#Video1 (Why MVC...)

**Servlet** - In the java code the HTML code is embeded. is one ot the first technology providing dynamically generated content to the user. The problem was that the servlet class extending HttpServlet had to contain all the logic and provide HTML code needed to generete HTML file. That was a bit messy.

**JSP** - In the hmtl code the java code is embeded. That was an alternative to the Servlets. One of the problew was to write the whole sql sequences connecting to the DB (<sql:setDataSource var="" driver="" url="" user="" password="".>, <sql:query dataSource="" var="">SELECT \* FROM Employees</sql:query>).

None of above provided clear separation between business logic and HTML(HTML, Java, CSS, JavaScript, JQuery in one file!). **Problem to write, edit and maintain.**

**MVC is the solution**. It separates:

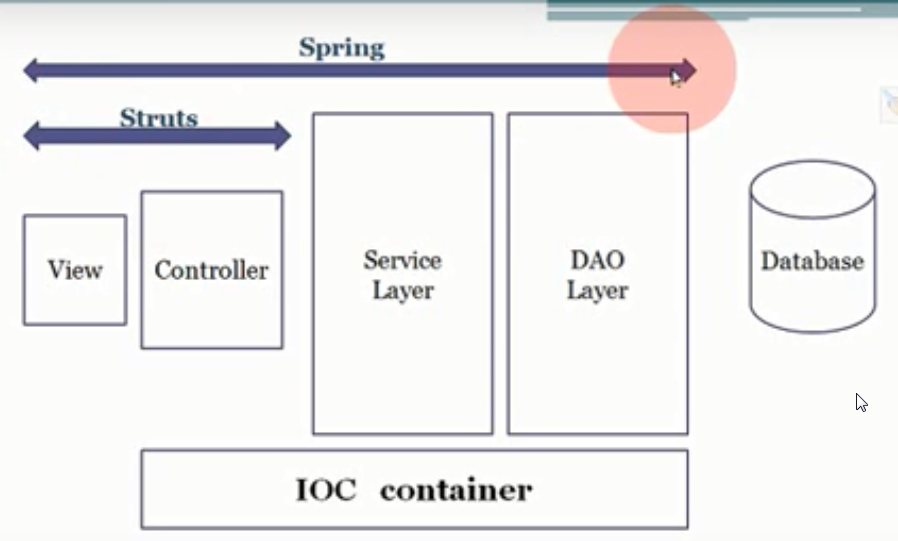
* **Model** - data to be displayed in web page: retrieving data from persistent storage
* **View** - Web page in HTML (simple JSP that contains only HTML and tag library) : HTML + CSS - presents model in user friendy interface (UI,gui)
* **Controller** - contains logic that solves domain problem

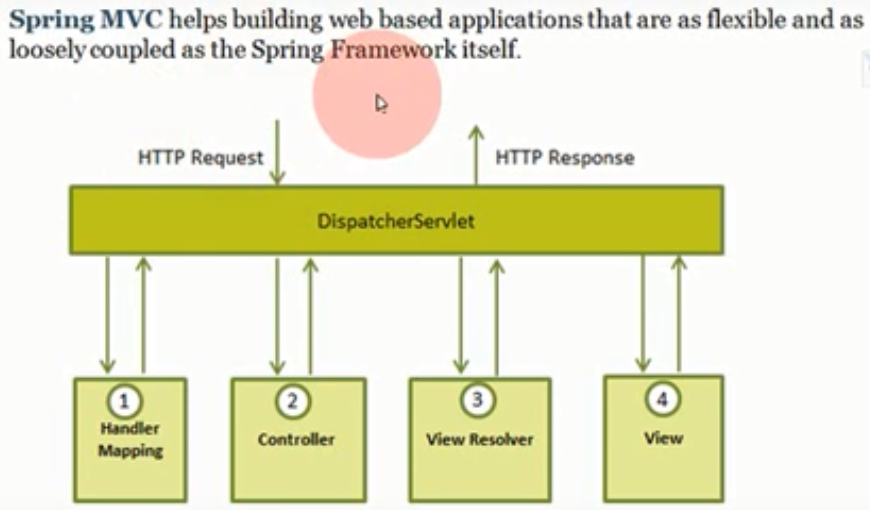
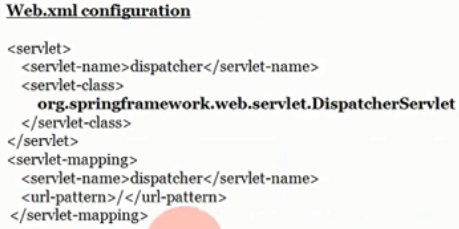
The Flow

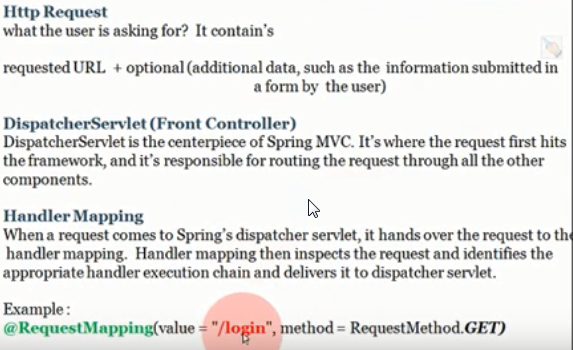
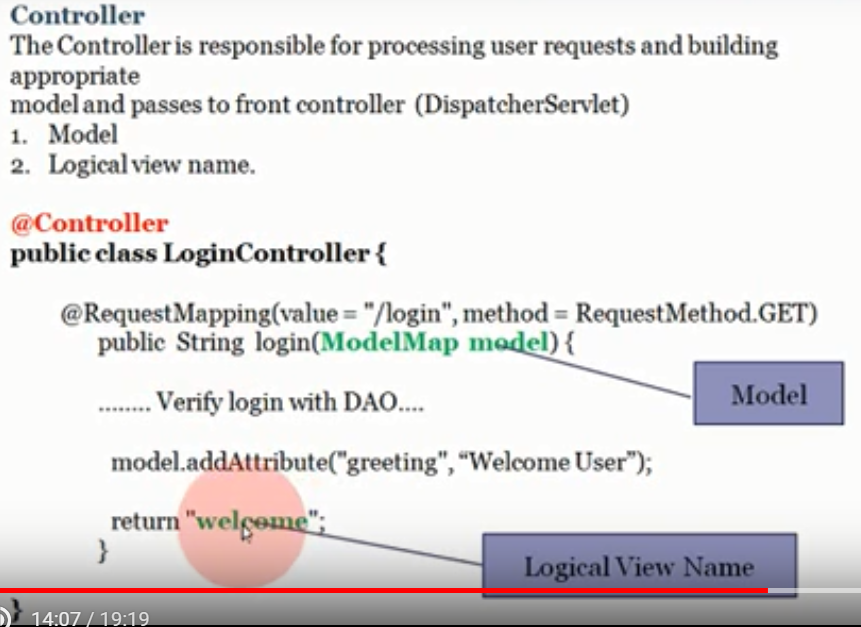
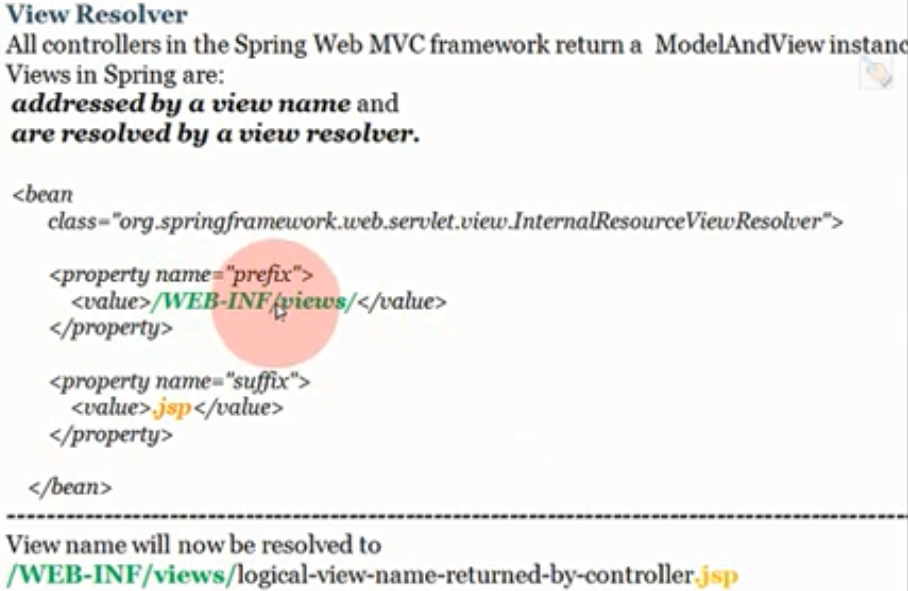
1. User request
2. Controller: prepares the model (call to model)
3. Controller: perform the logic
4. Controller: selects the view
5. Controller: sends the view to the model
6. View: renders the HTML page
7. Page is send to the client

#Video2 (Spring MVC Introduction & architecture)

**IOC Container** – Inversion of control container



**Dispatcher Servlet=Front Controller** – the center piece of the SpringMVC – it routs the information to different component. It is configured in the **Web.xml**. Below is the architecture of the spring framework.  
 

1. **Handler mapping** – selecting a controller that will handle the client request and passing it to the controller. Each request must match to unique **mapping** of a controller. 
2. **Controller** – contains add/delete/update/find. It processes the request, prepares a **model** and passes it to the front controller as well and the logical **view** name.  
   
3. **View resolver** – is a bean in the spring context configuration file. It adds prefix and suffix to build the path to the **view** (that is an instance of ModelAndView.class) and returns it to the front controller.
4. The view and the motel together can generate a HTML that is returned to the client

#video3 (Creating Environment for Spring MVC Using MAVEN)

Creating HelloWorldXml (xml in the name to indicate one of xml or java configuration)

1. Create Dynamic Web Project
   1. Configure the server (choose the server, point to the server installation directory,)
   2. Change source folder on build path to match maven structure. Remove “src” and add:
      1. “src/main/java”
      2. “src/main/resources” for static resources
      3. “src/main/webapp” for views
      4. “src/test/java”
   3. Change the content directory to “src/main/webapp” check generate web.xml.
2. Convert to maven project
   1. **GroupId** - is a base package name, where all the classes will be saved
   2. **ArtifactId** – is the project name
   3. **Version** – in the version of the project

Add dependencies to pom.xml  
 <properties>

<springframework.version>4.2.5.RELEASE</springframework.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>${springframework.version}</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>3.1.0</version>

</dependency>

<dependency>

<groupId>javax.servlet.jsp</groupId>

<artifactId>javax.servlet.jsp-api</artifactId>

<version>2.3.1</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

</dependencies>

#video 4 (Spring MVC Hello World Application)

1. Configure **web.xml** be defining a servlet with definition of **(1)DispatcherServlet**, indication to **(2)spring context configuration** (file location, it contains **view resolver** definition) in parameter and **(3)servlet mapping** for all strings after slash “/”.

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <web-app 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/web-app\_3\_1.xsd"*  id=*"WebApp\_ID"* version=*"3.1"*>  <display-name>HelloWorldXml</display-name>  <servlet>  <servlet-name>dispatcher</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>/WEB-INF/spring-servlet.xml</param-value>  </init-param>  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>dispatcher</servlet-name>  <url-pattern>/</url-pattern>  </servlet-mapping>  </web-app> |

1. Create a Controller in a package for conrollers **org.greysalmon.controller.HelloWorldController.java**
   1. @Controller stereotype annotation
   2. First public **handler** method that returns view name as a string and model, that is the data required by the view.
   3. Second handler method with different mapping

|  |
| --- |
| package org.greysalmon.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;  //stereotype annotation  @Controller  public class HelloWorldController {    // It supports only GET request  @RequestMapping(value="/", method=RequestMethod.GET)  public String sayHello(ModelMap model){    // it is a map, so it takes string as a key and the value, that might be any object  model.addAttribute("message", "Welcome form spring MVC");    // this is the logical view name, that will later go through the view resolver  return "welcome";  }    // providing another mapping for unique identifying the handler method  @RequestMapping(value="/helloAgain", method=RequestMethod.GET)  public String sayHelloAgain(ModelMap model){    // it is a map, so it takes string as a key and the value, that might be any object  model.addAttribute("message", "Welcome Again form spring MVC");    // this is the logical view name, that will later go through the view resolver  return "welcome";  }  } |

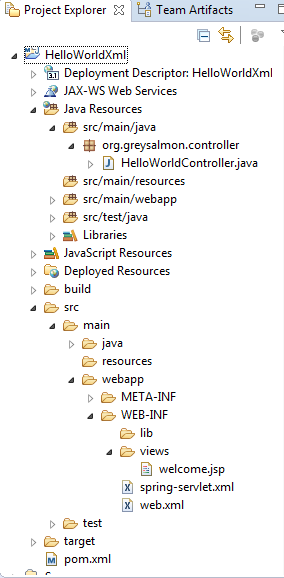
1. View Resolver definition in the context configuration **Spring-servlet.xml** file
   1. Has the name defined in web.xml file
   2. Contains beans required by the IOC container (view resolver)
   3. View Resolver is a bean itself defined in it and it will be loaded into IOC Container
   4. Define annotation driven configuration
   5. Define the location scanned in order to find components(beans)

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:mvc=*"http://www.springframework.org/schema/mvc"* 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-4.0.xsd*  *http://www.springframework.org/schema/mvc*  *http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context-4.0.xsd"*>  <!-- the schema might be found in the cods in the jar/docs/spring-framework-reference/html/ -->  <!-- define location where to scan for components -->  <context:component-scan base-package=*"org.greysalmon.controller"*/>    <!-- to define that the components are annotation driven -->  <mvc:annotation-driven/>  <bean  class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>  <property name=*"prefix"*>  <value>/WEB-INF/views/</value>  </property>  <property name=*"suffix"*>  <value>.jsp</value>  </property>  </bean>  </beans> |

1. **view.jsp** HTML containing some java
   1. charset=ISO-8859-1
   2. reference to the data from model provided by the controller by ${message}

|  |
| --- |
| <%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"*  pageEncoding=*"ISO-8859-1"*%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>  <title>Insert title here</title>  </head>  <body>  <h1>Hello from spring framework</h1>    <h3>${message}</h3>  </body>  </html> |

1. deploy the app on a server
   1. Download and install tomcat
   2. Choose jre location
   3. Wondows+R ->services.msc->tomcat.properties.startupType:manual
   4. run as -> run on server
   5. go to <http://localhost:8080/HelloWorldXml/> or <http://localhost:8080/HelloWorldXml/helloAgain>

That’s the project tree.

#video 5 (Spring MVC Get User Data With **@PathVariable**)

Getting data appended to the URL by using **@PathVariable** annotation.

e.g. http://localhost:8080/HelloWorldXml/home/**james**

URL is case sensitive!

|  |
| --- |
| @RequestMapping(**value = "/home/{user}",** method = RequestMethod.*GET*)  // getting a value form URL: declare variable matching the placeholder in  // the mapping and then the type and the java variable  public String sayHelloAgain**(@PathVariable("user") String user**, ModelMap model) {  model.addAttribute("message", "Welcome Again form spring MVC");  // adding the user variable to the model, so that it can be access in  // the .jsp by the attribute name  **model.addAttribute("userData", user);**  return "welcome";  } |
| In **.jsp** simple reference to the model by the attribute name provided while adding to the model in the controller: **<h2>Welcome ${userData}</h2>** |

#video 6 (Spring MVC Get User Data With **@RequestParam**)

Retrieving the data provided in the form by using **@RequestParam.**

1. New Controller

|  |
| --- |
| **package** org.greysalmon.controller;  **import** java.util.Map;  **import** org.springframework.stereotype.Controller;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RequestMethod;  **import** org.springframework.web.bind.annotation.RequestParam;  @Controller  **public** **class** UserController {  @RequestMapping(value = "/user/register", method = RequestMethod.***GET***)  **public** String register(Map<String, Object> map) {  // map.put("title", "User Registration");  **return** "user/register";  }  // method for handling the user data  @RequestMapping(value = "/user/details", method = RequestMethod.***POST***)  **public** String detils(@RequestParam("fName") String fName, @RequestParam("lName") String lName,  @RequestParam("email") String email, @RequestParam("mobile") String mobile, Map<String, Object> map) {  // putting all values form the form to a map that will be passed to the  // jsp  map.put("fName", fName);  map.put("lName", lName);  map.put("email", email);  map.put("mobile", mobile);  **return** "user/details";  }  } |

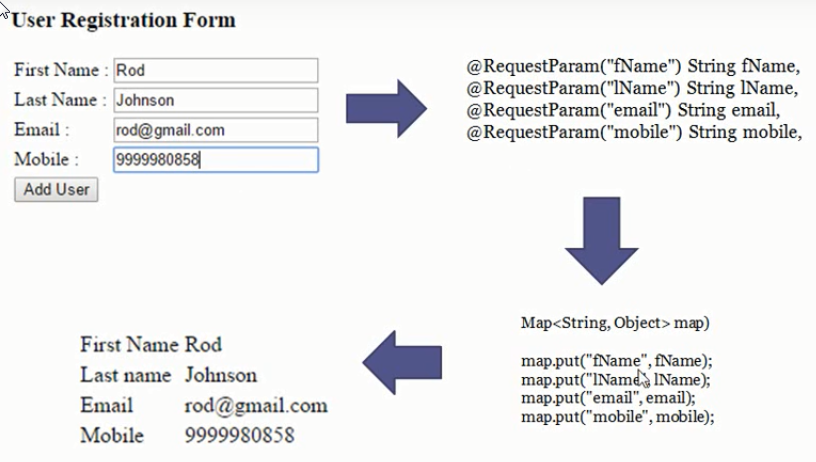
1. New jsp with a form; after the form is submitted the controller must handle the data.

|  |
| --- |
| <body>  <h3>User registration form</h3>  <form method=*"post"* action=*"/HelloWorldXml/user/details"*>  <!-- after clicking on the submit button the data from the form is  handed over to the controller that matches the request mapping from  the action, this method will add data to the model and pass it to  another jsp that displays it -->  <table>  <tr>  <td>First Name :</td>  <td><input type=*"text"* name=*"fName"* /></td>  </tr>  <tr>  <td>Last Name:</td>  <td><input type=*"text"* name=*"lName"* /></td>  </tr>  <tr>  <td>Email :</td>  <td><input type=*"text"* name=*"email"* /></td>  </tr>  <tr>  <td>Mobile :</td>  <td><input type=*"text"* name=*"mobile"* /></td>  </tr>  <tr>  <td colspan=*"2"*><input type=*"submit"* value=*"Add User"*></td>  </tr>  </table>  </form>  </body> |

1. Another jsp that displays the user’s data (just body provided).

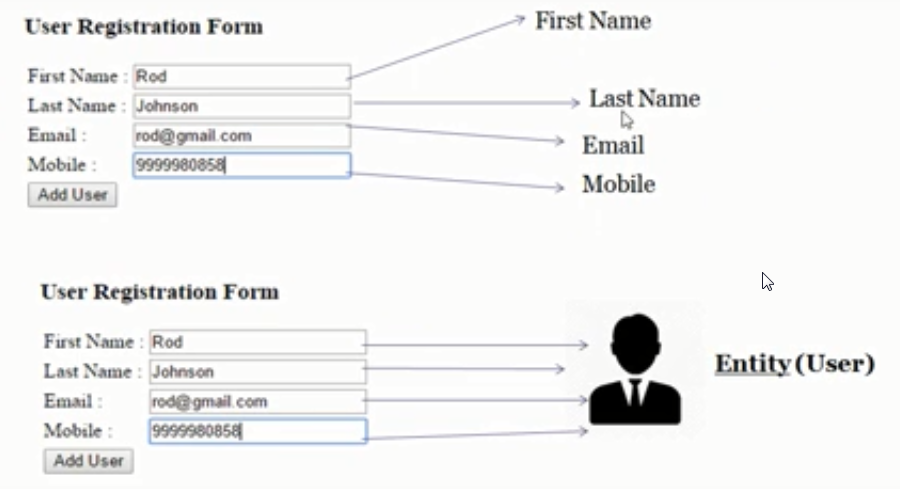
|  |
| --- |
| <body>  <table>  <tr>  <td>First Name</td>  <td>${fName}</td>  </tr>  <tr>  <td>Last Name</td>  <td>${lName}</td>  </tr>  <tr>  <td>Email</td>  <td>${email}</td>  </tr>  <tr>  <td>Mobile</td>  <td>${mobile}</td>  </tr>  </table>  </body> |

When the “submit” is being hit, the controller that matches method’s mapping to the form’s action, this handling method passes the data to the model and the to another jsp that displays it.

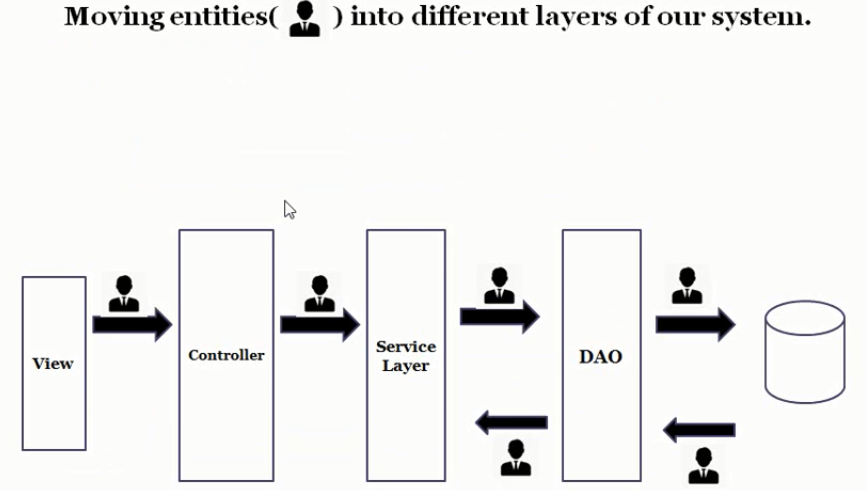


**NOT SMATR:** as many parameters, so many **@RequestParam** annotations and putting the values to the map, what is problematic when more than 1 or 2 parameters... It is smarter to use **Model** object.

#video 7 (Spring MVC Get User Data In Model Object)

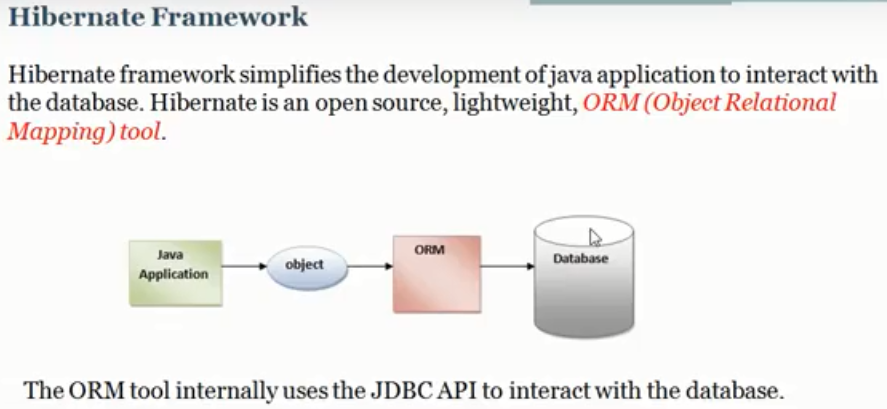
Introducting one entity that contins all the data. 

The scheme of a **Layered architecture.**

****

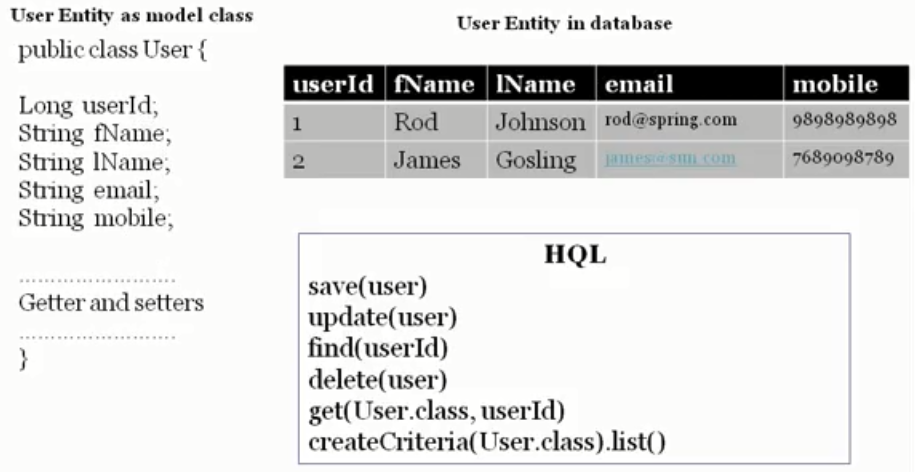
1. Create user entity with getters and setters
2. Delete all the parameters in the controller and pass the User object instead
3. The entity attributes should correspond to the form parameters (the attributes name should be the same).
4. Using getters to fetch the data from the user object and passing it to the map.
5. Move the common part “/user” of the **@RequestMapping** to the class level

#video 8 (Spring MVC Integrating Hibernate with spring)

The data must be persisted – stored in DB.

The java application is object oriented, but the DB is relational.

When the app wants to retrieve one row form the Users table it should be returned as an object -> Query the DB in an object-oriented fashion by the use of ORM flamework. Without ORM framework the code must contain SQL queries (which are DB dependent!). HQL (Hibernate Query Language) is db independent query language. **Thanks to it the migration from one db to another (e.g. oracle to MS) the cone doesn’t need any changes!**



Advantages:

1. Lightweight and open source
2. Fast performance
3. DB independent query – HQL
4. Automatic table creation

Configuration

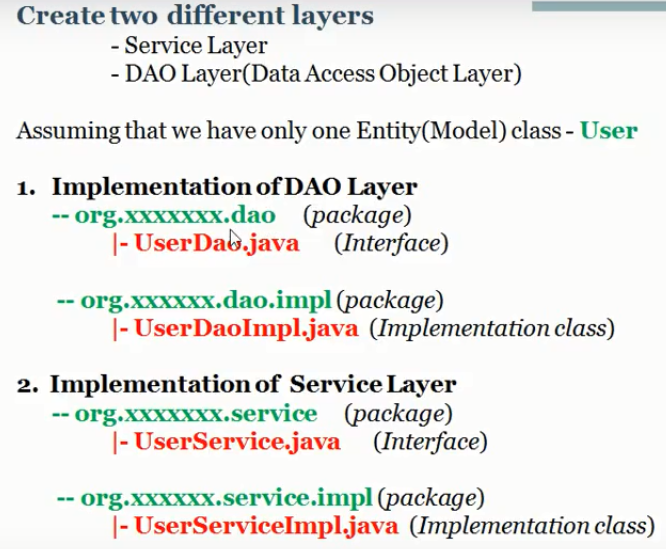
1. Add the file **database.properties** in src/main/resources  
   database.driver=com.mysql.jdbcDriver  
   database.dialect=org.hibernate.dialect.MySQLDialect  
   database.url=jdbc\:mysql\://localhost\:3306/oasis  
   database.user=root  
   database.password=root
2. In spring context configuration file:
   1. Add “**dataSource**” bean with db configuration
   2. Add “**sessionFactory**” bean as a connection object.
      1. It is used to inquire the DB = it provides the connection.
      2. It refers to the “dataSource” bean
      3. It refers to the entities
      4. The sql logs might be set to be displayed in the console
   3. Add “**transactionManager**” bean to remove pilot plain code -> It is responsible for avoiding of typing the same code: open transaction, commit… all queries will be auto-committed by the hibernate.
   4. Add “tx” **namespace** and “util” form spring framework
   5. Add model **package to be scanned** it
   6. Add <**context:property-placeholder**…/> that provides database.property, spring will look for it automatically.
3. Modify the user class
   1. Annotate with **@Entity** – it will force/cause hibernate to create a table if it won’t exist
   2. Add **id** field with annotations:
      1. **@Id**
      2. **@GeneratedValue** – the id value will be auto generated
4. Add dependencies for hibernate and SQL driver pom.xml

|  |
| --- |
| <packaging>war</packaging>  <properties>  <springframework.version>4.3.2.RELEASE</springframework.version>  </properties>  <dependencies>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  <version>${springframework.version}</version>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>javax.servlet-api</artifactId>  <version>3.1.0</version>  </dependency>  <dependency>  <groupId>javax.servlet.jsp</groupId>  <artifactId>javax.servlet.jsp-api</artifactId>  <version>2.3.1</version>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>jstl</artifactId>  <version>1.2</version>  </dependency>  **<dependency>**  **<groupId>mysql</groupId>**  **<artifactId>mysql-connector-java</artifactId>**  **<version>5.1.38</version>**  **</dependency>**  **<dependency>**  **<groupId>org.hibernate</groupId>**  **<artifactId>hibernate-core</artifactId>**  **<version>5.1.0.Final</version>**  **</dependency>**  **<dependency>**  **<groupId>org.hibernate</groupId>**  **<artifactId>hibernate-validator</artifactId>**  **<version>5.2.4.Final</version>**  **</dependency>**  **<!-- I had to add the two below, Kansal didn’t had to -->**  **<dependency>**  **<groupId>org.springframework</groupId>**  **<artifactId>spring-jdbc</artifactId>**  **<version>3.0.3.RELEASE</version>**  **</dependency>**  **<dependency>**  **<groupId>org.springframework</groupId>**  **<artifactId>spring-tx</artifactId>**  **<version>3.2.4.RELEASE</version>**  **</dependency>**  **<dependency>**  **<groupId>org.springframework</groupId>**  **<artifactId>spring-orm</artifactId>**  **<version>4.3.14.RELEASE</version>**  **</dependency>**  **<!-- it didn't work I copied the jar to the**  **c:\otherPrograms\Apache Software Foundation\Tomcat 8.0\lib\**  <dependency>  <groupId>com.oracle.weblogic</groupId>  <artifactId>ojdbc7</artifactId>  <version>12.1.3-0-0</version>  </dependency>  -->  </dependencies> |

1. Next step in hibernate-spring integration is to
   1. Implement the **Service & Data Access Layer** **(layered architecture)** in the application
   2. create CRUD operations

#video 9 (Spring MVC Creating Service and Data Access Layer)

Create 4 packages: 2 for DAO Layer(interface + impl) and 2 for Service Layer (interface + impl)



Both implements the same methods. Whenever controller(@Controller) needs to perform some business logic it calls the service(Service) that calls the ObjectDAO(@Repository) that calls the DB. Service is clear separation between controller and DAO layers

* Update/create
* Edit
* Delete
* find
* Projection – List for getting all the records in the DB.

Spring MVC - 10 (Spring MVC Inserting a record in database)

Perform the **insert=create** and **find=read** form the CRUD Goals:

1. Write the handler method
2. Create the view

* Controller: Adding backing object to the map in the handler method for the form  
  map.put("userr", **new** Userr());// the form backing object
* Controller: Injecting the UserrService (crud)
* Controller: Adding method for handling the action after the form’s submission, adding redirect in the return jsp with the userrId that will be displayed there.
* Controller: Modifying the handler for “details” jsp, so that it can **read** from db and provide the date for the view.
* JSP: **Taglib** in jsp with the form  
  <%@ taglib prefix=*"spring"* uri=*"http://www.springframework.org/tags"* %>  
  <%@ taglib prefix=*"form"* uri=*"http://www.springframework.org/tags/form"* %>  
  <%@ taglib prefix=*"c"* uri=*"http://java.sun.com/jsp/jstl/core"* %>
* JSP: Spring tags in the form, because later on validation is desired, on the submission the action will invoke one of the controller’s handling method
* Add package to scan for service and DAO in spring configuration file **spring-servlet.xml**

<http://localhost:8080/HelloWorldXml/user/register> to insert user to DB

Spring MVC - 11 (Spring MVC Displaying all records from database)

Adding handler method

|  |
| --- |
| @RequestMapping(value="/list", method= RequestMethod.***GET***)  **public** String list(Map<String, Object> map){  map.put("userrList", userrService.getAll());  **return** "user/list";  } |

Create new list.jsp to display the users

|  |
| --- |
| <%@ taglib prefix=*"spring"* uri=*"http://www.springframework.org/tags"*%>  <%@ taglib prefix=*"form"* uri=*"http://www.springframework.org/tags/form"*%>  <%@ taglib prefix=*"c"* uri=*"http://java.sun.com/jsp/jstl/core"*%>  <%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"*  pageEncoding=*"UTF-8"*%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=UTF-8"*>  <title>Insert title here</title>  </head>  <body>  <spring:url value=*"/register/"* var=*"register"* />  <a href=*"*${register}*"*>Register</a>  <c:if test=*"*${!**empty** userrList}*"*>  <table>  <tr>  <th>ID</th>  <th>First Name</th>  <th>Last Name</th>  <th>Email</th>  <th>Mobile</th>  </tr>  <c:forEach items=*"*${userrList}*"* var=*"userr"*>  <tr>  <td><c:out value=*"*${userr.userrId}*"* /></td>  <td><c:out value=*"*${userr.fName}*"* /></td>  <td><c:out value=*"*${userr.lName}*"* /></td>  <td><c:out value=*"*${userr.email}*"* /></td>  <td><c:out value=*"*${userr.mobile}*"* /></td>  </tr>  </c:forEach>  </table>  </c:if>  </body>  </html> |

<http://localhost:8080/HelloWorldXml/user/list> to see the list of the users

Spring MVC - 12 (Spring MVC Updating a record in database)

Write handler method + new view.jsp

1. Create **handler** method for editing the user, it gets userrId as parameter
2. Create **edit.jsp** that takes the userr object form the model passed by the controller and displays it, after submitting another controller will update the userr and redirect to details.jsp
3. Creating the **handler** for updating the userr and redirecting to details
4. Adding in **list.jps** new column that contains reference to update and details of the chosen userr

<http://localhost:8080/HelloWorldXml/user/list> to see the list of the users and start to editting and viewing the details.

NOTE: I’ve discovered that different browsers interpret **<a hfer=”/#”>** depending on the initial sing in the value: / -> adding the value to the base url; # -> adding value to current url ; or just providing valid url; <https://stackoverflow.com/questions/8764288/href-automatically-adds-to-current-page-url-in-php-cant-figure-it-out>

Spring MVC - 13 (Spring MVC Deleting a record from database)

Just creating controller and with redirection.

The difference is that I had to change the method in DAO Layer:

@Override

**public** **void** delete(Long userrId) {

// currentSession().delete(userrId); // this didn't work

currentSession().delete(find(userrId));

}

Spring MVC - 14 (Spring MVC Validation Part I)

1. Adding annotations form Hibernate validation jpa to entity userr class:  
   **@Email  
   @Size(min=2, max=5)**

|  |
| --- |
| @Email  **private** String email;    @Size(min=6, max=20)  **private** String mobile; |

1. In the controller to the argument list should be added **@Valid @ModelAttribute("userr")** before the user argument to indicate that is has to be validated.
2. It has to be defined what should happen when the submission doesn’t comply validation, otherwise just error will be shown like “The request sent by the client was syntactically incorrect.”. **Biding validations along with the object = bind all the errors within the model object.** by adding **BindingResult result** in the handler’s attributes. But it still doesn’t provide error message to the cient.

|  |
| --- |
| @RequestMapping(value = "/create", method = RequestMethod.***POST***)  //public String create(Userr userr, Map<String, Object> map) { //without validation  **public** String create**(@Valid @ModelAttribute("userr") Userr userr, BindingResult result**, Map<String, Object> map) {    // if the submission doesn't comply with validation redirect back to this form and the data remains in the form but no error is displayed now  **if(result.hasErrors()){**  **return "user/register";**  **}**    userrService.create(userr);  System.***out***.println("user crud: create");  map.put("userr", **new** Userr());    // to avoid double submission (when client clicks few times submit button)  **return** "redirect:/user/details/" + userr.getUserrId();  } |