Beginning with the Java EE 6 Platform
Hands-On Lab – 23421

Alexis Moussine-Pouchkine
Oracle Corporation

Pierre-Henri Dezanneau
SERLI

Table of Contents

Introduction

Back in 2006, Java EE 5 had really three themes: Ease of Development, Ease of Development and Ease of development! The new Java EE Platform version 6 builds on these major improvements and offers substantial enhancements to existing technologies (new features, simplifications) but also adds some new capabilities such as BeanValidation, JAX-RS (RESTful Web Services), and the new and exciting Context and Dependency Injection (CDI) specification.

This hands-on lab (HOL) is a stripped-down version of the material available at http://beginningee6.kenai.com which was developed as a companion set of samples to the "Beginning Java EE 6 with GlassFish 3, second Edition" book by Antonio Goncalves. This website contains more than a dozen examples structured as building blocks, starting with a very basic implementation and scaling to a fully-featured web application covering the new Java EE 6 features. The website also provides everything you should need to explore further the platform technologies – code, instructions, a forum, and a mailing list.

To be honest, this lab does not do justice to Java EE 6, but will try to give you a taste of how simple yet powerful the platform has become. It will cover mainly JavaServer Faces 2.0, JAX-RS 1.1, and CDI 1.0 with references made to ManagedBeans, EJB 3.1, JPA 2.0, Servlet 3.0, and more. The "Summarizing Previous Episodes..." section has a paragraph or two on each of them.

If you are in a hurry, then go straight to Exercise 1 (JSF) and start coding. Otherwise read on the following setup and code discussion sections.

Setup Requirements

As it is the case for the set of “Beginning Java EE 6” projects, the entire source code for this Hands-On Lab is available in the form of **Maven projects** to ease the portability across several development tools and environments. So there is no requirement for the development environment other than Maven itself. For instance, NetBeans 7.0.1 and above can simply open Maven projects as-is without adding any metadata. If fact, in the case of NetBeans, there even no need to install Maven as it ships with an embedded version of the softeware. The source code can be checked-out from the following URL :
http://kenai.com/projects/beginningee6/sources/src/show/HandsOnLab/tags/JavaOne2011_SF

With the increased focus on Java code (vs. XML deployment descriptors) and the general CoC theme (Convention over Configuration), support for Java EE 6 APIs and wizards in the development environment is a plus but not an absolute requirement. As such you can probably follow the entire HOL with a simple code editor. Having said this, **NetBeans 7.0.1** which is loaded on your environment is probably the best out-of-the-box experience for learning Java EE 6. Eclipse 3.7 (indigo), the Oracle Enterprise Pack for Eclipse (OEPE, which contains the GlassFish 3.x plugin), or IntelliJ are all good options.

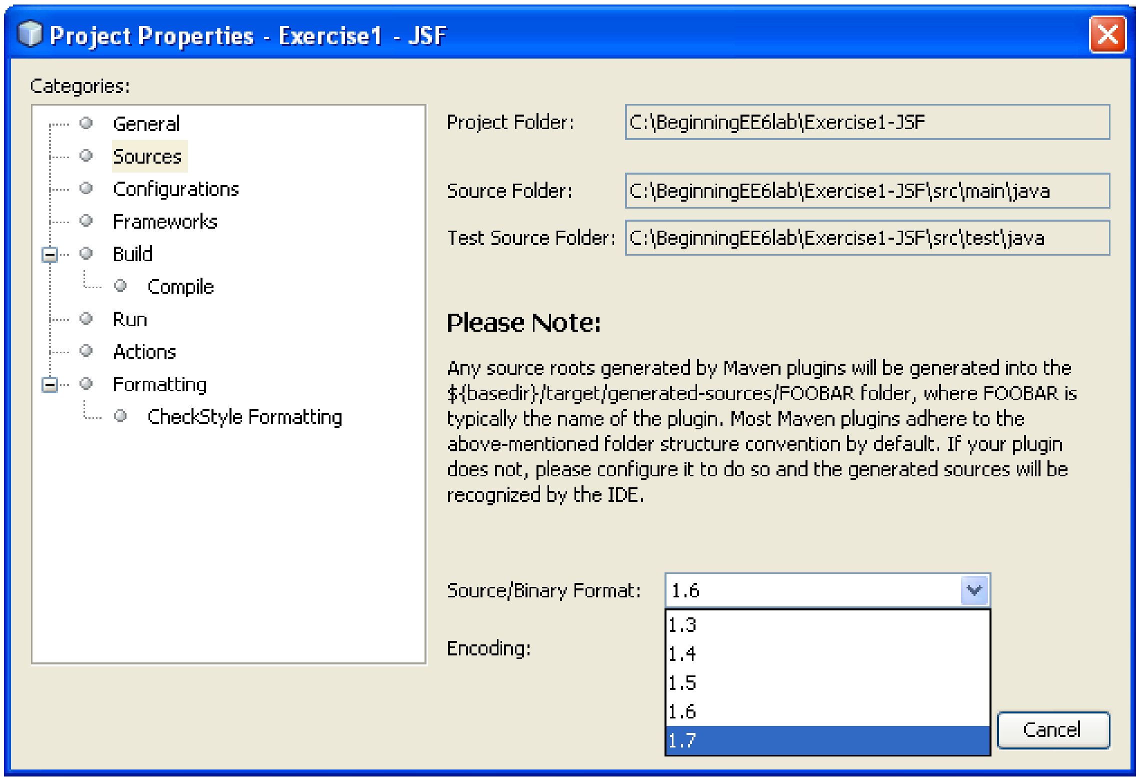
The target runtime (AppServer) for this lab is **GlassFish 3.1.1** as this is both the reference implementation for the Java EE platform as well as a production-quality open source application server focusing on developer experience and enterprise features. The code should also work unmodified on other Java EE 6 containers. This HOL requires a Java EE 6 ***Web Profile*** container and as such you'll notice how the Maven projects depend on a single platform artifact called `javax:javaee-web-api:jar:6.0` (version 6 of the JavaEE Web Profile) rather than expressing a set of dependencies on individual artifacts for the JSF, JPA, Web Container, EJB Container and other runtimes.

In this lab, the top level project define **three modules** (exercise1, exercise2, and exercise3), some properties for the runtime to be used, as well as the java.net Maven repository for the artifact and plugin dependencies. You are welcome to study these settings. A `Solution/` sub-directory also offers all three exercises completed.

NetBeans 7.0.1 is available from http://netbeans.org/downloads. The “Java” or “Full” downloads will ship with everything you need, including the GlassFish application server. A standalone version of GlassFish 3.1.1 is available from :
http://glassfish.org/downloads/3.1.1-final.html

Both the IDE and the server have already been installed on your systems. A browser such as **Firefox** and a command-line HTTP client (**wget**, **cURL**) are also installed.

We will be using Java 7 throughout this lab. It is used for running both the NetBeans IDE itself and the GlassFish application server. However the source code level for the various exercises has not been set to Java 7 to offer greater portability. If you are interested in taking advantage of the new features offered in Java 7 (such as the new “Project Coin” language features) you are welcome to change the source code level and benefit from the NetBeans IDE tips to convert existing code to the new language constructs.



Project Folder: C:\BeginningEE6lab\Exercise1-JSF

Source Folder: C:\BeginningEE6lab\Exercise1-JSF\src\main\java

Test Source Folder: C:\BeginningEE6lab\Exercise1-JSF\src\test\java

Please Note:

Any source roots generated by Maven plugins will be generated into the ${basedir}/target/generated-sources/FOOBAR folder, where FOOBAR is typically the name of the plugin. Most Maven plugins adhere to the above-mentioned folder structure convention by default. If your plugin does not, please configure it to do so and the generated sources will be recognized by the IDE.

Source/Binary Format: 1.6

Encoding:

Finally note that the projects are configured with *deploy-on-save* mode, a feature offered by the GlassFish plugin in NetBeans offering a save/reload paradigm. The moment you save an artifact (java source file, XHTML jsf page, …), the change is incrementally deployed to the server and you only need to switch to the browser and reload the page. No explicit compile or deploy/redeployed is required.

Summarizing Previous Episodes...

This is a summary of how we got to exercise #1 of this HOL starting from a couple of classes, no user-interface and growing from there. You are encouraged to read through this short introduction and time-permitting to study the Java code mentioned in each of the following paragraphs.

ManagedBeans 1.0

The application developed here manages a set of books. It uses a new component type – a ***managed bean***. This is a new lightweight component defined in Java EE 6 which you can think of as a the foundation for other components (EJB, Servlet, …) which offer additional services (transactions, security, HTTP, ...). In our case the component is annotated with `@javax.annotation.ManagedBean` and generates ISBN numbers with an interceptor logging calls to this service. Check out `IsbnGenerator.java` and `LoggingInterceptor.java`.

JPA 2.0

Books are JPA entities which are stored in a database with attributes such as ISBN (string), title (string), price (float), description (string), pages (integer), illustration (boolean), and a set of tags (a **collection of strings** mapped to a table using a new JPA 2.0 feature). Check out `Book.java`.

As always, `META-INF\persistence.xml` is where the JPA persistence unit (PU) is defined. We use EclipseLink (the JPA 2.0 reference implementation) and explicitly list the entities. The properties should be self-explanatory.

Servlet 3.0

The initial implementation takes user input from an HTML form and passes it to a servlet 3.0 (now a POJO using the **`@WebServlet`** annotation including the mapping information and removing altogether the need for the `web.xml` deployment descriptor). This servlet is responsible for interacting with the JPA entity manager.

EJB 3.1

In order to offer better separation of concerns, an EJB was introduced. The servlet retains its web front-end responsibility of gathering the user input while the transactional EJB component offers a clean persistence interaction removing the boiler-plate transactional code the servlet had to deal with. Check out `ItemEJB.java` and see how this EJB is packaged as part of a web application (straight into the WAR, a new feature of Java EE 6).

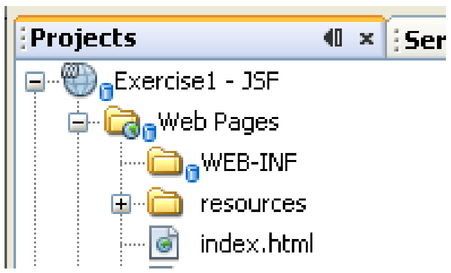
Another EJB is introduced as a **singleton** which is instantiated when the container starts up (yet another new EJB 3.1 features) to offer a language code resolver. Check out `LanguageSingleton.java`.

Both these EJBs can now be easily tested from a Java SE environment (JUnit, Maven, ...) using the new and standard **`EJBContainer`** API.

Exercise 1 (JSF)

With the basic features in place, it's time to put a nice face on this application with JavaServer Faces (JSF).

Since we use JSF, we no longer need to write our own servlet code (the JSF framework uses a servlet under the covers), so to begin with, study the code for **`ItemBean.java`**. This is a JSF managed bean (sometimes also called a backing bean) defined as a POJO and annotated with `@javax.faces.bean.ManagedBean` and `@javax.faces.bean.RequestScoped`. In earlier version of JSF, this metadata (bean definition and scoping) was defined in the `faces-config.xml` deployment descriptor which has now become optional. This makes for an empty `WEB-INF` directory with no `web.xml` either.



The JSF managed bean uses `@EJB` injection to obtain references to the `ItemEJB` and `LanguageSingleton` bean implementations.

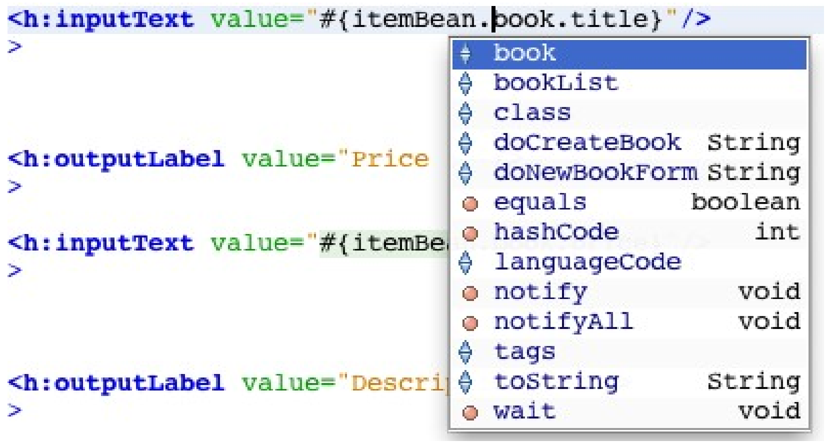
The `initList()` method (an arbitrary name) is annotated with `@PostConstruct` which can be applied to any method in the managed bean and where the list of books is populated with the help of the `ItemEJB` abstraction once the component is created by the container.

The `doCreateBook()` and `doNewBookForm()` methods will be referenced from buttons in the JSF view which we'll soon create using Facelets. These methods `return` statement indicate which page to go next (in our case we're staying on the current page). This is a new feature called implicit navigation which no longer requires expressing navigation rules in the `faces-config.xml` deployment descriptor.

Part of the new features in JSF 2.0 is the use by default of Facelets, a framework used by many in previous versions of JSF which has now become part of the standard. Facelets entirely express views (pages) with XHTML (well-formed HTML) while binding UI components such as text fields to the managed bean attributes and controls such as buttons and links to the aforementioned `doXYZ()` methods. Such binding is done using the Expression Language (EL) using the `#{...}` notation.

IDE Code Completion

`newBook.xhtml` (in the NetBeans "Web Pages" section of the project) contains a web form to add books as well as a table to list all the books that have been previously added. Using NetBeans, navigate to this file and place the cursor in the expression on line #19, and invoke code completion (`Shift-Ctrl-Space` on most systems) to see how the IDE can help you list existing attributes. The following two screenshots illustrate this :

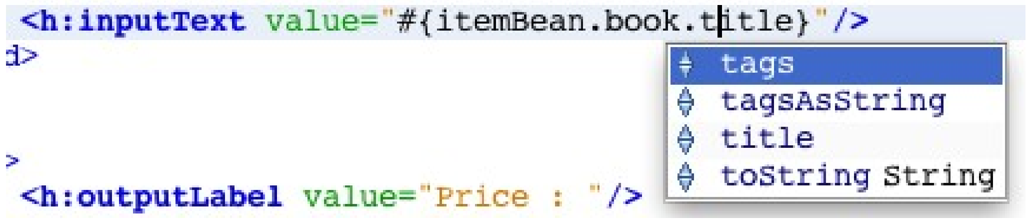


```
<h:inputText value="#{itemBean.book.title}"/>
>

<h:outputLabel value="Price
>
<h:inputText value="#{itemBe
>

<h:outputLabel value="Descri
>
```

```
book
bookList
class
doCreateBook     String
doNewBookForm    String
equals          boolean
hashCode            int
languageCode
notify             void
notifyAll          void
tags
toString         String
wait               void
```



```
<h:inputText value="#{itemBean.book.title}"/>
d>

>
<h:outputLabel value="Price : "/>
```

```
tags
tagsAsString
title
toString String
```

Also try Ctrl-clicking on `itemBean`, as used in the expression language and see how the IDE takes you to the definition of the JSF managed bean, `ItemBean.java` in our case.

Wire UI control to code behavior

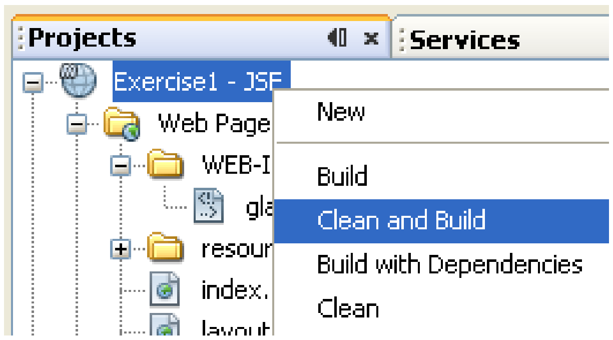
At the end of the book form (before the list of existing books) on line #35, replace the comment with a command button and use the Expression Language (EL) to associate it with the `doCreateBook()` method. Towards the middle of the XHTML source, the code should read (don't forget to save the file) :

```
   </h:panelGrid>

   <h:commandButton value="Create a book" action="#{itemBean.doCreateBook}"/>

</h:form>
```

Clean and build the project (right-click "Clean and Build" on the NetBeans project or `mvn clean install` from the command line).

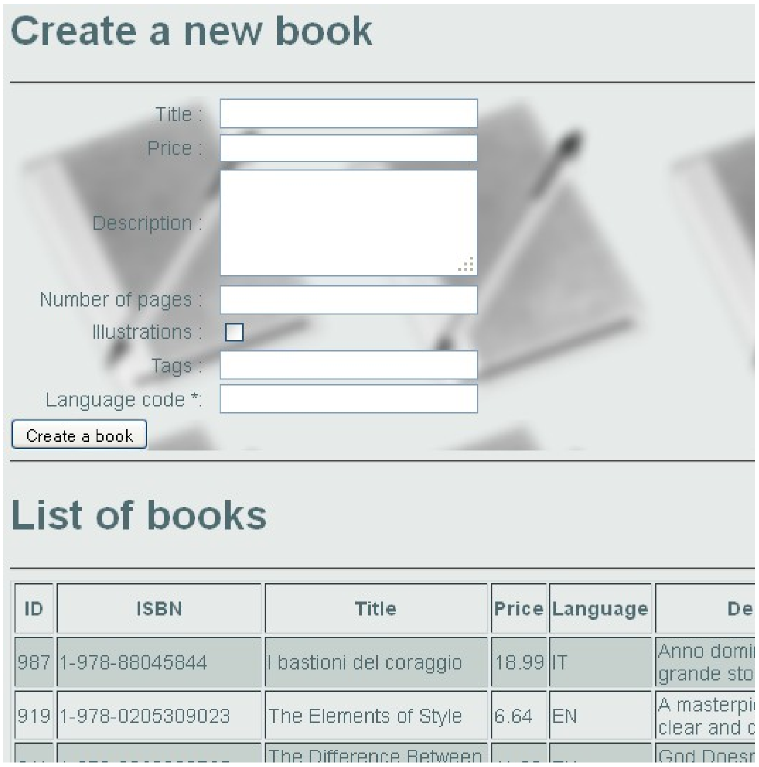


Make sure the database is started (check the troubleshooting section for how to do that if it doesn't start automatically).

If asked about which server to deploy to, you can chose to have the IDE remember the setting for this particular project which will add it a Maven *profile*.

Run the project (right-click "Run" on the NetBeans project). This will cause the following to happen :

- the GlassFish application server will be started (if not already)
- the web application (packaged as a WAR file) will be deployed to GlassFish
- a browser will be opened at the application URL: `http://localhost:8080/exercise1/newBook.faces`



Create a new book

Title :	
Price :	
Description :	
Number of pages :	
Illustrations :	☐
Tags :	
Language code *:	

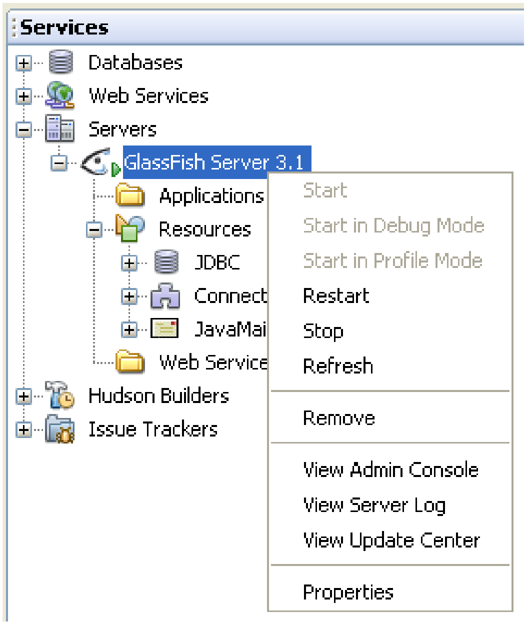
Create a book

List of books

ID	ISBN	Title	Price	Language	De
987	1-978-88045844	I bastioni del coraggio	18.99	IT	Anno domi grande sto
919	1-978-0205309023	The Elements of Style	6.64	EN	A masterpi clear and c
		The Difference Between			God Doesn

Exercise the application by adding a few books. These will be kept around for the rest of the lab.

You can control the web context URL using the runtime project properties. Right from NetBeans, GlassFish can be started or stopped, applications undeployed, etc, from the "Servers" section of the "Services" tab :



Use the "View Server Log" option to observe application logging messages as well as for troubleshooting information.
The raw log file is located in :

`%GLASSFISH_HOME%\glassfish\domains\domain1\logs\server.log`.

Other improvements in JSF 2.0 include the introduction of composite components (you can now write custom JSF components in a single XHTML file), new templating features and the full integration of Ajax in the form of a standardized javascript file and a `<f:ajax>` tag.

The next exercise has both composite components and templating features integrated, so check out the source code if you'd like to know more.

If you'd like to understand how Ajax can be used to implement a textfield to filter the content for the table of books, simply open C:\BeginningEE6Lab\Solution\Exercise1-JSF and study `newBook.xhtml`. Line #42 uses the `<f:ajax>` tag to implement a partial page refresh (the list of books) on every key press in the “`Filter`” textfield.

Exercise 2 (JAX-RS)

Before we get on to the RESTful Web Service part of this HOL, a few words about the starting point for this second exercise.

You are encouraged to study the following source code (if you haven't already done so, close the "`Exercise1-JSF`" project and open the project called "`Exercise2-JAX-RS`" and located in `$LAB_HOME\JavaEE6_Lab`) :

- `Book.java` now inherits from a base class defined in `Item.java`
- Under "Web Pages", `resources/demo/newItem.xhtml`, is a JSF composite component introduced to manage both books and CDs (see the new `newCD.xhtml` page and the updated `newBook.xhtml`).
- `layout.xhtml` illustrates how the new templating features of JSF 2.0 can be used.
- `Item.java` and `ItemBean.java` both contain BeanValidation (JSR 303, also a neat addition to Java EE 6) annotations constraining the values of the title and the language code (with a default hard-coded error message).

Note that `Exercise2-JAX-RS` will fail to deploy as-is with a "`SEVERE: The ResourceConfig instance does not contain any root resource classes`" error. Make sure you read the following paragraph to expose RESTful resources before you try to run the project.

Turn the EJB into a REST service

JAX-RS is a simple and elegant yet powerful technology to develop RESTful Web Services. The simplest form is a POJO decorated with an `@Path` annotation. In our case we will leverage one of the new features of JAX-RS 1.1 which is to combine the EJB and JAX-RS functionalities (version 1.0 of JAX-RS had been released earlier but was not yet part of the Java EE platform).

Expose a Java object as a RESTful resource

In `ItemEJB.java`, add the following `@Path` annotation to promote the class to a JAX-RS resource (and adjust your imports with `Ctrl+Shift-I` to include `javax.ws.rs.Path`) :

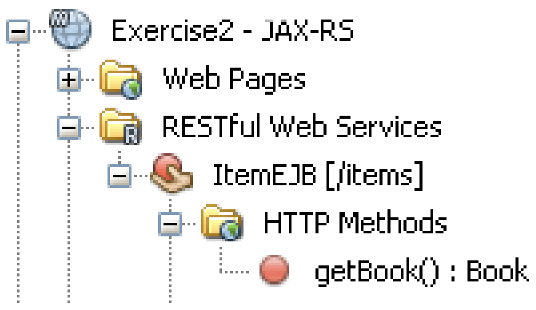
```
@Stateless
@Interceptors(LoggingInterceptor.class)
@Path("/items") // resource available from the /items path
public class ItemEJB {
```

and modify its `getBook()` method to use the following JAX-RS annotations (make sure the imports use classes prefixed with `javax.ws.rs`) :

```
@GET // HTTP's GET verb/operation
@Path("{bookKey}") // specializes the path with a parameter
@Produces(MediaType.APPLICATION_XML) // mime-type
public Book getBook(@PathParam("bookKey") Long bookKey) {
```

The bookKey variable will be initialized with the parameter extracted from the requested URL (**123** for instance with URL `http://localhost:8080/.../items/`**123**).

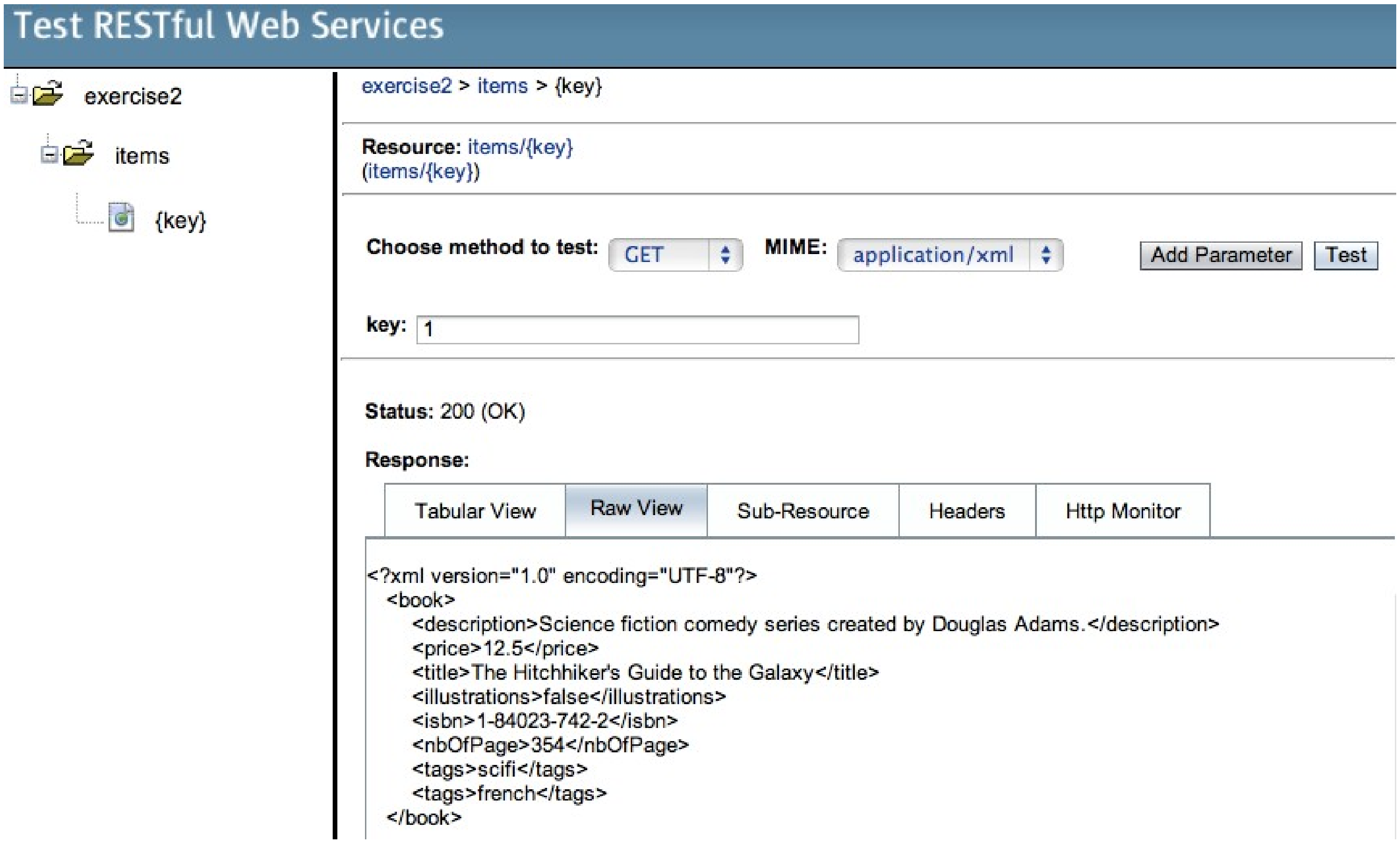
The above changes will trigger a new “RESTful Web Services” node in the NetBeans project to appear :



Finally, place an **`@XmlRootElement` annotation** on the Book class.

Now let's try this all out and **run the project** to cause a redeployment. Once the page appears in the browser, memorize any book ID and type the following address in the browser window `http://localhost:8080/exercise2/resources/items/`***`id`*** and observe the XML-formatted response. The `/resources` path is defined in the Jersey servlet mapping definition in `web.xml`. We'll see later how to go without this `web.xml` deployment descriptor.

Note that browsers such as safari or chrome will show unformatted XML. Use "view HTML source" to see the formatted version. As an alternative, you can use NetBeans' "Test RESTful Web Services" feature (right-click on the "RESTful Web Services" node) :



```
<?xml version="1.0" encoding="UTF-8"?>
  <book>
     <description>Science fiction comedy series created by Douglas Adams.</description>
     <price>12.5</price>
     <title>The Hitchhiker's Guide to the Galaxy</title>
     <illustrations>false</illustrations>
     <isbn>1-84023-742-2</isbn>
     <nbOfPage>354</nbOfPage>
     <tags>scifi</tags>
     <tags>french</tags>
  </book>
```

For the time being, trying out other MIME types will all produce XML. Also note that GET is the only method implemented in this lab (others will cause HTTP 405 – Method not allowed errors).

Support multiple media types and use a command-line client

Using a browser to access a RESTful resource is of limited use, so we'll move to using a command-line HTTP client to illustrate further JAX-RS features.

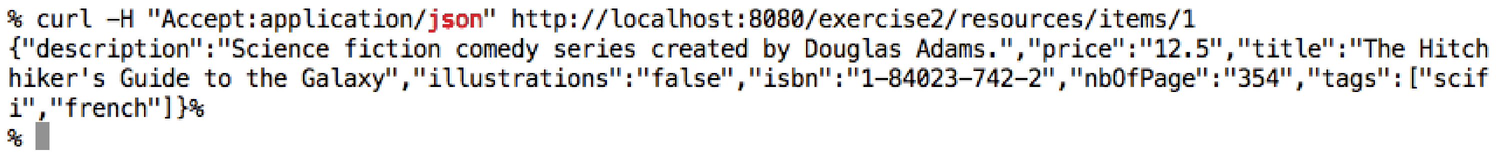
In `ItemEJB.java`, modify the `@Produces` annotation to offer json as an additional output format (this is the only thing required to do, courtesy of JAXB and the earlier `@XmlRootElement`) :

```
@Produces({MediaType.APPLICATION_XML,MediaType.APPLICATION_JSON})
```

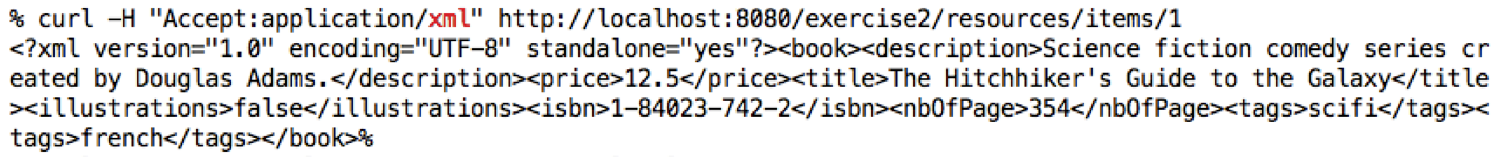
Now redeploy the application. Now you can open a terminal window and use the cURL tool to invoke the following commands from the command line :

```
1. curl http://localhost:8080/exercise2/resources/items/123
2. curl -H "Accept:application/json"
        http://localhost:8080/exercise2/resources/items/123
3. curl -H "Accept:application/xml"
        http://localhost:8080/exercise2/resources/items/123
```

- The first command will return an XML representation of book Id 123.
- The second command uses an HTTP "`Accept:`" header and will return a JSON representation of that same book.
- The third command explicitly switches back to an XML representation.



```
% curl -H "Accept:application/json" http://localhost:8080/exercise2/resources/items/1
{"description":"Science fiction comedy series created by Douglas Adams.","price":"12.5","title":"The Hitch
hiker's Guide to the Galaxy","illustrations":"false","isbn":"1-84023-742-2","nbOfPage":"354","tags":["scif
i","french"]}%
% 
```



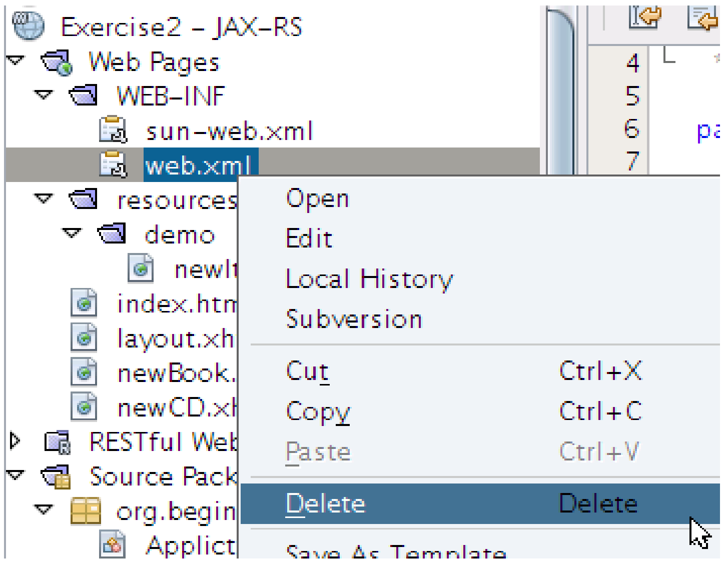
```
% curl -H "Accept:application/xml" http://localhost:8080/exercise2/resources/items/1
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><book><description>Science fiction comedy series cr
eated by Douglas Adams.</description><price>12.5</price><title>The Hitchhiker's Guide to the Galaxy</title
><illustrations>false</illustrations><isbn>1-84023-742-2</isbn><nbOfPage>354</nbOfPage><tags>scifi</tags><
tags>french</tags></book>%
```

Remove web.xml

Finally, as promised, to get rid of the `web.xml` configuration file, you can use a new feature of JAX-RS 1.1 which is to create the following empty class in the `org.beginningee6.tutorial` package :

```
@javax.ws.rs.ApplicationPath("resources")
public class ApplicationConfig extends
                     javax.ws.rs.core.Application {
```

This maps the JAX-RS (Jersey in our case) servlet to the "resources" path and `web.xml` can be deleted altogether (a new feature for Servlet 3.0).



You can now run the application one more time to verify that everything is style working as previously since this is merely a refactoring to remove XML configuration code.

Exercise 3 (CDI)

If you haven't already done so, close the "`Exercise2-JAX-RS`" project and open the project called "`Exercise3-CDI`" and located in `$LAB_HOME\JavaEE6_Lab`.

Introduction

CDI or Context and Dependency Injection (JSR 299) is an exciting new addition to the Java EE platform. From a high-level perspective it can be described as offering strongly-typed and loosely-coupled dependency injection, automatic context management (such as a "conversation" scope) as well as a number of other features (events, stereotypes, etc...). This HOL exercise is an introduction to CDI and focuses on the injection part.

CDI builds on top of the annotations defined in JSR 330 (Dependency Injection for Java), namely **`@Inject`**, **`@Qualifier`**, and **`@Named`**. We will use all three annotations in the context of our web application running in the application server. Note that CDI is a required component for the **Java EE 6 Web Profile** which means that any container supporting this standard will be able to execute the following code.

Enable CDI

In order to have CDI available to the runtime, the artifacts deployed need to carry along a **`beans.xml`** file (located in `WEB-INF/` in our case). Throughout this lab, this file will be empty but it needs to be present for CDI to kick in (this file can also be used for more advanced use-cases such as *alternatives* where separation of responsibilities is a key requirement).

Using the IDE or your favorite method (Unix `touch` for instance), create an **empty `WEB-INF/beans.xml`** file in your project. With NetBeans, either right-click on `WEB-INF` and chose `NEW > XML Document...` (make sure you type "beans" for the name as the extension will be automatically appended) or right-click on the project and chose `New > Other… > Context and Dependency Injection > beans.xml`.

Standardize on `@Inject`

In `ItemEJB.java`, replace occurrences of `@Resource` with `@Inject` and adjust imports (right-click in the source code and select "`Fix Imports`" or use `Ctrl+Shift+I`). This delegates to CDI the injection of the resources. The injection resolution will be calculated at startup so any error will be caught early (as we'll soon see) rather than bogus injections causing `NullPointerExceptions` at runtime.

Similarly in `ItemBean.java`, replace `@EJB` with `@Inject` and adjust imports.

At this point you could run the application again and understand that this is simply a refactoring not changing the behavior in any way.

Access from JSF facelets

Add `@Named("languages")` to the class defined in **`LanguageSingleton.java`** to make all of its attributes available from the EL (Expression Language) inside the JSF XHTML views. Also notice the new following method :

```
public String getListOfValues() {
    return languages.keySet().toString();
}
```

In **`newBook.xhtml`**, after the language code `<h:inputText>` on line #30, add the following code (ideally using code-completion to avoid typos) :

```
<h:outputLabel value="*" styleClass="alabel"/>
<h:outputLabel value="#{languages.listOfValues}"
               styleClass="ainput"/>
```

This uses the bean called "`languages`" (which was defined above with `@Named`) to enumerate valid options for the two-letter language codes. Note that the `@Named` annotation can be used without an attribute in which case the bean name is derived from the Java class name.

Strongly-typed injection using qualifiers

Consider now the `Customer.java` source file. It defines a simple Java interface with two methods implemented by concrete classes defined in `DefaultCustomer.java` and `SpecialCustomer.java`. As you can see from the straightforward code, the "default" customer cannot buy anything while the "special" customer can.

In `ItemBean.java`, underneath the JSF `@ManagedBean` annotation add the `@Named("controller")`. Also change also the scope of the bean to `@ConversationScoped` instead of `@RequestScoped`.

To start dealing with a customer, we need to inject the following attribute to the `ItemBean` class (make sure you use "`cust`" as the variable name) :

```
    @Inject Customer cust;
```

… and **uncomment** the following helper methods :

```
    getForbiddenToBuy()
    getBuyButtonLabel()
    doBuy()
```

The first two methods are bound from the JSF `newBook.xhtml` page (see lines #114 and #115) while the `doBuy()` method will actually place the order for a given ISBN number (line #113).

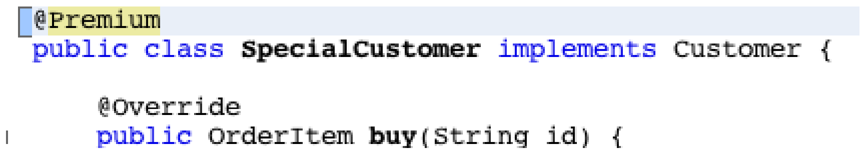
Try to **run the application** (right-click > Run).

The **deployment should fail** and the GlassFish log window (bring it up using a right-click on the GlassFish node in the "`Services`" tab) should show a message similar to this (press `Shift+Escape` to maximize/minimize the log window) :

```
Exception while loading the app : WELD-001409 Ambiguous dependencies for type
[Customer] with qualifiers [@Default] at injection point [[field] @Inject
org.beginningee6.tutorial.ItemBean.cust].
     Possible dependencies
[[Managed Bean [class org.beginningee6.tutorial.DefaultCustomer] with
qualifiers [@Any @Default],
  Managed Bean [class org.beginningee6.tutorial.SpecialCustomer] with
qualifiers [@Any @Default]]]
```

As you can see, the application server does not have enough information about which specific `Customer` implementation it should inject as both `SpecialCustomer` and `DefaultCustomer` both implement this interface.

To solve this we'll use a CDI **Qualifier** defined in `Premium.java`. This is a custom-defined annotation which you can use to mark implementations and to qualify injections. In our case we need to add `@Premium` to decorate the `SpecialCustomer` class :

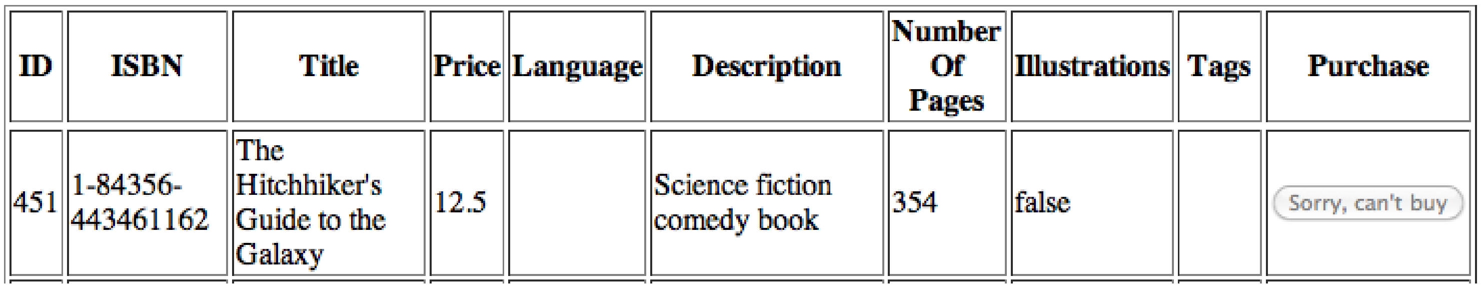


```
@Premium
public class SpecialCustomer implements Customer {

    @Override
    public OrderItem buy(String id) {
```

Now try to **re-run the application**.

You should see that the application server resolves the injection by providing the `DefaultCustomer` implementation. This means that the user is **not able to buy anything** and as such the UI should show disabled controls.

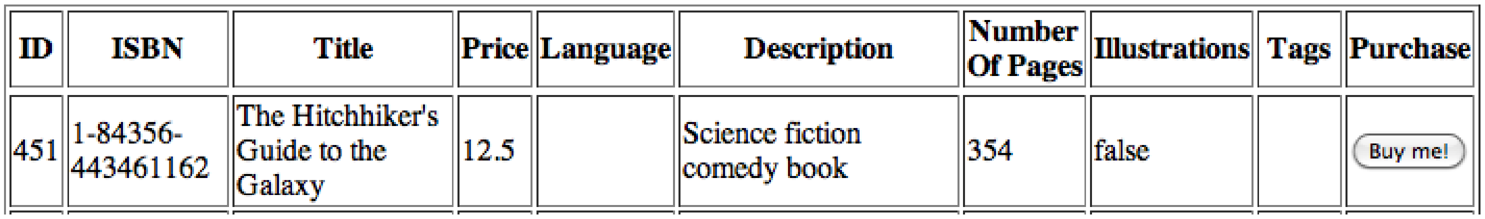


ID	ISBN	Title	Price	Language	Description	Number Of Pages	Illustrations	Tags	Purchase
451	1-84356-443461162	The Hitchhiker's Guide to the Galaxy	12.5		Science fiction comedy book	354	false		Sorry, can't buy

Now modify the injection in `ItemBean.java` to the following :

```
    @Inject @Premium Customer cust;
```

Run one more time the application and see that this time the implementation used is SpecialCustomer and that the user is indeed **able to buy**. There is no checkout feature implemented here so you'll need to check the server log for actual purchasing messages.



ID	ISBN	Title	Price	Language	Description	Number Of Pages	Illustrations	Tags	Purchase
451	1-84356-443461162	The Hitchhiker's Guide to the Galaxy	12.5		Science fiction comedy book	354	false		Buy me!

This concludes the CDI injection part of this lab. The key take-away is the ability to change the implementation without having to change the calling/injection code. This is the reason CDI is being considered as loosely-coupled while the qualifier part is what makes it strongly-typed (no string involved).

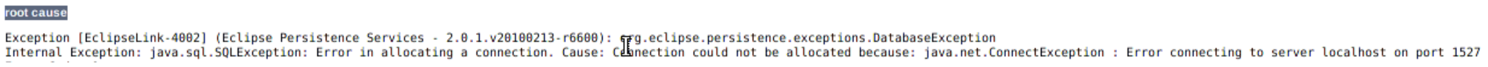
Conclusion

Hopefully this hands-on lab will have you curious enough to go download GlassFish and try out Java EE 6 for yourself. Start developing with Java EE 6 today!

See you on http://beginningee6.kenai.com/ for a more complete set of new Java EE 6 features.

Troubleshooting

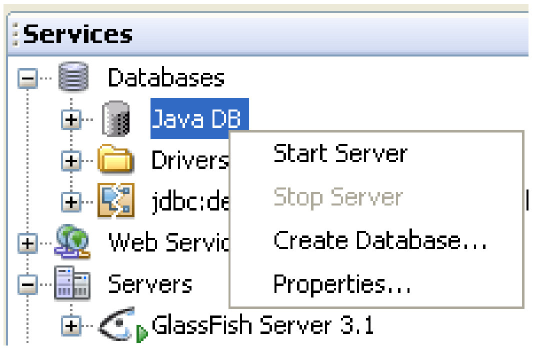
Error connection to server localhost on port 1527

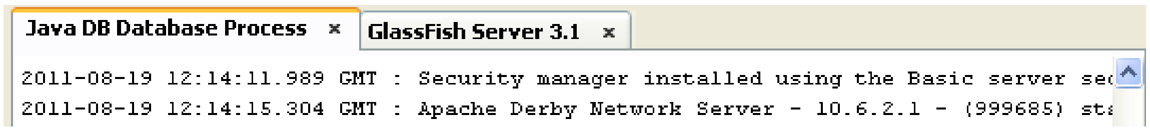


```
root cause

Exception [EclipseLink-4002] (Eclipse Persistence Services - 2.0.1.v20100213-r6600): org.eclipse.persistence.exceptions.DatabaseException
Internal Exception: java.sql.SQLException: Error in allocating a connection. Cause: Connection could not be allocated because: java.net.ConnectException : Error connecting to server localhost on port 1527
```

Port 1527 is the database (JavaDB/Derby) default port. If the database isn't already started locally, you'll need to start it straight from the NetBeans IDE using Services > Databases > Java DB > Start Server as shown in the following screenshots.

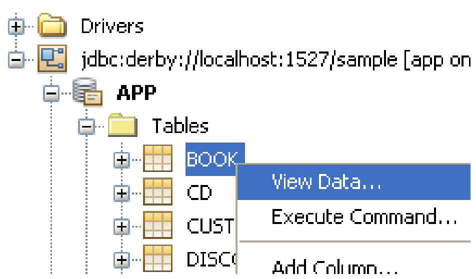


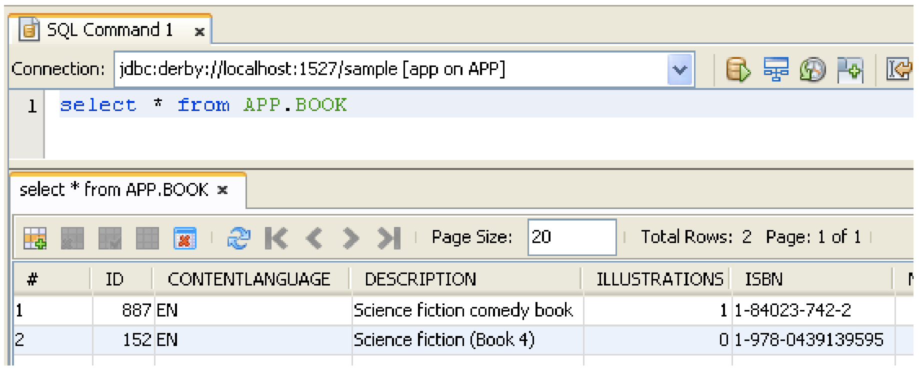


```
2011-08-19 12:14:11.989 GMT : Security manager installed using the Basic server sec
2011-08-19 12:14:15.304 GMT : Apache Derby Network Server - 10.6.2.1 - (999685) sta
```

Manipulating the Database

NetBeans makes it trivial for you to visualize the tables in the database, view the data, remove records, add new ones, drop tables, etc. All of this is possible from the "Databases" node in the Services tab :





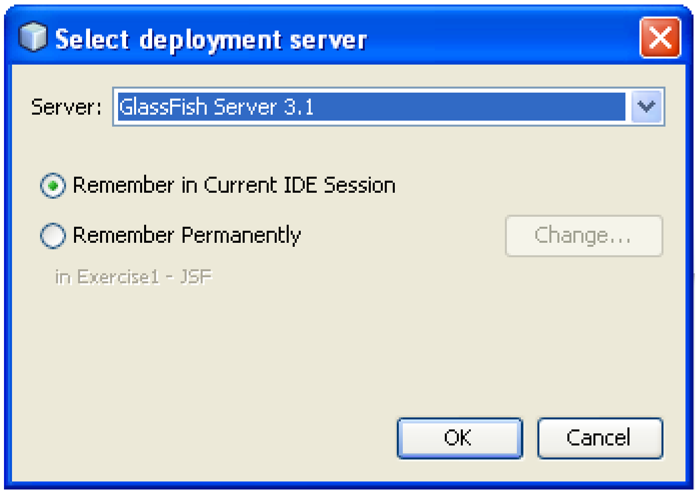
Connection: jdbc:derby://localhost:1527/sample [app on APP]

```
select * from APP.BOOK
```

#	ID	CONTENTLANGUAGE	DESCRIPTION	ILLUSTRATIONS	ISBN
1	887	EN	Science fiction comedy book	1	1-84023-742-2
2	152	EN	Science fiction (Book 4)	0	1-978-0439139595

Select deployment server

On the first run of each exercise you may be asked to select which server instance you want to deploy to. You are presented with a list of servers defined in the NetBeans IDE. For this lab, there should be only one choice. For faster development turnaround you are encouraged to ask the IDE to remember that choice.


Select deployment server

Server: GlassFish Server 3.1

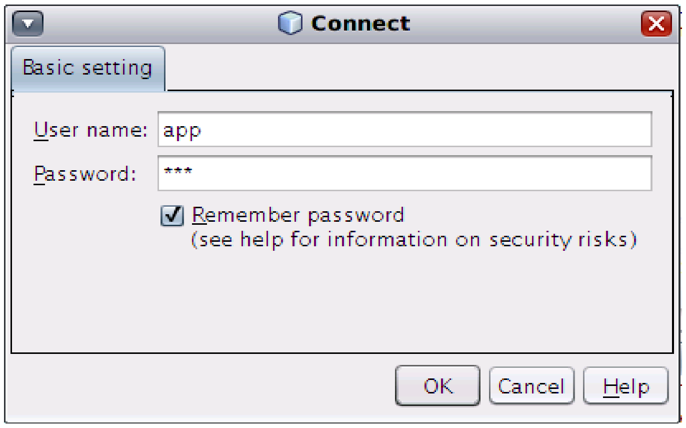
(●) Remember in Current IDE Session

( ) Remember Permanently

in Exercise1 - JSF

Database access credentials

This lab uses a default database hosted in a local JavaDB database. If you're requested a user/password to navigate through the JDBC connection from the NetBeans IDE (Services tab > Database), user “APP” and password “APP” should get you in.


Connect

Basic setting

User name: app

Password: ***

[✓] Remember password
(see help for information on security risks)

Getting further Database access information

The application uses JPA to interact (read/write) with the JavaDB/Derby database. To obtain further logging details on the dialog with the database (connection, persistence unit, SQL statements, …), you can set the EclipseLink logging level to `FINE` or `FINEST` in the project's `META-INF\persistence.xml` as follows :

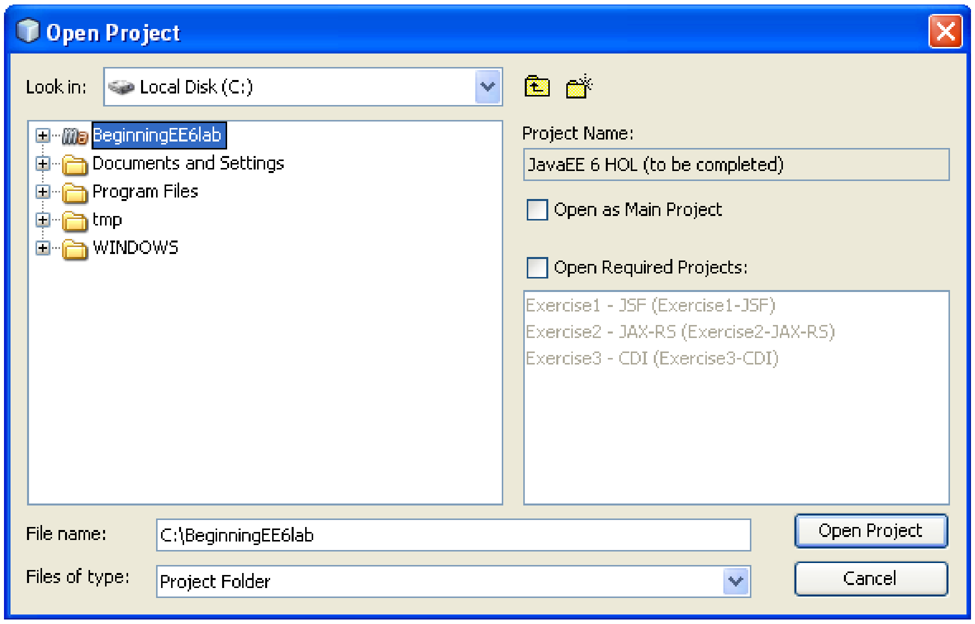
```
<property name="eclipselink.logging.level" value="FINE"/>
```

Accessing the Application Server (GlassFish) Admin Console

The GlassFish web administration console is available from the IDE (right-click on the GlassFish node) or directly from http://localhost:4848

No project loaded in the IDE

If for some reason the NetBeans IDE starts and no project is already open in the left "Project" tab, you'll need to open the main Maven project which is located in the user home directory. As you can see on the following screenshot on the right-hand side you can also open all the required project (all three exercises for this lab) :



Project Name: JavaEE 6 HOL (to be completed)

Open as Main Project

Open Required Projects:

Exercise1 - JSF (Exercise1-JSF)
Exercise2 - JAX-RS (Exercise2-JAX-RS)
Exercise3 - CDI (Exercise3-CDI)

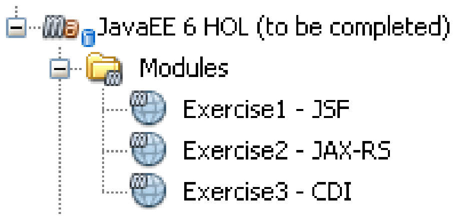
File name: C:\BeginningEE6lab

Files of type: Project Folder

Open next exercise project

If you don't have the main `“Java EE 6 HOL (to be completed)”` project open, please refer to the above entry on “No project loaded in the IDE”.

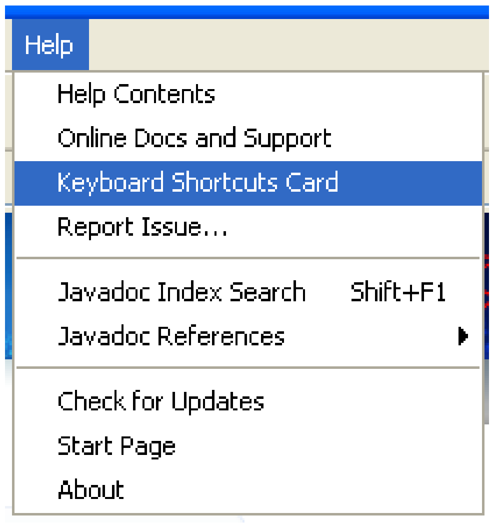
Once the main project is loaded in the IDE, you can open any of the three exercise projects by double-clicking on the project name in the “Modules” folder of the main project :



This will cause the given referenced Maven project to be opened in the IDE.

Where can I find a list of shortcuts for the NetBeans IDE?

Simply go to `Help > Keyboard Shortcuts Card`. Shortcuts can be customized in the IDE preferences and defaults vary on different platforms (Mac vs. Windows vs. *nix).



Ok, so GlassFish is behaving really bad and I'd like to restart it, how do I do it?

Start GlassFish :

```
c:> %GLASSFISH_HOME%\bin\asadmin start-domain
```

Stop GlassFish :

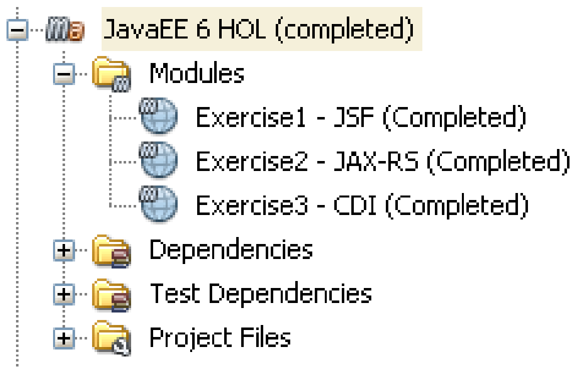
```
c:> %GLASSFISH_HOME%\bin\asadmin stop-domain
```

Restart GlassFish :

```
c:> %GLASSFISH_HOME%\bin\asadmin restart-domain
```

Ok, get me to the solution!

The completed code for all three exercises is available in a second top-level Maven project located in the Solution sub-directory. Note that both this project and the solutions are marked with the "`Completed`" tag :



```
JavaEE 6 HOL (completed)
    Modules
        Exercise1 - JSF (Completed)
        Exercise2 - JAX-RS (Completed)
        Exercise3 - CDI (Completed)
    Dependencies
    Test Dependencies
    Project Files
```