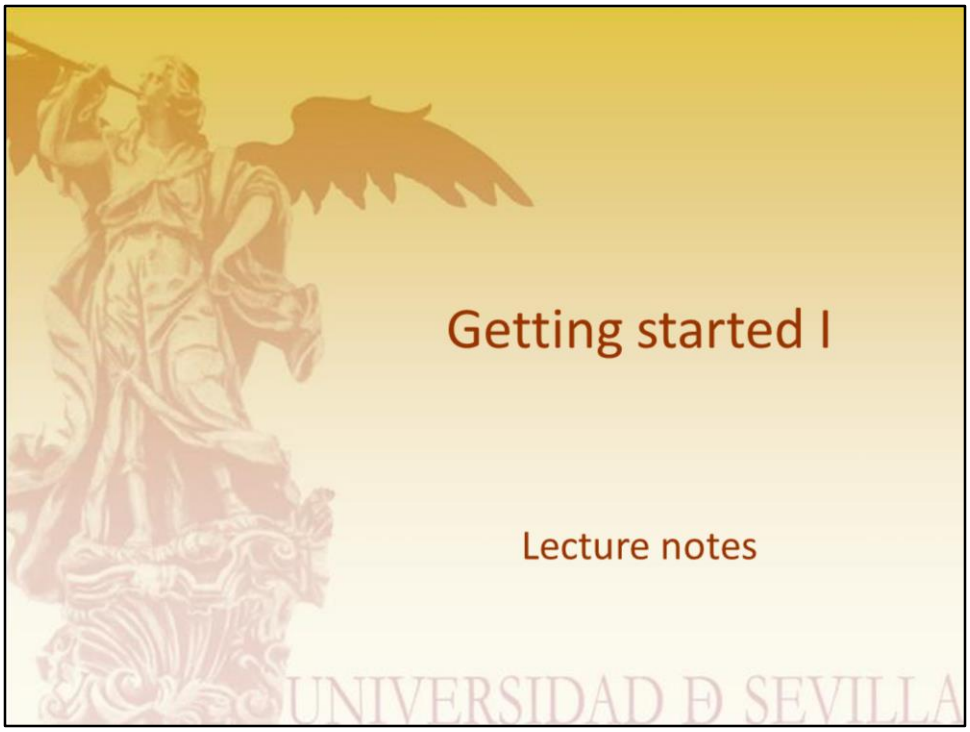
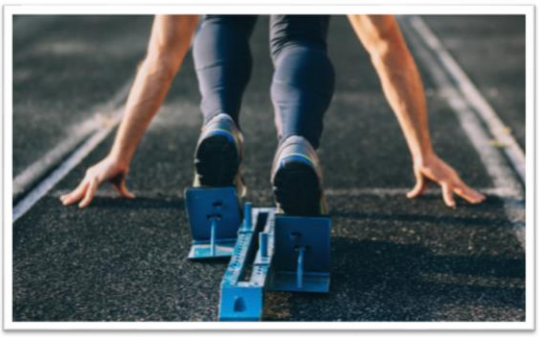


Welcome to this lesson in which our goal's to get started regarding our goal in this subject.

--

As usual, we start with a question: what's getting started about? Please, make a point of answering this question before you go on.

This is a good definition



It's about learning the foundations so that you can begin producing a web information system

UNIVERSIDAD Ð SEVILLA

3

Simply put, getting started means learning the foundations so that you can begin producing a web information system.

How can you get started?

UNIVERSIDAD Ð SEVILLA

4

This is also a god question: how can you get started? Please, try to produce your own answer without casting any glances at the following slides.

Step 1: preliminaries

UNIVERSIDAD Ð SEVILLA

5

The first step is to learn some preliminaries, that is, a few concepts that will be used throughout the lesson

Step 2: case study 1

Then we'll introduce a simple case study in which we'll produce a system that outputs some inspiring quotes.

Step 3: case study 2

UNIVERSIDAD Đ SEVILLA

7

We'll then go a bit further by producing a system that implements a simple calculator.

Step 4: case study 3

The next case study will be a system that lists some entities that are retrieved from the database.

Step 5: case study 4

UNIVERSIDAD D SEVILLA

9

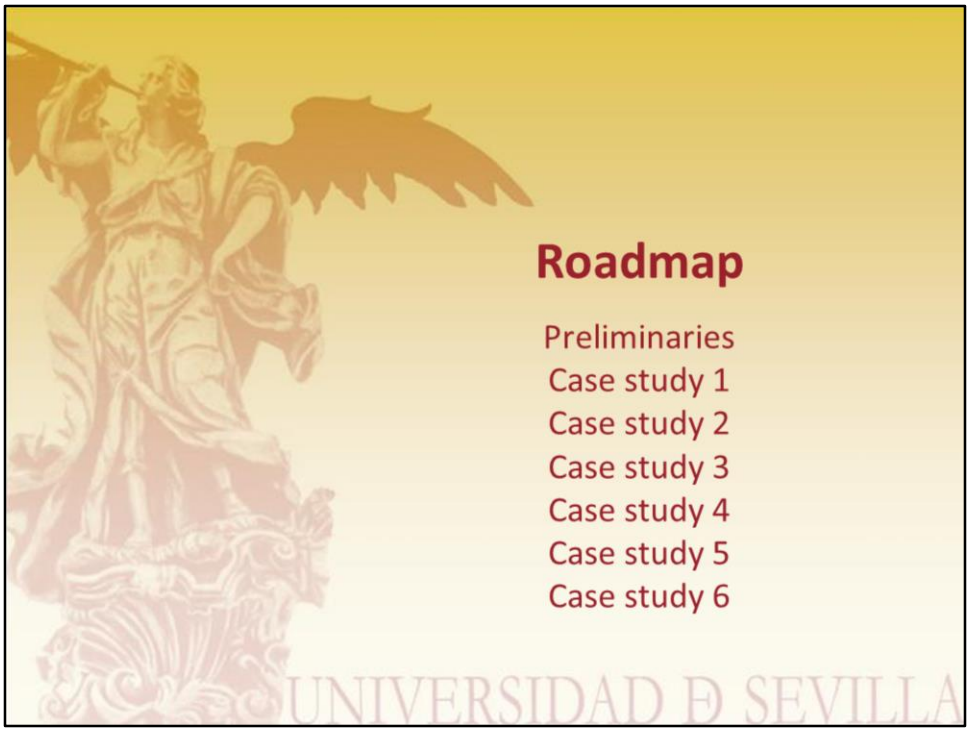
Then, we'll explore how to create an input form so that the user can create the entities that are shown in the previous listing.

Step 6: case study 5

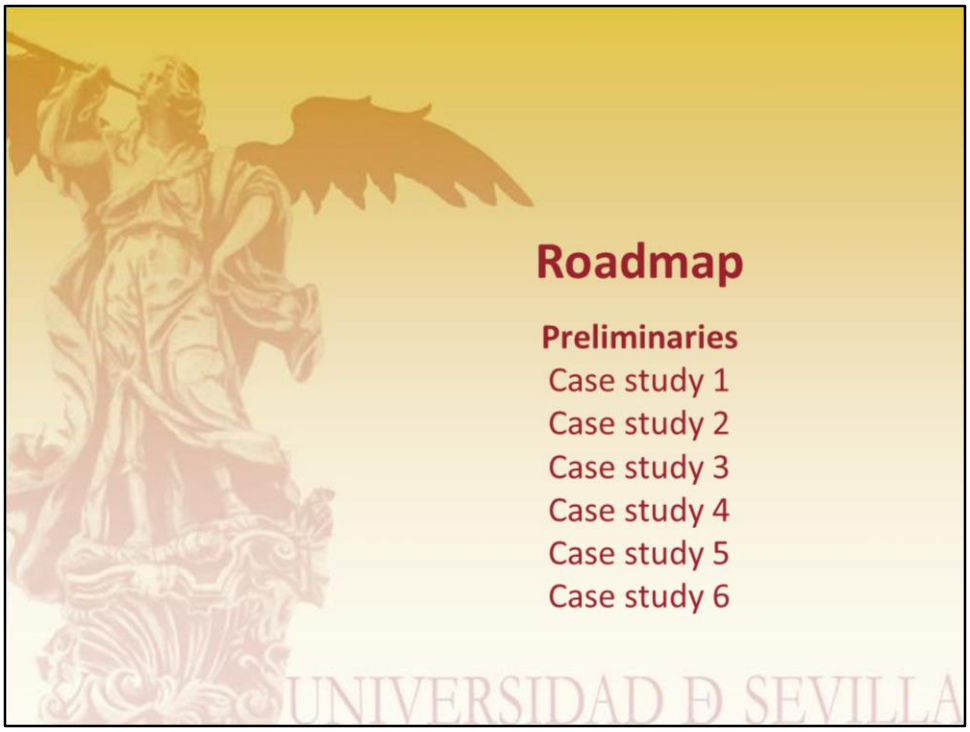
Then we'll study how to create a typical dashboard.

Step 7: case study 6

And we'll also provide a hint on how to display the dashboard graphically.

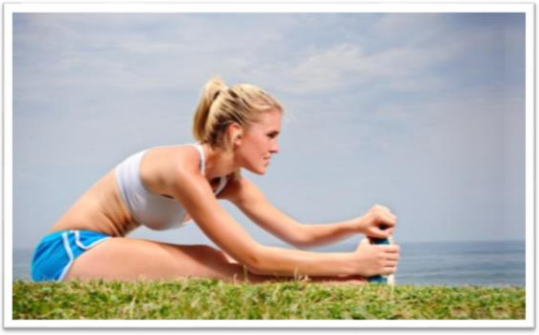


So, it shouldn't be surprising at all that this is our roadmap today.



Let's start with the preliminaries.

What are the preliminaries?



Some key concepts that are used throughout the lesson

UNIVERSIDAD Ð SEVILLA

14

The preliminaries are some key concepts that are used throughout the lesson.

We're going to use a simple project to illustrate the concepts, so we'll start with a guided tour through it; then we'll report on how to initialise your workspace, how to initialise your project, how to create the database, and, finally, we'll explore how you can run your first sample system.

Let's start with the guided tour.

This is the welcome page of Acme Shout. It's a simple web information system that is intended to help people shout. A shout is a record with a link and a piece of text. Please, consult the requirements elicitation that accompanies these lecture notes to learn more about this project.

This is the logging in page. There are three predefined user accounts, namely: administrator/administrator, customer/customer, and super/super. The super user account allows you to act as both an administrator and a customer at the same time.

The profile menu

This slide shows what the application must look like when a user logs in as super/super. Note that there are thee options in in the main menu, namely: administrator, profile, and customer. Let's start exploring the profile option, which is the simplest one. Note that there are four actions.

Profile action 1

The first action is intended to show three random inspiring quotes by famous people. That is, every time this action is executed, a different subset of quotes should be displayed.

Profile action 2

UNIVERSIDAD D SEVILLA

21

The second action in the profile menu must show a simple calculator that must allow the user to do sums, subtractions, multiplications, and divisions.

Profile action 3

The third action must raise an exception. It's encoded in the sample project, so we don't have to modify anything. We then won't pay any attention to this action.

Profile logout

UNIVERSIDAD D SEVILLA

23

The fourth action in the profile menu helps an authenticated user to logout from the system, which should return him or her to the welcome page.

The customer's menu

Let's now analyse the customer's menu.

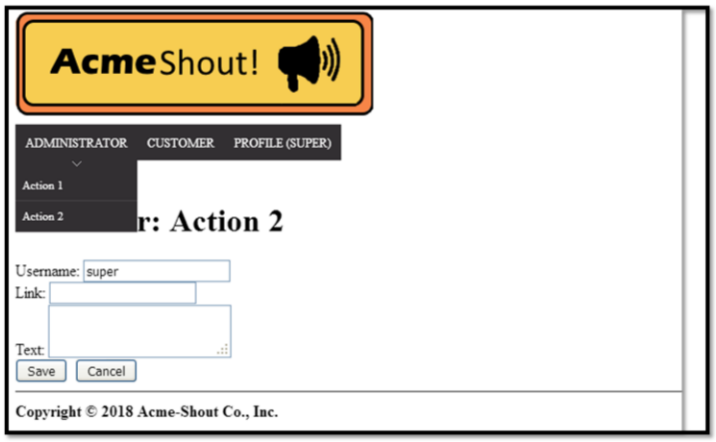
Customer's action 1

The first action in the customer's menu must show a listing of shouts that customers have already stored in the system. Note that the listing's paginated to facilitate reading it.

Customer's action 2

The second option in the customer's menu must show a form in which a customer can enter a new shout.

The administrator's menu

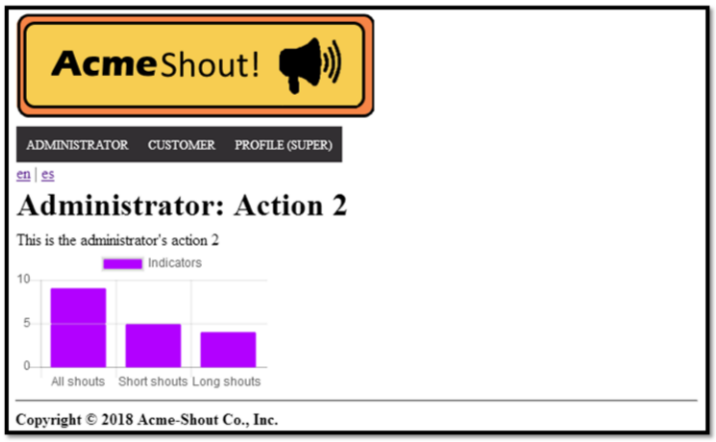


Finally, let's explore the administrator's menu.

Administrator's action 1

The first option in the administrator's menu must show a dashboard with some simple statistics.

Administrator's action 2



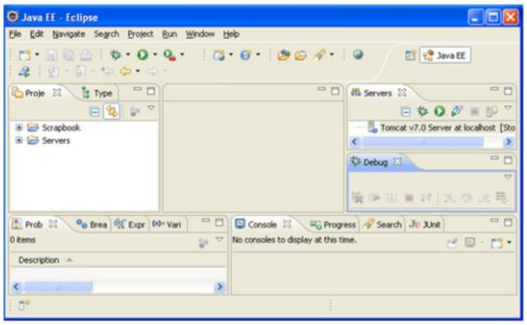
UNIVERSIDAD D SEVILLA

29

And the second action must show the dashboard using a vertical bar chart.

Pretty simple, right? In a few months, we'll be able to produce far more complex web information systems, but this is enough to get started. Let's now delve into how to initialise your workspace.

What's a workspace?

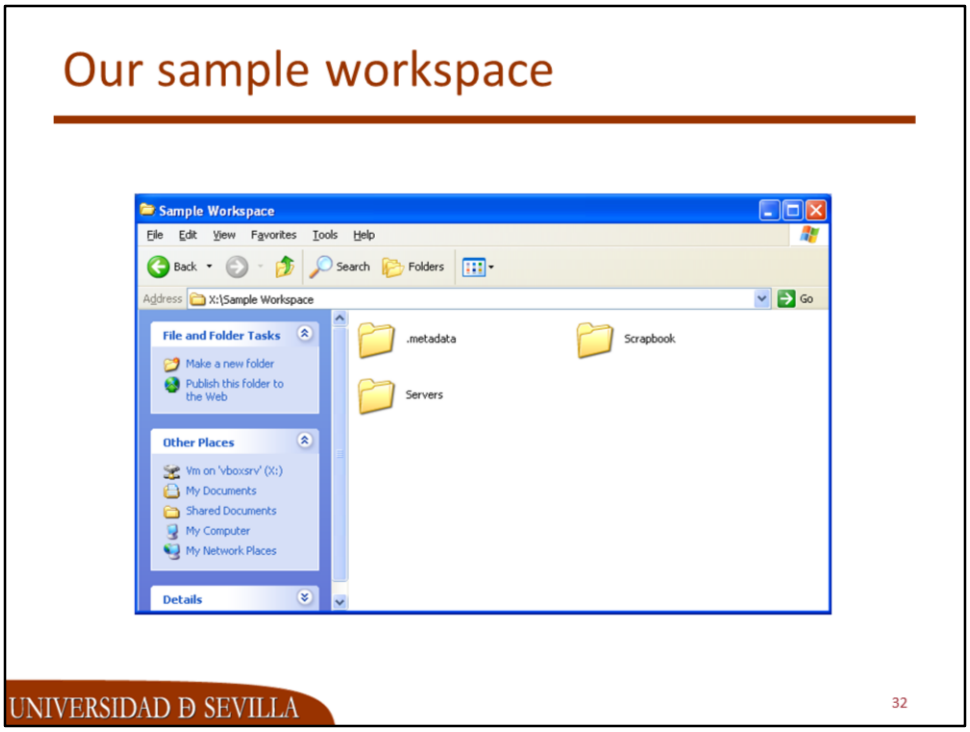


It's a configuration of your integrated development environment in which every view you may require is at your fingertips

UNIVERSIDAD Ð SEVILLA

31

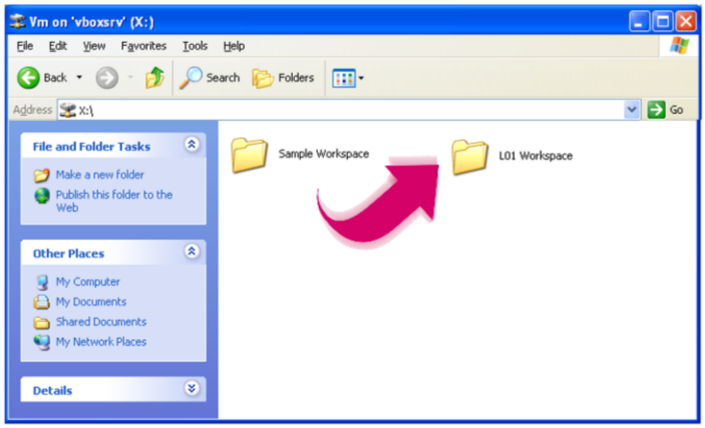
A workspace is a configuration of your integrated development environment in which every view you may require is at your fingertips.



There should be a folder called “Sample Workspace” in the materials that accompany this lecture. It should look like the folder in this slide. There’s a folder called “.metadata” in which the different plugins of Eclipse store data. There’s an additional folder called “Servers” that stores data about Tomcat, our application server. The “Scrapbook” folder’s intended to store useful documents that you need to use very frequently; in our template, it provides a number of SQL scripts to create databases, users, to grant privileges, and the like.

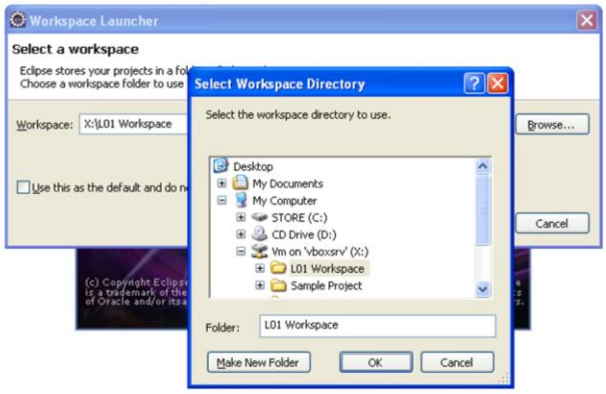
NOTE: the name of the folders in the materials are slightly different and they include version numbers since these resources are updated regularly.

Copy the workspace



To instantiate the template, you just need to copy folder “Sample Workspace” and paste it using a different name. We recommend that you should instantiate a workspace per lesson, so you should paste it as “L01 Workspace”.

Load it into Eclipse

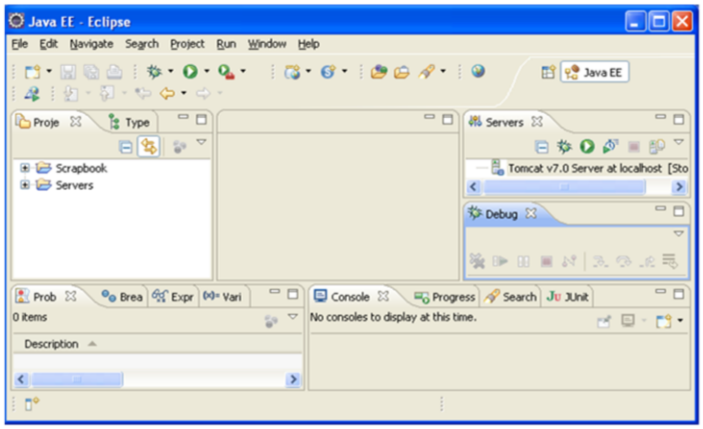


UNIVERSIDAD Ð SEVILLA

34

Let's load your first workspace into Eclipse. Launch Eclipse, and select folder "L01 Workspace" when prompted for a workspace.

The Java EE perspective



UNIVERSIDAD D SEVILLA

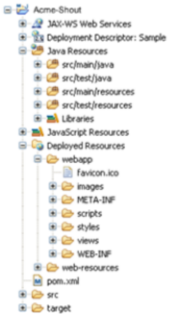
35

Eclipse should look like in this slide after loading your workspace. It displays the Java EE perspective and the following views: the project explorer, the type hierarchy, the build problems, the breakpoints, the expressions, the variables, the console, the progress, the search, the Junit, the servers, and the debug tools. We strongly recommend that you should keep this perspective as is since it builds on years of experience working on web information systems; it puts the most common tools you need at your fingertips and maximises the space on the screen.

NOTE: the workspace might look different in practice since these resources are updated regularly.

Let's now report on how to initialise your project.

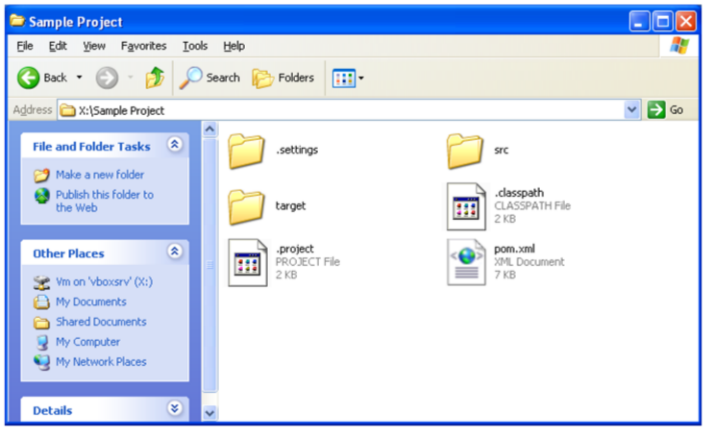
What's initialising your project?



It's instantiating a sample project that you can use as a starting point

Initialising your project consists in instantiating a sample project that you can use as a starting point. The sample project provides a hello-world project that will save you from a lot of cumbersome work that must be repeated every time you start working on a new project.

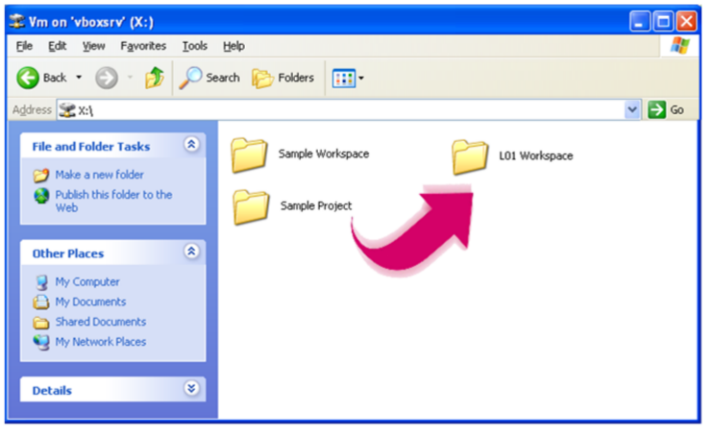
Our sample project



There's a folder called "Sample Project" that accompanies this lecture. It should look like in this slide. There's a folder called ".settings" and two files called ".project" and ".classpath" that have data required by Eclipse; please, never make changes to this folder or these files since this might render your project unusable. There's another folder called "target" to which Eclipse will output the results of compiling, testing, or packaging your project; you may freely delete this folder if necessary since it's reconstructed on the fly. Folder "src" provides the source code to your Java classes.

NOTE: please, note that these resources are updated regularly. So the actual folder that you've got with this lesson might be slightly different and include a version number.

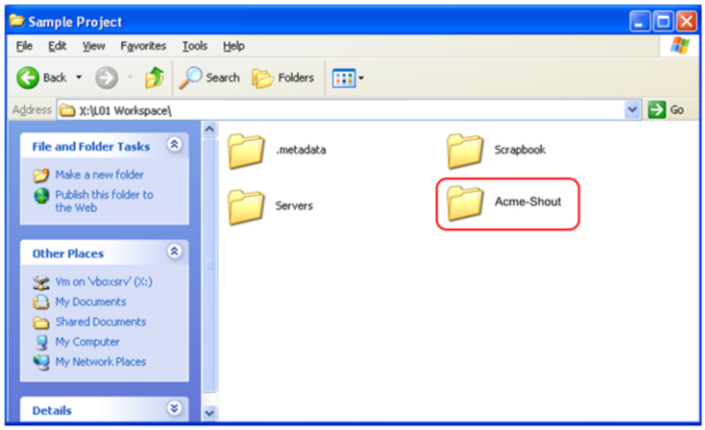
Copy the template and …



UNIVERSIDAD Ð SEVILLA

Instantiating the project template is very simple: just copy its entire folder into your workspace and rename it appropriately. Please, recall that we strongly recommend that you should have an independent workspace for each lesson. So you should copy folder “Sample Project” into folder “L01 Workspace” and then…

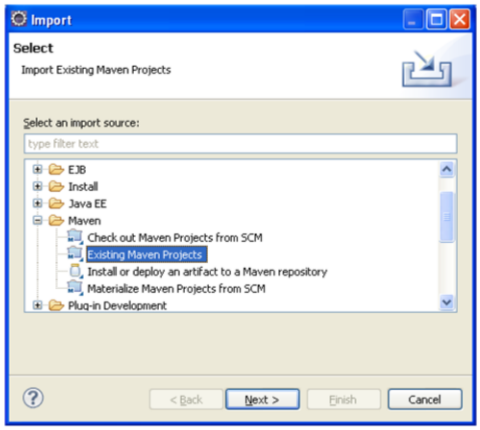
… rename it appropriately



UNIVERSIDAD D SEVILLA

… rename it appropriately.

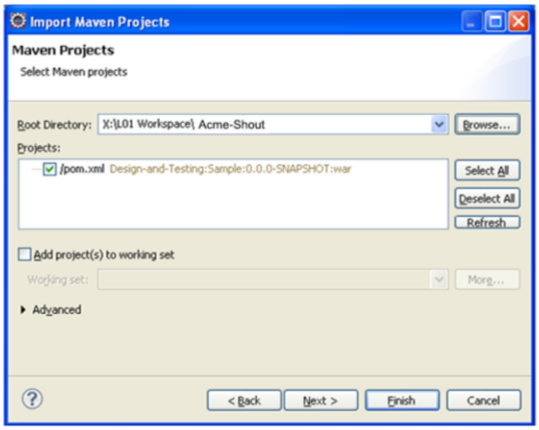
Import the template (I)



UNIVERSIDAD D SEVILLA

Once the folder's been copied, you have to import the project into your workspace. To do so, launch the "File > Import…" dialog box in Eclipse. Search this dialog box for "Existing Maven Projects" and then click the "Next >" button.

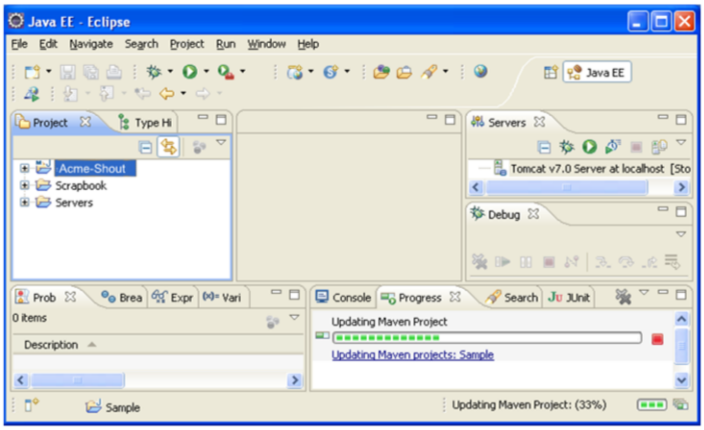
Import the template (II)



UNIVERSIDAD Ð SEVILLA

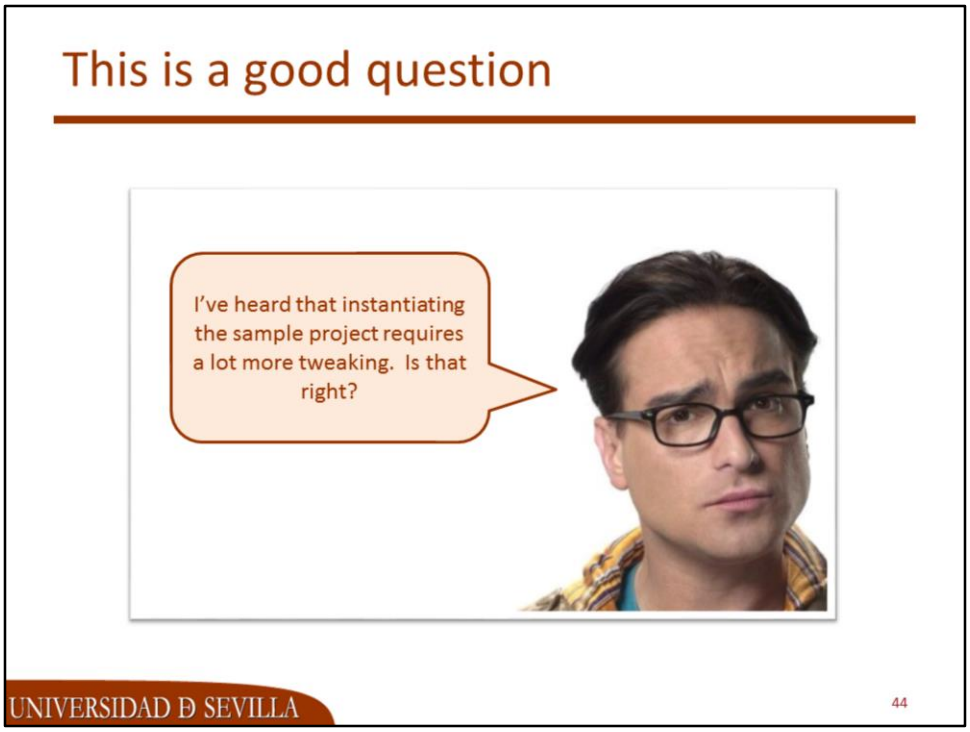
Click on the “Browse…” button and select folder “Acme-Shout”. Check the “pom.xml” file if it is not checked by default, and click on the “Finish” button. If you click on the “Next >” button, you’ll get a new dialog box to select Maven goals; just click the “Finish” button and go ahead. Maven is a plugin that Eclipse uses to compile your projects; in general, you won’t need to worry about it, but you must know that it exists.

Please, wait



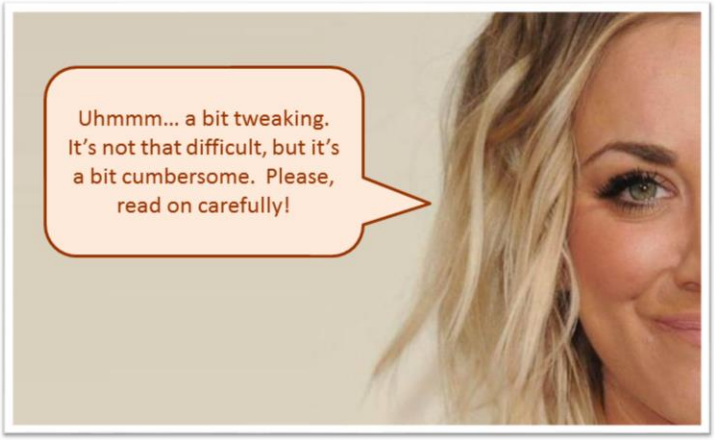
Please, wait. Maven will take a little to import the project into your workspace. Switch to the Progress view to see what's happening: Maven's downloading the required components, it's installing them to a local repository, and it's compiling your project.

NOTE: this step should take a few seconds in our virtual machine since the repository of components was downloaded prior to releasing the machine to you. If you're not working on our virtual machine, this step may take 20-30 minutes depending on your computer and your Internet connection. Note that you have to be connected to the Internet the first time that you instantiate a project since, otherwise, Maven won't be able to download the components that it requires.

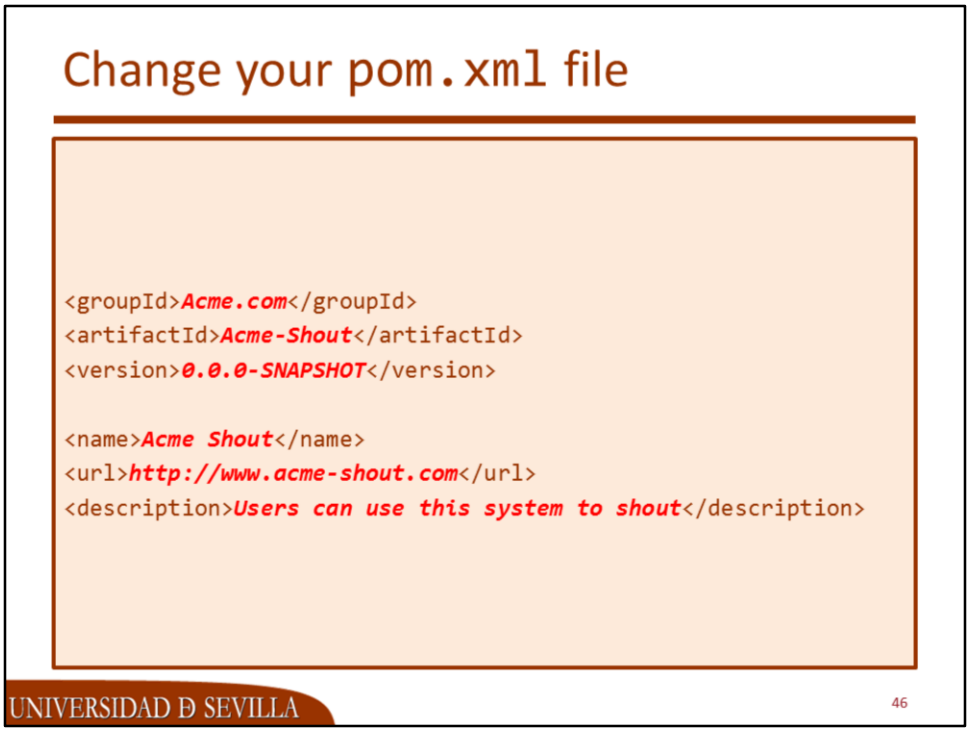


Unfortunately, copying and loading the template is not enough to start working.

This is a good answer

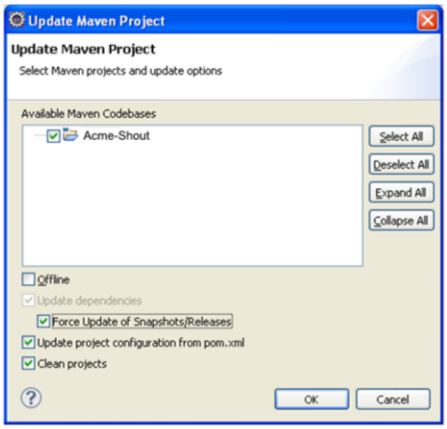


You need to tweak the project a bit, which is not difficult at all, but a bit cumbersome. So please, read on and pay attention to the details.

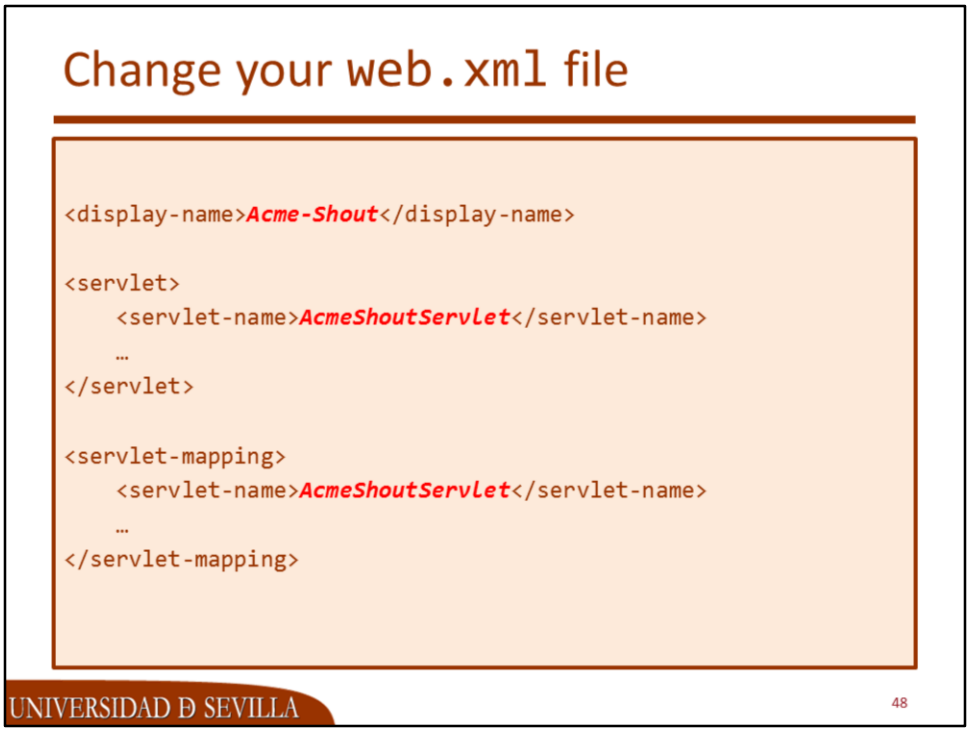


The first thing you have to do is to locate a file called “pom.xml” and change the information in the elements shown in this slide to adapt them to your project. “artifactId” and “version” refer to the name of your project and its version number, respectively. “groupId” refers to a name that you give to a number of related projects. “name”, “url”, and “description” are free-text fields that you can use to describe your project.

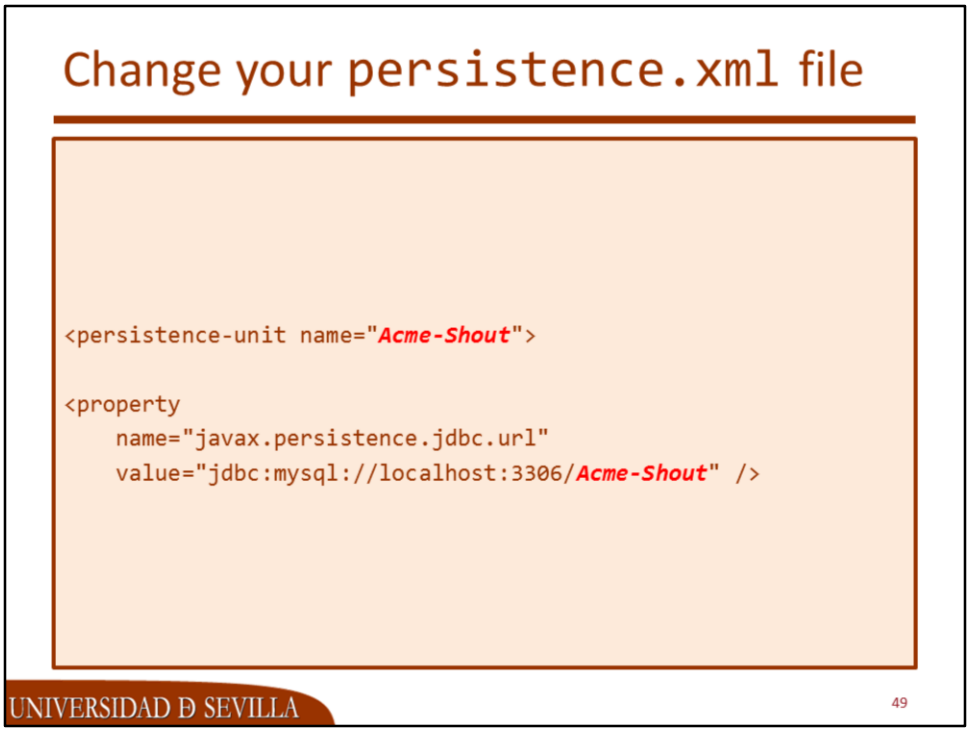
Let Maven update your project



It's necessary that Maven updates your project configuration after you change your "`pom.xml`" file; to do so, please, right-click your project, select option "Maven", and then option "Update project". This opens a dialog box in which you just have to select your project, check "Force updates of Snapshots/Releases", and click the OK button. Maven may take a little to reconfigure your project; please, wait until the re-configuration's finished.



Locate file “web.xml” file, and change the elements in this slide to reflect the features of your project. “display-name” refers to the name of your project within the application server, and “servlet-name” refers to the name of your servlet. A servlet is an object that acts as a interface in between the network and your project. We’ll provide some additional details latter in this lesion; so far, it’s enough to know that we recommend that the servlet name should be the name of your project with suffix “Servlet”.



In the “persistence.xml” file, there are two elements that you must change to adapt them to your new project. First, you must change the name of your persistence unit to the name of your project; then seek for property “javax.persistence.jdbc.url” and change it accordingly. We strongly recommend that each project should have a companion database with the same name.

"DatabaseConfig.java" is a configuration file that we need to populate and query the database of your project. You have to change this line to change the name of your persistence unit, which must be exactly the same name that you used in file "persistence.xml". This configuration file also allows to change the name of the default XML file in which the entities used to populate the database are specified ("PopulateDatabase.xml" by default) and the name of a map that associates their bean names with their identifiers in the database ("Entities.map" by default). It's unlikely that you need to modify those filenames, so keep them to their default values. We'll learn more about them in forthcoming lessons.

Change your data.xml file

```
<bean  id="dataSource"
       class="com.mchange.v2.c3p0.ComboPooledDataSource"
       destroy-method="close">
   <property name="driverClass"
       value="com.mysql.jdbc.Driver" />
   <property name="jdbcUrl"
       value="jdbc:mysql://localhost:3306/Acme-Shout" />
   <property name="user" value="acme-user" />
   <property name="password" value="ACME-Us3r-P@ssw0rd" />
</bean>
…
<bean id="persistenceUnit" class="java.lang.String">
    <constructor-arg value="Acme-Shout" />
</bean>
```

UNIVERSIDAD D SEVILLA

51

The “data.xml” file provides additional information about the database of your project. You must first configure a datasource, which is an object that manages several concurrent connections to the database, and then reference the persistence unit defined in “persistence.xml”.

Change your `footer.jsp` file

```
<b>Copyright &copy;
    <fmt:formatDate value="${date}" pattern="yyyy" />
    Acme Shout Co., Inc.</b>
```


This file contains the footer of your application. By default it simply shows a copyright message. You should adapt it to your project.

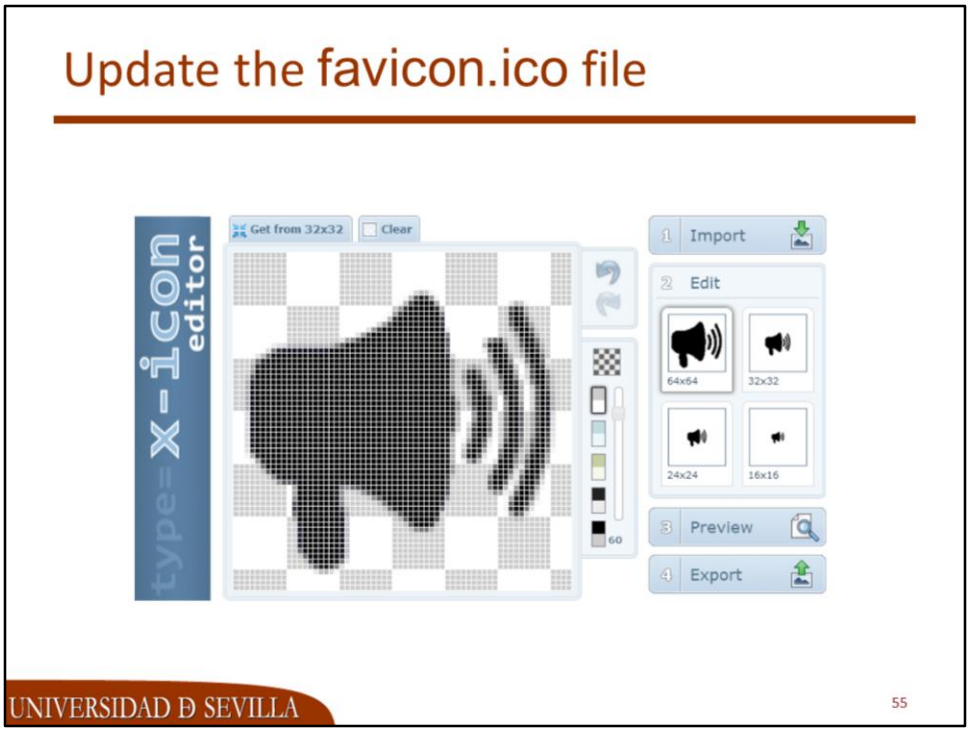
Change your header.jsp file (I)

```
<div>
    <img src="images/logo.png" alt="Acme Shout Co., Inc." />
</div>
```


This file contains the header of your application, which basically consists of a logo and a menu. Change the alternate text that is displayed on the logo to adapt it to your new project.

Update the logo.png file

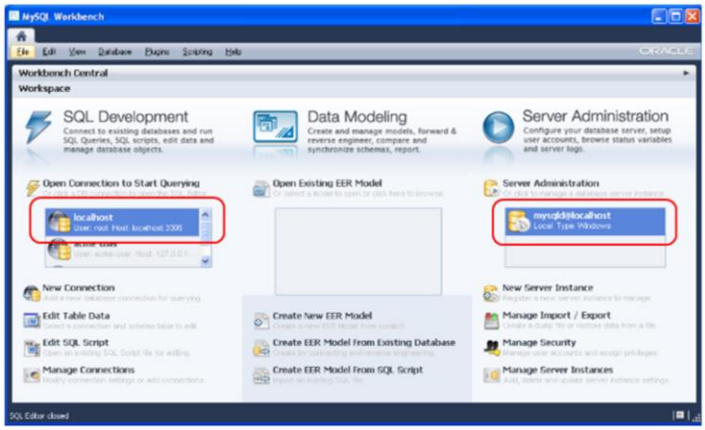
Now, update the “logo.png” file with the appropriate picture.



And, finally, update the “favicon.ico” file with the appropriate picture.

Once the project's properly initialised, it's time to create its database.

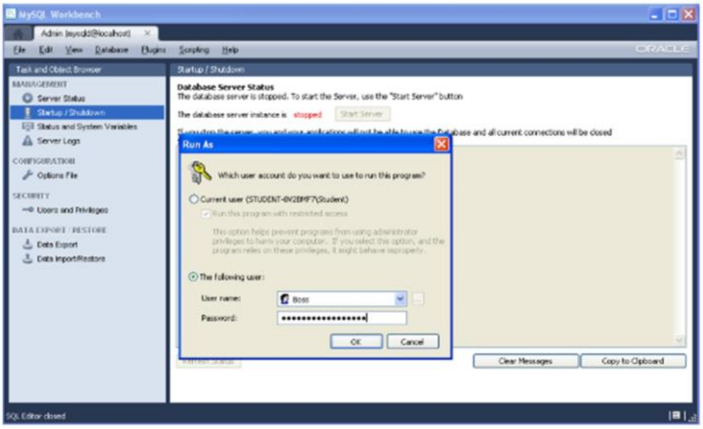
Open a root shell



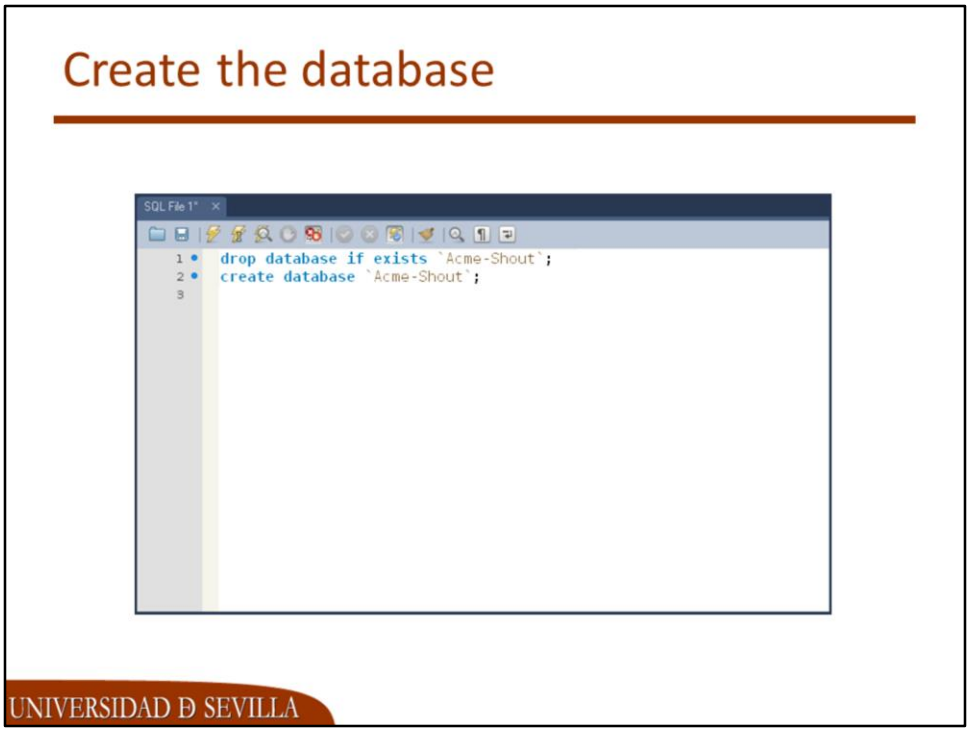
UNIVERSIDAD Ð SEVILLA

First, open a root shell in MySQL Workbench. If you're using our virtual machine, just launch MySQL Workbench, click on "mysqld@localhost" to start the server, and then on the preconfigured root connection to open a root shell. In MySQL's parlance, a root's a database administrator.

Start the server



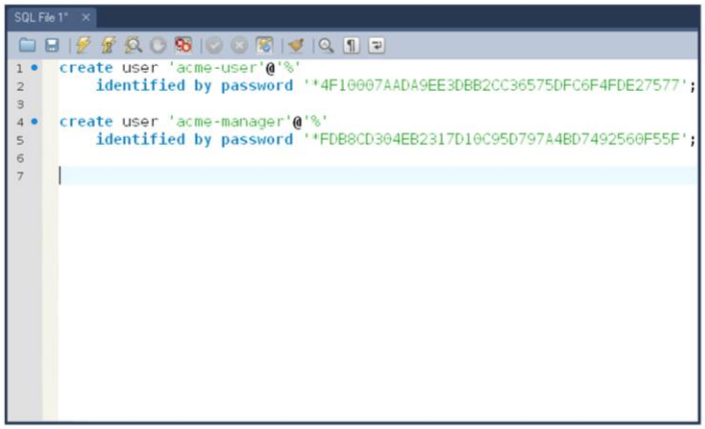
Start the MySQL, which requires you to log in as the administrator for a few seconds. Please, note that the administrator's account has the following credentials in our virtual machine: "Boss/$I=B0$$=U$3=P@$$".



To create the database, you just have to execute the following script:

drop database if exists `Acme-Shout`;
create database `Acme-Shout`;

Create the users



UNIVERSIDAD D SEVILLA

60

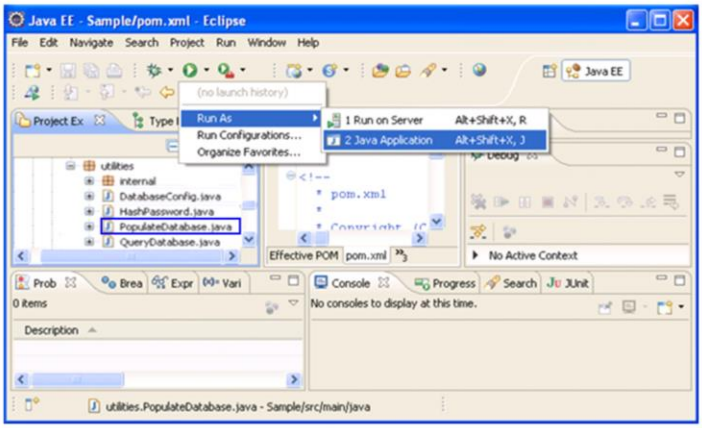
Next, you have to create the users that will allow your system to work with the database. By default, we'll create two users: a user called "acme-user" with password "ACME-Us3r-P@ssw0rd", which is allowed to operate on the data, and a user called "acme-manager" with password "ACME-M@n@ger-6874", which is allowed to operate on the schema of the database. Note that the passwords are provided to MySQL using the MD5 hashing algorithm; there's a utility called "HashPassword.java" in the project template that will surely help you generate the hashes.

We also have to grant privileges to users “acme-user” and “acme-manager”, which can be accomplished by means of the following script:

```
grant select, insert, update, delete
    on `Acme-Shout`.* to 'acme-user'@'%';

grant select, insert, update, delete, create, drop, references, index, alter,
    create temporary tables, lock tables, create view, create routine,
    alter routine, execute, trigger, show view
    on `Acme-Shout`.* to 'acme-manager'@'%';
```

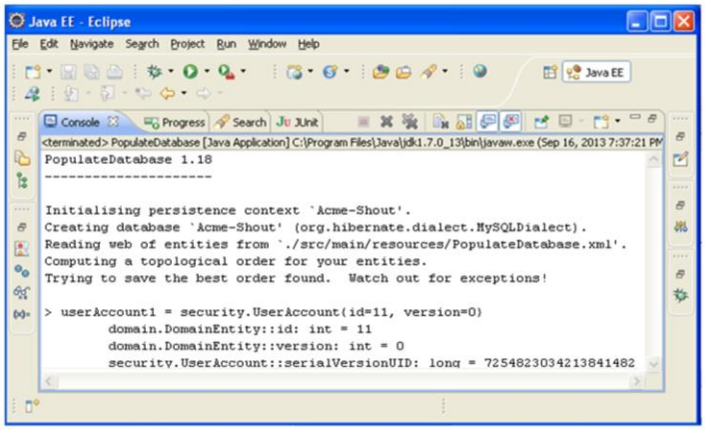
The first statement grants “acme-user” the minimum privileges to operate on data; the second statement grants “acme-manager” all of the privileges required to operate on the schema of your database.

Populate the database



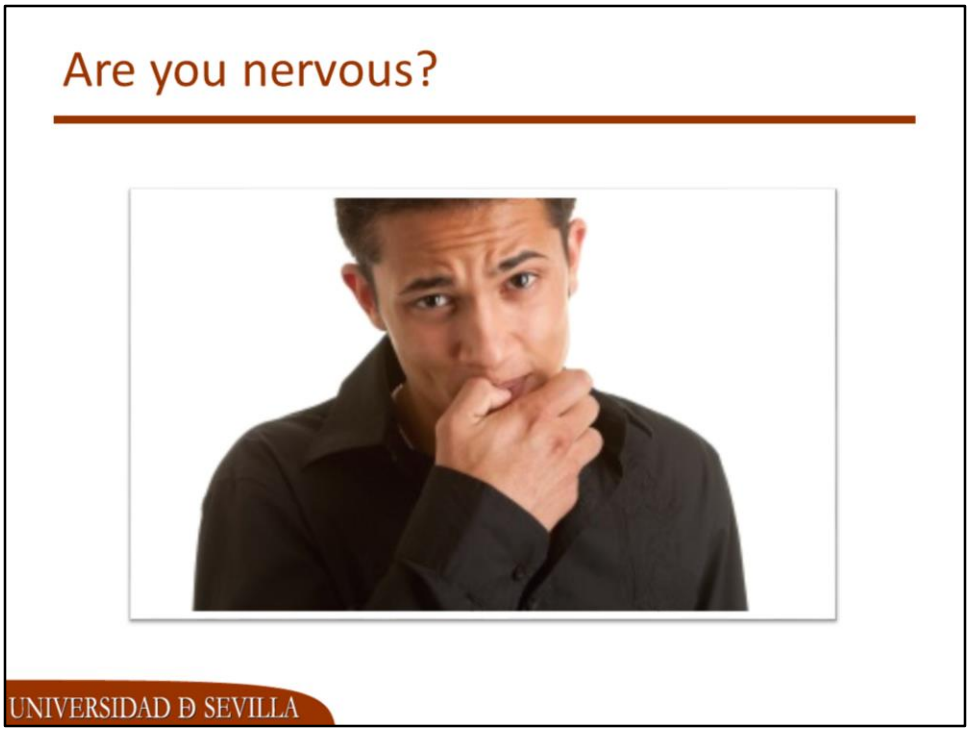
There's one more step before concluding this section: the database you've just created is empty. We have to populate it with some data. To do so, search for a file called "PopulateDatabase.java", which should be in the following folder: "Java Resources/src/main/java/utilities". Right-click it and select "Run as > Java Application".

Check the results



After a little activity, your Console view should look more or less like this. The information on the screen shows that the database's been populated with three sample user accounts, namely: "admin"/"admin", "customer"/"customer", and "super"/"super". We'll provide additional details later.

It’s time to give a try to your first project.



Are you nervous?

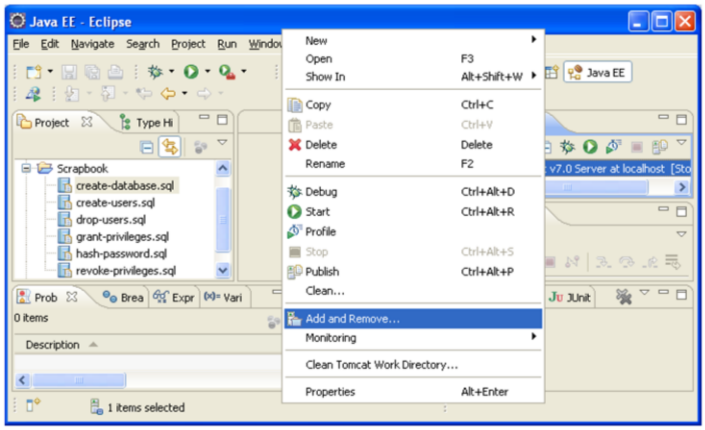
We too!

UNIVERSIDAD Ð SEVILLA

66

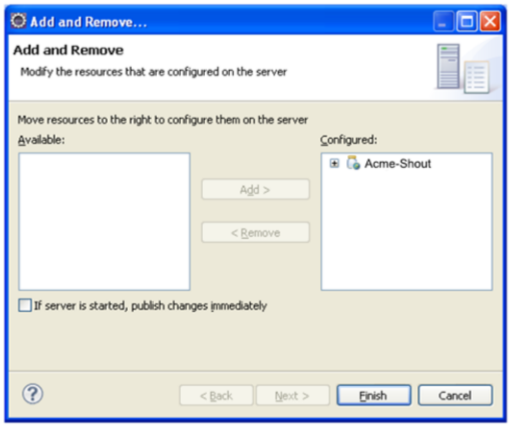
We too! We hope you've followed every guideline in this lecture so that you can start up the "Acme-Shout" project and get it running.

Let's deploy the project (I)



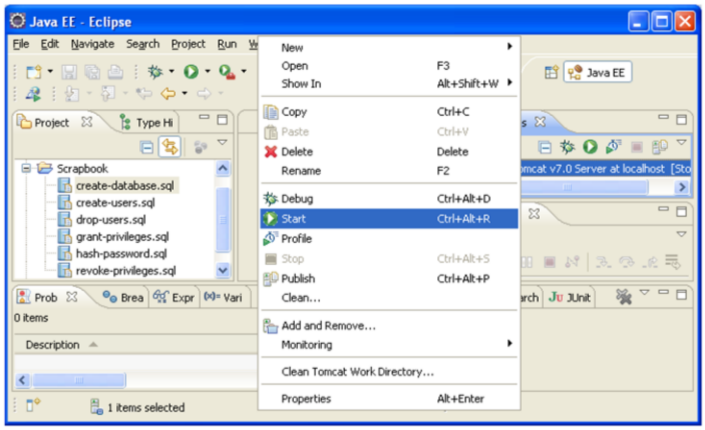
First of all, let's deploy your project to Tomcat. This is very simple: right-click "Tomcat v7.0 Server" in the Server view, and select "Add and Remove…".

Let's deploy the project (II)



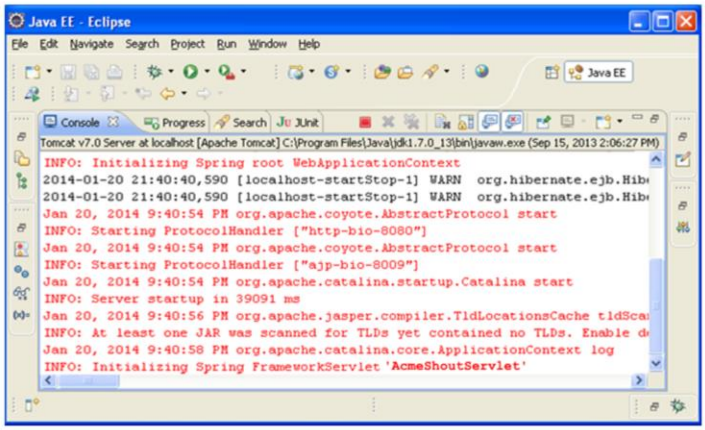
In this dialog box, move the "Acme-Shout" project from the "Available" column to the "Configured" column. Then, click the "Finish" button.

Let's start Tomcat



Once the project's been deployed, you have to start Tomcat. It's as easy as right-clicking the "Tomcat v7.0 Server" in the Servers view, and then selecting "Start".

Check the console



Starting Tomcat may take up to a minute depending on how powerful your computer is. Switch to the Console view to see what's happening. You'll see a lot of red messages, but don't worry, they are not error messages, just info messages. You should pay attention to the messages that are logged to the console. If everything's OK (everything should be OK if you've followed the guidelines in this lecture very carefully), then the following message should appear on the screen "INFO: Server startup in XXX seconds". That means that Tomcat is up and running, and that it's willing to accept requests to your web information system.

NOTE: unfortunately, there seems to be a problem with Spring Data 1.4.3: it displays the following warning every time your code has access to the database:

> WARN org.hibernate.ejb.HibernatePersistence - HHH015016: Encountered a deprecated javax.persistence.spi.PersistenceProvider [org.hibernate.ejb.HibernatePersistence]; use [org.hibernate.jpa.HibernatePersistenceProvider] instead.

Don't worry about this warning. Spring uses the correct class internally, but shows this warning mistakenly.

Open your browser and enjoy

UNIVERSIDAD D SEVILLA

71

Open your browser and make it for “http://localhost:8080/Acme-Shout”. After a little activity in your Tomcat server, your browser must display this screen. You can login using credentials “admin”/”admin”, “customer”/”customer”, or “super”/”super”. Please, give a try to the application and get familiar with it.

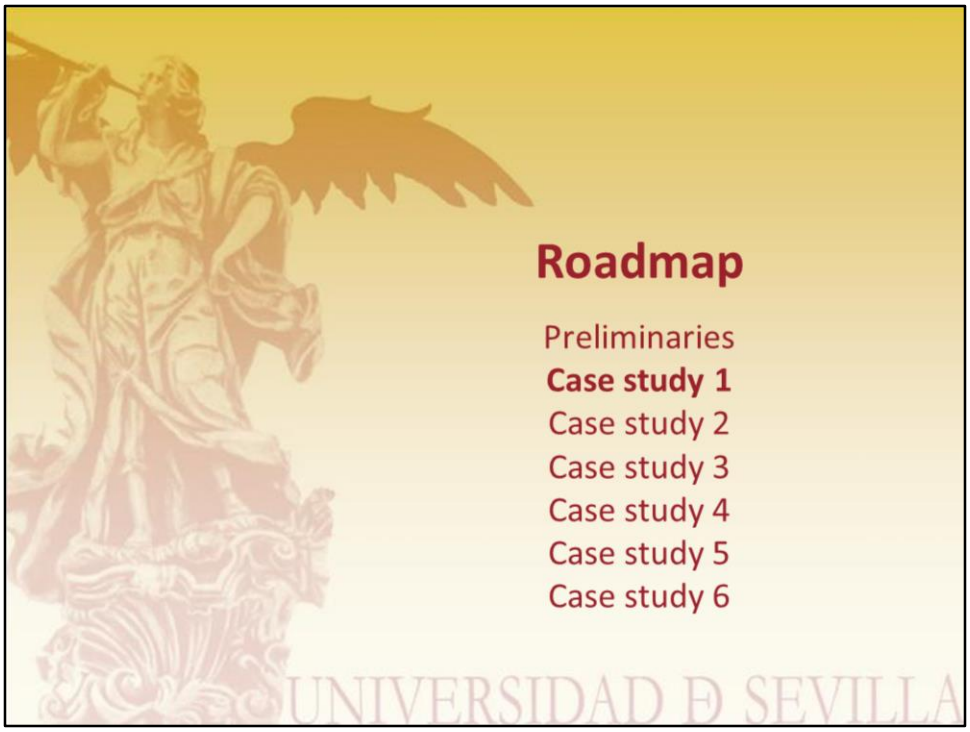
We sincerely expect that this is a picture of you. We hope you've followed every step and that the sample project's up and running now.

Or is this?

UNIVERSIDAD Ð SEVILLA

73

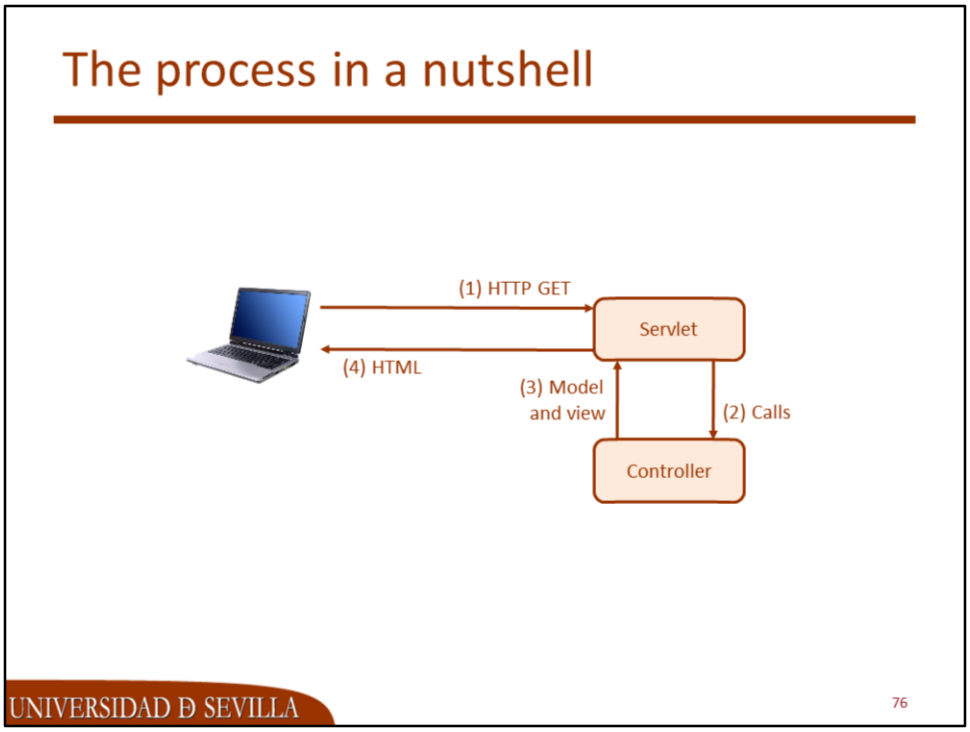
If not, please, get back, find what you didn't do the right way, and repeat it several times until your get familiar with the process. It's very important that you command the procedure to instantiate a workspace and a project template the sooner as possible. Please, bear in mind, that we, the lecturers, cannot help you with your computer or your project; we guarantee that the guidelines we provide work well; it's your duty to follow them very carefully.



It's enough about the preliminaries. It's time to start learning how to produce a web information system that is a bit more than a hello-world project. So, let's start with our first case study.

One picture and 1 000 words

Our goal's to change how the system performs when Action 1 is selected from the Profile menu. The idea is clear, but we'll surely have to speak more than one thousand words to explain the details. Let's go ahead!

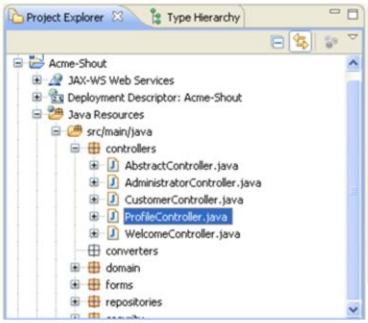


This slide shows the process that we're going to implement in a nutshell. The main components are the following:

- The browser, which allows your users to send request to your system using the HTTP protocol.
- The servlet, which intercepts the requests and routes them to user-defined controllers. The servlet is provided by the framework that we're using to design our systems, so you don't have to write it.
- The controller, which is a class that provides the methods that implement the business logic behind each request. This class must by written by you.

The process is as follows: (1) first, the user makes his or her browser for your system, which results in an HTTP/GET request; (2) the request is intercepted by the servlet, which calls the appropriate method in your controller; (3) the controller produces a model (which is a map of variables onto values) and a view (which is a specification of a user interface to render the model); (4) then the servlet processes the model and the view and generates an HTML document that is sent to the browser.

The controller (I)



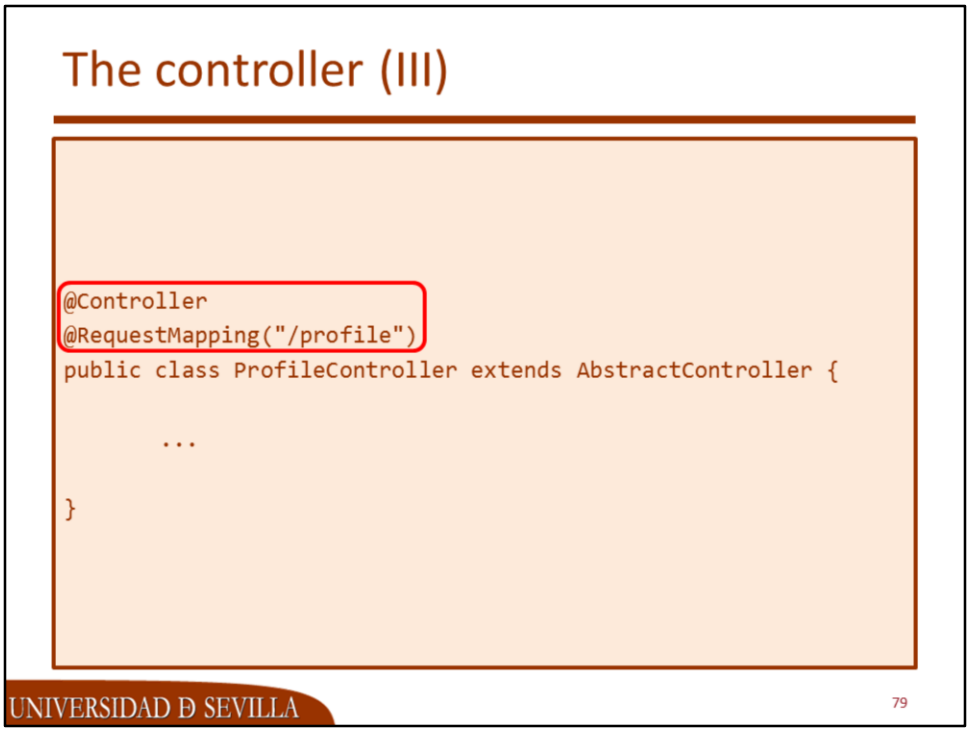
It's a Java class with one method that gets user requests to /profile/action-1.do, creates some quotes, initialises a model with three of them, and returns the model and the view to render it

Let's start with the controller, which is provided by file "ProfileController.java" in this case. The controller is a regular class that must have a method to handle requests to "/profile/action-1.do"; handling such a request consist of creating some quotes in a list, initialising a model with three of them, and then returning the model and the appropriate view to render it.

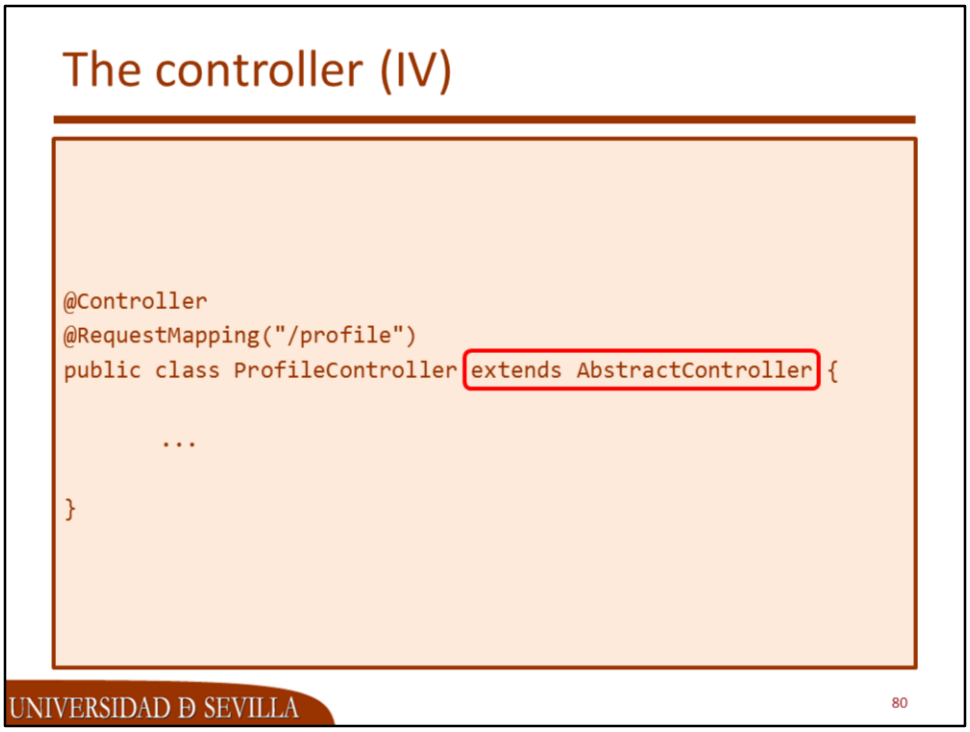
The controller (II)

```
@Controller
@RequestMapping("/profile")
public class ProfileController extends AbstractController {

        ...

}
```


This slide shows an excerpt of the controller that we're going to analyse in the following slides.



Note that the controller's introduced as a regular Java class with two annotations, namely: "@Controller", which indicates that it's a controller class, and "@RequestMapping", which indicates the prefix of the URLs for which this controller provides methods. Simply put, this class is a controller that will handle requests to URLs that start with "/profile", for instance: "http://localhost:8080/Acme-Shout/profile/action-1.do" or "http://www.acme.com/profile/action-1.do".



Note, too, that a controller must extend class “AbstractController”, which provides some additional methods that we’ll explore later in the subject.

The controller (V)

```
@RequestMapping(value = "/action-1", method = RequestMethod.GET)
public ModelAndView action1() {
    ModelAndView result;
    List<String> quotes;

    quotes = new ArrayList<String>();
    quotes.add("Make it simple, not simpler -- Albert Einstein");
    quotes.add("I have a dream -- Martin L. King");
    quotes.add("It always seem impossible until it's done -- Nelson Mandela");
    ...
    quotes.add("Cogito, ergo sum -- René Descartes");
    Collections.shuffle(quotes);
    quotes = quotes.subList(0, 3);

    result = new ModelAndView("profile/action-1");
    result.addObject("quotes", quotes);

    return result;
}
```


And this is the method that handles requests to "/profile/action-1.do". Let's examine it.

The controller (VI)

```
@RequestMapping(value = "/action-1", method = RequestMethod.GET)
public ModelAndView action1() {
    ModelAndView result;
    List<String> quotes;

    quotes = new ArrayList<String>();
    quotes.add("Make it simple, not simpler -- Albert Einstein");
    quotes.add("I have a dream -- Martin L. King");
    quotes.add("It always seem impossible until it's done -- Nelson Mandela");
    ...
    quotes.add("Cogito, ergo sum -- René Descartes");
    Collections.shuffle(quotes);
    quotes = quotes.subList(0, 3);

    result = new ModelAndView("profile/action-1");
    result.addObject("quotes", quotes);

    return result;
}
```


Note that the method has an annotation called “@RequestMapping”, which has two attributes, namely: “value”, which indicates the name of the action, and “method”, which indicates the HTTP method used to invoke it. Recall that the controller class has an annotation “@RequestMapping” that indicates the prefix of the URLs; the methods have a complementary annotation to indicate the action. This method serves requests to URL “http://localhost:8080/Acme-Shout/profile/action-1.do” using the HTTP/GET method, but not requests to “http://localhost:8080/Acme-Shout/customer/action-1.do”, which require a different controller.

The controller (VII)

```
@RequestMapping(value = "/action-1", method = RequestMethod.GET)
public ModelAndView action1() {
    ModelAndView result;
    List<String> quotes;

    quotes = new ArrayList<String>();
    quotes.add("Make it simple, not simpler -- Albert Einstein");
    quotes.add("I have a dream -- Martin L. King");
    quotes.add("It seem impossible until it's done -- Nelson Mandela");
    ...
    quotes.add("Cogito, ergo sum -- René Descartes");
    Collections.shuffle(quotes);
    quotes = quotes.subList(0, 3);

    result = new ModelAndView("profile/action-1");
    result.addObject("quotes", quotes);

    return result;
}
```

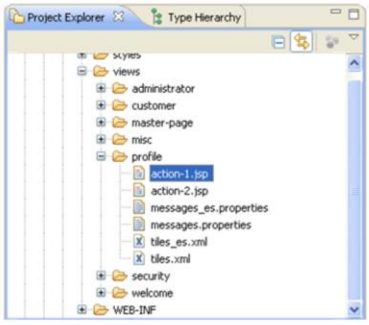

The body of the method consists of two parts. The first one creates a list of quotes, then shuffles them, and selects the first three quotes from the result.

The controller (VIII)

```
@RequestMapping(value = "/action-1", method = RequestMethod.GET)
public ModelAndView action1() {
    ModelAndView result;
    List<String> quotes;

    quotes = new ArrayList<String>();
    quotes.add("Make it simple, not simpler -- Albert Einstein");
    quotes.add("I have a dream -- Martin L. King");
    quotes.add("It seem impossible until it's done -- Nelson Mandela");
    ...
    quotes.add("Cogito, ergo sum -- René Descartes");
    Collections.shuffle(quotes);
    quotes = quotes.subList(0, 3);

    result = new ModelAndView("profile/action-1");
    result.addObject("quotes", quotes);

    return result;
}
```


The second part creates a “ModelAndView” object. The constructor gets the name of the view that must be used and then one can call the “addObject” method as many times as required to add new variables to the model. In this case, we’re injecting a variable called “quotes” within the model and its value is a list with three random quotes.

The view (I)



It's a JSP specification of a user interface to render the quotes in an ordered list

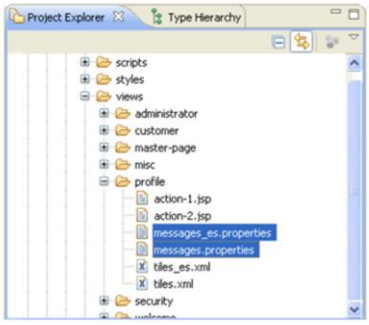
It's time to analyse the view, which is a JSP specification of a user interface to render the quotes in an ordered list.

This is the JSP code of the view, which interweaves regular HTML tags like “p”, “ol”, or “li” with processing tags like “spring:message”, “jstl:forEach”, or “jstl:out”. The first part of the view renders a paragraph that shows a message that should look like “This is action 1 in the profile”. Note that we do not write the message literally, but use tag “spring:message”, which takes a key as input and outputs the corresponding message in either Spanish or English depending on the user preferences. This is achieved by means of some so-called i18n&l10n bundles that we’re going to explore in a few slides.

The second part of the view introduces an ordered list and then uses a “jstl:forEach” loop to iterate over the list of quotes in the model; in each iteration, it outputs a quote in an “li” tag.

Note that the expressions that must be evaluated on the model must be introduced by means of the following construct: "${ exp }". This construct evaluates the inner expression and returns its result. It's a common mistake to write something like "<jstl:forEach var="quote" items="quotes"> … </jstl:forEach>" or "jstl:out value="quote" />", which will simply do not work.

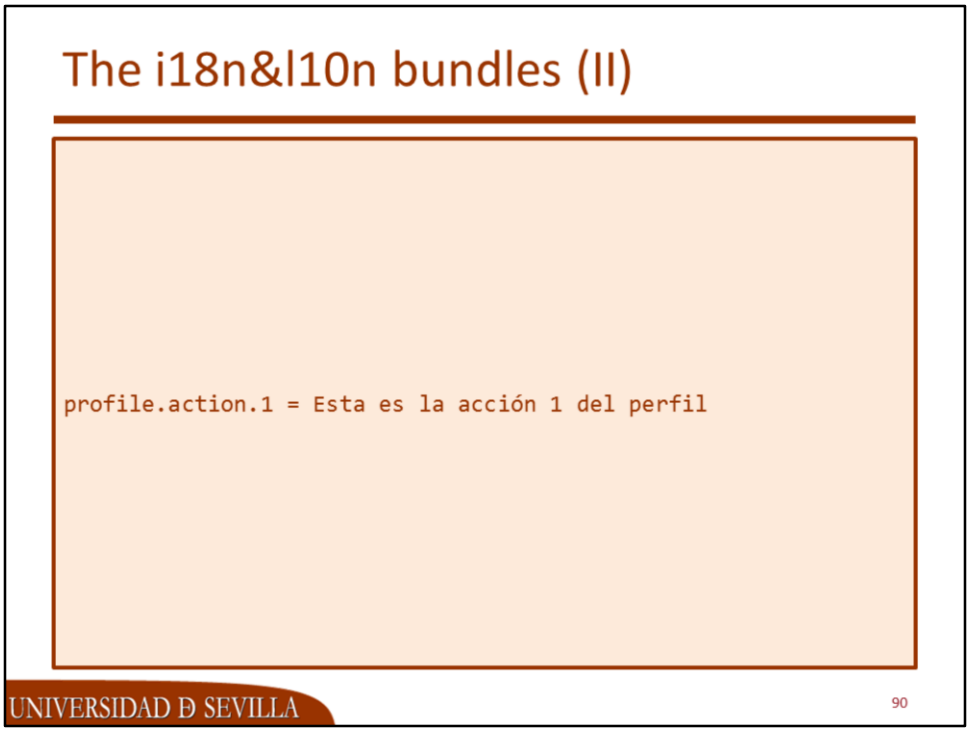
The i18n&l10n bundles (I)



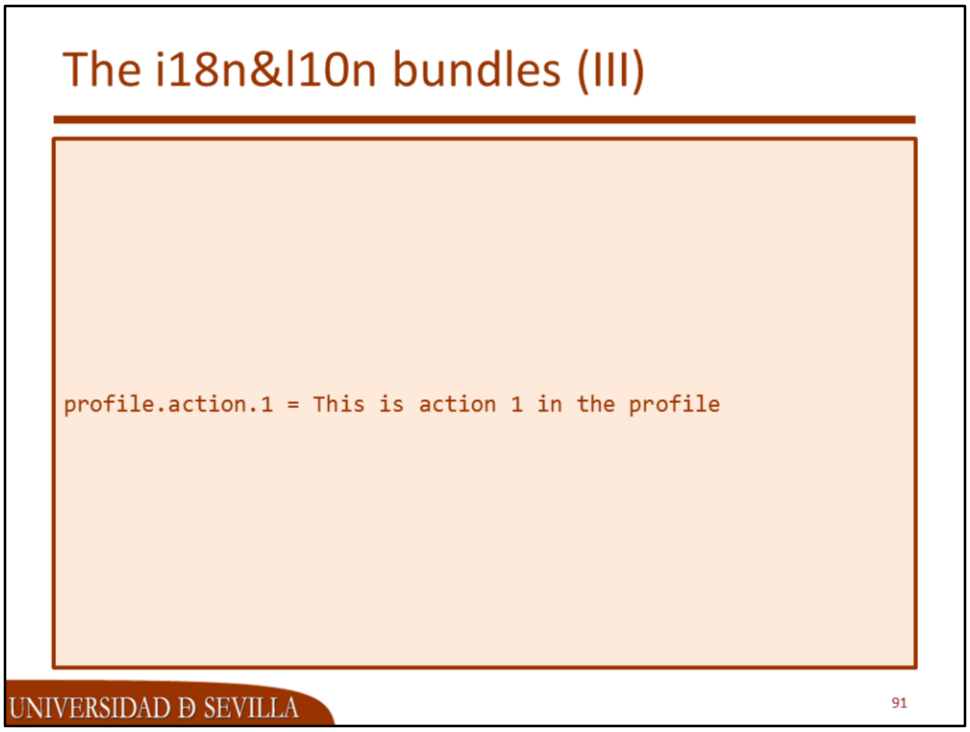
They're files that help translate keys into text in Spanish and English

A few slides ago, we introduced "spring:message" as a tag that allows to translate a key into a message written in Spanish or English. The translation is accomplished by means of so-called i18n&l10n bundles, which are simple key-value files.

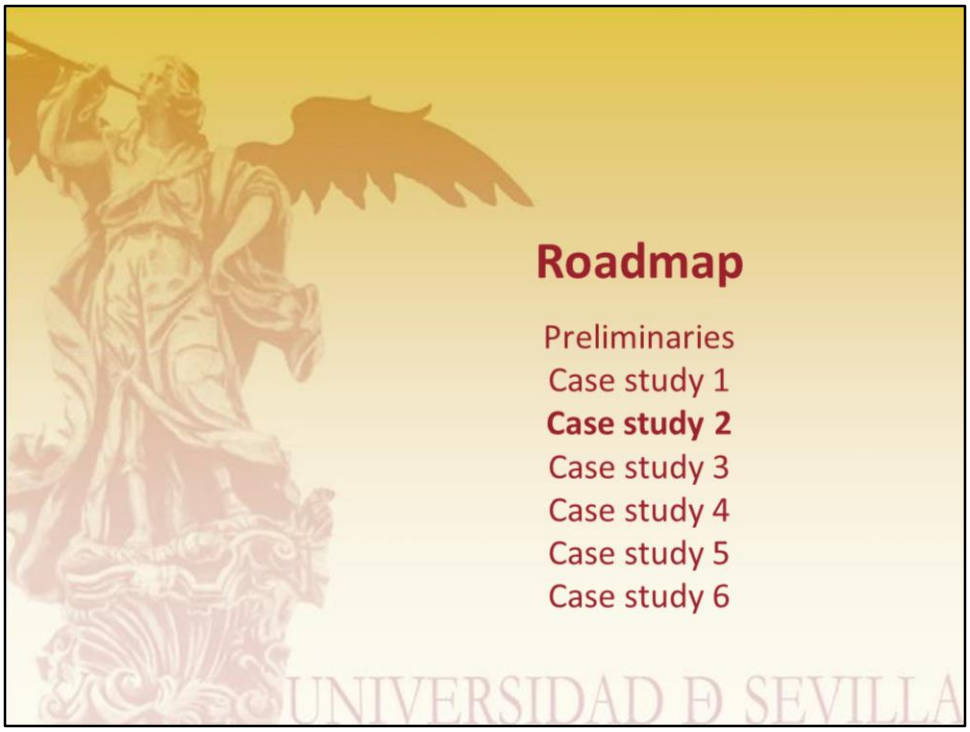
NOTE: "i18n" stands for "internationalisation" and "l10n" stands for "localisation". Just count the number of letters in between the leading "l" and the trailing "n" or the leading "l" and the trailing "n" to figure out the meaning of the abbreviations.



Here you can see the contents of “messages_es.properties”. It’s pretty simple, right? In the view, we used the following tag: “<spring:message code="profile.action.1" />”. The Spanish translation for key “profile.action.1” is in this file.



And file “messages.properties” contains the English translation for the key.



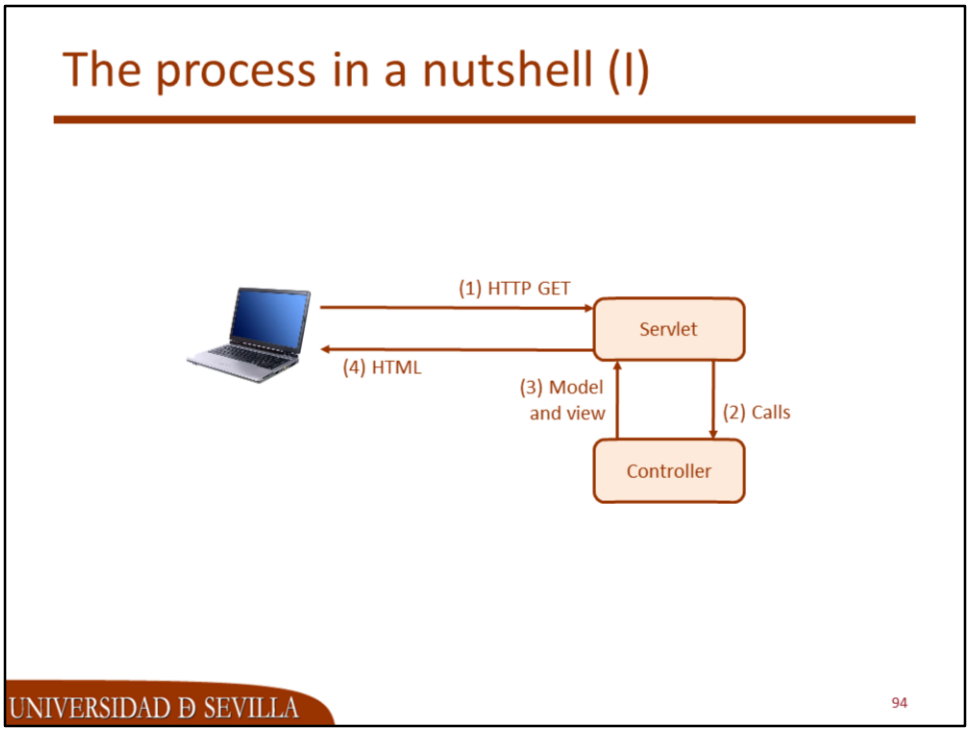
The first case study was simple, right? Let's go on and let's learn something a bit more difficult with the second case study.

One picture and 1 000 words

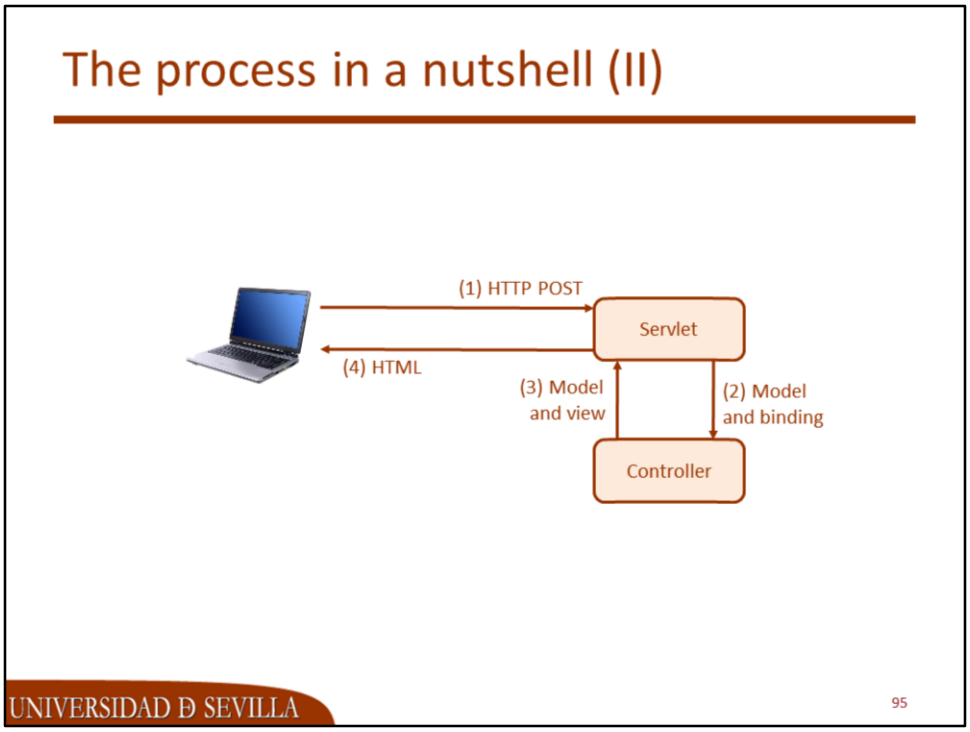
UNIVERSIDAD Ð SEVILLA

93

In this case study, we're going to extend our system so as to provide a simple arithmetic calculator when the user selects the second action of the profile menu.

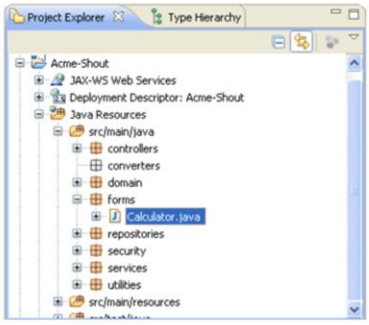


Implementing this case study requires two processes. The first one must sound familiar to you: (1) the user makes his or her browser for "http://localhost:8080/Acme-Shout/profile/action-2.do", for instance, which generates an HTTP/GET request to the servlet, which (2) calls your controller, which (3) produces a model and a view that the servlet uses to produce an HTML document that is sent back to the browser. This part of the process results in the blank form in the previous slide. The user can now enter two numbers and select an operator.



When the user presses the “Compute” button, the form must be sent to your system; in this case, the browser uses the HTTP/POST method. The request is intercepted by the servlet, which routes it to the appropriate controller, as usual. Just realise that the call to the controller gets a model and a binding as input. The model is an object that stores the data that the user’s entered in the form and the binding accounts for errors in the data (for instance, entering “two” instead of “2”). As usual, the controller processes the call and returns a model and a view to the servlet, which produces the HTML document that is sent back to the browser.

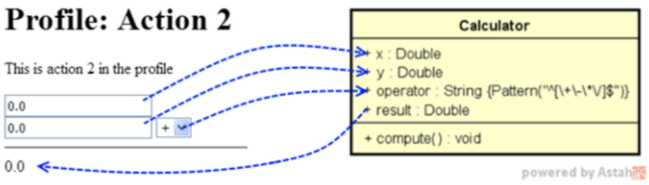
The form object (I)



It's an object that allows to store volatile data that a user enters in a form

Let's start our exploring this case study with the form object, which is an object that allows to store volatile data that a user enters in a form. Note that the emphasis in on volatile data, that is, data that is not stored in the data base. Later, we'll explore the case of persistent objects.

The form object (II)



The form object has attributes to store the data in the form plus the methods that you think might be appropriate to make computations on those data. In this slide, we show the calculator form object in UML, which is commonly referred to a the conceptual or the domain model. (Please, note that conceptual and domain models are not the same thing; in the forthcoming lesson, we'll study the difference.)

The form object (III)

```
public class Calculator {
    private double    x;
    private double    y;
    private String    operator;
    private double    result;

    public double getX() { return this.x; }
    public void setX(final double x) { this.x = x; }

    public double getY() { return this.y; }
    public void setY(final double y) { this.y = y; }

    @Pattern(regexp = "^[\\+\\-\\*\\/]$")
    public String getOperator() { return this.operator; }
    public void setOperator(final String operator) { ... }

    public double getResult() { return this.result; }
    public void setResult(final double result) { ... }
    ...
```

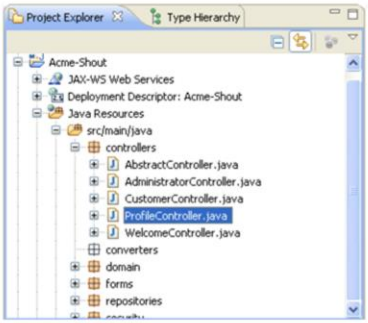

This is slide shows the code of the calculator form object. Note that it's a regular Java class with a number of properties that are introduced, as usual, by means of private attributes, getters, and setters. Note that the getter of the "operator" property has a "@Pattern" annotation; this annotation introduces a validation constraint that requires the value of the "operator" property to conform to the indicated regular expression; simply put, the operator must be either "+", "-", "*", or "/".

The form object (IV)

```
    ...

    public void compute() {
        switch (this.getOperator()) {
        case "+":
            this.setResult(this.getX() + this.getY());
            break;
        case "-":
            this.setResult(this.getX() - this.getY());
            break;
        case "*":
            this.setResult(this.getX() * this.getY());
            break;
        case "/":
            this.setResult(this.getX() / this.getY());
            break;
        }
    }
}
```


The calculator form object has a business method to compute the result, which is shown in this slide. Pretty simple, right?

The controller (I)



It's a Java class with two methods that get user requests to /profile/action-2.do using the GET or the POST methods; the first one returns an empty model an a view to render it; the second one validates the input form, updates the model with the result, and returns a view to render it

Let's now analyse the controller, which is a Java class with two methods that get user requests to "/profile/action-2.do" using either HTTP/GET or HTTP/POST. The first method returns an empty model and a view to render it; the second one gets the model as input, validates it, updates the model with the result, and returns a view to render it.

The controller (II)

```
@RequestMapping(value = "/action-2", method = RequestMethod.GET)
public ModelAndView action2Get() {
    ModelAndView result;
    Calculator calculator;

    calculator = new Calculator();

    result = new ModelAndView("profile/action-2");
    result.addObject("calculator", calculator);

    return result;
}
```


This slide shows the first method. The "@RequestMapping" annotation should now be familiar to you, so we won't provide any additional details.

The controller (III)

```
@RequestMapping(value = "/action-2", method = RequestMethod.GET)
public ModelAndView action2Get() {
    ModelAndView result;
    Calculator calculator;

    calculator = new Calculator();

    result = new ModelAndView("profile/action-2");
    result.addObject("calculator", calculator);

    return result;
}
```


The body of the method isn't difficult to understand: we first create a new calculator.

The controller (IV)

```
@RequestMapping(value = "/action-2", method = RequestMethod.GET)
public ModelAndView action2Get() {
    ModelAndView result;
    Calculator calculator;

    calculator = new Calculator();

    result = new ModelAndView("profile/action-2");
    result.addObject("calculator", calculator);

    return result;
}
```


And then create the “ModelAndView” object indicating the name of the view to render the model and injecting the model itself.

The controller (V)

```
@RequestMapping(value = "/action-2", method=RequestMethod.POST)
public ModelAndView action2Post(
        @Valid final Calculator calculator,
        final BindingResult binding) {
    ModelAndView result;

    calculator.compute();

    result = new ModelAndView("profile/action-2");
    result.addObject("calculator", calculator);

    return result;
}
```


And this is the second method. Note that it's prepended with an "@RequestMapping" annotation which indicates the HTTP/POST method in this case.

The controller (VI)

```
@RequestMapping(value = "/action-2", method=RequestMethod.POST)
public ModelAndView action2Post(
        @Valid final Calculator calculator,
        final BindingResult binding) {
    ModelAndView result;

    calculator.compute();

    result = new ModelAndView("profile/action-2");
    result.addObject("calculator", calculator);

    return result;
}
```


Note, too, that the method gets two parameters, namely: "calculator", which references the form object edited by the user, and "binding", which references an object with information about eventual validation errors. Note that the "calculator" parameter's prepended with an "@Valid" annotation, which requires it to be validated before it's passed into the method call.

The controller (VII)

```
@RequestMapping(value = "/action-2", method=RequestMethod.POST)
public ModelAndView action2Post(
        @Valid final Calculator calculator,
        final BindingResult binding) {
    ModelAndView result;

    calculator.compute();

    result = new ModelAndView("profile/action-2");
    result.addObject("calculator", calculator);

    return result;
}
```

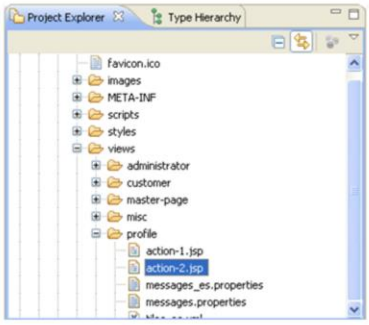

The body of the method is simple. First, the calculator is requested to compute its result.

The controller (VIII)

```
@RequestMapping(value = "/action-2", method=RequestMethod.POST)
public ModelAndView action2Post(
        @Valid final Calculator calculator,
        final BindingResult binding) {
    ModelAndView result;

    calculator.compute();

    result = new ModelAndView("profile/action-2");
    result.addObject("calculator", calculator);

    return result;
}
```


And then the “ModelAndView” object is created. If there were any errors, they would be shown automatically in the form when the view is transformed into HTML.

The view (I)



It's a JSP specification of a user interface to render a calculator in a form with a button to submit it

It's now time to analyse the view that we use to render the calculator.

The view (II)

```
<p><spring:message code="profile.action.2" /></p>

<form:form modelAttribute="calculator">
    <form:input path="x" /> <form:errors path="x" /> <br />
    <form:input path="y" /> <form:errors path="y" /> <br />
    <form:select path="operator" >
        <form:option value="+" />
        <form:option value="-" />
        <form:option value="*" />
        <form:option value="/" />
    </form:select>
    <form:errors path="operator" /> <br />
    <hr />
    <jstl:out value="${calculator.result}" />
    <br />
    <input type="submit"
           value="<spring:message code='profile.compute'/>" />
</form:form>
```


It starts with a paragraph that renders a message using the “spring:message” tag that we introduced in the previous case study.

The view (III)

```
<p><spring:message code="profile.action.2" /></p>

<form:form modelAttribute="calculator">
    <form:input path="x" /> <form:errors path="x" /> <br />
    <form:input path="y" /> <form:errors path="y" /> <br />
    <form:select path="operator" >
        <form:option value="+" />
        <form:option value="-" />
        <form:option value="*" />
        <form:option value="/" />
    </form:select>
    <form:errors path="operator" /> <br />
    <hr />
    <jstl:out value="${calculator.result}" />
    <br />
    <input type="submit"
           value="<spring:message code='profile.compute'/>" />
</form:form>
```


It then introduces a form by means of tag “form:form”. This tag requires you to indicate the name of the variable in the model that contains the object that is going to be edited in this form, which is injected in the controller.

The view (IV)

```
<p><spring:message code="profile.action.2" /></p>

<form:form modelAttribute="calculator">
    <form:input path="x" /> <form:errors path="x" /> <br />
    <form:input path="y" /> <form:errors path="y" /> <br />
    <form:select path="operator" >
        <form:option value="+" />
        <form:option value="-" />
        <form:option value="*" />
        <form:option value="/" />
    </form:select>
    <form:errors path="operator" /> <br />
    <hr />
    <jstl:out value="${calculator.result}" />
    <br />
    <input type="submit"
           value="<spring:message code='profile.compute'/>" />
</form:form>
```


Inside the form, we must put the input controls. Text boxes are introduced by means of “form:input” tags, which have an attribute called “path” that indicates the name of the attribute to be edited. Note that each input is accompanied by a “form:errors” tag that is used to display validation errors automatically.

The view (V)

```
<p><spring:message code="profile.action.2" /></p>

<form:form modelAttribute="calculator">
    <form:input path="x" /> <form:errors path="x" /> <br />
    <form:input path="y" /> <form:errors path="y" /> <br />
    <form:select path="operator" >
        <form:option value="+" />
        <form:option value="-" />
        <form:option value="*" />
        <form:option value="/" />
    </form:select>
    <form:errors path="operator" /> <br />
    <hr />
    <jstl:out value="${calculator.result}" />
    <br />
    <input type="submit"
           value="<spring:message code='profile.compute'/>" />
</form:form>
```

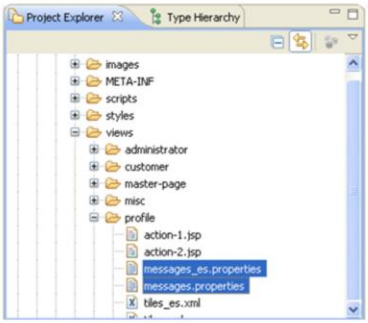

Tag “form:select” is used to introduce a drop-down list with the operators that are available.

The view (VI)

```
<p><spring:message code="profile.action.2" /></p>

<form:form modelAttribute="calculator">
    <form:input path="x" /> <form:errors path="x" /> <br />
    <form:input path="y" /> <form:errors path="y" /> <br />
    <form:select path="operator" >
        <form:option value="+" />
        <form:option value="-" />
        <form:option value="*" />
        <form:option value="/" />
    </form:select>
    <form:errors path="operator" /> <br />
    <hr />
    <jstl:out value="${calculator.result}" />
    <br />
    <input type="submit"
           value="<spring:message code='profile.compute'/>" />
</form:form>
```


And there's also a button to submit the form to the system. Note that there's a bit of heterogeneity here, since buttons must be introduced in plain HTML by means of an "input" tag with attribute "type" set to "submit"; note too that the value of the button must be translated into Spanish or English, which requires to embed a "spring:message" tag into the "input" tag.

The i18n&l10n bundles (I)



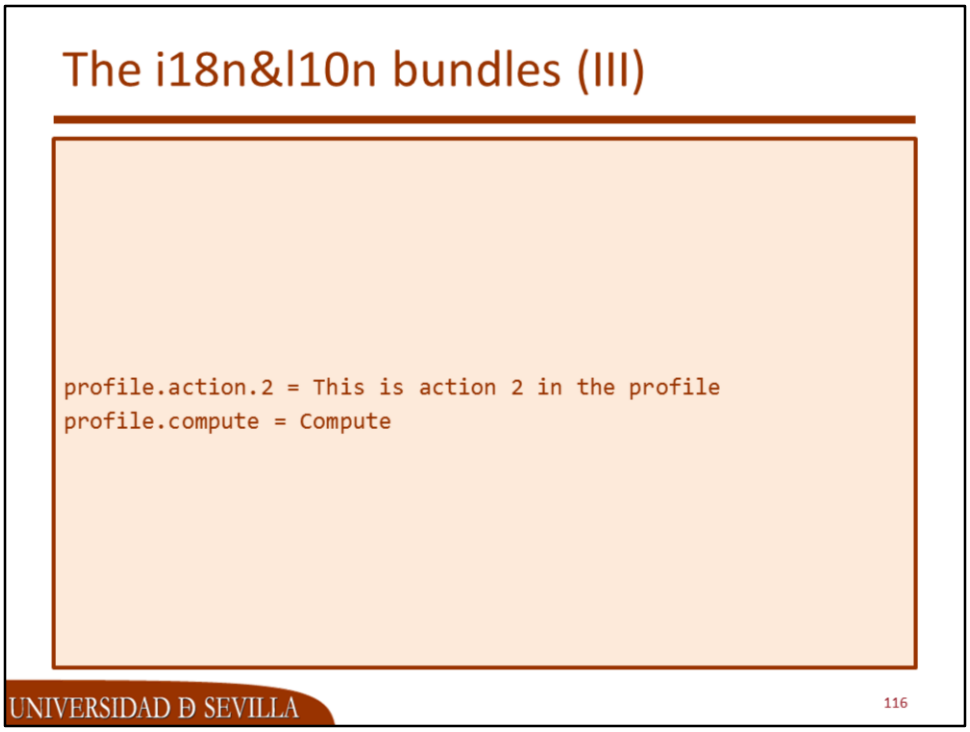
They're files that help translate keys into sentences in Spanish and English

And finally, we have to examine the i18n&l10n bundles that help translate the keys in the "spring:message" tags into appropriate sentences in Spanish or English.

The i18n&l10n bundles (II)

```
profile.action.2 = Esta es la acción 2 del perfil
profile.compute = Calcular
```


These are the contents of “messages_es.properties”.



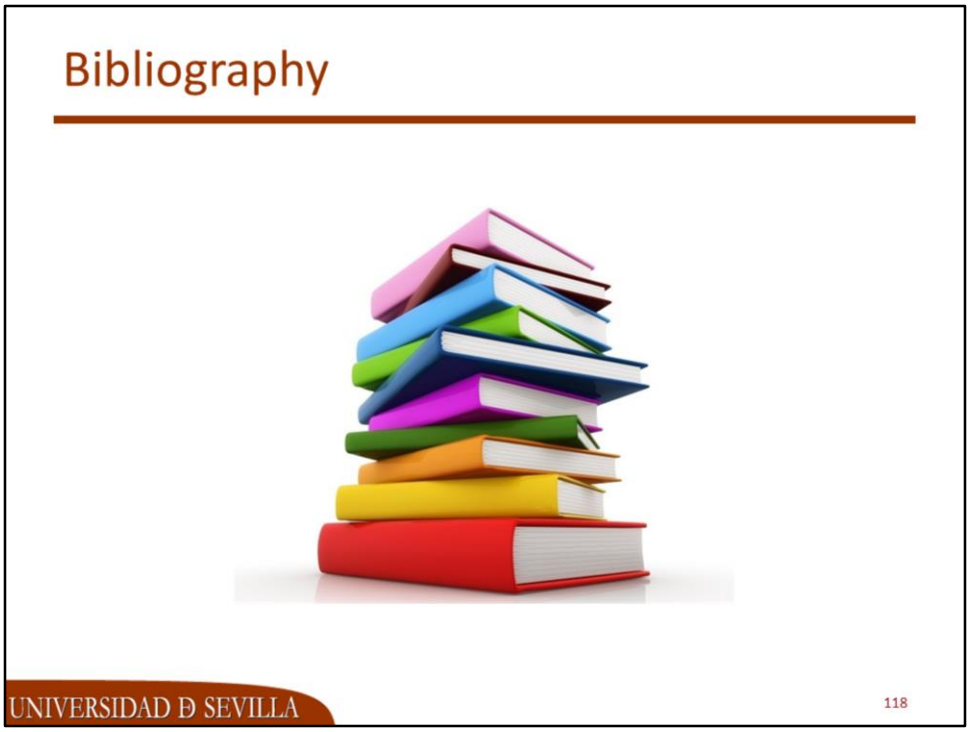
And these are the contents of “messages.properties”.

This is enough for today!

UNIVERSIDAD Ð SEVILLA

117

Today's lecture's been intense. We'll keep learning from new case studies in the next lecture. Please, stay tuned!



We recommend that you should take these lecture notes as your primary source of information. Should you need more information it might be a good idea to take a look at the following bibliography:

Pro Spring MVC
Marten Deinum, Koen Serneels, Colin Yates, Seth Ladd, Christophe Vanfletere
Apress, 2012

This bibliography is available in electronic format for our students at the USE's virtual library. If you don't know how to have access to the USE's virtual library, please, ask our librarians for help.

Time for questions, please

Time for questions, please. Note that we won't update the slides with the questions that are posed in the lectures. Please, attend them and take notes.

Thanks for attending this lecture! See you soon.