method contains the advisor logic, in this case, it logs the invoked method name, and invokes the `proceed` method on the `InvocationContext` interface to advise the container to proceed with the execution of the business method.

The implementation of the interceptor in EJB 3.*x* differs from the analogous implementation of the aspect-oriented programming (AOP) paradigm that you can find in frameworks, such as

Spring or AspectJ, because EJB 3.*x* does not support *before* or *after* advisors, only *around* interceptors.

However, *around* interceptors can act as *before* interceptors, *after* interceptors, or both. Interceptor code before the `invocationContext.proceed` call is run before the EJB method, and interceptor code after that call is run after the EJB method.

A common use of interceptors is to provide preliminary checks, such as validation and security, before the invocation of business logic tasks, and therefore, they can throw exceptions. Because the interceptor is called together with the session bean code at run time, these potential exceptions are sent directly to the invoking client.

In Example 11 on page 12, we see an interceptor applied on a specific method. Alternatively, the `@Interceptors` annotation can be applied at the class level. In this case, the interceptor is called for every method.

Furthermore, the `@Interceptors` annotation accepts a list of classes, so that multiple interceptors can be applied to the same object.

> **Default interceptor:** To give further flexibility, EJB 3.*x* introduces the concept of a default interceptor that can be applied on every session or MDB contained inside the same EJB module. A default interceptor cannot be specified by using an annotation. Instead, define it inside the deployment descriptor of the EJB module.
>
> Interceptors run in the following execution order:
> - ► Default interceptor
> - ► Class interceptors
> - ► Method interceptors

To disable the invocation of a default interceptor or a class interceptor on a specific method, you can use the `@ExcludeDefaultInterceptors` and `@ExcludeClassInterceptors` annotations, respectively.

Dependency injection

The new specification introduces a powerful mechanism for obtaining Java EE resources, such as Java Database Connectivity (JDBC) data source, JMS factories and queues, and EJB references to inject them into EJB, entities, or EJB clients.

The EJB 3.*x* specification adopts a *dependency injection* (DI) pattern, which is one of the best ways to implement loosely coupled applications. It is much easier to use and more elegant than older approaches, such as dependency lookup through Java Naming and Directory Interface (JNDI) or container callbacks.

The implementation of dependency injection in the EJB 3.*x* specification is based on annotations or XML descriptor entries, with which you can inject dependencies on fields or setter methods.

Instead of complicated XML EJB references or resource references, you can use the `@EJB` and `@Resource` annotations to set the value of a field or to call a setter method within your beans with anything registered within JNDI. With these annotations, you can inject EJB references and resource references, such as data sources and JMS factories.

In this section, we show the most common uses of dependency injection in EJB 3.*x*, such as the `@EJB` annotation and `@Resource` annotation.

@EJB annotation

The `@EJB` annotation is used for injecting session beans into a client. This injection is only possible within managed environments, such as another EJB, or a servlet. We cannot inject an EJB into a JavaServer Faces (JSF)-managed bean or Struts action.

The `@EJB` annotation has the following optional parameters:

name   Specifies the JNDI name that is used to bind the injected EJB in the environment naming context (`java:comp/env`).

beanInterface   Specifies the business interface to be used to access the EJB. By default, the business interface to be used is taken from the Java type of the field into which the EJB is injected. However, if the field is a supertype of the business interface, or if method-based injection is used rather than field-based injection, the `beanInterface` parameter is typically required. This parameter is required because the specific interface type to be used might be ambiguous without the additional information provided by this parameter.

beanName   Specifies a *hint* to the system of the `ejb-name` of the target EJB that must be injected. It is analogous to the *<ejb-link>* stanza that can be added to an *<ejb-ref>* or *<ejb-local-ref>* stanza in the XML descriptor.

Example 12 shows the code to access a session bean from a Java servlet.

Example 12   Injecting an EJB reference inside a servlet

```
import javax.ejb.EJB;
public class TestServlet extends javax.servlet.http.HttpServlet
                       implements javax.servlet.Servlet {

   // inject the remote business interface
   @EJB(beanInterface=MyRemoteBusinessInterface.class)
   MyAbstractBusinessInterface serviceProvider;

   protected void doGet(HttpServletRequest request,
                    HttpServletResponse response)
                       throws ServletException, IOException {
      // call ejb method
      serviceProvider.myBusinessMethod();
      ......
   }
}
```

Note the following points about Example 12:

► We specified the `beanInterface` attribute, because the EJB shows two business interfaces (`MyRemoteBusinessInterface` and `MyLocalBusinessInterface`).

► If the EJB shows only one interface, you are not required to specify this attribute. However, it can be useful to make the client code more readable.

Special notes for stateful EJB injection:

► Because a servlet is a multi-thread object, you cannot use dependency injection, but you must explicitly look up the EJB through the JNDI.

► You can safely inject a stateful EJB inside another session EJB (stateless or stateful), because a session EJB instance is executed by only a single thread at a time.

@Resource annotation

The @Resource annotation is the major annotation that can be used to inject resources in a managed component. Therefore, two techniques exist: the field technique and the setter injection technique. In the following section, we show the most commonly used scenarios of this annotation.

Example 13 shows how to inject a typical resource, such as a data source inside a session bean by using the field injection technique. A data source (jdbc/datasource) is injected inside a property that is used in a business method.

Example 13   Field injection technique for a data source

```
@Stateless
public class CustomerSessionBean implements CustomerServiceInterface {

    @Resource (name="jdbc/dataSource")
    private DataSource ds;
    public void businessMethod1() {
        java.sql.Connection c=null;
        try {
            c = ds.getConnection();
            // .. use the connection
        } catch (java.sql.SQLException e) {
            // ... manage the exception
        } finally {
            // close the connection
            if(c!=null) {
                try { c.close(); } catch (SQLException e) { }
            }
        }
    }
}
```

The @Resource annotation has the following optional parameters:

name   Specifies the component-specific internal name, which is the resource reference name, within the java:comp/env namespace. It does not specify the global JNDI name of the resource that is injected.

type   Specifies the resource manager connection factory type.

authenticationType   Specifies whether the container or the bean is to performs the authentication.

shareable   Specifies whether resource connections are shareable.

mappedName   Specifies a product-specific name to which the resource must be mapped. WebSphere does not use mappedName.

description   Description.

Another technique is to inject a setter method. The setter injection technique is based on JavaBean property naming conventions, as shown in Example 14 on page 16.

Example 14   Setter injection technique for a data source

```
@Stateless
public class CustomerSessionBean implements CustomerServiceInterface {

    private Datasource ds;

    @Resource (name="jdbc/dataSource")
    public void setDatasource(DataSource datasource) {
       this.ds = datasource;
    }
    ...
    public void businessMethod1() {
       ...
    }
}
```

Note the following points about Example 13 on page 15 and Example 14:

► We directly used the data source inside the session bean, which is not a good practice. Instead, place the JDBC code in specific components, such as data access objects.

► Use the setter injection technique, which gives more flexibility:

  – You can put initialization code inside the setter method.
  – The session bean is set up to be easily tested as a stand-alone component.

In addition, note the following use of the @Resource annotation:

► To obtain a reference to the EJB session context, as shown in Example 15.

Example 15   Resource reference to session context

```
@Stateless
public class CustomerSessionBean implements CustomerServiceInterface {
    ....
    @Resource javax.ejb.SessionContext ctx;
}
```

► To obtain the value of an environment variable, which is configured inside the deployment descriptor with env-entry, as shown in Example 16.

Example 16   Resource reference to environment variable

```
@Stateless
public class CustomerSessionBean implements CustomerServiceInterface {
    ....
    @Resource String myEnvironmentVariable;
}
```

For detailed information, see section 4.3.2 "Dependency Injection" in *JSR 318: Enterprise JavaBeans 3.1.*

Asynchronous invocations

All session bean invocations, regardless of which view, the Remote, Local, or the no-interface views, are synchronous by default. As a new feature of the EJB 3.1 specification, you can define a bean class or a method as asynchronous by using the annotation @Asynchronous. This approach to define a method as asynchronous avoids the behavior that one request blocks for the duration of the invocation until the process is completed. In case a request

invokes an asynchronous method, the container returns control back to the client immediately and continues processing the invocation on another thread. Therefore, the asynchronous method returns either `void` or `Future<T>`.

For detailed information, see section 3.4.8 "Asynchronous Invocations" in *JSR 318: Enterprise JavaBeans 3.1.*

EJB timer service

The EJB timer service is a container-managed service for scheduled callbacks. The definition of time-based events with the timer service is a new feature of EJB 3.1. Therefore, the method `getTimerService` exists, which returns the `javax.ejb.TimerService` interface. This method can be used by stateless and Singleton session beans. Stateful session beans cannot be timed objects. Time-based events can be calendar-based-scheduled, at a specific time, after a specific past duration, or for specific circular intervals.

To define timers to be created automatically by the container, use the `@Schedule` and `@Schedules` annotations. Example 17 shows how to define a timer method for every second of every minute of every hour of every day.

Example 17   Timer service definition

```
@Schedule(dayOfWeek="*",hour="*",minute="*",second="*")
public void calledEverySecond(){
    System.out.println("Called every second");
}
```

The definition of the attributes of the `@Schedule` annotation can be modified as well by using the Attributes view in Rational Application Developer, as shown in Figure 5.
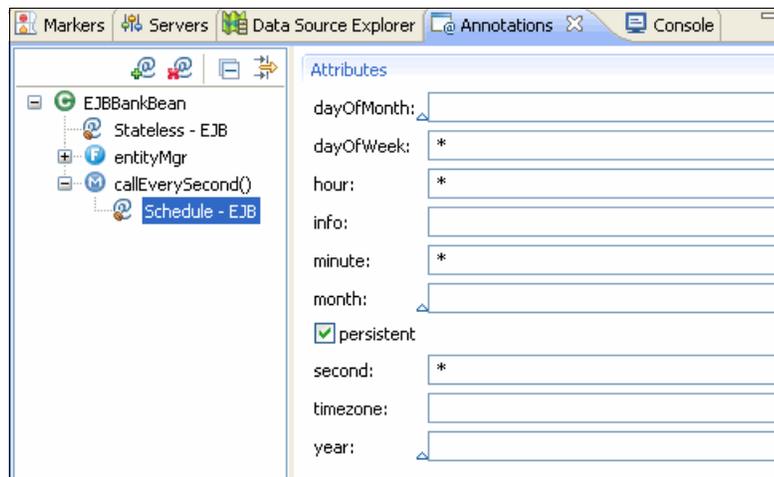

Figure 5   Annotation @Schedule attribute view

For detailed information about the EJB timer service, see Chapter 18, "Timer Service", in *JSR 318: Enterprise JavaBeans 3.1.*

Web services

For detailed information about how to show EJB 3.1 beans as web services by using the `@WebService` annotation, see *Developing Web Services Applications*, REDP-4884, which shows how to implement a web service from an EJB 3.1 session bean.

Portable JNDI name

As a new feature defined in the Java EE 6 specification, a standardized global JNDI namespace is defined. These portable JNDI names are defined with the following syntax:

```
java:global[/<app-name>]/<module-name>/<bean-name>[!<fully-qualified-interface-name>]
```

The parameters consist of the following content:

<app-name>	Name of the application. Because a session bean is packaged within an EAR file, this field is an optional value. It defaults to the name of the EAR file, just without the `.ear` file extension.
<module-name>	Name of the module in which the session bean is packaged. The value of the name defaults to the base name of the archive without the file extension. This archive can be a stand-alone JAR file or a WAR file.
<bean-name>	Name of the session bean.
<fully-qualified-interface-name>	Fully qualified name of each defined business interface. Because a session bean can have no interface, which means it has only a no-interface view, this field is an optional value.

For detailed information, see section 4.4 "Global JNDI Access" in *JSR 318: Enterprise JavaBeans 3.1.*

Embedded Container API

Defining an embedded container is a new feature in EJB 3.1. An embedded container provides the same managed environment as the Java EE runtime container. The services for injection, access to a component environment, and container-managed transactions (CMTs) are provided as well. This container is used to execute EJB components within a Java SE environment. Example 18 shows how to create an instance of an embeddable container, as a first step.

Example 18   Defining embedded container

```
EJBContainer ec = EJBContainer.createEJBContainer();
Context ctx = ec.getContext();
EJBBank bank = (EJBBank) ctx.lookup("java:global/EJBExample/EJBBank");
```

In the second step in Example 18, we get a JNDI context. In the third step, we use the lookup method to retrieve an EJB, in this case, the bean `EJBBank`.

For detailed information about the Embedded Container API, see Chapter 22.2.1, "EJBContainer", in *JSR 318: Enterprise JavaBeans 3.1.*

> **Important:** The `\4885code\ejb\antScriptEJB.zip` directory includes an Ant script that you use to create a `.jar` file for your EJB project. Update the `build.properties` file with your settings before by using the `build.xml` file. This script includes the configuration for the `RAD8EJB` project as an example.

Using deployment descriptors

In the previous sections, we saw how to define an EJB, how to inject resources into it, and how to specify annotations. We can get the same result by specifying a deployment descriptor (`ejb-jar.xml`) with the necessary information in the EJB module.

EJB 3.1 application packaging

Session and message-driven beans are packaged in Java standard JAR files. We map from the enterprise archive (EAR) project to the EJB project that contains the beans. To complete this mapping, we use the Deployment Assembly properties sheet, which replaces the J2EE Module Dependencies properties sheet used in previous versions of Rational Application Developer. The integrated development environment (IDE) automatically updates the `application.xml` file, if one exists.

However, in EJB 3.*x*, you are not required to define the EJB and related resources in an `ejb-jar.xml` file, because they are defined through the use of annotations. The major use of the deployment descriptor files is to override or complete behavior that is specified by annotations.

EJB 3.1 offers the capability to package and deploy EJB components directly in a WAR file as a new feature for the packaging approach.

EJB 3.1 Lite

Because the full EJB 3.*x* API consists of a large set of features with the support for implementing business logic in a wide variety of enterprise applications, EJB 3.1 Lite was defined to provide a minimal subset of the full EJB 3.1 API. This new defined runtime environment offers a selection of EJB features, as shown in Table 1.

Table 1   Overview comparison of EJB 3.1 Lite and full EJB 3.1

Feature	EJB 3.1 Lite	Full EJB 3.1
Session beans (stateless, stateful, and Singleton)	Yes	Yes
MDB	No	Yes
Entity beans 1.*x*/2.*x*	No	Yes
No-interface view	Yes	Yes
Local interface	Yes	Yes
Remote interface	No	Yes
2.*x* interfaces	No	Yes
Web services (JAX-WS, JAX-RS, and JAX-RPC)	No	Yes[a]
Timer service	No	Yes
Asynchronous calls	No	Yes
Interceptors	Yes	Yes
RMI/IIOP interoperability	No	Yes
Transaction support	Yes	Yes
Security	Yes	Yes

Feature	EJB 3.1 Lite	Full EJB 3.1
Embeddable API	Yes	Yes

a. Pruning candidates for future versions of the EJB specification

For detailed information, see Section 21.1, "EJB 3.1 Lite", in *JSR 318: Enterprise JavaBeans 3.1.*

EJB 3.1 features in Rational Application Developer

The following features are supported in Rational Application Developer:

► Singleton bean
► No interface-view for session beans
► Asynchronous invocations
► EJB timer service
► Portable JNDI name
► Embedded Container API
► War deployment, as mentioned in EJB 3.1 application packaging
► EJB 3.1 Lite

Developing an EJB module

The EJB module consists of a web module with a simple servlet, and an EJB module with an EJB 3.1 session bean that uses the JPA entities of the RAD8JPA project to access the database. This section describes the steps for developing the sample EJB module.

To develop EJB applications, you must enable the EJB development capability in Rational Application Developer (the capability might already be enabled):

1. Select **Window** → **Preferences**.
2. Select **General** → **Capabilities** → **Enterprise Java Developer** and click **OK**.

An EJB module, with underlying JPA entities, typically contains components that work together to perform business logic. This logic can be self-contained or access external data and functions as needed. It needs to consist of a facade (session bean) and the business entities (JPA entities). The facade is implemented by using one or more session beans and MDBs.

In this paper, we develop a session EJB as a facade for the JPA entities (Customer, Account, and Transaction), as shown in Figure 6 on page 21. The RAD8JPA project is available in your workspace. You can import the project from the \4885codesolution\jpa directory.

Furthermore, we assume that an instance of the WebSphere Application Server V8.0 is configured and available in your workspace.

Sample code: The sample code described in this paper can be completed by following the documented procedures. Alternatively, you can import the sample EJB project and corresponding JPA project provided in the \4885\codesolution\ejb\RAD8EJB.zip directory.

Sample application overview
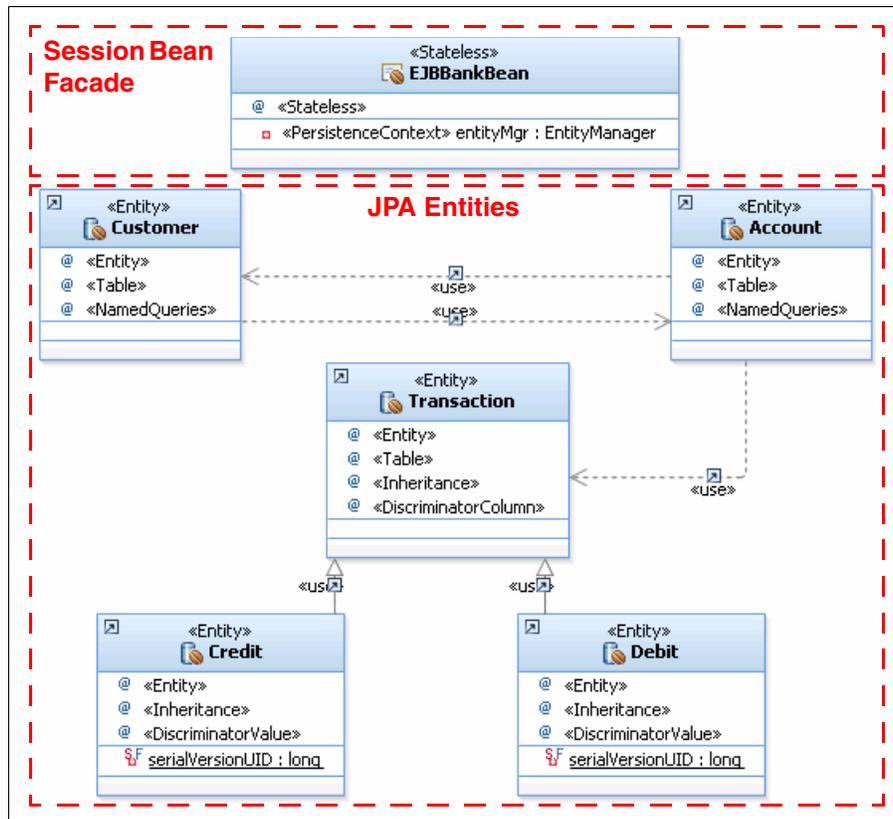
Figure 6 shows the sample application model layer design.

Figure 6   EJB module class diagram for the sample application

The `EJBBankBean` session bean acts as a facade for the EJB model. The business entities (`Customer`, `Account`, `Transaction`, `Credit`, and `Debit`) are implemented as JPA entity beans, as opposed to regular JavaBeans. By doing so, we automatically gain persistence, security, distribution, and transaction management services. The implication here is that the control and view layers are not able to reference these entities directly, because they can be placed in a separate Java virtual machine (JVM). Only the session bean `EJBBankBean` can access the business entities.

Figure 7 shows the application component model and the flow of events.
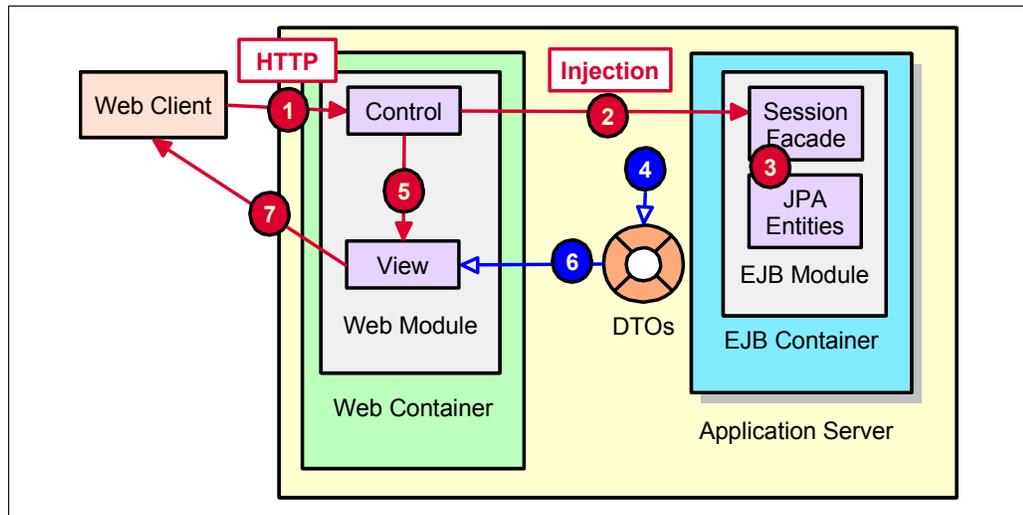
Figure 7   Application component model and workflow

Figure 7 shows the following flow of events:

1. The HTTP request is issued by the web client to the server. This request is answered by a servlet in the control layer, also known as the *front controller*, which extracts the parameters from the request. The servlet sends the request to the appropriate control JavaBean. This bean verifies whether the request is valid in the current user and application states.

2. If the request is valid, the control layer sends the request through the `@EJB` injected interface to the session EJB facade, which involves the use of JNDI to locate the interface of the session bean and creating an instance of the bean.

3. The session EJB executes the appropriate business logic related to the request and accesses JPA entities in the model layer.

4. The facade returns data transfer objects (DTOs) to the calling controller servlet with the response data. The DTO returned can be a JPA entity, a collection of JPA entities, or any Java object. In general, it is not necessary to create extra DTOs for entity data.

5. The front controller servlet sets the response DTO as a request attribute and forwards the request to the appropriate JSP in the view layer, which is responsible for rendering the response back to the client.

6. The view JSP accesses the response DTO to build the user response.

7. The result view, possibly in HTML, is returned to the client.

Creating an EJB project

To develop the session EJB, we create an EJB project. It is also typical to create an EAR project that is the container for deploying the EJB project.

To create the EJB project, perform the following steps:

1. In the Java EE perspective, within the Enterprise Explorer view, right-click and select **New → Project**.

2. In the New Project wizard, select **EJB → EJB Project** and click **Next**.

3. In the New EJB Project window, shown in Figure 8, define the project details:

    a. In the Name field, type RAD8EJB.

    b. For Target Runtime, select **WebSphere Application Server v8.0 Beta**.

    c. For EJB module version, select **3.1**.

    d. For Configuration, select **Minimal Configuration**. Optional: Click **Modify** to see the project facets (EJB Module 3.1, Java 6.0, and WebSphere EJB (Extended) 8.0).

    e. Select **Add project to an EAR** (default), and in the EAR Project Name field, type RAD8EJBEAR. By default, the wizard creates an EAR project, but you can also select an existing project from the list of options for the EAR Project Name field. If you want to create a project and configure its location, click **New**. For our example, we use the default value.

    f. Click **Next**.
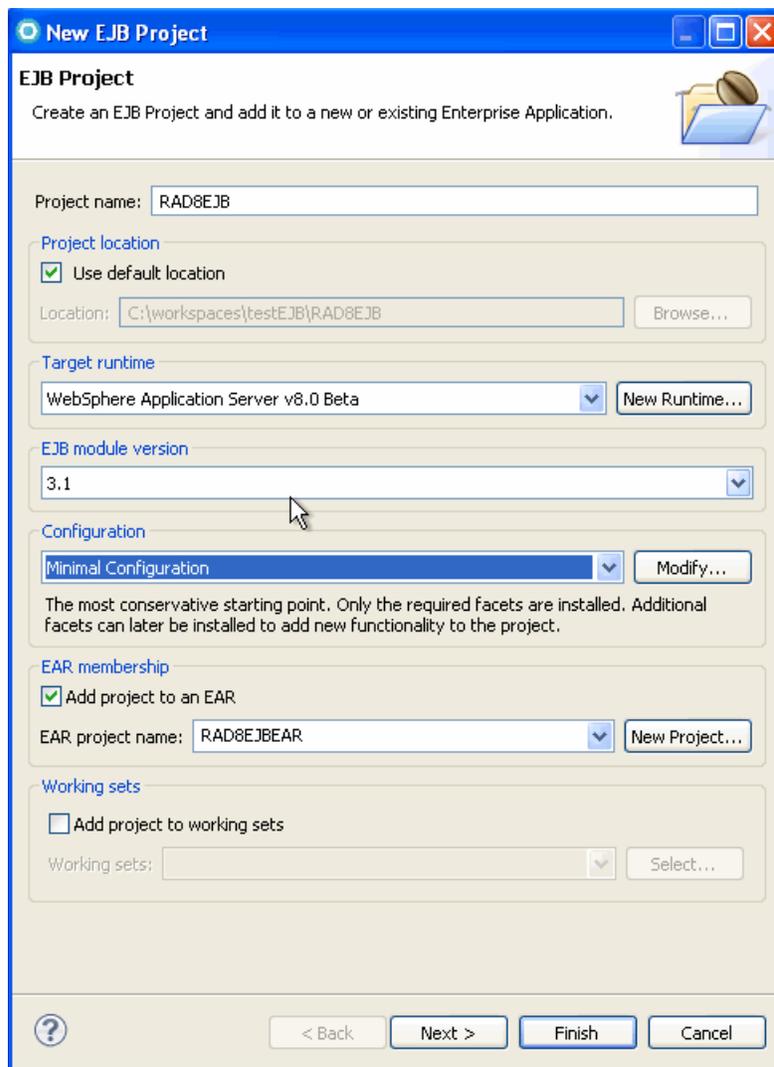
Figure 8   Creating an EJB project: EJB Project window

4. In the New EJB Project wizard, in the Java window, accept the default value ejbModule for the Source folder.

5. In the New EJB Project wizard, in the EJB Module window, perform the following steps, as shown in Figure 9:

   a. Clear **Create an EJB Client JAR module to hold client interfaces and classes** (default).

   b. Select **Generate ejb-jar.xml deployment descriptor** and click **Finish**.
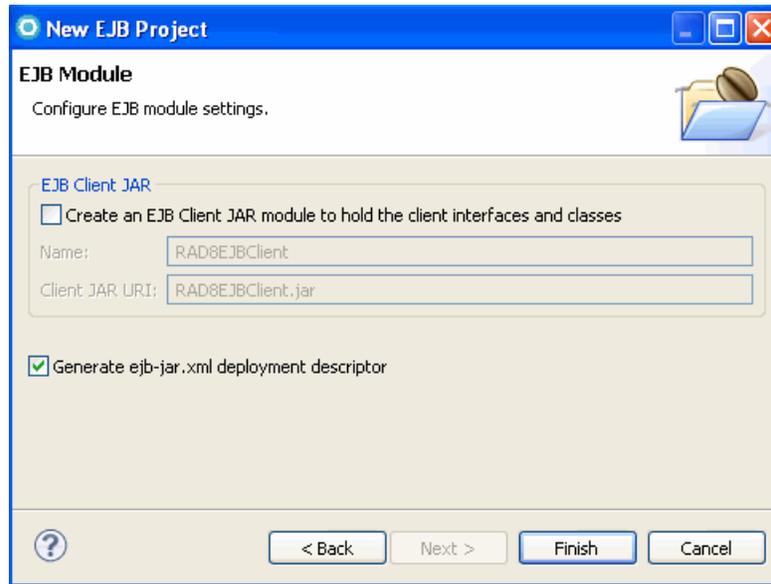
Figure 9   Creating an EJB project - EJB Module window

> **EJB client JAR file:** The EJB client JAR file holds the interfaces of the enterprise beans and other classes on which these interfaces depend. For example, it holds their superclasses and implemented interfaces, the classes and interfaces used as method parameters, results, and exceptions. The EJB client JAR can be deployed together with a client application that accesses the EJB. This results in a smaller client application compared to deploying the EJB project with the client application.

6. If the current perspective is not the Java EE perspective when you create the project, when Rational Application Developer prompts you to switch to the Java EE perspective, click **Yes**.

7. The Technology Quickstarts view opens. Close the view.

The Enterprise Explorer view contains the `RAD8EJB` project and the `RAD8EJBEAR` enterprise application. Rational Application Developer indicates that at least one EJB bean is defined within the `RAD8EJB` project. We create this session bean in "Implementing the session facade" on page 29, when we enable the JPA project.

Making the JPA entities available to the EJB project

We assume that you imported the JPA project RAD8JPA into your workspace. To make the JPA entities available to the EJB, add the RAD8JPA project to the RAD8EJBEAR enterprise application and create a dependency, while performing the following steps:

1. Right-click the **RAD8EJBEAR** project and select **Properties**.

2. In the Properties window, select **Deployment Assembly**, and for EAR Module Assembly, click **Add**.

3. In the New Assembly directive wizard, in the Select Directive Type window, select **Project**. Click **Next**.

4. In the New Assembly directive wizard, in the Project window, select **RAD8JPA**. Click **Finish**.

The Properties window now looks like Figure 10.
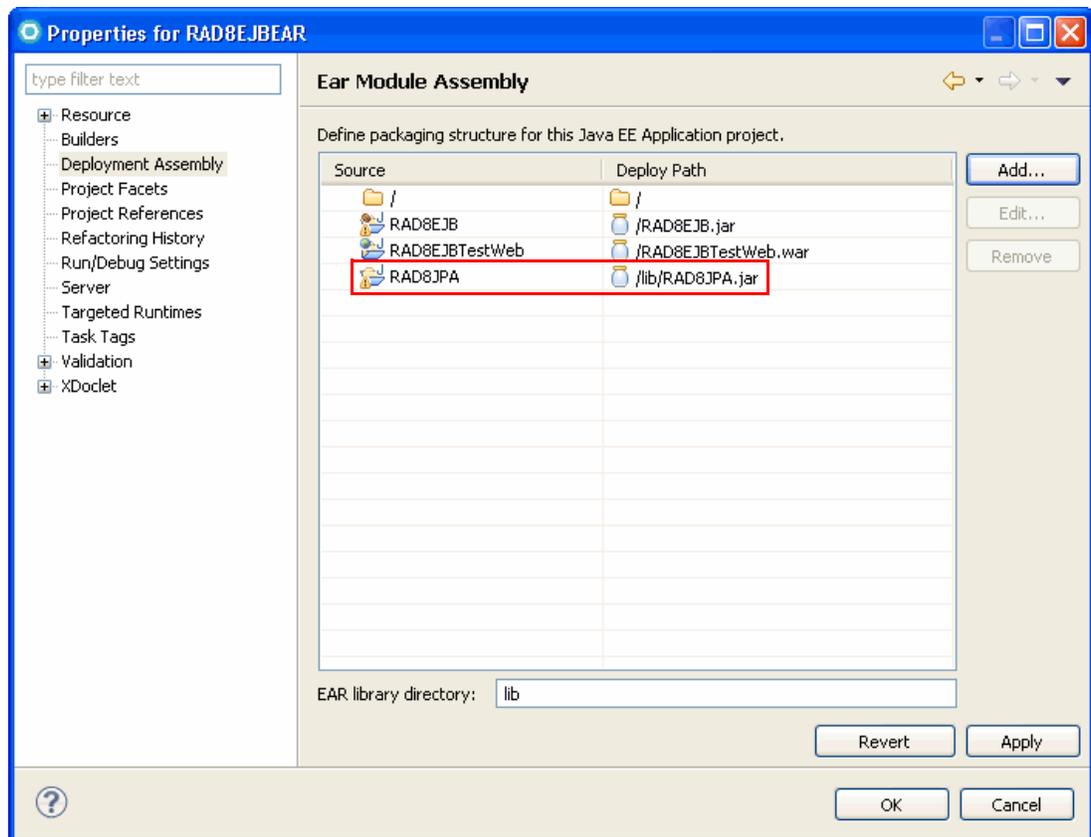
Figure 10   Selecting the RAD8JPA project

Setting up the ITSOBANK database

The JPA entities are based on the ITSOBANK database. Therefore, we must define a database connection within Rational Application Developer that the mapping tools use to extract schema information from the database.

We provide two implementations of the ITSOBANK database: Derby and IBM DB2® Universal Database. You can choose to implement either or both databases and then set up

the enterprise applications to use one of the databases. The Derby database system ships with WebSphere Application Server.

► Derby

The `\4885code\database\derby` directory provides command files to define and load the ITSOBANK database in Derby. For the `DerbyCreate.bat`, `DerbyLoad.bat`, and `DerbyList.bat` files, you must install WebSphere Application Server in the `C:\IBM\WebSphere\AppServer` folder. You must edit these files to point to your WebSphere Application Server installation directory if you installed the product in a separate folder.

In the `\4885code\database\derby` directory, you can perform the following actions:

– Execute the `DerbyCreate.bat` file to create the database and table.
– Execute the `DerbyLoad.bat` file to delete the existing data and add records.
– Execute the `DerbyList.bat` file to list the contents of the database.

These command files use the SQL statements and helper files that are provided in the following files:

– `itsobank.ddl`: Database and table definition
– `itsobank.sql`: SQL statements to load sample data
– `itsobanklist.sql`: SQL statement to list the sample data
– `tables.bat`: Command file to execute `itsobank.ddl` statements
– `load.bat`: Command file to execute `itsobank.sql` statements
– `list.bat`: Command file to execute `itsobanklist.sql` statements

The Derby ITSOBANK database is created in the `\4885code\database\derby\ITSOBANK` directory.

► DB2

The `\4885code\database\db2` folder provides the DB2 command files to define and load the ITSOBANK database. You can perform the following actions:

– Execute the `createbank.bat` file to define the database and table.
– Execute the `loadbank.bat` file to delete the existing data and add records.
– Execute the `listbank.bat` file to list the contents of the database.

These command files use the SQL statements that are provided in the following files:

– `itsobank.ddl`: Database and table definition
– `itsobank.sql`: SQL statements to load sample data
– `itsobanklist.sql`: SQL statement to list the sample data

Configuring the data source for the ITSOBANK

You can choose from multiple methods to configure the data source, including the use of the WebSphere administrative console or by using the WebSphere enhanced EAR, which stores the configuration in the deployment descriptor and is deployed with the application.

In this section, we explain how to configure the data source by using the WebSphere enhanced EAR capabilities. The enhanced EAR is configured in the Deployment tab of the EAR Deployment Descriptor editor. If you select to import the complete sample code, you must verify only that the value of the `databaseName` property in the deployment descriptor matches the location of the database.

Configuring the data source by using the enhanced EAR

Before you perform the following steps, you must start the server. To configure a new data source by using the enhanced EAR capability in the deployment descriptor, follow these steps:

1. Right-click the **RAD8EJBEAR** project. Select **Java EE** → **Open WebSphere Application Server Deployment**.

2. In the WebSphere Deployment editor, select **Derby JDBC Provider (XA)** from the JDBC provider list. This JDBC provider is configured by default.

3. Click **Add** next to data source.

4. Under the JDBC provider, select **Derby JDBC Provider (XA)** and **Version 5.0 data source**. Click **Next**.

5. In the Create a Data Source window, which is shown in Figure 11, define the following details:

   a. For Name, type `ITSOBANKejb`.
   b. For JNDI name, type `jdbc/itsobank`.
   c. For Description, type `Data Source for ITSOBANK EJBs`.
   d. Clear **Use this data source in container managed persistence (CMP)**.

Figure 11   Data source definition: Name

   e. Click **Next**.

6. In the Create Resource Properties window, select **databaseName** and enter the value `\4885code\database\derby\ITSOBANK`, which is the path where your installed database is located. Clear the description, as shown in Figure 12 on page 29.
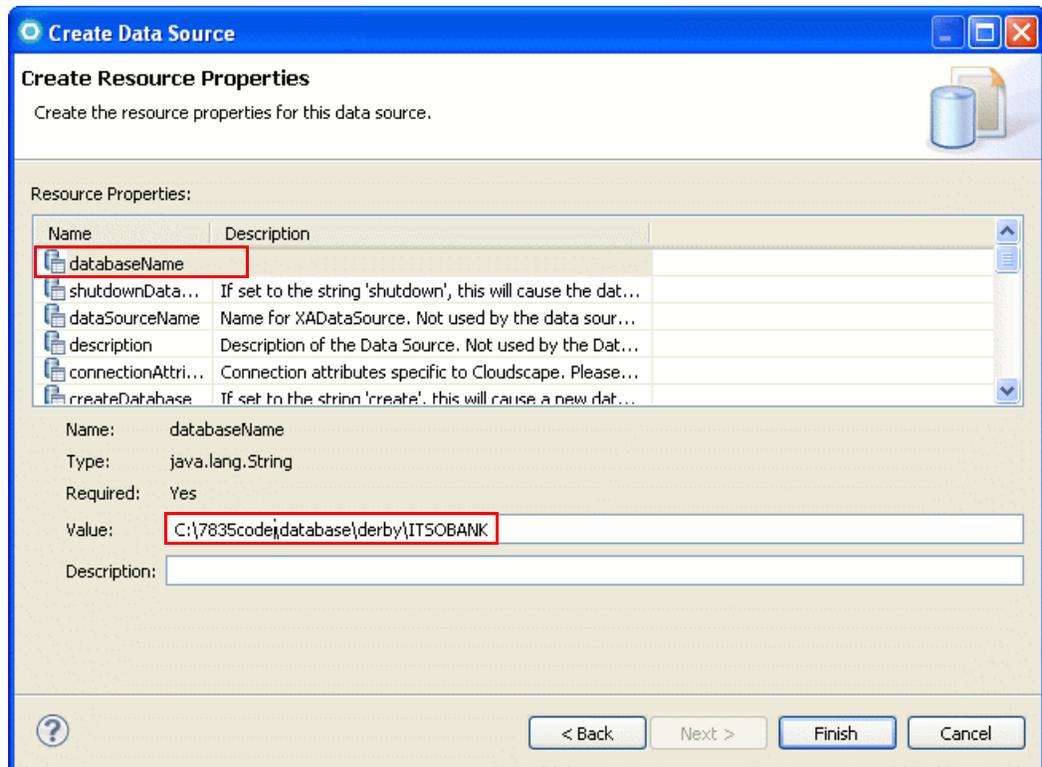
Figure 12   Data source definition: Database definition

7. Click **Finish**.

8. Save and close the deployment descriptor.

Implementing the session facade

The front-end application communicates with the JPA entity model through a session facade. This design pattern makes the entities invisible to the EJB client.

In this section, we build the session facade with the session bean `EJBBankBean`. Therefore, we describe all necessary steps to implement the session facade and add the facade methods that are used by clients to perform banking operations:

► Preparing an exception
► Creating the EJBBankBean session bean
► Defining the business interface
► Creating an Entity Manager
► Generating skeleton methods
► Completing the methods in EJBBankBean
► Deploying the application to the server

Preparing an exception

The business logic of the session bean throws an exception when errors occur. Create an application exception named `ITSOBankException`, when performing the following steps:

1. Right-click the **RAD8EJB** project and select **New** → **Class**.

2. In the New Java Class window, define the following details:

   a. For Package, type `itso.bank.exception`.
   b. For Name, type `ITSOBankException`.
   c. Set Superclass to **java.lang.Exception**.

3. Click **Finish**.

4. Complete the code in the editor, as shown in Example 19.

Example 19   Class definition ITSOBankException

```
public class ITSOBankException extends Exception {
   private static final long serialVersionUID = 1L;

   public ITSOBankException(String message) {
      super(message);
   }
}
```

5. Save and close the class.

Creating the EJBBankBean session bean

To create the session bean `EJBBankBean`, follow these steps:

1. Right-click the **RAD8EJB** project and select **New** → **Session Bean**.

2. In the Create EJB 3.x Session Bean window, as shown in Figure 13 on page 31, define the following details:

   a. For Java package, type `itso.bank.session`.

   b. For Class name, type `EJBBankBean`.

   c. For State type, select **Stateless**.

   d. For Create business interface, select **Local** and set the name to **itso.bank.service.EJBBankService**.
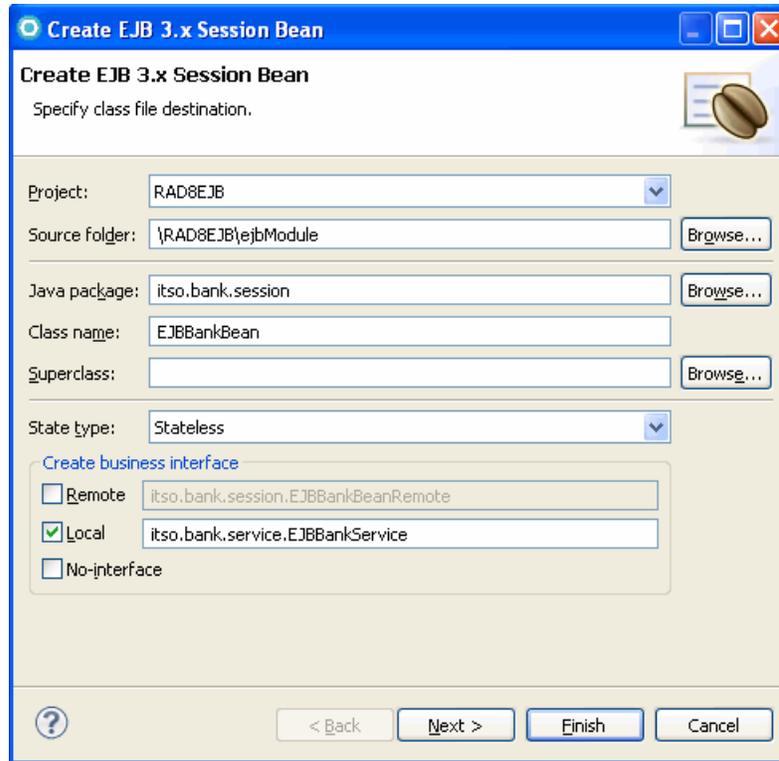
   e. Click **Next**.

Figure 13   Creating a session bean (part 1 of 2)

3. In the next window, which is shown in Figure 14, accept the default value **Container** for Transaction type and click **Next**.
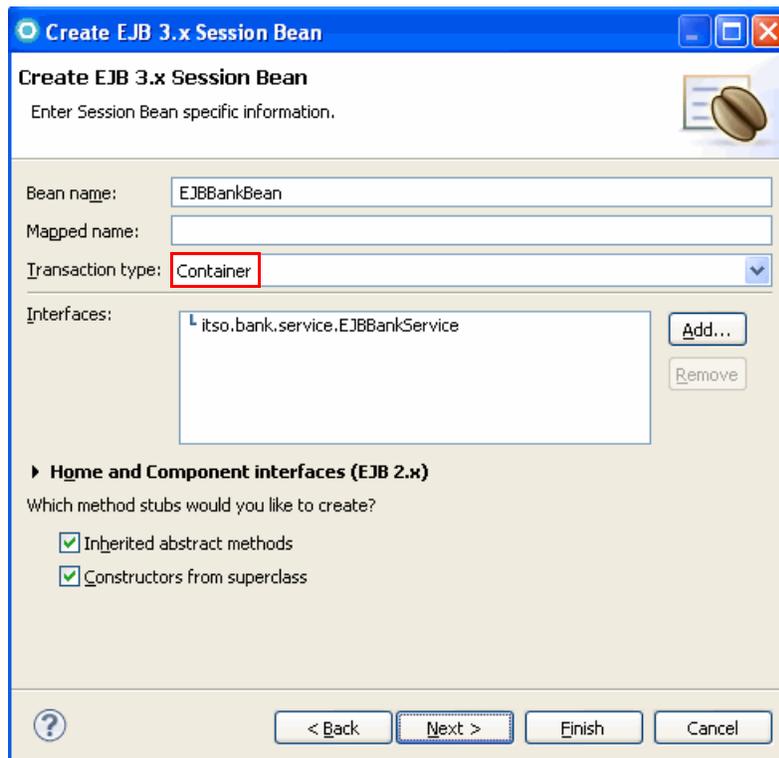

Figure 14   Creating a session bean (part 2 of 2)

4. In the Select Class Diagram for Visualization window, select **Add bean to Class Diagram** and accept the default name of **classdiagram.dnx**.

5. Click **Finish**.

6. When prompted for the enablement of EJB 3.1 Modeling, click **OK**.

7. Save and close the class diagram.

The `EJBBankBean` is open in the editor. Notice the `@Stateless` annotation.

Before you can write the session bean code, complete the business interface, `EJBBankService`.

Defining the business interface

EJB 3.1 also provides a business interface mechanism, which is the interface that clients use to access the session bean. The session bean can implement multiple interfaces, for example, a local interface and a remote interface. For now, we keep it simple with one local interface, `EJBBankService`.

The session bean wizard created the `EJBBankService` interface. To complete the code, follow these steps:

1. Open the **EJBBankService** interface. Notice the `@Local` annotation.

2. In the Java editor, add the methods to the interface, as shown in Example 20. The code is available in the `\4885code\ejb\source\EJBBankService.txt` file.

Example 20   Business interface of the session bean

```
@Local
public interface EJBBankService {

   public Customer getCustomer(String ssn) throws ITSOBankException;
   public Customer[] getCustomersAll();
   public Customer[] getCustomers(String partialName) throws ITSOBankException;
   public void updateCustomer(String ssn, String title, String firstName, String lastName)
throws ITSOBankException;
   public Account[] getAccounts(String ssn) throws ITSOBankException;
   public Account getAccount(String id) throws ITSOBankException;
   public Transaction[] getTransactions(String accountID) throws ITSOBankException;
   public void deposit(String id, BigDecimal amount) throws ITSOBankException;
   public void withdraw(String id, BigDecimal amount) throws ITSOBankException;
   public void transfer(String idDebit, String idCredit, BigDecimal amount) throws
ITSOBankException;
   public void closeAccount(String ssn, String id) throws ITSOBankException;
   public String openAccount(String ssn) throws ITSOBankException;
   public void addCustomer(Customer customer) throws ITSOBankException;
   public void deleteCustomer(String ssn) throws ITSOBankException;
}
```

3. To organize the imports, press Ctrl+Shift+O. When prompted, select **java.math.BigDecimal** and **itso.bank.entities.Transaction**. Save and close the interface.

Creating an Entity Manager

The session bean works with the JPA entities to access the `ITSOBANK` database. We require an Entity Manager that is bound to the persistence context. To create an Entity Manager, follow these steps:

1. Add these definitions to the `EJBBankBean` class:

```
@PersistenceContext (unitName="RAD8JPA",
                     type=PersistenceContextType.TRANSACTION)
private EntityManager entityMgr;
```

   The `@PersistenceContext` annotation defines the persistence context unit with transactional behavior. The unit name matches the name in the `persistence.xml` file in the `RAD8JPA` project:

```
<persistence-unit name="RAD8JPA">
```

   The `EntityManager` instance is used to execute JPA methods to retrieve, insert, update, delete, and query instances.

2. Organize the imports by selecting the **javax.persistence** package.

Generating skeleton methods

We can generate method skeletons for the methods of the business interface that must be implemented:

1. Open the **EJBBankBean** (if you closed it).

2. Select **Source** → **Override/Implement Methods**.

3. In the Override/Implement Methods window, select all the methods of the `EJBBankService` interface. For Insertion point, select **After 'EJBBankBean()'**. Click **OK**. The method skeletons are generated.

4. Delete the default constructor.

Completing the methods in EJBBankBean

Tip: You can copy the Java code for this section from the `\4885code\ejb\source\EJBBankBean.txt` file.

We complete the methods of the session bean in a logical sequence, not in the alphabetical sequence of the generated skeletons.

getCustomer method

The `getCustomer` method retrieves one customer by Social Security number (SSN). We use `entityMgr.find` to retrieve one instance. Alternatively, we might use the `getCustomerBySSN` query (code in comments). If no instance is found, null is returned, as shown in Example 21 on page 34.

Example 21   Session bean getCustomer method

```
public Customer getCustomer(String ssn) throws ITSOBankException {
   System.out.println("getCustomer: " + ssn);
   //Query query = null;
   try {
      //query = entityMgr.createNamedQuery("getCustomerBySSN");
      //query.setParameter("ssn", ssn);
      //return (Customer)query.getSingleResult();
      return entityMgr.find(Customer.class, ssn);
   } catch (Exception e) {
      System.out.println("Exception: " + e.getMessage());
      throw new ITSOBankException(ssn);
   }
}
```

getCustomers method

The getCustomers method uses a query to retrieve a collection of customers, as shown in Example 22. The query is created and executed. The result list is converted into an array and returned. Remember the defined query from the Customer entity:

```
@NamedQuery(name="getCustomersByPartialName",
         query="select c from Customer c where c.lastName like :name")
```

This query looks similar to SQL but works on entity objects. In our case, the entity name and the table name are the same, but they do not have to be identical.

Example 22   Session bean getCustomers method

```
public Customer[] getCustomers(String partialName) throws ITSOBankException {
   System.out.println("getCustomer: " + partialName);
   Query query = null;
   try {
      query = entityMgr.createNamedQuery("getCustomersByPartialName");
      query.setParameter("name", partialName);
      List<Customer> beanlist = query.getResultList();
      Customer[] array = new Customer[beanlist.size()];
      return beanlist.toArray(array);
   } catch (Exception e) {
      throw new ITSOBankException(partialName);
   }
}
```

The updateCustomer method

The updateCustomer method is simple, as shown in Example 23 on page 35. No call to the Entity Manager is necessary. The table is updated automatically when the method (transaction) ends.

Example 23   Session bean updateCustomer method

```
public void updateCustomer(String ssn, String title, String firstName,
                           String lastName) throws ITSOBankException {
   System.out.println("updateCustomer: " + ssn);
   Customer customer = getCustomer(ssn);
   customer.setTitle(title);
   customer.setLastName(lastName);
   customer.setFirstName(firstName);
   System.out.println("updateCustomer: " + customer.getTitle() + " "
               + customer.getFirstName() + " " + customer.getLastName());
}
```

The getAccount method

The getAccount method retrieves one account by key. It is similar to the getCustomer method.

The getAccounts method

The getAccounts method uses a query to retrieve all the accounts of a customer, as shown in Example 24. The Account entity has the following query:

```
select a from Account a, in(a.customers) c where c.ssn =:ssn
                                        order by a.id
```

This query looks for accounts that belong to a customer with an SSN. You can also use this alternate query in the Customer class:

```
select a from Customer c, in(c.accounts) a where c.ssn =:ssn
                                        order by a.id
```

Example 24   Session bean getAccounts method

```
public Account[] getAccounts(String ssn) throws ITSOBankException {
   System.out.println("getAccounts: " + ssn);
   Query query = null;
   try {
      query = entityMgr.createNamedQuery("getAccountsBySSN");
      query.setParameter("ssn", ssn);
      List<Account>accountList = query.getResultList();
      Account[] array = new Account[accountList.size()];
      return accountList.toArray(array);
   } catch (Exception e) {
      System.out.println("Exception: " + e.getMessage());
      throw new ITSOBankException(ssn);
   }
}
```

The getTransactions method

The getTransactions method retrieves the transactions of an account, as shown in Example 25 on page 36. It is similar to the getAccounts method.

Example 25   Session bean getTransactions method

```
public Transaction[] getTransactions(String accountID) throws ITSOBankException {
   System.out.println("getTransactions: " + accountID);
   Query query = null;
   try {
      query = entityMgr.createNamedQuery("getTransactionsByID");
      query.setParameter("aid", accountID);
      List<Transaction> transactionsList = query.getResultList();
      Transaction[] array = new Transaction[transactionsList.size()];
      return transactionsList.toArray(array);
   } catch (Exception e) {
      System.out.println("Exception: " + e.getMessage());
      throw new ITSOBankException(accountID);
   }
}
```

The deposit and withdraw methods

The deposit method adds money to an account by retrieving the account and calling its processTransaction method with the Transaction.CREDIT code. The new transaction instance is persisted, as shown in Example 26. The withdraw method is similar and uses the Transaction.DEBIT code.

Example 26   Session bean deposit method

```
public void deposit(String id, BigDecimal amount) throws ITSOBankException {
   System.out.println("deposit: " + id + " amount " + amount);
   Account account = getAccount(id);
   try {
      Transaction tx = account.processTransaction(amount, Transaction.CREDIT);
      entityMgr.persist(tx);
   } catch (Exception e) {
      throw new ITSOBankException(e.getMessage());
   };
}
```

The transfer method

The transfer method calls withdraw and deposit on two accounts to move funds from one account to the other account, as shown in Example 27.

Example 27   Session bean transfer method

```
public void transfer(String idDebit, String idCredit, BigDecimal amount)
                                     throws ITSOBankException  {
   System.out.println("transfer: " + idCredit + " " + idDebit + " amount "
                  + amount);
   withdraw(idDebit, amount);
   deposit(idCredit, amount);
}
```

The openAccount method

The openAccount method creates an account instance with a randomly constructed account number. The instance is persisted, and the customer is added to the customers, as shown in Example 28 on page 37.

Example 28   Session bean openAccount method

```
public String openAccount(String ssn) throws ITSOBankException {
   System.out.println("openAccount: " + ssn);
   Customer customer = getCustomer(ssn);
   int acctNumber = (new java.util.Random()).nextInt(899999) + 100000;
   String id = "00" + ssn.substring(0, 1) + "-" + acctNumber;
   Account account = new Account();
   account.setId(id);
   entityMgr.persist(account);
   List<Customer> custSet = Arrays.asList(customer);
   account.setCustomers(custSet);
   System.out.println("openAccount: " + id);
   return id;
}
```

Adding the "m:m" relationship: The m:m relationship must be added from the *owning* side of the relationship, in our case, from the Account. The code to add the relationship from the Customer side runs without error, but the relationship is not added.

The closeAccount method

The closeAccount method retrieves an account and all its transactions, then deletes all instances by using the Entity Manager remove method, as shown in Example 29.

Example 29   Session bean closeAccount method

```
public void closeAccount(String ssn, String id) throws ITSOBankException {
   System.out.println("closeAccount: " + id + " of customer " + ssn);
   Customer customer = getCustomer(ssn);
   Account account = getAccount(id);
   Transaction[] trans = getTransactions(id);
   for (Transaction tx : trans) {
        entityMgr.remove(tx);
   }
   entityMgr.remove(account);
   System.out.println("closed account with " + trans.length
                  + " transactions");
}
```

The addCustomer method

The addCustomer method accepts a fully constructed Customer instance and makes it persistent, as shown in Example 30 on page 38.

Example 30   Session bean addCustomer method

```
public void addCustomer(Customer customer) throws ITSOBankException {
    System.out.println("addCustomer: " + customer.getSsn());
    entityMgr.persist(customer);}
```

The deleteCustomer method

The `deleteCustomer` method retrieves a customer and all its accounts and then closes the accounts and deletes the customer, as shown in Example 31.

Example 31   Session bean deleteCustomer method

```
public void deleteCustomer(String ssn) throws ITSOBankException {
    System.out.println("deleteCustomer: " + ssn);
    Customer customer = getCustomer(ssn);
    Account[] accounts = getAccounts(ssn);
    for (Account acct : accounts) {
        closeAccount(ssn, acct.getId());
    }
    entityMgr.remove(customer);
}
```

Organize the imports (select `javax.persistence.Query`, and `java.util.List`).

The `EJBBankBean` session bean is now complete. In the following sections, we test the EJB by using a servlet and then proceed to integrate the EJB with a web application.

Testing the session EJB and the JPA entities

To test the session EJB, we can use the Universal Test Client, as described in "Testing with the Universal Test Client" on page 39. As a second approach, we develop a simple servlet that executes all the functions, as described in "Creating a web application to test the session bean" on page 41.

Deploying the application to the server

To deploy the test application, perform these steps:

1. Start WebSphere Application Server V8.0 in the Servers view.

> **JNDI name for data source:** Ensure that the data source for the `ITSOBANK` database is configured with a JNDI name of `jdbc/itsobank` either in the WebSphere Deployment editor or in the administrative console of the server.

2. Select the server and click **Add and Remove Projects**. Add the **RAD8EJBEAR** enterprise application.

3. Click **Finish** and wait for the publishing to finish.

   Notice the EJB binding messages in the console:

   ```
   [...] 00000010 ResourceMgrIm I   WSVR0049I: Binding ITSOBANKejb as
   jdbc/itsobank
   [...] 00000015 EJBContainerI I   CNTR0167I: The server is binding the
   EJBBankService interface of the EJBBankBean enterprise bean in the RAD8EJB.jar
   module of the RAD8EJBEAR application.  The binding location is:
   ejblocal:RAD8EJBEAR/RAD8EJB.jar/EJBBankBean#itso.bank .service.EJBBankService
   [...] 00000015 EJBContainerI I   CNTR0167I: The server is binding the
   EJBBankService interface of the EJBBankBean enterprise bean in the RAD8EJB.jar
   module of the RAD8EJBEAR application.  The binding location is:
   ejblocal:itso.bank.service.EJBBankService
   ```

Testing with the Universal Test Client

Before we integrate the EJB application with the web application, we test the session bean with the access to the JPA entities. We use the enterprise application Universal Test Client (UTC), which is included in Rational Application Developer.

In this section, we describe several operations that you can perform with the Universal Test Client. We use the test client to retrieve a customer and its accounts.

To test the session bean, follow these steps:

1. In the Servers view, right-click the server and select **Universal Test Client** → **Run**.

2. Accept the certificate and log in as `admin/admin` (the user ID that you set up when installing Rational Application Developer).

3. The Universal Test Client opens, as shown in Figure 15.
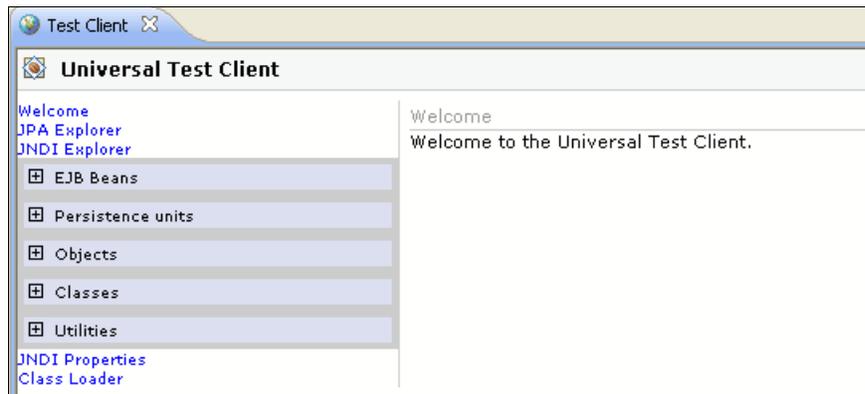
Figure 15   Universal Test Client welcome

4. In the Universal Test Client window, which is shown in Figure 16 on page 40, select **JNDI Explorer** on the left side. On the right side, expand **[Local EJB Beans]**.

5. Select **itso.bank.service.EJBBankService**. The `EJBBankService` is displayed under EJB Beans, as shown in Figure 16 on page 40.
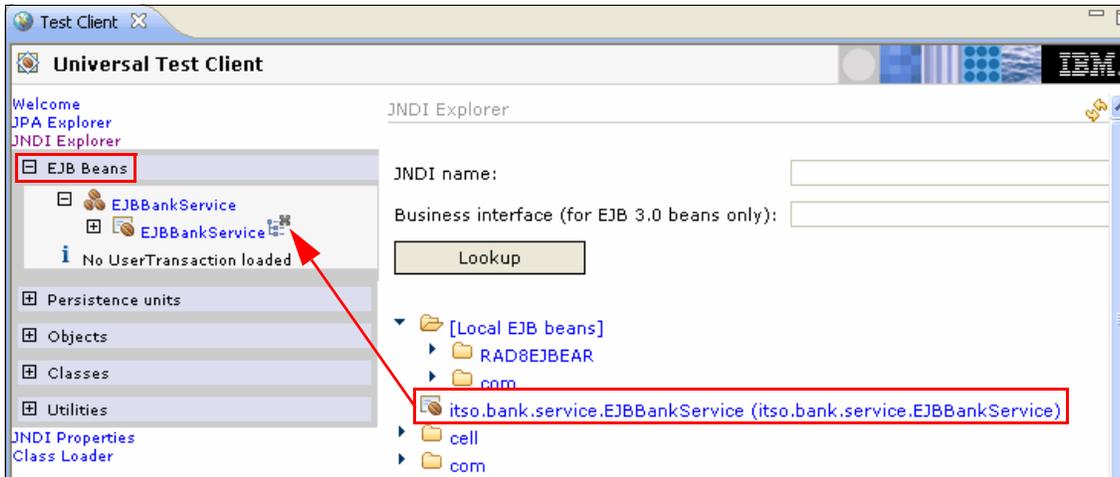
Figure 16   UTC: JNDI Explorer

6. Expand **EJBBankService** (on the left) and select the **getCustomer** method. The method with its parameter opens on the right, as shown in Figure 17.

7. Type `333-33-3333` for the value on the right and click **Invoke**.

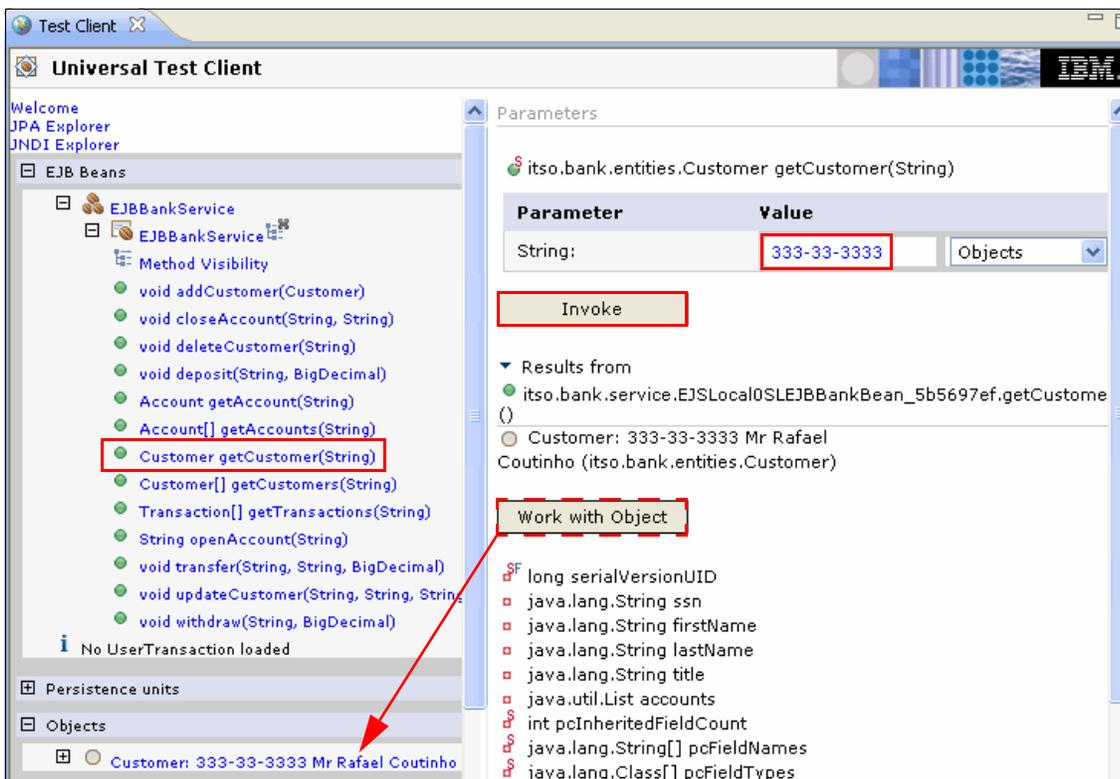A `Customer` instance is displayed as result, as shown in Figure 17 as well.

Figure 17   UTC: Retrieve a customer

8. Click **Work with Object**. The customer instance is displayed under Objects. You can expand the object and invoke its methods (for example, `getLastName`) to see the customer name.

Use the Universal Test Client to ensure that all of the EJB methods work. When you are done, close the Universal Test Client pane.

Creating a web application to test the session bean

To test the EJB 3.1 session bean and entity model, create a small web application with one servlet. Therefore, perform the following steps:

1. Within the Enterprise Explorer view, right-click and select **New** → **Project**.

2. In the New Project wizard, select **Web** → **Dynamic Web Project** and click **Next**.

3. In the New Dynamic Web Project wizard, define the project details, as shown in Figure 18 on page 42:

   a. For Name, type `RAD8EJBTestWeb`.

   b. For Dynamic Web Module version, select **3.0**.

   c. For Configuration, select **Default Configuration for WebSphere Application Server v8.0 Beta**.

   d. Select **Add the project to an EAR**. The value **RAD8EJBEAR** is set as the default (the name of the previously defined `EAR` project for the `RAD8EJB` project).

   e. Click **Finish** and close the help window that opens.
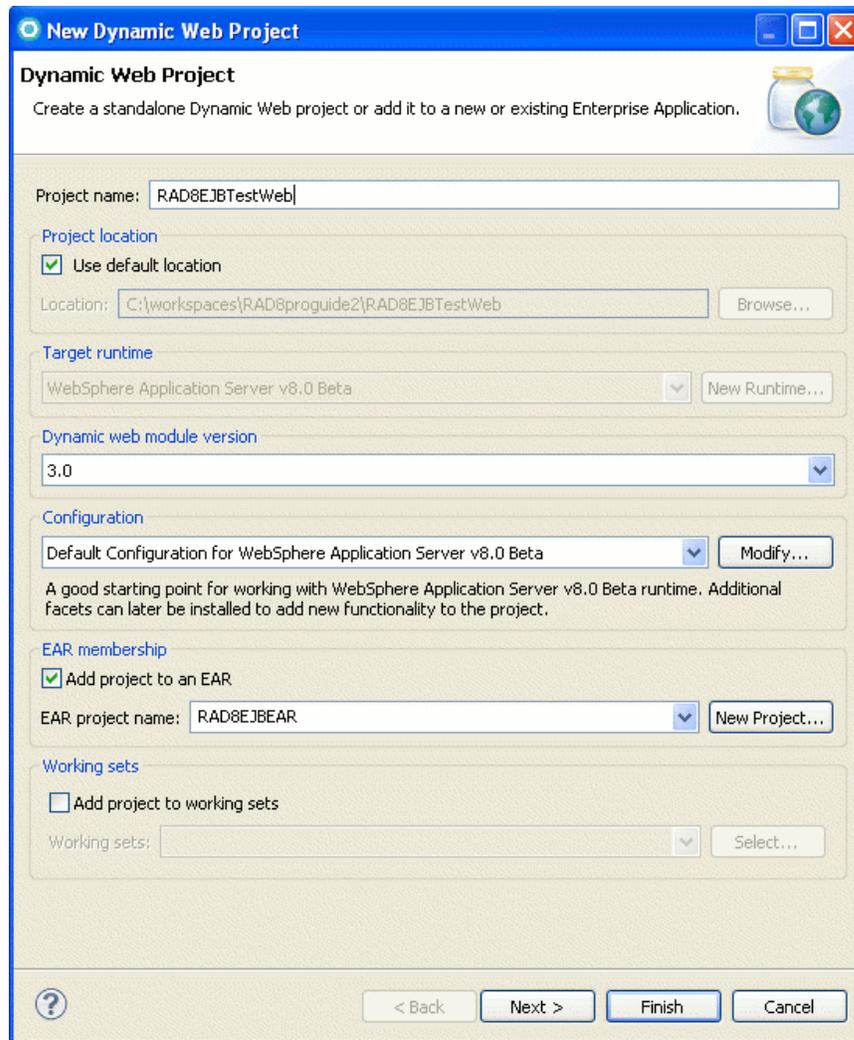
Figure 18   Create RAD8EJBTestWeb project

The Enterprise Explorer view contains the RAD8EJBTestWeb project, which is added to the RAD8EJBEAR enterprise application. Define the dependency to the EJB project RAD8EJB with the following steps:

1. Right-click the **RAD8EJBTestWeb** project and select **Properties**.

2. In the Properties window, select **Project References**, and for Project References, select the **RAD8JPA** module.

3. Click **OK**.

To create a servlet within this RAD8EJBTestWeb project, perform the following steps:

1. Right-click the **RAD8EJBTestWeb** project and select **New** → **Servlet**.

2. For Package name, type itso.test.servlet, and for Class name, type BankTest, as shown in Figure 19 on page 43.
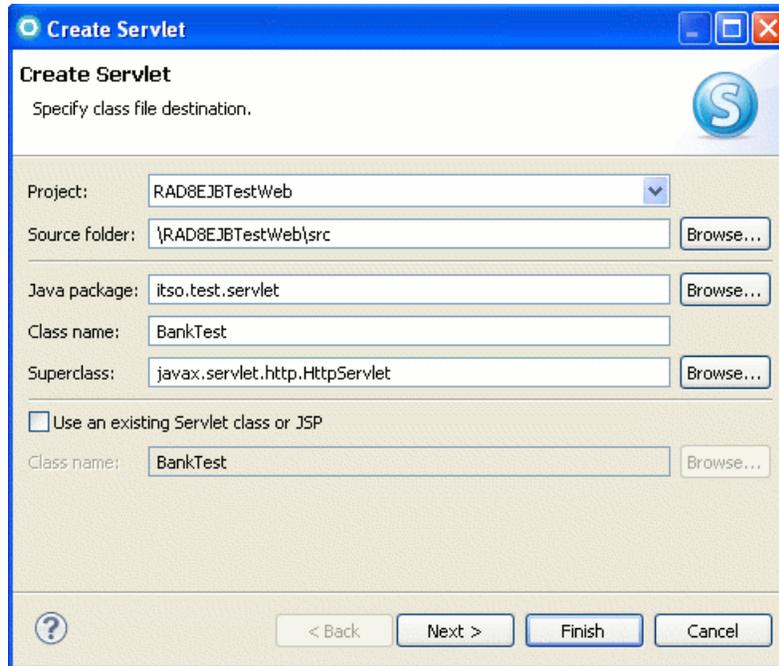
Figure 19 Creating servlet BankTest: Specifying the class file destination

3. Click **Next** twice.

4. Select to generate the **doPost** and **doGet** methods, as shown in Figure 20.
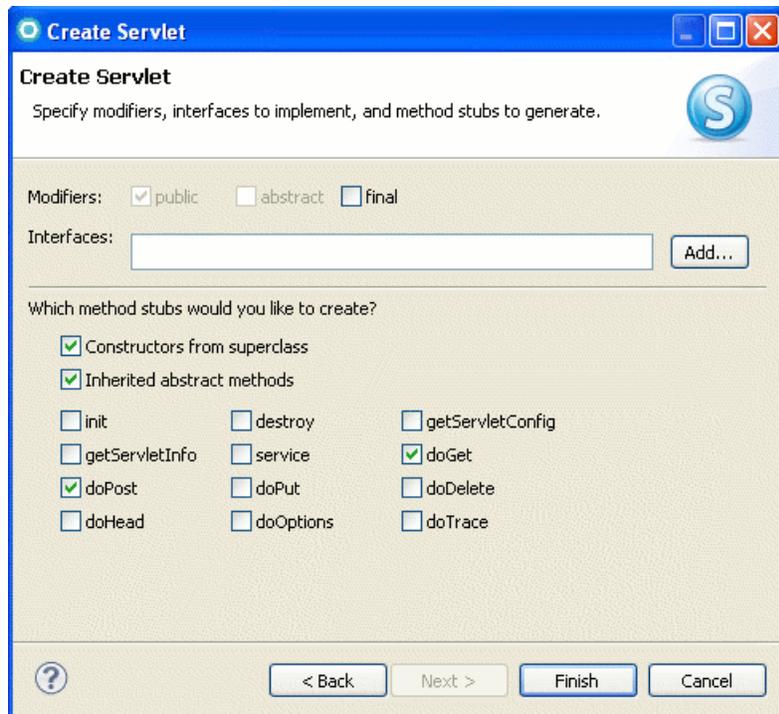
Figure 20 Creating servlet BankTest: Specifying interfaces and method stubs

5. Click **Finish**.

6. After the class definition `BankTest`, add an injector for the business interface:

```
@javax.ejb.EJB EJBBankService bank;
```

The injection of the business interface into the servlet resolves to the automatic binding of the session EJB.

7. In the `doGet` method, enter the code:

```
doPost(request, response);
```

8. Complete the `doPost` method with the code that is shown in Example 32, which is available in the `\4885code\ejb\source\BankTest.txt` file. This servlet executes the methods of the session bean, after getting a reference to the business interface.

Example 32   Servlet to test the EJB 3.1 module (abbreviated)

```
protected void doPost(HttpServletRequest request,
     HttpServletResponse response) throws ServletException, IOException {

   try {
      PrintWriter out = response.getWriter();
      String partialName = request.getParameter("partialName");
      out.println("<html><body><h2>Customer Listing</h2>");
      if (partialName == null) partialName = "%";
      else partialName = "%" + partialName + "%";

      out.println("<p>Customers by partial Name: " + partialName + "<br>");
      Customer[] customers = bank.getCustomers(partialName);

      for (Customer cust : customers) {
         out.println("<br>" + cust);
      }

      Customer cust1 = bank.getCustomer("222-22-2222");
      out.println("<p>" + cust1);

      Account[] accts = bank.getAccounts(cust1.getSsn());
      out.println("<br>Customer: " + cust1.getSsn() + " has " + accts.length + " accounts");
      Account acct = bank.getAccount("002-222002");
      out.println("<p>" + acct);

      out.println("<p>Transactions of account: " + acct.getId());
      Transaction[] trans = bank.getTransactions("002-222002");
      out.println("<p><table border=1><tr><th>Type</th><th>Time</th>...");
      for (Transaction t : trans) {
         out.println("<tr><td>" + t.getTransType() + "</td><td>" + ...);
      }
      out.println("</table>");

      String newssn = "xxx-xx-xxxx";
      bank.deleteCustomer(newssn);        // for rerun
      out.println("<p>Add a customer:  " + newssn);
      Customer custnew = new Customer();
      custnew.setSsn(newssn);
      custnew.setTitle("Mrs");
      custnew.setFirstName("Lara");
      custnew.setLastName("Keen");
      bank.addCustomer(custnew);
```

```
        Customer cust2 = bank.getCustomer(newssn);
        out.println("<br>" + cust2);

        out.println("<p>Open two accounts for customer: " + newssn);
        String id1 = bank.openAccount(newssn);
        String id2 = bank.openAccount(newssn);
        out.println("<br>New accounts: " + id1 + " " + id2);
        Account[] acctnew = bank.getAccounts(newssn);
        out.println("<br>Customer: " +newssn + " has " +acctnew.length ...);
        Account acct1 = bank.getAccount(id1);
        out.println("<br>" + acct1);

        out.println("<p>Deposit and withdraw from account: " + id1);
        bank.deposit(id1, new java.math.BigDecimal("777.77"));
        bank.withdraw(id1, new java.math.BigDecimal("111.11"));
        acct1 = bank.getAccount(id1);
        out.println("<br>Account: " +id1+ " balance " + acct1.getBalance());

        trans = bank.getTransactions(id1);
        out.println("<p><table border=1><tr><th>Type</th><th>Time</th>...");
        for (Transaction t : trans) {
            out.println("<tr><td>" + t.getTransType() + ...");
        }
        out.println("</table>");

        out.println("<p>Close the account: " + id1);
        bank.closeAccount(newssn, id1);

        out.println("<p>Update the customer: " + newssn);
        bank.updateCustomer(newssn, "Mrs", "Sylvi", "Sollami");
        cust2 = bank.getCustomer(newssn);
        out.println("<br>" + cust2);
        out.println("<p>Delete the customer: " + newssn);
        bank.deleteCustomer(newssn);

        out.println("<p>Retrieve non existing customer: ");
        Customer cust3 = bank.getCustomer("zzz-zz-zzzz");
        out.println("<br>customer: " + cust3);

        out.println("<p>End</body></html>");
    } catch (Exception e) {
        System.out.println("Exception: " + e.getMessage());
        e.printStackTrace();
    }
}
```

Testing the sample web application

To test the web application, run the servlet:

1. Expand the test web project **Deployment Descriptor** → **Servlets**. Select the **BankTest** servlet, right-click, and select **Run As** → **Run on Server**.

2. In the Run On Server window, select the **WebSphere Application Server v8.0 Beta** server, select **Always use this server when running this project**, and click **Finish**.

3. Accept the security certificate (if security is enabled).

Example 33 shows a sample output of the servlet.

Example 33   Servlet output (abbreviated)

```
Customer Listing
Customers by partial Name: %

Customer: 111-11-1111 Mr Henry Cui
Customer: 222-22-2222 Mr Craig Fleming
Customer: 333-33-3333 Mr Rafael Coutinho
Customer: 444-44-4444 Mr Salvatore Sollami
Customer: 555-55-5555 Mr Brian Hainey
Customer: 666-66-6666 Mr Steve Baber
Customer: 777-77-7777 Mr Sundaragopal Venkatraman
Customer: 888-88-8888 Mrs Lara Ziosi
Customer: 999-99-9999 Mrs Sylvi Lippmann
Customer: 000-00-0000 Mrs Venkata Kumari
Customer: 000-00-1111 Mr Martin Keen

Customer: 222-22-2222 Mr Craig Fleming
Customer: 222-22-2222 has 3 accounts

Account: 002-222002 balance 87.96

Transactions of account: 002-222002

Type Time Amount
Debit 2002-06-06 12:12:12.0 3.33
Credit 2003-07-07 14:14:14.0 6666.66
Credit 2004-01-08 23:03:20.0 700.77

Add a customer: xxx-xx-xxxx
Customer: xxx-xx-xxxx Mrs Lara Keen

Open two accounts for customer: xxx-xx-xxxx
New accounts: 00x-496969 00x-915357
Customer: xxx-xx-xxxx has 2 accounts
Account: 00x-496969 balance 0.00

Deposit and withdraw from account: 00x-496969
Account: 00x-496969 balance 666.66

Type Time Amount
Credit 2010-10-18 19:37:22.906 777.77
Debit 2010-10-18 19:37:23.0 111.11

Close the account: 00x-496969

Update the customer: xxx-xx-xxxx
Customer: xxx-xx-xxxx Mrs Sylvi Sollami

Delete the customer: xxx-xx-xxxx

Retrieve non existing customer:
```

```
customer: null

End
```

Visualizing the test application

You can improve the generated class diagram by adding the business interface, the entities, and the servlet to the diagram, as shown in Figure 21.
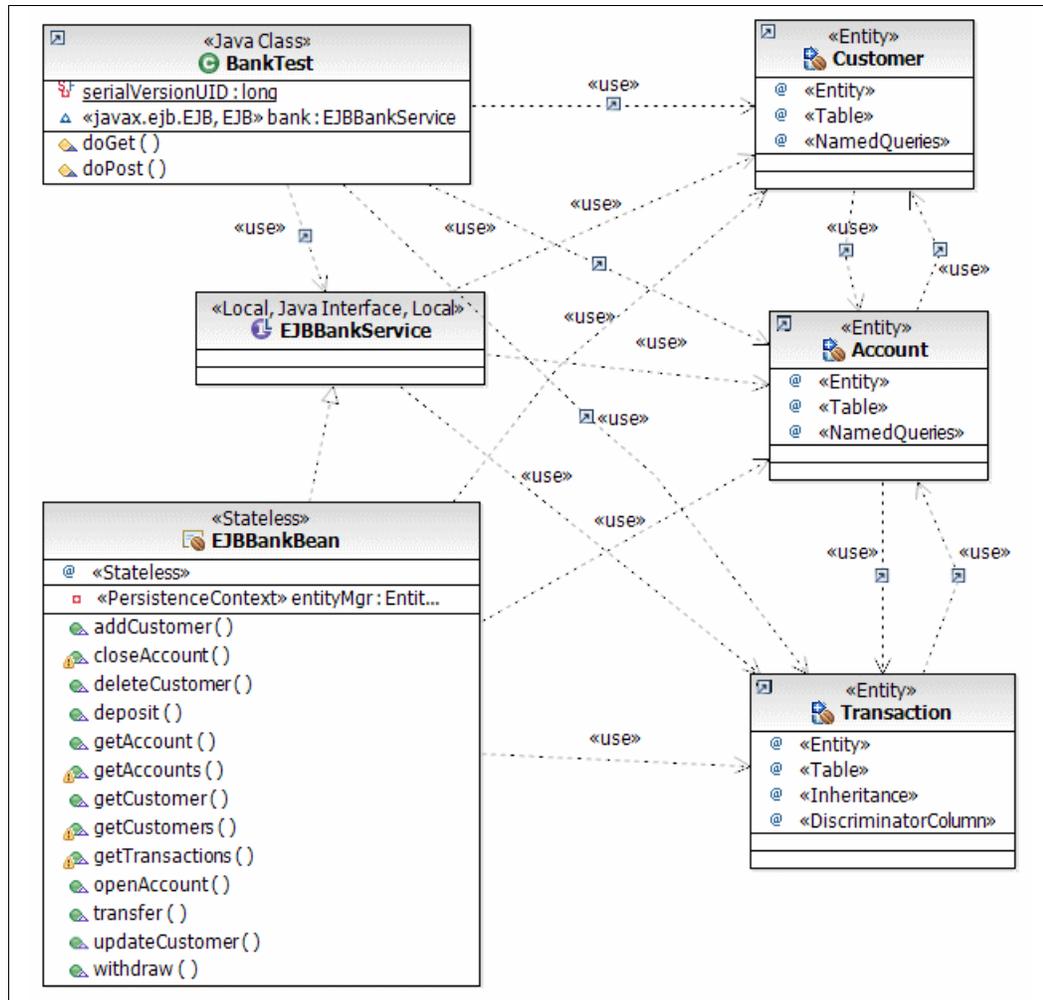

Figure 21   Class diagram of the test web application

Invoking EJB from web applications

In this section, we describe how to create a web application. The `RAD8EJBWeb` application uses the JPA entities that are provided by the `RAD8JPA` project and accesses these entities through the `EJBBankBean` session bean of the `RAD8EJB` project.

Implementing the RAD8EJBWeb application

The `RAD8EJBWeb` application uses EJB 3.1 APIs to communicate with the `EJBBankBean` session bean.

You can import the finished application from the `\4885codesolution\ejb\RAD8EJBWeb.zip` file.

> **Importing projects**: If you already have `RAD8EJB` and `RAD8JPA` projects in the workspace, import only `RAD8EJBWeb` and `RAD8EJBWebEAR`.

Web application navigation

Figure 22 shows the navigation between the web pages.
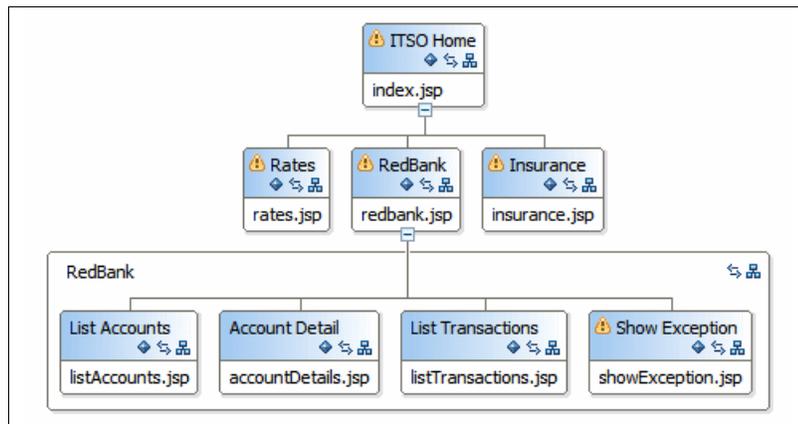
Figure 22   Website navigation

Note the following points:

► From the home page (`index.jsp`), there are three static pages (`rates.jsp`, `insurance.jsp`, and `redbank.jsp`).

► The `redbank.jsp` is the login panel for customers.

► After the login, the customer's details and the list of accounts are displayed (`listAccounts.jsp`).

► An account is selected in the list of accounts, and the details of the account and a form for transaction list, deposit, withdraw, and transfer operations are displayed (`accountDetails.jsp`).

► From the account details form, banking transactions are executed:

 – List transaction shows the list of previous debit and credit transactions (`listTransactions.jsp`).

 – Deposit, withdraw, and transfer operations are executed, and the updated account information is displayed in the same page.

► Additional functions are to delete an account, update customer information, add an account to a customer, and to delete the customer.

► If errors occur, an error page is displayed (`showException.jsp`).

The JSP are based on the template that provides navigation bars through headers and footers:

`/theme/itso_jsp_template.jtpl, nav_head.jsp, footer.jsp`

Servlets and commands

Several servlets provide the processing and switching between the web pages:

`ListAccounts`	Performs the customer login, retrieves the customer and the accounts, and forwards them to the `accountDetails.jsp`.
`AccountDetails`	Retrieves one account and forwards it to the `accountDetails.jsp`.
`PerformTransaction`	Validates the form values and calls one of the commands (`ListTransactionsCommand`, `DepositCommand`, `WithdrawCommand`, or `TransferCommand`). The commands perform the requested banking transaction and forwards it to the `listTransactions.jsp` or the `accountDetails.jsp`.
`UpdateCustomer`	Processes updates of customer information and the deletion of a customer.
`DeleteAccount`	Deletes an account and forwards it to the `listAccounts.jsp`.
`NewAccount`	Creates an account and forwards it to the `listAccounts.jsp`.
`Logout`	Logs out and displays the home page.

Java EE dependencies

The enterprise application (`RAD8EJBWebEAR`) includes the web module (`RAD8EJBWeb`), the EJB module (`RAD8EJB`), and the JPA Utility project (`RAD8JPA`).

The web module (`RAD8EJBWeb`) has a dependency on the EJB module (`RAD8EJB`), which has a dependency on the JPA project (`RAD8JPA`).

Accessing the session EJB

All database processing is done through the `EJBBankBean` session bean, by using the business interface `EJBBankService`.

The servlets use EJB 3.1 injection to access the session bean:

```
@EJB EJBBankService bank;
```

After this injection, all the methods of the session bean can be invoked, such as the following methods that are shown in Example 34.

Example 34   EJBBankService methods

```
Customer customer = bank.getCustomer(customerNumber);
Account{} accounts = bank.getAccounts(customerNumber);
bank.deposit(accountId, amount);
```

Additional functions

We improved the application and added the following functions:

► On the customer details panel (`listAccounts.jsp`), we added three buttons:

  – New Customer: Enter data into the title, first name, and last name fields, then click **New Customer**. A customer is created with a random Social Security number.

  – Add Account: This action adds an account to the customer, with a random account number and zero balance.

  – Delete Customer: Deletes the customer and all related accounts.

  The logic for adding and deleting a customer is in the `UpdateCustomer` servlet. The logic for a new account is in `NewAccount` servlet.

► On the account details page (`accountDetails.jsp`), we added the **Delete Account** button. You click this button to delete the account with all its transactions. The customer with its remaining accounts is displayed next.

The logic for deleting an account is in `DeleteAccount` servlet.

► For the Login panel, we added logic in the `ListAccounts` servlet so that the user can enter a last name instead of the SSN.

If the search by SSN fails, we retrieve all customers with that partial name. If only one result is found, we accept it and display the customer. This display parameter allows entry of partial names, such as `So%`, to find the `Sollami` customer.

Running the web application

Before running the web application, we must have the data source for the `ITSOBANK` database configured. See "Setting up the ITSOBANK database" on page 25, for instructions. You can either configure the enhanced EAR in the `RAD8EJBWebEAR` application or define the data source in the server.

To run the web application, perform these steps:

1. In the Servers view, right-click the server and select **Add and Remove Projects**. Remove the **RAD8EJBEAR** application and add the **RAD8EJBWebEAR** application. Then click **Finish**.

2. Right-click the **RAD8EJBWeb** project and select **Run As** → **Run on Server**.

3. When prompted, select **WebSphere Application Server v8.0 Beta**.

4. Your start page is the `redbank.jsp` login page, as shown in Figure 23. Because we want to focus on the `RedBank` application, we set the `redbank.jsp` as a welcome-list entry in the `web.xml` configuration file, as well.
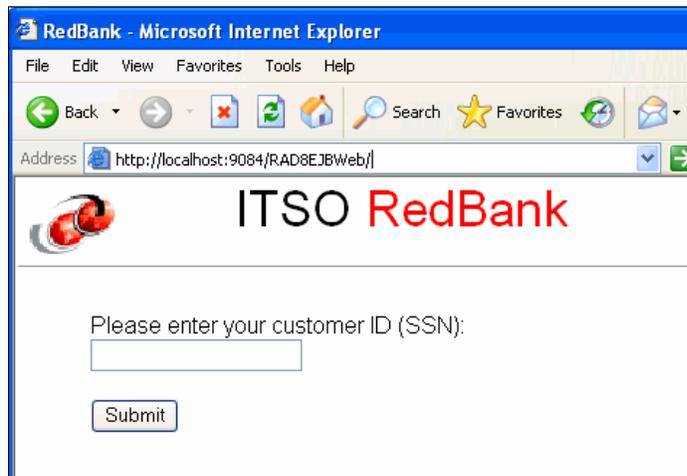

Figure 23   RedBank: Login

5. Enter a customer number, such as 333-33-3333, and click **Submit**. The customer details and the list of accounts are displayed as shown in Figure 24.

Figure 24   RedBank: Customer with accounts

6. Click an account, such as **003-333001**, and the details and possible actions are displayed, as shown in Figure 25 on page 52.
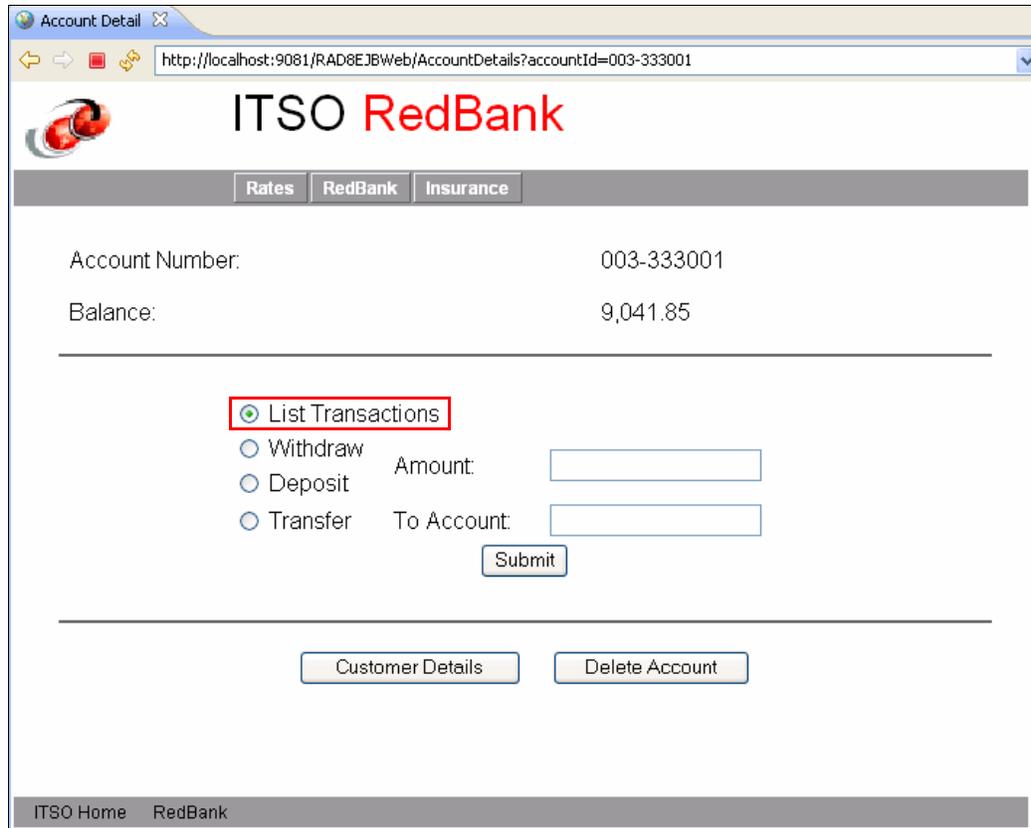
Figure 25   RedBank: Account details

7. Select **List Transactions** and click **Submit**. The transactions are listed, as shown in Figure 26.
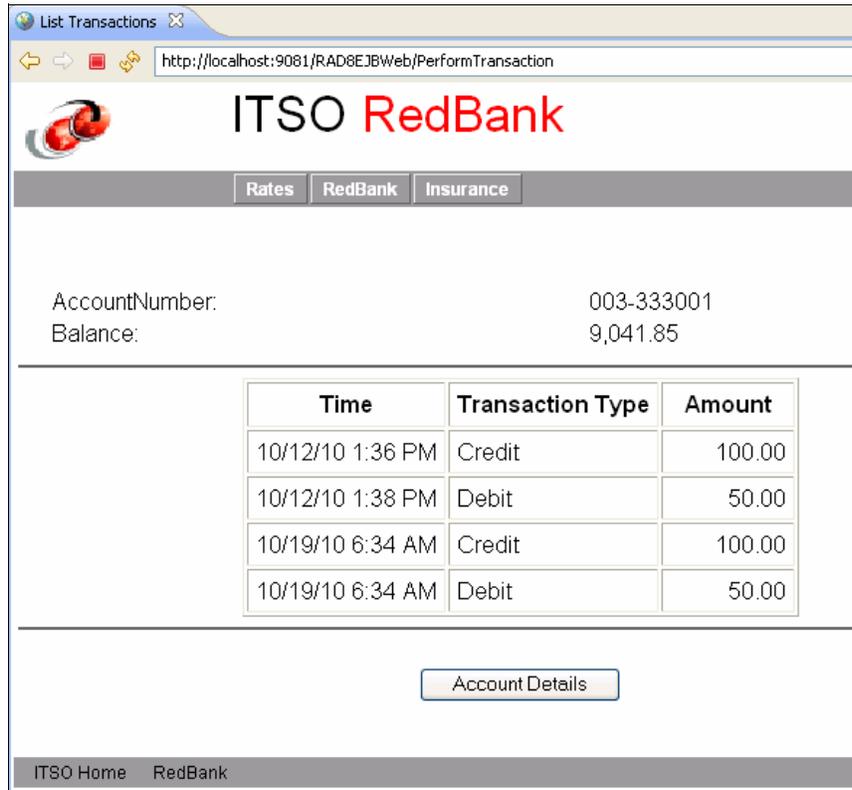

Figure 26   RedBank: Transactions

8. Click **Account Details** to return to the account.

9. Select **Deposit**, enter an amount (58.15), and click **Submit**. The balance is updated to 9,100.00.

10. Select **Withdraw**, enter an amount (100), and click **Submit**. The balance is updated to 9,000.00.

11. Select **Transfer**. Enter an amount (3000) and a target account (003-333002) and click **Submit**. The balance is updated to 6,000.00.

12. Select **List Transactions** and click **Submit**. The transactions are listed and there are three more entries, as shown in Figure 27.

Figure 27   RedBank: Transactions added

13. Click **AccountDetails** to return to the account. Click **Customer Details** to return to the customer.

14. Click the second account and then click **Submit**. You can see that the second account has a transaction from the transfer operation, as shown in Figure 28 on page 55.
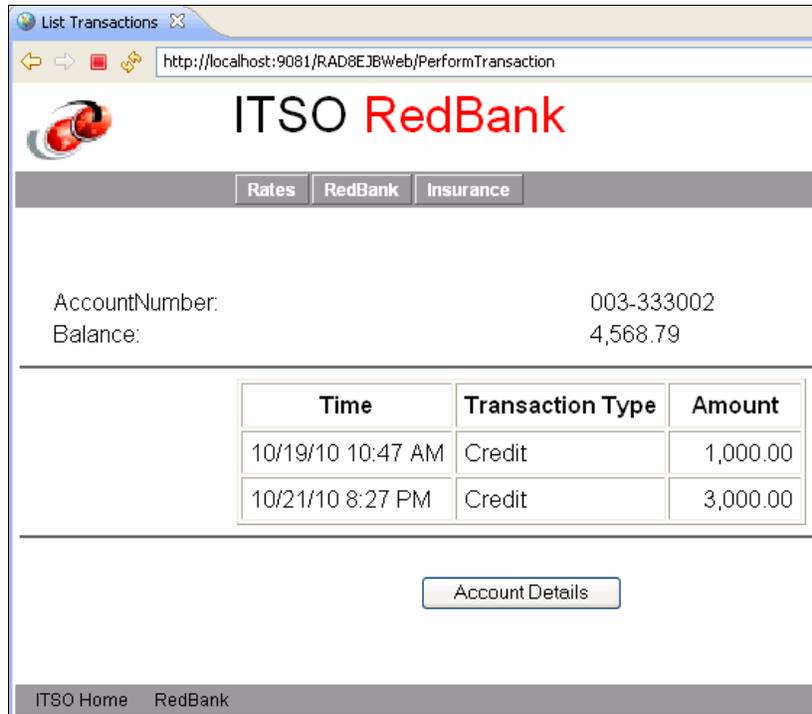
Figure 28   Transfer result to account 003-333002

15. Back in the customer details, change the last name and click **Update**. The customer information is updated.

16. Overtype the first and last names with `Jonny Lippmann` and click **New Customer**. Then a new customer with SSN `395-60-9710` is created, as shown in Figure 29 on page 56.

17. Click **Add Account**, and an account 003-365234 with balance 0.00 is added to the customer, as shown in Figure 29.
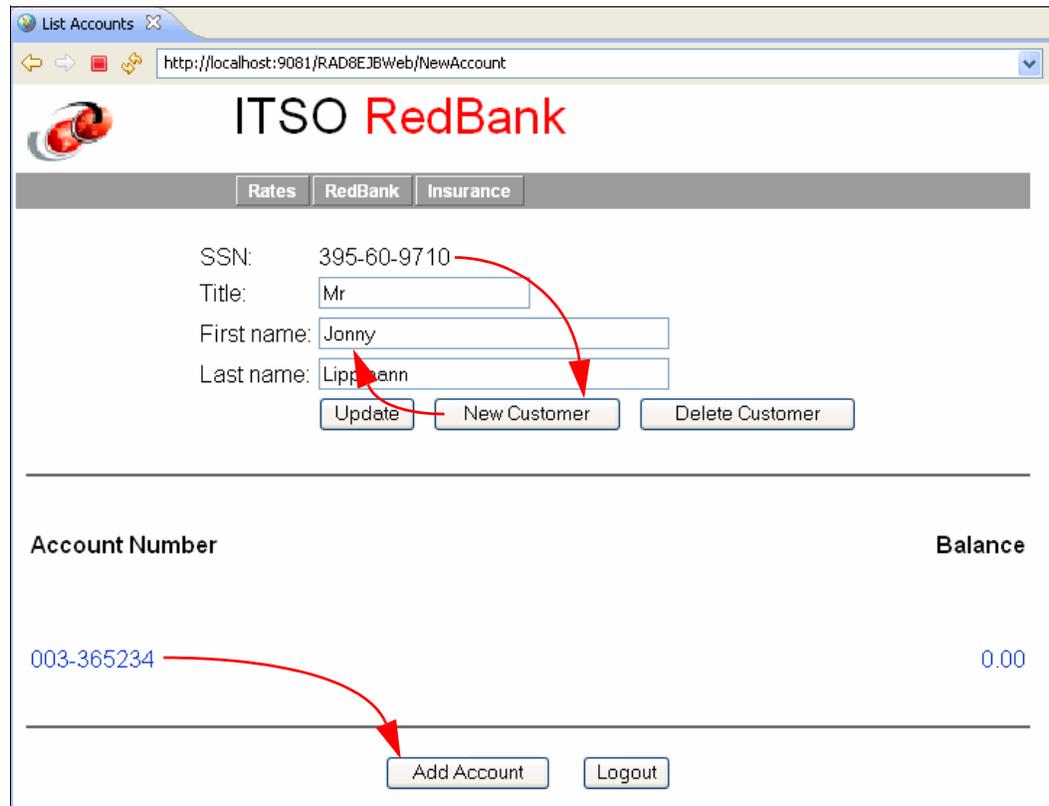

Figure 29   RedBank: New customer and new account

18. Perform transactions on the new account.

19. Go back to customer details and click **Delete Customer**.

20. In the Login panel, enter an incorrect value and click **Submit**. The customer details panel is displayed with a "NOT FOUND" last name.

21. Click **Logout**.

Cleaning up

Remove the RAD8EJBWebEAR application from the server.

Adding a remote interface

For testing by using JUnit and for certain web applications, we define a remote interface for the EJBBankBean session bean. Perform the following steps:

1. In the RAD8EJB project, itso.bank.service package, create an interface named EJBBankRemote, which extends the business interface, EJBBankService.

2. Add one method to the interface, getCustomersAll, to retrieve all the customers.

3. Add an @Remote annotation before your interface class definition. Example 35 shows the defined remote interface.

Example 35   Remote interface of the session bean

```
package itso.bank.service;
import itso.bank.entities.Customer;
import javax.ejb.Remote;
@Remote
public interface EJBBankRemote extends EJBBankService {
    public Customer[] getCustomersAll();
}
```

4. Open the **EJBBankBean** session bean:

   a. Add the EJBBankRemote interface to the implements list.

   b. Implement the getCustomersAll method, as shown in Example 36. This method is similar to the getCustomers method, by using the getCustomers named query (without a parameter).

Example 36   Extend EJBBankBean with remote interface

```
public class EJBBankBean implements EJBBankService, EJBBankRemote {
    ......
    public Customer[] getCustomersAll() {
        System.out.println("getCustomers: all");
        Query query = null;
        try {
            query = entityMgr.createNamedQuery("getCustomers");
            List<Customer> beanlist = query.getResultList();
            Customer[] array = new Customer[beanlist.size()];
            return beanlist.toArray(array);
        } catch (Exception e) {
            System.out.println("Exception: " + e.getMessage());
            return null;
        }
    }
}
```

More information

For more information about EJB 3.1 and EJB 3.0, see the following resources:

► *WebSphere Application Server Version 6.1 Feature Pack for EJB 3.0*, SG24-7611

► *JSR 318: Enterprise JavaBeans 3.1*:

   http://jcp.org/en/jsr/summary?id=318

► WebSphere Application Server Information Center:

   http://pic.dhe.ibm.com/infocenter/wasinfo/v8r0/index.jsp

Locating the web material

The web material that is associated with this paper is available in softcopy on the Internet from the IBM Redbooks web server. Enter the following URL in a web browser and then download the two `.zip` files:

ftp://www.redbooks.ibm.com/redbooks/REDP4885

Alternatively, you can go to the IBM Redbooks website:

http://www.ibm.com/redbooks

Accessing the web material

Select **Additional materials** and open the directory that corresponds with the IBM Redbooks publication form number, REDP-4885.

> **Additional information:** For more information about the additional material, see *Rational Application Developer for WebSphere Software V8 Programming Guide*, SG24-7835.

The additional web material that accompanies this paper includes the following files:

File name	Description
4885code.zip	Compressed file that contains sample code
4885codesolution.zip	Compressed file that contains solution interchange files

System requirements for downloading the web material

We recommend the following system configuration:

Hard disk space:	20 GB minimum
Operating system:	Microsoft Windows or Linux
Processor:	2 GHz
Memory:	2 GB

The team who wrote this paper

This paper was produced by a team of specialists from around the world working at the International Technical Support Organization, Raleigh Center.

Martin Keen is a Consulting IT Specialist at the ITSO, Raleigh Center. He writes extensively about WebSphere products and service-oriented architecture (SOA). He also teaches IBM classes worldwide about WebSphere, SOA, and enterprise service bus (ESB). Before joining the ITSO, Martin worked in the EMEA WebSphere Lab Services team in Hursley, UK. Martin holds a Bachelors degree in Computer Studies from Southampton Institute of Higher Education.

Rafael Coutinho is an IBM Advisory Software Engineer working for Software Group in the Brazil Software Development Lab. His professional expertise covers many technology areas ranging from embedded to platform-based solutions. He is currently working on IBM Maximo® Spatial, which is the geographic information system (GIS) add-on of IBM Maximo Enterprise Asset Management (EAM). He is a certified Java enterprise architect and

Accredited IT Specialist, specialized in high-performance distributed applications on corporate and financial projects.

Rafael is a computer engineer graduate from the State University of Campinas (Unicamp), Brazil, and has a degree in Information Technologies from the Centrale Lyon (ECL), France.

Sylvi Lippmann is a Software IT Specialist in the GBS Financial Solutions team in Germany. She has over seven years of experience as a Software Engineer, Technical Team Leader, Architect, and Customer Support representative. She is experienced in the draft, design, and realization of object-oriented software systems, in particular, the development of Java EE-based web applications, with a priority in the surrounding field of the WebSphere product family. She holds a degree in Business Informatic Engineering.

Salvatore Sollami is a Software IT Specialist in the Rational brand team in Italy. He has been working at IBM with particular interest in the change and configuration area and web application security. He also has experience in the Agile Development Process and Software Engineering. Before joining IBM, Salvatore worked as a researcher for Process Optimization Algorithmic, Mobile Agent Communication, and IT Economics impact. He developed the return on investment (ROI) SOA investment calculation tool. He holds the "Laurea" (M.S.) degree in Computer Engineering from the University of Palermo. In cooperation with IBM, he received an M.B.A. from the MIP - School of Management - polytechnic of Milan.

Sundaragopal Venkatraman is a Technical Consultant at the IBM India Software Lab. He has over 11 years of experience as an Architect and Lead working on web technologies, client server, distributed applications, and IBM System z®. He works on the WebSphere stack on process integration, messaging, and the SOA space. In addition to handling training on WebSphere, he also gives back to the technical community by lecturing at WebSphere technical conferences and other technical forums.

Steve Baber has been working in the Computer Industry since the late 1980s. He has over 15 years of experience within IBM, first as a consultant to IBM and then as an employee. Steve has supported several industries during his time at IBM, including health care, telephony, and banking and currently supports the IBM Global Finance account as a Team Lead for the Global Contract Management project.

Henry Cui works as an independent consultant through his own company, Kaka Software Solution. He provides consulting services to large financial institutions in Canada. Before this work, Henry worked with the IBM Rational services and support team for eight years, where he helped many clients resolve design, development, and migration issues with Java EE development. His areas of expertise include developing Java EE applications with Rational Application Developer tools and administering WebSphere Application Server servers, security, SOA, and web services. Henry is a frequent contributor of IBM developerWorks® articles. He also co-authored five IBM Redbooks publications. Henry holds a degree in Computer Science from York University.

Craig Fleming is a Solution Architect who works for IBM Global Business Services® in Auckland, New Zealand. He has worked for the last 15 years leading and delivering software projects for large enterprises as a solution developer and architect. His area of expertise is in designing and developing middleware solutions, mainly with WebSphere technologies. He has worked in several industries, including Airlines, Insurance, Retail, and Local Government. Craig holds a Bachelor of Science (Honors) in Computer Science from Otago University in New Zealand.

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Stay connected to IBM Redbooks publications

► Find us on Facebook:

http://www.facebook.com/IBMRedbooks

► Follow us on Twitter:

http://twitter.com/ibmredbooks

► Look for us on LinkedIn:

http://www.linkedin.com/groups?home=&gid=2130806

► Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm

► Stay current on recent Redbooks publications with RSS Feeds:

http://www.redbooks.ibm.com/rss.html

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

This document REDP-4885-00 was created or updated on July 31, 2012.

Send us your comments in one of the following ways:
► Use the online **Contact us** review Redbooks form found at:
  ibm.com/redbooks
► Send your comments in an email to:
  redbooks@us.ibm.com
► Mail your comments to:
  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400 U.S.A.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

ClearCase®	Maximo®	Redpaper™
DB2®	Rational Rose®	Redbooks (logo) ®
developerWorks®	Rational Team Concert™	System z®
Global Business Services®	Rational®	WebSphere®
IBM®	Redbooks®	

The following terms are trademarks of other companies:

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java, and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Other company, product, or service names may be trademarks or service marks of others.