

Department of Computer
and Systems Sciences
SU / KTH

COMPONENT BASED DEVELOPMENT WITH ENTERPRISE JAVABEANS COMPENDIUM

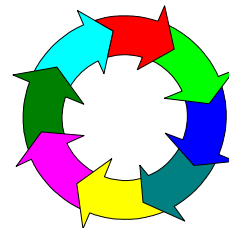
v. 1.0

IS7/2I1404

Model driven component based development

Autumn 2003

<http://L238.dsv.su.se:86/courses/IS7/>



nikos dimitrakas



Table of contents

1 INTRODUCTION	3
1.1 HOMEPAGE.....	3
1.2 THE ENVIRONMENT	3
2 JBOSS AND EJB	4
2.1 SETTING UP JBOSS.....	4
2.2 DEPLOYING EJBS	5
3 DATABASE.....	5
4 EXERCISES.....	8
4.1 HELLO WORLD.....	9
4.2 DATABASE EJBS	14
4.2.1 <i>Data Classes</i>	15
4.2.2 <i>Retrieving Data</i>	17
4.2.3 <i>Manipulating Data</i>	29
4.3 ASSIGNMENTS	34
4.4 WHEN THINGS HAVE GONE BAD!	35
5 INTERNET RESOURCES	35
6 EPILOGUE	35

Table of figures

FIGURE 1 BOOK DATABASE.....	6
FIGURE 2 HELLO WORLD SYSTEM OUTLINE	9
FIGURE 3 BOOKMGR SYSTEM OUTLINE.....	19

1 Introduction

This compendium contains the following:

- An introduction to the JBoss environment and its facilities for deploying and running EJBs
- A short presentation of the database used by some of the EJBs in the exercises
- Step-by-step exercises for creating, deploying, running and testing EJBs
- Assignments

It is strongly recommended that you read through the entire compendium before beginning with the exercises. It is of course necessary to have some basic understanding of Microsoft Windows, Relational Databases (and SQL), Java (including JDBC), Component Based Development (CBD) and the EJB architecture/component model. For the last two the following reading is recommended prior to reading this compendium:

- Lecture notes on CBD and EJB by Martin Henkel
- The J2EE Tutorial by SUN

1.1 Homepage

Information about this compendium can be found here:

<http://L238.dsv.su.se:86/courses/IS7>

The following can be found at this address:

- Files
The latest version of the compendium and all the files needed to complete the exercises in the compendium.
- FAQ
Here there is a list of corrections and explanations.
- Links
Internet resources that can be helpful when working with the compendium.

1.2 The environment

For completing the exercises in this compendium we will use the following facilities/software:

- Lite version of JBoss that is used to run EJBs
- MSAccess used for the example database
- ODBC driver/manager used for accessing the database from the EJBs
- Java tools (compiler, jar)
- Command Prompt (execution of commands, etc.)
- Text editor (of your choice) for editing of batch files, java source code and xml files

Most of this environment does not require any particular configuration. JBoss needs some configuration. How to set up the necessary environment for JBoss is described in chapter 2.

2 JBoss and EJB

In this chapter we will see how to set up the necessary environment for deploying and running the EJBs that we will create later. We will also take a look on the specific details of JBoss that are relevant for EJB deployment.

2.1 Setting up JBoss

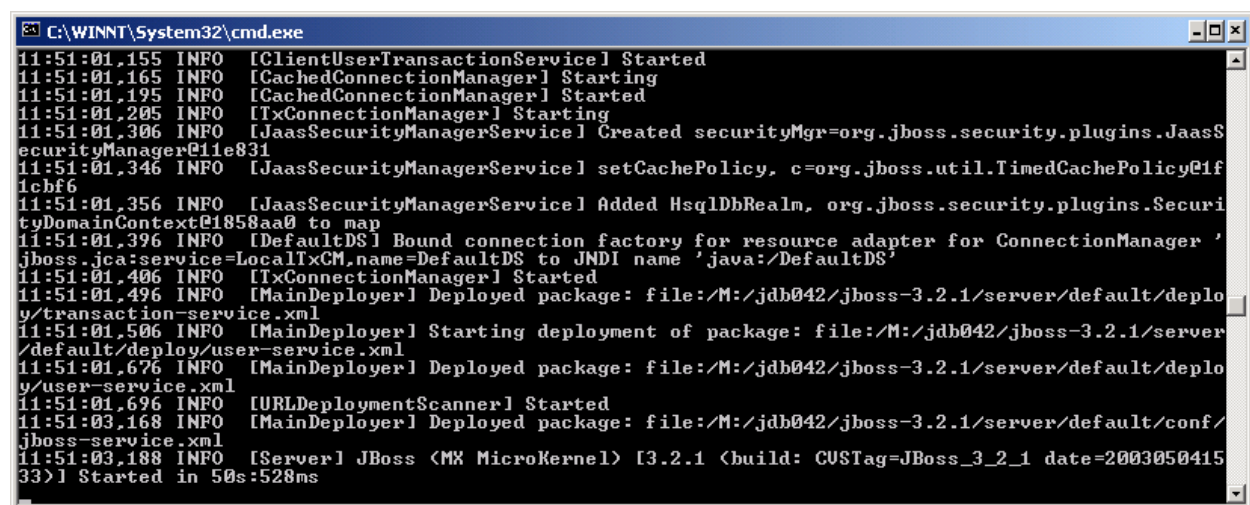
The standard version of JBoss is quite large since it includes a lot more than what we need for the exercises in this compendium. Therefore we have created a lite version that only includes the necessary components (the JBoss Java engine and the EJB components). This version of JBoss is zipped in a file named JBossIS7.zip and can be downloaded from the following locations:

- \\DB-SRV-1\StudKursInfo\IS7 ht2003\CBDwithEJB\JBossIS7.zip
- <http://L238.dsv.su.se:86/courses/IS7/JBossIS7.zip>

By just extracting the contents of the file at your home directory (normally found under M:), you get a working JBoss installation.

To start the JBoss server just execute the file `run.bat` that is located in the directory `jboss-3.2.1\bin` (relative to where you extracted the `JBossIS7.zip`). In the rest of this compendium we will assume that JBoss resides at `M:\jdb042\jboss-3.2.1`. In this case `jdb042` represents the current user-name. You will simply have to replace `jdb042` with your user-name to acquire the correct paths.

Starting the JBoss server takes approximately 1 minute. A command prompt window (JBoss's standard output) will during this time show the progress of loading the server and deploying configuration files and other java components (for example the EJB infrastructure). When the server has finished loading a message will appear stating that JBoss has started.



```
C:\WINNT\System32\cmd.exe
11:51:01.155 INFO [ClientUserTransactionService] Started
11:51:01.165 INFO [CachedConnectionManager] Starting
11:51:01.195 INFO [CachedConnectionManager] Started
11:51:01.205 INFO [TxConnectionManager] Starting
11:51:01.306 INFO [JaasSecurityManagerService] Created securityMgr=org.jboss.security.plugins.JaasSecurityManager@1e831
11:51:01.346 INFO [JaasSecurityManagerService] setCachePolicy, c=org.jboss.util.TimedCachePolicy@1f1cbf6
11:51:01.356 INFO [JaasSecurityManagerService] Added HsqlDbRealm, org.jboss.security.plugins.SecurityDomainContext@1858aa0 to map
11:51:01.396 INFO [DefaultDS] Bound connection factory for resource adapter for ConnectionManager 'jboss.jca:service=LocalTxCM,name=DefaultDS to JNDI name 'java:/DefaultDS'
11:51:01.406 INFO [TxConnectionManager] Started
11:51:01.496 INFO [MainDeployer] Deployed package: file:/M:/jdb042/jboss-3.2.1/server/default/deploy/transaction-service.xml
11:51:01.506 INFO [MainDeployer] Starting deployment of package: file:/M:/jdb042/jboss-3.2.1/server/default/deploy/user-service.xml
11:51:01.676 INFO [MainDeployer] Deployed package: file:/M:/jdb042/jboss-3.2.1/server/default/deploy/user-service.xml
11:51:01.696 INFO [URLDeploymentScanner] Started
11:51:03.168 INFO [MainDeployer] Deployed package: file:/M:/jdb042/jboss-3.2.1/server/default/conf/jboss-service.xml
11:51:03.188 INFO [Server] JBoss <MX MicroKernel> [3.2.1 <build: CUSTag=JBoss_3_2_1 date=200305041533>] Started in 50s:528ms
```

This window is now locked by JBoss. To stop the server press Ctrl-C. Closing the window will have a similar effect to pressing Ctrl-C, but windows may start complaining of the process not responding. It is therefore best to use Ctrl-C to shut down the JBoss server.

JBoss will use this window for any message during run-time. For example EJB deployment done while the server is running will produce a message in this window.

At this point it is also important to know that the JBoss server listens on port 1099. The server should have confirmed this during start-up with this message:

```
INFO [NamingService] Listening on port 1099
```

This is very useful information that we need when we later program java programs that need to access the JBoss server.

2.2 Deploying EJBs

To deploy an EJB in JBoss we need a jar file containing

1. The EJB class
2. The applicable interfaces for the EJB
3. All necessary helper classes
4. A deployment descriptor for the EJB

The jar file only needs to be placed in the `jboss-3.2.1\server\default\deploy` directory. The JBoss Server will then deploy it automatically. The reverse is also possible: Removing a deployed jar file from this directory will cause JBoss to undeploy it.

3 Database

In the exercises in chapter 4 we will create EJBs. Some of them will provide business logic that requires a database. To illustrate this database functionality we will use a sample database. This chapter describes this database. The database is an MS Access database named `book.mdb`. The database file can be downloaded from

- `\\DB-SRV-1\StudKursInfo\IS7 ht2003\CBDwithEJB\book.mdb`
- `http://L238.dsv.su.se:86/courses/IS7/book.mdb`

Download a copy of the database and place it on your home directory!

The figure bellow shows the tables included in the database and their relationships:

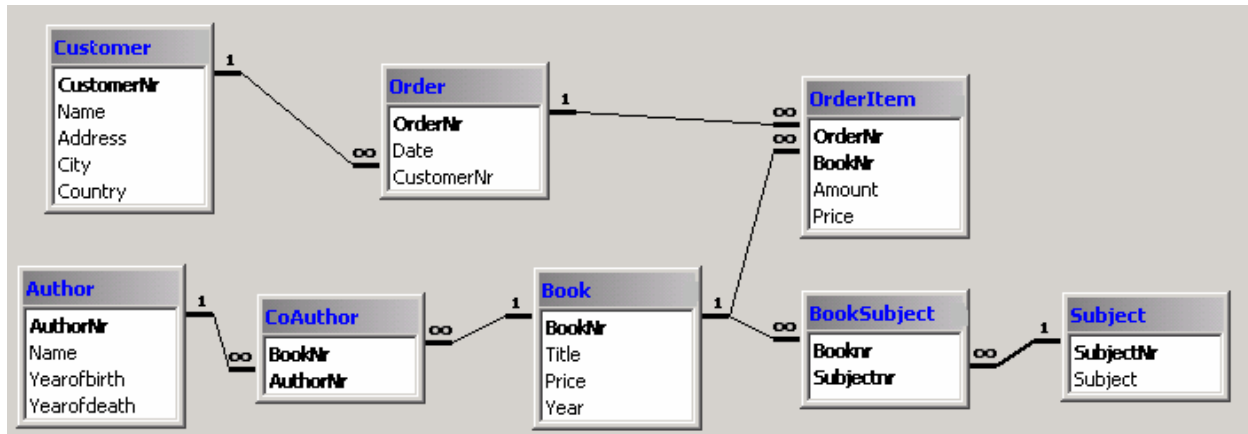


Figure 1 Book database

The main entities of the database are the customers and the books. Customers place orders that contain one or more orderitems. Each orderitem represents one book in one or more copies (attribute amount). Each book has a title, a year (of publishing) and a current price. (Since this is the current price the actual price at the time of an order is stored in OrderItem.) Each book has one or more authors and one or more subjects.

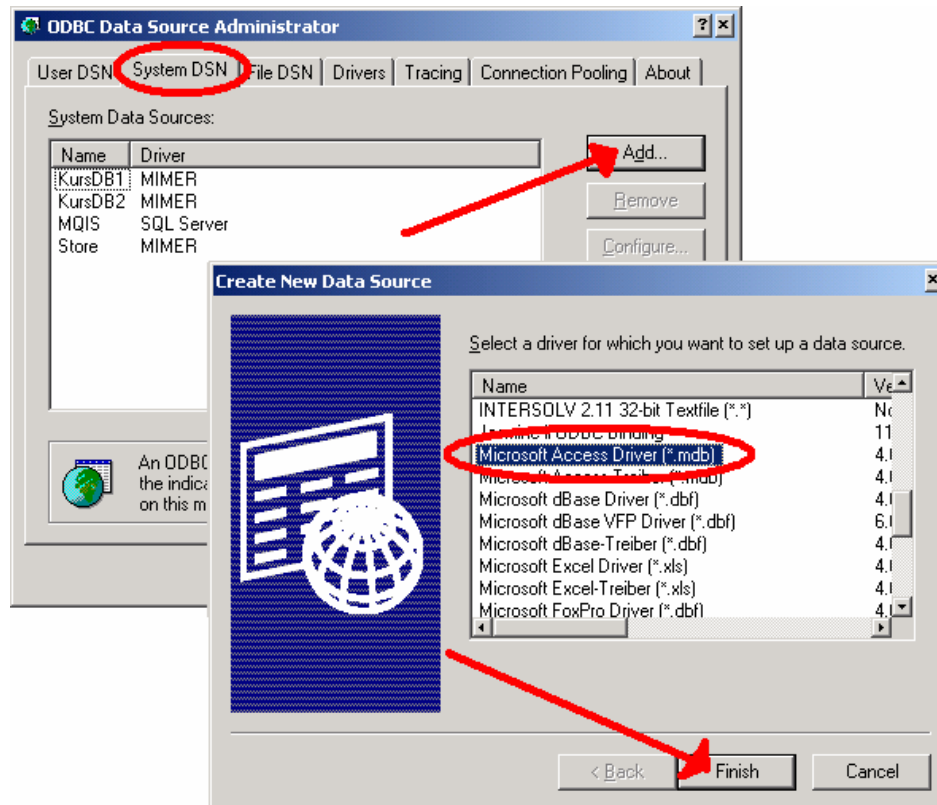
The database is populated with enough data for our simple testing purposes. MS Access can be used to browse and edit the database.

In order to connect to an MS Access database from a java program we need a driver. Since there is no native java driver for MS Access we will use an ODBC driver. In order to make our database available through an ODBC driver, we have to register it with the ODBC Data Source Administrator that is part of Windows. To invoke the ODBC Data Source Administrator execute the following file (for example through Start→Run...):

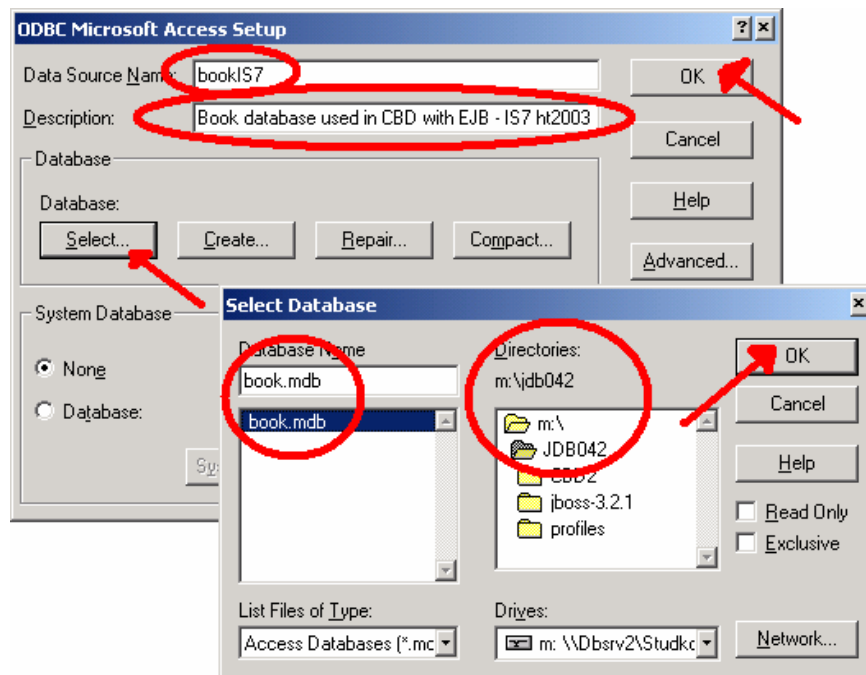
C:\WINNT\system32\odbcad32.exe

This will bring you to the ODBC Data Source Administrator.

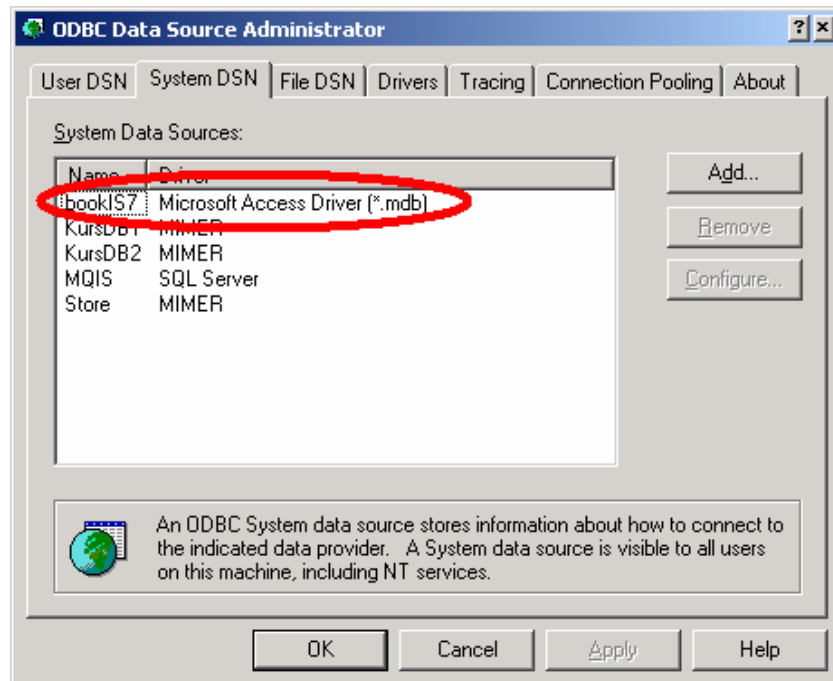
Create an ODBC alias (also known as DSN – Data Source Name) in the System DSN tab:



You can now give a name and a short description to your DSN and also select to database file associated to this DSN:



The new DSN should now be available under System DSN:



The database is now available through an ODBC driver and it is mapped to the alias bookIS7. In the next chapter we will see how to configure our java program to access this ODBC alias.

4 Exercises

In this chapter we will go through the entire process of creating, deploying and running/testing 3 EJBs. The first one (section 4.1) will be a simple "Hello World" EJB, while the next two (section 4.2) will provide some basic database functionality. All the files needed for the exercises in this chapter as well as the result files of the exercises are available here:

- \\DB-SRV-1\StudKursInfo\IS7 ht2003\CBDwithEJB
- <http://L238.dsv.su.se:86/courses/IS7>

The files used in the exercises in this chapter can be reused as templates for the assignments and the project work!

The exercises that follow contain quite a lot of java code. A good way to work with the step-by-step descriptions of creating the necessary java files is to open this compendium in MS Word and then copy and paste the java code from the compendium into the appropriate java files. Another possibility is, of course, to just download the complete files one by one and place them in the appropriate directories.

The exercises also include compiling, packaging, copying and running files. There are batch files that help with those operations and they are also available for download. These batch files need to be edited so that the correct home directory is defined.

4.1 Hello World

In this exercise we will create an EJB component that simply returns a string "Hello World" when its only business method `hello()` is called. Our EJB will be a stateless session bean and will only allow remote access. The following figure outlines the structure of the "full Hello World system":

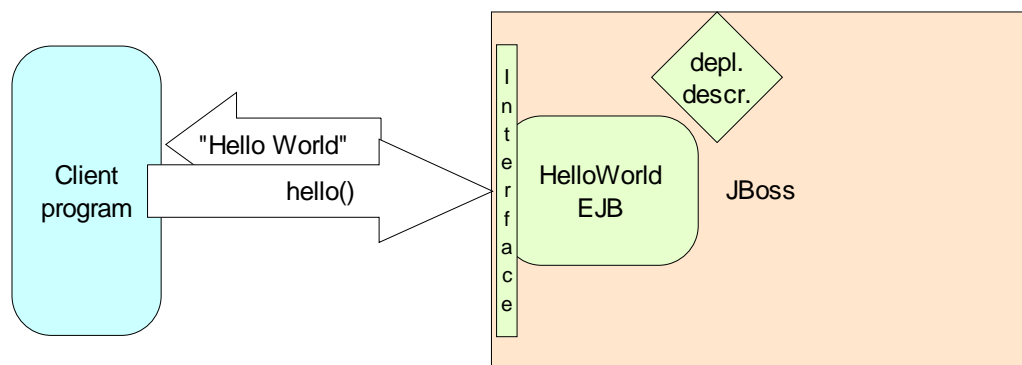
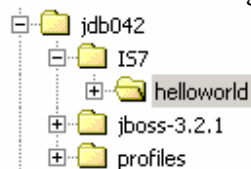


Figure 2 Hello World system outline

In short here is what we have to do:

- Create a java class `HelloWorldBean` where the method `hello()` will be implemented.
- Create a home interface `HelloWorldHome` with the signatures of the create methods.
- Create a remote interface `HelloWorld` with the signatures of the business methods.
- Create a deployment descriptor file `ejb-jar.xml`.
- Package our EJB in a jar file and deploy it.
- Create a test client for our EJB.
- Run the test client.

Before we begin programming we have to decide a package structure. As the root package we will use `IS7`. Under this we will have a package `helloworld` where all the files for the EJB will be. So we have to create two directories `IS7` and `helloworld`. We can put them under `M:/jdb042` so that we have the following structure:



In the directory `helloworld` we will place the interfaces and the class of the EJB and directory `META-INF` where the deployment descriptor will be placed.

So let's start with the class `HelloWorldBean`:

1. Create a file HelloWorldBean.java (in IS7\helloworld) and open it for editing (for example in SciTE or NetObjects ScriptBuilder)!
2. Define the package: `package IS7.helloworld;`
3. Define the class: `public class HelloWorldBean implements javax.ejb.SessionBean {}`
4. The session bean must implement at least one `ejbCreate()` method (A stateless session bean can only contain one `ejbCreate()` method which cannot take any arguments). In our case it's enough with an empty one: `public void ejbCreate() {}`
5. We also need to implement our business method `hello()`:

```
public String hello()
{
    return "Hello World!";
}
```
6. All session beans must also implement the following methods (that in this case can be left empty):

```
public void ejbRemove() {}
public void ejbActivate() {}
public void ejbPassivate() {}
public void setSessionContext(javax.ejb.SessionContext ctx) {}
```

The session bean HelloWorldBean is now complete and can be compiled! In order to compile it we must specify where the necessary EJB framework classes can be found:
M:\jdb042\jboss-3.2.1\server\default\lib\jboss-j2ee.jar
7. In a new command prompt window move to your home directory (`cd M:\jdb042` and `M:`) and compile the session bean with the following command:
`javac IS7\helloworld*.java -classpath jboss-3.2.1\server\default\lib\jboss-j2ee.jar`

Next we have to create a home interface for our session bean:

8. Create a file HelloWorldHome.java (in IS7\helloworld) and open it for editing!
9. Define the package: `package IS7.helloworld;`
10. Define the interface: `public interface HelloWorldHome extends javax.ejb.EJBHome {}`
11. This interface must contain a `create()` method for each `ejbCreate()` method in the class HelloWorldBean. These `create()` methods, unlike the `ejbCreate()` methods, must return an instance of the remote interface HelloWorld (that we have not yet defined) and must also throw a `RemoteException` and a `CreateException`:

```
HelloWorld create() throws java.rmi.RemoteException,
javax.ejb.CreateException;
```
12. Our home interface is now ready, but we cannot compile it until we have the remote interface.

Naturally we must now define the remote interface `HelloWorld`:

13. Create a file `HelloWorld.java` (in `IS7\helloworld`) and open it for editing!

14. Define the package: `package IS7.helloworld;`

15. Define the interface: `public interface HelloWorld extends javax.ejb.EJBObject {}`
This interface must contain the signatures of all business methods defined in the session bean. All the business methods must in this interface throw a `RemoteException`. In our case there is only the method `hello()`:
`public String hello() throws java.rmi.RemoteException;`

16. Now that both interfaces are ready and in place we can compile them (using the same compilation command as before).

The final component of our EJB before we can deploy it is the deployment descriptor. This is a simple xml file that tells the JBoss server the names and locations of the session bean and its interfaces.

17. Create a directory `META-INF` (in `IS7\helloworld`)!

18. Create a file `ejb-jar.xml` (in `IS7\helloworld\META-INF`) and open it for editing!

19. This is an xml document so it has to start with the two standard xml directives. (The DOCTYPE for the EJB deployment descriptor is defined by SUN. The full dtd can be found at

```
http://java.sun.com/dtd/):  
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE ejb-jar  
  PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"  
  'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>
```

20. The root element is `<ejb-jar>` and it contains all the necessary elements for defining the components of one or more EJBs. The main two sub-elements are `<enterprise-beans>` and `<assembly-descriptor>`. The first one is where the components and type of the EJBs are specified, while the second one contains security and transaction information for the EJBs and their methods. The elements `<description>` and `<display-name>` can also provide useful information. Here is the content of the `<ejb-jar>` element for the `HelloWorld` EJB:

```
<description>  
  Hello World EJB as part of the CBD with EJB compendium for IS7 ht2003  
</description>  
<display-name>Hello World EJB</display-name>  
<enterprise-beans>  
  <session>  
    <ejb-name>HelloWorld</ejb-name>  
    <home>IS7.helloworld.HelloWorldHome</home>  
    <remote>IS7.helloworld.HelloWorld</remote>
```

```
<ejb-class>IS7.helloworld.HelloWorldBean</ejb-class>
<session-type>Stateless</session-type>
<transaction-type>Container</transaction-type>
</session>
</enterprise-beans>
<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>HelloWorld</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
</assembly-descriptor>
```

This defines a stateless (<session-type>) session EJB (<session>) with container managed transactions (<transaction-type>). It also defines that all the methods (<method-name>) must be within transactions (<trans-attribute>).

21. With the deployment descriptor file ready we have all necessary files for deploying the HelloWorld EJB. The only thing we need to do is put all the files in a jar file HelloWorldJAR.jar and copy it to the deployment area of JBoss (the directory jboss-3.2.1\server\default\deploy). We do this with the following commands:

```
jar cMvf HelloWorldJAR.jar IS7\helloworld\*.class
jar uMvf HelloWorldJAR.jar -C IS7\helloworld META-INF\ejb-jar.xml
copy HelloWorldJAR.jar jboss-3.2.1\server\default\deploy
```

The first command adds the class files to the jar file. The second command adds the deployment descriptor to the jar file. The third one simply copies the jar file to JBoss which automatically deploys it.

22. At this point the HelloWorld EJB is available to clients. The only thing missing is a test client. In order to develop our test client we need to now the interfaces (home and remote) of the EJB. Since the developer of the EJB and the developer of the clients accessing it aren't necessarily the same, we have to assume that the developer of the client does not have access to the original IS7.helloworld package. Therefore we can create a jar file with the two interfaces and make it available to the client developers. We can call it HelloWorldInterface.jar and we can create it with the following command:

```
jar cMvf HelloWorldInterface.jar IS7\helloworld\HelloWorld.class
IS7\helloworld\HelloWorldHome.class
```

Taking now the role of the client developer we only have access to the HelloWorldInterface.jar. We also know that it is deployed on a JBoss server listening on port 1099. In order to access JBoss and the HelloWorld EJB we need to use the following classes:

```
javax.naming.Context
javax.naming.InitialContext
javax.rmi.PortableRemoteObject
```

1. We can start developing our test client by defining the package and class name. We will use a package `test` and call the class `HelloWorldTestClient`.
2. Create a directory `test` at your home directory (in our case `M:\jdb042\test`)!
23. Create a new file `HelloWorldTestClient.java` (in the directory `test`) and open it for editing!
24. Define the package: `package test;`
25. Import the necessary classes:

```
import javax.naming.Context;  
import javax.naming.InitialContext;  
import javax.rmi.PortableRemoteObject;  
import IS7.helloworld.*;
```
26. Define the class: `public class HelloWorldTestClient {}`
27. We only need a `main()` method, so we can start by defining it:

```
public static void main(String[] args) { }
```
28. Inside the `main()` method we will need to first establish a context (a description of how to access the server), then using this context look up the `HelloWorld EJB`, then request an instance of the session bean on which we can finally call the business method `hello()`. We start by creating a context and setting up its environment (We do all this within a `try` clause since there are possible exceptions). The values below are adjusted for the current configuration of `JBoss` and only for running the client on the same machine as the `JBoss` server:

```
try  
{  
    Context ctx = new InitialContext();  
    ctx.addToEnvironment(Context.INITIAL_CONTEXT_FACTORY,  
                          "org.jnp.interfaces.NamingContextFactory");  
    ctx.addToEnvironment(Context.PROVIDER_URL, "127.0.0.1:1099");  
    //instead of 127.0.0.1 you can use localhost  
    ctx.addToEnvironment("java.naming.factory.url.pkgs",  
                          "org.jboss.naming:org.jnp.interfaces");  
}
```
29. Still inside the `try` block we have to ask the context to look up the `HelloWorld EJB`. The context will give us then an object which we can (in a bit unusual way) cast into a `HelloWorldHome` instance. Using this instance we can call the `create()` method to acquire an instance of `HelloWorld`:

```
Object obj = ctx.lookup("HelloWorld");  
HelloWorldHome home = (HelloWorldHome)  
    PortableRemoteObject.narrow(obj, HelloWorldHome.class);  
HelloWorld helloWorld = home.create();
```
30. The last thing to do before the end of the `try` block is to call the business method `hello()`. We can for example print the result of the `hello()` method:

```
System.out.println(helloWorld.hello());
```

31. We can now finish the try block and add a catch block to print any unexpected exception:

```
} //end of try block
catch (Exception ex)
{
    System.err.println("Caught an unexpected exception!");
    ex.printStackTrace();
}
```

32. We are now ready to compile and run our test client. In order to compile the test client we need the classes that we have imported and some classes included in the EJB framework. All the classes are available in the following two jar files:

```
HelloWorldInterface.jar
jboss-3.2.1\client\jbossall-client.jar
    (or jboss-3.2.1\server\default\lib\jboss-j2ee.jar)
```

We can compile our test client with the following command:

```
javac      test\*.java      -classpath      jboss-3.2.1\client\jbossall-
client.jar;HelloWorldInterface.jar
```

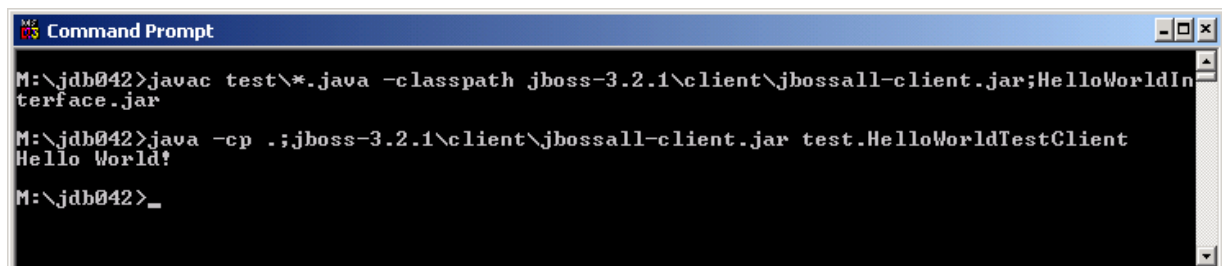
33. To run the test client we only need to have the classes necessary for the environment of the context. They are all included in the following jar file:

```
jboss-3.2.1\client\jbossall-client.jar
```

We can run our test client with the following command:

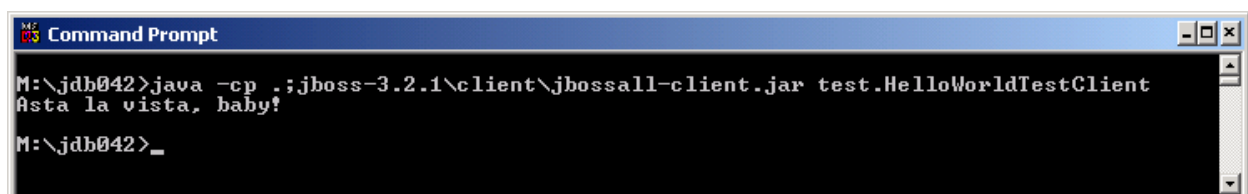
```
java -cp .;jboss-3.2.1\client\jbossall-client.jar test.HelloWorldTestClient
```

Running the test client will cause the message "Hello World!" to be printed:



```
Command Prompt
M:\jdb042>javac test\*.java -classpath jboss-3.2.1\client\jbossall-client.jar;HelloWorldInterface.jar
M:\jdb042>java -cp .;jboss-3.2.1\client\jbossall-client.jar test.HelloWorldTestClient
Hello World!
M:\jdb042>_
```

Should we change something in the implementation of our HelloWorld EJB, the client would not be affected. We can for example change the message returned by the business method hello() to "Asta la vista, baby!" and run the client:



```
Command Prompt
M:\jdb042>java -cp .;jboss-3.2.1\client\jbossall-client.jar test.HelloWorldTestClient
Asta la vista, baby!
M:\jdb042>_
```

4.2 Database EJBs

Now that we have familiarized ourselves with JBoss and to the basic EJB structure, let's try to do something that benefits from the use of EJBs.

In the sections that follow we will create two EJB components that work against the database described in chapter 3. The first one will retrieve data (about books) from the database and the second one will insert a new customer into the database. In order to transfer the data between the client and the server we will need some data classes (so that we can send a book object instead of just strings and other simple objects). In section 4.2.1 we will define those data classes that we will later use when developing the EJBs and test clients.

34. It is also necessary to decide the package structure for our EJBs and data classes. We will use the same root package as before (`is7`) and under this we will create a new package `bookdb`. In this package we will place our data classes and one sub-package for each EJB. We will place the test client in the package `test` (as before).

4.2.1 Data Classes

The database contains 8 tables, so we could think that we need 8 data classes. Since our EJBs are only going to be using book objects and customer objects it is enough to create data classes for those two types of objects.

Let's start with a data class for book objects called `Book`!

1. Create a new file `Book.java` (in `IS7\bookdb`) and open it for editing!
2. Define the package: `package IS7.bookdb;`
3. Define the class: `public class Book implements java.io.Serializable {}`

Notice that it must implement the `java.io.Serializable` Interface in order for the instances to be transmittable between the server and the client!

4. Define a private field for each interesting field/relation in the database:

```
/**private variable for the BookNr of this Book object***/  
private int booknr;  
/**private variable for the Title of this Book object***/  
private String title;  
/**private variable for the Price of this Book object***/  
private int price;  
/**private variable for the Year of this Book object***/  
private int year;
```

Here we could create, for example, vectors of strings for the subjects or the author names, but let's keep it simple. The four fields above will be enough in this exercise.

5. Define getters and setters for the four fields:

```
/**Returns the BookNr of this Book object***/  
public int getBooknr() {return booknr;}  
/**Returns the Title of this Book object***/
```

```
public String getTitle() {return title;}
/**Returns the Price of this Book object***/
public int getPrice() {return price;}
/**Returns the Year of this Book object***/
public int getYear() {return year;}
/**Sets the BookNr for this Book object***/
public void setBooknr(int value) {booknr = value;}
/**Sets the Title for this Book object***/
public void setTitle(String value) {title = value;}
/**Sets the Price for this Book object***/
public void setPrice(int value) {price = value;}
/**Sets the Year for this Book object***/
public void setYear(int value) {year = value;}
```

6. Define the following constructors:

```
public Book() {}
public Book(int booknr, String title, int price, int year)
{
    this.booknr=booknr;
    this.title=title;
    this.price=price;
    this.year=year;
}
```

7. Book.java is now complete and can be compiled with the following command:

```
javac IS7\bookdb\Book.java
```

We can now create the other data class:

8. Create a new file Customer.java (in IS7\bookdb) and open it for editing!

9. Define the package: package IS7.bookdb;

10. Define the class: public class Customer implements java.io.Serializable {}

11. Define a private field for each interesting field/relation in the database:

```
/**private variable for the CustomerNr of this Customer object***/
private int customernr;
/**private variable for the Name of this Customer object***/
private String name;
/**private variable for the Address of this Customer object***/
private String address;
/**private variable for the City of this Customer object***/
private String city;
/**private variable for the Country of this Customer object***/
private String country;
```

Here too, we could create a vector for Order objects, but we won't.

12. Define getters and setters for the five fields:

```
/**Returns the CustomerNr of this Customer object***/
public int getCustomernr() {return customernr;}
/**Returns the Name of this Customer object***/
```



```
public String getName() {return name;}
/**Returns the Address of this Customer object***/
public String getAddress() {return address;}
/**Returns the City of this Customer object***/
public String getCity() {return city;}
/**Returns the Country of this Customer object***/
public String getCountry() {return country;}
/**Sets the CustomerNr for this Customer object***/
public void setCustomernr(int value) {customernr = value;}
/**Sets the Name for this Customer object***/
public void setName(String value) {name = value;}
/**Sets the Address for this Customer object***/
public void setAddress(String value) {address = value;}
/**Sets the City for this Customer object***/
public void setCity(String value) {city = value;}
/**Sets the Country for this Customer object***/
public void setCountry(String value) {country = value;}
```

13. Define the following constructors:

```
public Customer() {}
public Customer(int customernr, String name, String address, String city,
String country)
{
    this.customernr=customernr;
    this.name=name;
    this.address=address;
    this.city=city;
    this.country=country;
}
```

14. Customer.java is now complete and can be compiled with the following command:

```
javac IS7\bookdb\Customer.java
```

These two classes are now available for use in our EJBs and test clients.

4.2.2 Retrieving Data

In this section we will create an EJB that will provide business methods for retrieving books from the database. Our EJB will return books given one of the following:

- nothing – return all books
- a subject – return all books about this subject
- an author – return all books written by this author
- a title – return all books with this title
- a booknr – return the book with this booknr

In order to provide this functionality we will need to define 5 business methods (one for each type of request).

We can start by creating the package (directory) where our EJB will be. We will call this package bookmgr and the EJB BookMgr.

3. Create a directory bookmgr (in IS7\bookdb)!
4. Create a new file BookMgrBean.java (in IS7\bookdb\bookmgr) and open it for editing!
5. Define the package: `package IS7.bookdb.bookmgr;`
6. Define all necessary imports:
`import IS7.bookdb.Book;`

We will probably have to add more imports here later. We will certainly need to import some classes that are necessary for our business methods.

7. Define the class: `public class BookMgrBean implements javax.ejb.SessionBean {}`
8. Define an empty `ejbCreate()` method: `public void ejbCreate() {}`
9. Define the rest of the necessary methods:
`public void ejbRemove() {}`
`public void ejbActivate() {}`
`public void ejbPassivate() {}`
`public void setSessionContext(javax.ejb.SessionContext ctx) {}`

We have now completed the standard parts of the session bean. We must now define our business methods.

10. We can start by defining their signatures:

```
/**Returns a vector of Books containing all the books***/  
public Vector getAllBooks()  
/**Returns a vector of Books containing all the books about this  
subject***/  
public Vector getBooksBySubject(String subject)  
/**Returns a vector of Books containing all the books written by the given  
author(s)***/  
public Vector getBooksByAuthorName(String author)  
/**Returns a vector of Books containing all the books with the given  
title***/  
public Vector getBooksByTitle(String title)  
/**Returns the Book for the given booknr***/  
public Book getBook(int booknr)
```

11. In order to use the class `Vector` without qualifying it every time we must add an `import` for it:
`import java.util.Vector;`

Before we start coding our business methods, let's take a look at the blueprint of our EJB:

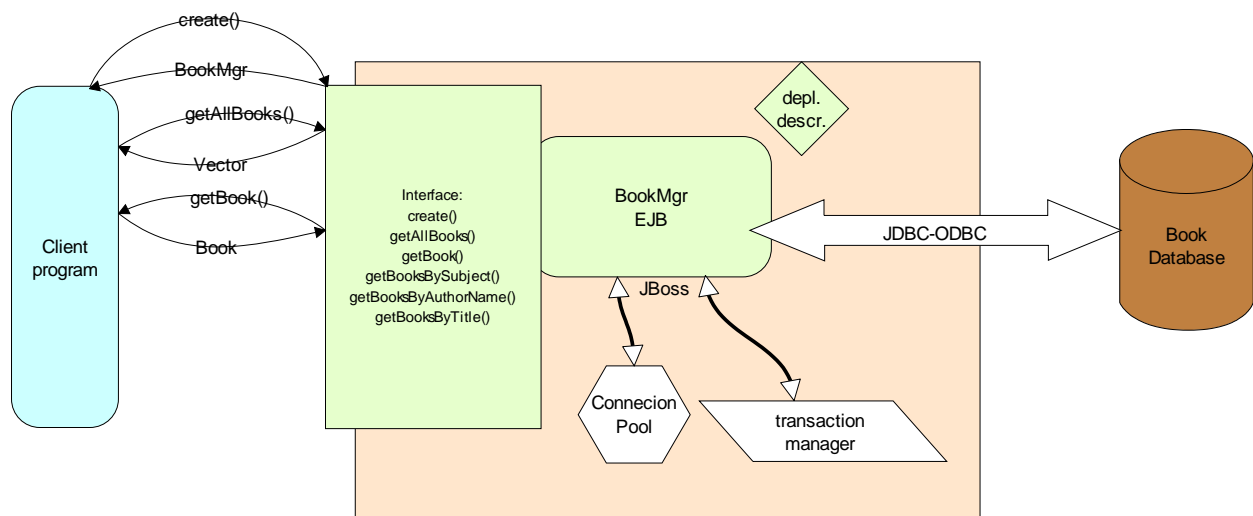


Figure 3 BookMgr system outline

As we can see it is our EJB that contacts the database with any requests, but the connection to the database and the transactions associated to our EJB are handled by the JBoss server. That means that the EJB must request a database connection from the JBoss server and not directly from the database. If we establish a connection directly to the database then any requests send on this connection would not be visible to the transaction manager of the JBoss server.

We also know that all of the business methods need access to the database. In order to avoid writing the same code (for requesting a connection from the connection pool) five times we can create a private method in our session bean that the business methods can use.

We can start by defining this method and then implement the business method that will call it.

12. Define a private method getConnection():

```
private Connection getConnection()
{
    Connection con = null;
    try
    {
        Context jndiCtx = new InitialContext();
        DataSource ds = (javax.sql.DataSource)
            jndiCtx.lookup("java:/jdbc/BookDB");
        con = ds.getConnection();
        return con;
    }
    catch (SQLException ex)
    {
        ex.printStackTrace();
        System.err.println("getConnection failed." + ex.getMessage());
    }
    catch (NamingException e)
    {
        e.printStackTrace();
    }
}
```

```
        System.err.println("lookup failed." + e.getMessage());
    }
    return null;
}
```

What we do is request a `Context` so that we can look up our database alias (we haven't defined this yet) in the local dictionary (managed by the JBoss server) through the Java Naming and Directory Interface (`jndi`). From there we retrieve a `DataSource` which can provide us with the database connection which in turn we return. The method also catches two possible exceptions.

13. In the method defined above we used a number of classes. We must therefore add `import` statements for them:

```
import java.sql.*;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.InitialContext;
import javax.sql.DataSource;
```

We can import the entire `java.sql` package since we are going to use more classes included there in our business methods.

We can now start defining one by one the business methods:

14. Define the implementation of the `getAllBooks()` method:

```
Vector books = new Vector();
Connection con = getConnection();
if (con == null)
    return null;

String query = "SELECT * FROM Book";
try
{
    Statement stmt = con.createStatement();

    ResultSet rs = stmt.executeQuery(query);

    while (rs.next())
    {
        Book newbook = new Book(rs.getInt("booknr"), rs.getString("title"),
rs.getInt("price"), rs.getInt("year"));
        books.add(newbook);
    }
    stmt.close();
    con.close();
}
catch (SQLException ex)
{
    ex.printStackTrace();
    System.err.println("Database error in getAllBooks() " + ex.getMessage());
}
return books;
```

15. Similarly we define the other four business methods:

```
/**Returns a vector of Books containing all the books about this
subject***/
public Vector getBooksBySubject(String subject)
{
    Vector books = new Vector();
    Connection con = getConnection();
    if (con == null)
        return null;

    String query = "SELECT * FROM Book WHERE booknr IN (SELECT booknr FROM
BookSubject bs, Subject s WHERE s.subjectnr = bs.subjectnr AND s.subject =
?)";
    try
    {
        PreparedStatement stmt = con.prepareStatement(query);

        stmt.setString(1, subject);

        ResultSet rs = stmt.executeQuery();

        while (rs.next())
        {
            Book newbook = new Book(rs.getInt("booknr"),
rs.getString("title"), rs.getInt("price"), rs.getInt("year"));
            books.add(newbook);
        }
        stmt.close();
        con.close();
    }
    catch (SQLException ex)
    {
        ex.printStackTrace();
        System.err.println("Database error in getBooksBySubject() " +
ex.getMessage());
    }
    return books;
}
/**Returns a vector of Books containing all the books written by the given
author(s)***/
public Vector getBooksByAuthorName(String author)
{
    Vector books = new Vector();
    Connection con = getConnection();
    if (con == null)
        return null;

    String query = "SELECT * FROM Book WHERE booknr IN (SELECT booknr FROM
CoAuthor ca, Author a WHERE a.authornr = ca.authornr AND a.name = ?)";
    try
    {
        PreparedStatement stmt = con.prepareStatement(query);

        stmt.setString(1, author);

        ResultSet rs = stmt.executeQuery();
```

```
        while (rs.next())
        {
            Book      newbook      =      new      Book(rs.getInt("booknr"),
rs.getString("title"), rs.getInt("price"), rs.getInt("year"));
            books.add(newbook);
        }
        stmt.close();
        con.close();
    }
    catch (SQLException ex)
    {
        ex.printStackTrace();
        System.err.println("Database error in getBooksByAuthorName() " +
ex.getMessage());
    }
    return books;
}
/**Returns a vector of Books containing all the books with the given
title***/
public Vector getBooksByTitle(String title)
{
    Vector books = new Vector();
    Connection con = getConnection();
    if (con == null)
        return null;

    String query = "SELECT * FROM Book WHERE title = ?";
    try
    {
        PreparedStatement stmt = con.prepareStatement(query);

        stmt.setString(1, title);

        ResultSet rs = stmt.executeQuery();

        while (rs.next())
        {
            Book      newbook      =      new      Book(rs.getInt("booknr"),
rs.getString("title"), rs.getInt("price"), rs.getInt("year"));
            books.add(newbook);
        }
        stmt.close();
        con.close();
    }
    catch (SQLException ex)
    {
        ex.printStackTrace();
        System.err.println("Database error in getBooksByTitle() " +
ex.getMessage());
    }
    return books;
}
/**Returns the Book for the given booknr***/
public Book getBook(int booknr)
{
    Connection con = getConnection();
```

```
        if (con == null)
            return null;

        String query = "SELECT * FROM Book WHERE booknr = ?";
        try
        {
            PreparedStatement stmt = con.prepareStatement(query);

            stmt.setInt(1, booknr);

            ResultSet rs = stmt.executeQuery();

            if (rs.next())
            {
                Book thebook = new Book(rs.getInt("booknr"),
rs.getString("title"), rs.getInt("price"), rs.getInt("year"));
                return thebook;
            }
            stmt.close();
            con.close();
        }
        catch (SQLException ex)
        {
            ex.printStackTrace();
            System.err.println("Database error in getBook() " + ex.getMessage());
        }
        return null;
    }
}
```

16. Our session bean is now complete and can be compiled with the following command (it must be run from the M:\jdb042 directory and the Book class must have already been compiled):

```
javac IS7\bookdb\bookmgr\BookMgrBean.java -classpath .;M:\jdb042\jboss-3.2.1\server\default\lib\jboss-j2ee.jar
```

We still have a lot to do:

- We must create the home interface
- We must create the remote interface
- We must create the ejb-jar.xml
- We must configure our database alias in the local jndi dictionary

Only then we can package our BookMgr EJB in a jar file and deploy it. The first three tasks are not different from the HelloWorld EJB. Let's start by fixing the interfaces:

17. Create a file BookMgrHome.java (in IS7\bookdb\bookmgr) and open it for editing!

18. Define the package: `package IS7.bookdb.bookmgr;`

19. Define the interface: `public interface BookMgrHome extends javax.ejb.EJBHome {}`

20. Define the create() method:

```
BookMgr create() throws java.rmi.RemoteException, javax.ejb.CreateException;
```

21. Create a file BookMgr.java (in IS7\bookdb\bookmgr) and open it for editing!

22. Define the package: `package IS7.bookdb.bookmgr;`

23. Define the interface: `public interface BookMgr extends javax.ejb.EJBObject {}`

24. Define the interfaces of the business methods (adding a throws clause):

```
/**Returns a vector of Books containing all the books***/  
public Vector getAllBooks() throws java.rmi.RemoteException;  
/**Returns a vector of Books containing all the books about this  
subject***/  
public Vector getBooksBySubject(String subject) throws  
java.rmi.RemoteException;  
/**Returns a vector of Books containing all the books written by the given  
author(s)***/  
public Vector getBooksByAuthorName(String author) throws  
java.rmi.RemoteException;  
/**Returns a vector of Books containing all the books with the given  
title***/  
public Vector getBooksByTitle(String title) throws java.rmi.RemoteException;  
/**Returns the Book for the given booknr***/  
public Book getBook(int booknr) throws java.rmi.RemoteException;
```

25. The signatures of the business methods refer to classes `Vector` and `Book`. These classes must either be qualified or stated in an import statement. Add the following import statements:

```
import java.util.Vector;  
import IS7.bookdb.Book;
```

26. Both interfaces are now complete and can be compiled with the following command (also compiles the session bean class):

```
javac IS7\bookdb\bookmgr\*.java -classpath .;M:\jdb042\jboss-  
3.2.1\server\default\lib\jboss-j2ee.jar
```

Next we can create the deployment descriptor:

27. Create a directory META-INF under IS7\bookdb\bookmgr!

28. Create a new file ejb-jar.xml (in the new META-INF directory) and open it for editing!

29. Define the contents of the deployment descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE ejb-jar  
  PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"  
  'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>  
<ejb-jar>  
  <description>BookMgr EJB as part of the CBD with EJB compendium for IS7  
ht2003</description>  
  <display-name>Book Manager EJB</display-name>  
  <enterprise-beans>  
    <session>
```



```
<ejb-name>BookMgr</ejb-name>
<home>IS7.bookdb.bookmgr.BookMgrHome</home>
<remote>IS7.bookdb.bookmgr.BookMgr</remote>
<ejb-class>IS7.bookdb.bookmgr.BookMgrBean</ejb-class>
<session-type>Stateless</session-type>
<transaction-type>Container</transaction-type>
</session>
</enterprise-beans>
<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>BookMgr</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
</assembly-descriptor>
</ejb-jar>
```

The last thing to do before we can deploy the BookMgr EJB is to define the database alias in the local jndi dictionary. This is done in an xml file that is then placed in the deploy directory of JBoss that does the rest! This xml file already exists in the `jboss-3.2.1\server\default\deploy` directory and it is called `msaccess-ds.xml`. It contains no database aliases but it has an empty element `<datasources>`. This element can contain zero or more `<local-tx-datasource>` elements. We will add one such element for our book database.

30. Locate and open the file `msaccess-ds.xml`!

31. Add the following element inside the `<datasources>` element:

```
<local-tx-datasource>
  <jndi-name>jdbc/BookDB</jndi-name>
  <!-- format of URL is "jdbc:odbc:DSNAME" -->
  <connection-url>jdbc:odbc:BookIS7</connection-url>
  <driver-class>sun.jdbc.odbc.JdbcOdbcDriver</driver-class>
  <user-name></user-name>
  <password></password>
</local-tx-datasource>
```

32. Saving the file `msaccess-ds.xml` will cause JBoss to re-deploy it, activating any changes we made.

33. We can now also deploy the BookMgr EJB. We start by packaging everything in a jar file `BookMgrJAR.jar` with the following commands:

```
jar cMvf BookMgrJAR.jar IS7\bookdb\Book.class IS7\bookdb\bookmgr\*.class
jar uMvf BookMgrJAR.jar -C IS7\bookdb\bookmgr META-INF\ejb-jar.xml
```

34. We deploy our new jar file with the following command:

```
copy BookMgrJAR.jar M:\jdb042\jboss-3.2.1\server\default\deploy
```

35. We can also create a jar file `BookMgrInterface.jar` with the classes and interfaces needed by the client developers. This jar file must therefore include the two interfaces and the `Book` class. We can create this file with the following command:

```
jar cMvf BookMgrInterface.jar IS7\bookdb\Book.class  
IS7\bookdb\bookmgr\BookMgrHome.class IS7\bookdb\bookmgr\BookMgr.class
```

We can once again change roles and assume the role of the client developer. We can now design a little test client for the `BookMgr` EJB:

36. Create a new file `BookMgrTestClient.java` in the `M:\jdb042\test` directory! (The class `BookMgrTestClient` will be in the package `test`.)

37. Open the file for editing!

35. Define the package: `package test;`

36. Import the necessary classes:

```
import javax.naming.Context;  
import javax.naming.InitialContext;  
import javax.rmi.PortableRemoteObject;  
import IS7.bookdb.bookmgr.*;  
import IS7.bookdb.Book;  
import java.util.Vector;  
import java.io.BufferedReader;  
import java.io.InputStreamReader;
```

37. Define the class: `public class BookMgrTestClient {}`

38. We only need a `main()` method, so we can start by defining it:

```
public static void main(String[] args) { }
```

39. Inside the `main()` method we will need to first establish a context in order to lookup the `BookMgr` EJB (this is exactly the same we did in the `HelloWorldTestClient`):

```
try  
{  
    Context ctx = new InitialContext();  
    ctx.addToEnvironment(Context.INITIAL_CONTEXT_FACTORY,  
        "org.jnp.interfaces.NamingContextFactory");  
    ctx.addToEnvironment(Context.PROVIDER_URL, "127.0.0.1:1099");  
    //instead of 127.0.0.1 you can use localhost  
    ctx.addToEnvironment("java.naming.factory.url.pkgs",  
        "org.jboss.naming:org.jnp.interfaces");  
  
    Object obj = ctx.lookup("BookMgr");  
    BookMgrHome home = (BookMgrHome)  
        PortableRemoteObject.narrow(obj, BookMgrHome.class);  
    BookMgr bookMgr = home.create();  
}
```

40. The only thing missing now is the call (or calls) to the business methods. We can for example create a little loop that interacts with the user and the calls the appropriate business method of

the EJB. We can also create a couple of private function for printing the list of books on the screen. Let's start by completing the main() method:

```
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        String input;
        boolean stay = true;
        while (stay)
        {
            System.out.println("Choose one of the following options:");
            System.out.println("-----");
            System.out.println("1. Show all books!");
            System.out.println("2. Show books about a specific subject!");
            System.out.println("3. Show books by a specific author!");
            System.out.println("4. Show books with a specific title!");
            System.out.println("5. Show a specific book (by specifying
booknr)!");
            System.out.println("6. Exit!");
            System.out.println("-----");
            System.out.print("Enter your choice: ");
            input = br.readLine();
            switch ((new Integer(input)).intValue())
            {
                case 1:
                    printBookList(bookMgr.getAllBooks());
                    break;
                case 2:
                    System.out.print("Enter a subject: ");
                    String subject = br.readLine();
                    printBookList(bookMgr.getBooksBySubject(subject));
                    break;
                case 3:
                    System.out.print("Enter an author: ");
                    String author = br.readLine();
                    printBookList(bookMgr.getBooksByAuthorName(author));
                    break;
                case 4:
                    System.out.print("Enter a title: ");
                    String title = br.readLine();
                    printBookList(bookMgr.getBooksByTitle(title));
                    break;
                case 5:
                    System.out.print("Enter a booknr: ");
                    String temp = br.readLine();
                    int booknr = (new Integer(temp)).intValue();
                    printHeader();
                    printBook(bookMgr.getBook(booknr));
                    break;
                case 6:
                    stay=false;
                    break;
            } //end of switch
        } //end of while
    } //end of try block
    catch (Exception ex)
    {
```

```
        System.err.println("Caught an unexpected exception!");  
        ex.printStackTrace();  
    } //end of catch
```

41. We can now create the private methods `printBookList()`, `printBook()` and `printHeader()` (The layout is not very good, but there is no reason why we should make it any better just for a test client):

```
private static void printHeader()  
{  
    System.out.println("Booknr    Title                                Price    Year");  
    System.out.println("-----");  
}  
  
private static void printBook(Book book)  
{  
    if (book != null)  
    {  
        System.out.print(book.getBooknr()+"\t");  
        System.out.print(book.getTitle()+"\t");  
        System.out.print(book.getPrice()+"\t");  
        System.out.println(book.getYear());  
    }  
}  
  
private static void printBookList(Vector books)  
{  
    if (books != null)  
    {  
        printHeader();  
        for (int i=0;i<books.size();i++)  
        {  
            printBook((Book) books.elementAt(i));  
        }  
    }  
}
```

42. We are now ready to compile and run our test client. We can compile our test client with the following command:

```
javac test\BookMgrTestClient.java -classpath jboss-3.2.1\client\jbossall-  
client.jar;BookMgrInterface.jar
```

38. We can run our test client with the following command:

```
java -cp .;jboss-3.2.1\client\jbossall-client.jar test.BookMgrTestClient
```

Here is an example of the test client in action:

```
Command Prompt - java -cp .\jboss-3.2.1\client\jbossall-client.jar test.BookMgrTestClient
M:\jdb042>java -cp .\jboss-3.2.1\client\jbossall-client.jar test.BookMgrTestClient
Choose one of the following options:
1. Show all books!
2. Show books about a specific subject!
3. Show books by a specific author!
4. Show books with a specific title!
5. Show a specific book <by specifying booknr>!
6. Exit!
-----
Enter your choice: 3
Enter an author: George Reese
Booknr Title Price Year
-----
9 Java Database Best Practices 450 1998
10 Managing and Using MySQL 195 2002
11 MySQL Pocket Reference 110 1999
12 Database Programming with JDBC and Java 260 2001
13 MySQL and mSQL 300 2000
Choose one of the following options:
1. Show all books!
2. Show books about a specific subject!
3. Show books by a specific author!
4. Show books with a specific title!
5. Show a specific book <by specifying booknr>!
6. Exit!
-----
Enter your choice: 4
Enter a title: mysql and msql
Booknr Title Price Year
-----
13 MySQL and mSQL 300 2000
Choose one of the following options:
1. Show all books!
2. Show books about a specific subject!
3. Show books by a specific author!
4. Show books with a specific title!
5. Show a specific book <by specifying booknr>!
6. Exit!
-----
Enter your choice: 5
Enter a booknr: 13
Booknr Title Price Year
-----
13 MySQL and mSQL 300 2000
Choose one of the following options:
1. Show all books!
2. Show books about a specific subject!
3. Show books by a specific author!
4. Show books with a specific title!
5. Show a specific book <by specifying booknr>!
6. Exit!
-----
Enter your choice: 5
```

4.2.3 Manipulating Data

So far we have worked with only retrieving data from the database. In this section we will make an EJB that updates the database. The EJB will provide the clients with one business method for inserting a new customer into the table `Customer`. The business method will take one argument (a `Customer` object) and insert the values in the database. If something is incorrect with/in the received `Customer` object it will throw an exception `CustomerMgrException`.

This EJB will be called `CustomerMgr` and it will be in the `IS7.bookdb.customermgr` package. We will need the data class `IS7.bookdb.Customer` (that we created earlier) and the exception class `CustomerMgrException` which we will create and place in the package `IS7.bookdb.customermgr`.

Most of the EJB structure is the same as the previous one. The only thing that differs is the business method and its implementation. We also have to create a new exception class. We can start with just that:

1. Create the file `CustomerMgrException.java` in the package `IS7.bookdb.customermgr` and add the following content:

```
package IS7.bookdb.customermgr;

/** Exception to be thrown by CustomerMgr */
public class CustomerMgrException extends Exception {

    public CustomerMgrException () { }

    public CustomerMgrException (String msg) {
        super(msg);
    }
}
```

The exception class is ready, so continue with the session bean:

2. Create the session bean file (`CustomerMgrBean.java`) and add the standard session bean content:

```
package IS7.bookdb.customermgr;

public class CustomerMgrBean implements javax.ejb.SessionBean
{
    public void ejbCreate() {}
    public void ejbRemove() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void setSessionContext(javax.ejb.SessionContext ctx) {}
}
```

3. We can also add a private method for retrieving a database connection (-the same private method we had in `BookMgrBean.java`-):

```
private Connection getConnection()
{
    Connection con = null;
    try
    {
        Context jndiCtx = new InitialContext();
        DataSource ds = (javax.sql.DataSource)
            jndiCtx.lookup("java:/jdbc/BookDB");
        con = ds.getConnection();
        return con;
    }
    catch (SQLException ex)
    {
        ex.printStackTrace();
        System.err.println("getConnection failed." + ex.getMessage());
    }
}
```

```
        catch (NamingException e)
        {
            e.printStackTrace();
            System.err.println("lookup failed." + e.getMessage());
        }
        return null;
    }
}
```

4. The only thing missing is the business method that can be defined as follows:

```
public void insertCustomer(Customer customer) throws CustomerMgrException
{
    if (customer == null)
        throw new CustomerMgrException("Insert failed: A customer must be
specified!");

    if (customer.getName() == null)
        throw new CustomerMgrException("Insert failed: A customer must have a
name!");

    Connection con = getConnection();
    if (con == null)
        throw new CustomerMgrException("Insert failed: No connection to
database available!");

    String name = customer.getName();
    String address = "";
    if (customer.getAddress() != null)
        address = customer.getAddress();
    String city = "";
    if (customer.getCity() != null)
        city = customer.getCity();
    String country = "";
    if (customer.getCountry() != null)
        country = customer.getCountry();

    String query = "INSERT INTO Customer (name, address, city, country)
VALUES (?, ?, ?, ?)";
    try
    {
        PreparedStatement stmt = con.prepareStatement(query);

        stmt.setString(1, name);
        stmt.setString(2, address);
        stmt.setString(3, city);
        stmt.setString(4, country);
        stmt.executeUpdate();
        stmt.close();
        con.close();
    }
    catch (SQLException ex)
    {
        ex.printStackTrace();
        System.err.println("Database error in getBook() " + ex.getMessage());
        throw new CustomerMgrException("Insert failed: Database said: " +
ex.getMessage());
    }
}
```

```
    }  
}
```

5. These methods use some classes that must be defined in import statements:

```
import IS7.bookdb.Customer;  
import java.sql.*;  
import javax.naming.Context;  
import javax.naming.NamingException;  
import javax.naming.InitialContext;  
import javax.sql.DataSource;
```

6. We can quickly define the two interfaces:

Remote interface CustomerMgr:

```
package IS7.bookdb.customermgr;
```

```
import IS7.bookdb.Customer;
```

```
public interface CustomerMgr extends javax.ejb.EJBObject  
{  
    /** Inserts a new customer in the database */  
    public void insertCustomer(Customer customer) throws  
        java.rmi.RemoteException, CustomerMgrException;  
}
```

Home interface CustomerMgrHome:

```
package IS7.bookdb.customermgr;
```

```
public interface CustomerMgrHome extends javax.ejb.EJBHome  
{  
    CustomerMgr create() throws java.rmi.RemoteException,  
        javax.ejb.CreateException;  
}
```

7. We can also create a deployment descriptor ejb-jar.xml (in a META-INF directory) With the following content:

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE ejb-jar  
    PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"  
    'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>  
<ejb-jar>  
    <description>CustomerMgr EJB as part of the CBD with EJB compendium for  
IS7 ht2003</description>  
    <display-name>Customer Manager EJB</display-name>  
    <enterprise-beans>  
        <session>  
            <ejb-name>CustomerMgr</ejb-name>  
            <home>IS7.bookdb.customermgr.CustomerMgrHome</home>  
            <remote>IS7.bookdb.customermgr.CustomerMgr</remote>  
            <ejb-class>IS7.bookdb.customermgr.CustomerMgrBean</ejb-class>  
            <session-type>Stateless</session-type>  
            <transaction-type>Container</transaction-type>  
        </session>  
    </enterprise-beans>  
</ejb-jar>
```



```
<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>CustomerMgr</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
</assembly-descriptor>
</ejb-jar>
```

8. We can now compile, jar and deploy the CustomerMgr EJB with the following commands:

```
javac      IS7\bookdb\customermgr\*.java      -classpath      .;M:\jdb042\jboss-
3.2.1\server\default\lib\jboss-j2ee.jar
jar        cMvf      CustomerMgrJAR.jar        IS7\bookdb\Customer.class
IS7\bookdb\customermgr\*.class
jar uMvf CustomerMgrJAR.jar -C IS7\bookdb\customermgr META-INF\ejb-jar.xml
copy CustomerMgrJAR.jar M:\jdb042\jboss-3.2.1\server\default\deploy
```

9. We can also create ajar file for the client developers with the following command:

```
jar        cMvf      CustomerMgrInterface.jar      IS7\bookdb\Customer.class
IS7\bookdb\customermgr\CustomerMgrHome.class
IS7\bookdb\customermgr\CustomerMgr.class
```

10. The EJB is now deployed. We need a test client to test our business method! Here is a simple test client (test.CustomerMgrTestClient):

```
package test;
```

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import IS7.bookdb.customermgr.*;
import IS7.bookdb.Customer;
import java.io.BufferedReader;
import java.io.InputStreamReader;
```

```
public class CustomerMgrTestClient
{
    public static void main(String[] args)
    {
        try
        {
            Context ctx = new InitialContext();
            ctx.addToEnvironment(Context.INITIAL_CONTEXT_FACTORY,
"org.jnp.interfaces.NamingContextFactory");
            ctx.addToEnvironment(Context.PROVIDER_URL, "127.0.0.1:1099");
            ctx.addToEnvironment("java.naming.factory.url.pkgs",
"org.jboss.naming:org.jnp.interfaces");

            Object obj = ctx.lookup("CustomerMgr");
            CustomerMgrHome home = (CustomerMgrHome)
PortableRemoteObject.narrow(obj, CustomerMgrHome.class);
            CustomerMgr customerMgr = home.create();
```

```
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));

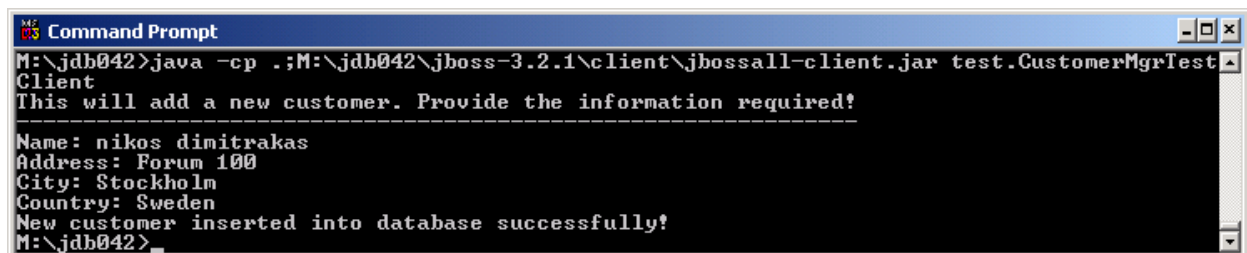
        System.out.println("This will add a new customer. Provide the
information required!");
        System.out.println("-----");
        System.out.print("Name: ");
        String name = br.readLine();
        System.out.print("Address: ");
        String address = br.readLine();
        System.out.print("City: ");
        String city = br.readLine();
        System.out.print("Country: ");
        String country = br.readLine();
        customerMgr.insertCustomer(new Customer(0, name, address, city,
country));
    } //end of try block
    catch (CustomerMgrException ex)
    {
        System.out.println(ex.getMessage());
    } //end of catch
    catch (Exception ex)
    {
        System.err.println("Caught an unexpected exception : " + ex);
        ex.printStackTrace();
    } //end of catch
    System.out.println("New customer inserted into database
successfully!");
} //end of main

}
```

11. We compile and run our test client with the following commands:

```
javac test\CustomerMgrTestClient.java -classpath .;M:\jdb042\jboss-
3.2.1\client\jbossall-client.jar;CustomerMgrInterface.jar
java -cp .;M:\jdb042\jboss-3.2.1\client\jbossall-client.jar
test.CustomerMgrTestClient
```

Here is an example of our test client in action:



```
Command Prompt
M:\jdb042>java -cp .;M:\jdb042\jboss-3.2.1\client\jbossall-client.jar test.CustomerMgrTest
Client
This will add a new customer. Provide the information required!
-----
Name: nikos dimitrakas
Address: Forum 100
City: Stockholm
Country: Sweden
New customer inserted into database successfully!
M:\jdb042>
```

4.3 Assignments

One of the assignments in this section is compulsory and must be presented for either nikos dimitrakas or Martin Henkel in order to acquire a pass grade for the practical part of the course.

You may, of course, complete more than one of the assignments, but only one is required. Choose one of the following four assignments!

Assignments:

1. Create an EJB and a test client for adding and retrieving subjects (two business methods)!
2. Create an EJB and a test client for retrieving authors! Two business methods should be available:
 - Get all authors
 - Get all authors for a particular book (booknr)
3. Create an EJB and a test client for adding authors to the database!
4. Modify the CustomerMgr EJB so that it also provides business methods for the following:
 - Retrieve all the customers that have ordered a particular book (given a book title).
 - Retrieve all the customers that have ordered books for more than a particular total price.Modify also the test client (or write a new test client) to test the new business methods!

It is of course possible to combine many EJBs in one client and provide a more complete functionality for our database.

4.4 When things have gone bad!

Luckily all the files for the exercises in this compendium and the JBoss server are available to download. Should you, by mistake, delete or change files, you can quickly restore the JBoss server and exercise files. Should the JBoss server crash (-has not happened during the development of this compendium-), it is enough to close command prompt window where it is running. This will force Windows to kill the JBoss process. If it still won't go away, one can always reboot windows!

5 Internet Resources

The most important site is <http://java.sun.com/> where there are tutorials and documentation for all java frameworks.

Java: <http://java.sun.com/>

J2EE: <http://java.sun.com/j2ee/>

J2EE Tutorial: <http://java.sun.com/j2ee/tutorial/>

J2EE v1.4 API Documentation: <http://java.sun.com/j2ee/1.4/docs/api/index.html>

Java Basics Tutorial: <http://java.sun.com/docs/books/tutorial/>

JDBC Tutorial: <http://java.sun.com/docs/books/tutorial/jdbc/index.html>

6 Epilogue

When all this is done, you should have a fair understanding of the EJB framework. This compendium only covered a part of the EJB framework (session beans), but hopefully this is enough for illustrating the advantages of component based development and the Enterprise JavaBeans framework.

Department of Computer
and Systems Sciences
SU/KTH

CBD with EJB Lab
IS7/2I1404 autumn 2003

Stockholm
July 2003
nikos dimitrakas

I hope you have enjoyed this compendium.

The Author

nikos dimitrakas