EJB 3 Struts Framework Integration Tutorial

This tutorial explains step by step how to set up a EJB 3 project using a web framework like Struts. Struts can be replaced by any framework you like, including Spring MVC with Tapestry, JSF etc.

The **first EJB 3 tutorial** is required as, we will not explain EJB 3 basics here. Nevertheless, this tutorial is easy to follow.

General

Author: Sebastian Hennebrueder

Date: March, 15th 2006

Used software and frameworks

Eclipse 3.x

MyEclipse 4 (optional but recommended)

Source code: Source code

PDF version of the tutorial: first-ejb3-tutorial-en.pdf

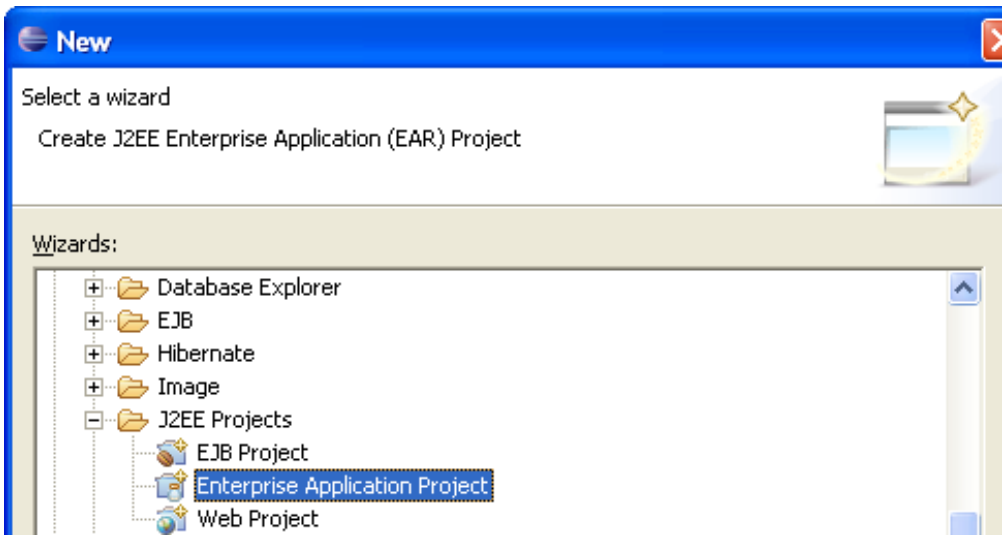
Source code: http://www.laliluna.de/download/ejb3-struts-tutorial.zip

PDF version of the tutorial:

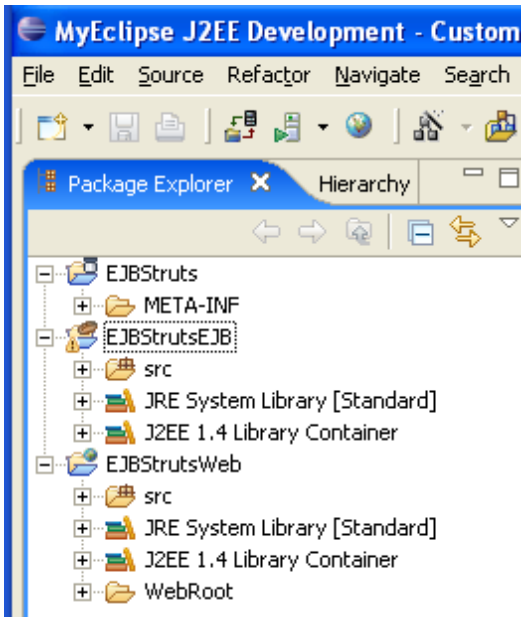
http://www.laliluna.de/download/ejb3-struts-tutorial-en.pdf

Set up a project with MyEclipse

Create a new EAR project. You can use the same project as for EJB 2. The only difference is a file we will add later. I chose **EJBStruts** as name.



Use the wizard to create an EJB project and a Web project, as well. After this you should find the following projects in your package view.



As we are going to use Entity beans, we need some kind of datasource. This can be configured in a file named persistence.xml.

Create a file named **persistence.xml** in the folder **META-INF.**

JBoss supports the tag hibernate.hbm2ddl.auto to define if your tables are created or udpated during redeployment. I chose create-drop to have them dropped after each undeployment, so that they can be nicely recreated. The option update does not work sometimes.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<persistence>
  <persistence-unit name="Ejb3StrutsTutorial">
    <jta-data-source>java:/ejb3ProjectDS</jta-data-source>
```

```
    <properties>
      <property name="hibernate.hbm2ddl.auto"
                value="create-drop"/>
<!-- other supported properties, we are not using here. Do not forget to put them into
a <property ... tag.

hibernate.transaction.manager_lookup_class=org.hibernate.transaction.JBossTransactionM
anagerLookup
hibernate.connection.release_mode=after_statement
hibernate.transaction.flush_before_completion=true
hibernate.transaction.auto_close_session=false
hibernate.query.factory_class=org.hibernate.hql.ast.ASTQueryTranslatorFactory
#hibernate.hbm2ddl.auto=create-drop
#hibernate.hbm2ddl.auto=create
hibernate.cache.provider_class=org.hibernate.cache.HashtableCacheProvider
# Clustered cache with TreeCache
#hibernate.cache.provider_class=org.jboss.ejb3.entity.TreeCacheProviderHook
#hibernate.treecache.mbean.object_name=jboss.cache:service=EJB3EntityTreeCache
#hibernate.dialect=org.hibernate.dialect.HSQLDialect
hibernate.jndi.java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
hibernate.jndi.java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
hibernate.cglib.use_reflection_optimizer=false
-->
    </properties>
  </persistence-unit>
</persistence>
```

JNDI data source

Download your database driver and put it into JBOSS_HOME\server\default\lib. Restart your server.

Create a file named myFavouriteName**-ds.xml**. There is a naming convention. Please keep the bold text. You can find a lot of examples for different databases in the installation path of JBoss. **JBOSS_HOME/docs/examples/jca**

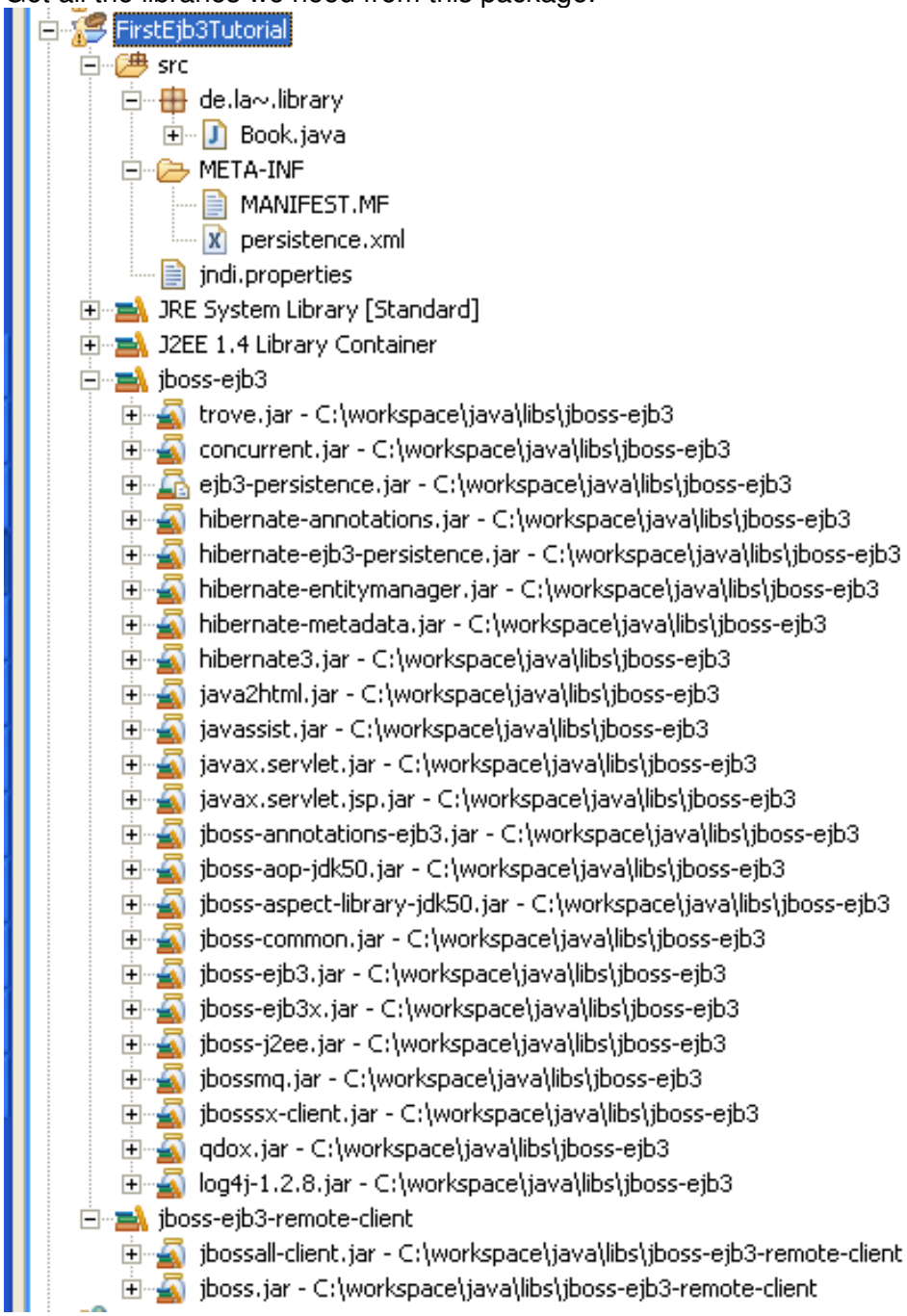
A datasource for PostgreSql looks like

```
<?xml version="1.0" encoding="UTF-8"?>

<datasources>
    <local-tx-datasource>
    <jndi-name>ejb3ExampleDS</jndi-name>
    <connection-url>jdbc:postgresql://localhost:5432/examples</connection-url>
    <driver-class>org.postgresql.Driver</driver-class>
    <user-name>postgres</user-name>
    <password>p</password>
    <!-- the minimum size of the connection pool -->
    <min-pool-size>1</min-pool-size>
    <!-- The maximum connections in a pool/sub-pool -->
     <max-pool-size>4</max-pool-size>
  </local-tx-datasource>
</datasources>
```

Add needed libraries to the project.

We will need some libraries during development of ejb3 and some for using a remote client to test

our application later on.
You need to collect the libraries together. I recommend to pack them into a user library. Than you will have this work only once.
Download JBoss EJB3 at http://www.jboss.org/products/list/downloads
Get all the libraries we need from this package.

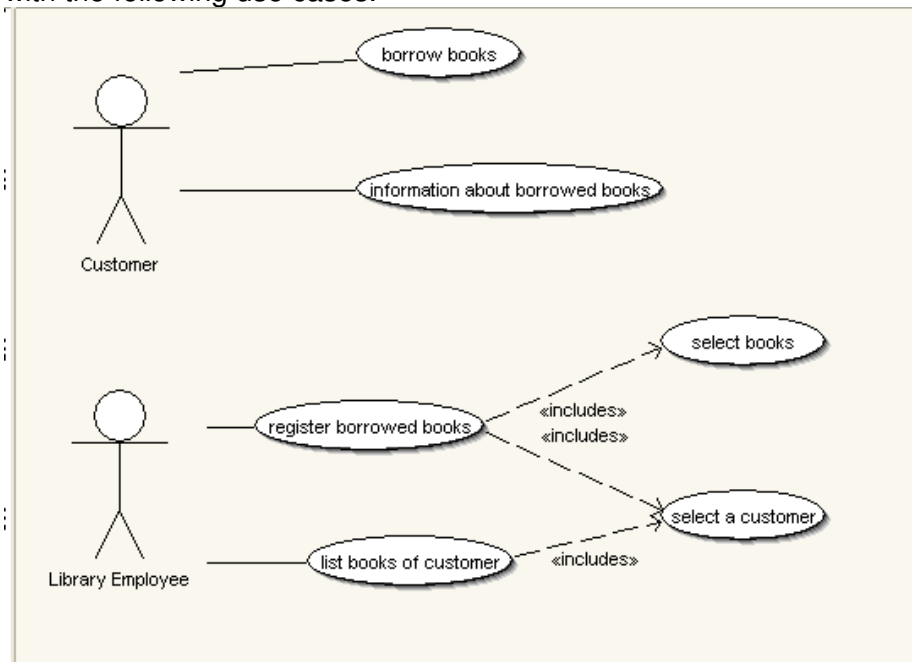


Than have a look into your JBoss directory. We will need the jbossallclient.jar and the jboss.jar.
My directory when I am working under Windows:
E:\jboss-4.0.4RC1\client
There is a fair chance that I selected to many libraries. Try if you like which one you can delete.

EJB 3 Business Layer

Use case overview

You have successfully acquired a large project for a library. After weeks of analysis you came up with the following use cases.



Further analysis showed that we need to classes representing so called domains:

Book and Customer.

Further we need a business class implementing our use cases.

Congratulations, you have finished the complex analysis and we will continue to implement our project.

Creating Entity Beans

We need two Entity beans: Book and Customer.

Create the Book Entity Bean

Create a new class **Book** in the package **de.laliluna.library**

Add the attributes

```
private Integer id;
private String title;
private String author;
```

Select **Generate Getter/Setter** from the Source Menu.

In Eclipse you can reach the function with **Alt+Shift + S** or with the context menu (right mouse click) of the source code.

Add the following constructors and implement the toString method. (Alt+Shift+S + Override/Implement methods). This is useful for debugging.

```
public Book() {
```

```
  super();
}

public Book(Integer id, String title, String author) {
  super();
  this.id = id;
  this.title = title;
  this.author = author;
}
@Override
public String toString() {

  return "Book: " + getId() + " Title " + getTitle() + " Author "
      + getAuthor();
}
```

Implement the interface java.io.Serializable. It is a marker interface. This means you do not have to implement any methods (normally).

Finally, this is our full source code now:

```
package de.laliluna.library;

import java.io.Serializable;

/**
 * @author hennebrueder
 *
 */
public class Book implements Serializable {
 /**
  *
  */
 private static final long serialVersionUID = 7422574264557894633L;

 private Integer id;

 private String title;

 private String author;

 public Book() {
   super();
 }

 public Book(Integer id, String title, String author) {
   super();
   this.id = id;
   this.title = title;
   this.author = author;
 }
 @Override
 public String toString() {

   return "Book: " + getId() + " Title " + getTitle() + " Author "
       + getAuthor();
```

```
}
public String getAuthor() {
  return author;
}
public void setAuthor(String author) {
  this.author = author;
}
public Integer getId() {
  return id;
}
public void setId(Integer id) {
  this.id = id;
}
public String getTitle() {
  return title;
}
public void setTitle(String title) {
  this.title = title;
}
}
```

Recommendation

In general I recommend to do the following with all Domain objects, especially when you use them as Entity Beans. Domain objects are things like Address, Book, Customer in contrast to business logic like MailFactory, AuthorizeFilter.

Create an empty constructor and a useful one. The empty is sometimes needed for reflection.

Implement the interface java.io.Serializable as entity beans are frequently serialized by caches, by the entity manager etc.

Overwrite the toString method because a meaningful output is useful for debugging.

Adding the Annotations

Now, we will add the annotations:

```
@Entity
@Table(name="book")
@SequenceGenerator(name = "book_sequence", sequenceName = "book_id_seq")
public class Book implements Serializable {
```

Entity defines that this is an entity bean. The second defines the table name. The last one defines a sequence generator.

Primary keys can be generated in different ways:

You can assign them. For example a language table and the primary key is the ISO-Country code id: EN,DE,FR, ...

Use a sequence for PostgreSql, SapDb, Oracle and other . A sequence is a database feature. It returns the next Integer or Long value each time it is called.

In MsSql and other you can use identity.

Sequence primary key

I am using PostgreSql, so I defined the sequence first in Order to use it later for my primary key. In

front of the getId I configure the ID and the generation approach.

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "book_sequence")
public Integer getId() {
 return id;
}
```

Important

generator = "**book_sequence**" referes to the named defined in front of your class

Identity primary key

For MSSql Server you will probably only need

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
public Integer getId() {
 return id;
}
```

I am sorry, but I could not test this. It may not work.

Table based primary key

Here is one solution that always works: It safes the primary keys in a separated table. One row for each primary key. Define it in front of your class:

```
@TableGenerator(  name="book_id", table="primary_keys", pkColumnName="key",
pkColumnValue="book",
  valueColumnName="value")
```

and use it:

```
@Id
@GeneratedValue(strategy = GenerationType.TABLE, generator = "book_id")
public Integer getId() {
```

Important

generator = "**book_id** refers to the name defined in front of your class @TableGenerator(name="**book_id**"

Create the Customer Entity Bean

Repeat the steps explained for the books. Here is the full source code, you should have created.

```
/**
 *
 * @author Sebastian Hennebrueder
 * created Mar 15, 2006
 * copyright 2006 by http://www.laliluna.de
 */
package de.laliluna.library;

import javax.persistence.Entity;
```

```
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;

/**
 * @author hennebrueder
 *
 */
@Entity
@Table(name="customer")
@SequenceGenerator(name="customer_sequence", sequenceName="customer_id_seq")
public class Customer {
 private Integer id;

 private String name;

 public Customer(){
   super();
 }
 public Customer(Integer id, String name){
   super();
   this.id=id;
   this.name=name;
 }

 @Id
 @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="customer_sequence")
 public Integer getId() {
   return id;
 }

 public void setId(Integer id) {
   this.id = id;
 }

 public String getName() {
   return name;
 }

 public void setName(String name) {
   this.name = name;
 }

 @Override
 public String toString() {
   return "Customer: " + getId() + " Name " + getName();
 }

}
```

Defining the relation between customer and book

Every customer can borrow many books but a book can only be borrowed by one customer at the same time. Consequently, we have to deal with a 1:n relation.

Add a attribute called books to the customer class.

```
private List books =new ArrayList();
```

I recommend to initialize it always directly or you will have to check for null everywhere you access the books.

Create the getter and setter methods.

```
public List getBooks() {
  return books;
}

public void setBooks(List books) {
  this.books = books;
}
```

Now we have to add the annotations:

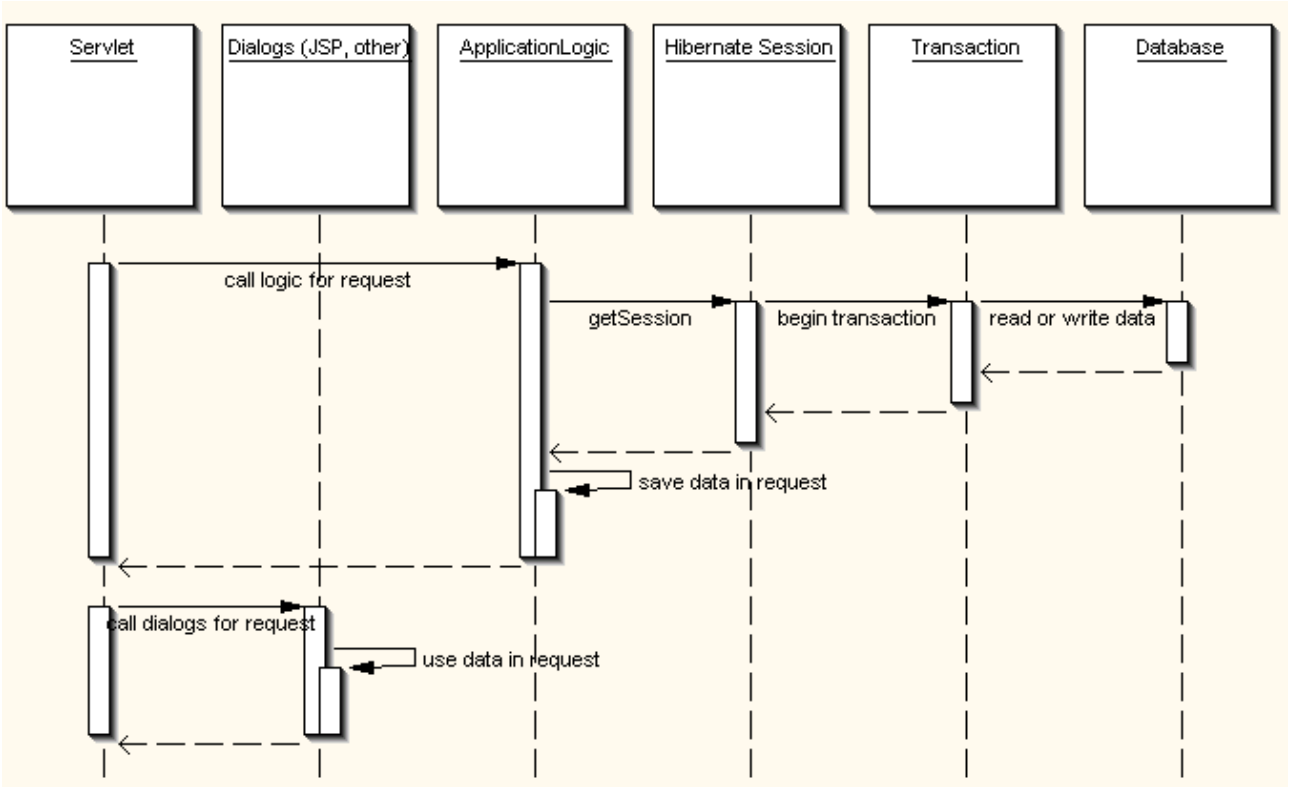
```
@OneToMany(cascade=CascadeType.ALL, fetch = FetchType.EAGER, mappedBy="customer",
targetEntity=Book.class)
public List getBooks() {
  return books;
}
```

The annotations **cascade=CascadeType.ALL** defines that when we update a customer, we update the book as well. When we create a customer, we create the books as well and if we delete the customer ..... you are wrong when you think that the books will be deleted. Only the relation will be deleted.

I used cascading here for ease when creating test data. This is only partial useful for our use cases.

The annotation **fetch = FetchType.EAGER** defines that if we select a customer all books borrowed by this customer are loaded as well. The reason is that EJB 3 does not allow loading of data when your current persistence context is closed.

The persistence context is started in your business logic normally and closed after the business logic has been processed. It is bound to a JTA transaction of the application server. If you know the Hibernate framework you know this concept as well. The following diagram is taken from my eBook Hibernate developer guide.



The diagram explains that your session (Hibernate) which is equivalent to persistence context (EJB 3) is already closed when you render your dialogue.

Further information about Lazy Loading and other strategies can taken either from my Hibernate eBook (Hibernate is being used by JBoss as EJB 3 implementation) or you wait for my EJB 3 eBook, which hopefully is finished in a first version in April (2006).

The last two annotations **mappedBy="customer", targetEntity=Book.class** belong together. They define that the other side of the relation is the Book class and that the relation is defined by the customer property.

That's it. You have successfully implemented the entity bean part.

Stateless Session Bean

A stateless session bean has not state, i.e. It performs some actions and is thrown away afterwards. Therefore it is not suitable as shopping cart class. The shopping cart must save the cart information during multiple requests. It has a state => you would use a stateful session bean.

Create local and remote interfaces

The local interface should be used by default, because it is much faster. The remote interface should only be used when the client is not running in the same virtual machine. Remote access even works over the network and has a lot of overhead.

Create a interface named **BookTestBeanLocal** in the package **de.laliluna.library**.

We mark this interface as local interface by the annotation **@Local**.

```
package de.laliluna.library;

import javax.ejb.Remote;

@Remote
public interface BookTestBeanRemote {
```

```
public void testBook();
public void testCustomer();
public void testRelation();
}
```

Create a interface named **BookTestBeanRemote** in the package **de.lalilluna.library**;

```
package de.lalilluna.library;
import javax.ejb.Remote;

@Remote
public interface BookTestBeanRemote {
public void testBook();
public void testCustomer();
public void testRelation();
}
```

Now we will create the actual Stateless Session Bean.

Create a new class named in the same package as the interfaces and let it implement the local and the remote interface.

You configure a class as stateless bean by adding the **@Stateless** annotation.

```
package de.lalilluna.library;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
public class BookTestBean implements BookTestBeanLocal, BookTestBeanRemote {
@PersistenceContext
EntityManager em;
public static final String RemoteJNDIName =  BookTestBean.class.getSimpleName() +
"/remote";
public static final String LocalJNDIName =  BookTestBean.class.getSimpleName() +
"/local";
}
```

We want to access the book bean, so we need a EntityManager. The EntityManager provides all methods needed to select, update,lock or delete entities, to create SQL and EJB-QL queries.

```
@PersistenceContext
EntityManager em;
```

The annotation **@PersistenceContext** tells the application server to inject a entity manager during deployment. Injection means that the entity manager is assigned by the application server.

This is very useful approach frequently used in the Spring Framework or other Aspect Oriented Framework. The idea is:

A data access class should not be responsible for the persistenceContext. My configuration decides which context for which database it receives. Imagine you hard code a context in 25 classes and than want to change the context.

I like it to have the JNDI name of my class somewhere, so I do not have to type it. This is why I added the following lines.

```
public static final String RemoteJNDIName =  BookTestBean.class.getSimpleName() +
"/remote";
public static final String LocalJNDIName =  BookTestBean.class.getSimpleName() +
"/local";
```

Implementing the test method. The following test method creates an entry, selects some and deletes an entry as well. Everything is done using the entity manager. You may read in the API documentation about the other methods of this manager.

```
/**
 *
 * @author Sebastian Hennebrueder
 * created Mar 15, 2006
 * copyright 2006 by http://www.laliluna.de
 */
package de.laliluna.library;

import java.util.Iterator;
import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

/**
 * @author hennebrueder
 *
 */
@Stateless
public class BookTestBean implements BookTestBeanLocal, BookTestBeanRemote {

 @PersistenceContext
 EntityManager em;

 public static final String RemoteJNDIName = BookTestBean.class
     .getSimpleName()
     + "/remote";

 public static final String LocalJNDIName = BookTestBean.class
     .getSimpleName()
     + "/local";

 /*
  * (non-Javadoc)
  *
  * @see de.laliluna.library.BookTestBean#test()
  */
 public void testBook() {
   Book book = new Book(null, "My first bean book", "Sebastian");
   em.persist(book);
   Book book2 = new Book(null, "another book", "Paul");
   em.persist(book2);
   Book book3 = new Book(null, "EJB 3 developer guide, comes soon",
       "Sebastian");
   em.persist(book3);

   System.out.println("list some books");
```

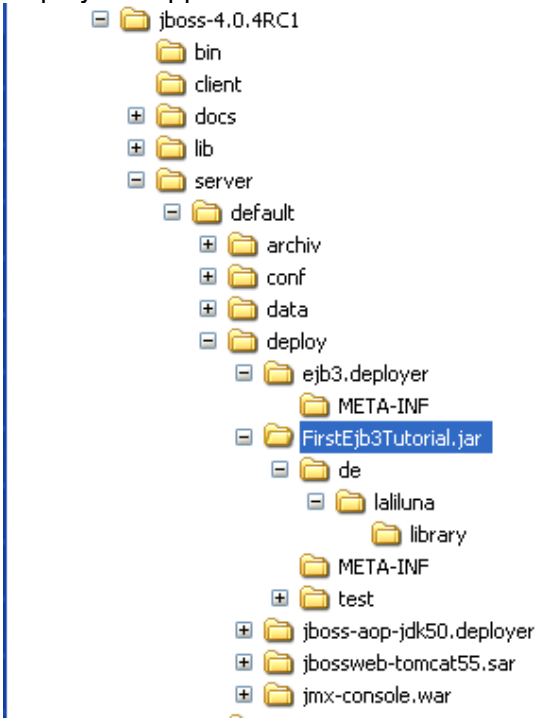
```
  List someBooks = em.createQuery("from Book b where b.author=:name")
      .setParameter("name", "Sebastian").getResultList();
  for (Iterator iter = someBooks.iterator(); iter.hasNext();)
  {
    Book element = (Book) iter.next();
    System.out.println(element);
  }
  System.out.println("List all books");
  List allBooks = em.createQuery("from Book").getResultList();
  for (Iterator iter = allBooks.iterator(); iter.hasNext();)
  {
    Book element = (Book) iter.next();
    System.out.println(element);
  }
  System.out.println("delete a book");
  em.remove(book2);
  System.out.println("List all books");
  allBooks = em.createQuery("from Book").getResultList();
  for (Iterator iter = allBooks.iterator(); iter.hasNext();)
  {
    Book element = (Book) iter.next();
    System.out.println(element);
  }
}

public void testCustomer() {
  Customer customer1 = new Customer(null, "Sebastian the reader");
  em.persist(customer1);
  Customer customer2 = new Customer(null, "Sebastian the reader");
  em.persist(customer2);
  Customer customer3 = new Customer(null, "Sebastian the reader");
  em.persist(customer3);

  System.out.println("list some customers");
  List someCustomers = em.createQuery(
      "from Customer c where c.name like :name").setParameter("name",
      "Sebastian%").getResultList();
  for (Iterator iter = someCustomers.iterator(); iter.hasNext();)
  {
    Customer element = (Customer) iter.next();
    System.out.println(element);
  }
  System.out.println("List all customers");
  List allCustomers = em.createQuery("from Customer").getResultList();
  for (Iterator iter = allCustomers.iterator(); iter.hasNext();)
  {
    Customer element = (Customer) iter.next();
    System.out.println(element);
  }
  System.out.println("delete a customer");
  em.remove(customer2);
  System.out.println("List all customers");
  allCustomers = em.createQuery("from Customer").getResultList();
  for (Iterator iter = allCustomers.iterator(); iter.hasNext();)
  {
```

```
    Customer element = (Customer) iter.next();
    System.out.println(element);
  }

}

public void testRelation() {

  Customer customer1 = new Customer(null, "Sebastian the reader");
  em.persist(customer1);
  Customer customer2 = new Customer(null, "Sebastian the reader");
  em.persist(customer2);
  Book book = new Book(null, "My first bean book", "Sebastian");
  em.persist(book);
  Book book2 = new Book(null, "another book", "Paul");
  em.persist(book2);
  Book book3 = new Book(null, "EJB 3 developer guide, comes soon",
      "Sebastian");
  em.persist(book3);

  // all objects are in peristence state, so any changes will be reflected
  // in the db
  /*
   * important you must set both side of a relation, EJB3 will not update
   * the other side automatically. Your objects in your application could
   * be inconsistent, if you do not do this. This is only a problem when
   * setting relations not when you load data.
   */
  customer1.getBooks().add(book);
  book.setCustomer(customer1);
  customer2.getBooks().add(book2);
  customer2.getBooks().add(book3);

  System.out.println("list some customers");
  List someCustomers = em.createQuery(
      "from Customer c where c.name like :name").setParameter("name",
      "Sebastian%").getResultList();
  for (Iterator iter = someCustomers.iterator(); iter.hasNext();)
  {
    Customer element = (Customer) iter.next();
    System.out.println(element);
    for (Iterator iterator = element.getBooks().iterator(); iterator
        .hasNext();)
    {
      Book aBook = (Book) iterator.next();
      System.out.println("--" + aBook);
    }
  }
  System.out.println("List all customers");
  List allCustomers = em.createQuery("from Customer").getResultList();
  for (Iterator iter = allCustomers.iterator(); iter.hasNext();)
  {
    Customer element = (Customer) iter.next();
    System.out.println(element);
  }
```

```
   System.out.println("delete a customer");
   em.remove(customer2);
   System.out.println("List all customers");
   allCustomers = em.createQuery("from Customer").getResultList();
   for (Iterator iter = allCustomers.iterator(); iter.hasNext();)
   {
     Customer element = (Customer) iter.next();
     System.out.println(element);
   }
   System.out
       .println("let us see what is happening with the books after we deleted the
customer");
   System.out.println("List all customers");
   List allBooks = em.createQuery("from Book").getResultList();
   for (Iterator iter = allBooks.iterator(); iter.hasNext();){
     Book aBook = (Book) iter.next();
     System.out.println(aBook);
     System.out.println("--"+aBook.getCustomer());
   }
 }

}
```

Deploy the application

Deploy the application now. It must be deployed as jar. Here is how it looks like in my explorer:



Than open the jmx-console to verify the deployment.
http://localhost:8080/jmx-console/
Select the JNDI View and list to show all deployed beans.
You should see something like the following when the deployment was successful.

```
|   NonContext: null
+- TransactionPropagationContextImporter (class: org.jboss.tm.TransactionPropagationContextImporter)
+- ejb3ExampleDS (class: org.jboss.resource.adapter.jdbc.WrapperDataSource)
+- TransactionManager (class: org.jboss.tm.TxManager)
```

Global JNDI Namespace

```
+- UserTransaction (class: org.jboss.tm.usertx.client.ClientUserTransaction)
+- LibraryCart (class: org.jnp.interfaces.NamingContext)
+- BookDaoImp (class: org.jnp.interfaces.NamingContext)
+- UserTransactionSessionFactory (proxy: $Proxy27 implements interface org.jboss.tm.usertx.interfaces
+- jmx (class: org.jnp.interfaces.NamingContext)
|   +- invoker (class: org.jnp.interfaces.NamingContext)
|   |   +- RMIAdaptor (proxy: $Proxy26 implements interface org.jboss.jmx.adaptor.rmi.RMIAdaptor,inte
|   +- rmi (class: org.jnp.interfaces.NamingContext)
|   |   +- RMIAdaptor[link -> jmx/invoker/RMIAdaptor] (class: javax.naming.LinkRef)
+- CustomerDaoImp (class: org.jnp.interfaces.NamingContext)
+- BookTestBean (class: org.jnp.interfaces.NamingContext)
|   +- local (proxy: $Proxy120 implements interface de.lalilуna.library.BookTestBeanLocal,interface o
|   +- remote (proxy: $Proxy119 implements interface de.lalilуna.library.BookTestBeanRemote,interface
```

Create a test client

First, create a simple **log4j.properties** file in the **src** folder.
Add the following content:

```
### direct log messages to stdout ###
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n

### direct messages to file hibernate.log ###
#log4j.appender.file=org.apache.log4j.FileAppender
#log4j.appender.file.File=hibernate.log
#log4j.appender.file.layout=org.apache.log4j.PatternLayout
#log4j.appender.file.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n

### set log levels - for more verbose logging change 'info' to 'debug' ###

log4j.rootLogger=debug, stdout
```

Then create a class named **FirstEJB3TutorialClient** in the package **test.de.lalilуna.library**.
Either create a file named jndi.properties in the src directory.

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
java.naming.provider.url=localhost:1099
```

and use

```
context = new InitialContext();
```

Or configure the JNDI in your application:

```
 Properties properties = new Properties();

properties.put("java.naming.factory.initial","org.jnp.interfaces.NamingContextFactory"
```

```
);

properties.put("java.naming.factory.url.pkgs","=org.jboss.naming:org.jnp.interfaces");
  properties.put("java.naming.provider.url","localhost:1099");
  Context context = new InitialContext(properties);
```

Anyway here is the full source code of the test client:

```
/**
 *
 * @author Sebastian Hennebrueder
 * created Mar 15, 2006
 * copyright 2006 by http://www.laliluna.de
 */
package test.de.laliluna.library;

import java.util.Properties;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import de.laliluna.library.BookTestBean;
import de.laliluna.library.BookTestBeanRemote;

/**
 * @author hennebrueder
 *
 */
public class FirstEJB3TutorialClient {

 /**
  * @param args
  */
 public static void main(String[] args) {
   /* get a initial context. By default the settings in the file
    * jndi.properties are used.
    * You can explicitly set up properties instead of using the file.
Properties properties = new Properties();
properties.put("java.naming.factory.initial","org.jnp.interfaces.NamingContextFactory"
);
properties.put("java.naming.factory.url.pkgs","=org.jboss.naming:org.jnp.interfaces");
 properties.put("java.naming.provider.url","localhost:1099");
 */

   Context context;
   try
   {
      context = new InitialContext();
      BookTestBeanRemote beanRemote = (BookTestBeanRemote)
       context.lookup(BookTestBean.RemoteJNDIName);
           beanRemote.testBook();
      beanRemote.testCustomer();
      beanRemote.testRelation();
   } catch (NamingException e)
   {
     e.printStackTrace();
```

```
    /* I rethrow it as runtimeexception as there is really no need to continue if an
exception happens and I
     * do not want to catch it everywhere.
     */
    throw new RuntimeException(e);
   }
 }
}
```

Create a DAO layer

A Data Access Object (DAO) is a design pattern frequently used to encapsulate access to databases or to entity beans. It provides all methods to save, delete or select our entities. I will implement it as Stateless Session Bean as well.

Create a interface **BookDao.** We only need to create the local interface. As the DAO will only be used locally by other session beans.

```
package de.laliluna.library;

import java.util.List;

import javax.ejb.Local;

import org.jboss.annotation.ejb.LocalBinding;

@Local
public interface BookDao {
 public void save(Book book);
 public void merge(Book book);
 public List findAll();
 public List findByCustomer(Customer customer);
 public Book findById(Integer id);

}
```

Create a new class **BookDaoImp** and add the following content. This class provides all the methods we will need throughout the application when accessing the bean.

```
package de.laliluna.library;

import java.util.ArrayList;
import java.util.List;

import javax.ejb.Stateful;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import org.apache.log4j.Logger;

@Stateless
public class BookDaoImp implements BookDao {

 @PersistenceContext
 private EntityManager em;
```

```
private Logger log = Logger.getLogger(this.getClass());

/*
 * (non-Javadoc)
 *
 * @see de.laliluna.example.ejb.BookDao#save(de.laliluna.example.ejb.Book)
 */
public void save(Book book) {
  log.debug("Persist book: " + book);
  em.persist(book);
}

public void merge(Book book) {
  em.merge(book);
}

public List findAll() {
  log.debug("find All books");
  return em.createQuery("from Book").getResultList();
}

public static final String RemoteJNDIName = BookDaoImp.class
    .getSimpleName()
    + "/remote";

public static final String LocalJNDIName = BookDaoImp.class.getSimpleName()
    + "/local";

public Book findById(Integer id) {
  return em.find(Book.class, id);
}

public List findByCustomer(Customer customer) {
  log.debug("find by customer");
  return em.createQuery("from Book b where b.customer = :customer")
      .setParameter("customer", customer).getResultList();
}
}
```

Then we will do the same for our Customer class. Here is the interface:

```
package de.laliluna.library;

import java.util.List;
import javax.ejb.Local;

@Local
public interface CustomerDao {
public void save(Customer customer);
public void merge(Customer customer);
public List findAll();
public Customer findById(Integer id);
}
```

The implementation is not very complex and does not anything new as well.

```
package de.laliluna.library;

import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.Entity;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
public class CustomerDaoImp implements CustomerDao {

 @PersistenceContext
 private EntityManager em;

 public static final String RemoteJNDIName =  CustomerDaoImp.class.getSimpleName() +
"/remote";

 public static final String LocalJNDIName =  CustomerDaoImp.class.getSimpleName() +
"/local";

 public void save(Customer customer) {
   em.persist(customer);
 }

 /* (non-Javadoc)
  * @see de.laliluna.example.ejb.CustomerDao#reattach(de.laliluna.example.ejb.Book)
  */
 public void merge(Customer customer) {
 em.merge(customer);

 }

 /* (non-Javadoc)
  * @see de.laliluna.example.ejb.CustomerDao#findAll()
  */
 public List findAll() {

   return em.createQuery("from Customer").getResultList();
 }

 /* (non-Javadoc)
  * @see de.laliluna.example.ejb.CustomerDao#findById(java.lang.Integer)
  */
 public Customer findById(Integer id) {
   Customer customer = em.find(Customer.class, id);
   return customer;
 }

}
```

That's all. You are a master of DAO objects, now. To read more about DAO you may look in my eBook Hibernate Developer Guide by example.

Stateful Session Bean - business logic

I want to recall you the use cases we have. In order to register the borrow of books, we need to

select a customer and books. In fact we have some kind of cart holding the books for the current customer. When we checkout the books are marked as borrowed and the cart is cleaned up. We will need to get the cart content, as well. Here is our interface.

```
package de.laliluna.library;

import java.util.Map;


import javax.ejb.Local;
@Local
public interface LibraryCartLocal {
 /**
  * ad a book to the cart
  * @param book
  */
 public void addBook(Book book);
 /**
  * find a Book for the give Id and add it to the cart
  * @param id
  */
 public void addBook(Integer id);
 /**
  * remove a book from the cart
  * @param book
  */
 public void removeBook(Book book);
 /**
  *  find a Book for the give Id and remove it from the cart
  * @param id
  */
 public void removeBook(Integer id);
 /**
  * remove all books from the cart
  *
  */
 public void clearCart();
 /**
  * return the cart as java.util.Map containing the id of the book as key and the book
itself as value.
  * @return
  */
 public Map getCart();
 /** check out and save the borrow of the book
  * mark all books as borrowed by the specified customer, destroy the stateful session
bean afterwards
  * @param customer
  */
 public void checkOut(Customer customer);
 /**
  * mark a borrowed book as returned which is equivalent to removing the relation
between customer and book
  * @param id
  */
 public void returnBook(Integer id);
}
```

The remote interface is only needed when you want to test the application. Apart from the annotation the content is the same. You are nearly an EJB 3 expert, so you will manage to set up this interface for yourself.

I will continue to our **LibraryCart** class. Have a look at it, I will explain some interesting details afterwards.

```
package de.laliluna.library;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Set;
import javax.ejb.Remove;
import javax.ejb.Stateful;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import org.apache.log4j.Logger;
import de.laliluna.library.Common;


@Stateful
public class LibraryCart implements LibraryCartLocal {
 @PersistenceContext
 private EntityManager em;

 private Logger log = Logger.getLogger(this.getClass());

 private Map<Integer, Book> books = new HashMap<Integer, Book>();

 public static final String RemoteJNDIName = LibraryCart.class
     .getSimpleName()
     + "/remote";

 public static final String LocalJNDIName = LibraryCart.class
     .getSimpleName()
     + "/local";

 /*
  * (non-Javadoc)
  *
  * @see de.laliluna.example.ejb.BookCartLocal#addBook(de.laliluna.example.ejb.Book)
  */
 public void addBook(Book book) {
   if (books.get(book.getId()) == null)
     books.put(book.getId(), book);

 }

 public void addBook(Integer id) {
   if (books.get(id) == null)
```

```
  {
    BookDao bookDao;
    try
    {
      Context context = new InitialContext();
      bookDao = (BookDao) context.lookup(BookDaoImp.LocalJNDIName);
    } catch (NamingException e)
    {
      e.printStackTrace();
      throw new RuntimeException(e);
    }
    Book book = bookDao.findById(id);
    if (book != null)
      books.put(id, book);
  }

}

/*
 * (non-Javadoc)
 *
 * @see
de.laliluna.example.ejb.BookCartLocal#removeBook(de.laliluna.example.ejb.Book)
 */
public void removeBook(Book book) {
  books.remove(book.getId());
}

/*
 * (non-Javadoc)
 *
 * @see
de.laliluna.example.ejb.BookCartLocal#removeBook(de.laliluna.example.ejb.Book)
 */
public void removeBook(Integer id) {
  if (id != null)
    books.remove(id);

}

@Remove
// remove the statefull bean after this method is called
public void checkOut(Customer customer) {
  BookDao bookDao;
  CustomerDao customerDao;
  try
  {
    Context context = new InitialContext();
    bookDao = (BookDao) context.lookup(BookDaoImp.LocalJNDIName);
    customerDao = (CustomerDao) context
        .lookup(CustomerDaoImp.LocalJNDIName);

  } catch (NamingException e)
  {
    e.printStackTrace();
```

```
    throw new RuntimeException(e);
  }
  log.debug("checkout for: " + customer);
  for (Iterator iter = books.keySet().iterator(); iter.hasNext();)
  {
    Book book = books.get(iter.next());
    log.debug("checkout: " + book);
    book.setCustomer(customer);
    customer.getBooks().add(book);
    bookDao.merge(book);
  }
}

public void returnBook(Integer id) {
  BookDao bookDao;
  CustomerDao customerDao;
  try
  {
    Context context = new InitialContext();
    bookDao = (BookDao) context.lookup(BookDaoImp.LocalJNDIName);
    customerDao = (CustomerDao) context
        .lookup(CustomerDaoImp.LocalJNDIName);

  } catch (NamingException e)
  {
    e.printStackTrace();
    throw new RuntimeException(e);
  }
  Book book = bookDao.findById(id);
  if (book != null)
  {
    Customer customer = book.getCustomer();
    customer.getBooks().remove(book);
    book.setCustomer(null);

  }

}

/*
 * (non-Javadoc)
 *
 * @see de.laliluna.example.ejb.BookCartLocal#clearCart()
 */
public void clearCart() {
  books.clear();
}

/*
 * (non-Javadoc)
 *
 * @see de.laliluna.example.ejb.BookCartLocal#getCart()
 */
public Map getCart() {
  return books;
```

```
}
```

There are only few interesting things you might come across.

Firstly, we define a Stateful Session bean by adding the annotation **@Stateful** to the class.

```
@Stateful
public class LibraryCart implements LibraryCartLocal {
```

The class uses an entity manager. An entity manager provides all methods needed to insert,update or delete entities or to create queries. This was explained in the **First EJB 3 tutorial.**

By adding the annotation @PersistenceContext we instruct the container to inject an entity manager to the attribute. To inject means that he initializes the attribute and sets it to a working entity manager.

```
@PersistenceContext
private EntityManager em;
```

The next interesting annotation is @Remove. This instructs the EJB container to remove the StatefulSessionBean after this method is processed.

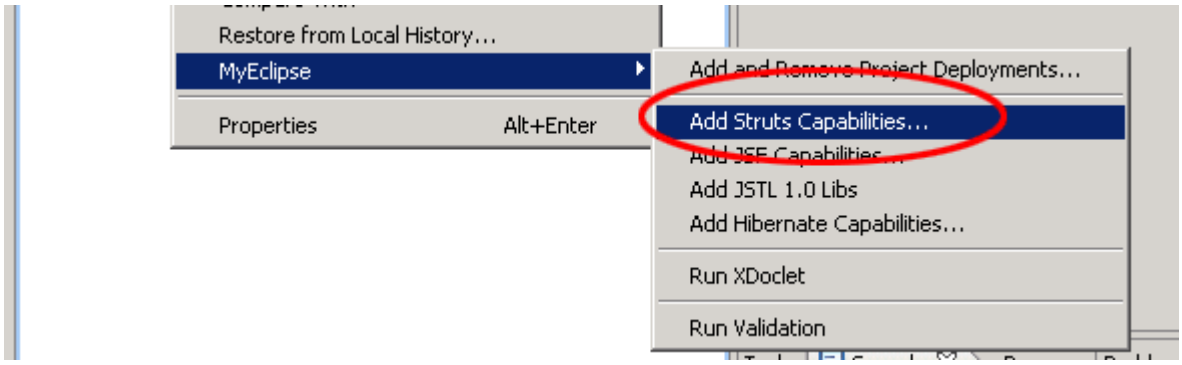
```
@Remove
// remove the statefull bean after this method is called
public void checkOut(Customer customer)
```

This is all for our EJB layer. You may set up a test now, calling your Stateful Session Bean.

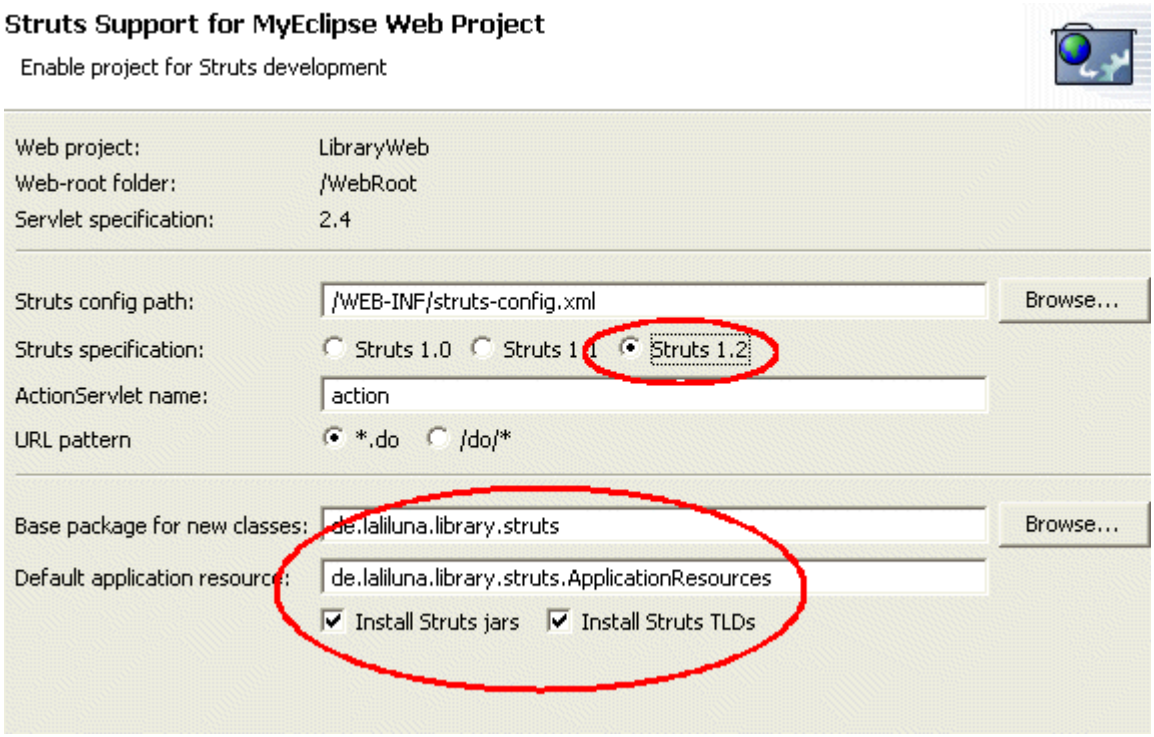
Web layer

Preparing the Struts framework

For now your project is still a normal Web project, so we need to add the struts capabilities. Right click on the project and add the capabilities for struts with **Add Struts Capabilities.**



Change the **Base package** to de.laliluna.library.struts for new classes and the **Default application resource**.



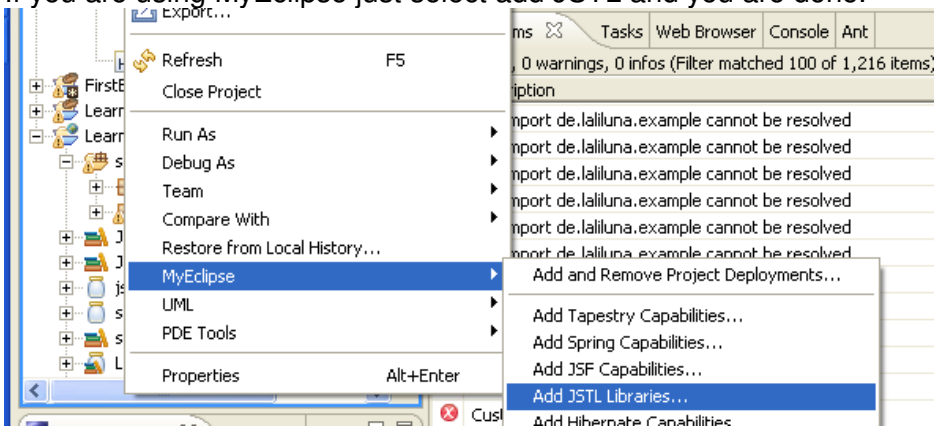
Notice

You can use the libraries and tlds found in the struts-blanc.war when you do not have MyEclipse. Download struts from

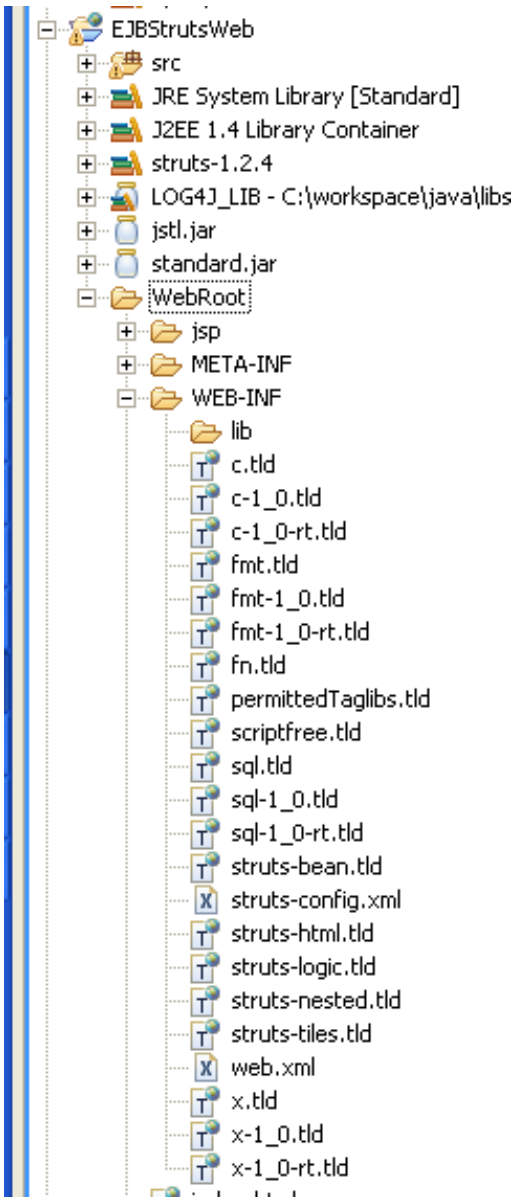
You can find the struts-blank.war in the folder jakarta-struts-1.2.4/webapps.

I like and recommend using JSTL instead of the Struts tags. It is by far more flexible and powerful. So we need to add these libraries and the tld files, as well. You can find them on Java Sun http://java.sun.com/products/jsp/jstl/downloads/index.html

If you are using MyEclipse just select add JSTL and you are done.



Your project should look like the following now:



If you are not using MyEclipse you must configure your **web.xml** by hand. You can find it in the web project in the **WEB-INF** folder.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.4"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <init-param>
      <param-name>debug</param-name>
      <param-value>3</param-value>
    </init-param>
```

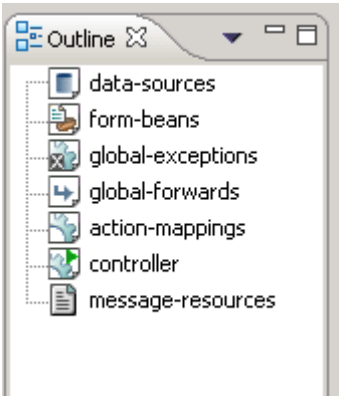
```
    <init-param>
      <param-name>detail</param-name>
      <param-value>3</param-value>
    </init-param>
    <load-on-startup>0</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

Welcome page

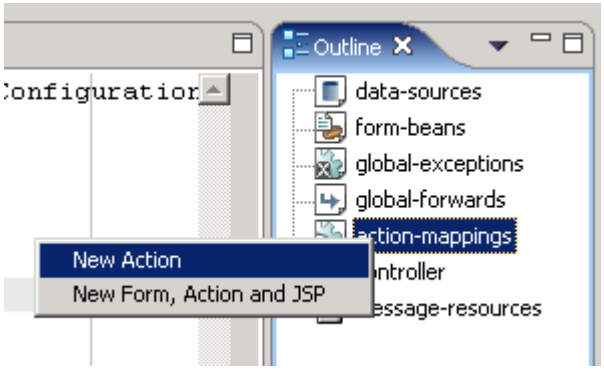
We will start with a welcome page. Create an **index.html** and just forward to our welcome action we are going to create in the next step.

```
<meta http-equiv="refresh" content="0; URL=welcome.do">
```

MyEclipse provides some nice features for creating struts files. Open the struts-config.xml and the Outline View. If you cannot see the **outline view**, open the menu **Windows -> show views** and select this view.



Click with the right mouse button on the entry action-mappings to create a new action with the wizard.



Use the wizard to create a forward action. Open the source editor and add **unknown="true"** to the configuration. This is not yet supported by MyEclipse.

For non MyEclipse user, here is the struts-config.xml we have created so far:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts
Configuration 1.2//EN" "http://struts.apache.org/dtds/struts-config_1_2.dtd">

<struts-config>
  <data-sources />
  <form-beans />
  <global-exceptions />
  <global-forwards />
  <action-mappings >
    <action forward="/jsp/welcome.jsp" path="/welcome" unknown="true" />
  </action-mappings>

  <message-resources parameter="de.laliluna.library.struts.ApplicationResources" />
</struts-config>
```

Create a new JSP **welcome.jsp** in the directory **jsp**.

It is quite simple and does include the **navigation.jsp**, we will create in the next step.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>Welcome to the library</title>
  </head>
  <body>
    <jsp:include page="/jsp/navigation.jsp"></jsp:include>
    <H2> Welcome to the library application.</H2>
     Sebastian Hennebrueder<br>
     <a href="http://www.laliluna.de">http://www.laliluna.de</a>
  </body>
</html>
```

Create a JSP named **navigation.jsp** in the same folder. It will contain our navigation reused in all of the JSPs. At the moment, we will only add one navigation option to create sample data for our application.

```
<%@ page language="java" pageEncoding="UTF-8"%>

<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
```

```
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-tiles" prefix="tiles" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<table> <TR>
<TD>
<html:link action="createSampleData">Create Sample Data</html:link>
</TD>
</TR>
</table>
```

Common variables

If you use Struts you must frequently use String variables to define a forward etc.
Quite often this is done using a static string.

```
return mapping.findForward(“success”);
```

I do not recommend this approach. Try to use static strings rarely as you can, because you can easily make typing mistakes. This is why I define standard variables in a global file. Create a class named **Common** in the package **de.laliluna.library**.

```
package de.laliluna.library;

public class Common {
 private Common(){
 }

 public static final String JNDI_PREFIX = "LearnEJB3/";

 // forward used as success
 public static final String SUCCESS = "success";
// forward used for failure
 public static final String FAILURE = "failure";
// forward used for self
 public static final String SELF = "self";

 // session key for customer
 public static final String CUSTOMER_KEY = "customer_key";

// session key for cart
 public static final String CART_KEY = "cart";
}
```

Use case: create sample data

This use case shall only facilitate the testing of our application. It creates one customer and one book each time it is called. It was not included in the analysis.

If you use MyEclipse, create a new action using the wizard. Create a forward named **success** and define it to redirect to /**welcome.do**. We do not need any FormBean.

```
    <action
      path="/createSampleData"
      type="de.laliluna.library.struts.action.CreateSampleDataAction"
```

```
      validate="false">
      <forward
        name="success"
        path="/welcome.do"
        redirect="true" />
    </action>
```

Then create the following Action class. The action gets the DAOs we created as Stateless Session Beans and uses them to create a customer and a book.

```
package de.laliluna.library.struts.action;

import java.util.Random;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

import de.laliluna.library.*;
import de.laliluna.library.struts.CustomAction;

public class CreateSampleDataAction extends CustomAction {


 /**
  * Method execute
  *
  * @param mapping
  * @param form
  * @param request
  * @param response
  * @return ActionForward
  */
 public ActionForward execute(ActionMapping mapping, ActionForm form,
     HttpServletRequest request, HttpServletResponse response) {
   log.debug("CreateSampleData");
   BookDao bookDao;
   CustomerDao customerDao;
   Random r = new Random();
   try
   {
     Context context = new InitialContext();
     bookDao = (BookDao) context.lookup(BookDaoImp.LocalJNDIName);
     customerDao = (CustomerDao) context.lookup(CustomerDaoImp.LocalJNDIName);
   } catch (NamingException e)
   {
     e.printStackTrace();
     throw new RuntimeException(e);
   }
   Book book = new Book(null, "EJB 3 Developer "
       + r.nextInt(100), "Sebastian");
   bookDao.save(book);
   Customer customer = new Customer(null, "Sebastian "+r.nextInt(100));
   customerDao.save(customer);
   return mapping.findForward(Common.SUCCESS);
 }

}
```

Deploy your application and test it a first time. If you can find new entries in the database than this part is working fine. You should reach your application at
http://localhost:8080/EJBStrutsWeb/welcome.do

Use case: Manage customer

This use case will display all books the customer has already borrowed. Furthermore, it will allow the library employee to mark books as returned.

Use the MyEclipse wizard to create a new **DispatchAction** with **action** as **parameter**. Create a forward named **success** to the JSP /**jsp**/**manageCustomer.jsp**.

```
    <action
     parameter="action"
      path="/manageCustomer"
      type="de.laliluna.library.struts.action.ManageCustomerAction"
      validate="false">
      <forward name="success" path="/jsp/manageCustomer.jsp" />
     </action>
```

Edit the action class or create it, if you are not using MyEclipse. The class provides two methods: **unspecified,** which is called when no parameter is specified. It tries to get the customer from the current session and loads his borrowed books from the database to show them later in the JSP.

The second method **returnBook** marks a book as returned.

```
package de.laliluna.library.struts.action;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionMessage;
import org.apache.struts.action.ActionMessages;
import org.apache.struts.actions.DispatchAction;

import de.laliluna.library.*;


public class ManageCustomerAction extends DispatchAction {

 // ------------------------------------------------------- Instance Variables

 // ----------------------------------------------------------- Methods

 /**
  * Method execute
  * @param mapping
  * @param form
  * @param request
  * @param response
  * @return ActionForward
  */
 public ActionForward unspecified(
   ActionMapping mapping,
   ActionForm form,
   HttpServletRequest request,
   HttpServletResponse response) {

   BookDao bookDao;
   try
   {
     Context context = new InitialContext();
```

```
    bookDao = (BookDao) context.lookup(BookDaoImp.LocalJNDIName);
   } catch (NamingException e)
   {
    e.printStackTrace();
    throw new RuntimeException(e);
   }
   Customer customer = (Customer)
request.getSession().getAttribute(Common.CUSTOMER_KEY);
   if (customer != null)
    request.setAttribute("books", bookDao.findByCustomer(customer));
   return mapping.findForward(Common.SUCCESS);
 }

 public ActionForward returnBook(
    ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response) {

   ActionMessages actionMessages = new ActionMessages();
   HttpSession session = request.getSession();
   // first try to get an existing cart from the current session
   LibraryCartLocal cartLocal = (LibraryCartLocal) session
       .getAttribute("cart");

   // when there is no cart create a new one and save it in the session
   if (cartLocal == null)
   {
    try
    {
     Context context = new InitialContext();
     cartLocal = (LibraryCartLocal) context
         .lookup(LibraryCart.LocalJNDIName);

     session.setAttribute("cart", cartLocal);
    } catch (NamingException e)
    {
     e.printStackTrace();
     throw new RuntimeException(e);
    }
   }
   String sId = request.getParameter("id");
   if (sId == null)
   {
    actionMessages.add(ActionMessages.GLOBAL_MESSAGE,
       new ActionMessage("error.param.noentry"));
   } else
   {
    Integer id = Integer.parseInt(sId);
    cartLocal.returnBook(id);
   }
   if (!actionMessages.isEmpty())
    saveErrors(request, actionMessages);
   return mapping.findForward(Common.SUCCESS);
   }
}
```

Create a JSP **manageCustomer.jsp** in the folder **JSP.** This JSP displays the customer name and all the books he has borrowed.

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-tiles" prefix="tiles" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%String path = request.getContextPath();
    String basePath = request.getScheme() + "://"
```

```
        + request.getServerName() + ":" + request.getServerPort()
        + path + "/";
%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <base href="<%=basePath%>">
    <title>Manage customer</title>
  </head>

  <body>
<jsp:include page="/jsp/navigation.jsp"></jsp:include>
  <c:if test="${sessionScope.customer_key == null}">
      <h3>Warning! No customer selected.</h3>
    </c:if>
    <c:if test="${sessionScope.customer_key != null}">
      Selected customer: <c:out value="${sessionScope.customer_key.name}"></c:out>
    </c:if>
    <br>
      <TABLE border="1">
        <tr>
          <td>
            id
          </td>
          <td>
            author
          </td>
          <td>
            title
          </td>
        <td></td>
        </tr>
        <c:forEach items="${requestScope.books}" var="book">
        <tr>
          <td>
            ${book.id}
          </td>
          <td>
            ${book.author}
          </td>
          <td>
            ${book.title}
          </td>
          <td>
          <c:url value="/manageCustomer.do" var="manageCustomerUrl">
          <c:param name="action">returnBook</c:param>
          <c:param name="id">${book.id}</c:param>
          </c:url>
          <A href="${manageCustomerUrl}">Return book</A>
          </td>
        </tr>
        </c:forEach>
      </TABLE>
```

```
  </body>
</html>
```

Use case: Select a customer

This use case will list all customers and allow the library employee to select one. The selected customer will be saved in the session.

Create a new action using the MyEclipse wizard. Select **DispatchAction** as type and action as parameter. Then create two forwards **success** and **self**.

Self will be called when the customers have to be displayed. **Success** forwards to the manageCustomer action once a customer is selected. After using the wizard you should find the following action in your **structs-config.xml**

```
    <action
     parameter="action"
      path="/selectCustomer"
      type="de.laliluna.library.struts.action.SelectCustomerAction"
      validate="false">
      <forward
      name="success"
      path="/manageCustomer.do"
      redirect="true" />
      <forward name="self" path="/jsp/selectCustomer.jsp" />
    </action>
```

The parameter will distinguish which method in the action is called. When it is called with no parameter the method **unspecified** is called. This method will load all customers using the CustomerDao and display them. When the action is called with the parameter **selectCustomer** then the method selectCustomer is called by Struts. The link could be:

http://localhost:8080/EJBStrutsWeb/selectCustomer.do?action=selectCustomer&id=1

Here is the complete class.

```
package de.laliluna.library.struts.action;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionMessage;
import org.apache.struts.action.ActionMessages;
import org.apache.struts.actions.DispatchAction;
import de.laliluna.library.*;


public class SelectCustomerAction extends DispatchAction {

 // --------------------------------------------------------- Instance
 // Variables

 // ----------------------------------------------------------- Methods

 /**
 * Method execute
```

```
 *
 * @param mapping
 * @param form
 * @param request
 * @param response
 * @return ActionForward
 */
public ActionForward unspecified(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response) {
  CustomerDao customerDao;
  try
  {
    Context context = new InitialContext();
    customerDao = (CustomerDao) context
        .lookup(CustomerDaoImp.LocalJNDIName);
  } catch (NamingException e)
  {
    e.printStackTrace();
    throw new RuntimeException(e);
  }
  request.setAttribute("customers", customerDao.findAll());
  return mapping.findForward(Common.SELF);
}

/**
 * Method execute
 *
 * @param mapping
 * @param form
 * @param request
 * @param response
 * @return ActionForward
 */
public ActionForward selectCustomer(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response) {
  CustomerDao customerDao;
  ActionMessages messages = new ActionMessages();
  try
  {
    Context context = new InitialContext();
    customerDao = (CustomerDao) context
        .lookup(CustomerDaoImp.LocalJNDIName);
  } catch (NamingException e)
  {
    e.printStackTrace();
    throw new RuntimeException(e);
  }
  // look for the customer in the db for the given id
  String sId = request.getParameter("id");
  if (sId == null)
  {
    messages.add(ActionMessages.GLOBAL_MESSAGE, new ActionMessage(
        "error.param.noentry"));

  }
  Customer customer = customerDao.findById(Integer.parseInt(sId));
  if (customer == null)
    messages.add(ActionMessages.GLOBAL_MESSAGE, new ActionMessage(
    "error.param.noentry"));

  // when we found a customer, save it in the session, else go back to the customer
select
  if (messages.isEmpty()){
    request.getSession().setAttribute(Common.CUSTOMER_KEY, customer);

    return mapping.findForward(Common.SUCCESS);}
  else
  {
    request.setAttribute("customers", customerDao.findAll());
    saveErrors(request, messages);
    return mapping.findForward(Common.SELF);
```

```
    }
  }
}
```

Create a JSP named selectCustomer.jsp in the folder JSP and add the following content. JSTL is used to display a list of customers and to create a link for each row. When you click on this link the selectCustomer action is called with the id of the customer as parameter.

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-tiles" prefix="tiles" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%String path = request.getContextPath();
    String basePath = request.getScheme() + "://"
        + request.getServerName() + ":" + request.getServerPort()
        + path + "/";
%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <base href="<%=basePath%>">
    <title>Select customer</title>

  </head>

  <body>
<jsp:include page="/jsp/navigation.jsp"></jsp:include>
      <TABLE border="1">
        <tr>
          <td>
            id
          </td>
          <td>
            author
          </td>
          <td>
            title
          </td>

        </tr>
        <c:forEach items="${requestScope.customers}" var="customer">
        <tr>
          <td>
            ${customer.id}
          </td>
          <td>
            ${customer.name}
          </td>
          <td>
          <c:url value="/selectCustomer.do" var="selectCustomerUrl">
          <c:param name="action">selectCustomer</c:param>
```

```
          <c:param name="id">${customer.id}</c:param>
          </c:url>
          <A href="${selectCustomerUrl}">Select Customer</A>
          </td>
        </tr>
        </c:forEach>
      </TABLE>
  </body>
</html>
```

Update your navigation.jsp

```
<%@ page language="java" pageEncoding="UTF-8"%>

<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-tiles" prefix="tiles" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<table> <TR>
<TD>
<html:link action="selectCustomer">Select a customer</html:link>
</TD>
<TD>
<html:link action="manageCustomer">Manage customer</html:link>
</TD>
<TD>
<html:link action="createSampleData">Create Sample Data</html:link>
</TD>


</TR>
</table>
```

You can test your application now, list customers and select them.

Use case: logout

This use case cleans up the session, so that you can restart from the beginning during testing. It is fairly simple action invalidating the session and forwarding to the welcome page.

```
    <action
      path="/logout"
      type="de.laliluna.library.struts.action.LogoutAction"
      validate="false">
      <forward name="success" path="/jsp/welcome.jsp" />
    </action>
```

```
package de.laliluna.library.struts.action;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
```

```
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

import de.laliluna.library.*;


public class LogoutAction extends Action {

 /**
  * Method execute
  * @param mapping
  * @param form
  * @param request
  * @param response
  * @return ActionForward
  */
 public ActionForward execute(
   ActionMapping mapping,
   ActionForm form,
   HttpServletRequest request,
   HttpServletResponse response) {

   HttpSession session = request.getSession();
   session.invalidate();
   return mapping.findForward(Common.SUCCESS);
 }

}
```

Use case: Manage Cart

This use case displays our current shopping cart and customer and allows to checkout, i.e. Marks all books as borrowed and removes the cart.

Create a new **DispatchAction**. The **parameter** is **action** one again. Add two forwards **self** and **success**.

```
    <action
      parameter="action"
      path="/manageCart"
      type="de.laliluna.library.struts.action.ManageCartAction"
      validate="false">
      <forward name="success" path="/jsp/checkout.jsp" />
      <forward name="self" path="/jsp/cart.jsp" />
    </action>
```

Our dispatch action has four methods:

- unspecified, which just forwards to the JSP showing the cart content.
- addToCart, which will add a book to the cart
- removeFromCart, which removes a book from the cart
- checkOut, which does mark the book as borrowed, which is equivalent to saving a relation between customer and book. Finally this method destroys the cart.

```
package de.laliluna.library.struts.action;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
```

```
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionMessage;
import org.apache.struts.action.ActionMessages;
import org.apache.struts.actions.DispatchAction;

import de.laliluna.library.*;


public class ManageCartAction extends DispatchAction {


 public ActionForward addToCart(ActionMapping mapping, ActionForm form,
     HttpServletRequest request, HttpServletResponse response) {
   HttpSession session = request.getSession();
   ActionMessages actionMessages = new ActionMessages();
   BookDao bookDao;
   // first try to get an existing cart from the current session
   LibraryCartLocal cartLocal = (LibraryCartLocal) session
       .getAttribute("cart");

   // when there is no cart create a new one and save it in the session
   if (cartLocal == null)
   {
     try
     {
       Context context = new InitialContext();
       cartLocal = (LibraryCartLocal) context
           .lookup(LibraryCart.LocalJNDIName);

       session.setAttribute("cart", cartLocal);
     } catch (NamingException e)
     {
       e.printStackTrace();
       throw new RuntimeException(e);
     }
   }
   String sId = request.getParameter("id");
   if (sId == null)
   {
     actionMessages.add(ActionMessages.GLOBAL_MESSAGE,
         new ActionMessage("error.param.noentry"));
   } else
   {
     Integer id = Integer.parseInt(sId);
     cartLocal.addBook(id);
   }
   if (!actionMessages.isEmpty())
     saveErrors(request, actionMessages);
   return mapping.findForward(Common.SELF);
 }

 public ActionForward removeFromCart(ActionMapping mapping, ActionForm form,
     HttpServletRequest request, HttpServletResponse response) {
   HttpSession session = request.getSession();
   ActionMessages actionMessages = new ActionMessages();

   // first try to get an existing cart from the current session
   LibraryCartLocal cartLocal = (LibraryCartLocal) session
       .getAttribute(Common.CART_KEY);

   // when there is no cart there is nothing to remove so no further actions
   if (cartLocal == null)
   {
     return mapping.findForward(Common.SELF);
   }
   // read the id and try to remove the book for this id from the cart
   String sId = request.getParameter("id");
   if (sId == null)
```

```
  {
    actionMessages.add(ActionMessages.GLOBAL_MESSAGE,
        new ActionMessage("error.param.noentry"));
  } else
  {
    Integer id = Integer.parseInt(sId);
    cartLocal.removeBook(id);
  }
  if (!actionMessages.isEmpty())
    saveErrors(request, actionMessages);
  return mapping.findForward(Common.SELF);
}

@Override
public ActionForward unspecified(ActionMapping mapping, ActionForm form,
HttpServletRequest request, HttpServletResponse response) throws Exception {
  return mapping.findForward(Common.SELF);
}

public ActionForward checkOut(ActionMapping mapping, ActionForm form,
HttpServletRequest request, HttpServletResponse response) throws Exception {
  HttpSession session = request.getSession();
  ActionMessages actionMessages = new ActionMessages();

  // first try to get an existing cart from the current session
  LibraryCartLocal cartLocal = (LibraryCartLocal) session
      .getAttribute(Common.CART_KEY);
  Customer customer = (Customer) session.getAttribute(Common.CUSTOMER_KEY);
  if (cartLocal == null || customer == null){
    actionMessages.add(ActionMessages.GLOBAL_MESSAGE, new
ActionMessage("error.checkout.missing_cart_customer"));
    saveErrors(request,actionMessages);
    return mapping.findForward(Common.SELF);
  }
  else{
    cartLocal.checkOut(customer);
    // checkOut is configured to delete the stateful bean, so me must delete the
reference to it as well
    session.removeAttribute(Common.CART_KEY);
  return mapping.findForward(Common.SUCCESS);
  }
}

}
```

We need two JSPs. The first one displays the cart and the second one outputs a success message when we have checkout successfully.

Create a JSP named **cart.jsp** in the **jsp** folder.

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean"%>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html"%>
<%@ taglib uri="http://struts.apache.org/tags-tiles" prefix="tiles"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>Cart</title>
  </head>
  <body>
    <logic:messagesPresent>
      <html:messages id="message">
        <bean:write name="message" />
        <br>
```

```
      </html:messages>
      <br>
    </logic:messagesPresent>
    <jsp:include page="/jsp/navigation.jsp"></jsp:include>
    <br>
    <c:url var="checkOutUrl" value="manageCart.do">
      <c:param name="action">checkOut</c:param>
    </c:url>
    <a href="${checkOutUrl}">Checkout this cart</a>
    <br>
    <c:if test="${sessionScope.customer_key == null}">
      <h3>Warning! No customer selected you cannot checkout</h3>
    </c:if>
    <c:if test="${sessionScope.customer_key != null}">
      Selected customer: <c:out value="${sessionScope.customer_key.name}"></c:out>
    </c:if>
    <br>
    <TABLE border="1">
      <tr>
        <td>
          id
        </td>
        <td>
          author
        </td>
        <td>
          title
        </td>

      </tr>
      <c:forEach items="${sessionScope.cart.cart}" var="book">
        <tr>
          <td>
            ${book.value.id}
          </td>
          <td>
            ${book.value.author}
          </td>
          <td>
            ${book.value.title}
          </td>
          <td>
            <c:url value="/manageCart.do" var="removeFromCartUrl">
              <c:param name="action">removeFromCart</c:param>
              <c:param name="id">${book.value.id}</c:param>
            </c:url>
            <A href="${removeFromCartUrl}">Remove from Cart</A>
          </td>
        </tr>
      </c:forEach>
    </TABLE>
  </body>
</html>
```

Create a JSP named **checkout.jsp** in the **jsp** folder.

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <base href="<%=basePath%>">
    <title>Cart</title>
  </head>

  <body>
    <jsp:include page="/jsp/navigation.jsp"></jsp:include>
      <h2>Checkout of the cart was successful</2>
  </body>
</html>
```

Use case: list books

This use case displays all available books, allows to add a book to the cart and shows which book is borrowed by which customer. Create a simple **action** class with a **forward** named **success** to the **listBook.jsp**.

```
<action
   path="/listBooks"
   type="de.laliluna.library.struts.action.ListBooksAction"
   validate="false">
   <forward name="success" path="/jsp/listBooks.jsp" />
</action>
```

Create the action class.

```
package de.laliluna.library.struts.action;

import java.util.List;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import de.laliluna.library.Book;
import de.laliluna.library.BookDao;
import de.laliluna.library.BookDaoImp;
import de.laliluna.library.Common;
import de.laliluna.library.struts.CustomAction;

public class ListBooksAction extends Action {

 /**
  * Method execute
  *
  * @param mapping
  * @param form
  * @param request
  * @param response
  * @return ActionForward
  */
 public ActionForward execute(ActionMapping mapping, ActionForm form,
     HttpServletRequest request, HttpServletResponse response) {
```

```
  BookDao bookDao = null;
  List<Book> books = null;
  try
  {
    Context context = new InitialContext();
    bookDao = (BookDao) context.lookup(BookDaoImp.LocalJNDIName);
  } catch (NamingException e)
  {
    e.printStackTrace();
    throw new RuntimeException("error accessing EJB");

  }
  books = bookDao.findAll();
  request.setAttribute("books", books);
  return mapping.findForward(Common.SUCCESS);
 }
}
```

Finally, create the JSP **listBooks.jsp** in the **jsp** folder.

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-tiles" prefix="tiles" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>My JSP 'selectBooks.jsp' starting page</title>
  </head>
  <body>
<jsp:include page="/jsp/navigation.jsp"></jsp:include>
      <TABLE border="1">
        <tr>
          <td>
            id
          </td>
          <td>
            author
          </td>
          <td>
            title
          </td>
          <td></td>
          <td>
           Borrowed to
           </td>
        </tr>
        <c:forEach items="${requestScope.books}" var="book">
        <tr>
          <td>
            ${book.id}
          </td>
          <td>
            ${book.author}
          </td>
          <td>
            ${book.title}
          </td>
```

```
          <td>
          <c:if test="${book.customer == null}">
          <c:url value="/manageCart.do" var="addToCartUrl">
          <c:param name="action">addToCart</c:param>
          <c:param name="id">${book.id}</c:param>
          </c:url>
          <A href="${addToCartUrl}">Add to Cart</A>
          </c:if>
          </td>
          <TD>
           <c:if test="${book.customer != null}"> ${book.customer.name}
           </c:if>
          </TD>
        </tr>
        </c:forEach>
      </TABLE>
  </body>
</html>
```

Update your navigation.jsp

```
<%@ page language="java" pageEncoding="UTF-8"%>

<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-tiles" prefix="tiles" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<table> <TR>
<TD>
<html:link action="selectCustomer">Select a customer</html:link>
</TD>
<TD>
<html:link action="manageCustomer">Manage customer</html:link>
</TD>
<TD>
<html:link action="listBooks">List Books</html:link>
</TD>
<TD>
<html:link action="createSampleData">Create Sample Data</html:link>
</TD>
<TD>
<html:link action="manageCart">Show Cart</html:link>
</TD>
<TD>
<html:link action="logout">Log out</html:link>
</TD>
</TR>
</table>
```

Congratulations, you have finished a complete EJB Struts application.

Copyright and disclaimer

The tutorial is provided as is. I do not give any warranty or guaranty any fitness for a particular purpose. In no event shall I be liable to any party for direct, indirect, special, incidental, or consequential damages, including lost profits, arising out of the use of this tutorial, even if I has been advised of the possibility of such damage.