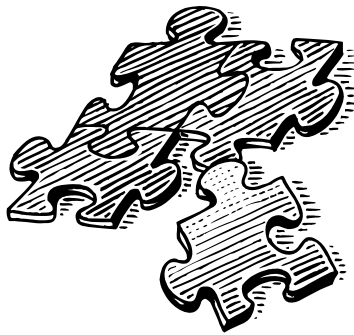# MODEL DRIVEN DEVELOPMENT OF COMPONENTS (IS7/IV2009)

Tutorial 2

Component Based Development with Enterprise JavaBeans

nikos dimitrakas

&

Martin Henkel

Department of Computer and Systems Sciences (DSV),
Stockholm University/Royal Institute of Technology (KTH)

Department of Computer
and Systems Sciences
SU/KTH

EJB Tutorial
IS7/IV2009

Stockholm
July 2009

Table of contents

1 Introduction

This compendium contains the following:
- An introduction to the JBoss J2EE server 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 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, Component Based Development (CBD) and the EJB architecture/component model. An overview of EJB is given in the lectures.

Note that this Tutorial describes the implementation of *EJB3.0* components, which differs *significantly* from prior versions of EJB!

1.1 Homepage

Information about this compendium can be found here:

- `http://dsv.su.se/~martinh/IS7`

The latest version of the compendium and all the files needed to complete the exercises in the compendium can be found at the above address.

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
- MS Access used for the example database
- ODBC driver/manager used for accessing the database from the EJBs
- Java tools (compiler, jar-tool)
- 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.

Department of Computer
and Systems Sciences
SU/KTH

EJB Tutorial
IS7/IV2009

Stockholm
July 2009

2 JBoss and EJB

In this chapter we will 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 Installing 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 engine and the EJB container). This version of JBoss is zipped in a file named JBossIS7.zip and can be downloaded from the following locations:
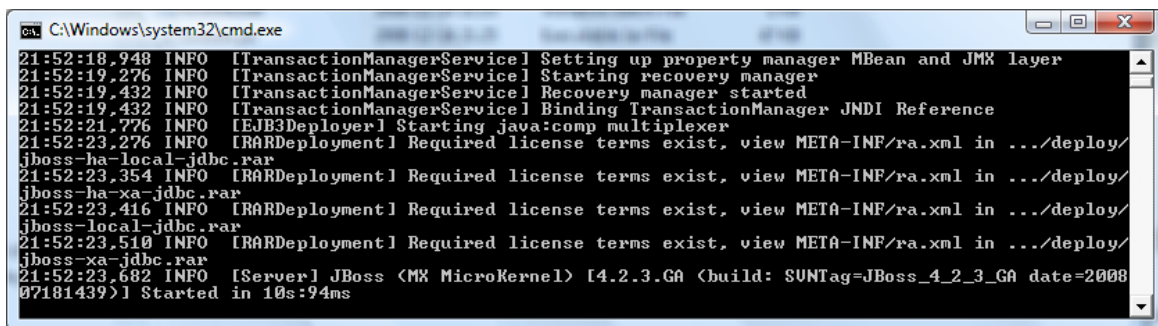
- `http://dsv.su.se/~martinh/IS7/JBossIS7.zip`
- `http://coursematerial.nikosdimitrakas.com/ejb/JBossIS7.zip`

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

2.2 Starting JBoss

To start the JBoss server, just execute the file `run.bat` that is located in the folder `jboss\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`. 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 can take up to 1 minute. A command prompt window (JBoss 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 container). When the server has finished loading, a message will appear stating that JBoss has started:

This window is now locked by JBoss. To stop the server press Ctrl-C while the window is active, or use the shutdown.bat file located in the bin folder. 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.

Department of Computer   EJB Tutorial   Stockholm
and Systems Sciences   IS7/IV2009   July 2009
SU/KTH

JBoss will use this window for any messages during run-time. For example EJB deployment done while the server is running will produce a message in this window. *Also, server error messages appear in this window*, so make sure to keep an eye on the message window.

At this point it is also important to know that the JBoss server listens on port 1099. We need to use this port when we build java programs that access the JBoss server.

2.3 Deploying EJBs

To deploy an EJB in JBoss we need a jar (java archive) file containing
1. The EJB bean class
2. The remote interface for the EJB
3. All necessary helper classes, such as data classes and exception classes

The jar file only needs to be placed in the `jboss\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 The Book Example 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 Microsoft Access database. The database is stored in an MS Access database file named book.mdb. The file can be downloaded from:

- `http://dsv.su.se/~martinh/IS7/book.mdb`
- `http://coursematerial.nikosdimitrakas.com/ejb/book.mdb`

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

The figure below shows the tables included in the database and their relationships:
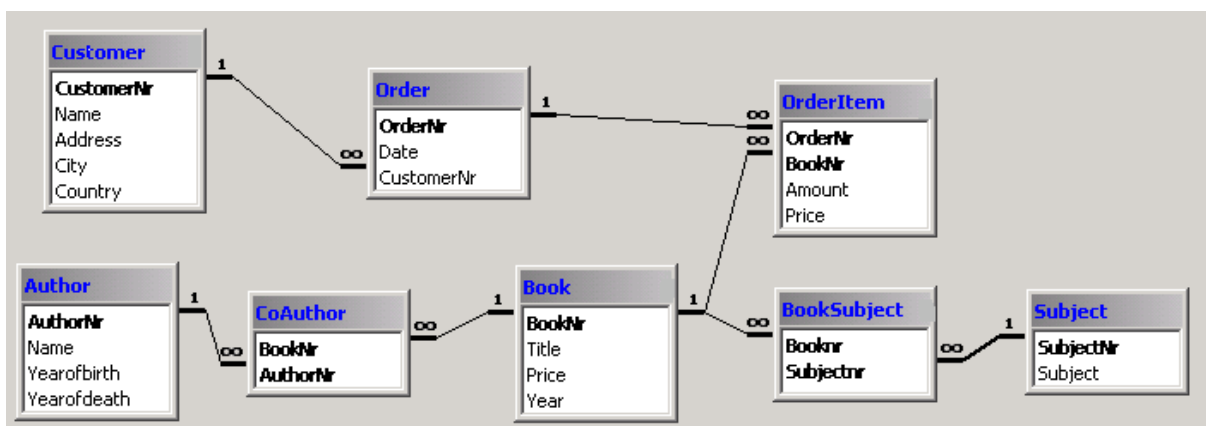
Figure 1 Book database

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

Department of Computer   EJB Tutorial   Stockholm
and Systems Sciences   IS7/IV2009   July 2009
SU/KTH

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.

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 (completed) are available here:

- `http://dsv.su.se/~martinh/IS7`

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

The exercises that follows 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.

A note on Java configuration

This tutorial assumes that the commands "javac" and "jar" are in the command path of Windows. If you get the error *"'javac' is not recognized as an internal or external command, operable program or batch file."* You have to set the command path manually by issuing the following command at the command prompt:

```
set path=%path%;C:\Java\jdk1.6.0\bin
```

(Exchange the path given above to a path that point to your java installation, if needed)

Department of Computer
and Systems Sciences
SU/KTH

EJB Tutorial
IS7/IV2009

Stockholm
July 2009

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":
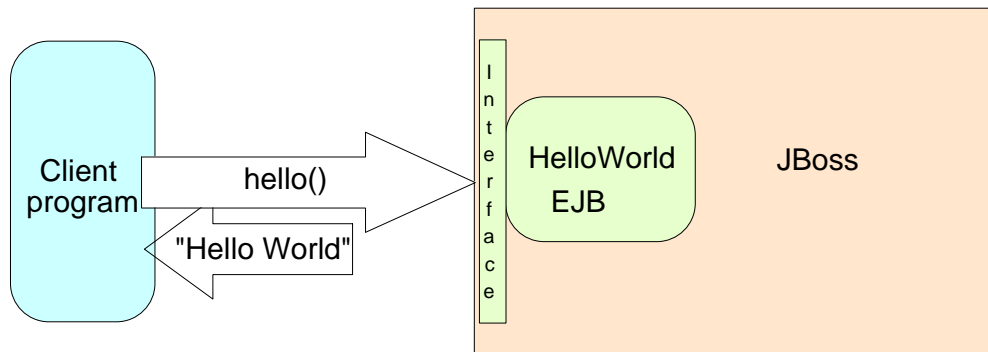

Figure 2 Hello World system outline

In short, here is what we have to do:
- Create a java class `HelloWorldBean` where the method `hello()` will be implemented.
- Create a remote interface `HelloWorld` with the signatures of the business methods.
- 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 `hello` where all the files for the EJB will be. So we have to create two directories `IS7` and `hello`. We can put them under `M:/jdb042`. In the directory `hello` we will place the interface and the bean class of the EJB. We should have the following directory structure (the `profiles` directory should already be available at your home directory!):
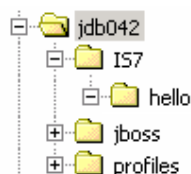
Note: Java package names are case sensitive, while the windows file system is not. Always write "IS7" in upper case/capitals to avoid problems during compilation.

4.1.1 Remote interface

We need to define the interface of our HelloWorld bean, so let's start with creating the Java interface `HelloWorld`:

1. Create a file HelloWorld.java (in IS7\hello) and open it for editing (for example using notepad, SciTE, Eclipse, or some other editor that you are familiar with)!

2. Define the package: `package IS7.hello;`

3. Define the interface: `public interface HelloWorld {}`
    This interface must contain the signatures of all business methods defined in the session bean. In our case there is only the method `hello()`:
    `public String hello();`
    The complete content of the `HelloWorld.java` should be the following:

```
package IS7.hello;

public interface HelloWorld
{
    public String hello();
}
```

4.1.2 Bean class

Now its time to write the Bean implementation in the Java class `HelloWorldBean`:

4. Create a file HelloWorldBean.java (in IS7\hello) and open it for editing!

5. Define the package:
    `package IS7.hello;`

6. Define the imports needed for the EJB annotations by adding the following:
    `import javax.ejb.Stateless;`
    `import javax.ejb.Remote;`

7. The Bean class must be annotated with the EJB-annotation "@Stateless" to indicate that it is a stateless session bean. Furthermore, we need to assign the Bean a name so we can access it later on. Add the following annotation before defining the bean-class:
    `@Stateless(name = "HelloWorld")`

8. To indicate that the HelloWorld interface is the remote interface of this bean add the following annotation:
    `@Remote(HelloWorld.class)`

9. Now its time to define the bean class:
    `public class HelloWorldBean implements HelloWorld {}`

10. We also need to implement our business method `hello()`:
    ```
    public String hello()
    {
        return "Hello World!";
    }
    ```

The complete content of the `HelloWorldBean.java` should be the following:

```
package IS7.hello;

import javax.ejb.Stateless;
import javax.ejb.Remote;

@Stateless(name = "HelloWorld")
@Remote(HelloWorld.class)
public class HelloWorldBean implements HelloWorld
{
    public String hello()
    {
        return "Hello World!";
    }
}
```

11. The session bean `HelloWorldBean` is now complete and can be compiled! In order to compile the Bean class and its interface the compiler must know where the necessary EJB framework classes can be found. They are available in the files
    `M:\jdb042\jboss\server\default\lib\jboss-ejb3x.jar`
    and
    `M:\jdb042\jboss\server\default\lib\jboss-j2ee.jar.`

    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 (written in one line):
    `javac IS7\hello\*.java -classpath jboss\server\default\lib\jboss-ejb3x.jar;jboss\server\default\lib\jboss-j2ee.jar`

4.1.3 Packaging and deployment

12. Start the JBoss server if it is not already running! (This will make sure that deployment error messages appear after all start up messages of the server)

13. With the compiled files (.class files), we have all necessary files for packaging and deploying the HelloWorld EJB. The only thing we need to do is put the compiled interface and the compiled bean class in a jar file `HelloWorld.jar` (packaging) and copy it to the deployment directory of JBoss (deployment) which is found by following the path `jboss\server\default\deploy`).

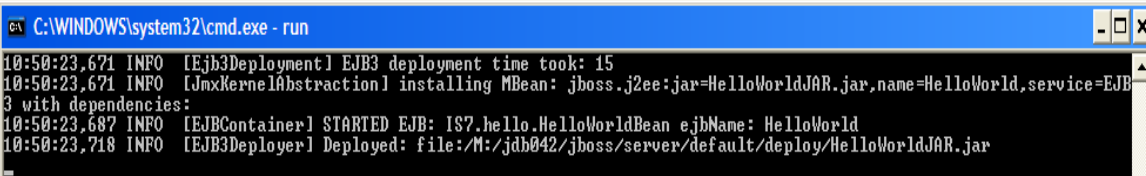   Do the packaging with the following command!:
    `jar cMvf HelloWorld.jar IS7\hello\*.class`

    The command adds the class files to a new jar file (the first `c` in the command stands for "create").

14. The EJB is now packaged and, deploy it with the following command:
    `copy HelloWorld.jar jboss\server\default\deploy`

    This command simply copies the jar file to JBoss which automatically deploys it. If JBoss was not running, the EJB would be deployed when JBoss started.

15. Check for any deployment error or warnings at the JBoss server window. A normal deployment should give the following output in the server window:

16. 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 have access to the remote interface 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.hello` package. Therefore we can create a jar file with the interface 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\hello\HelloWorld.class
```

17. In this simple example we just add one .class file to the jar file, however in later examples we will add more files need by the client into the jar.

4.1.4 Test client

Taking now the role of the client developer we only have access to the HelloWorldInterface.jar. We also know that the helloworld EJB 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
```

We can start developing our test client by defining a new class `HelloWorldTestClient` inside the package `test`.

18. Create a directory `test` at your home directory (in our case `M:\jdb042\test`)!

19. Create a new file `HelloWorldTestClient.java` (in the directory test) and open it for editing!

20. Define the package: `package test;`

21. Import the necessary classes:

```
import javax.naming.Context;
import javax.naming.InitialContext;
import IS7.hello.HelloWorld;
```

22. Define the class: `public class HelloWorldTestClient {}`

23. We only need a `main()` method, so we can start by defining it:
```
public static void main(String[] args) { }
```

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 our 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, "localhost:1099");
   ctx.addToEnvironment("java.naming.factory.url.pkgs",
                        "org.jboss.naming:org.jnp.interfaces");
```

24. Still inside the `try` block we have to ask the context to look up the HelloWorld EJB. The context will return an object which we can cast into an reference to the `HelloWorld` interface. Note that when looking up the bean we need to use the Beans name, as set by the @stateless annotation in the Bean class, we add "/remote" to this name to indicate that we want access to the remote interface:
```
HelloWorld helloWorld = (HelloWorld) ctx.lookup("HelloWorld/remote");
```

25. 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());
```

26. 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();
}
```

The complete content of the `HelloWorldTestClient.java` should be the following:

```java
package test;

import javax.naming.Context;
import javax.naming.InitialContext;

import IS7.hello.HelloWorld;

public class HelloWorldTestClient
{
    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,
                "localhost:1099");
            ctx.addToEnvironment("java.naming.factory.url.pkgs",
                "org.jboss.naming:org.jnp.interfaces");

            // Lookup the bean using its name + "/remote"
            HelloWorld helloWorld = (HelloWorld)
                        ctx.lookup("HelloWorld/remote");

            System.out.println("The component says: " +
                                helloWorld.hello());
        }
        catch (Exception ex)
        {
            System.err.println("Caught an unexpected exception!");
            ex.printStackTrace();
        }
    }
}
```

27. 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\client\jbossall-client.jar
```

We can compile our test client with the following command:
```
javac    test\HelloWorldTestClient.java    -classpath    jboss\client\jbossall-
client.jar;HelloWorldInterface.jar
```
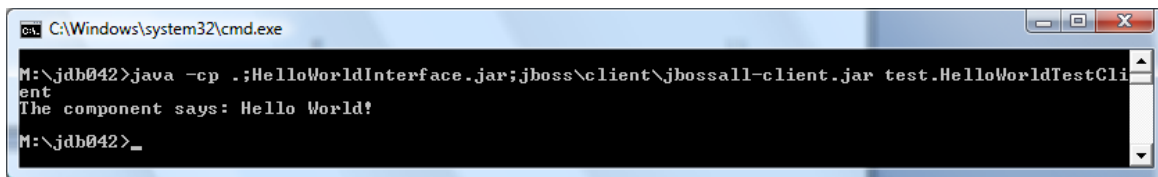
28. To run the test client we need to have:
- the jar file with the remote interface of the EJB. (We used the same jar file when we compiled our test client)
- JBoss run-time support libraries, located in jboss\client\jbossall-client.jar

We can run our test client with the following command:
```
java -cp .;HelloWorldInterface.jar;jboss\client\jbossall-client.jar
test.HelloWorldTestClient
```

Department of Computer   EJB Tutorial   Stockholm
and Systems Sciences   IS7/IV2009   July 2009
SU/KTH

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

If you get an error at this stage, don't forget to also check the server window for errors.

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 more 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 primitive objects). In section 4.2.3 we will define those data classes that we will later use when developing the EJBs and test clients. We will also need to configure our JBoss server to access the database. We will do this in section 4.2.1 and 4.2.2.

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 Creating an ODBC Data Source

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:\Windows\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:
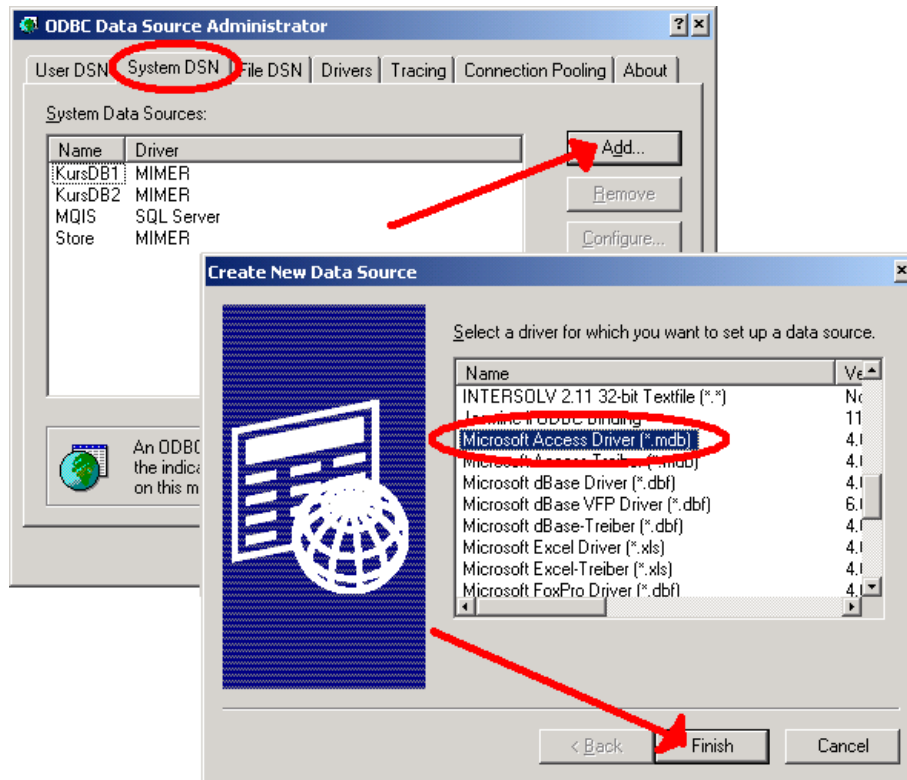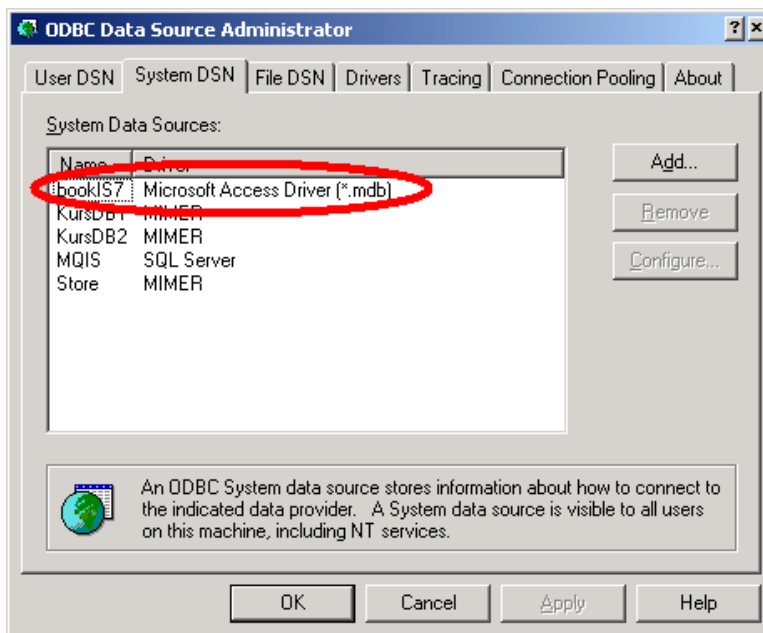

If this is not possible due to user rights, then you can create a User DSN. A user DSN will work fine as long as the same user that created the DSN is logged in when Jboss is running.

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

14
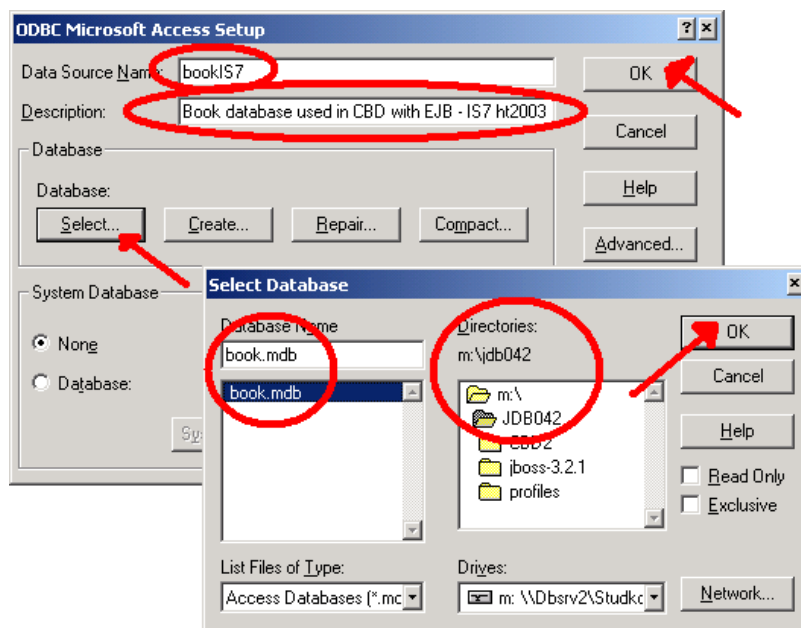
The new DSN should now be available under System DSN:

The database is now available through an ODBC driver and it is mapped to the alias bookIS7. In the next section we will configure our JBoss connection pool for this ODBC data source.

4.2.2 Configuring the JBoss connection pool

In this section we will define the database connection pool properties. This is done in a xml file that is then placed in the deploy directory of JBoss. This xml is named `msaccess-ds.xml` and it must have a root element `<datasources>`. This element can contain zero or more `<local-tx-datasource>` elements. We will need one such element for our book database according to the following:

```
<local-tx-datasource>
    <jndi-name>jdbc/BookDB</jndi-name>
    <!-- format of URL is "jdbc:odbc:DSNNAME" -->
    <connection-url>jdbc:odbc:BookIS7</connection-url>
    <driver-class>sun.jdbc.odbc.JdbcOdbcDriver</driver-class>
    <user-name></user-name>
    <password></password>
    <min-pool-size>0</min-pool-size>
    <max-pool-size>5</max-pool-size>
</local-tx-datasource>
```

1. Download the file `msaccess-ds.xml` and place it in the directory `jboss\server\default\deploy`. The file can be downloaded from here:

- `http://dsv.su.se/~martinh/IS7/msaccess-ds.xml`

It is important to check that the ODBC DSN alias is the same as the DSNNAME (the part after `jdbc:odbc:`) in the `<connection-url>` element. The jndi-name is the name we will use in the java code in order to connect to the database.

4.2.3 Data Classes – Book

The database contains eight tables. 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 (or download it) in the directory 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 int booknr;
   private String title;
   private int price;
   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:
   ```
   public int getBooknr() {return booknr;}
   public String getTitle() {return title;}
   public int getPrice() {return price;}
   public int getYear() {return year;}
   public void setBooknr(int value) {booknr = value;}
   public void setTitle(String value) {title = value;}
   public void setPrice(int value) {price = value;}
   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. Compile it with the following command:
   ```
   javac IS7\bookdb\Book.java
   ```

We can now create the other data class:
8. Create (or download it!) a new file Customer.java (in IS7\bookdb) and open it for editing!

9. Define its content according to the following:

```
package IS7.bookdb;

public class Customer implements java.io.Serializable
{
   private int customernr;
   private String name;
   private String address;
   private String city;
   private String country;
   //Here too, we could create a vector for Order objects, but we won't.

   public int getCustomernr() {return customernr;}
   public String getName() {return name;}
   public String getAddress() {return address;}
   public String getCity() {return city;}
   public String getCountry() {return country;}
   public void setCustomernr(int value) {customernr = value;}
   public void setName(String value) {name = value;}
   public void setAddress(String value) {address = value;}
   public void setCity(String value) {city = value;}
   public void setCountry(String value) {country = value;}

   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;
   }
}
```

10. 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.

4.2.4 Retrieving Data – BookMgr EJB

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
- a booknr – return the book with this booknr

In order to provide this functionality we will need to define 3 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.

Department of Computer   EJB Tutorial   Stockholm
and Systems Sciences   IS7/IV2009   July 2009
SU/KTH

4.2.4.1 Remote interface

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

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

3. Define the interface: `public interface BookMgr {}`

4. Define the interfaces of the business methods. In this example we will throw a basic exception class java.lang.Exception whenever an error occurs. For returning a list of Books we use the Vector class. We use *java generics* to define that the Vector contains instances of the Book class by writing <Book> after Vector:

```
 // Returns a vector of Books containing all the books
 public Vector<Book> getAllBooks() throws Exception;

 // Returns a vector of Books containing all the books about this subject,
 // empty Vector if nothing found
 public Vector<Book>  getBooksBySubject(String subject) throws Exception;

 // Returns the Book for the given booknr, or null
 public Book getBook(int booknr) throws Exception;
```

5. 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;
```

6. The interface is now complete and can be compiled with the following command:
```
javac IS7\bookdb\bookmgr\BookMgr.java
```

4.2.4.2 Bean class

7. Create a new file BookMgrBean.java (in IS7\bookdb\bookmgr) and open it for editing!

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

9. Define the necessary imports:
```
import IS7.bookdb.Book;
import javax.ejb.Stateless;
import javax.ejb.Remote;
```

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.

10. Define the class: `public class BookMgrBean implements BookMgr {}`

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

Department of Computer   EJB Tutorial   Stockholm
and Systems Sciences   IS7/IV2009   July 2009
SU/KTH

11. We can start by defining their signatures:

    ```
    public Vector<Book> getAllBooks()

    public Vector<Book> getBooksBySubject(String subject)

    public Book getBook(int booknr)
    ```

12. 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:
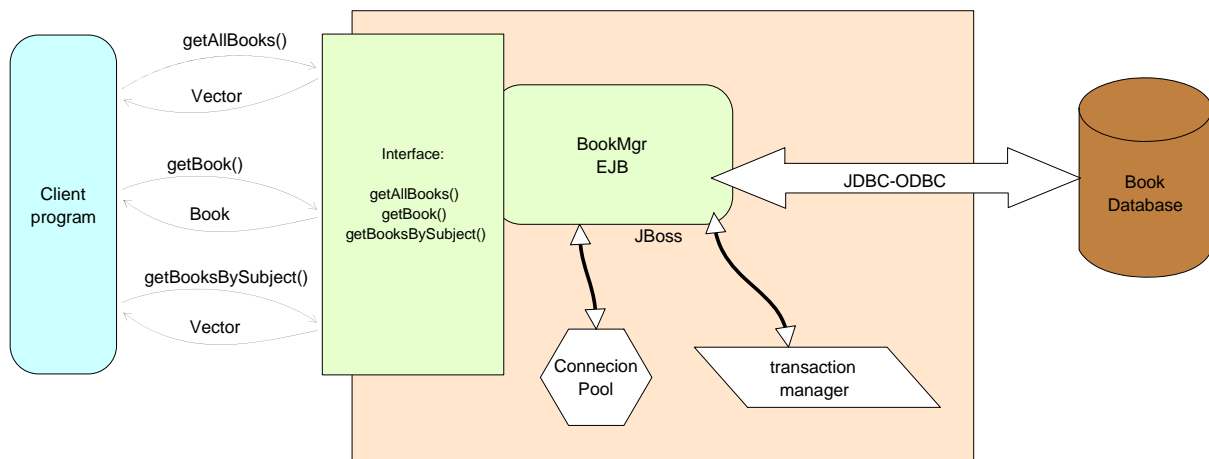
Figure 3 BookMgr system outline

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 sent 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 attribute that hold a reference to the data source. By using a specific @Resource annotation we can then let JBOSS "inject" a reference to the data source into our attribute.

13. Start with adding the necessary import for the @Resource annotation and the data source:
    ```
    import javax.annotation.Resource;
    import javax.sql.DataSource;
    ```

14. Define a private attribute mDatasource, with an annotation linking the attribute to the MS Access BookDB (remember that we gave the database the jndi name "BookDB" in the msaccess-ds.xml file):

```
@Resource(mappedName = "java:/jdbc/BookDB")
private DataSource mDataSource;
```

We can now start defining business methods one by one:

15. Start with adding an import for the SQL classes needed:

```
import java.sql.*;
```

16. Define the implementation of the getAllBooks() method:

```java
public Vector<Book> getAllBooks() throws Exception
{
   try
   {
      Vector<Book> books = new Vector<Book>();

      Connection con = mDataSource.getConnection();
      String query = "SELECT * FROM Book";
      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();
      return books;
   }
   catch (SQLException ex)
   {
      // Throw an error that the client can catch
      throw new Exception("getAllBooks() - DB Error: " + ex.getMessage());
   }
}
```

17. Similarly we define the other two business methods:

```java
public Vector<Book> getBooksBySubject(String subject) throws Exception
{
   try
   {
      Vector<Book> books = new Vector<Book>();

      Connection con = mDataSource.getConnection();
      String query = "SELECT * FROM Book WHERE booknr IN (SELECT booknr FROM
BookSubject bs, Subject s WHERE s.subjectnr = bs.subjectnr AND s.subject =
?)";
      PreparedStatement stmt = con.prepareStatement(query);
      stmt.setString(1, subject);
      ResultSet rs = stmt.executeQuery();
```

Department of Computer   EJB Tutorial   Stockholm
and Systems Sciences   IS7/IV2009   July 2009
SU/KTH

```java
      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();
      return books;
    }
    catch (SQLException ex)
    {
      // Throw an error that the client can catch
      throw new Exception("getBooksBySubject() - DB Error: " +
ex.getMessage());
    }
  }

  public Book getBook(int booknr)  throws Exception
  {
    try
    {
      Connection con = mDataSource.getConnection();
      String query = "SELECT * FROM Book WHERE booknr = ?";
      PreparedStatement stmt = con.prepareStatement(query);
      stmt.setInt(1, booknr);
      ResultSet rs = stmt.executeQuery();

      Book thebook = null;

      if (rs.next())
      {
        thebook = new Book(     rs.getInt("booknr"), rs.getString("title"),
                   rs.getInt("price"), rs.getInt("year"));
      }
      stmt.close();
      con.close();
      return thebook;
    }
    catch (SQLException ex)
    {
      // Throw an error that the client can catch
      throw new Exception("getBook() - DB Error: " + ex.getMessage());
    }
  }
```

18. Our bean class is now complete and can be compiled with javac. Javac must be run from the
    M:\jdb042 directory and the Book class must already have been compiled, note also that we
    include the current directory (".") in the classpath:
    ```
    javac IS7\bookdb\bookmgr\BookMgrBean.java -classpath
    .;jboss\server\default\lib\jboss-ejb3x.jar;jboss\server\default\lib\jboss-
    j2ee.jar
    ```

4.2.4.3 Packaging and deployment

19. We can now also deploy the BookMgr EJB. We start by packaging everything in a jar file
    `BookMgr.jar` with the following commands:
    ```
    jar cMvf BookMgr.jar IS7\bookdb\Book.class IS7\bookdb\bookmgr\*.class
    ```

20. We deploy our new jar file with the following command (remember to check the server
    window for error messages):
    ```
    copy BookMgr.jar jboss\server\default\deploy
    ```

21. 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 remote interface and the
    `Book` class. We can create this file with the following command:
    ```
    jar cMvf BookMgrInterface.jar IS7\bookdb\Book.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:

4.2.4.4 Test client

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

23. Open the file for editing!

24. Define the package: `package test;`

25. Import the necessary classes:
    ```
    import javax.naming.Context;
    import javax.naming.InitialContext;
    import IS7.bookdb.bookmgr.BookMgr;
    import IS7.bookdb.Book;
    import java.util.Vector;
    import java.io.BufferedReader;
    import java.io.InputStreamReader;
    ```

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

27. We only need a `main()` method, so we can start be defining it:
    ```
    public static void main(String[] args) { }
    ```

28. 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, "localhost:1099");
       ctx.addToEnvironment("java.naming.factory.url.pkgs",
                            "org.jboss.naming:org.jnp.interfaces");

       BookMgr bookMgr = (BookMgr)ctx.lookup("BookMgr/remote");
    ```

22

Department of Computer   EJB Tutorial   Stockholm
and Systems Sciences   IS7/IV2009   July 2009
SU/KTH

29. 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:

```java
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 a specific book (by specifying booknr)!");
  System.out.println("4. 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 a booknr: ");
    String temp = br.readLine();
    int booknr = (new Integer(temp)).intValue();
    printHeader();
    printBook(bookMgr.getBook(booknr));
    break;
  case 4:
    stay = false;
    break;
  }// end of switch
}// end of while
} // end of try block
catch (Exception ex)
{
  System.err.println("Caught an unexpected exception :" + ex);
  ex.printStackTrace();
}// end of catch
```

30. 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):

```java
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<Book> books)
{
  printHeader();
  for (Book b : books)
  {
    printBook(b);
  }

}
```

31. 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\client\jbossall-
client.jar;BookMgrInterface.jar
```
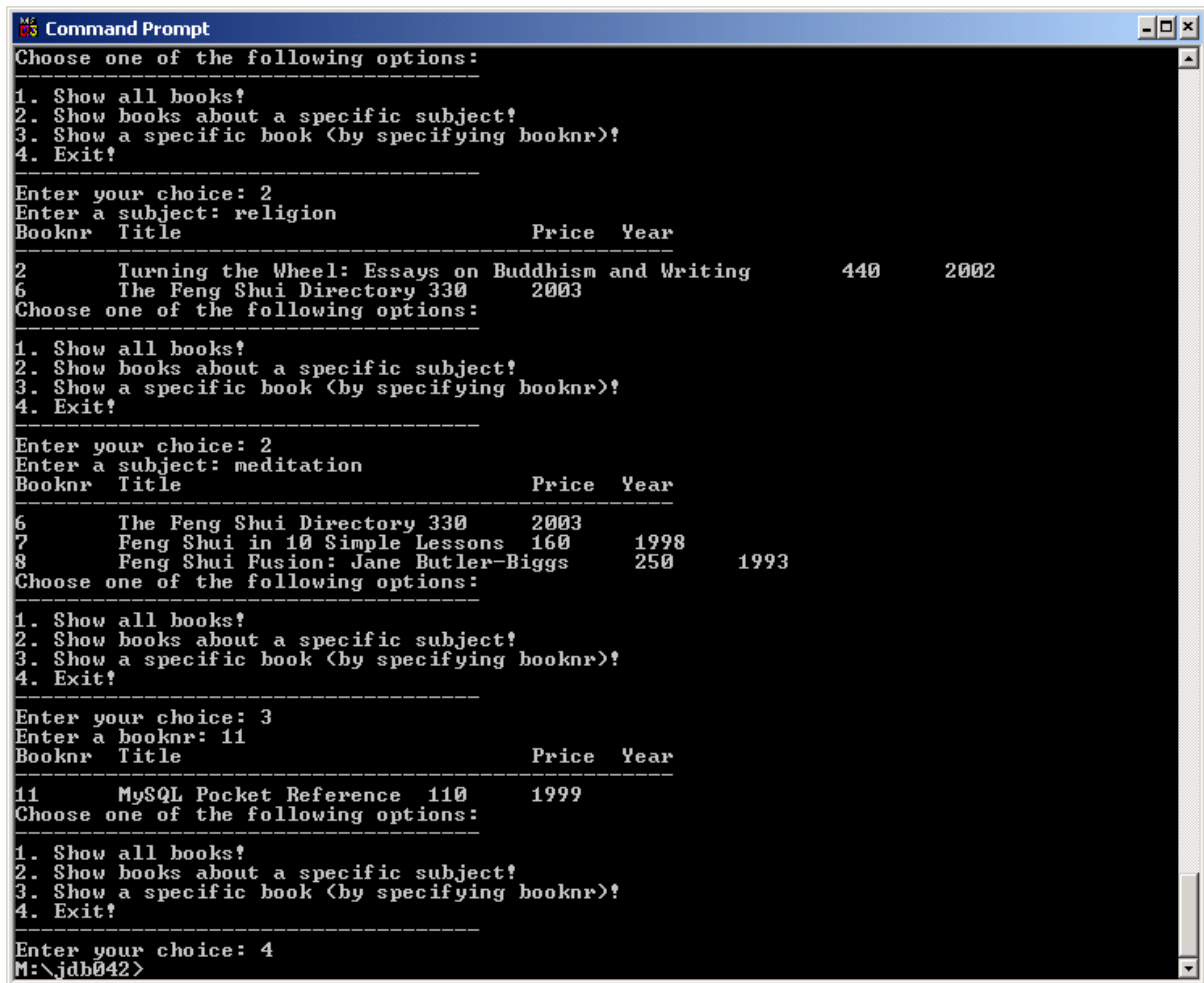
32. We can run our test client with the following command:
```
java -cp .;BookMgrInterface.jar;jboss\client\jbossall-client.jar
test.BookMgrTestClient
```

Here is an example of the test client in action:

4.2.5 Manipulating Data – CustomerMgr EJB

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. Since this Bean will update the database, we will configure the Bean to use transactions using the @TransactionAttribute annotation. Another difference from previous beans is that we also will create a new exception class. We can start with just that:

4.2.5.1 Exception class

Creating and handling our own exception enables the application to handle errors in a structured way. It becomes simpler to distinguish application-generated errors from data base errors and other exceptions. Another advantage with defining your own exceptions is the ability to create custom error messages. We will use this advantage to create an exception class with two types of message texts; one text describing the error in a short, user-friendly manner, and one messages containing the technical details of the error.

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

```java
package IS7.bookdb.customermgr;

/** Exception to be thrown by CustomerMgr **/
public class CustomerMgrException extends javax.ejb.EJBException
{
   // contains a user-friendly description of the error
   private String mUserMsg;

   public CustomerMgrException(String userMsg, String message)
   {
      super(message);
      mUserMsg = userMsg;

   }
   public String getUserMessage()
   {
      return mUserMsg;
   }
}
```

We inherit `EJBException` (instead of just `Exception`) so that the current transaction will be automatically rolled back if something goes wrong. Note that the base class EJBException will contain the technical error text in this case.

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

4.2.5.2 Remote interface

We can quickly define the remote interface `CustomerMgr`:

```java
package IS7.bookdb.customermgr;

import IS7.bookdb.Customer;

public interface CustomerMgr
{
   // Inserts a new customer in the database
   // Throws CustomerMgrException if an DB error ocurs, or if the parameter
   // is invalid
   public void insertCustomer(Customer customer)
      throws CustomerMgrException;

}
```

26

Department of Computer
and Systems Sciences
SU/KTH

EJB Tutorial
IS7/IV2009

Stockholm
July 2009

4.2.5.3 Bean class

2. Create the session bean file (CustomerMgrBean.java) and add the standard session bean content (note the two extra imports and the annotation to configure the handling of transactions):

```java
package IS7.bookdb.customermgr;

import IS7.bookdb.Customer;

import java.sql.*;
import javax.sql.DataSource;

import javax.ejb.Stateless;
import javax.ejb.Remote;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import javax.annotation.Resource;

@Remote(CustomerMgr.class)
@Stateless(name = "CustomerMgr")
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class CustomerMgrBean implements CustomerMgr
{
    // Declare database connection
    @Resource(mappedName = "java:/jdbc/BookDB")
    private DataSource mDataSource;
```

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

```java
public void insertCustomer(Customer customer) throws CustomerMgrException
{
    if (customer == null)
        throw new CustomerMgrException(
            "Insert failed: Customer data is missing",
            "insertCustomer() - the parameter customer is null!");

    if (customer.getName() == null)
        throw new CustomerMgrException(
            "Insert failed: Customer name is missing",
            "insertCustomer() - the parameter customer.name is null!");

    try
    {

        Connection con = mDataSource.getConnection();

        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 (?, ?, ?, ?)";
        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)
    {
      throw new CustomerMgrException(
           "Insert failed, database failure",
           "CustomerMgrBean.insertCustomer() Insert of customer '"
           + customer.getName() + "' failed: Database said: "
           + ex.getMessage());
    }
  }
```

4.2.5.4 Packaging and deployment

4. We can now compile, package and deploy the CustomerMgr EJB with the following commands:

```
javac IS7\bookdb\customermgr\*.java –classpath
.;jboss\server\default\lib\jboss-ejb3x.jar;jboss\server\default\lib\jboss-
j2ee.jar

jar cMvf CustomerMgr.jar IS7\bookdb\Customer.class
IS7\bookdb\customermgr\*.class

copy CustomerMgr.jar jboss\server\default\deploy
```

5. Create a jar file for the client developers with the following command:

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

4.2.5.5 Test client

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

```java
package test;

import javax.naming.Context;
import javax.naming.InitialContext;

import IS7.bookdb.Customer;
import IS7.bookdb.customermgr.CustomerMgr;
import IS7.bookdb.customermgr.CustomerMgrException;

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, "localhost:1099");
        ctx.addToEnvironment("java.naming.factory.url.pkgs",
            "org.jboss.naming:org.jnp.interfaces");

        CustomerMgr customerMgr = (CustomerMgr) ctx
            .lookup("CustomerMgr/remote");

        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));
        System.out.println("New customer inserted into database
successfully!");
      } // end of try block
      catch (CustomerMgrException cex)
      {
        System.err.println("Server exception: " + cex.getUserMessage() +
"\n" + cex.getMessage());
      }
```
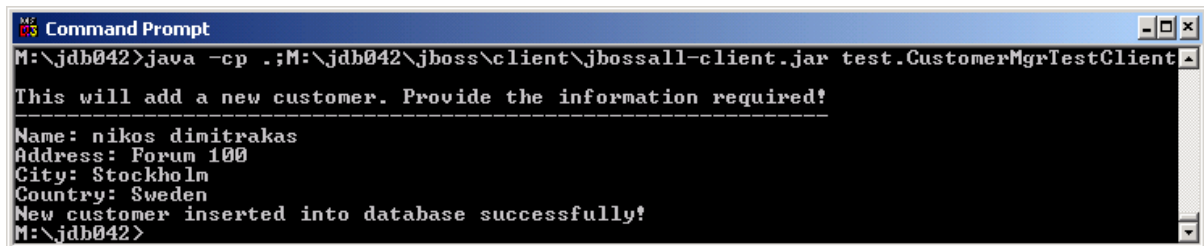
```
   catch (Exception ex)
   {
      System.err.println("Caught an unexpected exception :" + ex);
   }

 }

}
```

7. We compile and run our test client with the following commands:
    ```
    javac test\CustomerMgrTestClient.java -classpath .;jboss\client\jbossall-
    client.jar;CustomerMgrInterface.jar

    java    -cp    .;jboss\client\jbossall-client.jar;CustomerMgrInterface.jar
    test.CustomerMgrTestClient
    ```

Here is an example of our test client in action:

```
Command Prompt                                                                  _ □ ×
M:\jdb042>java -cp .;M:\jdb042\jboss\client\jbossall-client.jar test.CustomerMgrTestClient
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 Bean-to-Bean Communication

(Note: this part is not needed in order to perform the tutorial assignment. However it will be valuable when later doing the EJB project assignment.)

When constructing larger systems consisting of both system and business components there is a need for one component to call another component according to the scenario below:

JBoss EJB Server

Client → System EJB → Business EJB → DB
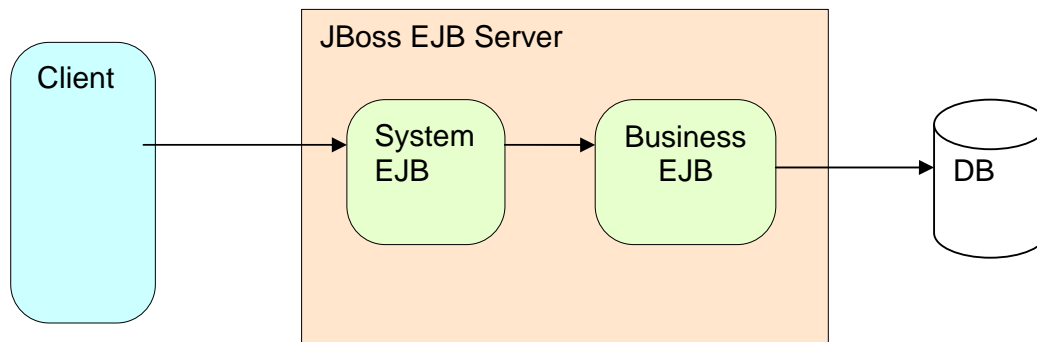
Two layers of components; System and Business EJB components

One option when implementing the communication between system and business components is to let the system component do a JNDI lookup on the business component, just like the client does. However in EJB3 there is a much simpler way of doing this using the @EJB annotation:

1. In the system component declare an attribute that is a reference to the business component remote interface. In this example we declare a reference to the CustomerMgr bean, and annotate it with the CustomerMgr beans name using the @EJB annotation:

Department of Computer   EJB Tutorial   Stockholm
and Systems Sciences   IS7/IV2009   July 2009
SU/KTH

```
        @EJB(beanName = "CustomerMgr")
        private CustomerMgr mCustomerMgr;
```

2. Import the EJB annotation:
```
        import javax.ejb.EJB;
```

When the EJB is deployed in JBoss the server will check that a bean with the name given in the @EJB annotation are deployed. If there is no such bean running (for example, if the bean name is misspelled in the @EJB annotation, or if the referred bean failed to deploy) the following exception will show up in the server window:

```
java.lang.RuntimeException: Failed to populate ENC:
env/IS7.bookdb.BookMgr/mCustomerMgr global jndi name was null.
```

To avoid this problem, make sure that the business components are deployed before the system components.

The JBoss EJB server will create an instance of the CustomerMgr bean the first time the mCustomerMgr attribute is accessed.

5 Assignments

One of the assignments in this section is compulsory and must be presented for the examinator(s) found in the course notes in order to acquire a pass grade for the practical part of the course. Your group should be able to demonstrate a working solution, also be prepared to answer some general questions on how you have built your EJBs and EJB in general. You may, of course, complete more than one of the assignments, but only one is required. Choose one of the following four assignments!

Assignments:
a) Create an EJB and a test client for adding and retrieving subjects (two business methods)!
b) Create an EJB and a test client for retrieving authors! One business method should be available:
   - Get all authors for a particular book (booknr)
c) Create an EJB and a test client for adding authors to the database!
d) 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 your database.

6 Troubleshooting - When things have gone bad!

6.1 Installing JBOSS

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 the 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!

6.2 Common errors

Debugging EJB applications requires quite a lot of skill and experience of the component server. Without prior experience with EJB servers, some errors that JBOSS gives might seem a bit "odd" and vague.

Here are a short checklist that you can use for finding and correcting common errors:

Common error	How to correct it
You cannot connect to your EJB.	1) Check that JBOSS started without errors, watch the server window for errors when JBOSS starts. Restart JBOSS if you need to clear old errors from the window. 2) Check that JBOSS deploys your EJB without errors (watch the console window). 3) Double-check that your EJB-name in the annotated bean class file and in the client are the same, do not forget to add "/remote" to the name.
Your changes to an EJB have no effect, even thought you re-deploy it.	1) Check the time of the jar file in the deployment directory. Make sure that the latest jar file is actually copied to the deployment folder. 2) If needed, add some printouts (System.err.println) to your EJB implementation. Make sure these printouts appear in the server window when you run your test application. If not – you are running against an old version of the EJB component!
You get a nullpointer exception when trying to access the database	1) Check that you have the same JDBC data source name in the odbc administrator (run odbcad32) and in the msaccess-ds.xml file 2) Check that you got the same JNDI name in the msaccess-ds.xml and in your Bean-class.
JBOSS reports that some (of your) class files cannot be found, even though they are in a jar file.	1) Check the contents of your jar files to make sure that the packaging is correct. A jar file is a simple zip-file, so you can open it in winzip. If you do not have winzip installed, rename the file so that it ends with ".zip" and open it. 2) Check the order of deployment. At start up JBOSS deploys the jar files in alphabetical order. Make sure that the "shared" class files are deployed first (deploy business components before system components). If this is not the case, change the name of the jar file, or write a bat-file that deploys the EJB in the correct order. 3) Using hot-deploy (deploying while JBOSS is running) can sometimes create versioning conflicts between EJBs that are dependent on each

other. *To make sure that JBOSS start from "scratch" (without old versions of your components), stop JBOSS, delete your jar files from jboss/server/default/deploy, delete the contents of the jboss/server/default/tmp folder, restart JBOSS.*

7 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/
JEE: http://java.sun.com/javaee/
JEE API Documentation: http://java.sun.com/javaee/6/docs/api/
JDBC Tutorial: http://java.sun.com/docs/books/tutorial/jdbc/index.html

Department of Computer
and Systems Sciences
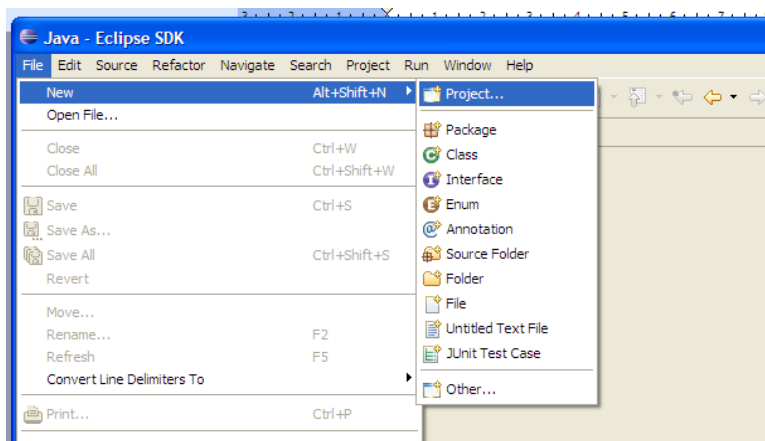SU/KTH

EJB Tutorial
IS7/IV2009

Stockholm
July 2009

8 Compiling EJBs with the Eclipse Environment

Working with Java applications can be greatly simplified by using a development environment such as Netbeans or Eclipse. This chapter contains some quick hints on how to set up Eclipse 3.2 to compile, package and deploy your EJBs.
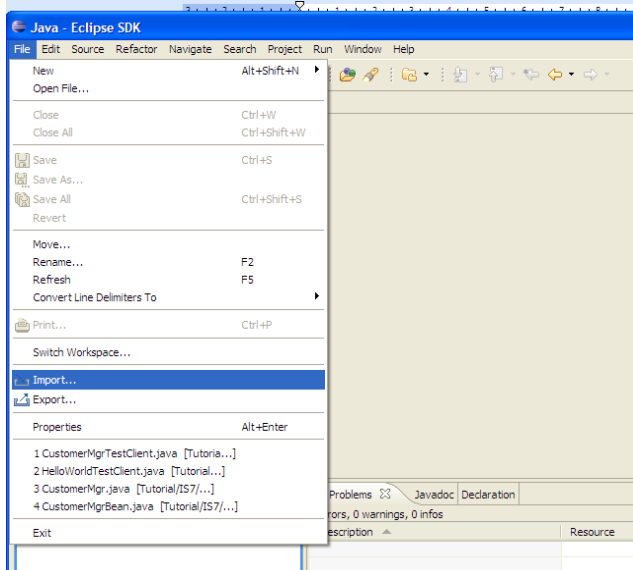
Note: This chapter is provided for those who would like to use a development environment, there is no tutoring/support available for the use of Eclipse or Netbeans.
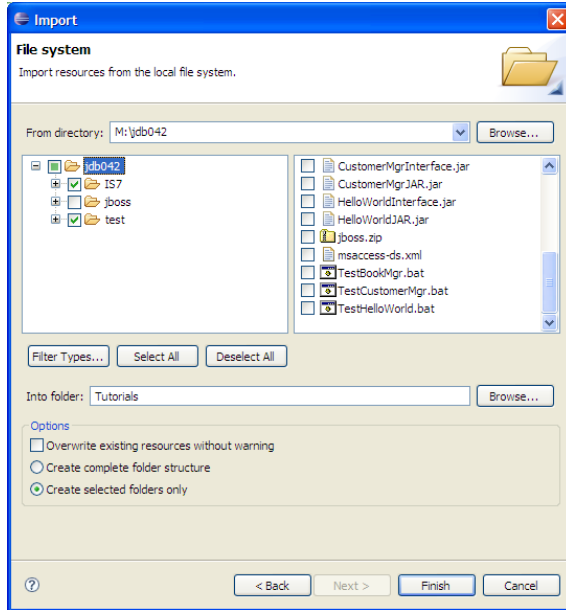
## 8.1 Creating an Eclipse Project

1. Create a new Java Project in Eclipse:

2. Import existing java-classes from the file system (if desired):

Department of Computer
and Systems Sciences
SU/KTH

EJB Tutorial
IS7/IV2009

Stockholm
July 2009

(Note that we only select folders with the java classes, "IS7" and "test" in this case)

3. Include the JBOSS EJB3 libraries in the build path ("build path" is the classpath of Eclipse) by right-clicking the newly created project and select "Configure Build Path…":
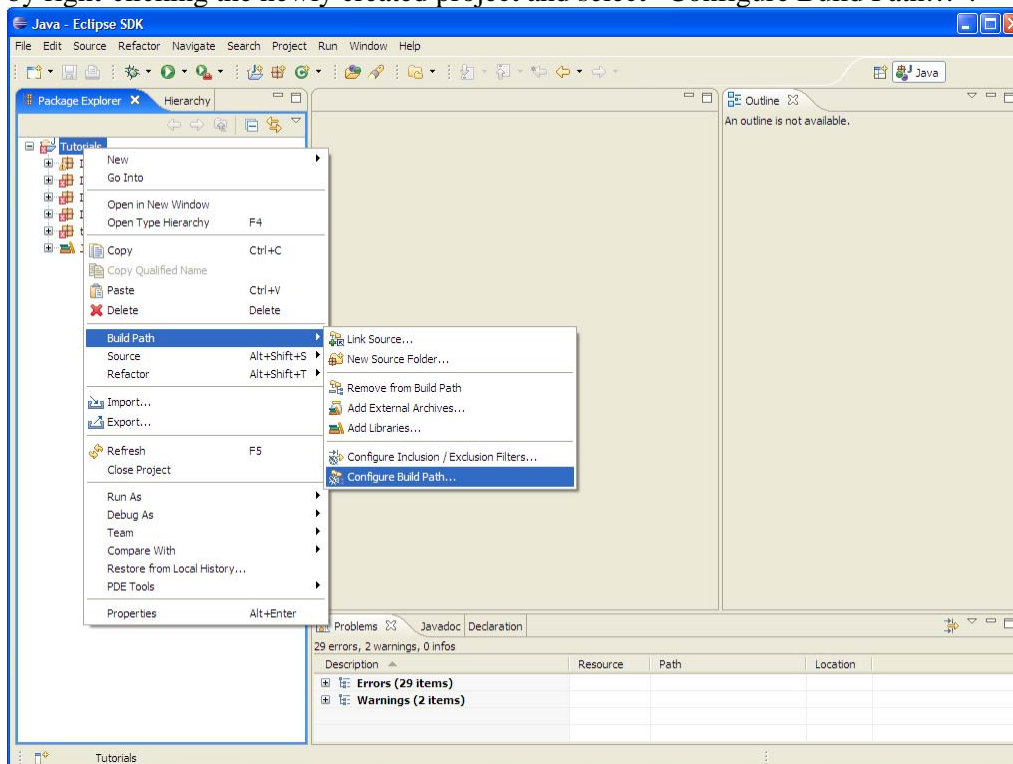
Select "Add external JAR…" and add jboss-ejb3x.jar and jboss-j2ee.jar. Note that both of these libraries can be found in \jboss\server\default\lib\ in newer versions of JBOSS (the picture below is from using an older version).
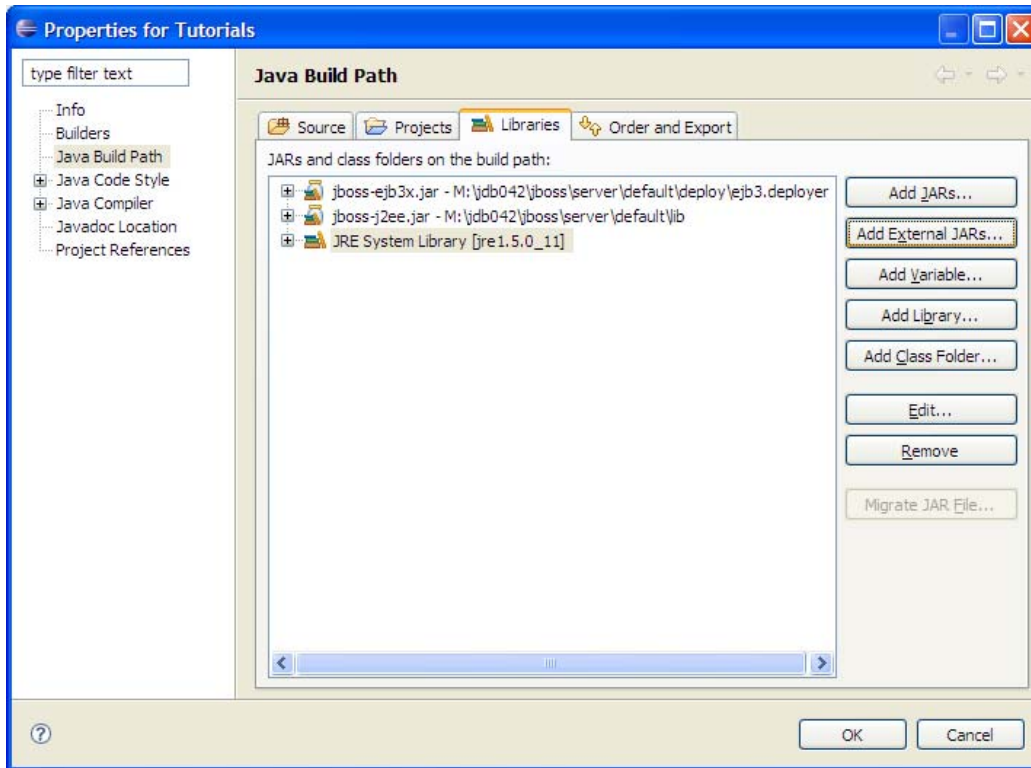
You can now edit your java files in Eclipse, the classes are compiled as soon as you save your changes.
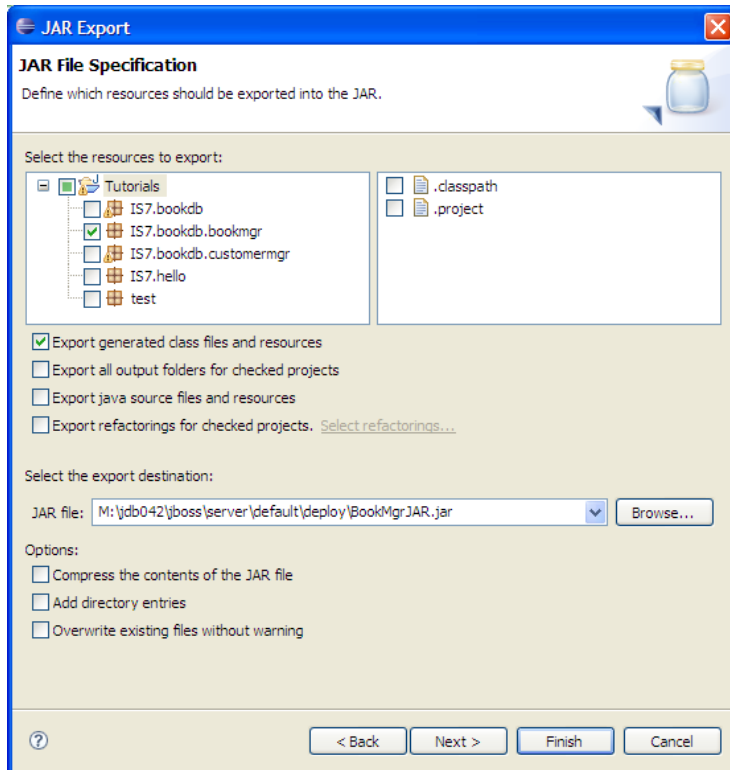
8.2 Running test clients

If you would like to run testclients from within Eclipse you need to add more jars:
•   jbossall-client.jar

8.3 Packaging and deployment of EJB Jars

1. To create a jar file from within Eclipse, select a package in the package explorer and select the menu option "Export…" in the file menu. Select the option to export as a Jar file:

Note that the export destination is set to the deployment directory of JBOSS, the component will be deploy directly when it is created by Eclipse.

2. Select to save the jar description in the Eclipse workspace so you can use it later on:
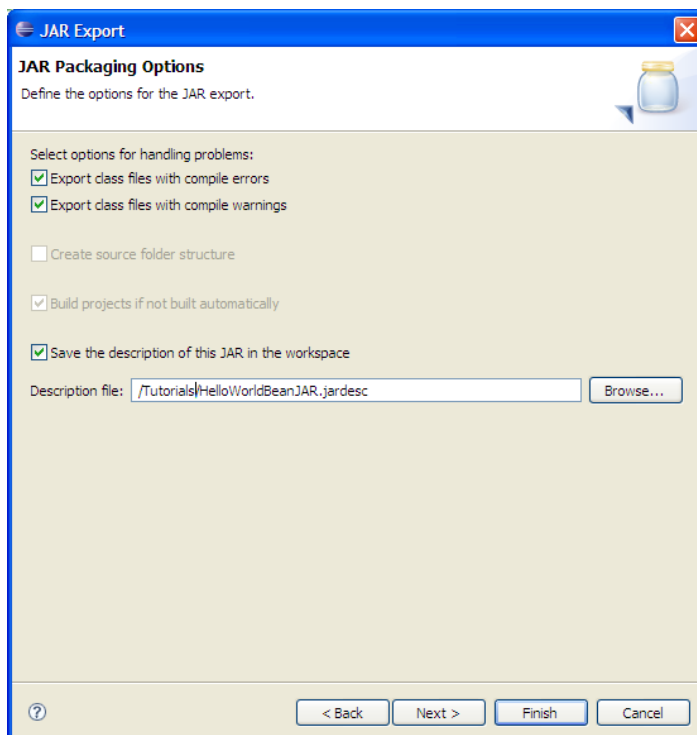
3. When you need to re-build and re-deploy your component just select "Create JAR":