# INF5750 - Assignment 1

Deadline Date September 9, 23:59

You must pass this assignment to pass the course. Meet up to your lab sessions to ask your group teacher if you have any questions.

***Submitting the delivery: When you are done, you need to go to https://devilry.ifi.uio.no/ and upload a zip file containing your project code into the Assignment1 folder.***

***Copy paste: Important note about copy/paste from this document: When copying commands from this document into the command-line, sometimes the "-character turns out wrong, and the command or settings fail. If so, you need to write the "-character in manually from the keyboard.***

***Maven: Each time you update your pom.xml with new dependencies, remember to run mvn install (some IDEs do this automatically)***

# Introduction

This assignment walks you through some of the basic setup and use of your development environment, Spring, Maven, Hibernate.

You will have to implement this assignment either on your own laptop or the university computers.

If you have problems with the assignment, try googling for solutions, but also meet to the lab sessions to ask your group teacher.

# Assignment

These are some basic instructions on how to set up your development environment. We strongly recommend using your own laptop, and the prefered operating system for development and running the software is Linux (Ubuntu).

If you have to use University computers, some of the below programs are already installed.

From a bare-bone Linux Ubuntu, you need to install:

- Java (already installed on UIO computers)
- Eclipse (already installed on UIO computers, but we prefer a different version, so we would recommend installing your own version in your home folder)
- Maven (already installed on UIO computers)

First create the ~/opt directory if you don't have it.

*$ mkdir ~/opt*

*$ cd ~/opt*

## Installing Java (already installed on university computers)

There are several different suppliers of Java. Oracle is one, and OpenJDK is another. You can use either, but these are the instructions for openjdk. Type:

*$ sudo apt-get install openjdk-8-jdk*

## Installing Maven (already installed on university computers)

Maven is the build system. It helps by downloading the external objects you depend on automatically, but you have to tell it which components your files are dependent on. These dependencies are defined in a file called pom.xml. If maven is not available on the IFI-computer, please download from the maven site and unzip it:

Unpack maven:

$ tar xzvf apache-maven-3.3.9-bin.tar.gz

Move the files into ~/opt/mvn.

$ nano ~/.bashrc

Add the following line

Export PATH=$PATH:~/opt/mvn/bin

Save the file, ( Ctrl+x, then y and enter)

Restart your terminal(or execute bashrc) and check that maven installed properly

$ mvn -version

## Installing Eclipse (Enterprise Edition, not SDE)

### There is an eclipse installed on university computers, but the version you should use is the JEE-package, which is not installed. So you should follow these instructions to install the appropriate version on your home area.

Do not use apt-get to install eclipse. Instead, download the installation file from:

https://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/neonr

Unpack (or use the archive manager if you know how to do that)

*$ tar xzvf eclipse-<filename>.tar.gz*

Move the files into ~/opt/eclipse

Create a symlink in /usr/bin (not possible in university computers, so you have to add eclipse to your user path)

| On your machine | On university computer (TO ADD TO PATH) |
|---|---|
| *$ sudo ln -s ~/opt/eclipse/eclipse /usr/bin/eclipse* | $ cd ~/<br><br>$ nano .bashrc<br><br>Add the following line<br><br>*export PATH=$PATH:~/opt/eclipse*<br><br>*(Save the file. Ctrl+X, then Y and Enter)*<br><br>$ source .bashrc |

Run Eclipse from the terminal for the first time (after the first time, you should use: *$ eclipse &*)

*$ eclipse -clean &*

(Running this command at the university computers is likely to start up the default eclipse, which is not what you want. Instead use the path to your own eclipse. For example ~/opt/eclipse -clean & )

# Spring Framework

Spring is a framework to connect different java objects together into an application. The connections between java objects are managed using XML files or auto-discovery instead of hard-coding connections inside Java classes. This helps create a more configurable and flexible system. You will now create an eclipse project with some simple Spring components, using something called Spring MVC.

Spring documentation

For setting up a Maven project in eclipse (parts of this assignment is based on that web page, but not everything is the same), check out the concept of Archetypes

It is now time to create a directory where you want to keep your Java projects for this course. After creating this directory, continue to create a maven project with the

command "mvn archetype ...". However, instead of ‘Assignment1’, please use Assignment1_<username>. This will help your group teacher distinguish between the different projects. You do not have to change the package names in the project to include your username (but you must use (Assignment1_<username> for your project name/artifactid). This also means you have to change some of the example files in this assignment to include _<username>.

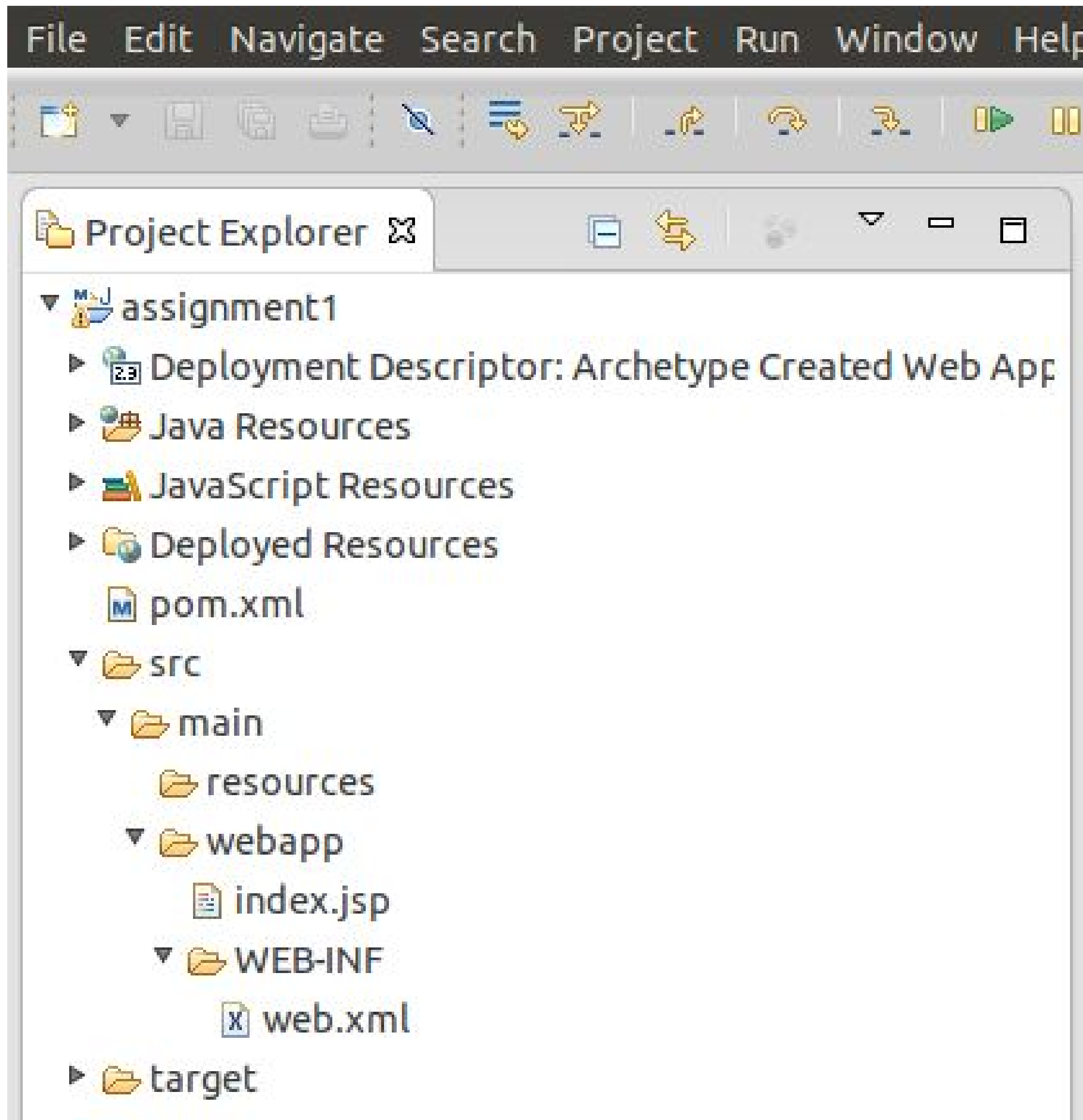
```
$ mvn archetype:generate -DgroupId=no.uio.inf5750  -DartifactId=assignment1
-DarchetypeArtifactId=maven-archetype-webapp -DinteractiveMode=false
```

Cd into the Assignment1-directory and look at the file structure. Note that you now have a pom.xml, which is the file that tells Maven how to build your project.

Import the maven project into eclipse using the user interface inside Eclipse by using the menu item "File->import->Maven->Existing Maven projects".

If you cannot see the "Import->Maven" selection, you are using an eclipse without the Maven plugin. Check out the top of the assignment. You have either not installed the JEE Eclipse package, or you have several eclipse installations and you are running the wrong one. We recommend using the JEE package with the Maven plugins.

Select the directory where your pom file is. Select the project in the list, and click Next and Finish. You should now see the project in Eclipse, similar to the diagram below.




If you get build about a servlet library not being available, this should probably not matter at this point. You can remove it a bit later in the assignment after you've installed Tomcat, by adding the Tomcat runtime libraries to your project path.

**Read web.xml and pom.xml and try to understand it.**

Change your pom.xml file to the following. Note which components we are adding. This is the file that defines the build process, and dependencies. You can see that we are now adding Spring. After editing the pom.xml, select "Run As -> Maven Install" in eclipse to build the project (this may happen automatically depending on your eclipse).

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
```

```xml
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>no.uio.inf5750</groupId>
  <artifactId>assignment1</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Assignment1 Maven Webapp</name>
  <url>http://maven.apache.org</url>

  <properties>
		<spring.version>4.3.2.RELEASE</spring.version>
		<junit.version>4.12</junit.version>
		<jdk.version>1.8</jdk.version>
	</properties>

	<dependencies>

		<!-- Spring 4 dependencies -->
		<dependency>
			<groupId>org.springframework</groupId>
			<artifactId>spring-core</artifactId>
			<version>${spring.version}</version>
		</dependency>

		<dependency>
			<groupId>org.springframework</groupId>
			<artifactId>spring-web</artifactId>
			<version>${spring.version}</version>
		</dependency>

		<dependency>
			<groupId>org.springframework</groupId>
			<artifactId>spring-webmvc</artifactId>
			<version>${spring.version}</version>
		</dependency>

		<dependency>
			<groupId>junit</groupId>
			<artifactId>junit</artifactId>
			<version>${junit.version}</version>
			<scope>test</scope>
```

```
            </dependency>

            <dependency>
                  <groupId>javax.servlet</groupId>
                  <artifactId>javax.servlet-api</artifactId>
                  <version>3.1.0</version>
                  <scope>provided</scope>
            </dependency>

  </dependencies>

  <build>
      <finalName>Assignment1</finalName>
<!--
      <plugins>
         <plugin>
            <groupId>org.eclipse.jetty</groupId>
            <artifactId>jetty-maven-plugin</artifactId>
         </plugin>
      </plugins>
-->
   </build>
</project>
```

Use the ‘New source folder’ item to create the directory "/src/main/java" if it doesn’t exist already.

Adding a source folder may be a bit tricky if the above gives the error: “the folder is already a source folder”. Then you have to add the folder using "File->New->New Folder", select Assignment1 as the project and type "/src/main/java". Further, you may need to manually assign the java folder for the compiler, inside "project properties -> Java Build path" -> "Add folder" (add /src/main/java).

Create a package no.uio.inf5750.Assignment1.controller inside that folder. Do this by right-clicking the src/main/java in the project explorer and selecting "New->Package"

The user interface is placed in *.jsp files, while the user interface controller logic is placed in java classes that are annotated using the @Controller annotation. You will now create such a controller.

Create a file BaseController.java inside the controller Java package, with the contents

below.

@RequestMapping("/") maps the root url to this method. So for example, the URL to access this method from a web browser could be:
http://localhost:8080/Assignment1-username/

@RequestMapping("/welcome") maps the /welcome url to this method.

/src/main/java/no/uio/inf5750/Assignment1/controller/BaseController.java

```
package no.uio.inf5750.assignment1.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.PathVariable;


@Controller
public class BaseController {

	@RequestMapping(value="/", method = RequestMethod.GET)
	public String welcome(ModelMap model) {

		model.addAttribute("message", "Maven Web Project + Spring 4 MVC -
welcome()");

		//Spring uses InternalResourceViewResolver and return back index.jsp
		return "index";

	}

	@RequestMapping(value="/welcome/{name}", method = RequestMethod.GET)
	public String welcomeName(@PathVariable String name, ModelMap model) {

		model.addAttribute("message", "Maven Web Project + Spring 4 MVC - " +
name);
		return "index";

	}
```

```
}
```

Create a Spring configuration file called mvc-dispatcher-servlet.xml (see below).

Make sure your context:component-scan path in the file matches the name of your package where your controller is. The component-scan is the tag that tells Spring where to looks for classes with annotations such as @Controller in the file above. It will scan sub-directories.

The file below also defines that the JSP files will be in a different sub-directory where they are not directly accessible by a browser. You'll have to create this folder and put the JSP files there later. There are several alternatives to using Java Server Pages (JSP) to implement the presentation logic. You will also learn about Velocity as a JSP alternative in the course.

/src/main/webapp/WEB-INF/mvc-dispatcher-servlet.xml:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-4.3.xsd">

       <context:component-scan base-package="no.uio.inf5750.assignment1" />

       <bean
              class="org.springframework.web.servlet.view.InternalResourceViewResolver">
              <property name="prefix">
                     <value>/WEB-INF/pages/</value>
              </property>
              <property name="suffix">
                     <value>.jsp</value>
              </property>
       </bean>

</beans>
```

Update existing web.xml to support Servlet 2.5 (the default Servlet 2.3 is too old), and also integrates Spring framework into this web application project via Spring's listener

ContextLoaderListener. *If you get an error “Cannot change version of project facet Dynamic Web Module to 2.5.” then check [this solution] here.*

This is a good time to google or read in your spring-book about the DispatcherServlet.

Note that the location of the above xml-file (mvc-dispatcher-servlet.xml) is configured into this file.

/src/main/webapp/WEB-INF/web.xml (replace the existing file with this content)

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
	     http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
	version="2.5">
	<display-name>Counter Web Application</display-name>
	<servlet>
		<servlet-name>mvc-dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
		<load-on-startup>1</load-on-startup>
	</servlet>
	<servlet-mapping>
		<servlet-name>mvc-dispatcher</servlet-name>
		<url-pattern>/</url-pattern>
	</servlet-mapping>
	<context-param>
		<param-name>contextConfigLocation</param-name>
		<param-value>/WEB-INF/mvc-dispatcher-servlet.xml</param-value>
	</context-param>
	<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
	</listener>
	<filter>
		<filter-name>encodingFilter</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
		<init-param>
			<param-name>encoding</param-name>
			<param-value>UTF-8</param-value>
		</init-param>
	</filter>
```

```
	<filter-mapping>
		<filter-name>encodingFilter</filter-name>
		<url-pattern>/*</url-pattern>
	</filter-mapping>

</web-app>
```

Move the existing index.jsp inside folder WEB-INF/pages, to protect user access it directly. In additional, edit the file to print out the ${message} variable that is passed by the controller.

/src/main/webapp/WEB-INF/pages/index.jsp

```
<html>
<body>
<h2>Hello World!</h2>

<h4>Message : ${message}</h4>
</body>
</html>
```

Review the final directory structure.

Create directory ~/opt/tomcat. Download the Tomcat zip file from [http://tomcat.apache.org/download-90.cgi](http://tomcat.apache.org/download-90.cgi) and unzip it in the new directory.

Select Run On Server -> Apache > Tomcat v9.0

Click on Browse and go to ~/opt/tomcat/apache-tomcat-9.0.0.M9

Check that you get a web page in your web browser. Likely url is something like

http://localhost:8080/assignment1/

and

http://localhost:8080/assignment1/welcome/john (the url may be different from Assignment1, including _<username> depending on what you've called your project)

Now edit the BaseController.java class, adding a new method called helloWorld, which maps to the url "/hello" and takes one url parameter as a input (similar to welcomeName).

Also add an additional JSP-file, called ‘hello.jsp’.

Edit these two files so that the following URL prints “Hello <name> and hello world.”

Now edit index.jsp and hello.jsp so that they have links between them.

If you want to, add your own personal touch to the index.jsp and hello.jsp files. Edit them, create some more @RequestMapping for paths within the same BaseController class, compile it and test that it works.

You should now have three URLs that respond, where <name> is any word that then gets printed on the page.

http://localhost:8080/assignment1_<userid>/hello/<name>

http://localhost:8080/assignment1_<userid>/welcome/<name>

http://localhost:8080/assignment1_<userid>/

## Hibernate

In this part of the assignment, you’ll be adding database access to your new web application. For this purpose, we will be using Hibernate, which is a framework that translates between a relational database and Java objects. The framework handles a lot of the details of doing database work, so that you can focus on making great web applications. In this assignment, we will be using only one table and one object mapping, but Hibernate can be used to represent quite advanced java object models into a relational database.

Hibernate is used a lot both in open source projects and in professional companies, so knowing how to use it will be useful for you later on. It is also used in DHIS2.

First let us update the pom.xml file, so that Maven knows which libraries to download and compile with.

/pom.xml (you may want to change to assignment1_<username> if you’ve used this above)

**Remember:** To run mvn install after updating pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>no.uio.inf5750</groupId>
```

```
        <artifactId>assignment1</artifactId>
        <packaging>war</packaging>
        <version>1.0-SNAPSHOT</version>
        <name>assignment1 Maven Webapp</name>
        <url>http://maven.apache.org</url>

        <properties>
                <spring.version>4.3.2.RELEASE</spring.version>
                <junit.version>4.12</junit.version>
                <jdk.version>1.8</jdk.version>
                <hibernate.version>5.2.2.Final</hibernate.version>
        </properties>

        <repositories>
                <repository>
                        <id>JBoss</id>
                        <url>https://repository.jboss.org/nexus/content/groups/public/</url>
                </repository>
        </repositories>

        <dependencies>

                <dependency>
                        <groupId>junit</groupId>
                        <artifactId>junit</artifactId>
                        <version>${junit.version}</version>
                        <scope>test</scope>
                </dependency>


    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
        <scope>provided</scope>
    </dependency>

                <!-- Spring -->

                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-core</artifactId>
                        <version>${spring.version}</version>
                </dependency>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-context</artifactId>
                        <version>${spring.version}</version>
                </dependency>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-tx</artifactId>
                        <version>${spring.version}</version>
```

```
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-orm</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>${spring.version}</version>
        </dependency>

        <!-- Hibernate -->

        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>${hibernate.version}</version>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-entitymanager</artifactId>
            <version>${hibernate.version}</version>
        </dependency>
        <dependency>
            <groupId>geronimo-spec</groupId>
            <artifactId>geronimo-spec-jta</artifactId>
            <version>1.0-M1</version>
        </dependency>
        <dependency>
            <groupId>c3p0</groupId>
            <artifactId>c3p0</artifactId>
            <version>0.9.1.2</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>1.5.8</version>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <version>1.2.136</version>
        </dependency>
        <dependency>
            <groupId>postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <version>8.4-701.jdbc4</version>
```

```
                    </dependency>

            </dependencies>

              <build>
        <finalName>assignment1</finalName>
<!--
        <plugins>
            <plugin>
                <groupId>org.eclipse.jetty</groupId>
                <artifactId>jetty-maven-plugin</artifactId>
            </plugin>
        </plugins>
-->
    </build>

</project>
```

You need to implement the Java objects that represent the database content. In this assignment, this object will be called Message.java. Create the following file:

/src/main/java/no/inf5750/assignment1/model/Message.java

```
package no.uio.inf5750.assignment1.model;

public class Message
{
    private int id;

    public Message()
    {
    }

    public int getId() {
                    return id;
            }
            public void setId(int id) {
                    this.id = id;
            }
            public String getContent() {
                    return content;
            }
            public void setContent(String content) {
                    this.content = content;
            }

private String content;

}
```

You have to tell Hibernate how to map this Java object Message.java into the database. This can be done using an XML file or by using annotations inside the Java object. In this example we'll be using an XML file. Hibernate has many functions to allow various mappings of Java objects into the relational tables, but we'll be using a pretty simple one. Create the following XML file. It basically says that the no.uio.inf5750.assignment1.model.Message object maps to a table called message, and that this table has the values 'id' and 'content'.

/src/main/resources/hibernate/Message.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="no.uio.inf5750.assignment1.model.Message" table="message">
    <id name="id" column="id">
      <generator class="native"/>
    </id>
    <property name="content" not-null="true"/>
  </class>
</hibernate-mapping>
```

You then have to create some classes that you use to access the database. These are Database Access Objects (DAO). Because you may want to replace Hibernate with other database layers, or even use files to store the data, you will first create an interface that represents the data storage. You can see that this interface has methods for saving a message, getting a message of a certain id and getting the last saved message.

/src/main/java/no/inf5750/assignment1/dao/MessageDao.java

```
package no.uio.inf5750.assignment1.dao;

import no.uio.inf5750.assignment1.model.Message;

public interface MessageDao
{
    int save( Message message );
    Message get( int id );
    Message getLast();
}
```

Then you implement a java object that implements actual methods for accessing Hibernate to do these methods. You don't have to understand all the details of this yet, as it hasn't been covered in lectures, but look at the class and try to understand what's happening.

/src/main/java/no/inf5750/assignment1/dao/HibernateMessageDao.java

```
package no.uio.inf5750.assignment1.dao;

import java.util.List;
import no.uio.inf5750.assignment1.model.Message;
import org.apache.log4j.Logger;
import org.hibernate.HibernateException;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.Session;
import org.springframework.transaction.annotation.Transactional;


@Transactional
public class HibernateMessageDao
  implements MessageDao
{

    static Logger logger = Logger.getLogger(HibernateMessageDao.class);
    private SessionFactory sessionFactory;

    public void setSessionFactory( SessionFactory sessionFactory )
    {
        this.sessionFactory = sessionFactory;
    }

    public int save( Message message )
    {
        return (Integer) sessionFactory.getCurrentSession().save( message );
    }

    public Message get( int id )
    {
        return (Message) sessionFactory.getCurrentSession().get( Message.class, id );
    }

    public Message getLast()
    {
        Session session = sessionFactory.openSession();
        Transaction tx = null;
        Message message = null;
        try{
          tx = session.beginTransaction();
          @SuppressWarnings("unchecked")
          // This is an HQL query, not an SQL query (HQL is based on SQL, but is not 100% the same)
          List<Message> messages = session.createQuery("FROM Message ORDER by id DESC").list();
          if (!messages.isEmpty()) {
               message = messages.iterator().next();
          }
          tx.commit();
        }catch (HibernateException e) {
          if (tx!=null) tx.rollback();
          logger.error("DB query failed", e);
        }finally {
```

```
        session.close();
      }
      return message;
    }
}
```

You now have an updated pom.xml file, a model object, a Hibernate mapping file, an interface for accessing the database and an implementation that actually does the work of accessing the database.

You now have to tell Spring how to insert the Hibernate objects you just created into the application. In this assignment we will be using an in-memory database called H2. It is a good way to develop and test applications, but for real live webapps, you'd want to use something else like Postgres. H2 can also store data in files, if you want a lightweight database for development that is persistent across runs.

In the xml file below we are also configuring some HIbernate properties, by using Spring to inject these properties into LocalSessionFactoryBean.

Also note that we are using Spring to tell HibernateMessageDao where it can find the SessionFactory that it uses to access Hibernate. You can see in the Java code of HibernateMessageDao that it looks like HibernateMessageDao is not set. In fact, Spring sets this property when it creates the HibernateMessageDao object. This config is into bottom of the file below.

/src/main/webapp/WEB-INF/mvc-dispatcher-servlet.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:tx="http://www.springframework.org/schema/tx"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.3.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-4.3.xsd">

        <context:component-scan base-package="no.uio.inf5750.assignment1" />

        <bean
                class="org.springframework.web.servlet.view.InternalResourceViewResolver">
                <property name="prefix">
                        <value>/WEB-INF/pages/</value>
                </property>
                <property name="suffix">
                        <value>.jsp</value>
                </property>
        </bean>

  <!-- Hibernate-part of assignment 1 -->
```

```
  <tx:annotation-driven transaction-manager="transactionManager"/>

  <bean id="transactionManager"
class="org.springframework.orm.hibernate5.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
    <property name="dataSource" ref="dataSource"/>
  </bean>

  <bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="mappingResources">
            <list>
                    <value>hibernate/Message.hbm.xml</value>
            </list>
    </property>
    <property name="hibernateProperties">
            <props>
              <prop key="hibernate.dialect">org.hibernate.dialect.H2Dialect</prop>
                    <!--
        <prop key="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</prop>
        -->
                    <prop key="hibernate.hbm2ddl.auto">create-drop</prop>
            </props>
    </property>
  </bean>

  <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
destroy-method="close">

    <property name="driverClass" value="org.h2.Driver"/>
    <property name="jdbcUrl" value="jdbc:h2:file:inf5750;DB_CLOSE_ON_EXIT=FALSE"/>
          <property name="user" value="sa"/>
          <property name="password" value=""/>

    <!-- The postgres configuration is commented out. Left here as a hint for future assignments -->
    <!--
    <property name="driverClass" value="org.postgresql.Driver"/>
    <property name="jdbcUrl" value="jdbc:postgresql:product"/>
          <property name="user" value="sa"/>
          <property name="password" value="sa"/>
          -->
  </bean>

  <bean id="messageDao" class="no.uio.inf5750.assignment1.dao.HibernateMessageDao">
    <property name="sessionFactory" ref="sessionFactory"/>
  </bean>

</beans>
```

You should also create a log4j property file. You can use this one as a template:

/src/main/resources/log4j.properties

```
# Configuration file for log4j

# Log to file setup
log4j.appender.file = org.apache.log4j.RollingFileAppender
log4j.appender.file.File = nf.log
log4j.appender.file.MaxFileSize = 100KB
log4j.appender.file.MaxBackupIndex = 3
log4j.appender.file.layout = org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern = * %-5p %d{ABSOLUTE} %m (%F [%t])%n

# Log to console setup
log4j.appender.console = org.apache.log4j.ConsoleAppender
log4j.appender.console.layout = org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern = * %-5p %d{ABSOLUTE} %m (%F [%t])%n

# Categories (order: DEBUG, INFO, WARN, ERROR, FATAL)
log4j.rootCategory = WARN, console
log4j.category.no.uio.inf5750 = INFO
```

Now you have set up everything you need to access the database, and the last thing you need to do is to create a Controller file and JSP file that actually uses the database and presents it well as a web page. It would be good if you can create your own **personal touch** to these two files, but you can use the below files as a starting point. Make sure your application can both send and receive messages. You can't just copy straight from the below box, since you'll lose some of the changes you did on your own earlier.

/src/main/webapp/WEB-INF/pages/index.jsp (you may need to edit the URL the message is submitted to)

```
<html>
<body>
<h2>Leave a message</h2>

<h1>${message}</h1>

<form name="input" action="/assignment1/send" method="get">
Message content: <input type="text" name="content">
<input type="submit" value="Submit">
</form>

<p><a href="/assignment1/read">Get last message</a></p>
</body>
</html>
```

You can see that the below file has a Spring annotations called @Autowired. This annotation takes the only implementation of the MessageDao interface and puts it into the messageDao variable. You can then use

messageDao to send and receive messages (save and retrieve from the database).

Don’t get confused by the variable called ‘model’. This is a variable used by the Spring MVC to give data to the JSP page, not a variable that’s part of the database model.

/src/main/java/no/uio/inf5750/assignment1/controller/BaseController.java (don’t just copy this file, since you’ll lose your earlier helloWorld work. Merge it with the earlier work.)

```
package no.uio.inf5750.assignment1.controller;

import no.uio.inf5750.assignment1.dao.MessageDao;
import no.uio.inf5750.assignment1.model.Message;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
@RequestMapping("/")
public class BaseController {

	@Autowired
	private MessageDao messageDao;

	/*
	 * Intro page. No database interaction.
	 */
	@RequestMapping(value="/", method = RequestMethod.GET)
	public String welcome(ModelMap model) {

		model.addAttribute("message", "Leave a message using the form");

		//Spring uses InternalResourceViewResolver and return back index.jsp
		return "index";
	}

	/*
	 * Saves a message in the database
	 */
	@RequestMapping(value="/send", method = RequestMethod.GET)
	public String send(@RequestParam(value="content") String content, ModelMap model) {

		Message msg = new Message();
		msg.setContent(content);
		int id = messageDao.save(msg);
		model.addAttribute("message", "Message id of stored message=" + id);
		return "index";

	}
```

```
	/*
	 * Prints the last message entered
	 */
	@RequestMapping(value="/read", method = RequestMethod.GET)
	public String read(ModelMap model) {

		Message message = messageDao.getLast();
		if (message != null) {
			model.addAttribute("message", "The last message was: 
"+message.getContent());
		}
		else {
			model.addAttribute("message", "No message found");
		}

		//Spring uses InternalResourceViewResolver and return back index.jsp
		return "index";
	}

	/*
	 * Prints the message with a given id
	 */
	@RequestMapping(value="/read/{id}", method = RequestMethod.GET)
	public String readId(@PathVariable int id, ModelMap model) {

		Message message = messageDao.get(id);
		if (message != null) {
			model.addAttribute("message", "Message number "+id+" was: 
"+message.getContent());
		}
		else {
			model.addAttribute("message", "No message found");
		}

		//Spring uses InternalResourceViewResolver and return back index.jsp
		return "index";
	}

}
```

Test that your code compiles and runs ok. You should be getting a web page in your browser that lets you send and receive messages.

If you are getting the following error: “java.lang.ClassNotFoundException: org.springframework.web.context.ContextLoaderListener” have a look at this solution: [http://stackoverflow.com/questions/6210757/java-lang-classnotfoundexception-org-springframework-web-context-contextloaderl](http://stackoverflow.com/questions/6210757/java-lang-classnotfoundexception-org-springframework-web-context-contextloaderl) (the one involving setting Deployment assembly to Maven dependencies).

If you would like to inspect the database that Hibernate has generated, go to

http://www.h2database.com/html/main.html and download the H2 Browser based Console application. This will run on port 8082 and allow you to query the structure.

You can use the default username ("sa") and no password, just have to set the correct path to the db file. To locate your file, you run this on the command line: find / -iname *.h2.db

Then your path will be jdbc:h2:file:<PATH>/inf5750 (inf5750.h2.db is the name of the file)

**You need to go to https://devilry.ifi.uio.no/ and upload your zipped project into the Assignment1. Talk to your group teacher if this doesn't work (we're setting up Devilry, so it may not be 100% operational yet).**

You are now finished with the assignment1, but it would be useful for you to look through the files and make sure you understand them. It would be very good learning for you to research on your own how to move this onto using Postgres instead of the H2 in-memory database.