University of Central Lancashire
UCLan

CO3409 Distributed Enterprise Systems

Lab : Enterprise Java Beans

Summary

Use NetBeans to develop a variety of Enterprise Beans and interact with them in different ways. The skills learnt here would be important in the development of a full application.

Purpose

On completion of this you should be able to

- Create a variety of Beans: message beans, session beans, entity classes.
- Connect to EJB applications from Web applications and standalone clients.
- Explain the role of Dependency Injection
- Use Interceptors to modify the way methods are used.

Things to Remember

Some things to remember...

- Run Netbeans as "Administrator".
- When running your application, if you are using a database please ensure your database is running first (see previous lab worksheets).
- In this exercise, if you restart Payara or when deploying applications, make sure that the action is complete before moving on.
- To deploy an Enterprise Application, I suggest using Clean and Build and then Deploy.

Activities

Before you begin, Check on the Java Version

1. Make sure that JAVA_HOME is pointing to a JDK 8 installation folder: open a command prompt window and type:

   ```
   echo %JAVA_HOME%
   ```

   You should see a path output to the command prompt window like:

```
C:\Program Files\Java\jdk1.8.0_271
```

If not, follow the instructions here.

2. Run Netbeans, make sure Payara is being launched with JDK 8. **Tools**, **Servers**, **Java** tab, **Java Platform**.

1. Creating a Java Class Library

The Java Class Library project created here will contain the remote interface that can be used by clients to communicate with the EJB.

1. Choose **File** > **New Project** and select **Java Class Library** in the **Java with Ant** category. Click Next.

2. Use `EJBRemoteInterface` for the Project Name. Uncheck **Create Main Class**. Click Finish.

The application will provide a trivial calculator that provides two methods: one to add its parameters, the other to divide them.

- ***Should the bean that provides this service be a stateful or stateless bean?***

  > Stateless, because the methods should return the same results every time they are called. They do not need to preserve any information (state) from one call to the next.

The application will also provide a cart that provides two methods: one to add items to the cart, the other to get a list of the items in the cart.

- ***Should the bean that provides this cart service be a stateful or stateless bean?***

  > Stateful, because the system must retain the list of items: it must preserve this information (state) from one call to the next.

2. Creating an Enterprise Application With a Stateless Session Bean

1. Create a new Enterprise Application project from the Java with Ant -> Java Enterprise category. Call it `EDemo`. Note: use Java EE 7. Check **Create EJB Module** and **Create Web Application** Module. **Finish**.

2. Right-click on `EDemo-ejb` in the Projects pane and create a new session bean called `Calc` (**New**, **Other**, **Enterprise JavaBeans**, **Session Bean**. Use the package `EABean`. Make sure the Stateless option is checked. Create just a Remote interface (i.e. not a Local interface). Make sure the Remote Interface is in the project: `EJBRemoteInterface`.

- ***What is the role of a Session Bean?***

  > It provides the main logic (the business logic) of the specific application. It doesn't provide the user interface, but it provides the service that the client requires. It may provide a response to a single request (stateless bean) or participate in a dialogue (conversation) with the client (stateful bean).

- ***What do you think is the difference between the Remote Interface and the Local Interface that a bean could provide?***

  > The remote interface is made available to clients interacting across the Web. The local interface is only available to beans in the same application.

2.1 Creating Business Methods on the Bean

1. Right-click in the body of `Calc` and use **Insert Code** to **add the business method** `int add (int a, int b)`. You will have to work out for yourself how to fill in the form, but just put it in the Remote Interface. Repeat with divide, but add an `ArithmeticException` to the method parameters. Enter the bodies of the methods.

2. Add another Business Method, `getTotalCalls();`, which will return the number of calls that have been made to methods of this bean. (This is not a particularly useful method, but it will demonstrate state in a stateless bean. It might be used if you were charging for each use of a method.) To implement this method, you will need a property (`totalCalls`), initialised to 0, that you increment each time a method (add or divide) is called.

2.2 Adding a stateful session bean to the Enterprise Application

1. Right-click on `EDemo-ejb` and create a new stateful Session Bean called `Cart` in the `EABean` package. Create only a Remote interface and put it in the `EJBRemoteInterface` project. Like a shopping cart, this bean will store items for the user.

2. Create an instance variable (a property of the class):

```
private ArrayList items; // hold the user's items
```

3. Fix the imports.

4. Add a parameter-less method called void `initialize()` by right-clicking on the class and selecting Insert Code, Add Business Method. The body of `initialize()` should be:

```
items = new ArrayList();
```

5. Add the annotation, `@PostConstruct`, to `initialize()`. This asks the container to call the method automatically after an object from class has been constructed.

6. Right-click and fix imports.

7. `@PostConstruct` is a lifecycle annotation. What does this mean?

   > A lifecycle annotation is used to tell the container to call the method at particular times in the beans lifecycle. A method with this annotation is called just after the bean has been constructed.

8. Add a void `addItem(String item)` Business Method and provide a suitable body.

9. Add the following business methods with appropriate bodies: `public void removeItem(String item)` and `public Collection getItems()`.

10. Your cart class should have the following body:

```
private ArrayList items; // hold the user's items

@PostConstruct
public void initialize() {
    items = new ArrayList();
}

public void addItem(String item) {
    items.add(item);
}
public void removeItem(String item) {
    items.remove(item);
}
public Collection getItems() {
    return (Collection) items.clone();
}
```

11. Click on the little hints on the left of each method (in `Calc` and `Cart` beans) and choose "***Expose method in remote business interface***" if this option is available. Add `@Override` to each method. Check what has happened to your `EJBRemoteInterface` project.

3. Accessing an Enterprise Bean from a Web Application (Servlet)

3.1 Create a Servlet

1. Right-click on the Web Application project, `EDemo-war`, which is part of the `EDemo` Enterprise Application. Add a new Servlet, `EAServlet` in a package `EAWebUnit`. Note the relative address in the URL Pattern(s).

2. Test the Servlet by running the `EAWeb project`. This will display the `index.html` page in the browser. Note the address (this is probably http://localhost:8080/EAWeb/) and then run the servlet by adding `EAServlet` to this address.

   This just shows that we can create a working Servlet, but it's useful to check this before we add any complex code. Add a link to the Servlet to the home page (Servlet).

3.2 Accessing Session Bean from a Servlet

1. Right-click in the body of the Servlet class, and select `Call` Enterprise Bean from Insert Code. Select `Calc`. Add the following code before the `</body>` is output:

```
out.println("<p>1 + 2 = " + calc.add(1, 2)+"</p>");
out.println("<p>Total calls = " + calc.getTotalCalls()+"</p>");
```

2. Now Run `EADemo` and Test the servlet.

> **Please Note**
>
> Any problems deploying that don't arise from Java syntax errors, try switching to the Services tab. Undeploy all the applications and deploy the Enterprise Application `EDemo`, then refresh the servlet in the browser. It can sometimes be helpful to stop the server. Look at the Server Tab at the bottom to see when it has finished.

3. Set up the Servlet to use the cart bean: Right-click in the Servlet class, and select `Call` Enterprise Bean from Insert Code. Add the following code to the Servlet `processRequest()` method before the `</body>` is output:

```
cart.addItem("first");
cart.addItem("second");
ArrayList al = (ArrayList) cart.getItems();
out.println("<table border=\"1\">");
for (Object s : al) {
    out.println("<tr><td>" + s + "</td></tr>");
}
out.println("</table>");
```

4. Fix the imports.

5. Right-click on `EADemo` project and select Properties. In the Run category, enter `/EAServlet` in the Relative URL – this will be the "home page" for the project.

6. Redeploy and run the application, which should launch the Servlet. Refresh the page several times.

   - ***So what happens?***

     > The state of the bean is preserved so each time the method is called, additional items are added to the cart – the list grows.

- ***Where is dependency injection being used?***

  > The container initialises the `calcBean` variable – there is no explicit code in the application to set it up. So the bean that the main routine **depends** on is **injected** into the `calcBean` variable. Notice how the `@EJB` annotation is used. Of course this only works if the container is sufficiently sophisticated to carry out the injection.

  The container must provide the same session bean each time the client makes a request during a session (sessions may be closed explicitly by the client, e.g. when a shopping cart is emptied explicitly or its contents are bought, or after a period of time when they are not used.). You might think that the container would treat any bean with an instance variable (a property) as a stateful bean.

  Can you think of any reason why the developer has to use the `@stateful`, `@stateless` ***annotation explicitly?***

  > A business method in a bean may call other bean methods. It may be useful to use an instance variable to hold shared information (e.g. a connection to a database) that several of these methods need to access. If this information does not need to be retained for the next time a business method is called, there is no reason for this to be a stateful bean. Stateless beans are more efficient because the container can use any instance of that bean with any client.

4. Creating a Client Application that looks up the Enterprise Application

1. Use **File, New Project** to create a new **Java with Ant**, **Java Application** called `Edemoclient` with a Main class.

2. Right-click on the project properties and navigate to the libraries Classpath and add the library, `JavaEE` from Glassfish.

3. Also, Add the Jar `gf-client.jar` which will be in Payara installation folder `<Payara installation folder>\glassfish\lib`.

4. Also add the libraries from the following directories `<Glassfish folder>\modules\gf-client-module.jar` and `<Glassfish folder>\modules\orb-iiop.jar`.

5. Add the `EJBRemoteInterface.jar` to the library – you can do this by adding the `EJBRemoteInterface` **Project** to the library.

6. Right-click on `CalcRemote.java` in the `EJBRemoteInterface` project in the Projects tab and select Copy. Paste this into `Edemoclient` sources folder (without refactor). Open the copy of `CalcRemote.java` in the editor and Right-click on the light-bulb and select **Move class to correct folder**. Make sure the package name is `EABean`.

7. Replace the content of the main of `Edemoclient` class with the following:

```
try {
    System.out.println("Looking up Calculator");

    Properties props = new Properties();
    props.setProperty("org.omg.CORBA.ORBInitialHost", "127.0.0.1");
    props.setProperty("org.omg.CORBA.ORBInitialPort", "1072");
    InitialContext ic = new InitialContext(props);

    System.out.println("Context found");
    CalcRemote calcBean = (CalcRemote)
                            ic.lookup("java:global/EDemo/EDemo-
ejb/Calc!EABean.CalcRemote");
    System.out.println("Calling calculator");
    // Call any of the Remote methods to access the EJB
    System.out.println("1 + 2 = " + calcBean.add(1, 2));
    System.out.println("Total = " + calcBean.getTotalCalls());
    System.out.println("Looking up cart");
    CartRemote cart = (CartRemote)
                                ic.lookup("java:global/EDemo/EDemo-
ejb/Cart!EABean.CartRemote");

    System.out.println("got cart");
    cart.addItem("able");
    cart.addItem("baker");
    Collection c = cart.getItems();
    System.out.println("collection: " + c.toString());
    System.out.println("Done");
        } catch (NamingException ex) {
    ex.printStackTrace();
        }
```

8. Find the appropriate port (`org.omg.CORBA.ORBInitialPort` was `1072` for me) by right-clicking on the Payara instance in the Services tab, starting it if necessary, and choosing, View Domain Admin Console.

9. In the left-hand pane, expand Configurations, server-config, ORB. Select IIOP listeners and use the port for orb-listener-1 as the port set in the program.

10. Fix the imports. You will get a warning about using `printStackTrace()`, but it's a reasonable way of handling an error during debugging.

11. Right-click on the `EDemo` project and select **Deploy**. Right-click on `Edemoclient` project and select *Run.*
    - ***So why do we look for the remote interface rather than the specific bean that implements it?***

      > Because we don't care which component we get as long as it delivers that interface – it would be stupid to tie our client to one specific implementation – the client would then break if the bean was replaced by a new, better implementation.

5. Using A Message Bean and an Entity Class to Access a Database

5.1 Preparation : Creating a Database

On the Services tab, right-click on Java DB and create a database called `log`, with a suitable userid and password. (I suggest you use `APP` and `APP` to avoid having to create a schema to correspond to that userid. Creating a schema was dealt with in an earlier worksheet but there's little point creating the extra work today.)

5.2 Preparation : Registering the Database with the Server

1. Add a `jdbc` resource and a `jdbc` connection pool to Payara. Open the Payara Admin Console (Services Tab, Servers, Right-click on Payara server). Click on **Common Tasks**, **Resources**, **JDBC**, **JDBC Connection Pools**. Click on **New**.

2. In the general tab, input the following information:

   ```
   Pool Name: logpool,
   Resource Type: javax.sql.DataSource,
   Database Driver Vendor: Derby,
   ```

   Next

   ```
   Datasource Classname: org.apache.derby.jdbc.ClientDataSource.
   ```

   In the Additional Properties, enter

   ```
   Password: APP
   User: APP
   serverName: localhost
   DatabaseName: log
   connectionAttributes: create=true
   ```

3. Open the Payara Admin Console (Services Tab, Servers, Right-click on Payara server). Click on **Common Tasks**, **Resources**, **JDBC**, **JDBC Resources**. Click on **New**.

   ```
   JNDI Name:     jdbc/log. Pool Name: logpool.
   ```

5.3 Creating an Entity Class

1. Right-click on `EDemo-ejb` and from the Persistence category, create a new Entity Class called `LogRecord` and put it in the `EABean` package. Create a Persistence Unit, leaving all the settings at the default but adding a new data source that links to the log database. (Use the JNDI name jdbc/log and connect to the database connection, log. Make sure the Data Source just shows as jdbc/log).

2. Switch to source mode and add the following property to persistence.xml

```
<property name="eclipselink.ddl-generation" value="drop-and-create-tables"/>
```

> **Note**
>
> To change the value, rub out all the string for the value, including the quotes. Type a " (value=") and wait for the options to download.

3. Add the following properties to the class `LogRecord`.

```
private String subject;
private String messageBody;
private String theTime;
```

4. Right-click in the class and select Insert Code. Select Getters and Setters and add getter and setter methods for all the new properties.

5.4 Creating a Message Bean

1. Right-click on `EDemo-ejb` and create a new `Message-Driven` Bean from the Java EE category. Call it `Logger`, and put it in the `EABean` package. Add Project Destinations, name `LogQueue`, destination type `Queue`.

2. Right-click in the Bean and select **Insert Code**, **Use Entity Manager**. Note that an `EntityManager` variable is created. It will be initialised by **Dependency Injection**.

3. Insert the following instance property:

```
@Resource
private MessageDrivenContext mdc;
```

4. Fix any imports, making sure you choose the right packages.

5. Add the following to the body of the `onMessage` event handler:

```
ObjectMessage msg;
try {
    if (message instanceof ObjectMessage) {
        msg = (ObjectMessage) message;
        LogRecord e = (LogRecord) msg.getObject();
        persist(e);
    }
} catch (JMSException e) {
    e.printStackTrace();
    mdc.setRollbackOnly();
} catch (Throwable te) {
    te.printStackTrace();
}
```

6. Fix the imports.

5.4 Sending Messages to a Message Bean

1. Open `Calc.java`, right-click inset the following properties and method.

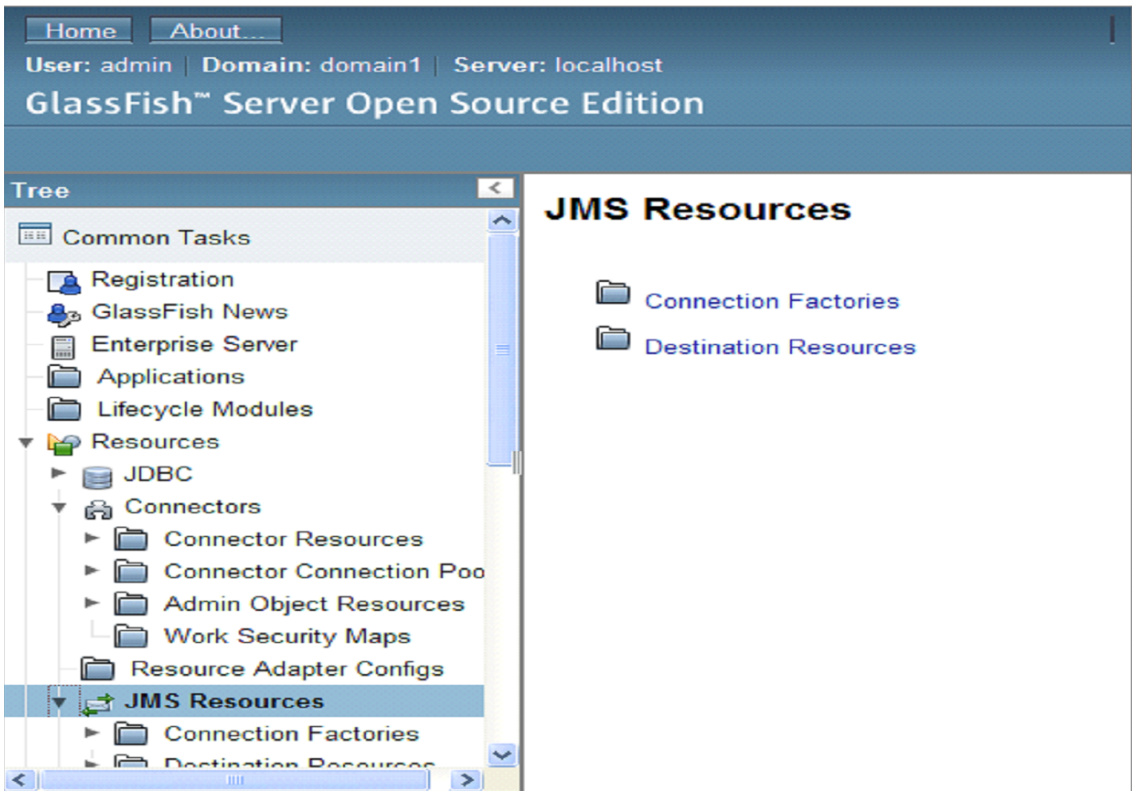
```
@JMSConnectionFactory("java:comp/DefaultJMSConnectionFactory")
    @Inject
    private JMSContext context;

    @Resource(lookup = "jms/LogQueue")
    private Queue logQueue;

        private void logIt(String MessageBody) {
        DateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");
        Date date = new Date();
        System.out.println();
        LogRecord theRecord = new LogRecord();
        theRecord.setSubject("CalcBean Log");
        theRecord.setMessageBody(MessageBody);
        theRecord.setTheTime(dateFormat.format(date) );

            try {
            ObjectMessage m = context.createObjectMessage(theRecord);
            context.createProducer().send(logQueue, m);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

2. Add the following statement to `add()` before the return:

```
logIt("Adding "+ a + " "+ b);
```

5.5 Creating a message queue and a factory (if not already created for you)

1. On the Services tab, right-click on Payara Server and run the domain admin console. Login if necessary try: **userid**: `admin`, **password**: `adminadmin`. Select Common Tasks. Click on JMS Resources.



2. Click on Connection Factories.

3. If the right connection factory doesn't exist, click on **New ...**

    1. Enter the name of the factory required by the annotation in `Calc`:

```
@JMSConnectionFactory("java:comp/DefaultJMSConnectionFactory")
// @Resource(name = "jms/LogQueueFactory")
```

2. Select the type `QueueConnectionFactory`

3. Click on OK.

4. Click on Destination Resources under JMS Resources.

5. Create a queue to satisfy: `@Resource(lookup = "jms/LogQueue")`.

6. Click on OK.

7. Restart Glassfish. You can do this from Netbeans.

5.6 Testing the Application

1. Build and deploy the Enterprise application.

2. If you get the error below:

```
error: java.lang.ClassFormatError: Absent Code attribute in method that is
not native or abstract in class file javax/persistence/PersistenceException
```

   right click on libraries for `EJBRemoteInterface`, if Java EE 6 API library is present, remove it and replace it with Java EE 7 API library.

3. Watch the outputs carefully and, if necessary because the deployments fail, undeploy the applications and, if necessary, shut down and restart Glassfish (or ultimately NetBeans). This should ensure that NetBeans can delete all the deployed files that it needs to replace.

4. Run the client application.

5. Check that the results have been inserted into the database by connecting to the log database in the Services tab, expanding the Tables node, right-clicking on the LOGRECORD and choosing View Data. >

   > If you get any errors, check your code and redeploy.

6. Execute the SQL.

5.7 Using an Interceptor Class

1. Create a new class called `Profiler` in the package `eaBean`.

2. Add the following method to Profiler:

```
@AroundInvoke
public Object profile(InvocationContext ctx) throws Exception {
    System.out.println("Profiling Interceptor called");
    long startTime = 0;
    long endTime = 0;
    try {
        startTime = System.currentTimeMillis();
        return ctx.proceed();
    } finally {
        endTime = System.currentTimeMillis();
        System.out.println("Method " + ctx.getMethod()
                           + " took " + (endTime - startTime) + "ms ");
    }
}
```

3. Fix the imports.

5.7.1 Applying the Interceptor to a target class

1. Modify `CalcBean.java` by adding the following annotation just before `public class CalcBean`.

```
@Interceptors(Profiler.class) /* identify external interceptors */
```

2. Fix the imports. Build all the altered programs.

5.7.2 Testing the Interceptor

1. Run (or debug) the application. (If there are problems deploying, undeploy the application from the Server in the Services tab before using run/debug)

2. Check that the results have been displayed in the Server output tab. Not surprisingly, `System.out` for beans refers to the server output window, not the NetBeans output window.

   > If you get any errors, check your code and redeploy.

5.7.3 Adding an Interceptor Method to a Bean

1. The Interceptor uses the following property that should be added to `CalcBean`:

```
@Resource
private SessionContext ctx;
```

2. Add the following method to `CalcBean`:

```
@AroundInvoke // mark this method as a bean interceptor
public Object checkPermission(InvocationContext ic) throws Exception {
System.out.println("*** checkPermission interceptor invoked by "+
ctx.getCallerPrincipal().getName());

// you can implement your own security framework using interceptors
if (!ctx.getCallerPrincipal().getName().equals("Chris")) {
throw new SecurityException("Caller: '" + ctx.getCallerPrincipal().getName()
+ "' does not have permission for method " + ic.getMethod());
}
return ic.proceed();
}
```

3. Run (or debug) the application. Check that the output is what you would expect: you should get a message from the `checkPermission` interceptor and `exception` information in the Server output window.

4. Replace the name `Chris` in the `CheckPermission` method with the Principal name (e.g. ANONYMOUS) just displayed as the caller and check that the method can now be called.

5.7.4 Creating a Default Interceptor

1. Create a new class called `DefaultInterceptor` in `eaBean`.

2. Add the following method:

```
@AroundInvoke
public Object monitor(InvocationContext ctx) throws Exception {
System.out.println("*** DEFAULT Interceptor called for " +ctx.getTarget());

System.out.println("*** Method: " + ctx.getMethod());

String parmString = "";
Object parm[] = ctx.getParameters();

for (Object p: parm){
    parmString = parmString + p.toString()+" ";
}

System.out.println("*** Parameters: " + parmString);
return ctx.proceed();
}
```

3. Fix the imports.

5.7.5 Attaching the Default Interceptor using a Deployment Descriptor

The `DefaultInterceptor` will be attached to all methods in `CalcBean` using a deployment descriptor. This means that you do not have to attach annotations to every method.

1. Right-click on the `EDemo-ejb` and choose New, Other, Enterprise JavaBeans, Standard Deployment Descriptor. This will create `ejb-jar.xml` in the configuration folder. Give it the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/ejb-jar_3_2.xsd"
version="3.2">

<interceptors>
<interceptor>
<interceptor-class>EABean.DefaultInterceptor</interceptor-class>
</interceptor>
</interceptors>

<!-- interceptor activation -->
<assembly-descriptor>
<interceptor-binding>
<ejb-name>Calc</ejb-name>
<interceptor-class>EABean.DefaultInterceptor</interceptor-class>
</interceptor-binding>
</assembly-descriptor>
</ejb-jar>
```

2. Run the application as before and check the output from Glassfish - it should report on the method called and its parameters. You may have to search through all sorts of other logging information.

   - ***What would happen if you used*** `<ejb-name>*</ejb-name>` ***instead of*** `<ejb-name>Calc</ejb-name>`?

     > The interceptor would be applied to all methods of all beans

   - ***What are the advantages and disadvantages of using annotations instead of descriptors to attach interceptors?***

     > With annotations, it's obvious which beans/methods interceptors are attached to. If you use an incorrect name for a bean in a descriptor, it can't be detected until deployment time and may even be missed then.
     >
     > To modify annotations, you have to recompile the source. Deployment descriptors can be modified and then it's just a matter of redeploying.
     >
     > You can attach interceptors to all beans, even beans that are added in the future with deployment descriptors, without having to modify the source of the beans.

> If you get an annotation wrong, you usually get an error to help find the problem. With deployment descriptors, if you are lucky you will get an error somewhere within the server log, but in many cases it will just go wrong and be very hard to track down.

6. Additional Work and Further Reading

6.1 Recording Method Times

Investigate the use of a database to store logging information. To reduce the time delay with the method call, use a message bean so that the interceptor can post the information to a message queue. The message bean can then save the information to the database. You will need to create a message bean and an entity class with the following properties:

```
private String methodName;
private int duration;
```

6.1 Further Reading

Read the following resources surrounding EJB and its features.

- **Overview of EJB3.1 features**
  - http://www.adam-bien.com/roller/abien/entry/ejb_3_session_for_absolute
  - http://www.adam-bien.com/roller/abien/entry/ejb_3_persistence_jpa_for

- **EJB specification**
  - https://download.oracle.com/otndocs/jcp/ejb-3_2-fr-eval-spec/index.html

    > (Pick JSR-000345 EJB 3.2 Core Specification for Evaluation). This is surprisingly readable for a formal document.

- **Overview of security:**
  - https://javaee.github.io/tutorial/security-intro001.html

- **Interceptors**
  - https://abhirockzz.wordpress.com/2015/01/03/java-ee-interceptors/http://webdev.jhuep.com/~jcs/ejava-javaee/coursedocs/content/html_single/javaee-interceptors-book.html