Enterprise JavaBeans Fundamentals

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Tutorial tips

Should I take this tutorial?

This tutorial provides an introduction to Enterprise JavaBeans technology with particular attention to the role of Enterprise JavaBean components in distributed-computing scenarios, the architecture, the extension APIs, and the fundamentals of working with EJB technologies.

Concepts

After completing this tutorial, you will understand:

* The overall architecture
* The roles of clients and server
* The life cycles of session and entity beans

Objectives

By the end of this tutorial you will be able to:

* Build EJB technology-based distributed systems
* Create entity beans
* Create session beans
* Deploy solutions in a server
* Create standalone enterprise bean clients
* Use entity beans from within session beans

Tutorial navigation

Navigating through the tutorial is easy:

* Use the Next, Next Section, and Previous buttons to move forward and backward through the tutorial.
* Use the Main menu button to return to the tutorial menu. Within a section, use the Section menu to see the list of topics in that section.
* If you'd like to tell us what you think, or if you have a question for the author about the content of the tutorial, use the Feedback button.

Contact

jGuru has been dedicated to promoting the growth of the Java technology community through evangelism, education, and software since 1995. You can find out more about their activities, including their huge collection of FAQs at *jGuru.com* . To send feedback to jGuru about this course, send mail to *producer@jguru.com* .

Course Authors:
* *Richard Monson-Haefel*

* *Tim Rohaly*

For information about the authors of this course, click on the links above to see their Bio pages at jGuru.

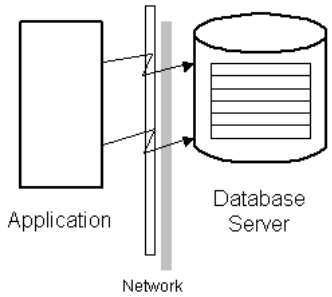
Section 2. EJB technology overview

Introduction

The Enterprise JavaBeans (EJB) 1.1 specification defines an architecture for the development and deployment of transactional, distributed object applications-based, server-side software components. Organizations can build their own components or purchase components from third-party vendors. These server-side components, called enterprise beans, are distributed objects that are hosted in Enterprise JavaBean containers and provide remote services for clients distributed throughout the network.

Two-tier and three-tier environments

In two-tier client-server environments, programmers write applications that are closely tied to vendor-specific software. Typically, two-tier applications access database services or transaction services directly from the client. Such applications are sometimes called fat clients because the application logic resides on the client, making the clients large and complex. This is depicted by the following diagram:



Two-tier Computing Environments

Three-tier client-server applications employ an intermediary, or middle-tier, application server, which operates between client applications and the back-end databases. The middle tier houses the business logic of the system and coordinates the interaction of the presentation on the client with the back-end databases.

There are two fundamental motivations for using a three-tier architecture over a two-tier model:

* Improved scalability, availability, and performance
* Improved flexibility and extensibility of business systems

Two-tier systems perform well by leveraging the processing power of the client, but the dedicated nature of many clients to a single back-end resource, like a database, produces a bottleneck that inhibits scalability, availability, and performance as client populations grow large. Three-tier systems attempt to mitigate this bottleneck by managing back-end resources more effectively. This is accomplished through resource management techniques like pooling and clustering of middle-tier servers. Pooling makes three-tier systems more effective by allowing many clients to share scarce resources like database connections, which reduces the workload on back-end servers. Clustering makes three-tier systems

more available and scalable because multiple servers and resources can support fail-over and balance the loads of a growing client population.

Three-tier systems are more flexible and extensible than their two-tier counterparts because the business logic and services, such as security and transactions, reside on the middle-tier and are largely hidden from the client applications. If properly implemented, as is the case with Enterprise JavaBeans, services are applied automatically to client requests and are therefore invisible. Because services are not visible to the client, changes to services are also invisible. Changes and enhancements to business logic on the middle tier can also be hidden from client applications if implemented correctly.

Additionally, when clients and middleware components are implemented in the Java programming language, there is a tremendous potential for portability. The class files that implement the clients as well as the application servers can be relocated quite easily to the hosts that are currently most appropriate.

Over the last two to three years, several vendors have released Java-based three-tier application servers, all of which are capable of interacting with and managing back-end server operations. Even though these middleware products support distributed architectures that are a very significant advance over two-tier designs (and over pre-Java application servers), their fundamental limitation is that their programming models tend to be vendor specific. This means that an organization must invest heavily in a vendor's model and that systems are not portable, leading to vendor lock-in.

As the object-oriented programming paradigm has grown in popularity, distributed-object systems have thrived. Several distributed-object technologies now exist. The most popular are CORBA, created by the *Object Management Group* , Java RMI (JRMP) from Sun Microsystems, and DCOM and MTS (a.k.a. COM+) from Microsoft. Each has its strengths and weaknesses. Enterprise JavaBeans from Sun Microsystems is the most recent addition to this mix and in some regards, it's both a competitor and a partner in these technologies.

CORBA (Common Object Request Broker Architecture) went a long way toward addressing vendor lock-in issues in three-tier computing as did other open standards like LDAP. Unfortunately, while CORBA has revolutionized distributed computing, the programming model proved too complex and the vendor adherence to the specification unbalanced. CORBA has advanced distributed computing, but has proven too difficult to implement and less portable than expected.

Enterprise JavaBeans (EJB) is Sun Microsystems' solution to the portability and complexity of CORBA. EJB introduces a much simpler programming model than CORBA, allowing developers to create portable distributed components called *enterprise beans*. The EJB programming model allows developers to create secure, transactional, and persistent business objects (enterprise beans) using a very simple programming model and declarative attributes. Unlike CORBA, facilities such as access control (authorization security) and transaction management are extremely simple to program. Where CORBA requires the use of complex APIs to utilize these services, EJB applies these services to the enterprise bean automatically according to declarations made in a kind of property file called a deployment descriptor. This model ensures that bean developers can focus on writing business logic while the container manages the more complex but necessary operations automatically.

Portability in EJB works because the EJB specification mandates a well-defined set of contracts between the EJB container (the vendor's server) and the EJB component (the

business object). These contracts or rules state exactly what services a container must make available to an enterprise bean and what APIs and declarative attributes the bean developer needs to create an enterprise bean. The life cycle of an enterprise bean is specified in detail, so the vendor knows how to manage the bean at run time and the bean developer knows exactly what an enterprise bean can do at any moment in its existence.

Enterprise JavaBeans simplifies the development, deployment, and access to distributed objects. The developer of an EJB distributed object, an enterprise bean, simply implements the object according to the conventions and protocol established for Enterprise JavaBeans. EJB-capable application servers may, and do, use any distributed network protocol including the native Java RMI protocol (JRMP), proprietary protocols, or CORBA's network protocol (IIOP). Regardless of the underlying network protocol used in a particular product, EJB uses the same programming API and semantics to access distributed objects as Java RMI-IIOP. The details of the protocol are hidden from the application and bean developer; the mechanics of locating and using distributed beans are the same for all vendors.

Note: An enterprise bean is not the same as a JavaBean. A JavaBean is developed using the `java.beans` package, which is part of the Java 2 Standard Edition. JavaBeans are components that run on one machine, within a single address space. JavaBeans are process components. An enterprise bean is developed using the `javax.ejb` package, a standard JDK extension, which is a part of the Java 2 Enterprise Edition. Enterprise beans are components that run on multiple machines, across several address spaces. Enterprise beans are thus *inter*process components. JavaBeans are typically used as GUI widgets, while enterprise beans are used as distributed business objects.

Section 3. The EJB specification

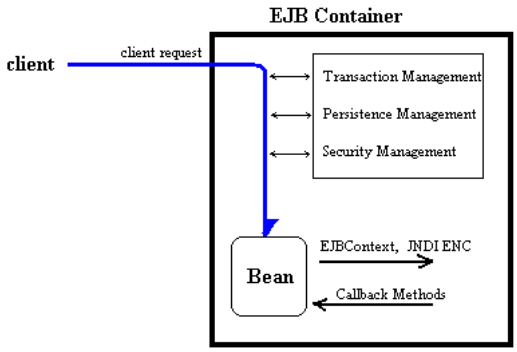
Introduction

The Enterprise JavaBeans specification defines an architecture for a transactional, distributed object system based on components. The specification mandates a programming model, that is, conventions or protocols and a set of classes and interfaces that make up the EJB API. The EJB programming model provides bean developers and EJB server vendors with a set of contracts that defines a common platform for development. The goal of these contracts is to ensure portability across vendors while supporting a rich set of functionality.

The EBJ container

Enterprise beans are software components that run in a special environment called an EJB container. The container hosts and manages an enterprise bean in the same manner that the Java Web server hosts a servlet or an HTML browser hosts a Java applet. An enterprise bean cannot function outside of an EJB container. The EJB container manages every aspect of an enterprise bean at run time including remote access to the bean, security, persistence, transactions, concurrency, and access to and pooling of resources.

The container isolates the enterprise bean from direct access by client applications. When a client application invokes a remote method on an enterprise bean, the container first intercepts the invocation to ensure persistence, transactions, and security are applied properly to every operation a client performs on the bean. The container manages security, transactions, and persistence automatically for the bean, so the bean developer doesn't have to write this type of logic into the bean code itself. The enterprise bean developer can focus on encapsulating business rules, while the container takes care of everything else.



EJB Containers manage enterprise beans at runtime

Containers will manage many beans simultaneously in the same fashion that the Java WebServer manages many servlets. To reduce memory consumption and processing, containers pool resources and manage the life cycles of all the beans very carefully. When a bean is not being used, a container will place it in a pool to be reused by another client, or possibly evict it from memory and only bring it back when it's needed. Because client applications don't have direct access to the beans -- the container lies between the client and bean -- the client application is completely unaware of the container's resource

management activities. A bean that is not in use, for example, might be evicted from memory on the server, while its remote reference on the client remains intact. When the client invokes a method on the remote reference, the container simply re-incarnates the bean to service the request. The client application is unaware of the entire process.

An enterprise bean depends on the container for everything it needs. If an enterprise bean needs to access a JDBC connection or another enterprise bean, it does so through the container; if an enterprise bean needs to access the identity of its caller, obtain a reference to itself, or access properties it does so through the container. The enterprise bean interacts with its container through one of three mechanisms: callback methods, the `EJBContext` interface, or JNDI.

* Callback Methods: Every bean implements a subtype of the `EnterpriseBean` interface which defines several methods, called callback methods. Each callback method alerts the bean of a different event in its lifecycle and the container will invoke these methods to notify the bean when it's about to pool the bean, persist its state to the database, end a transaction, remove the bean from memory, and so on. The callback methods give the bean a chance to do some housework immediately before or after some event. Callback methods are discussed in more detail in Section 4.
* EJBContext: Every bean obtains an `EJBContext` object, which is a reference directly to the container. The `EJBContext` interface provides methods for interacting with the container so that that bean can request information about its environment, like the identity of its client or the status of a transaction, or can obtain remote references to itself.
* Java Naming and Directory Interface (JNDI): JNDI is a standard extension to the Java platform for accessing naming systems like LDAP, NetWare, file systems, etc. Every bean automatically has access to a special naming system called the Environment Naming Context (ENC). The ENC is managed by the container and accessed by beans using JNDI. The JNDI ENC allows a bean to access resources like JDBC connections, other enterprise beans, and properties specific to that bean.

The EJB specification defines a bean-container contract, which includes the mechanisms (callbacks, `EJBContext`, JNDI ENC) described above as well as a strict set of rules that describe how enterprise beans and their containers will behave at run time, how security access is checked, how transactions are managed, how persistence is applied, and so on. The bean-container contract is designed to make enterprise beans portable between EJB containers so that enterprise beans can be developed once, then run in any EJB container. Vendors like BEA, IBM, and GemStone sell application servers that include EJB containers. Ideally, any enterprise bean that conforms to the specification should be able to run in any conformant EJB container.

Portability is central to the value that EJB brings to the table. Portability ensures that a bean developed for one container can be migrated to another if another brand offers more performance, features, or savings. Portability also means that the bean developer's skills can be leveraged across several EJB container brands, providing organizations and developers with better opportunities.

In addition to portability, the simplicity of the EJB programming model makes EJB valuable. Because the container takes care of managing complex tasks like security, transactions, persistence, concurrency and resource management the bean developer is free to focus attention on business rules and a very simple programming model. A simple programming model means that beans can be developed faster without requiring a Ph.D. in distributed objects, transactions and other enterprise systems. EJB brings transaction processing and

distributed objects development into the mainstream.

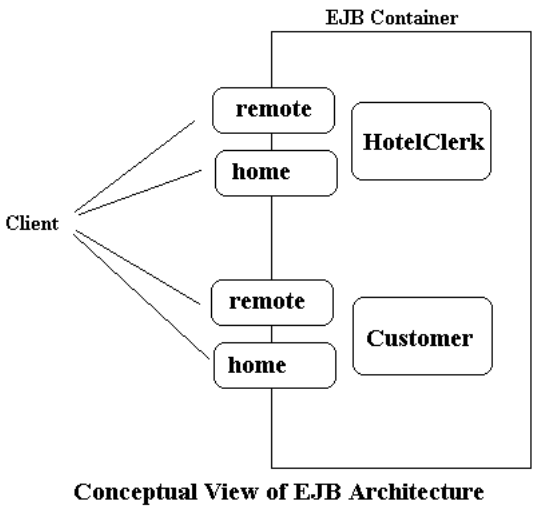
Exercise

Exercise 1. How to install and configure Sun's J2EE Reference Implementation: Tasks on page 40

Enterprise beans

To create an EJB server-side component, an enterprise bean developer provides two interfaces that define a bean's business methods, plus the actual bean implementation class. The client then uses a bean's public interfaces to create, manipulate, and remove beans from the EJB server. The implementation class, to be called the bean class, is instantiated at run time and becomes a distributed object.

Enterprise beans live in an EJB container and are accessed by client applications over the network through their remote and home interfaces. The remote and home interfaces expose the capabilities of the bean and provide all the methods needed to create, update, interact with, and delete the bean. A bean is a server-side component that represents a business concept like a Customer or a HotelClerk.



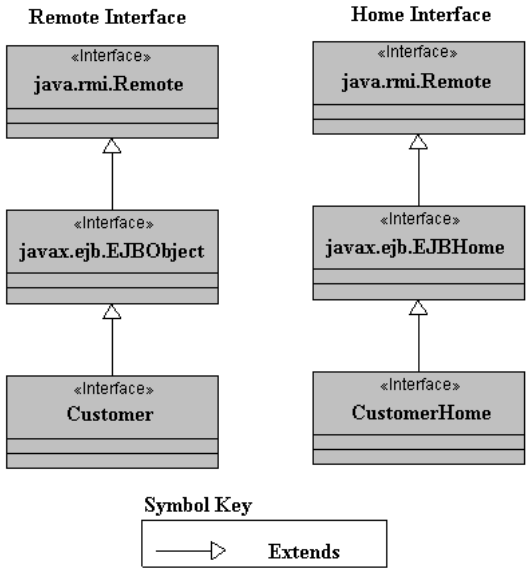
Conceptual View of EJB Architecture

Remote and home interfaces

The remote and home interfaces represent the bean, but the container insulates the beans from direct access from client applications. Every time a bean is requested, created, or deleted, the container manages the whole process.

The home interface represents the life-cycle methods of the component (create, destroy, find) while the remote interface represents the business method of the bean. The remote and home interfaces extend the `javax.ejb.EJBObject` and `javax.ejb.EJBHome` interfaces respectively. These EJB interface types define a standard set of utility methods

and provide common base types for all remote and home interfaces.



Class Diagram of Remote and Home Interfaces

Clients use the bean's home interface to obtain references to the bean's remote interface. The remote interface defines the business methods like accessor and mutator methods for changing a customer's name, or business methods that perform tasks like using the HotelClerk bean to reserve a room at a hotel. Below is an example of how a Customer bean might be accessed from a client application. In this case the home interface is the `CustomerHome` type and the remote interface is the `Customer` type.

```
CustomerHome home = // ... obtain a reference that
                    // implements the home interface.

// Use the home interface to create a
// new instance of the Customer bean.
Customer customer = home.create(customerID);

// using a business method on the Customer.
customer.setName(someName);
```

The remote interface defines the business methods of a bean, the methods that are specific to the business concept it represents. Remote interfaces are subclassed from the *`javax.ejb.EJBObject`* interface, which is a subclass of the *`java.rmi.Remote`* interface. The importance of the remote interfaces inheritance hierarchy is discussed later in Section 4. For now, focus on the business methods and their meaning. Below is the definition of a remote interface for a Customer bean:

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Customer extends EJBObject {

    public Name getName() throws RemoteException;
    public void setName(Name name) throws RemoteException;
    public Address getAddress() throws RemoteException;
    public void setAddress(Address address) throws RemoteException;

}
```

The remote interface defines accessor and mutator methods to read and update information about a business concept. This is typical of a type of bean called an entity bean, which represents a persistent business object (business objects whose data is stored in a database). Entity beans represent *business data* in the database and add behavior specific to that data.

Business methods

Business methods can also represent *tasks* that a bean performs. Although entity beans often have task-oriented methods, tasks are more typical of a type of bean called a session bean. Session beans do not represent data like entity beans. They represent business processes or agents that perform a service, like making a reservation at a hotel. Below is the definition of the remote interface for a HotelClerk bean, which is a type of session bean:

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface HotelClerk extends EJBObject {

    public void reserveRoom(Customer cust, RoomInfo ri,
                            Date from,     Date to)
                throws RemoteException;

    public RoomInfo availableRooms(Location loc, Date from, Date to)
                    throws RemoteException;
}
```

The business methods defined in the `HotelClerk` remote interface represent processes rather than simple accessors. The HotelClerk bean acts as an agent in the sense that it performs tasks on behalf of the user, but is not itself persistent in the database. You don't need information about the HotelClerk; you need the hotel clerk to perform tasks for you. This is typical behavior for a session bean.

There are two basic types of enterprise beans: entity beans, which represent data in a database, and session beans, which represent processes or act as agents performing tasks. As you build an EJB application you will create many enterprise beans, each representing a different business concept. Each business concept will be manifested as either an entity bean or a session bean. You will choose which type of bean a business concept becomes based on how it is intended to be used.

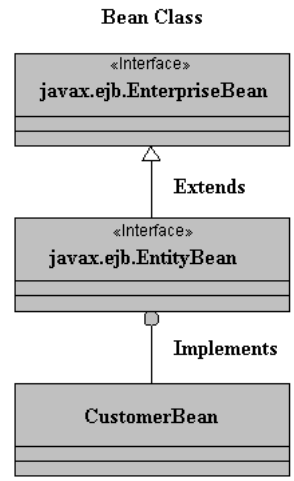
Entity beans

For every remote interface there is an implementation class, a business object that actually implements the business methods defined in the remote interface. This is the *bean class*; the key element of the bean. Below is a partial definition of the Customer bean class:

```
import javax.ejb.EntityBean;

public class CustomerBean implements EntityBean {

    Address    myAddress;
    Name       myName;
    CreditCard myCreditCard;

    public Name getName() {
```

```
        return myName;
    }

    public void setName(Name name) {
        myName = name;
    }

    public Address getAddress() {
        return myAddress;
    }

    public void setAddress(Address address) {
        myAddress = address;
    }

    ...
}
```

`CustomerBean` is the implementation class. It holds the data and provides accessor methods and other business methods. As an entity bean, the CustomerBean provides an object view of customer data. Instead of writing database access logic in an application, the application can simply use the remote interface to the Customer bean to access customer data. Entity beans implement the *`javax.ejb.EntityBean`* type, which defines a set of notification methods that the bean uses to interact with its container. These notification methods are examined in detail under "Callback methods" in Section 4.



Class Diagram for the Bean Class

Session beans

The HotelClerk bean is a session bean, which is similar in many respects to an entity bean. Session beans represent a set of processes or tasks, which are performed on behalf of the client application. Session beans may use other beans to perform a task or access the database directly. A little bit of code shows a session bean doing both. The `reserveRoom()` method shown below uses several other beans to accomplish a task, while the `availableRooms()` method uses JDBC to access the database directly:

```
import javax.ejb.SessionBean;

public class HotelClerkBean implements SessionBean {

    public void reserveRoom(Customer cust, RoomInfo ri,
```

```
                          Date from,     Date to) {
        CreditCard card = cust.getCreditCard();
        RoomHome roomHome = // ... get home reference
        Room room = roomHome.findByPrimaryKey(ri.getID());
        double amount = room.getPrice(from,to);
        CreditServiceHome creditHome = // ... get home reference
        CreditService creditAgent = creditHome.create();
        creditAgent.verify(card, amount);
        ReservationHome resHome = // ... get home reference
        Reservation reservation = resHome.create(cust,room,from,to);
    }

    public RoomInfo[] availableRooms(Location loc,
                                     Date from, Date to) {
        // Make an SQL call to find available rooms
        Connection con = // ... get database connection
        Statement stmt = con.createStatement();
        ResultSet results = stmt.executeQuery("SELECT ...");
        ...
        return roomInfoArray;
    }
}
```

You may have noticed that the bean classes defined above do not implement the remote or home interfaces. EJB doesn't require that the bean class implement these interfaces; in fact it's discouraged because the base types of the remote and home interfaces (`EJBObject` and `EJBHome`) define a lot of other methods that are implemented by the container automatically. The bean class does, however, provide implementations for all the business methods defined in the remote interface as well as methods. Callback methods are discussed in more detail in Section 4.

Life-cycle methods

In addition to a remote interface, all beans have a home interface. The home interface provides life-cycle methods for creating, destroying, and locating beans. These life-cycle behaviors are separated out of the remote interface because they represent behaviors that are not specific to a single bean instance. Below is the definition of the home interface for the Customer bean. Notice that it extends the *`javax.ejb.EJBHome`* interface, which extends the *`java.rmi.Remote`* interface.

```
import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import java.rmi.RemoteException;

public interface CustomerHome extends EJBHome {

    public Customer create(Integer customerNumber)
                   throws RemoteException, CreateException;

    public Customer findByPrimaryKey(Integer customerNumber)
                   throws RemoteException, FinderException;

    public Enumeration findByZipCode(int zipCode)
                      throws RemoteException, FinderException;
}
```

The `create()` method is used to create a new entity. This will result in a new record in the database. A home may have many `create()` methods. The number and datatype of the arguments of each `create()` are left up to the bean developer, but the return type must be the remote interface datatype. In this case, invoking `create()` on the `CustomerHome`

interface will return an instance of `Customer`. The `findByPrimaryKey()` and `findByZipCode()` methods are used to locate specific instance of the customer bean. Again, you may define as many find methods as you need.

Back to the remote and home interfaces

The remote and home interfaces are used by applications to access enterprise beans at run time. The home interface allows the application to create or locate the bean, while the remote interface allows the application to invoke a bean's business methods, as shown here:

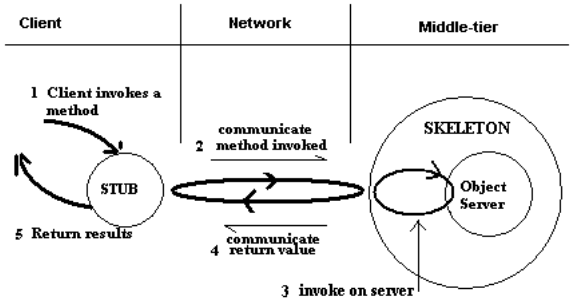
```
CustomerHome home = // Get a reference to the CustomerHome object

Customer customer = home.create(new Integer(33));

Name name = new Name("Richard", "Wayne", "Monson-Haefel");
customer.setName(name);

Enumeration enumOfCustomers = home.findByZip(55410);

Customer customer2 = home.findByPrimaryKey(new Integer(33));

Name name2 = customer2.getName();

// output is "Richard Wayne Monson-Haefel"
System.out.println(name);
```

The *`javax.ejb.EJBHome`* interface also defines other methods that the `CustomerBean` automatically inherits, including a set of `remove()` methods that allow the application to destroy bean instances.

Enterprise beans as distributed objects

The remote and home interfaces are types of Java RMI Remote interfaces. The *`java.rmi.Remote`* interface is used by distributed objects to represent the bean in a different address space (process or machine). An enterprise bean is a distributed object. That means that the bean class is instantiated and lives in the container, but it can be accessed by applications that live in other address spaces.

To make an object instance in one address space available in another requires a little trick involving network sockets. To make the trick work, wrap the instance in a special object called a skeleton that has a network connection to another special object called a stub. The stub implements the remote interface so it looks like a business object. But the stub doesn't contain business logic; it holds a network socket connection to the skeleton. Every time a business method is invoked on the stub's remote interface, the stub sends a network message to the skeleton telling it which method was invoked. When the skeleton receives a network message from the stub, it identifies the method invoked and the arguments, and then invokes the corresponding method on the actual instance. The instance executes the business method and returns the result to the skeleton, which sends it to the stub. The diagram below illustrates this concept:



The stub returns the result to the application that invoked its remote interface method. From the perspective of the application using the stub, it looks like the stub does the work locally. Actually, the stub is just a dumb network object that sends the requests across the network to the skeleton, which in turn invokes the method on the actual instance. The instance does all the work; the stub and skeleton just pass the method identity and arguments back and forth across the network.

In EJB, the skeletons for the remote and home interfaces are implemented by the container, not the bean class. This is to ensure that every method invoked on these reference types by a client application are first handled by the container and then delegated to the bean instance. The container must intercept these requests intended for the bean so that it can apply persistence (entity beans), transactions, and access control automatically.

Distributed object protocols define the format of network messages sent between address spaces. Distributed object protocols get pretty complicated, but luckily you don't see any of it because it's handled automatically. Most EJB servers support either the Java Remote Method Protocol (JRMP) or CORBA's Internet Inter-ORB Protocol (IIOP). The bean and application programmer only see the bean class and its remote interface; the details of the network communication are hidden.

With respect to the EJB API, the programmer doesn't care whether the EJB server uses JRMP or IIOP -- the API is the same. The EJB specification requires that you use a specialized version of the Java RMI API, when working with a bean remotely. Java RMI is an API for accessing distributed objects and is somewhat protocol agnostic -- in the same way that JDBC is database agnostic. So, an EJB server can support JRMP or IIOP, but the bean and application developer always uses the same Java RMI API. In order for the EJB server to have the option of supporting IIOP, a specialized version of Java RMI, called Java RMI-IIOP was developed. Java RMI-IIOP uses IIOP as the protocol and the Java RMI API. EJB servers don't have to use IIOP, but they do have to respect Java RMI-IIOP restrictions, so EJB 1.1 uses the specialized Java RMI-IIOP conventions and types, but the underlying protocol can be anything.

Section 4. Entity beans

Introduction

The entity bean is one of two primary bean types: entity and session. The entity bean is used to represent data in the database. It provides an object-oriented interface to data that would normally be accessed by the JDBC or some other back-end API. More than that, entity beans provide a component model that allows bean developers to focus their attention on the business logic of the bean, while the container takes care of managing persistence, transactions, and access control.

There are two basic kinds of entity beans: container-managed persistence (CMP) and bean-managed persistence (BMP). With CMP, the container manages the persistence of the entity bean. Vendor tools are used to map the entity fields to the database and absolutely no database access code is written in the bean class. With BMP, the entity bean contains database access code (usually JDBC) and is responsible for reading and writing its own state to the database. BMP entities have a lot of help with this since the container will alert the bean as to when it's necessary to make an update or read its state from the database. The container can also handle any locking or transactions, so that the database maintains integrity.

Container-managed persistence

Container-managed persistence beans are the simplest for the bean developer to create and the most difficult for the EJB server to support. This is because all the logic for synchronizing the bean's state with the database is handled automatically by the container. This means that the bean developer doesn't need to write any data access logic, while the EJB server is supposed to take care of all the persistence needs automatically -- a tall order for any vendor. Most EJB vendors support automatic persistence to a relational database, but the level of support varies. Some provide very sophisticated object-to-relational mapping, while others are very limited.

In this panel, you will expand the CustomerBean developed earlier to a complete definition of a Container-managed persistence bean. In the panel on bean-managed persistence, you will modify the CustomerBean to manage its own persistence.

Bean class

An enterprise bean is a complete component that is made up of at least two interfaces and a bean implementation class. All these types will be presented and their meaning and application explained, starting with the bean class, which is defined below:

```
import javax.ejb.EntityBean;

public class CustomerBean implements EntityBean {

    int        customerID;
    Address    myAddress;
    Name       myName;
    CreditCard myCreditCard;

    // CREATION METHODS
```

```
    public Integer ejbCreate(Integer id) {
        customerID = id.intValue();
        return null;
    }

    public void ejbPostCreate(Integer id) {
    }

    public Customer ejbCreate(Integer id, Name name) {
        myName = name;
        return ejbCreate(id);
    }

    public void ejbPostCreate(Integer id, Name name) {
    }

    // BUSINESS METHODS
    public Name getName() {
        return myName;
    }

    public void setName(Name name) {
        myName = name;
    }

    public Address getAddress() {
        return myAddress;
    }

    public void setAddress(Address address) {
        myAddress = address;
    }

    public CreditCard getCreditCard() {
        return myCreditCard;
    }

    public void setCreditCard(CreditCard card) {
        myCreditCard = card;
    }

    // CALLBACK METHODS
    public void setEntityContext(EntityContext cntx) {
    }

    public void unsetEntityContext() {
    }

    public void ejbLoad() {
    }

    public void ejbStore() {
    }

    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    public void ejbRemove() {
    }
}
```

This is a good example of a fairly simple CMP entity bean. Notice that there is no database access logic in the bean. This is because the EJB vendor provides tools for mapping the fields in the CustomerBean to the database. The `CustomerBean` class, for example, could be mapped to any database providing it contains data that is similar to the fields in the bean. In this case, the bean's instance fields are composed of a primitive `int` and simple dependent objects (`Name`, `Address`,and `CreditCard`) with their own attributes. Below are the definitions for these dependent objects:

```
// The Name class
public class Name implements Serializable {

    public String lastName, firstName, middleName;

    public Name(String lastName, String firstName,
                String middleName) {
        this.lastName   = lastName;
        this.firstName  = firstName;
        this.middleName = middleName;
    }

    public Name() {}
}

// The Address class
public class Address implements Serializable {

    public String street, city, state, zip;

    public Address(String street, String city,
                   String state, String zip) {
        this.street = street;
        this.city   = city;
        this.state  = state;
        this.zip    = zip;
    }

    public Address() {}
}

// The CreditCard class
public class CreditCard implements Serializable {

    public String number, type, name;
    public Date expDate;

    public CreditCard(String number, String type,
                      String name, Date expDate) {
        this.number  = number;
        this.type    = type;
        this.name    = name;
        this.expDate = expDate;
    }

    public CreditCard() {}
}
```

These fields are called *container-managed fields* because the container is responsible for synchronizing their state with the database; the container manages the fields. Container-managed fields can be any primitive data types or serializable type. This case uses both a primitive `int` (`customerID`) and serializable objects (Address, Name, CreditCard). To map the dependent objects to the database, a fairly sophisticated mapping tool would be needed. Not all fields in a bean are automatically container-managed fields; some may be just plain instance fields for the bean's transient use. A bean developer distinguishes container-managed fields from plain instance fields by indicating which fields are container-managed in the deployment descriptor.

The container-managed fields must have corresponding types (columns in RDBMS) in the database either directly or through object-relational mapping. The CustomerBean might, for example, map to a CUSTOMER table in the database that has the following definition:

```
CREATE TABLE CUSTOMER
{
    id            INTEGER PRIMARY KEY,
    last_name     CHAR(30),
    first_name    CHAR(20),
    middle_name   CHAR(20),
```

```
    street        CHAR(50),
    city          CHAR(20),
    state         CHAR(2),
    zip           CHAR(9),
    credit_number CHAR(20),
    credit_date   DATE,
    credit_name   CHAR(20),
    credit_type   CHAR(10)
}
```

With container-managed persistence, the vendor must have some kind of proprietary tool that can map the bean's container-managed fields to their corresponding columns in a specific table, CUSTOMER in this case.

Once the bean's fields are mapped to the database, and the Customer bean is deployed, the container will manage creating records, loading records, updating records, and deleting records in the CUSTOMER table in response to methods invoked on the Customer bean's remote and home interfaces.

A subset (one or more) of the container-managed fields will also be identified as the bean's primary key. The primary key is the index or pointer to a unique record(s) in the database that makes up the state of the bean. In the case of the CustomerBean, the `id` field is the primary key field and will be used to locate the bean's data in the database. Primitive single field primary keys are represented as their corresponding object wrappers. The primary key of the Customer bean for example is a primitive `int` in the bean class, but to a bean's clients it will manifest itself as the *`java.lang.Integer`* type. Primary keys that are made up of several fields, called compound primary keys, will be represented by a special class defined by the bean developer. Primary keys are similar in concept to primary keys in a relational database -- actually when a relational database is used for persistence, they are often the same thing.

Home interface

To create a new instance of a CMP entity bean, and therefore insert data into the database, the `create()` method on the bean's home interface must be invoked. The Customer bean's home interface is defined by the `CustomerHome` interface; the definition is shown below:

```
public interface CustomerHome extends javax.ejb.EJBHome {

    public Customer create(Integer customerNumber)
                   throws RemoteException,CreateException;

    public Customer create(Integer customerNumber, Name name)
                   throws RemoteException,CreateException;

    public Customer findByPrimaryKey(Integer customerNumber)
                   throws RemoteException, FinderException;

    public Enumeration findByZipCode(int zipCode)
                      throws RemoteException, FinderException;
}
```

Below is an example of how the home interface would be used by an application client to create a new customer:

```
CustomerHome home = // Get a reference to the CustomerHome object

Name name = new Name("John", "W", "Smith");
```

```
Customer customer = home.create(new Integer(33), name);
```

A bean's home interface may declare zero or more `create()` methods, each of which must have corresponding `ejbCreate()` and `ejbPostCreate()` methods in the bean class. These creation methods are linked at run time, so that when a `create()` method is invoked on the home interface, the container delegates the invocation to the corresponding `ejbCreate()` and `ejbPostCreate()` methods on the bean class.

When the `create()` method on a home interface is invoked, the container delegates the `create()` method call to the bean instance's matching `ejbCreate()` method. The `ejbCreate()` methods are used to initialize the instance state before a record is inserted into the database. In this case, they initialize the customerID and `Name` fields. When the `ejbCreate()` method is finished (they return `null` in CMP) the container reads the container-managed fields and inserts a new record into the CUSTOMER table indexed by the primary key, in this case `customerID` as it maps to the `CUSTOMER.ID` column.

In EJB, an entity bean doesn't technically exist until after its data has been inserted into the database, which occurs during the `ejbCreate()` method. Once the data has been inserted, the entity bean exists and can access its own primary key and remote references, which isn't possible until after the `ejbCreate()` method completes and the data is in the database. If a bean needs to access its own primary key or remote reference after it's created, but before it services any business methods, it can do so in the `ejbPostCreate()` method. The `ejbPostCreate()` allows the bean to do any post-create processing before it begins serving client requests. For every `ejbCreate()` there must be a matching (matching arguments) `ejbPostCreate()` method.

The methods in the home interface that begin with "find" are called the find methods. These are used to query the EJB server for specific entity beans, based on the name of the method and arguments passed. Unfortunately, there is no standard query language defined for find methods, so each vendor will implement the find method differently. In CMP entity beans, the find methods are not implemented with matching methods in the bean class; containers implement them when the bean is deployed in a vendor specific manner. The deployer will use vendor specific tools to tell the container how a particular find method should behave. Some vendors will use object-relational mapping tools to define the behavior of a find method while others will simply require the deployer to enter the appropriate SQL command.

There are two basic kinds of find methods: single-entity and multi-entity. Single-entity find methods return a remote reference to the one specific entity bean that matches the find request. If no entity beans are found, the method throws an *`ObjectNotFoundException`* . Every entity bean **must** define the single-entity find method with the method name `findByPrimaryKey()`, which takes the bean's primary key type as an argument. (In the above example, you used the `Integer` type, which wrapped the `int` type of the `id` field in the bean class.) The multi-entity find methods return a collection (*`Enumeration`* or *`Collection`* type) of entities that match the find request. If no entities are found, the multi-entity find returns an empty collection. (Note that an empty collection is *not* the same thing as a null reference.)

Remote interface

Every entity bean must define a remote interface in addition to the home interface. The remote interface defines the business methods of the entity bean. The following is the remote

interface definition for the Customer bean:

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Customer extends EJBObject {

    public Name getName()
                throws RemoteException;

    public void setName(Name name)
                throws RemoteException;

    public Address getAddress()
                   throws RemoteException;

    public void setAddress(Address address)
                throws RemoteException;

    public CreditCard getCreditCard()
                     throws RemoteException;

    public void setCreditCard(CreditCard card)
                throws RemoteException;
}
```

Below is an example of how a client application would use the remote interface to interact with a bean:

```
Customer customer = // ... obtain a remote reference to the bean

// get the customer's address
Address addr = customer.getAddress();

// change the zip code
addr.zip = "56777";

// update the customer's address
customer.setAddress(addr);
```

The business methods in the remote interface are delegated to the matching business methods in the bean instance. In the Customer bean, the business methods are all simple accessors and mutators, but they could be more complicated. In other words, an entity's business methods are not limited to reading and writing data; they can also perform tasks and computations.

If customers had, for example, a loyalty program that rewarded frequent use, there might be methods to upgrade membership in the program based on an accumulation of overnight stays. See below:

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Customer extends EJBObject {

    public MembershipLevel addNights(int nights_stayed)
                           throws RemoteException;

    public MembershipLevel upgradeMembership()
                           throws RemoteException;

    public MembershipLevel getMembershipLevel()
                           throws RemoteException;

    ... Simple accessor business methods go here
}
```

The `addNights()` and `upgradeMembership()` methods are more sophisticated than simple accessor methods. They apply business rules to change the membership level and go beyond reading and writing data.

Callback methods

The bean class defines create methods that match methods in the home interface and business methods that match methods in the remote interface. The bean class also implements a set of *callback* methods that allow the container to notify the bean of events in its life cycle. The callback methods are defined in the *`javax.ejb.EntityBean`* interface that is implemented by all entity beans, including the `CustomerBean` class. The `EntityBean` interface has the following definition. Notice that the bean class implements these methods.

```
public interface javax.ejb.EntityBean {

    public void setEntityContext();

    public void unsetEntityContext();

    public void ejbLoad();

    public void ejbStore();

    public void ejbActivate();

    public void ejbPassivate();

    public void ejbRemove();
}
```

The `setEntityContext()` method provides the bean with an interface to the container called the `EntityContext`. The `EntityContext` interface contains methods for obtaining information about the context under which the bean is operating at any particular moment. The `EntityContext` interface is used to access security information about the caller; to determine the status of the current transaction or to force a transaction rollback; or to get a reference to the bean itself, its home, or its primary key. The `EntityContext` is set only once in the life of an entity bean instance, so its reference should be put into one of the bean instance's fields if it will be needed later.

The Customer bean above doesn't use the `EntityContext`, but it could. For example, it could use the `EntityContext` to validate the caller's membership in a particular security role. Below is an example, where the `EntityContext` is used to validate that the caller is a `Manager`, the only role (security identity) that can set the credit card type on a customer to be a `WorldWide` card, the no-limit card of the super wealthy. (Customers with this card are automatically tagged for extra service.)

```
import javax.ejb.EntityBean;

public class CustomerBean implements EntityBean {

    int           customerID;
    Address       myAddress;
    Name          myName;
    CreditCard    myCreditCard;
    EntityContext ejbContext;
```

```
    // CALLBACK METHODS
    public void setEntityContext(EntityContext cntx) {
        ejbContext = cntx
    }

    public void unsetEntityContext() {
        ejbContext = null;
    }

    // BUSINESS METHODS
    public void setCreditCard(CreditCard card) {
        if (card.type.equals("WorldWide"))
            if (ejbContext.isCallerInRole("Manager"))
                myCreditCard = card;
            else
                throw new SecurityException();
        else
            myCreditCard = card;
    }

    public CreditCard getCreditCard() {
        return myCreditCard;
    }
    ...
}
```

The `unsetEntityContext()` method is used at the end of the bean's life cycle -- before the instance is evicted from memory -- to dereference the `EntityContext` and perform any last-minute clean-up.

The `ejbLoad()` and `ejbStore()` methods in CMP entities are invoked when the entity bean's state is being synchronized with the database. The `ejbLoad()` is invoked just after the container has refreshed the bean container-managed fields with its state from the database. The `ejbStore()` method is invoked just before the container is about to write the bean container-managed fields to the database. These methods are used to modify data as it's being synchronized. This is common when the data stored in the database is different than the data used in the bean fields. The methods might be used, for example, to compress data before it is stored and decompress it when it is retrieved from the database.

In the Customer bean, the `ejbLoad()` and `ejbStore()` methods might be used to convert the dependent objects (`Name`, `Address`, `CreditCard`) to simple `String` objects and primitive types, if the EJB container is not sophisticated enough to map the dependent objects to the CUSTOMER table. Below is an example of how the bean might be modified:

```
import javax.ejb.EntityBean;

public class CustomerBean implements EntityBean {

    //container-managed fields
    int    customerID;
    String lastName;
    String firstName;
    String middleName;
    ...

    // not-container-managed fields
    Name       myName;
    Address    myAddress;
    CreditCard myCreditCard;


    // BUSINESS METHODS
    public Name getName() {
        return myName;
    }
```

```
    public void setName(Name name) {
        myName = name;
    }
    ...

    public void ejbLoad() {

        if (myName == null)
            myName = new Name();

        myName.lastName = lastName;
        myName.firstName = firstName;
        myName.middleName = middleName;
        ...
    }

    public void ejbStore() {

        lastName   = myName.lastName;
        firstName  = myName.firstName;
        middleName = myName.middleName;
        ...
    }
}
```

The `ejbPassivate()` and `ejbActivate()` methods are invoked on the bean by the container just before the bean is passivated and just after the bean is activated, respectively. *Passivation* in entity beans means that the bean instance is disassociated with its remote reference so that the container can evict it from memory or reuse it. It's a resource conservation measure the container employs to reduce the number of instances in memory. A bean might be passivated if it hasn't been used for a while or as a normal operation performed by the container to maximize reuse of resources. Some containers will evict beans from memory, while others will reuse instances for other more active remote references. The `ejbPassivate()` and `ejbActivate()` methods provide the bean with a notification as to when it's about to be passivated (disassociated with the remote reference) or activated (associated with a remote reference).

Exercise

Exercise 2. How to create an entity bean: Tasks on page 43

Bean-managed persistence

The bean-managed persistence (BMP) enterprise bean manages synchronizing its state with the database as directed by the container. The bean uses a database API (usually JDBC) to read and write its fields to the database, but the container tells it when to do each synchronization operation and manages the transactions for the bean automatically. Bean-managed persistence gives the bean developer the flexibility to perform persistence operations that are too complicated for the container or to use a data source that is not supported by the container -- custom and legacy databases, for example.

In this panel, you'll modify the `CustomerBean` class to be a BMP bean instead of a CMP bean. This modification will not impact the remote or home interface at all. In fact, you won't modify the original `CustomerBean` directly. Instead, you'll change it to bean-managed persistence by extending the bean and overriding the appropriate methods. Below is the definition of the class that will extend the `Customer` bean class to make it a BMP entity. In most cases, you will not extend a bean to make it BMP, you will just implement the bean as

BMP directly. This strategy (extending the CMP bean) is done for two reasons: it allows the bean to be either a CMP or BMP bean; and it conveniently cuts down on the amount of code needed to display. Below is the definition of the BMP class that will be added to as this panel proceeds:

```
public class CustomerBean_BMP extends CustomerBean {
    public void ejbLoad() {
        // override implementation
    }
    public void ejbStore() {
        // override implementation
    }
    public void ejbCreate() {
        // override implementation
    }
    public void ejbRemove() {
        // override implementation
    }
    private Connection getConnection() {
        // new helper method
    }
}
```

With BMP beans, the `ejbLoad()` and `ejbStore()` methods are used differently by the container and bean than was the case in CMP. In BMP, the `ejbLoad()` and `ejbStore()` methods contain code to read the bean's data from the database and to write changes to the database, respectively. These methods are called on the bean automatically, when the EJB server decides it's a good time to read or write data.

The `CustomerBean_BMP` bean manages its own persistence. In other words, the `ejbLoad()` and `ejbStore()` methods must include database access logic, so that the bean can load and store its data when the EJB container tells it to. The container will execute the `ejbLoad()` and `ejbStore()` methods automatically when appropriate.

The `ejbLoad()` method is usually invoked by the container at the beginning of a transaction, just before the container delegates a business method to the bean. The code below shows how to implement the `ejbLoad()` method using JDBC:

```
import java.sql.Connection;

public class CustomerBean_BMP extends CustomerBean {

  public void ejbLoad() {
    Connection con;
    try {
      Integer primaryKey = (Integer)ejbContext.getPrimaryKey();
      con = this.getConnection();
      Statement sqlStmt =
      con.createStatement("SELECT * FROM Customer " +
                          " WHERE customerID = " +
                          primaryKey.intValue());
      ResultSet results = sqlStmt.executeQuery();
      if (results.next()) {
        // get the name information from the customer table
        myName = new Name();
        myName.first = results.getString("FIRST_NAME");
        myName.last = results.getString("LAST_NAME");
        myName.middle = results.getString("MIDDLE_NAME");
        // get the address information from the customer table
        myAddress = new Address();
        myAddress.street = results.getString("STREET");
        myAddress.city = results.getString("CITY");
        myAddress.state = results.getString("STATE");
        myAddress.zip = results.getInt("ZIP");
        // get the credit card information from the customer table
```

```
        myCreditCard = new CreditCard();
        myCreditCard.number = results.getString("CREDIT_NUMBER");
        myCreditCard.expDate = results.getString("CREDIT_DATE");
        myCreditCard.type = results.getString("CREDIT_TYPE");
        myAddress.name = results.getInt("CREDIT_NAME");
      }
    }
    catch (SQLException sqle) {
      throw new EJBException(sqle);
    }
    finally {
      if (con!=null)
        con.close();
    }
  }
}
```

In the `ejbLoad()` method, use the `ejbContext()` reference to the bean's `EntityContext` to obtain the instance's primary key. This approach ensures that you use the correct index to the database. Obviously, the `CustomerBean_BMP` will need to use the inherited `setEntityContext()` and `unsetEntityContext()` methods as illustrated earlier.

The `ejbStore()` method is invoked by the container on the bean, at the end of a transaction, just before the container attempts to commit all changes to the database.

```
import java.sql.Connection;

public class CustomerBean_BMP extends CustomerBean {

  public void ejbLoad() {
    ... read data from database
  }

  public void ejbStore() {
    Connection con;
    try {
      Integer primaryKey = (Integer)ejbContext.getPrimaryKey();
      con = this.getConnection();
      PreparedStatement sqlPrep =
      con.prepareStatement(
         "UPDATE customer set " +
         "last_name = ?, first_name = ?, middle_name = ?, " +
         "street = ?, city = ?, state = ?, zip = ?, " +
         "card_number = ?, card_date = ?, " +
         "card_name = ?, card_name = ?, " +
         "WHERE id = ?"
      );
      sqlPrep.setString(1,myName.last);
      sqlPrep.setString(2,myName.first);
      sqlPrep.setString(3,myName.middle);
      sqlPrep.setString(4,myAddress.street);
      sqlPrep.setString(5,myAddress.city);
      sqlPrep.setString(6,myAddress.state);
      sqlPrep.setString(7,myAddress.zip);
      sqlPrep.setInt(8, myCreditCard.number);
      sqlPrep.setString(9, myCreditCard.expDate);
      sqlPrep.setString(10, myCreditCard.type);
      sqlPrep.setString(11, myCreditCard.name);
      sqlPrep.setInt(12,primaryKey.intValue());
      sqlPrep.executeUpdate();
    }
    catch (SQLException sqle) {
      throw new EJBException(sqle);
    }
    finally {
      if (con!=null)
        con.close();
    }
  }
```

```
}
```

In both the `ejbLoad()` and `ejbStore()` methods the bean synchronizes its own state with the database using JDBC. If you studied the code carefully you may have noticed that the bean obtains its database connection from the mysterious `this.getConnection()` method invocation, a method yet to be implemented. The `getConnection()` method is not a standard EJB method; it's just a private helper method implemented to conceal the mechanics of obtaining a database connection. Below is the definition of the `getConnection()` method:

```
import java.sql.Connection;

public class CustomerBean_BMP extends CustomerBean {

  public void ejbLoad() {
    ... read data from database
  }

  public void ejbStore() {
    ... write data to database
  }

  private Connection getConnection() throws SQLException {

     InitialContext jndiContext = new InitialContext();
     DataSource source = (DataSource)
                jndiContext.lookup("java:comp/env/jdbc/myDatabase");
     return source.getConnection();
  }
}
```

Database connections are obtained from the container using a default JNDI context called the JNDI Environment Naming Context (ENC). The ENC provides access to transactional, pooled, JDBC connections through the standard connection factory, the *`javax.sql.DataSource`* type.

In BMP, the `ejbLoad()` and `ejbStore()` methods are invoked by the container to synchronize the bean instance with data in the database. To insert and remove entities in the database, the `ejbCreate()` and `ejbRemove()` methods are implemented with similar database access logic. The `ejbCreate()` methods are implemented so that a new record is inserted into the database and the `ejbRemove()` methods are implemented so that the entity's data is deleted from the database. The `ejbCreate()` methods and the `ejbRemove()` method of a BMP entity are invoked by the container in response to invocations on their corresponding methods in the home and remote interfaces. The implementations of these methods are shown below:

```
  public void ejbCreate(Integer id) {
    this.customerID = id.intValue();
    Connection con;
    try {
      con = this.getConnection();
      Statement sqlStmt =
      con.createStatement("INSERT INTO customer id VALUES (" +
                          customerID + ")");
      sqlStmt.executeUpdate();
      return id;
    }
    catch(SQLException sqle) {
      throw new EJBException(sqle);
    }
    finally {
      if (con!=null)
        con.close();
```

```
    }
  }

  public void ejbRemove() {
    Integer primaryKey = (Integer)ejbContext.getPrimaryKey();
    Connection con;
    try {
      con = this.getConnection();
      Statement sqlStmt =
      con.createStatement("DELETE FROM customer WHERE id = "
                          primaryKey.intValue());
      sqlStmt.executeUpdate();
    }
    catch(SQLException sqle) {
      throw new EJBException(sqle);
    }
    finally {
      if (con!=null)
        con.close();
    }
  }
```

In BMP, the bean class is responsible for implementing the find methods defined in the home interface. For each find method defined in the home interface there must be corresponding `ejbFind()` method in the bean class. The `ejbFind()` methods locate the appropriate bean records in the database and return their primary keys to the container. The container converts the primary keys into bean references and returns them to the client. Below is an example implementation of the `ejbFindByPrimaryKey()` method in the `CustomerBean_BMP` class, which corresponds to the `findByPrimaryKey()` defined in the `CustomerHome` interface:

```
public Integer ejbFindByPrimaryKey(Integer primaryKey)
               throws ObjectNotFoundException {
    Connection con;
    try {
        con = this.getConnection();
        Statement sqlStmt =
            con.createStatement("SELECT * FROM Customer " +
                                " WHERE customerID = " +
                                primaryKey.intValue());

        ResultSet results = sqlStmt.executeQuery();
        if (results.next())
            return primaryKey;
        else
            throw ObjectNotFoundException();
    }
    catch (SQLException sqle) {
        throw new EJBException(sqle);
    }
    finally {
        if (con!=null)
        con.close();
    }
}
```

Single-entity find methods like the one above return a single primary key or throw the *`ObjectNotFoundException`* if no matching record is located. Multi-entity find methods return a collection (*`java.util.Enumeration`* or *`java.util.Collection`*) of primary keys. The container converts the collection of primary keys into a collection of remote references, which are returned to the client. Below is an example of how the multi-entity `ejbFindByZipCode()` method, which corresponds to the `findByZipCode()` method defined in the `CustomerHome` interface, would be implemented in the `CustomerBean_BMP` class:

```
public Enumeration ejbFindByZipCode(int zipCode)  {
```

```
    Connection con;
    try {
        con = this.getConnection();
        Statement sqlStmt =
        con.createStatement("SELECT id FROM Customer " +
                            " WHERE zip = " +zipCode);

        ResultSet results = sqlStmt.executeQuery();
        Vector keys = new Vector();
        while(results.next()){
            int id = results.getInt("id");
            keys.addElement(new Integer(id));
        }
        return keys.elements();
    }
    catch (SQLException sqle) {
        throw new EJBException(sqle);
    }
    finally {
        if (con!=null)
        con.close();
    }
}
```

If no matching bean records are found, an empty collection is returned to the container, which returns an empty collection to the client.

With the implementation of all these methods and a few minor changes to the bean's deployment descriptor, the `CustomerBean_BMP` is ready to be deployed as a BMP entity.

Section 5. Session beans

Introduction

Session beans are used to manage the interactions of entity and other session beans, access resources, and generally perform tasks on behalf of the client. Session beans correspond to the controller in a model-view-controller architecture because they encapsulate the business logic of a three-tier architecture.

There are two basic kinds of session bean: stateless and stateful. Stateless session beans are made up of business methods that behave like procedures; they operate only on the arguments passed to them when they are invoked. Stateless beans are called stateless because they are transient; they do not maintain business state between method invocations. Stateful session beans encapsulate business logic and state specific to a client. Stateful beans are called "stateful" because they do maintain business state between method invocations, held in memory and not persistent.

Stateless session beans

Stateless session beans, like all session beans, are not persistent business objects as are entity beans. They do not represent data in the database. Instead, they represent business processes or tasks that are performed on behalf of the client using them. The business methods in a stateless bean act more like traditional procedures in a legacy transaction-processing monitor. Each invocation of a stateless business method is independent from previous invocations. Because stateless session beans are stateless, they are easier for the EJB container to manage, so they tend to process requests faster and use less resources. But this performance advantage comes at a price: stateless session beans are stupid. They don't remember anything from one method invocation to the next.

An example of a stateless session bean is a CreditService bean, representing a credit service that can validate and process credit card charges. A hotel chain might develop a CreditService bean to encapsulate the process of verifying a credit card number, making a charge, and recording the charge in the database for accounting purposes. Below are the remote and home interfaces for the CreditService bean:

```
// remote interface
public interface CreditService extends javax.ejb.EJBObject {
    public void verify(CreditCard card, double amount)
    throws RemoteException, CreditServiceException;
    public void charge(CreditCard card, double amount)
    throws RemoteException, CreditServiceException;
}

// home interface
public interface CreditServiceHome extends java.ejb.EJBHome {
     public CreditService create()
                          throws RemoteException, CreateException;
}
```

The remote interface, `CreditService`, defines two methods, `verify()` and `charge()`, which are used by the hotel to verify and charge credit cards. The hotel might use the `verify()` method to make a reservation, but not charge the customer. The `charge()` method would be used to charge a customer for a room. The home interface, `CreditServiceHome` provides one `create()` method with no arguments. All home

interfaces for stateless session beans will define just one method, a no-argument `create()` method, because session beans do not have find methods and they cannot be initiated with any arguments when they are created. Stateless session beans do not have find methods, because stateless beans are all equivalent and are not persistent. In other words, there is no unique stateless session beans that can be located in the database. Because stateless session beans are not persisted, they are transient services. Every client that uses the same type of session bean gets the same service.

Below is the bean class definition for the CreditService bean. This bean encapsulates access to the Acme Credit Card processing services. Specifically, this bean accesses the Acme secure Web server and posts requests to validate or charge the customer's credit card.

```
import javax.ejb.SessionBean;

public class CreditServiceBean implements SessionBean {
    URL acmeURL;
    HttpURLConnection acmeCon;

    public void ejbCreate() {
        try {
            InitialContext jndiContext = new InitialContext();
            URL acmeURL = (URL)
            jndiContext.lookup("java:comp/ejb/env/url/acme");
            acmeCon = acmeURL.openConnection();
        }
        catch (Exception e) {
            throws new EJBException(e);
        }
    }

    public void verify(CreditCard card, double amount) {
        String response = post("verify:" + card.postString() +
                               ":" + amount);
        if (response.substring("approved")== -1)
            throw new CreditServiceException("denied");
    }

    public void charge(CreditCard card, double amount)
                throws CreditCardException {
        String response = post("charge:" + card.postString() +
                               ":" + amount);
        if (response.substring("approved")== -1)
            throw new CreditServiceException("denied");
    }

    private String post(String request) {
        try {
            acmeCon.connect();
            acmeCon.setRequestMethod("POST "+request);
            String response = acmeCon.getResponseMessage();
        }
        catch (IOException ioe) {
            throw new EJBException(ioe);
        }
    }

    public void ejbRemove() {
        acmeCon.disconnect();
    }

    public void setSessionContext(SessionContext cntx) {}

    public void ejbActivate() {}

    public void ejbPassivate() {}
}
```

The CreditService stateless bean demonstrates that a stateless bean can represent a

collection of independent but related services. In this case, credit card validation and charges are related but not necessarily interdependent. Stateless beans might be used to access databases or unusual resources, as is the case with the CreditService bean, or to perform complex computations. The CreditService bean is used as an example in this tutorial to demonstrate the service nature of a stateless bean and to provide a context for discussing the behavior of the call back methods. The use of a stateless session beans is not limited to the behavior illustrated in this example; stateless beans can be used to perform any kind of service.

The CreditServiceBean class uses a URL resource factory (`acmeURL()`) to obtain and maintain a reference to the Acme Web server which exists on another computer very far away. The CreditServiceBean uses the `acmeURL()` to obtain a connection to the Web server and post requests for the validation and charge of credit cards. The CreditService bean is used by clients instead of a direct connection so that the service can be better managed by the EJB container, which will pool connections and managed transactions and security automatically for the EJB client.

The `ejbCreate()` method is invoked at the beginning of its lifetime and is invoked only once. The `ejbCreate()` method is a convenient method for initiating resource connections and variables that will be of use to the stateless bean for its lifetime. In the example above, the `CreditServiceBean` uses the `ejbCreate()` to obtain a reference to the `HttpURLConnection` factory, which it will use throughout its lifetime to obtain connections to the Acme Web server.

The `CreditServiceBean` uses the JNDI ENC to obtain a `URL` connection factory in the same way that the `CustomerBean` used the JNDI ENC to obtain a `DataSource` resource factory for JDBC connections. The JNDI ENC is a default JNDI context that all beans have access to automatically. The JNDI ENC is used to access static properties, other beans, and resource factories like the *`java.net.URL`* and JDBC *`javax.sql.DataSource`* . In addition, the JNDI ENC also provides access to JavaMail and Java Messaging Service resource factories.

The `ejbCreate()` and `ejbRemove()` methods are each only invoked once in its lifetime by the container; when the bean is first created and when it's finally destroyed. Invocations of the `create()` and `remove()` methods on its home and remote interfaces by the client do not result in invocations of the `ejbCreate()` and `ejbRemove()` methods on the bean instance. Instead, an invocation of the `create()` method provides the client with a reference to the stateless bean type and the `remove()` methods invalidate the reference. The container will decide when bean instances are actually created and destroyed and will invoke the `ejbCreate()` and `ejbRemove()` methods at these times. This allows stateless instances to be shared between many clients without impacting the clients' references.

In the `CreditServiceBean`, the `ejbCreate()` and `ejbRemove()` methods are used to obtain a URL connection at the beginning the bean instance's life and to disconnect from it at the end of the bean instance's life. Between the times that the URL connection is obtained and disconnected it is used by the business methods. This approach reduces the number of times that a connection needs to be obtained and disconnected, conserving resources. It may seem wasteful to maintain the URL connection between method invocations, but stateless session beans are designed to be shared between many clients, so that they are in constant use. As soon as a stateless instance completes a method invocation for a client, it can immediately service another client. Ideally, there is little down time for a stateless session bean instance, so it makes sense to maintain the URL connection.

The `verify()` and `charge()` methods delegate their requests to the `post()` method, a private helper method. The `post()` method uses an `HttpURLConnection` to submit the credit card information to the Acme Web server and return the reply to the `verify()` or `charge()` method. The `HttpURLConnection` may have been disconnected automatically by the container -- this might occur if, for example, a lot of time elapsed since its last use -- so the `post()` method always invokes the `connect()` method, which does nothing if the connection is already established. The `verify()` and `charge()` methods parse the return value looking for the substring "approved," which indicates that the credit card was not denied. If "approved" was not found, it's assumed that the card was denied and a business exception is thrown.

The `setSessionContext()` method provides the bean instance with a reference to the `SessionContext`, which serves the same purpose as the `EntityContext` did for the `CustomerBean` in the panel on . The `SessionContext` is not used in this example.

The `ejbActivate()` and `ejbPassivate()` methods are not implemented in the CreditService bean because passivation is not used in stateless session beans. These methods are defined in the *`javax.ejb.SessionBean`* interface for the stateful session beans, and so an empty implementation must be provided in stateless session beans. Stateless session bean will never provide anything but empty implementations of these methods.

Stateless session beans can also be used to access the database as well as coordinate the interaction of other beans to accomplish a task. Below is the definition of the HotelClerkBean shown earlier in this tutorial:

```
import javax.ejb.SessionBean;
import javax.naming.InitialContext;

public interface HotelClerkBean implements SessionBean {

    InitialContext jndiContext;

    public void ejbCreate() {}

    public void reserveRoom(Customer cust, RoomInfo ri,
                            Date from, Date to) {
        CreditCard card = cust.getCreditCard();

        RoomHome roomHome = (RoomHome)
        getHome("java:comp/env/ejb/RoomEJB", RoomHome.class);
        Room room = roomHome.findByPrimaryKey(ri.getID());
        double amount = room.getPrice(from,to);

        CreditServiceHome creditHome =
                          (CreditServiceHome) getHome(
                              "java:comp/env/ejb/CreditServiceEJB",
                              CreditServiceHome.class);
        CreditService creditAgent = creditHome.create();
        creditAgent.verify(card, amount);

        ReservationHome resHome =
                        (ReservationHome) getHome(
                              "java:comp/env/ejb/ReservationEJB",
                              ReservationHome.class);
        Reservation reservation = resHome.create(cust.getName(),
                                                 room,from,to);
    }

    public RoomInfo[] availableRooms(Location loc,
                                     Date from, Date to) {
        // do a SQL call to find available rooms
        Connection con = db.getConnection();
        Statement stmt = con.createStatement();
```

```
        ResultSet results = stmt.executeQuery("SELECT ...");
        ...
        return roomInfoArray;
    }

    private Object getHome(String path, Class type) {
        Object ref = jndiContext.lookup(path);
        return PortableRemoteObject.narrow(ref,type);
    }
}
```

The `HotelClerkBean` is also a stateless bean. All the information needed to process a reservation or to query a list of available rooms is obtained from the method arguments. In the `reserveRoom()` method, operations on several other beans (Room, CreditService, and Reservation) are coordinated to accomplish one larger task, reserving a room for a customer. This is an example of a session bean managing the interactions of other beans on behalf of the client. The `availableRooms()` method is used to query the database and obtain a list of rooms -- the information is returned to the client as a collection of data wrappers defined by the `RoomInfo` class. The use of this class, shown below, is a design pattern that provides the client with a light-weight wrapper of just the information needed:

```
public class RoomInfo {
    public int     id;
    public int     bedCount;
    public boolean smoking;
}
```

To obtain a reference to another bean, as is done three times in the `reserveRoom()` method, the private `getHome()` helper method is used. The `getHome()` method uses the JNDI ENC to obtain references to other beans:

```
private Object getHome(String path, Class type) {
   Object ref = jndiContext.lookup(path);
   return PortableRemoteObject.narrow(ref,type);
}
```

In the EJB 1.1 specification, RMI over IIOP is the specified programming model, so CORBA references types must be supported. CORBA references cannot be cast using Java language native casting. Instead the `PortableRemoteObject.narrow()` method must be used to explicitly narrow a reference from one type to its subtype. Because JNDI always returns an `Object` type, all bean references should be explicitly narrowed to support portability between containers.

Exercise

Exercise 3. How to create a stateless session bean: Tasks on page 45

Stateful session beans

Stateful session beans are dedicated to one client and maintain *conversational state* between method invocations. Unlike stateless session beans, clients do not share stateful beans. When a client creates a stateful bean, that bean instance is dedicated to service only that client. This makes it possible to maintain conversational state, which is business state that can be shared by methods in the same stateful bean.

As an example, the HotelClerk bean can be modified to be a stateful bean which can maintain conversational state between method invocations. This would be useful, for example, if you want the HotelClerk bean to be able to take many reservations, but then process them together under one credit card. This occurs frequently, when families need to reserve two or more rooms or when corporations reserve a block of rooms for some event. Below the `HotelClerkBean` is modified to be a stateful bean:

```
import javax.ejb.SessionBean;
import javax.naming.InitialContext;

public class HotelClerkBean implements SessionBean {

    InitialContext jndiContext;

    //conversational-state
    Customer cust;
    Vector resVector = new Vector();

    public void ejbCreate(Customer customer) {}
        cust = customer;
    }

    public void addReservation(Name name, RoomInfo ri,
                               Date from, Date to) {
        ReservationInfo resInfo =
                        new ReservationInfo(name,ri,from,to);
        resVector.addElement(resInfo);
    }

    public void reserveRooms() {
        CreditCard card = cust.getCreditCard();
        Enumeration resEnum = resVector.elements();

        while (resEnum.hasMoreElements()) {

        ReservationInfo resInfo =
                       (ReservationInfo) resEnum.nextElement();

        RoomHome roomHome = (RoomHome)
        getHome("java:comp/env/ejb/RoomEJB", RoomHome.class);
        Room room =
             roomHome.findByPrimaryKey(resInfo.roomInfo.getID());
        double amount = room.getPrice(resInfo.from,restInfo.to);

        CreditServiceHome creditHome = (CreditServiceHome)
        getHome("java:comp/env/ejb/CreditServiceEJB",
                CreditServiceHome.class);
        CreditService creditAgent = creditHome.create();
        creditAgent.verify(card, amount);

        ReservationHome resHome = (ReservationHome)
        getHome("java:comp/env/ejb/ReservationEJB",
                ReservationHome.class);
        Reservation reservation =
        resHome.create(resInfo.getName(),
        resInfo.roomInfo,resInfo.from,resInfo.to);
    }

    public RoomInfo[] availableRooms(Location loc,
                                     Date from, Date to) {
        // Make an SQL call to find available rooms
    }

   private Object getHome(String path, Class type) {
        Object ref = jndiContext.lookup(path);
        return PortableRemoteObject.narrow(ref,type);
   }
}
```

In the stateful version of the `HotelClerkBean` class, the conversational state is the

Customer reference, which is obtained when the bean is created, and the *Vector* of ReservationInfo objects. By maintaining the conversational state in the bean, the client is absolved of the responsibility of keeping track of this session state. The bean keeps track of the reservations and processes them in a batch when the serverRooms() method is invoked.

To conserve resources, stateful session beans may be passivated when they are not in use by the client. Passivation in stateful session beans is different than for entity beans. In stateful beans, passivation means the bean conversational-state is written to a secondary storage (often disk) and the instance is evicted from memory. The client's reference to the bean is not affected by passivation; it remains alive and usable while the bean is passivated. When the client invokes a method on a bean that is passivated, the container will activate the bean by instantiating a new instance and populating its conversational state with the state written to secondary storage. This passivation/activation process is often accomplished using simple Java serialization but it can be implemented in other proprietary ways as long as the mechanism behaves the same as normal serialization. (One exception to this is that transient fields do not need to be set to their default initial values when a bean is activated.)

Stateful session beans, unlike stateless beans, do use the ejbActivate() and ejbPassivate() methods. The container will invoke these methods to notify the bean when it's about to be passivated (ejbPassivate()) and immediately following activation ejbActivate()). Bean developers should use these methods to close open resources and to do other clean-up before the instance's state is written to secondary storage and evicted from memory.

The ejbRemove() method is invoked on the stateful instance when the client invokes the remove() method on the home or remote interface. The bean should use the ejbRemove() method to do the same kind of clean-up performed in the ejbPassivate() method.

Section 6. Deploying EJB technology

Introduction

As mentioned previously, the container handles persistence, transactions, concurrency, and access control automatically for the enterprise beans. The EJB specification describes a *declarative* mechanism for how these things will be handled, through the use of an XML deployment descriptor. When a bean is deployed into a container, the container reads the deployment descriptor to find out how transaction, persistence (entity beans), and access control should be handled. The person deploying the bean will use this information and specify additional information to hook the bean up to these facilities at run time.

A deployment descriptor has a predefined format that all EJB-compliant beans must use and all EJB-compliant servers must know how to read. This format is specified in an XML Document Type Definition, or DTD. The deployment descriptor describes the type of bean (session or entity) and the classes used for the remote, home, and bean class. It also specifies the transactional attributes of every method in the bean, which security roles can access each method (access control), and whether persistence in the entity beans is handled automatically or is performed by the bean. Below is an example of a XML deployment descriptor used to describe the Customer bean:

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">

<ejb-jar>
  <enterprise-beans>
    <entity>
      <description>
        This bean represents a customer
      </description>
      <ejb-name>CustomerBean</ejb-name>
      <home>CustomerHome</home>
      <remote>Customer</remote>
      <ejb-class>CustomerBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>Integer</prim-key-class>
      <reentrant>False</reentrant>

      <cmp-field><field-name>myAddress</field-name></cmp-field>
      <cmp-field><field-name>myName</field-name></cmp-field>
      <cmp-field><field-name>myCreditCard</field-name></cmp-field>
    </entity>
  </enterprise-beans>

  <assembly-descriptor>
    <security-role>
      <description>
        This role represents everyone who is allowed full access
        to the Customer bean.
      </description>
      <role-name>everyone</role-name>
    </security-role>

    <method-permission>
      <role-name>everyone</role-name>
      <method>
        <ejb-name>CustomerBean</ejb-name>
        <method-name>*</method-name>
      </method>
    </method-permission>

    <container-transaction>
      <description>
```

```
        All methods require a transaction
      </description>
      <method>
        <ejb-name>CustomerBean</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

EJB-capable application servers usually provide tools that can be used to build the deployment descriptors; this greatly simplifies the process.

When a bean is to be deployed, its remote, home, and bean class files and the XML deployment descriptor must be packaged into a jar file. The deployment descriptor must be stored in the jar under the special name `META-INF/ejb-jar.xml`. This jar file, called an ejb-jar, is vendor neutral; it can be deployed in any EJB container that supports the complete EJB specification. When a bean is deployed in an EJB container, its XML deployment descriptor is read from the jar to determine how to manage the bean at run time. The person deploying the bean will map attributes of the deployment descriptor to the container's environment. This will include mapping access security to the environment's security system, adding the bean to the EJB container's naming system, and so on. Once the bean developer has finished deploying the bean, it will become available for client applications and other beans to use.

Exercises

Exercise 4. How to set up the database: Tasks on page 47

Exercise 5. How to deploy enterprise beans in Sun's J2EE Reference Implementation: Tasks on page 49

Section 7. Enterprise JavaBeans clients

Introduction

Enterprise JavaBeans clients may be stand-alone applications, servlets, applets, or even other enterprise beans. For instance, the HotelClerk session bean shown earlier was a client of the Room entity beans. All clients use the server bean's home interface to obtain references to the server bean. This reference has the server bean's remote interface datatype, therefore the client interacts with the server bean solely through the methods defined in the remote interface.

The remote interface defines the business methods, such as accessor and mutator methods for changing a customer's name, or business methods that perform tasks like using the HotelClerk bean to reserve a room at a hotel. Below is an example of how a Customer bean might be accessed from a client application. In this case the home interface is `CustomerHome` and the remote interface is `Customer`.

```
CustomerHome home;
Object       ref;

// Obtain a reference to the CustomerHome
ref  = jndiContext.lookup("java:comp/env/ejb/Customer");

// Cast object returned by the JNDI lookup to the
// appropriate datatype
home = PortableRemoteObject.narrow(ref, CustomerHome.class);

// Use the home interface to create a
// new instance of the Customer bean.
Customer customer = home.create(customerID);

// Use a business method on the Customer.
customer.setName(someName);
```

A client first obtains a reference to the home interface by using JNDI ENC to look up the server beans. In EJB 1.1, Java RMI-IIOP is the specified programming model. As a consequence, all CORBA references types must be supported. CORBA references cannot be cast using Java native casting. Instead the `PortableRemoteObject.narrow()` method must be used to explicitly narrow a reference from one type to its subtype. Because JNDI always returns an *`Object`* type, all bean references should be explicitly narrowed to support portability between containers.

After the home interface is obtained, the client uses the methods defined on the home interface to create, find, or remove the server bean. Invoking one of the `create()` methods on the home interface returns to the client a remote reference to the server bean, which the client uses to perform its job.

Exercise

Exercise 6. How to create EJB clients: Tasks on page 55

Section 8. Exercises

About the exercises

These exercises are designed to provide help according to your needs. For example, you might simply complete the exercise given the information and the task list in the exercise body; you might want a few hints; or you may want a step-by-step guide to successfully complete a particular exercise. You can use as much or as little help as you need per exercise. Moreover, because complete solutions are also provided, you can skip a few exercises and still be able to complete future exercises requiring the skipped ones.

Each exercise has a list of any prerequisite exercises, a list of skeleton code for you to start with, links to necessary API pages, and a text description of the exercise goal. In addition, there is a help page, with help for each task, and a solution page with links to files that comprise a solution to the exercise.

Exercise 1. How to install and configure Sun's J2EE Reference Implementation: Tasks

This exercise steps you through the process of downloading and installing Sun's Java 2 Enterprise Edition Reference Implementation application server on your machine. This exercise is specific to the J2EE RI. If you want to use a different application server for the remainder of these exercises, then you should ensure it is EJB 1.1 compliant and install it now.

For more help with exercises, see About the exercises on page 40 .

Prerequisites

* None

Task 1: Check your system requirements to make sure you have an adequate hardware and software platform for installing and running J2EE RI.

Task 2: Ensure you have the Java 2 SDK version 1.2.2 or higher installed on your machine.

Task 3: Download the J2EE RI server and documentation from *Sun's J2EE download site* . The software distribution and *the documentation site* include installation instructions.

Task 4: Run the installer for both the J2EE and the documentation.

Task 5: Set the `J2EE_HOME` environment variable to point to the base directory where you installed the J2EE RI.

Task 6: Put `J2EE_HOME/bin` into your PATH environment variable.

Task 7: Start the Cloudscape database with the command

```
cloudscape -start
```

Task 8: Start the J2EE RI with the command

```
j2ee -verbose
```

Task 9: J2EE is now installed and running. Explore the J2EE Reference Implementation documentation within the *documentation site* to familiarize yourself more with J2EE.

Help for each of these tasks can be found on the next panel.

Exercise 1. How to install and configure Sun's J2EE Reference Implementation: Help

Task 1: Check your system requirements to make sure you have an adequate hardware and software platform for installing and running J2EE RI.

Help for task 1: System requirements are Windows NT 4.0 or Solaris 2.5.1+. Windows 2000 and Windows 98 also are reported to work, but Sun does not yet officially support them. A minimum of 128 MB of RAM is needed to adequately complete the exercises.

Visit Sun's Web page, which lists *J2EE Reference Implementation Tested Configurations* for more information.

Task 2: Ensure you have the Java 2 SDK version 1.2.2 or higher installed on your machine.

Help for task 2: The J2EE RI cannot run without the full Java 2 SDK installed. The J2EE RI cannot run if the HotSpot performance engine has been installed on top of the SDK; if HotSpot has been installed, be sure to uninstall then reinstall the Java 2 SDK.

Task 3: Download the J2EE RI server and documentation from *Sun's J2EE download site* . The software distribution and *the documentation site* include installation instructions.

Help for task 3: The Windows and Solaris distributions are each approximately 11 MB. The documentation is an additional 7 MB.

Task 4: Run the installer for both the J2EE and the documentation.

Help for task 4: Double click on the installer that you downloaded in the previous step.

Task 5: Set the `J2EE_HOME` environment variable to point to the base directory where you installed the J2EE RI.

Help for task 5: In Windows, this may be done by the command:

```
set J2EE_HOME=C:\j2sdkee
```

Make sure to point to the proper folder for your machine.

Task 6: Put `J2EE_HOME/bin` into your PATH environment variable.

Help for task 6: In Windows, this may be done by the command:

```
set PATH=%PATH%;C:\j2sdkee\bin
```

Make sure to point to the proper folder for your machine.

Task 7: Start the Cloudscape database with the command

```
cloudscape -start
```

Help for task 7: `cloudscape -start` should produce output similar to the following:

```
myhost> cloudscape -start
Wed Jan 12 11:51:20 PST 2000:
     [RmiJdbc] COM.cloudscape.core.JDBCDriver
        registered in DriverManager
Wed Jan 12 11:51:20 PST 2000:
     [RmiJdbc] Binding RmiJdbcServer...
Wed Jan 12 11:51:20 PST 2000:
     [RmiJdbc] No installation of RMI Security Manager...
Wed Jan 12 11:51:22 PST 2000:
     [RmiJdbc] RmiJdbcServer bound in rmi registry
```

Task 8: Start the J2EE RI with the command

```
j2ee -verbose
```

Help for task 8: Running the J2EE RI via the command-line command `j2ee -verbose` should produce output similar to the following:

```
myhost> j2ee -verbose

Naming service started: :1050
Published the configuration object ...
Binding DataSource, name = jdbc/Cloudscape,
    url = jdbc:cloudscape:rmi:CloudscapeDB;create=true
Configuring web service using "default"
Web service started: :9191
Web service started: :7000
Configuring web service using "default"
Configuring web service using
  "file:/D:/j2sdkee1.2/public_html/WEB-INF/web.xml"
Web service started: :8000
Endpoint [SSL:
  ServerSocket[addr=0.0.0.0/0.0.0.0,port=0,localport=7000]]
    shutdown due to exception:
    javax.net.ssl.SSLException: No available certificate corresponds
        to the SSL cipher suites which are enabled.
endpoint down: :7000
J2EE server startup complete.
```

Task 9: J2EE is now installed and running. Explore the J2EE Reference Implementation documentation within the *documentation site* to familiarize yourself more with J2EE.

Help for task 9: There is extensive documentation on all aspects of J2EE, along with example enterprise beans API documentation for the `javax.ejb` packages, and other goodies.

Exercise 1. How to install and configure Sun's J2EE Reference Implementation: Solution

There is no solution to this exercise. When the tasks in this exercise have been completed, the J2EE RI will be installed, running, and available for the subsequent exercises.

Exercise 2. How to create an entity bean: Tasks

This exercise implements a `MusicCD` entity bean, which represents data stored in a database describing an audio CD. The MusicCD bean will be used by subsequent exercises.

For more help with exercises, see About the exercises on page 40 .

Prerequisites

* Exercise 1. How to install and configure Sun's J2EE Reference Implementation: Tasks on page 40

Skeleton code

* *MusicCD.java*
* *MusicCDHome.java*
* *MusicCDBean.java*

Task 1: Design a `MusicCD` remote interface and a `MusicCDHome` home interface. The MusicCD bean must encapsulate the persistent fields `upc`, `title`, `artist`, `type`, and `price`, so the `MusicCD` remote interface should define accessors and mutators for these persistent fields. The `MusicCDHome` interface should define a `findByPrimaryKey()` method which accepts a `String` argument for the `upc`, and a `create()` method that accepts a `upc`.

Task 2: Code the `MusicCDBean` entity bean implementation class. You need to provide implementations for all the methods defined in the remote interface, plus you need to implement all the methods defined in the `EntityBean` interface. You also need to provide an implementation of `ejbCreate()` with the same number and datatypes of arguments as you defined in the `create()` method of your home interface.

Task 3: Compile all the classes that make up your bean.

Task 4: Package your entity bean into a jar file, using the provided XML deployment descriptor.

Help for each of these tasks can be found on the next panel.

Exercise 2. How to create an entity bean: Help

Task 1: Design a `MusicCD` remote interface and a `MusicCDHome` home interface. The MusicCD bean must encapsulate the persistent fields `upc`, `title`, `artist`, `type`, and

price, so the MusicCD remote interface should define accessors and mutators for these persistent fields. The MusicCDHome interface should define a findByPrimaryKey() method which accepts a String argument for the upc, and a create() method that accepts a upc.

Help for task 1: Remember that all of the methods in the remote interface must throw RemoteException and that the create() methods in the home interface must return an object of type MusicCD.

Task 2: Code the MusicCDBean entity bean implementation class. You need to provide implementations for all the methods defined in the remote interface, plus you need to implement all the methods defined in the EntityBean interface. You also need to provide an implementation of ejbCreate() with the same number and datatypes of arguments as you defined in the create() method of your home interface.

Help for task 2: MusicCDHome should prescribe a single bean-creation method with the signature:

```
public MusicCD create(String upc)
               throws CreateException, RemoteException;
```

but the implementation of this in the bean class must return null in the case of container-managed persistence.

Because MusicCDBean is an implementation of an entity bean, it has a primary key and it must have creation methods with arguments to initialize the key: create() and ejbCreate().

Task 3: Compile all the classes that make up your bean.

Help for task 3: Don't forget that the compiler needs to be able to find the javax.ejb classes, which are in J2EE_HOME/j2ee.jar. You may use the following command to compile:

```
javac -classpath %CLASSPATH%;%J2EE_HOME%\lib\j2ee.jar;. musicstore\*.java
```

Task 4: Package your entity bean into a jar file, using the provided XML deployment descriptor.

Help for task 4: Execute the command:

```
jar cvf MusicCD.jar musicstore\*.class META-INF\ejb-jar.xml
```

to create the jar file.

Exercise 2. How to create an entity bean: Solution

The following files contain a complete implementation of the MusicCD entity bean:

* *solution/musicstore/MusicCD.java*
* *solution/musicstore/MusicCDHome.java*

* *solution/musicstore/MusicCDBean.java*
* *solution/META-INF/ejb-jar.xml*

Exercise 3. How to create a stateless session bean: Tasks

This exercise implements an `Inventory` session bean, which can perform inventory operations via the `MusicCD` bean, with both beans operating in the middle-tier. Inventory-management operations are initiated by a distributed client, `MusicInventoryClient`.

For more help with exercises, see About the exercises on page 40 .

Prerequisites

* Exercise 1. How to install and configure Sun's J2EE Reference Implementation: Tasks on page 40
* Exercise 2. How to create an entity bean: Tasks on page 43

Skeleton code

* *Inventory.java*
* *InventoryHome.java*
* *InventoryBean.java*
* *MusicData.java*

Task 1: Design a wrapper class called `MusicData`, that can represent the persistent fields in `MusicCD` (namely `upc`, `title`, `artist`, `type`, `price`).

Task 2: Design an `Inventory` session bean that provides an `addInventory()` service. This argument should take an array of `MusicData` objects as an argument. Within the `addInventory()` method, this bean should establish a `MusicCDHome` reference and then loop through the data passed as an argument to `addInventory()`, creating `MusicCD` beans and setting their attributes.

Task 3: Compile all the classes that make up your bean.

Task 4: Package your session bean into a jar file, using the provided XML deployment descriptor.

Help for each of these tasks can be found on the next panel.

Exercise 3. How to create a stateless session bean: Help

Task 1: Design a wrapper class called `MusicData`, that can represent the persistent fields in `MusicCD` (namely `upc`, `title`, `artist`, `type`, `price`).

Help for task 1: Don't forget to implement `java.io.Serializable`.

`MusicData` should provide a constructor for setting its instance fields with the `MusicCD` attributes:

```
...
public MusicData(String upc, String title,
                 String artist, String type, float price) {
  this.upc    = upc;
  this.title  = title;
  this.artist = artist;
  this.type   = type;
  this.price  = price;
}
...
```

Task 2: Design an `Inventory` session bean that provides an `addInventory()` service. This argument should take an array of `MusicData` objects as an argument. Within the `addInventory()` method, this bean should establish a `MusicCDHome` reference and then loop through the data passed as an argument to `addInventory()`, creating `MusicCD` beans and setting their attributes.

Help for task 2: `InventoryHome` should prescribe a single bean-creation method with the signature:

```
public Inventory create() throws CreateException, RemoteException;
```

`Inventory` should prescribe one method only, `addInventory()`, with an interface similar to the following, depending on your design:

```
public boolean addInventory(MusicData[] data)
               throws RemoteException;
```

Because `Inventory` is a session bean, it has no primary key and its creation methods have no arguments: `create()` and `ejbCreate()`.

Note that `InventoryBean.addInventory()` is similar to `MusicClient` in that it must establish a JNDI context, look up the `MusicCD` bean, and so on.

Task 3: Compile all the classes that make up your bean.

Help for task 3: Don't forget that the compiler needs to be able to find the `javax.ejb` classes, which are in `J2EE_HOME/j2ee.jar`. You may use the following command to compile:

```
javac -classpath %CLASSPATH%;%J2EE_HOME%\lib\j2ee.jar;. musicstore\*.java
```

Task 4: Package your session bean into a jar file, using the provided XML deployment descriptor.

Help for task 4: Execute the command:

```
jar cvf Inventory.jar musicstore\*.class META-INF\ejb-jar.xml
```

to create the jar file.

Exercise 3. How to create a stateless session bean: Solution

The following files contain a complete implementation of the Inventory session bean:

* *solution/musicstore/Inventory.java*
* *solution/musicstore/InventoryHome.java*
* *solution/musicstore/InventoryBean.java*
* *solution/musicstore/MusicData.java*
* *solution/META-INF/ejb-jar.xml*

Exercise 4. How to set up the database: Tasks

As illustrated in the previous exercises, the database has a music store theme. This exercise sets up the database with one table for music CDs. This exercise includes a pre-built database for the Cloudscape database system in jar format.

Also, the skeleton code and solutions sections include programs for building the database via JDBC. Note that all utility programs associated with this project have `-usage`, `-help`, and `-h` command-line options.

For more help with exercises, see About the exercises on page 40 .

Prerequisites

* Exercise 1. How to install and configure Sun's J2EE Reference Implementation: Tasks on page 40

Resources

* *DatabaseTool.java*
* *MusicStoreDB.jar*
* *MusicCDCreateTables.java*
* *MusicCDInsertRecords.java*

Task 1: Choose a database system appropriate for your environment and then either install or build the database system with the music data that you prefer.

Task 2: Verify the integrity of the database using a vendor-supplied graphical tool or the `DatabaseTool.java` program provided.

Help for each of these tasks can be found on the next panel.

Exercise 4. How to set up the database: Help

Task 1: Choose a database system appropriate for your environment and then either install or build the database system with the music data that you prefer.

Help for task 1: Sun's J2EE Reference Implementation comes with a built-in version of the Cloudscape RDBMS. If you are using J2EE RI, you should initially use Cloudscape as your database to avoid configuration problems. You can learn how to substitute a different database after you become comfortable with creating and deploying applications using Cloudscape.

If you insist on initially using a database other than Cloudscape, you should carefully read and follow the instructions in the following J2EE RI documentation before you begin:

* *Configuration Guide*
* *Release Notes*

Note that Microsoft Access doesn't provide adequate SQL support for the tasks required by EJB technology servers.

For Cloudscape, simply copy the `MusicStoreDB.jar` archive into the directory `%J2EE_HOME%\cloudscape` and then unjar the database. (All files unjar into the directory `MusicStoreDB` within the current working directory.)

For other database environments, simply compile and run the programs `MusicCDCreateTables.java` and `MusicCDInsertRecords.java` after creating an empty database. You can modify the source for the appropriate driver and URL values, or supply them as command-line options. You can also perform these tasks using `DatabaseTool`, or `JDBCTest` (`TestTool`) from the *Sun's JDBC area* .

One last step is needed when running with J2EE RI: You need to modify the `%J2EE_HOME%\config\default.properties` file to define the datasources available to J2EE RI. This is done by changing the line that reads (with everything on one line):

```
jdbc.datasources=jdbc/Cloudscape|
  jdbc:cloudscape:rmi:CloudscapeDB;create=true
```

to be:

```
jdbc.datasources=jdbc/Cloudscape|
  jdbc:cloudscape:rmi:CloudscapeDB;create=true|
  jdbc/MusicStore|jdbc:cloudscape:rmi:MusicStoreDB;create=false
```

(with everything on one line) and then restarting both Cloudscape and J2EE RI.

Barnes and Noble and other sites provide easy access to music CD data such as UPCs, if you want to customize the MusicCD database with your own entries.

Task 2: Verify the integrity of the database using a vendor-supplied graphical tool or the `DatabaseTool.java` program provided.

Help for task 2: The exercise solution on the next panel has example output for the Cloudscape View tool and for `DatabaseTool.java`.

Exercise 4. How to set up the database: Solution

There is no explicit solution to this exercise. If you have set up the database correctly, then

when you run `cloudscape -start` you will see output similar to the following:

```
myhost> cloudscape -start
Wed Jan 12 11:51:20 PST 2000:
     [RmiJdbc] COM.cloudscape.core.JDBCDriver registered
        in DriverManager
Wed Jan 12 11:51:20 PST 2000:
     [RmiJdbc] Binding RmiJdbcServer...
Wed Jan 12 11:51:20 PST 2000:
     [RmiJdbc] No installation of RMI Security Manager...
Wed Jan 12 11:51:22 PST 2000:
     [RmiJdbc] RmiJdbcServer bound in rmi registry
```

Likewise, when you startup J2EE RI you should see something like the following:

```
myhost> j2ee -verbose

Naming service started: :1050
Published the configuration object ...
Binding DataSource, name = jdbc/Cloudscape,
    url = jdbc:cloudscape:rmi:CloudscapeDB;create=true
Binding DataSource, name = jdbc/MusicStore,
    url = jdbc:cloudscape:rmi:MusicStoreDB;create=false
Configuring web service using "default"
Web service started: :9191
Web service started: :7000
Configuring web service using "default"
Configuring web service using
  "file:/D:/j2sdkee1.2/public_html/WEB-INF/web.xml"
Web service started: :8000
Endpoint [SSL:
  ServerSocket[addr=0.0.0.0/0.0.0.0,port=0,localport=7000]]
    shutdown due to exception:
    javax.net.ssl.SSLException: No available certificate corresponds
        to the SSL cipher suites which are enabled.
endpoint down: :7000
J2EE server startup complete.
```

Exercise 5. How to deploy enterprise beans in Sun's J2EE Reference Implementation: Tasks

This exercise takes you through the process of deploying the MusicCD entity bean and Inventory session bean into the J2EE Reference Implementation container. Deploying is usually a major task in and of itself, and it is *always* application-server specific. If you are using an application server other than J2EE RI, you should consult that server's documentation for the details of how to deploy your beans. You will need to have the MusicCD and Inventory beans deployed for the subsequent exercises.

For more help with exercises, see About the exercises on page 40 .

Prerequisites

* Exercise 1. How to install and configure Sun's J2EE Reference Implementation: Tasks on page 40
* Exercise 2. How to create an entity bean: Tasks on page 43
* Exercise 3. How to create a stateless session bean: Tasks on page 45
* Exercise 4. How to set up the database: Tasks on page 47

Skeleton code

* *Inventory.jar*
* *MusicCD.jar*

Task 1: Save the two jar files from the skeleton code section above into your working directory. From that working directory, start the `deploytool` utility. This runs an application for deploying enterprise beans into a running J2EE RI server.

Task 2: In `deploytool`, select **File/New Application...** to create a new application called MusicStore saved in a file called `MusicStore.ear`.

Task 3: Select the MusicStore application from the list of deployed applications on the left side of the `deploytool` GUI. Select **File/Add EJB JAR to Application...** and use the dialog box to open `MusicCD.jar`, which is provided as the skeleton code for this exercise.

Task 4: Follow the same procedure to add the `Inventory.jar` to the application.

Task 5: With the MusicStore application selected, choose the JNDI panel from the main portion of the `deploytool` window. Fill in the JNDI names for each component as shown:

MusicCDBean	MusicCDBean	ejb/MusicCD
InventoryBean	InventoryBean	ejb/Inventory
InventoryBean	ejb/MusicCD	ejb/MusicCD

Task 6: Now set up the database mapping for the MusicCD entity bean. You will have to select the MusicCDBean component from the left window. Then select the **Entity** tab in the main window and click the **Deployment Settings** button to get the input dialog box. Set the database JNDI name to jdbc/MusicStore in this dialog, uncheck the boxes **Create table on deploy** and **Delete table on deploy**, then click the **Generate SQL now** button.

Task 7: The MusicStoreDB that you set up in the previous exercise uses a table called MusicCD to store the MusicCD beans. The SQL commands that the J2EE RI automatically generates do not use this exact table name, so you need to select each of the life-cycle methods and edit the SQL being generated for each to change them to operate on the MusicCD table.

Task 8: In `deploytool`, select **Tools/Deploy Application...** from the menu. Check the box to **Return Client Jar**, accept all the other defaults, and step through the wizard. Click the **Finish** button at the end.

Task 9: You should see MusicStore application show up in the lower section of the `deploytool` screen in the panel labeled **Server Applications**. You have successfully deployed your beans.

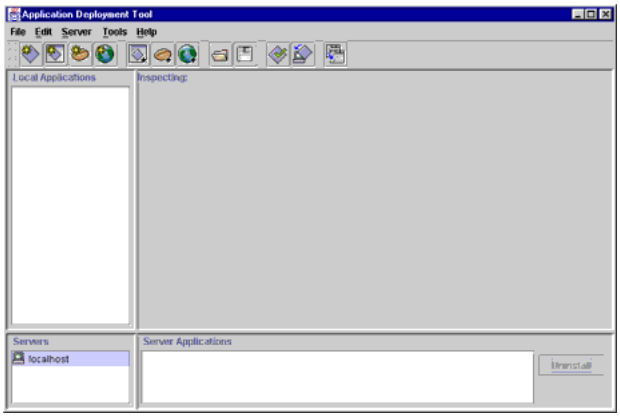
Help for each of these tasks can be found on the next panel.

Exercise 5. How to deploy enterprise beans in Sun's J2EE Reference Implementation: Help

Task 1: Save the two jar files from the skeleton code section above into your working

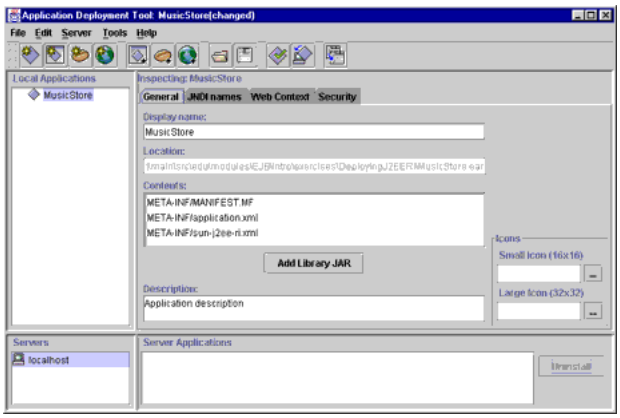
directory. From that working directory, start the `deploytool` utility. This runs an application for deploying enterprise beans into a running J2EE RI server.

Help for task 1: First make sure that your J2EE RI server and database server are running. If they aren't, execute `cloudscape -start` and `j2ee -verbose` from the command-line. You will need the `J2EE_HOME/bin` directory in your PATH. Then you can run `deploytool` from the command-line. This should bring up the following screen:



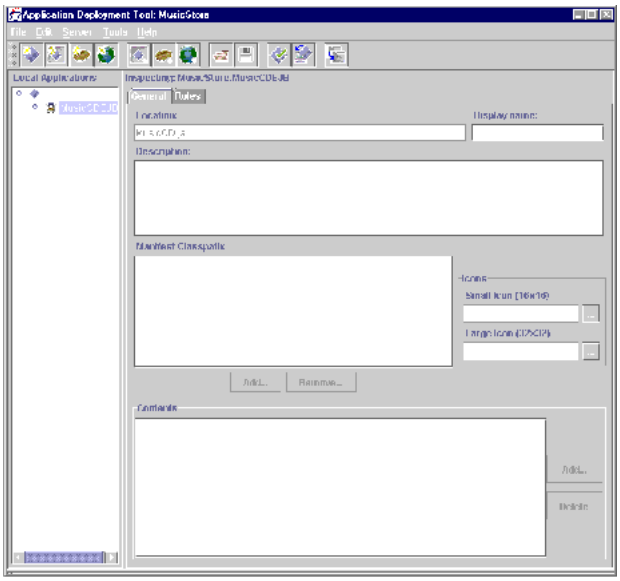
Task 2: In `deploytool`, select **File/New Application...** to create a new application called MusicStore saved in a file called `MusicStore.ear`.

Help for task 2: The left portion of the `deploytool` GUI should now show MusicStore as an application:



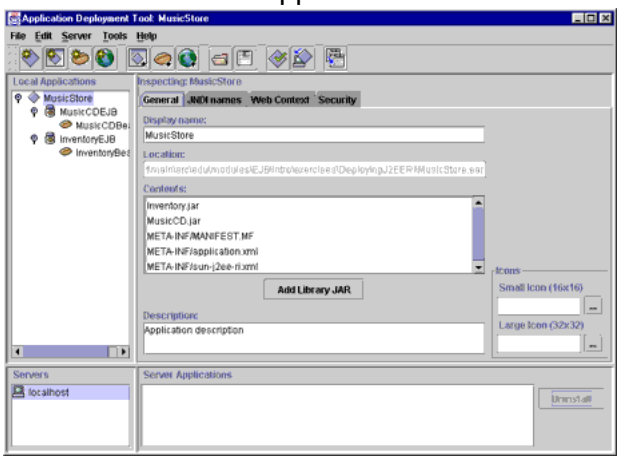
Task 3: Select the MusicStore application from the list of deployed applications on the left side of the `deploytool` GUI. Select **File/Add EJB JAR to Application...** and use the dialog box to open `MusicCD.jar`, which is provided as the skeleton code for this exercise.

Help for task 3: The left portion of the `deploytool` GUI should now show MusicCDEJB beneath the MusicStore application as shown here:



Task 4: Follow the same procedure to add the `Inventory.jar` to the application.

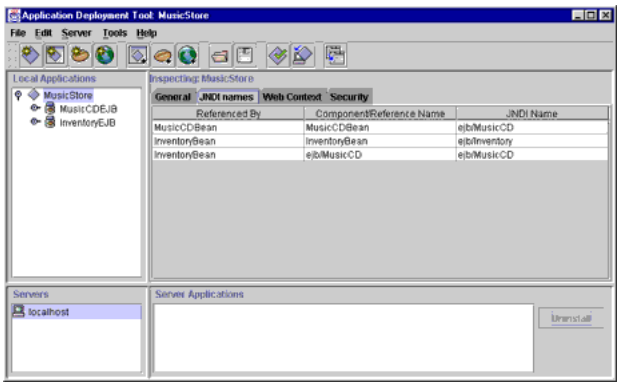
Help for task 4: The left portion of the `deploytool` GUI should now show MusicCDEJB and InventoryEJB beneath the MusicStore application as shown here:



Task 5: With the MusicStore application selected, choose the JNDI panel from the main portion of the `deploytool` window. Fill in the JNDI names for each component as shown:

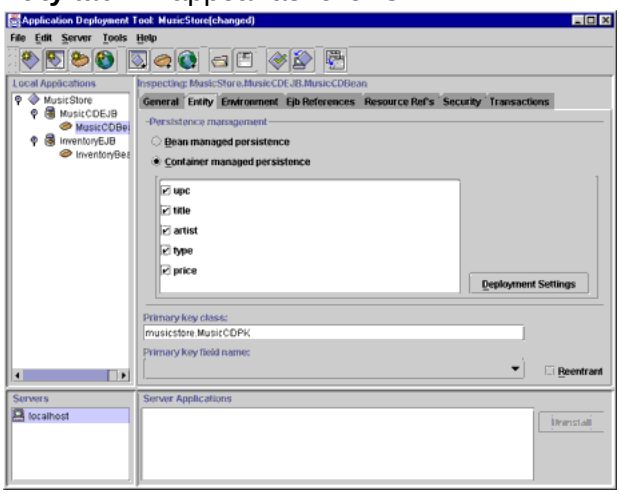
MusicCDBean	MusicCDBean	ejb/MusicCD
InventoryBean	InventoryBean	ejb/Inventory
InventoryBean	ejb/MusicCD	ejb/MusicCD

Help for task 5: You should see a list of three component with blank JNDI names. Fill in the names as shown here:



Task 6: Now set up the database mapping for the MusicCD entity bean. You will have to select the MusicCDBean component from the left window. Then select the **Entity** tab in the main window and click the **Deployment Settings** button to get the input dialog box. Set the database JNDI name to jdbc/MusicStore in this dialog, uncheck the boxes **Create table on deploy** and **Delete table on deploy**, then click the **Generate SQL now** button.

Help for task 6: The **Entity** tab will appear as follows:



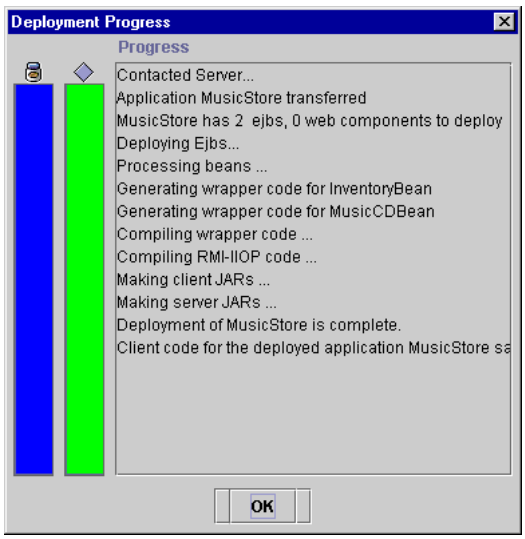
`jdbc/MusicStore` is the JNDI name of the database you set up in the . If `deploytool` complains about this database name, make sure you have completed the previous exercise properly and that the J2EE RI server has been restarted since then (so that it will pick up the database you created). After this is done, it should look something like this:

Task 7: The MusicStoreDB that you set up in the previous exercise uses a table called MusicCD to store the MusicCD beans. The SQL commands that the J2EE RI automatically generates do not use this exact table name, so you need to select each of the life-cycle methods and edit the SQL being generated for each to change them to operate on the MusicCD table.

Help for task 7: See the help for the previous task to see what the screen should look like when you edit the SQL commands.

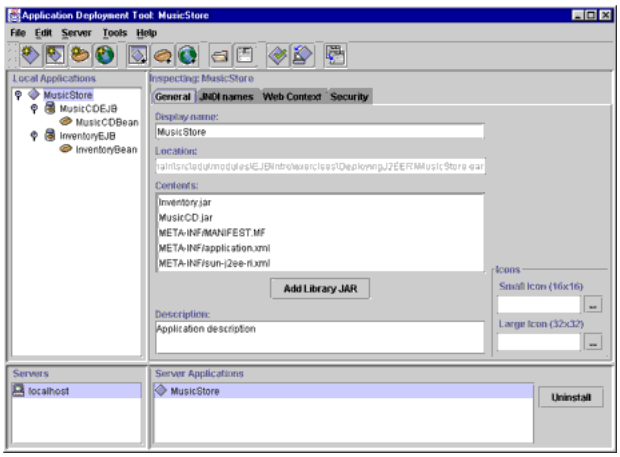
Task 8: In `deploytool`, select **Tools/Deploy Application...** from the menu. Check the box to **Return Client Jar**, accept all other defaults, and step through the wizard. Click the **Finish** button at the end.

Help for task 8: After this is done, and if the deployment was successful, you should see a dialog something like this:



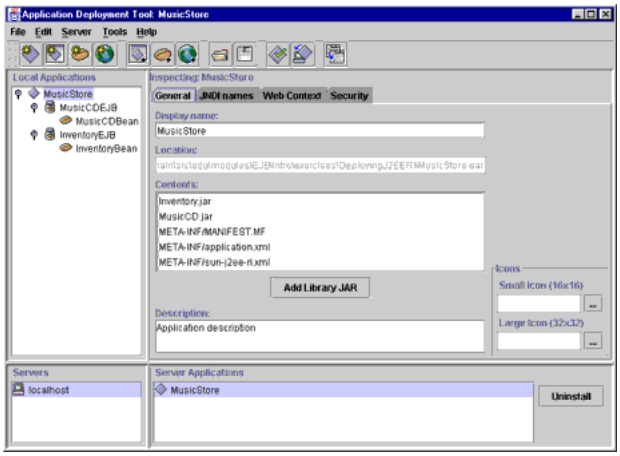
Task 9: You should see MusicStore application show up in the lower section of the `deploytool` screen in the panel labeled **Server Applications**. You have successfully deployed your beans.

Help for task 9: Note that the FCS version of J2EE RI has a bug that prevents the **Server Applications** panel to be initially shown. Use your mouse and drag the divider bar which appears along the bottom of the screen until this panel is visible. It should look something like this:



Exercise 5. How to deploy enterprise beans in Sun's J2EE Reference Implementation: Solution

When you have sucessfully deployed the MusicStore application, your `deploytool` screen should look something like this:



Exercise 6. How to create EJB clients: Tasks

This exercise implements two stand-alone client applications to use the `MusicCD` entity bean and `Inventory` session bean which you developed and deployed in the previous exercises.

Note: Due to a bug in the 1.0 release of the J2EE RI, these client programs will not run properly. The J2EE RI does not conform to the J2EE specification regarding the use of compound primary keys. The J2EE RI 1.0 only works properly with single-valued primary keys. In order to make this exercise work with the J2EE RI, you will have to use the `upc` field of the `MusicCDBean` class as the primary key and remove all mention of `MusicCDPK` in the `MusicCD` entity bean, the `Inventory` session bean, and the clients in this exercise. This bug will be fixed in the next J2EE RI release.

For more help with exercises, see About the exercises on page 40 .

Prerequisites

* Exercise 1. How to install and configure Sun's J2EE Reference Implementation: Tasks on page 40
* Exercise 2. How to create an entity bean: Tasks on page 43
* Exercise 3. How to create a stateless session bean: Tasks on page 45
* Exercise 4. How to set up the database: Tasks on page 47
* Exercise 5. How to deploy enterprise beans in Sun's J2EE Reference Implementation: Tasks on page 49

Skeleton code

* *MusicClient.java*
* *MusicInventoryClient.java*
* *Inventory.jar*
* *MusicCD.jar*

Task 1: Start the Cloudscape database server with the command:

```
cloudscape -start
```

Task 2: Start the J2EE RI with the command:

```
j2ee -verbose
```

Pay attention to the output to make sure that J2EE loads the MusicStoreDB datasource that you set up in the previous exercise.

Task 3: Edit the `MusicClient.java` skeleton to obtain the JNDI context, and use this context to get a reference to the MusicCDHome.

Task 4: Create a new MusicCD bean with the UPC code given in the skeleton, and set its fields to the values shown in the skeleton. Or if you have a favorite CD of your own, you can use that data.

Task 5: Add code to find the MusicCD bean you just created in the database, then print out the database values of the fields you set in the previous step.

Task 6: Compile and run your client application.

Task 7: Edit the `MusicInventoryClient.java` skeleton to obtain the JNDI context, and use this context to get a reference to the InventoryHome.

Task 8: Create a new Inventory bean.

Task 9: Create an instance of `MusicData` and fill it with information about CDs you want to insert into the database, then use the Inventory bean's business method to perform this task.

Task 10: Compile and run your client application.

Help for each of these tasks can be found on the next panel.

Exercise 6. Creating EJB clients: Help

Task 1: Start the Cloudscape database server with the command:

```
cloudscape -start
```

Help for task 1: You may already have Cloudscape running from the previous exercise. If so, there is no need to start it again (in fact, you can't have two copies running). When shutting down Cloudscape, be sure to use `cloudscape -stop` instead of ^C.

Task 2: Start the J2EE RI with the command:

```
j2ee -verbose
```

Pay attention to the output to make sure that J2EE loads the MusicStoreDB datasource that you set up in the previous exercise.

Help for task 2: Running the J2EE RI via the command-line command `j2ee -verbose` should produce output similar to the following:

```
myhost> j2ee -verbose

Naming service started: :1050
Published the configuration object ...
Binding DataSource, name = jdbc/Cloudscape,
    url = jdbc:cloudscape:rmi:CloudscapeDB;create=true
Binding DataSource, name = jdbc/MusicStore,
    url = jdbc:cloudscape:rmi:MusicStoreDB;create=false
Configuring web service using "default"
Web service started: :9191
Web service started: :7000
Configuring web service using "default"
Configuring web service using
    "file:/D:/j2sdkee1.2/public_html/WEB-INF/web.xml"
Web service started: :8000
Endpoint [SSL:
    ServerSocket[addr=0.0.0.0/0.0.0.0,port=0,localport=7000]]
    shutdown due to exception:
    javax.net.ssl.SSLException: No available certificate corresponds
        to the SSL cipher suites which are enabled.
endpoint down: :7000
Loading jar:/D:/j2sdkee1.2/repository/myhost/applications/
    MusicStore947749297345Server.jar
/D:/j2sdkee1.2/repository/myhost/applications/
    MusicStore947749297345Server.jar
Looking up authenticator...
Binding name:java:comp/env/ejb/MusicCD
J2EE server startup complete.
```

Note in particular the lines:

```
Binding DataSource, name = jdbc/MusicStore,
    url = jdbc:cloudscape:rmi:MusicStoreDB;create=false
```

that let you know that the MusicStore database is known to the J2EE server, and

```
Loading jar:/D:/j2sdkee1.2/repository/myhost/applications/
    MusicStore947749297345Server.jar
/D:/j2sdkee1.2/repository/myhost/applications/
    MusicStore947749297345Server.jar
Looking up authenticator...
Binding name:java:comp/env/ejb/MusicCD
```

that let you know the MusicStore application has been deployed.

Task 3: Edit the `MusicClient.java` skeleton to obtain the JNDI context, and use this context to get a reference to the MusicCDHome.

Help for task 3: Obtain the JNDI context by creating a new instance of `javax.naming.InitialContext` and storing it in a reference variable of datatype `javax.naming.Context`

Task 4: Create a new MusicCD bean with the UPC code given in the skeleton, and set its fields to the values shown in the skeleton. Or if you have a favorite CD of your own, you can use that data.

Help for task 4: Use the `create()` method that takes the UPC code as an argument to create the bean. Use the accessor and mutator methods defined in the remote interface to modify the bean's data.

Task 5: Add code to find the MusicCD bean you just created in the database, then print out the database values of the fields you set in the previous step.

Help for task 5: Create an instance of `MusicCDPK`, set the `upc` field to an appropriate value, and use the `findByPrimaryKey()` method of the MusicCD home interface to find the MusicCD bean. When you have a reference to the bean, you can read the data using accessor methods on that bean.

Task 6: Compile and run your client application.

Help for task 6: Because the client needs to use both the MusicCD bean and the Inventory bean, you will need to put `Inventory.jar` and `MusicCD.jar` in your CLASSPATH in order to compile. You will also have to make sure that `j2ee.jar` is in your CLASSPATH, because that is where the `javax.ejb` classes are stored.

Task 7: Edit the `MusicInventoryClient.java` skeleton to obtain the JNDI context, and use this context to get a reference to the InventoryHome.

Help for task 7: You obtain the JNDI context by creating a new instance of `javax.naming.InitialContext` and storing it in a reference variable of datatype `javax.naming.Context`.

Task 8: Create a new Inventory bean.

Help for task 8: Use the `create()` method on the Inventory home interface.

Task 9: Create an instance of `MusicData` and fill it with information about CDs you want to insert into the database, then use the Inventory bean's business method to perform this task.

Help for task 9: The `addInventory()` method of the Inventory bean takes an instance of `MusicData` as an argument, and inserts those records into the database.

Task 10: Compile and run your client application.

Help for task 10: Because the client needs to use both the MusicCD bean and the Inventory bean, you will need to put `Inventory.jar` and `MusicCD.jar` in your CLASSPATH in order to compile. You will also have to make sure that `j2ee.jar` is in your CLASSPATH, because that is where the `javax.ejb` classes are stored.

Exercise 6. How to create EJB clients: Solution

As a solution, both the source code for the client applications and the `MusicStoreClient.jar` file, which holds the container-generated stubs and skeletons and other things needed for a stand-alone client to communicate with the J2EE server, are provided.

* *solution/musicstore/MusicClient.java*
* *solution/musicstore/MusicInventoryClient.java*
* *solution/MusicStoreClient.jar*

Section 9. Summary

Further reading and references

There are many resources available on and off the Internet to learn about Enterprise JavaBeans technology. Below is a sampling of the available Web sites, Sun's documentation pages for related EJB technologies, and printed books.

developerWorks tutorials and articles

* *"Introduction to Enterprise JavaBeans"* tutorial
* *"Efficient Entity EJB development"*
* "What are EJB components?, *part 1* , *part 2* , and *part 3* "
* *"Developing enterprise beans with VisualAge for Java"*
* *"Read all about EJB 2.0"* (JavaWorld reprint)
* *"Enterprise JavaBeans: Answers to every developer's top questions"*

Web sites

The following sites have product information as well as white papers on EJB technology, Java application servers, and other distributed technologies:

* *Sun Microsystems, Inc. EJB vendor list*
* *Object Management Group, Inc.*
* *EJB-INTEREST -- EJB mailing list archive*
* *EJBNow -- a resource for EJB technology developers*
* *jGuru's EJB technology FAQ*

Documentation and specifications

java.sun.com , the Sun Microsystems Web site for Java technology, includes a *Products and APIs* page packed with links to Java enterprise-related products, APIs, and general information, including:

* *EJB*
* *J2EE*
* *Writing Enterprise Applications with J2EE*
* *JNDI*
* *Servlet API*
* *JDBC API*
* *Java Transaction API*
* *Java Transaction Service*
* *RMI over IIOP compiler*

Books

Recent books with Enterprise JavaBeans and distributed computing technology themes include:

* Monson-Haefel, R. " *Enterprise JavaBeans* ," second edition. Sebastopol, CA: O'Reilly & Associates, 2000, ISBN: 1-56592-869-5.
* Roman, E. " *Mastering Enterprise JavaBeans and the Java 2 Platform* ." New York:

John Wiley & Sons, 1999, ISBN: 0-471-33229-1.
* Farley, J. " *Java Distributed Computing* ." Sebastopol, CA: O'Reilly & Associates, 1998, ISBN: 1-56592-206-9.
* David Flanagan, et al. " *Java Enterprise in a Nutshell : A Desktop Quick Reference* ." New York: O'Reilly & Associates; ISBN: 1-56592-483-5.
* Orfali, R., Harkey, D., and Edwards, J. " *The Client/Server Survival Guide, 3rd Ed.* " New York: John Wiley & Sons, 1999, ISBN: 0-471-31615-6.

Miscellaneous

Database-related issues are discussed in Chris Date's classic text:

* Date, C. " *An Introduction to Database Systems, 7th Ed.* " Reading, MA: Addison-Wesley, 1999, ISBN: 0-201-38590-2.

Feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like to see covered. Thanks!

For questions about the content of this tutorial, contact the authors Richard Monson-Haefel (*rmh@jguru.com*) or Tim Rohaly (*rohaly@jguru.com*).

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The Toot-O-Matic tool is a short Java program that uses XSLT stylesheets to convert the XML source into a number of HTML pages, a zip file, JPEG heading graphics, and PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML.