Database Access with Enterprise JavaBeans EJB

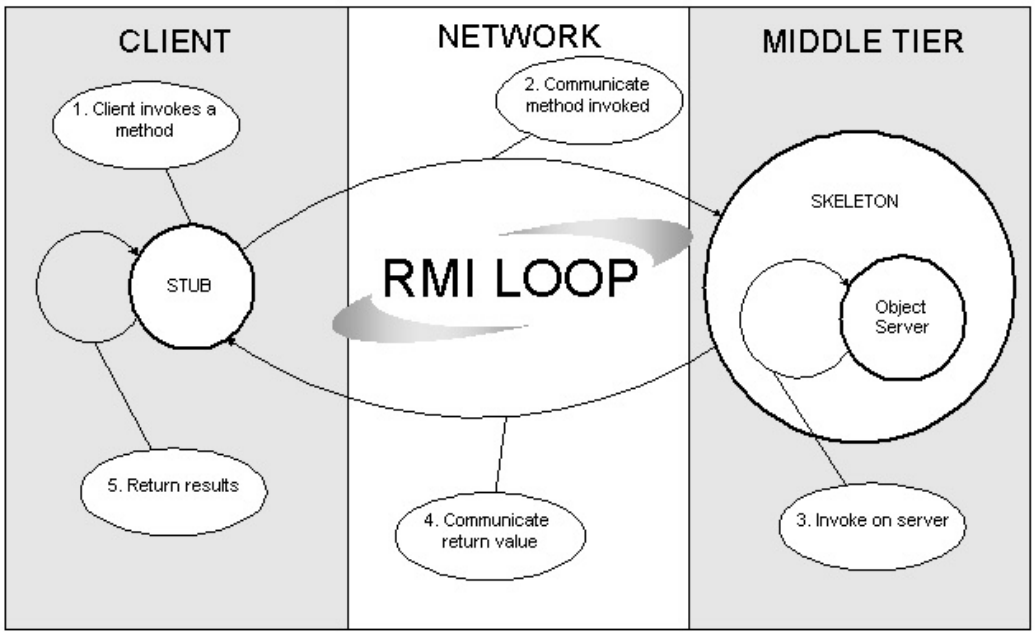
1 Introduction

1.1 Distributed Object Architectures

Enterprise JavaBeans (EJB) is a standard server-side component model for component transaction monitors, which are based on distributed object technologies. Distributed object systems are the foundation for modern three-tier architectures. Essentially, there are three parts to this architecture: the object server, the skeleton, and the stub.

The *object server* is the business object that resides on the middle tier. The object server is an instance of an object with its own unique state. Every object server class has matching stub and skeleton classes built specifically for that type of object server.

The *stub* and *skeleton* are responsible for making the object server, which lives on the middle tier, look as if it is running locally on the client machine. This is accomplished through some kind of *remote method invocation* (RMI) protocol.



1.2 Component Models

Enterprise JavaBeans specifies a *server-side* component model. Using a set of classes and interfaces from the `javax.ejb` packages, developers can create, assemble, and deploy components that conform to the EJB specification.

The original JavaBeans (the `java.beans` package) is intended to be used for *intraprocess* purposes, while EJB (the `javax.ejb` package) is designed to be used for *interprocess* components.

1.3 Component Transaction Monitors

The Component Transaction Monitors (CTM) is a hybrid of Object Request Broker (ORB) and Transaction Processing monitor (TP monitor). The CTM provides a powerful, robust distributed object platform. This is a comprehensive environment for server-side components by managing automatically

- concurrency
- transactions
- object distribution
- load balancing
- security
- resource management

1.4 The Enterprise Bean Components

Enterprise JavaBeans server-side components come in two fundamentally different types: *entity beans* and *session beans*. Entity beans model business concepts or model real-word objects. These objects are usually persistent records in a database.

Session beans are an extension of the client application and are responsible for managing processes or tasks. A session bean models interactions but does not have persistent state.

1.5 Classes and Interfaces

To implement an enterprise bean, you need to define two interfaces and one or two classes.

1.5.1 Remote Interface

The remote interface for an EJB defines the bean's business methods: the methods a bean presents to the outside world to do its work. The remote interface extends `javax.ejb.EJBOjbect`, which in turn extends `java.rmi.Remote`. We will call the object that implements this interface the *EJB object*.

1.5.2 Home Interface

The home interface defines the bean's life cycle methods: methods for creating new beans, removing beans, and finding beans. The home interface extends `javax.ejb.EJBHome`, which in turn extends `java.rmi.Remote`. We will call the object that implements the home interface the *EJB home*.

1.5.3 Bean Class

The bean class actually implements the bean's business methods. Note, however, that the bean class usually does not implement the bean's home or remote interfaces. However, it must have methods matching the signatures of the methods defined in the remote interface and must have methods corresponding to some of the methods in the home interface. An entity bean must implement `javax.ejb.EntityBean`, a session bean must implement `javax.ejb.SessionBean`, both extend `javax.ejb.EnterpriseBean`.

1.5.4 Primary Key

The primary key is very simple class that provides a pointer into the database. Only entity beans need primary keys. The only requirement for this class is that it implements `java.io.Serializable`.

1.6 Deployment Descriptor and JAR Files

Much of the information about how beans are managed at runtime is not addressed in the interfaces and classes. Deployment descriptors serve a function very similar to property files. They allow us to customise behaviour of enterprise beans at runtime without having to change the software itself.

When a bean class and its interfaces have been defined, a deployment descriptor for the bean is created and populated with data about the bean. IDEs will allow developers to set up the deployment descriptors for a bean. The descriptor is saved to a file and can be packaged in a JAR file for deployment.

JAR (Java ARchive) files are ZIP files that are used specifically for packaging Java classes (and other resources). A JAR file containing one or more enterprise beans includes the bean classes, remote interfaces, home interfaces, and primary keys, for each bean. It also contains one deployment descriptor, which is used for all the beans in the JAR files.

1.7 Further Documentation

[1] Enterprise JavaBeans, Richard Monson-Haefel, O'Reilly, ISBN 1-56592-869-5

[2] Enterpise JavaBeans Technology:
http://java.sun.com/products/ejb/

[3] Enterprise JavaBeans Tutorial:
http://developer.java.sun.com/developer/onlineTraining/Beans/EJBTutorial/

[4] Java Blueprints, Java Pet Store Sample Application:
http://java.sun.com/blueprints/code/

2 Developing Your First Enterprise Beans

The following samples use the EJB 1.1 specification. The EJB server you choose should be compliant with this version.

The samples use the well known tables DEPT and EMP: Every employee has exactly one department assigned. One department has no, one, or more employees. Based on that some business methods can be defined.

2.1 Developing an Entity Bean

There are two types of entity beans, and they are distinguished by how they manage persistence. Container-Managed Persistence (CMP) Beans have their persistence automatically managed by the EJB container. The container knows how a bean instance's fields map to the database and automatically takes care of inserting, updating, and deleting the data associated with entities in the database.

Beans using Bean-Managed Persistence (BMP) do all the work explicitly: The EJB container tells the bean instance when it is safe to insert, update, and delete its data from the database, but it provides no other help. The bean instance does all the persistent work itself.

The next subchapters will describe the definition and implementation of a CMP bean.

2.1.1 Emp – The Remote Interface

The remote interface defines the bean's business purpose; the methods of this interface must capture the concept of the entity.

```
import java.util.Date;
import java.rmi.RemoteException;

public interface Emp extends javax.ejb.EJBObject
{
  public int getEmpNo() throws RemoteException;
  public void setEmpNo( int empNo ) throws RemoteException;
  public String getEName() throws RemoteException;
  public void setEName( String eName ) throws RemoteException;
  public Date getHireDate() throws RemoteException;
  public void setHireDate( Date hireDate ) throws RemoteException;
  public int getDeptNo() throws RemoteException;
  public void setDeptNo( int deptNo ) throws RemoteException;
}
```

The `Emp` interface defines four properties: the `employee number`, `employee name`, `hire date`, and `department number`. Properties are attributes of a bean that can be accessed by public set and get methods.

2.1.2 EmpHome – The Home Interface

The bean's home interface specifies how the bean can be created, located, and destroyed. In other words, the Emp bean's life-cycle behaviour.

```
import java.util.Collection;
import java.util.Date;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;

public interface EmpHome extends javax.ejb.EJBHome
{
  public Emp create(
      int empNo
    , String eName
    , Date hireDate
    , int deptNo
  )
    throws RemoteException, CreateException;

  public Emp findByPrimaryKey( EmpPK primaryKey )
    throws RemoteException, FinderException;

  public Collection findByDeptNo( int deptNo )
     throws RemoteException, FinderException;
}
```

The `EmpHome` interface extends the `javax.ejb.EJBHome` and defines two life-cycle methods `create()` and `findByPrimaryKey()`. Theses methods create and locate Emp beans. The find method `findByDeptNo()` is defined for additionally search functionality. Remove methods for deleting beans are defined in the `javax.ejb.EJBHome` interface, so the `EmpHome` interface inherits them.

2.1.3 EmpPK – The Primary Key Class

The EmpPK is the primary key class of the Emp bean. All entity beans must have a serializable primary key that can be used to uniquely identify an entity bean in the database.

```
public class EmpPK implements java.io.Serializable
{
  public int mEmpNo = -1;

  public EmpPK()
  {
  }

  public EmpPK( int empNo )
  {
    mEmpNo = empNo;
  }

  // {...} full implementation see www.akadia.com
}
```

The primary key implements the `java.io.Serializable` interface and defines one public attribute, the `mEmpNo`. This employee number is used to locate specific Emp entities or records in the database at runtime. EJB 1.1 requires that we override the `Object.hashCode()` and `Ojbect.equals()` methods.

2.1.4 EmpBean – The Bean Class

The `EmpBean` class is an entity bean that uses container-managed persistence. In addition to the callback methods, we must also define EJB implementations for most of the methods defined in the `Emp` and `EmpHome` interfaces.

```
import java.util.Date;
import javax.ejb.EntityContext;

public class EmpBean implements javax.ejb.EntityBean
{
  public EntityContext mCtx = null;
  public int mEmpNo;
  public String mEName;
  public Date mHireDate;
  public int mDeptNo;

  public EmpBean()
  {
  }

  public EmpPK ejbCreate(
      int empNo
    , String eName
    , Date hireDate
    , int deptNo
  )
  {
    mEmpNo = empNo;
    mEName = eName;
    mHireDate = hireDate;
    mDeptNo = deptNo;
    return null;
  }

  // {...} empty  method implementation
  // {...} setters and getters
  // full code sample see www.akadia.com
}
```

The `mEmpNo`, `mEName`, `mHireDate`, and `mDeptNo` are persistent and property fields. The persistent fields will be mapped to the database at deployment time. These fields are also properties because they are publicly available through remote interface.

The `findByPrimaryKey()` and `findByDeptNo()` methods are not defined in container-managed bean classes. With container-managed beans you do not explicitly

declare find methods in the bean class. Instead, find methods are generated at deployment and implemented by the container.

The business methods in the `EmpBean` match the signatures of the business methods defined in the remote interface. When a client invokes one of these methods on the remote interface, the method is delegated to the matching method on the bean class.

2.2 Differences of BMP Entity Beans

Bean-managed persistence (BMP) is more complicated than container-managed persistence because you must explicitly write the persistence logic into the bean class.

Bean-managed persistence gives you more flexibility in how state is managed between the bean instance and the database. Since BMP beans mange their own persistence, find methods must be defined in the bean class, i.e. the advantage is that you can write your own optimised and performant SQL statements. The disadvantage of BMP beans is that more work is required to define the bean.

The next subchapters will describe the definition and implementation of a BMP bean.

2.2.1 Dept Remote Interface

The remote interface of the BMP bean is identical to the interface of a CMP bean. Setters and getters for the bean's properties are defined here.

```
import java.rmi.RemoteException;

public interface Dept extends javax.ejb.EJBObject
{
  public int getDeptNo() throws RemoteException;
  public void setDeptNo( int deptNo ) throws RemoteException;
  public String getDName() throws RemoteException;
  public void setDName( String dName ) throws RemoteException;
  public String getLoc() throws RemoteException;
  public void setLoc( String loc ) throws RemoteException;
}
```

2.2.2 DeptHome – The Home Interface

The home interface of the BMP bean is identical to the interface of a CMP bean. The Dept bean's life-cycle behaviour is defined here.

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

public interface DeptHome extends javax.ejb.EJBHome
{
  public Dept create(
      int deptNo
    , String eName
    , String loc
  )
    throws RemoteException, CreateException;

  public Dept findByPrimaryKey( DeptPK primaryKey )
    throws RemoteException, FinderException;

}
```

2.2.3 DeptPK – The Primary Key Class

The primary key class of the BMP bean is identical to the primary key class of a CMP bean.

```
public class DeptPK implements java.io.Serializable
{
```

```
  public int mDeptNo = -1;

  public DeptPK()
  {
  }

  public DeptPK( int deptNo )
  {
    mDeptNo = deptNo;
  }

  // {...} full implementation see www.akadia.com
}
```

2.2.4 DeptBean – The Bean Class

The DeptBean class is an entity bean that uses bean-managed persistence. This means that all methods for persistence and all finder methods must be explicitly implemented in the class. In addition to the callback methods, we must also define EJB implementations for most of the methods defined in the Dept and DeptHome interfaces.

```
import java.lang.StringBuffer;
import java.rmi.RemoteException;
import java.sql.*;
import javax.ejb.*;
import javax.naming.*;
import javax.sql.DataSource;

public class DeptBean implements javax.ejb.EntityBean
{
  private static final String DATABASE_NAME = "java:comp/env/jdbc/ejbIntro";

  public EntityContext mCtx = null;
  public int mDeptNo;
  public String mDName;
  public String mLoc;

  public DeptBean()
  {
  }

  public DeptPK ejbCreate(
      int deptNo
    , String dName
    , String loc
  )
    throws CreateException
  {
    if ( deptNo < 0)
      throw new CreateException("Invalid parameter");
    mDeptNo = deptNo;
    mDName = dName;
    mLoc = loc;

    Connection con = null;
    PreparedStatement ps = null;
    StringBuffer sb = new StringBuffer( 128 );
    try
    {
      con = this.getConnection();
      ps = con.prepareStatement( sb
        .append( "INSERT INTO dept (" )
        .append( " deptno" )
        .append( ",dname" )
        .append( ",loc" )
        .append( ") VALUES (?,?,?)" )
        .toString()
      );
      ps.setInt( 1, deptNo );
      ps.setString( 2, dName );
      ps.setString( 3, loc );
      if ( ps.executeUpdate() != 1 )
      {
        throw new CreateException( "ejbCreate: Failed to add dept to database");
      }
      DeptPK primaryKey = new DeptPK();
      primaryKey.setDeptNo( deptNo );
      return primaryKey;
    }
```

```
    catch ( SQLException sqlEx )
    {
      sqlEx.printStackTrace();
      throw new EJBException( sqlEx.getMessage() );
    }
    finally
    {
      try
      {
        if ( ps != null ) ps.close();
        if ( con != null ) con.close();
      }
      catch ( SQLException sqlEx )
      {
        sqlEx.printStackTrace();
      }
    }
  }

  // {...} full implementation see www.akadia.com
}
```

To get access to the database in the `getConnection()` method using JDBC we simply request a connection from a DataSource, which we obtain from the JNDI environment naming context (ENC). In EJB 1.1, every bean has access to its JNDI ENC, which is part of the bean-container contract. In the beans deployment descriptor, resources such as the JDBC DataSource.

```
<enterprise-beans>
  <entity>
    <resource-ref>
      <description/>
      <res-ref-name>jdbc/ejbIntro</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
    </resource-ref>
  </entity>
</enterprise-beans>
```

In this example we are declaring that the JNDI name "`jdbc/ejbIntro`" refers to a `javax.sql.DataSource` resource manager. The JNDI name specified in the deployment descriptor is always relative to the JNDI ENC context name, "`java:comp/env`".

2.3 Developing a Session Bean

As you learned in the previous chapters, entity beans provide an object-oriented interface that makes it easier for developers to create, modify, and delete data from the database.

However, entity beans are not the entire story. We have also seen another kind of enterprise bean: the session bean. Session beans are useful for describing interactions between other beans (workflow) or for implementing particular tasks. Unlike entity bean, session beans don't represent shared data in the database, but they can access shared data too.

Session beans are divided into tow basic types: *stateless* and *stateful*. A *stateless* session bean is a collection of related services, each represented by a method; the bean maintains no state from one method invocation to the next.

A *stateful* session bean is an extension of the client application. It performs tasks on behalf of the client and maintains state related to that client. This state is called *conversational state* because it represents a continuing conversation between the stateful session bean and the client. Methods invoked on a stateful session bean can write and read data to and from this conversational state, which is shared among all methods in the bean.

The following chapters describe the already known EJB interfaces or classes for a stateless session bean, the CompanyAgent bean. This agent bean supports client functionality such as

- creating a XML representation of the employee data,
- searching for all employees within a given department,
- saving employee data, i.e. inserting new data or updating existing data,
- and deleting an entire department including its employees within one transaction.

2.3.1 CompanyAgent – The Remote Interface

A stateless session bean, like any other bean, needs a remote interface. We can identify the mentioned methods to fulfil the required functionality.

```
import java.util.Date;
import java.rmi.RemoteException;

 public interface CompanyAgent extends javax.ejb.EJBObject
{
  public String empToXml( int empNo )
    throws RemoteException;

  public int getNofEmps( int deptNo )
    throws RemoteException;

  public void saveEmp(
      int empNo
    , String eName
    , Date hireDate
    , int deptNo
  )
    throws RemoteException;

  public void deleteDept(
      int deptNo
  )
    throws RemoteException;

}
```

2.3.2 CompanyAgentHome – The Home Interface

The home interface of all stateless session bean contains one `create()` method with no arguments. This is a requirement of the EJB specification. Here is the definition for the CompanyAgent bean:

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

public interface CompanyAgentHome extends javax.ejb.EJBHome
{
  public CompanyAgent create()
    throws RemoteException, CreateException;
}
```

2.3.3 CompanyAgentBean – The Bean Class

This bean represents a set of independent operations that can be invoked and then thrown away – another indication that it's a good candidate for a stateless session bean.

- The method `empToXml()` uses the already defined entity bean Emp.
- The method `getNofEmps()` shows the direct data access within the session bean.
- The method `saveEmp()` checks for existing data through the Emp entity bean and inserts or updates the data.
- The method `deleteDept()` uses the already defined entity beans Emp and Dept within one single transaction.

```
import java.io.*;
import java.rmi.RemoteException;
import java.sql.*;
```

```
import java.util.*;
import javax.ejb.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import javax.sql.DataSource;
import com.akadia.ejbintro.dept.*;
import com.akadia.ejbintro.emp.*;

public class CompanyAgentBean implements javax.ejb.SessionBean
{
  private static final String DEPT_HOME_NAME="java:comp/env/ejb/TheDeptHomeAlias";
  private static final String EMP_HOME_NAME ="java:comp/env/ejb/TheEmpHomeAlias";
  private static final String DATABASE_NAME ="java:comp/env/jdbc/ejbIntro";

  private DataSource mDataSource = null;
  private DeptHome mDeptHome = null;
  private EmpHome mEmpHome = null;
  private SessionContext mCtx = null;

  public CompanyAgentBean()
  {
  }

  public void saveEmp(
      int empNo
    , String eName
    , Date hireDate
    , int deptNo
  )
  {
    Emp emp = null;
    try
    {
      try
      {
        emp = getEmpHome().findByPrimaryKey(new EmpPK( empNo ) );
      }
      catch ( javax.ejb.FinderException ex )
      {
        emp = null;
      }
      if ( emp != null )
      {
        if ( eName != null) emp.setEName( eName );
        if ( hireDate != null) emp.setHireDate( hireDate );
        if ( deptNo != -1 ) emp.setDeptNo( deptNo );
      }
      else
      {
        emp = getEmpHome().create(
            empNo
          , eName
          , hireDate
          , deptNo
        );
      }
    }
    catch ( java.rmi.RemoteException ex )
    {
      throw new EJBException( "RemoteException saving emp: "
                              + ex.getMessage() );
    }
    catch ( javax.ejb.CreateException ex )
    {
      throw new EJBException( "CreateException saving emp: "
                              + ex.getMessage() );
    }
  }

  // {...} full implementation see www.akadia.com
}
```

3 Deploying Enterprise Beans

Much of the information about how beans are managed at runtime is not addressed in the interfaces and classes. Deployment descriptors serve a function very similar to

property files. They allow us to customise behaviour of enterprise beans at runtime without having to change the software itself.

When a bean class and its interfaces have been defined, a deployment descriptor for the bean is created and populated with data about the bean. IDEs will allow developers to set up the deployment descriptors for a bean. The descriptor is saved to a file and can be packaged in a JAR file for deployment.

After the development and compiling tasks, the beans must be assembled, deployed, and configured eventually.

3.1 Assembler Task

- Define the bean's persistent type (entity beans only).
- Define the bean's state management (session beans only).
- Define datasources, e.g. JDBC connections.
- Define references, e.g. aliases to resolve entity beans.
- Define the transaction attribute for each method.

3.2 Deployer Task

- Define the jar file in the `[SaEjbJars]` section of your sajava.ini file.
- Add the settings to enable database connectivity in the section `[DataSources]` of your sajava.ini file.

See Appendix A for an example of the sajava.ini entries.

3.3 CMP Configuration Task

This task is used for Container-Managed Persistence (CMP) Beans only.

- All fields of the entity bean are mapped to the database.
- Use the Data Source `jdbc/ejbIntro` for database connectivity.
- Map all fields of the entity bean with the attributes of the appropriate table.
- Define the filters for the given find-methods.
- Generate the map, the result will be stored in the jar file.

4 Using Enterprise Beans

Once the enterprise beans are developed, assembled, and deployed, the EJB server supports the beans' functionality to the clients. In this chapter some experiences and suggestions are discussed.

4.1 Using Entity Beans Directly

The simplest – but not always best – way is to directly access the entity bean through its remote interface. This can be done in a simple Java application, in a Java Server Page (JSP), or in a Tag Library. The following steps are needed:

1. Get the bean's home from the EJB server: Depending on the server this can be done through RMI or JNDI lookup of the named context.

2. Based on the bean's home the methods of the home interface can be invoked.

3. Depending on the result of the called method above, the bean can be managed e.g. through its remote interface to manage the bean's data.

The following table shows the executed method calls on server side initiated by a simple `findByPrimaryKey()` call:

Client	Server

`DeptPK pk = new DeptPK(deptNo);`
`DeptHome deptHome = getDeptHome();`
`Dept dept =`
`deptHome.findByPrimaryKey(pk);`
	`DeptBean: ejbFindByPrimaryKey`

`dept.getDeptNo();`	`DeptBean: ejbLoad`
	`DeptBean: getDeptNo`
	`DeptBean: ejbStore`

`dept.getDName();`	`DeptBean: ejbLoad`
	`DeptBean: getDName`
	`DeptBean: ejbStore`

`dept.getLoc();`	`DeptBean: ejbLoad`
	`DeptBean: getLoc`
	`DeptBean: ejbStore`

The call to the `findByPrimaryKey()` method is straight forward executed on server side. The subsequent calls to get the information such as department number, department name, and location enforces the EJB server to load the bean, get the appropriate information, and stores(!) the information immediately. At least the last step seems to be surprisingly.

The EJB server handles each getXxx() method call as single, independent request. The server loads the bean (again) to get the most recent information and stores this information back because it could have changed during the method processing of the bean's getXxx() method and because of the object's life cycle.

This behaviour seems to be some kind of overkill: keep in mind that every data selection is immediately followed by an update statement in the store method. Moreover this behaviour is the same for CMP and BMP beans, but CMP beans (which are generated by J2EE) avoid the unnecessary update. Only when the bean's data has changed the update takes place on database.

To avoid this on BMP beans too, some additional thoughts must be made. Some common ideas:

- Use a data model class instead of single properties: Instead of getting each attribute by a single data selection we can fetch all the data once and fill it into the model representing the entity. This solution has not only the advantage of one single access to the database but also reduces client-server communication dramatically. Image that we only need one call to get one (slightly bigger) information as answer.

- Use a "data changed" or "dirty data" mechanism to avoid unnecessary updates: We can not avoid the call to the `ejbStore()` method (since this is defined by the EJB specification) but we can omit the update statement on database whenever we are sure that the bean's data has not changed.

4.2 Using Entity Beans Through Session Beans

In contrast accessing the entity bean directly we discuss here the usage of session beans. The following steps are needed:

1. Get the session bean's home from the EJB server: Depending on the server this can be done through RMI or JNDI lookup of the named context.

2. Based on the bean's home the `create()` method is invoked to create the bean.

3. Now all methods of the session bean are available. In our case we can create a XML representation of the employee data, searching for all employees within a given

department, saving employee data, and deleting an entire department including its employees.

4. When the session bean has done its work we can remove it. This can be done by calling the bean's `remove()` method.

The following table shows the executed methods calls on server side initiated by the supported methods:

Client	Server

`xml = mCompanyAgent.empToXml(empNo);`	`EmpBean: ejbFindByPrimaryKey`
	`EmpBean: ejbLoad`
	`EmpBean: getEmpNo`
	`EmpBean: getEName`
	`EmpBean: getHireDate`
	`EmpBean: getDeptNo`
	`EmpBean: ejbStore`

`cnt = mCompanyAgent.getNofEmps(deptNo);`	`direct db access, no entity beans involved`

`mCompanyAgent.saveEmp(`
`empNo`
`, eName`
`, hireDate`
`, deptNo`
`);`
	`EmpBean: ejbFindByPrimaryKey`
	`EmpBean: ejbLoad`
	`EmpBean: setEName`
	`EmpBean: setHireDate`
	`EmpBean: setDeptNo`
	`EmpBean: ejbStore`

`mCompanyAgent.deleteDept(deptNo);`	`EmpBean: ejbRemove`
	`EmpBean: ejbRemove`
	`...`
	`DeptBean: ejbFindByPrimaryKey`
	`DeptBean: ejbLoad`
	`DeptBean: ejbRemove`

The call to `empToXml()` method shows the expected result: Instead of multiple reloads of the entity bean all the information can be handled in the same transaction. Nevertheless we have the final store method call but the CMP bean is clever enough to avoid the update on database due to no data cache changes.

The call to `getNofEmps()` shows the direct database access through the session bean. Another nice possibility to do some database actions within a given transaction!

In the call to `saveEmp()` method some of the entity attributes are updated. Again the entity bean is loaded once and then its attributes are updated but only once stored on database. Please note that this is done within one single transaction!

Finally the call to the `deleteDept()` method deletes first all employees of a given department. After this the department itself is deleted. Please note that all deletes are done within one single transaction!

4.3 Conclusion

We have discussed now some basics about Enterprise JavaBeans. With these ideas in mind it should be possible to do the first steps with J2EE technology. Let us summarise some relevant issues:

- CMP beans are developed very fast but need some more effort during runtime due to field mapping. Moreover we have minor possibility to influence data access or sort order (i.e. the SQL statement itself will be generated automatically).

- CMP beans have a sophisticated data cache mechanism. This leads to minimal database access to avoid additional data requests.

- BMP beans need more effort from developer's side. But the developer can tune and optimise the data access directly.

- BMP beans need a data cache mechanism or “dirty data” mechanism too. This mechanism must be designed and implemented by the developer him-/herself.

- Session beans open the full functionality and support transaction handling in a very simple way. It is the only way to gain the full power of entity beans!

Appendix A: sajava.ini Example

```
[SaEjbJars]
IntroBeans=<full path>\ejbIntro.jar

[DataSources]
jdbc/ejbIntro=jdbc/ejbIntro

[jdbc/ejbIntro]
type=javax.sql.DataSource
driver=oracle.jdbc.driver.OracleDriver
vendorUrl=jdbc:oracle:thin:@diamond:1523:DIA3
vendor=oracle
vendorName=oracle
serverName=jdbc/ejbIntro
databaseName=DIA3
schema=scott
useSchema=false
username=scott
user=scott
password=tiger
connectString=jdbc:oracle:thin:@diamond:1523:DIA3
```

Contact

Christoph Gächter
E-Mail christoph.gaechter@akadia.com

Akadia AG
Information Technology
Arvenweg 4
CH-3604 Thun
Phone 033 335 86 20
Fax 033 335 86 25
E-Mail info@akadia.com
Web www.akadia.com