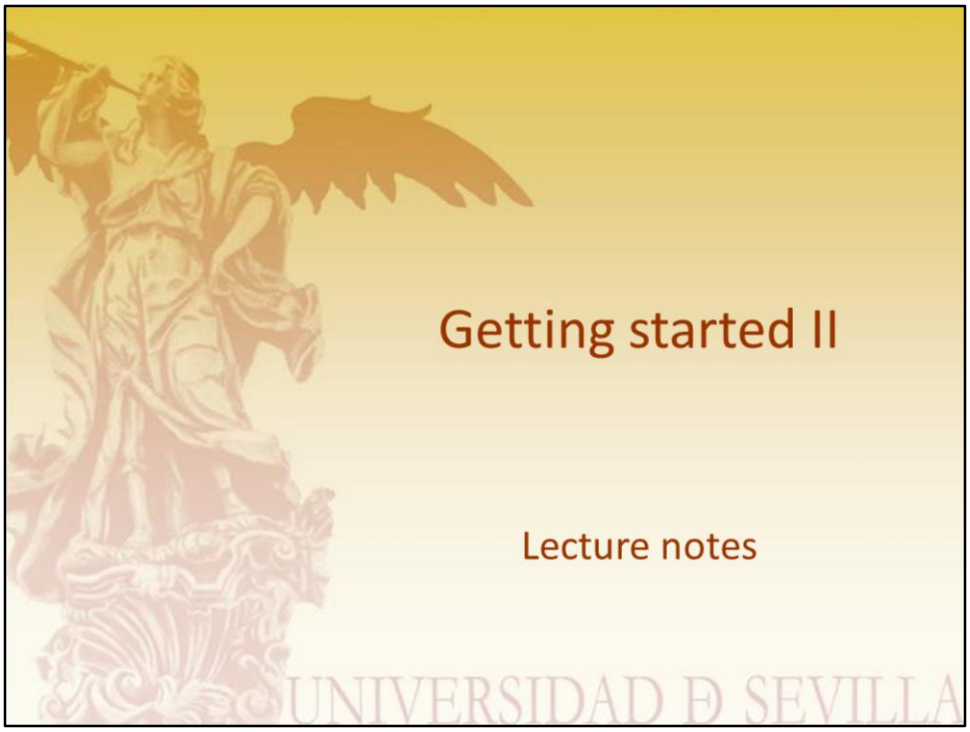


Welcome to this lesson in which our goal's to get started regarding our goal in this subject.

--

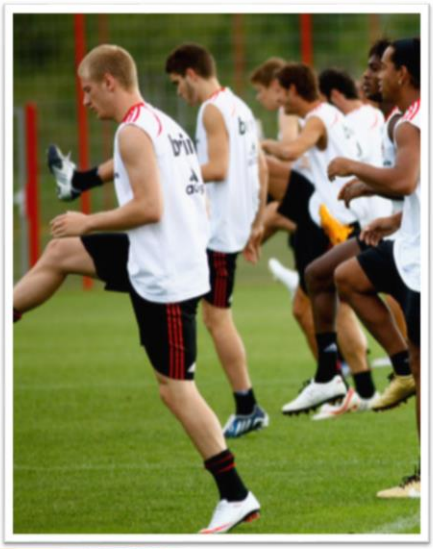
Can you remember last lecture?

UNIVERSIDAD Ð SEVILLA

2

Can you remember the last lecture? We hope so!

Refreshing our minds

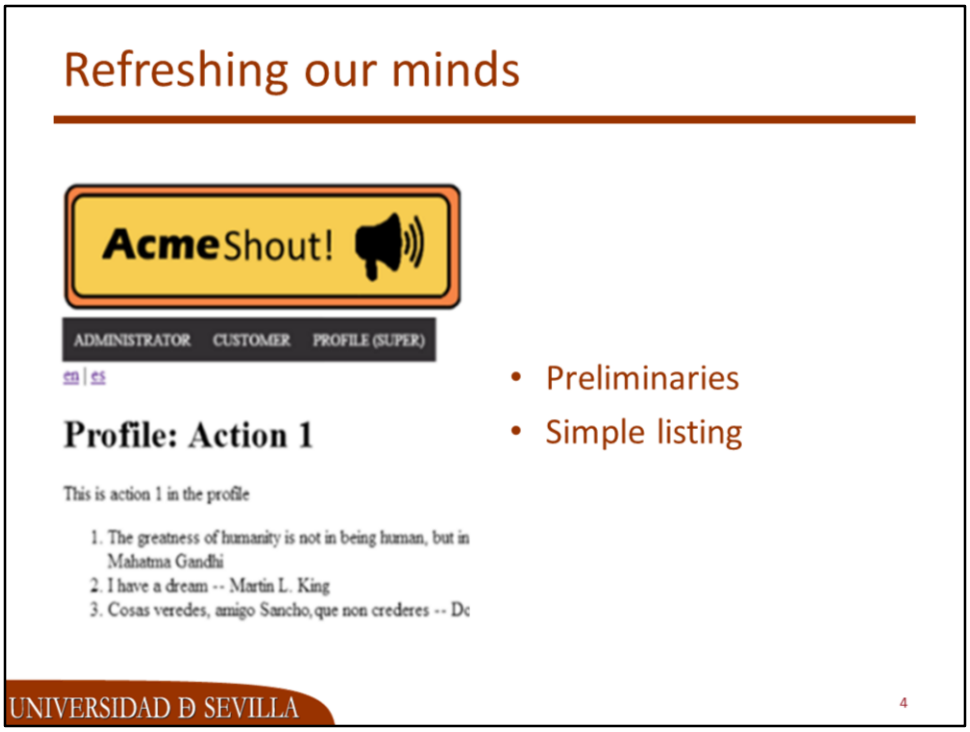


- Preliminaries

UNIVERSIDAD Ð SEVILLA

3

In the last lecture, we learnt some preliminaries regarding instantiating the sample work space, the sample project, creating a database, and getting it all running.



We then reported on Case Study 1, which consisted of a simple listing to show three inspiring quotes.

Refreshing our minds

- Preliminaries
- Simple listing
- Simple form

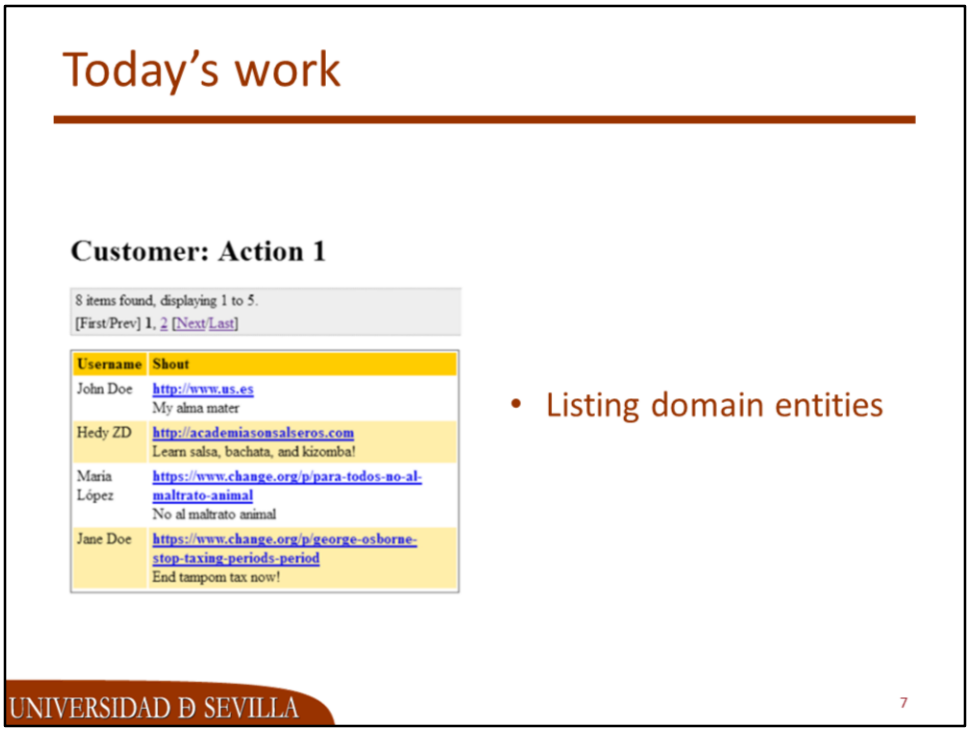
Next, we reported on Case Study 2, in which the goal was to introduce a simple calculator.

What about today?

UNIVERSIDAD Ð SEVILLA

6

What about today's lecture?

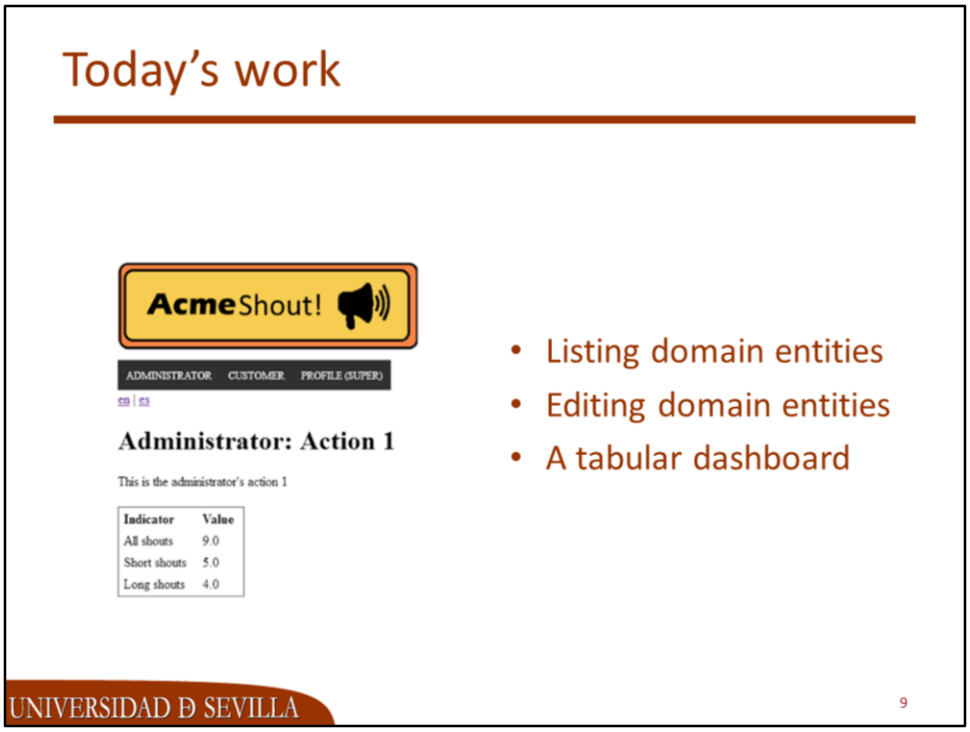


Today, we're going to explore Case Study 3, in which our goal is to list domain entities, shouts in this case.

Today's work

- Listing domain entities
- Editing domain entities

Then, we'll explore Case Study 4, in which the goal is to edit domain entities.



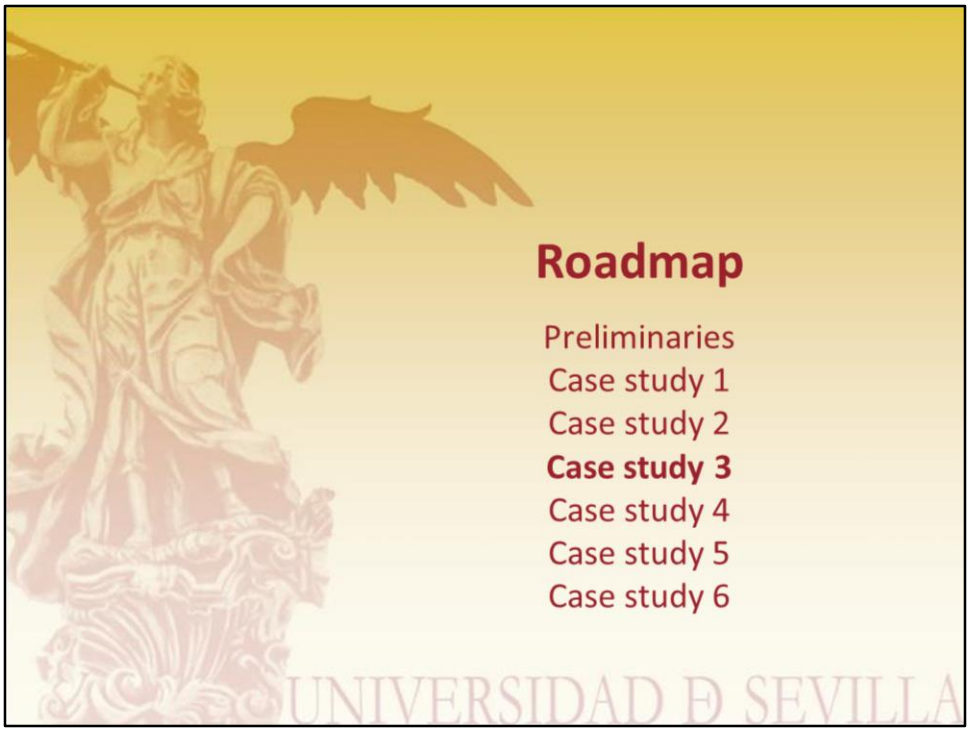
We'll then move on to Case Study 5, in which the goal is to learn how to implement a tabular dashboard.

Today's work

- Listing domain entities
- Editing domain entities
- A tabular dashboard
- A graphic dashboard

Finally, we'll explore Case Study 6, in which the goal's to display the dashboard graphically.

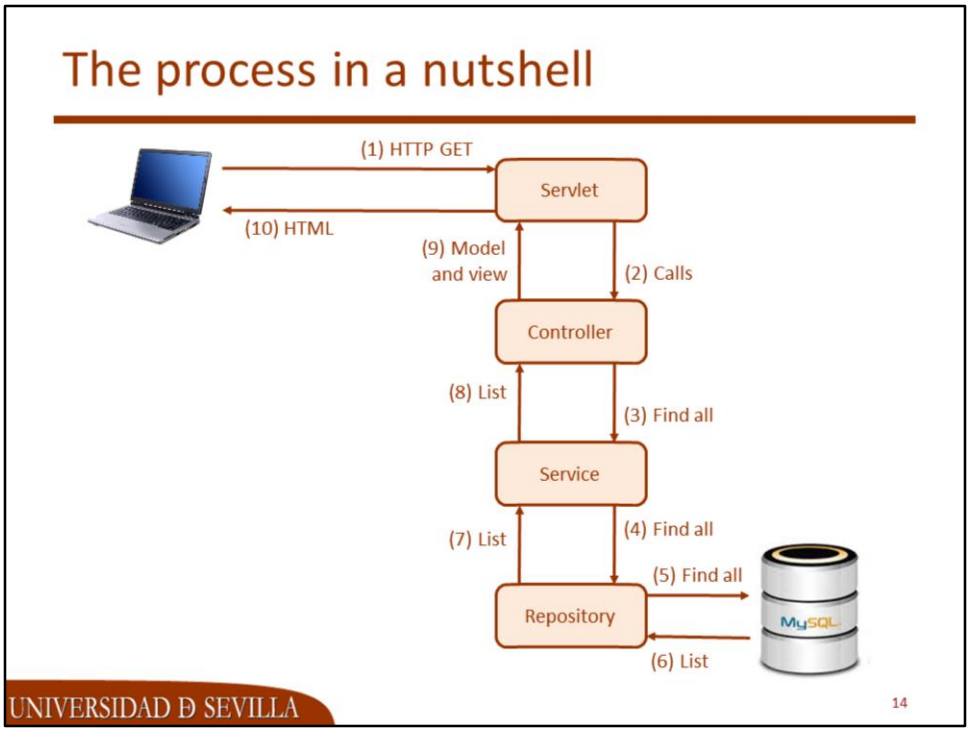
Come on! Let's resume working.



So let's start with Case Study 3.

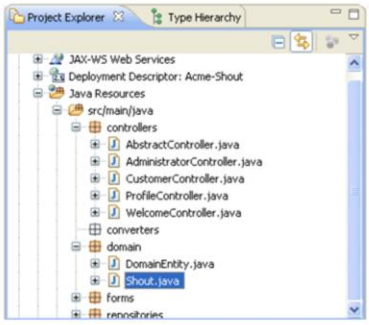
One picture and 1 000 words

Action 1 of the customer's menu must display a listing of shouts that are recorded in the database, which must look more or less like in this slide.



The process is a bit more involved, but not difficult to understand. As usual, it starts with (1) a user who makes an HTTP/GET request from his or her browser, which is (2) intercepted by the servlet, which (3) calls the appropriate controller. The difference is that the controller cannot handle requests that involve persistent objects directly, but must (3) delegate this task to a service, which, in turn, (4) invokes a repository to (5) fetch the appropriate data from the database. The database is expected to (6) return the list of results, which is (7) passed on to the service, which is, in turn, (8) passed on to the controller, which (10) creates a "ModelAndView" object that the servlet uses to (10) create the HTML document that is sent back to the user. In simple cases like our case study, the service is a mere by-pass method which adds very little value; later in this subject, we'll learn that it allows to manage transactions, implement business rules, and so on. So, please, never omit them.

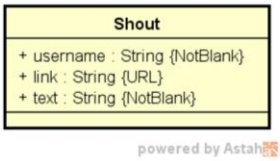
The domain entity (I)



It a Java class that is used to store persistent data

In this case study, we're going to use a domain entity called "Shout". Generally speaking, a domain entity is a Java class that is used to store persistent data.

The domain entity (II)

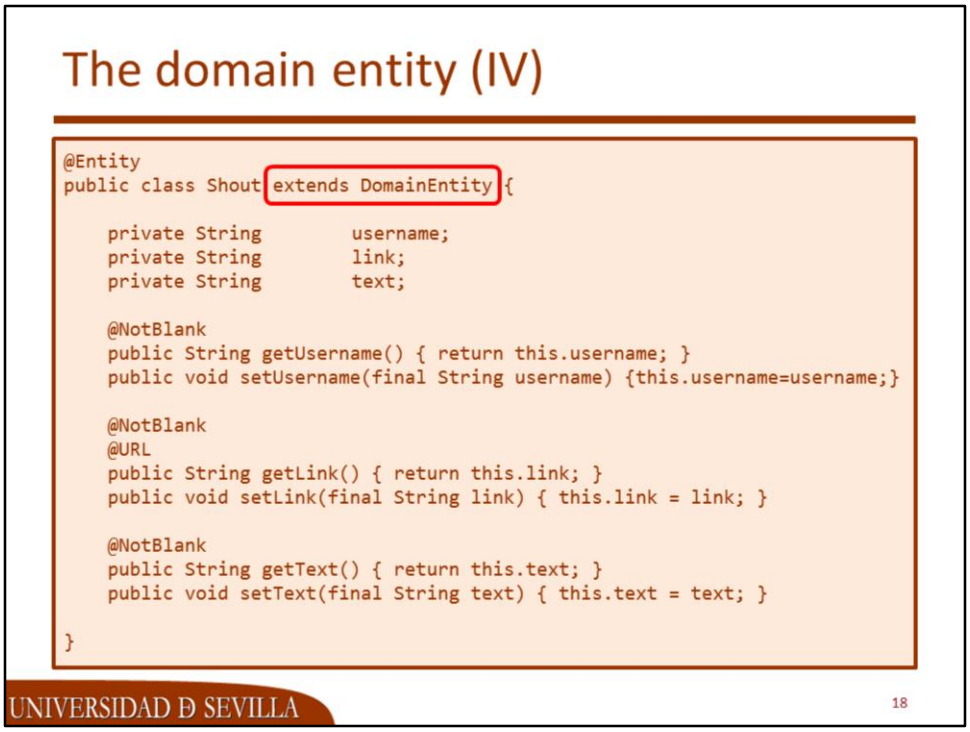


This is the conceptual/domain model of the “Shout” entity. It has three properties to store a username, a link, and a piece of text.

The domain entity (III)

```
@Entity
public class Shout extends DomainEntity {

    private String        username;
    private String        link;
    private String        text;

    @NotBlank
    public String getUsername() { return this.username; }
    public void setUsername(final String username) {this.username=username;}

    @NotBlank
    @URL
    public String getLink() { return this.link; }
    public void setLink(final String link) { this.link = link; }

    @NotBlank
    public String getText() { return this.text; }
    public void setText(final String text) { this.text = text; }

}
```


And this is the implementation. Note that it's a regular Java class that is prepended with the "@Entity" annotation, which makes it persistent, and …

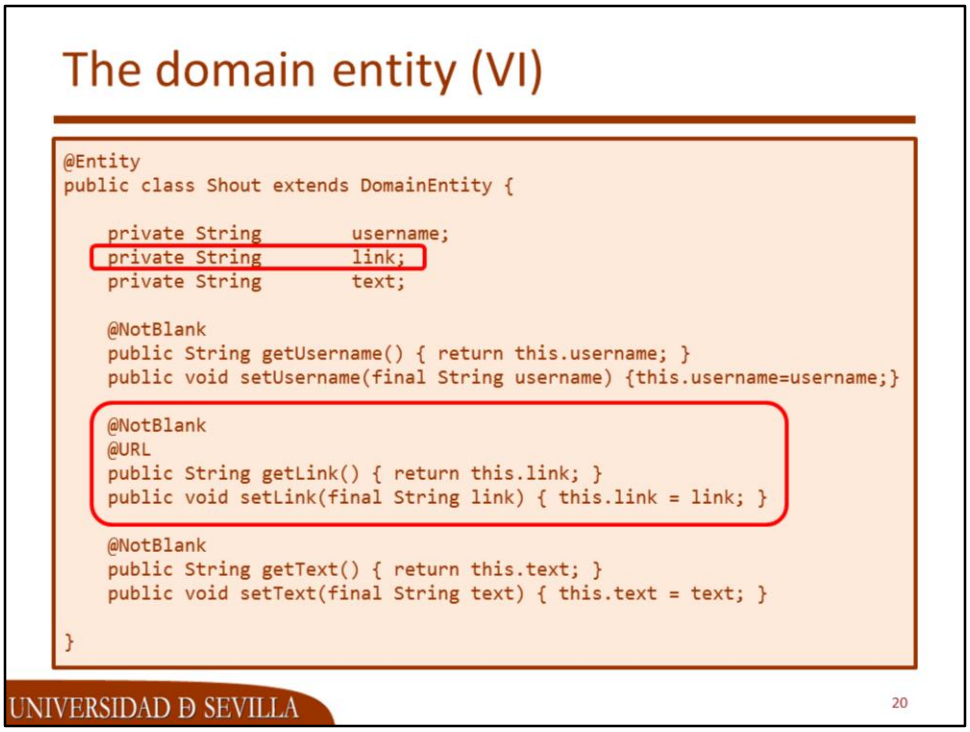


…it extends the “DomainEntity” class, which endows every entity with an integer identifier called “id” and a version number called “version”.

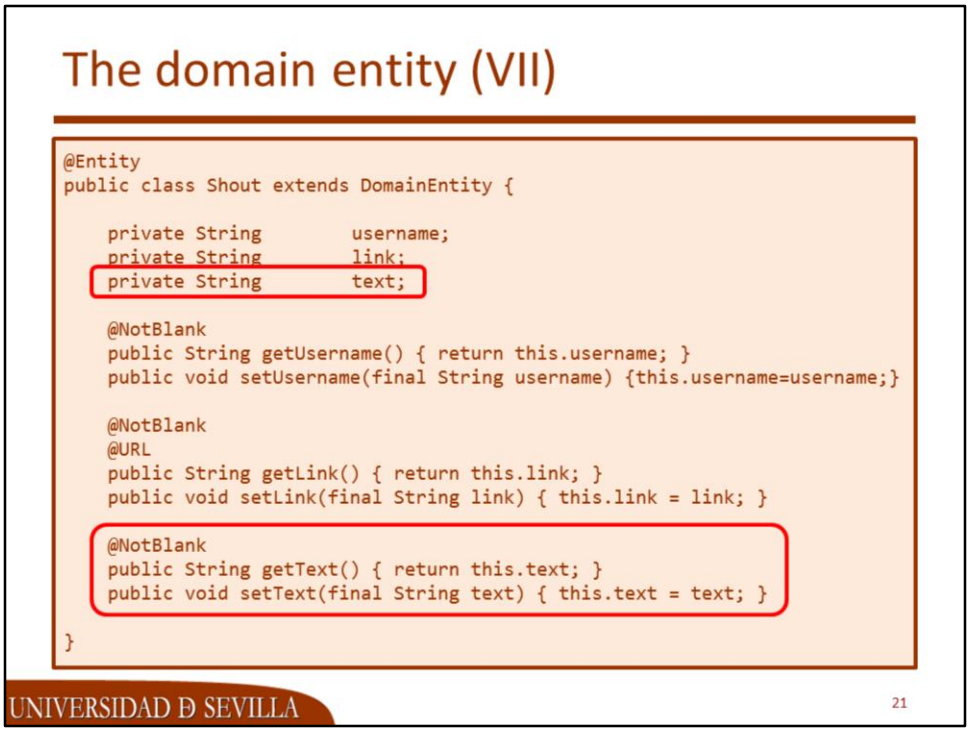
The domain entity (V)

```
@Entity
public class Shout extends DomainEntity {

   private String        username;
   private String        link;
   private String        text;

   @NotBlank
   public String getUsername() { return this.username; }
   public void setUsername(final String username) {this.username=username;}

   @NotBlank
   @URL
   public String getLink() { return this.link; }
   public void setLink(final String link) { this.link = link; }

   @NotBlank
   public String getText() { return this.text; }
   public void setText(final String text) { this.text = text; }

}
```


The body of the class is composed is several property declarations. As usual the properties are introduced by means of private attributes, getters, and setters. This slide shows the "username" property; note that the getter method has an "@NotBlank" annotation, which requires valid shouts to have a non-blank username.

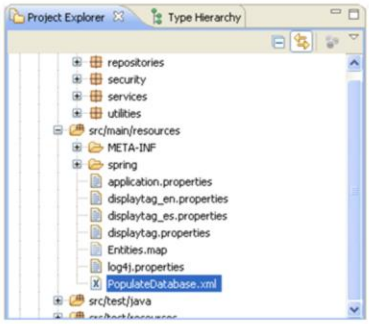


The “link” property is similar, but note that it has two validation constraints: “@NotBlank”, which requires it not to be blank, and “@URL”, which requires it to conform to the pattern of a valid URL.



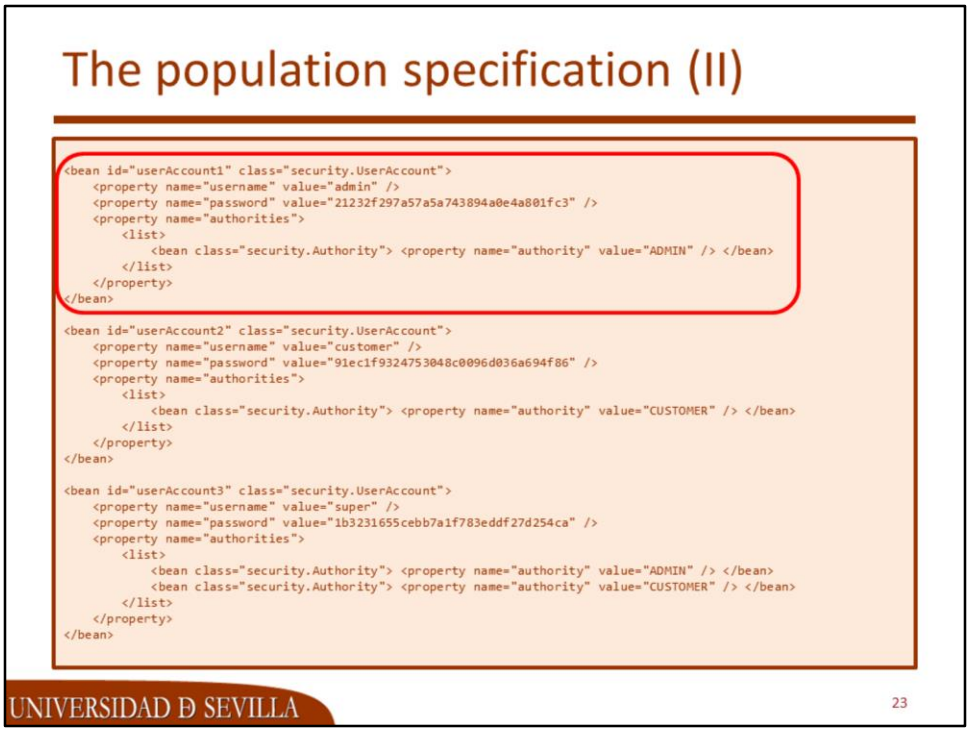
The last property is “text”, which is also required not to be blank.

The population specification (I)



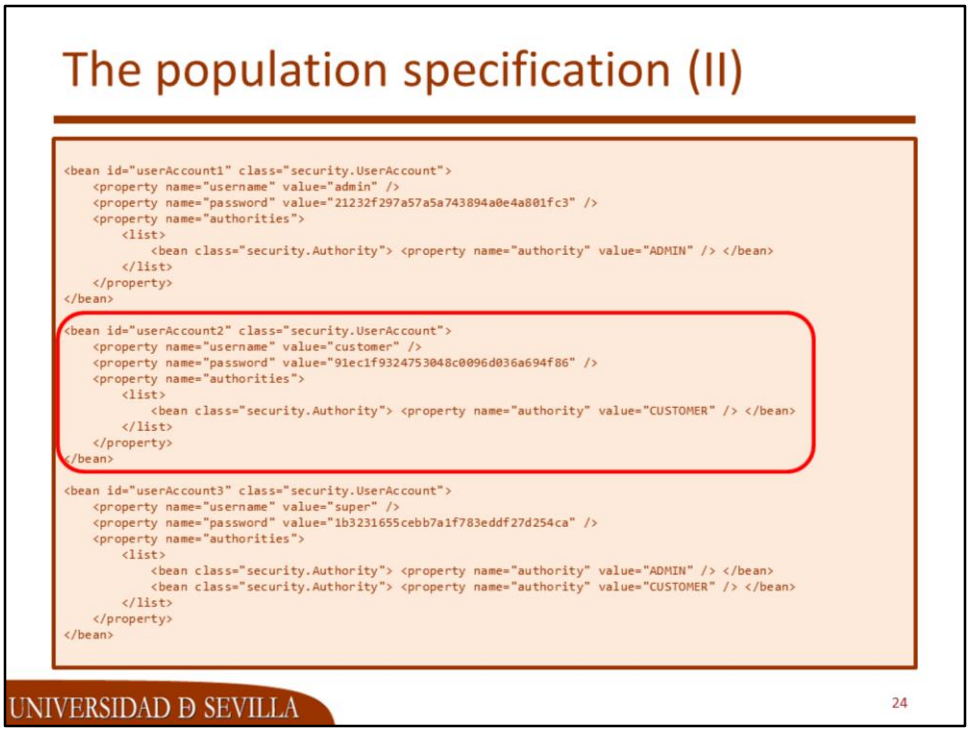
It's an XML file that provides a specification of the entities that are used to populate the database with initial and/or test data

Domain entities are persistent, which means that we have to populate the database with some instances that will help us try our system before we can enter data using the appropriate forms. To persist some domain entities for testing purposes, we need to use a population specification.

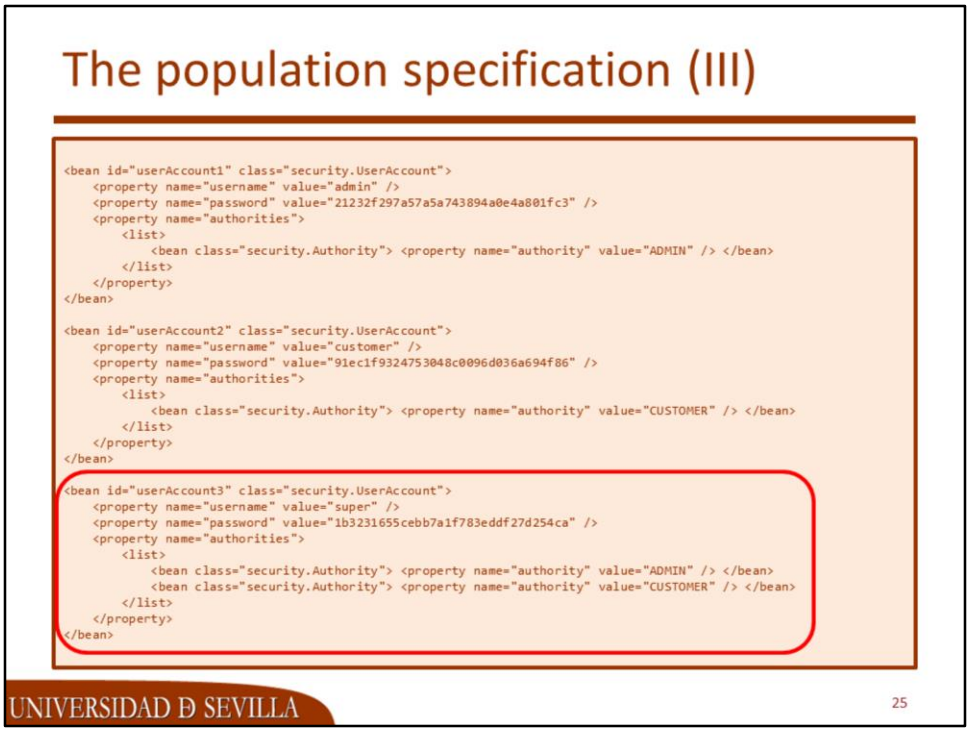


This slide shows the default contents of the “PopulateDatabase.xml” file. It has three user account specifications. The first one’s the default administrator’s account. The user accounts have three properties, namely: the username, the password (which is hashed by means of the MD5 algorithm), and the authorities (which are the roles of the corresponding user).

NOTE: there’s a utility called “HashPassword.java” in the sample project that you can use to hash your passwords. Unless otherwise stated, usernames and passwords are the same. That is, the credentials of the administrator are “admin/admin”.



The second user account corresponds to a user with role “CUSTOMER”. That’s the default non-administrative role in the project template. Later in this subject, we’ll learn how to add additional roles.



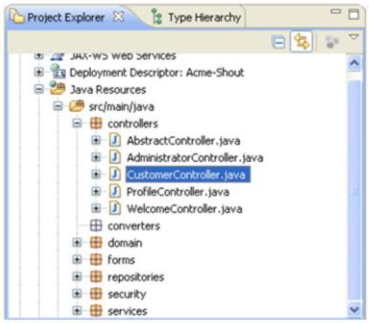
And the third user account corresponds to a user called “super” who has both the “ADMIN” and the “CUSTOMER” roles. This user account’s useful to see all of the options in your system at a glance.

The population specification (III)

```
<bean id="shout1" class="domain.Shout">
    <property name="username" value="John Doe" />
    <property name="link" value="http://www.us.es" />
    <property name="text" value="My alma mater" />
</bean>

<bean id="shout2" class="domain.Shout">
    <property name="username" value="Hedy ZD" />
    <property name="link" value="http://academiasonsalseros.com" />
    <property name="text" value="Learn salsa, bachata, and kizomba!" />
</bean>

<bean id="shout3" class="domain.Shout">
    <property name="username" value="María López" />
    <property name="link" value="https://www.change.org/p/para-todos-no..." />
    <property name="text" value="No al maltrato animal" />
</bean>

<bean id="shout4" class="domain.Shout">
    <property name="username" value="Jane Doe" />
    <property name="link" value="https://www.change.org/p/george-Osborne..." />
    <property name="text" value="End tampom tax now!" />
</bean>
```

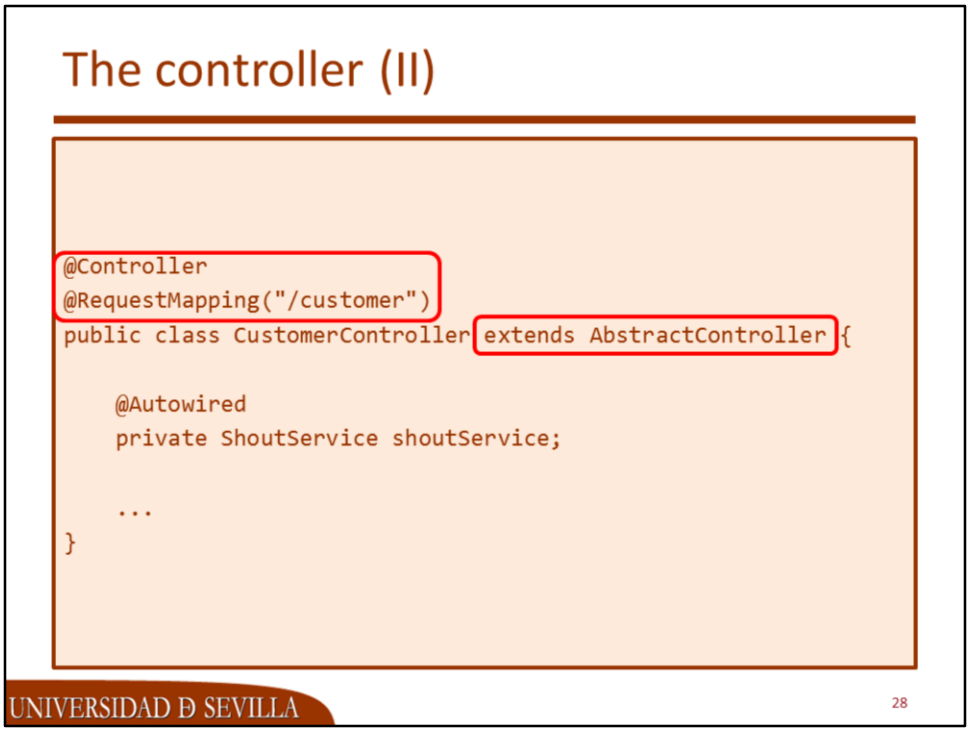

After the default user accounts, you must add additional specifications so as to populate the database with enough shouts for testing purposes. This slide shows a few of them.

The controller (I)

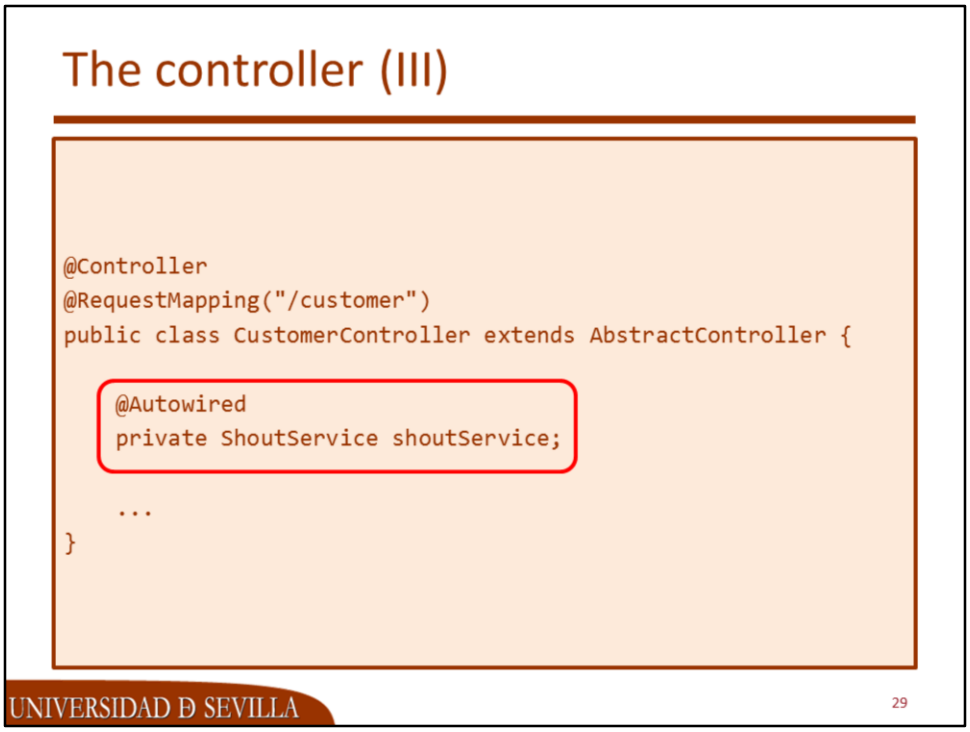


It's a Java class with a method that gets a user request to /customer/action-1.do, invokes the shout service to find all of the shouts, injects them into a model, and returns a view to render the list

It's now time to explore the controller, which is a class with a method that handles requests to URL "/customer/action-1.do". This method invokes the shout service to find all of the shouts, injects them into a model, and returns a view to render the list.



This is the class. Like every other controller, it's prepended with annotations "@Controller" and "@RequestMapping", and it also extends class "AbstractController".



Unlike the controllers in the previous case studies, this includes a private attribute called “shoutService” that is prepended with annotation “@Autowired”. This annotation allows to inject a reference to an instance of the “ShoutService” class into the instances of this controller. Simply put, you can use attribute “shoutService” to have access to an instance of the shout service, but you don’t have to care about creating the instances and wiring them.

The controller (IV)

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

    shouts = this.shoutService.findAll();

    result = new ModelAndView("customer/action-1");
    result.addObject("Shouts", Shouts);

    return result;
}
```


This is the method. Like all other controller methods, it's prepended by the "@RequestMapping" annotation to indicate the action and the request method.

The controller (V)

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

    shouts = this.shoutService.findAll();

    result = new ModelAndView("customer/action-1");
    result.addObject("Shouts", Shouts);

    return result;
}
```

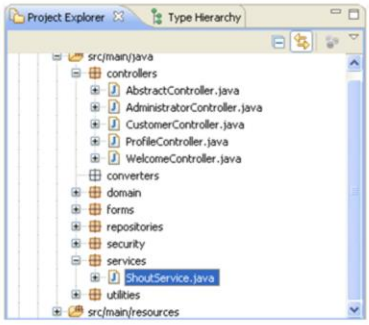

The body of the method is pretty simple, since it first calls a method in the shout service that finds all of the shouts in the database …

The controller (VI)

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

    shouts = this.shoutService.findAll();

    result = new ModelAndView("customer/action-1");
    result.addObject("shouts", shouts);

    return result;
}
```


… and then creates a “ModelAndView” object that is returned.

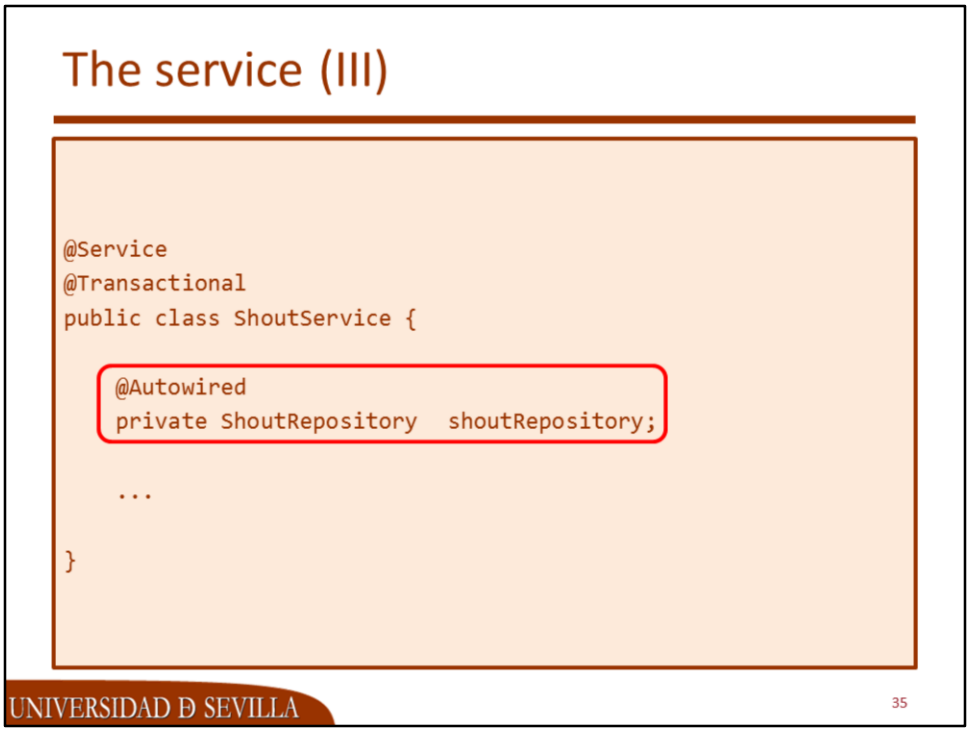
The service (I)



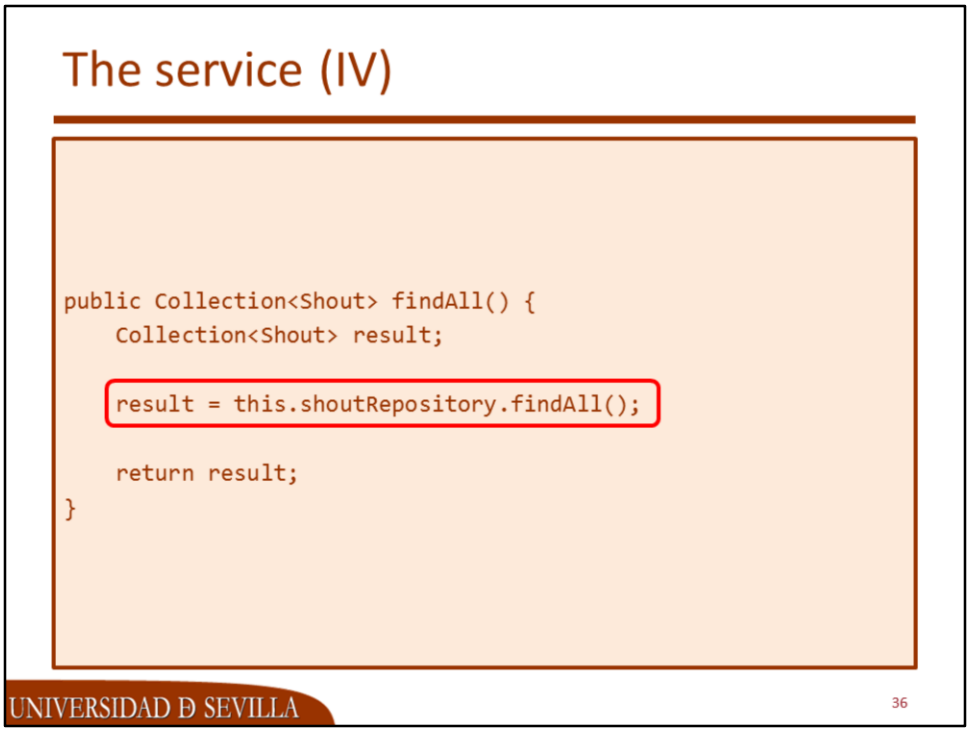
It's a Java class with a method that returns the list of all shouts in the database

Let's explore the service, which is a Java class with a method that returns the list of all shouts in the database.

This is how it looks like. It's a regular Java class that is prepended with annotations "@Service" and "@Transactional". These annotations indicate that the class implements a transactional service, that is, a service whose methods execute completely and commit their changes to the database or fail and rollback the changes.

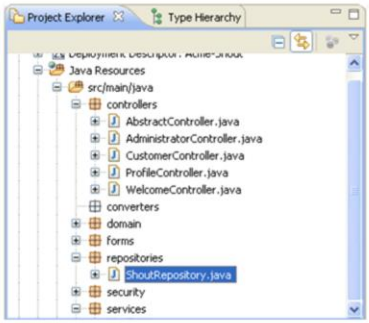


Services typically delegate many tasks to repositories, which are injected by means of autowired private attributes.



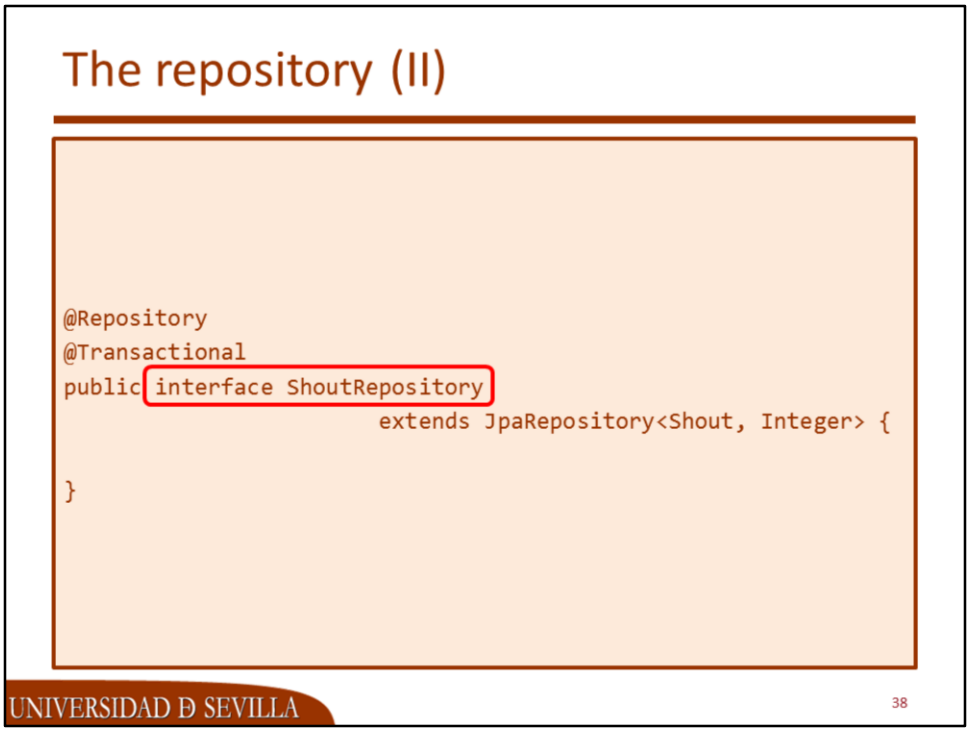
And this is the method. Realise that in simple case studies like this, the method merely by-passes its task to the repository.

The repository (I)

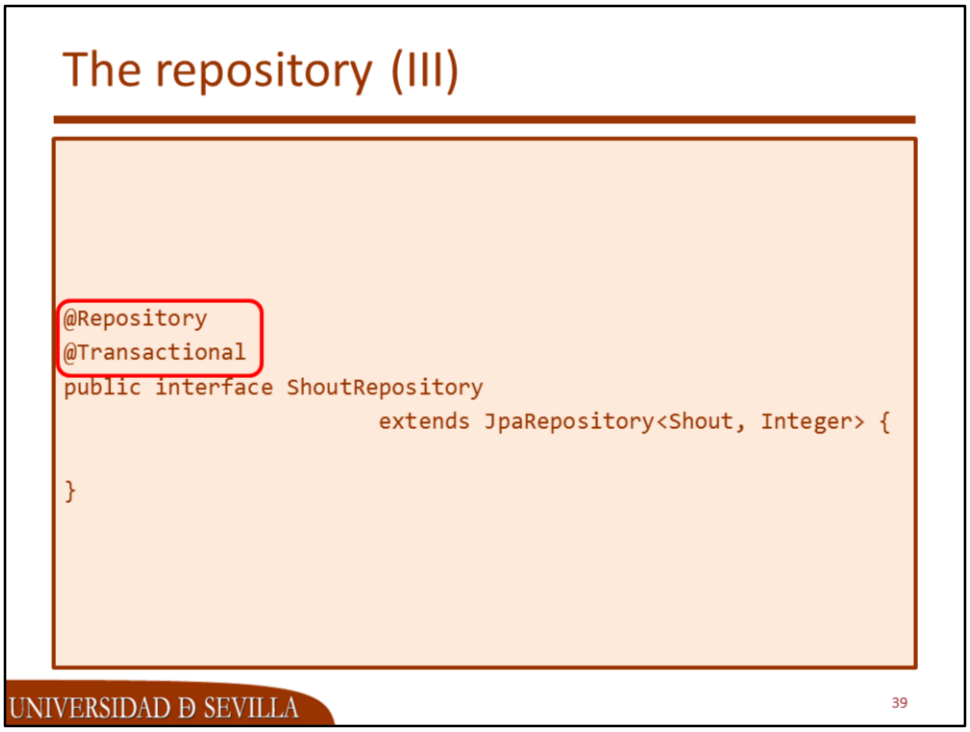


It's a Java interface with CRUD and custom methods to interact with the database

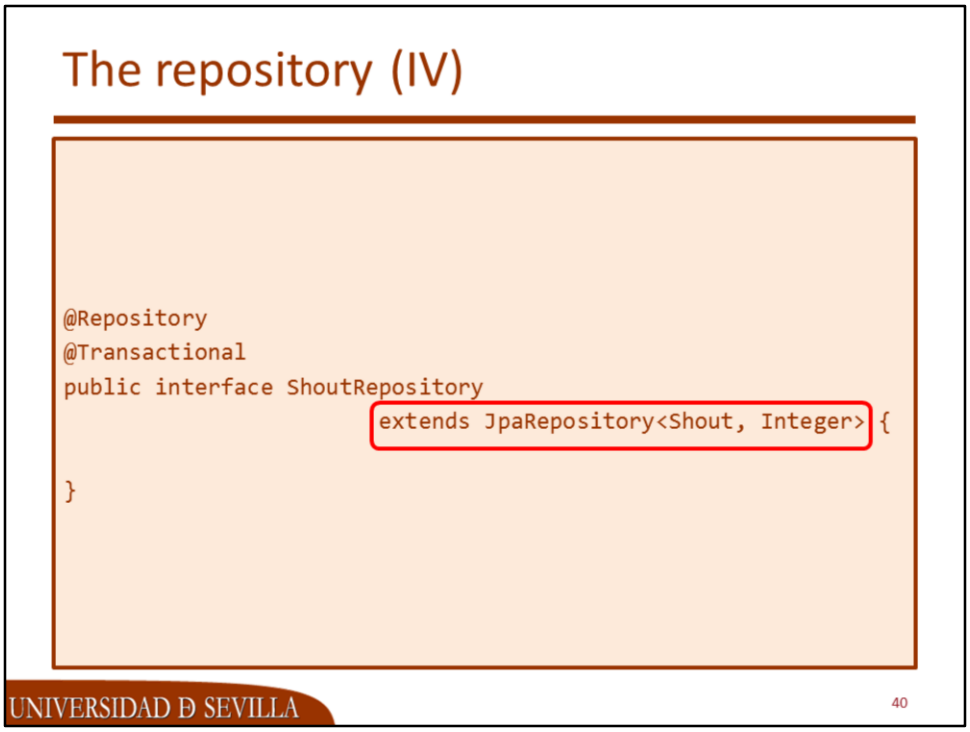
The repository is implemented as a Java interface that provides both CRUD methods and custom methods that interact with the database.



This is the interface. Please, note that it's an interface, not a class. The class is generated from the interface automatically.

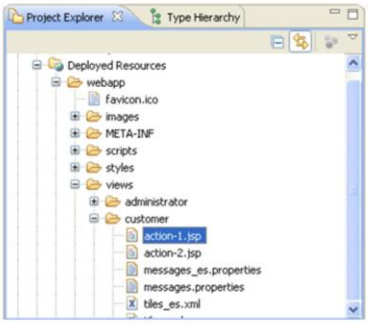


Note, too, that the interface is prepended with the "@Repository" and the "@Transactional" annotations, which indicate that this interface is a transactional repository, that is: its methods either commit their changes to the database or rollback them in case of failure.



The interface extends another interface called “JpaRepository”, which is parametric. You must specify the domain entity with which this repository deals and the type of its identifier. In our sample project, the type of the identifier must be “Integer”. The “JpaRepository” provides you with typical CRUD methods like find, findAll, save, or delete; so, in this case study, we needn’t add any other custom methods.

The view (I)



It's a JSP specification of a user interface to render a list of shouts

It's time to explore the view that we're going to use to render the listing of shouts.

The view (II)

```
<p><spring:message code="customer.action.1" /></p>

<display:table
    pagesize="5" name="shouts" id="row"
    requestURI="customer/action-1.do" >

    <display:column property="username"
                    titleKey="customer.username" />
    <display:column titleKey="customer.shout">
        <strong>
            <a href="${row.link}">
                <jstl:out value="${row.link}" />
             </a>
        </strong> <br/>
        <jstl:out value="${row.text}" />
    </display:column>

</display:table>
```


As usual it starts with a message. There's nothing new here.

The view (III)

```
<p><spring:message code="customer.action.1" /></p>

<display:table
    pagesize="5" name="shouts" id="row"
    requestURI="customer/action-1.do" >

    <display:column property="username"
                    titleKey="customer.username" />
    <display:column titleKey="customer.shout">
        <strong>
            <a href="${row.link}">
                <jstl:out value="${row.link}" />
             </a>
        </strong> <br/>
        <jstl:out value="${row.text}" />
    </display:column>

</display:table>
```


It then has a “display:table” tag, which is used to render a listing. This tag has a lot of attributes, namely: “pagesize”, which indicates the default number of rows per listing, “name”, which indicates the name of the model variable that has the list of entities to be displayed, “id”, which indicates the name of the variable that will be used to iterate over the list of entities, and “requestURI”, which indicates the URL that must be requested to display a new page.

The view (IV)

```
<p><spring:message code="customer.action.1" /></p>

<display:table
    pagesize="5" name="shouts" id="row"
    requestURI="customer/action-1.do" >

    <display:column property="username"
                    titleKey="customer.username" />
    <display:column titleKey="customer.shout">
        <strong>
            <a href="${row.link}">
                <jstl:out value="${row.link}" />
             </a>
        </strong> <br/>
        <jstl:out value="${row.text}" />
    </display:column>

</display:table>
```

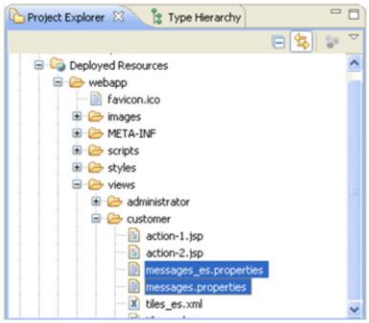

Inside the “display:table” tag, we must put several “display:column” tags to specify how to render the columns of the listing. In its simplest form, the “display:column” tag displays a property of the entities that is specified by means of attribute “property“; the header of the column is indicated by means of the “titleKey” attribute, whose value is a key in an i18n&l10n bundle.

The view (V)

```
<p><spring:message code="customer.action.1" /></p>

<display:table
    pagesize="5" name="shouts" id="row"
    requestURI="customer/action-1.do" >

    <display:column property="username"
                    titleKey="customer.username" />
    <display:column titleKey="customer.shout">
        <strong>
            <a href="${row.link}">
                <jstl:out value="${row.link}" />
             </a>
        </strong> <br/>
        <jstl:out value="${row.text}" />
    </display:column>

</display:table>
```

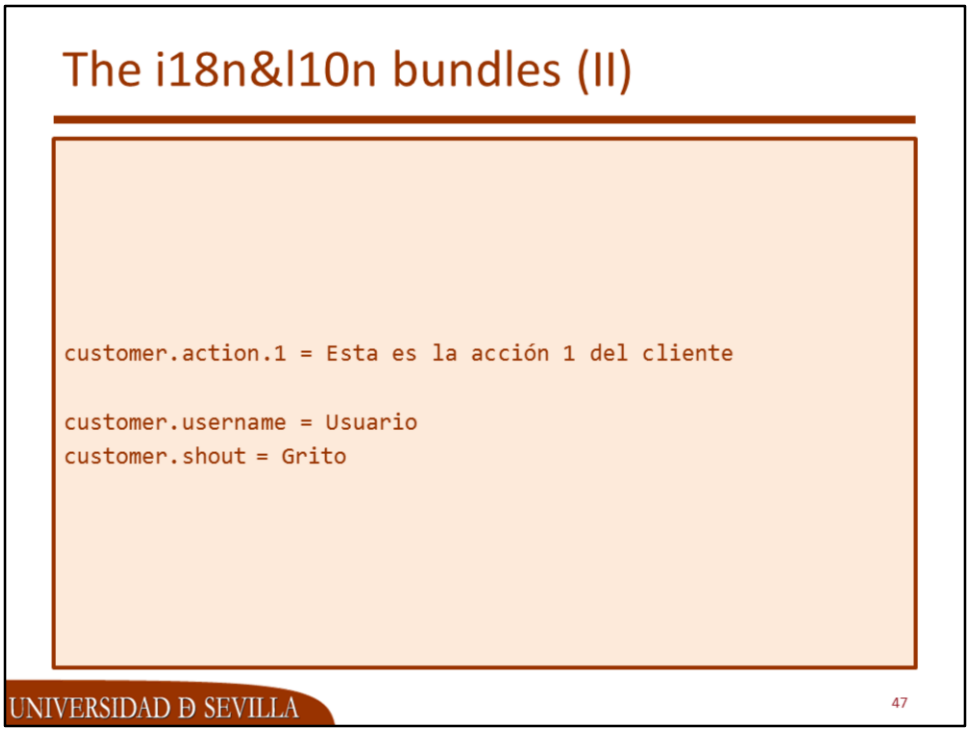

You can easily create custom columns by omitting the “property” attribute and writing a template inside the “display:column” tag. In our example, we show the link in bold face and then the text in regular font. Note that we use expressions like “${row.link}” and “${row.text}” to have access to the link and the text of the shouts through which the “display:table” iterates using variable “row”.

The i18n&l10n bundles (I)



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

Finally, it's time to explore the i18n&l10n bundles.

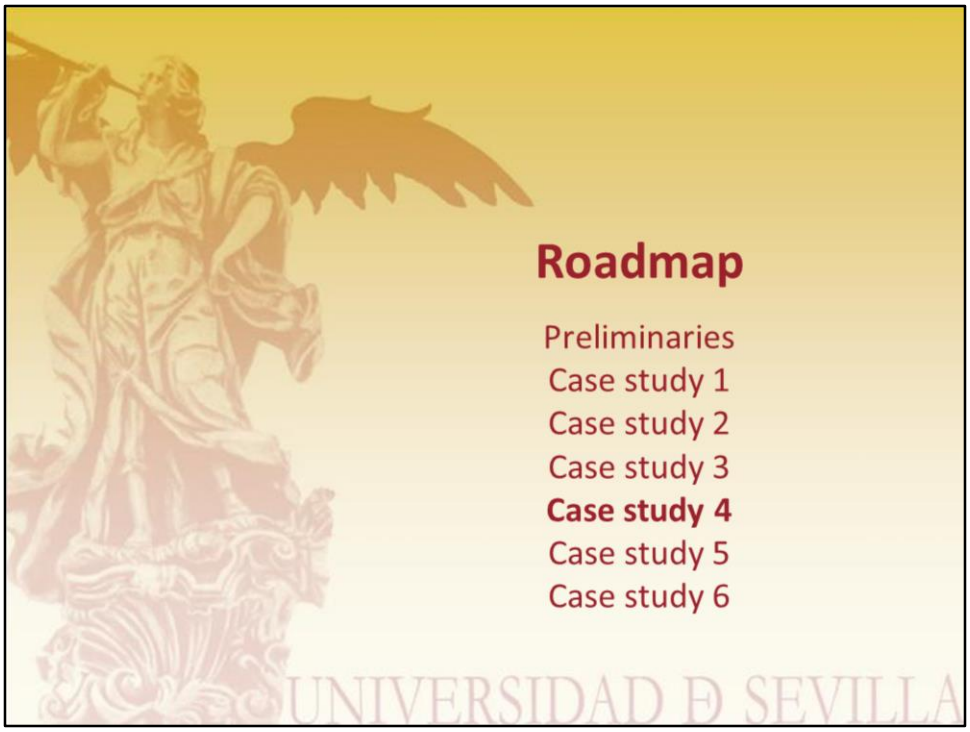


This is the bundle that provides translations into Spanish.

The i18n&l10n bundles (III)

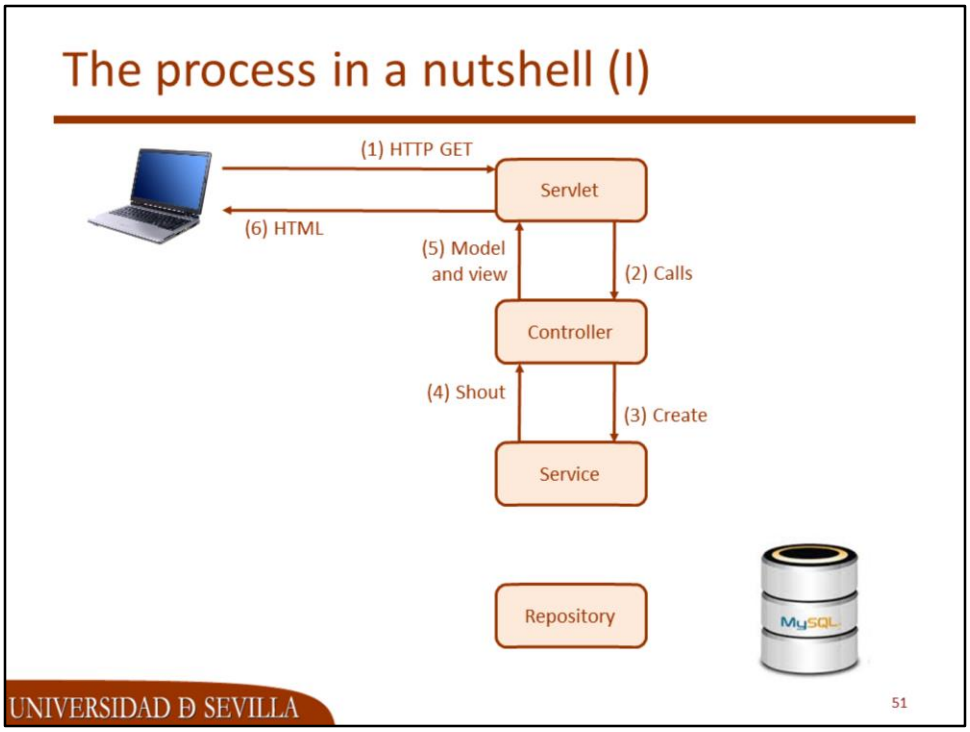
```
customer.action.1 = This is the customer's action 1

customer.username = Username
customer.shout = Shout
```


And this is the bundle that provides translations into English.

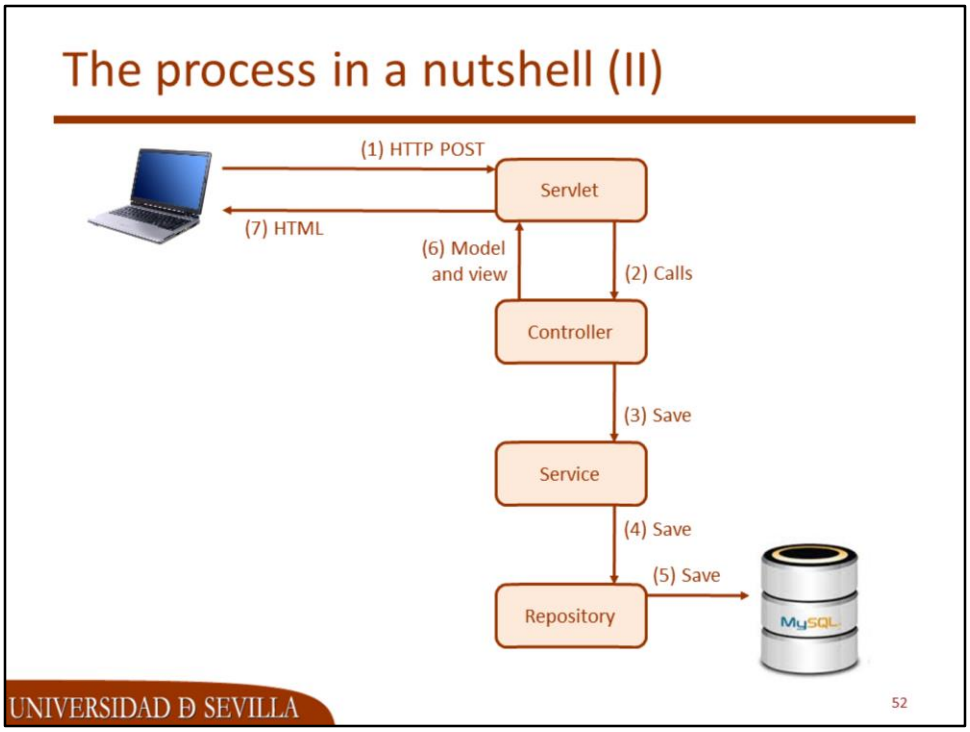


Displaying a listing of shouts is a must… but sooner or later we have to provide a means for the users to enter the shouts using a form. That's our goal in the fourth case study.

This is how the form to enter shouts is expected to look like. There are two input boxes to enter the username and the link, a text area to enter a piece of text, a button to save the shout, and a button to cancel the edition.

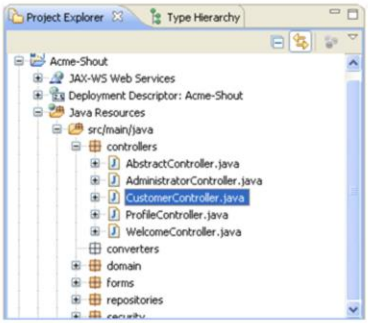


The process is performed in two stages. In the first stage, (1) the user makes his or her browser for and URL like "http://localhost:8080/Acme-Shout/customer/action-2.do" using he HTTP/GET method; (2) the request is intercepted by the servlet, which, in turn, calls the appropriate controller; (3) it then calls the appropriate service to create a new blank shout; (4) the service returns it to the controller, which (5) creates a model and a view that is used by the servlet to generate an HTML document that is returned to the user's browser. Note that no persistence is involved in this stage.



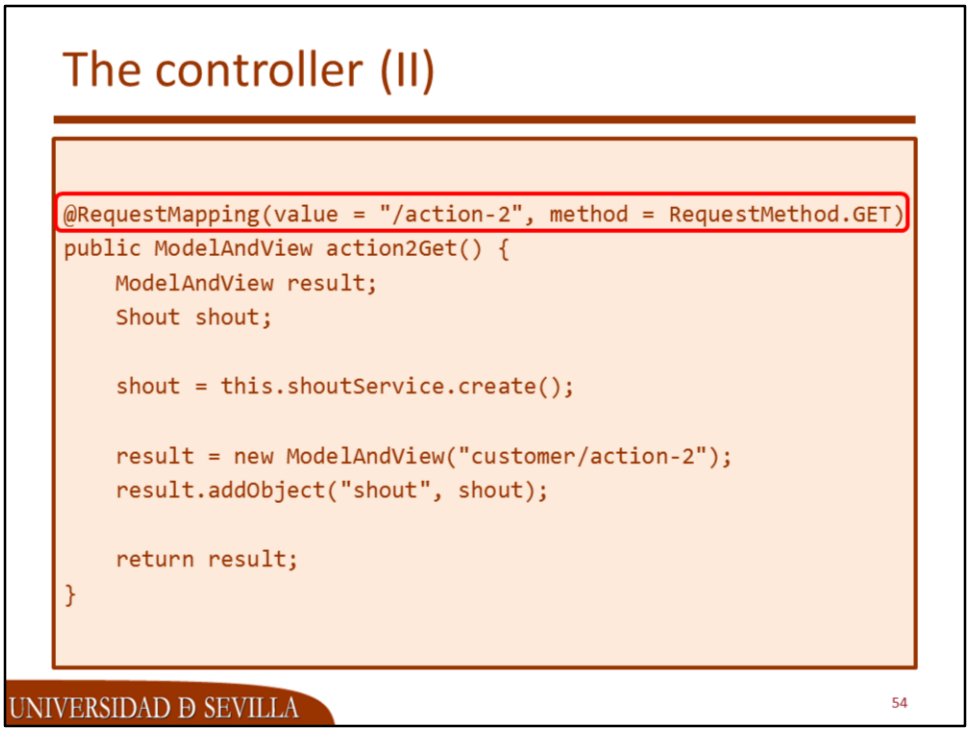
The previous stage returns a form in which the user is expected to enter data. When he or she pressed the "Save" button, (1) the form is submitted to the servlet using the HTTP/POST method; (2) the servlet then locates the appropriate controller and passes the request on to it; (3) the controller delegates saving the corresponding entity to a service, which, in turn, (4) delegates the task to a repository that (5) interacts with the database. If everything's OK, then (6) the controller creates a model and a view that redirects to the listing of shouts; otherwise it creates a model and a view that shows the same form with error messages. In any case, (7) the servlet uses the model and the view to create the HTML document that is sent back to the user.

The controller (I)



It's a Java class with two methods that get user requests to /customer/action-2.do; the first one creates an empty shout and returns a form to edit it; the second one gets the form and a binding, saves the shout, and returns to the listing or displays errors

Let's explore the controller, which is a Java class with two methods that get user requests to "/customer/action-2.do". The first method creates an empty shout and returns a form to edit it; the second one gets the form and a binding, saves the shout, and returns to the listing or displays errors.



This is the code of the first method, which should now be very simple to you. Like every other controller method, it has an "@RequestMapping" annotation to indicate the name of the action and the request method.

The controller (II)

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

    shout = this.shoutService.create();

    result = new ModelAndView("customer/action-2");
    result.addObject("shout", shout);

    return result;
}
```


It then invokes the service to create a new shout.

The controller (III)

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

    shout = this.shoutService.create();

    result = new ModelAndView("customer/action-2");
    result.addObject("shout", shout);

    return result;
}
```


And then creates the “ModelAndView” object that is used by the servlet to create the form that is returned to the user.

The controller (IV)

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

    if (!binding.hasErrors()) {
        this.shoutService.save(shout);
        result = new ModelAndView("redirect:action-1.do");
    } else {
        result = new ModelAndView("customer/action-2");
        result.addObject("shout", shout);
    }

    return result;
}
```


This is the code of the second method, which is prepended with the "@RequestMapping" annotation, as usual.

The controller (V)

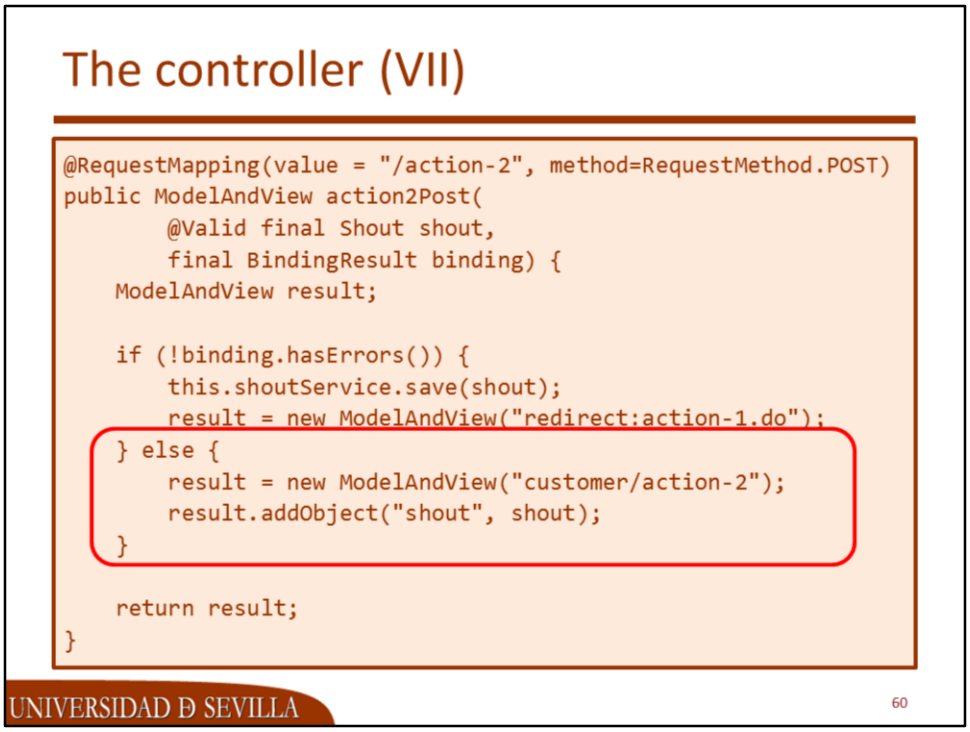
```
@RequestMapping(value = "/action-2", method=RequestMethod.POST)
public ModelAndView action2Post(
        @Valid final Shout shout,
        final BindingResult binding) {
    ModelAndView result;

    if (!binding.hasErrors()) {
        this.shoutService.save(shout);
        result = new ModelAndView("redirect:action-1.do");
    } else {
        result = new ModelAndView("customer/action-2");
        result.addObject("shout", shout);
    }

    return result;
}
```


Since this is a method that handles HTTP/POST requests, it must get two parameters: the object that stores the information that the user's entered in the corresponding form and a binding that stores validation errors.

The controller (VI)

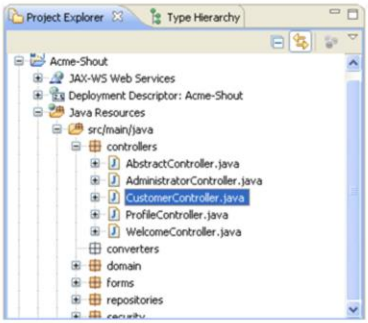
```
@RequestMapping(value = "/action-2", method=RequestMethod.POST)
public ModelAndView action2Post(
        @Valid final Shout shout,
        final BindingResult binding) {
    ModelAndView result;

    if (!binding.hasErrors()) {
        this.shoutService.save(shout);
        result = new ModelAndView("redirect:action-1.do");
    } else {
        result = new ModelAndView("customer/action-2");
        result.addObject("shout", shout);
    }

    return result;
}
```


In the body of the method, we make two cases apart. First, we deal with the case in which there aren't any validation errors. In such cases, we request the shout service to save the new shout and then redirect to "action-1.do" to display the listing of shouts.



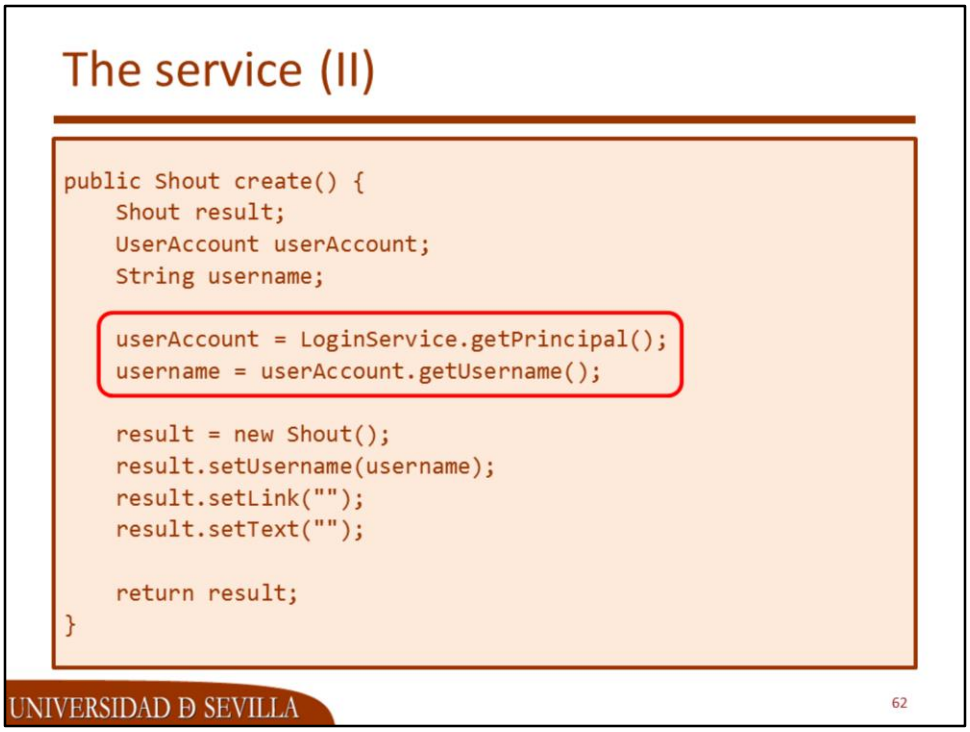
In the opposite case, we create a “ModelAndView” object into which we inject the same object that was passed as a parameter to the method and the same view; recall that when such a “ModelAndView” is returned from a controller, the servlet renders the corresponding errors in the HTML document that is returned to the user.

The service (I)



It's a Java class with a method to create an empty shout and a method to save shouts to the database

The service is pretty simple, since it's a Java class with a method to create an empty shout and another method to save shouts to the database.

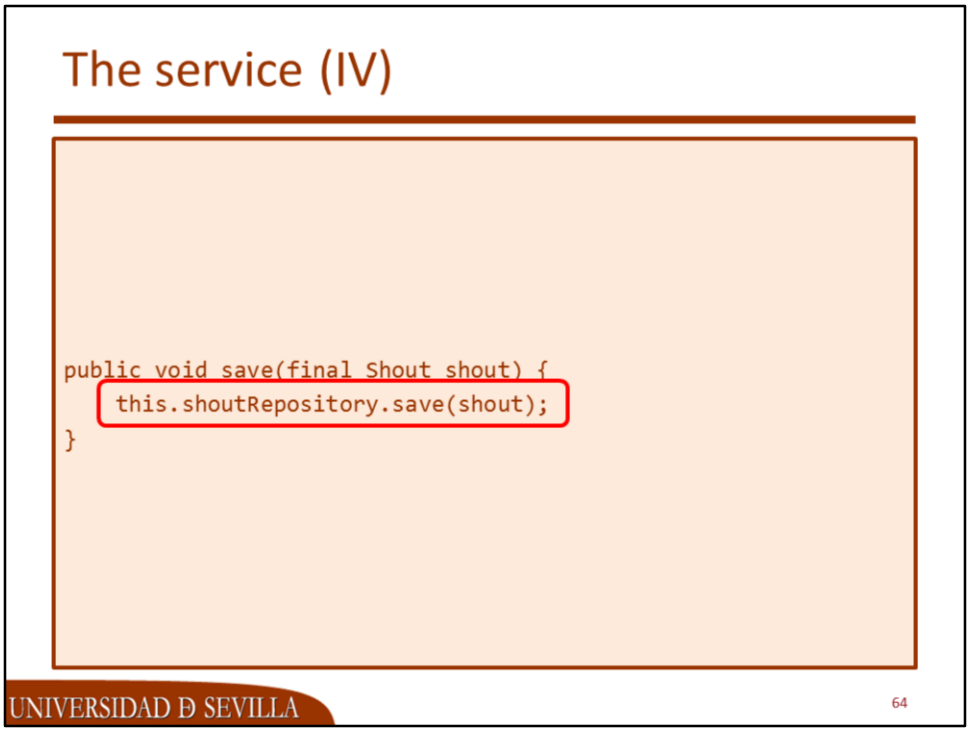


This is the method to create a new shout. Note that we initialise the attributes of the new shout with blank strings, but the username. The idea is to use a service called "LoginService" to retrieve the principal, which is the user account of the user who is making the request that resulted in invoking this method. We use his or her user account to initialise the username in the shout.

The service (III)

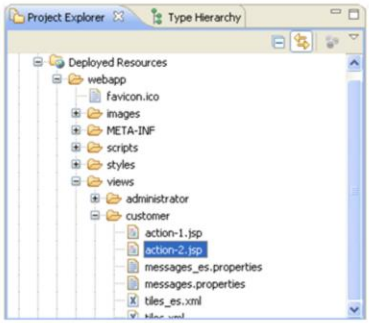
```
public Shout create() {
    Shout result;
    UserAccount userAccount;
    String username;

    userAccount = LoginService.getPrincipal();
    username = userAccount.getUsername();

    result = new Shout();
    result.setUsername(username);
    result.setLink("");
    result.setText("");

    return result;
}
```


The second part of the method is straightforward.



And this is the “save” method, which simply delegates the task to the repository.

The view (I)



It's a JSP view that helps render a user interface with a form to edit a shout

Let's explore the view, which is a JSP specification that helps renders a user interface with a form to edit a shout.

The view (II)

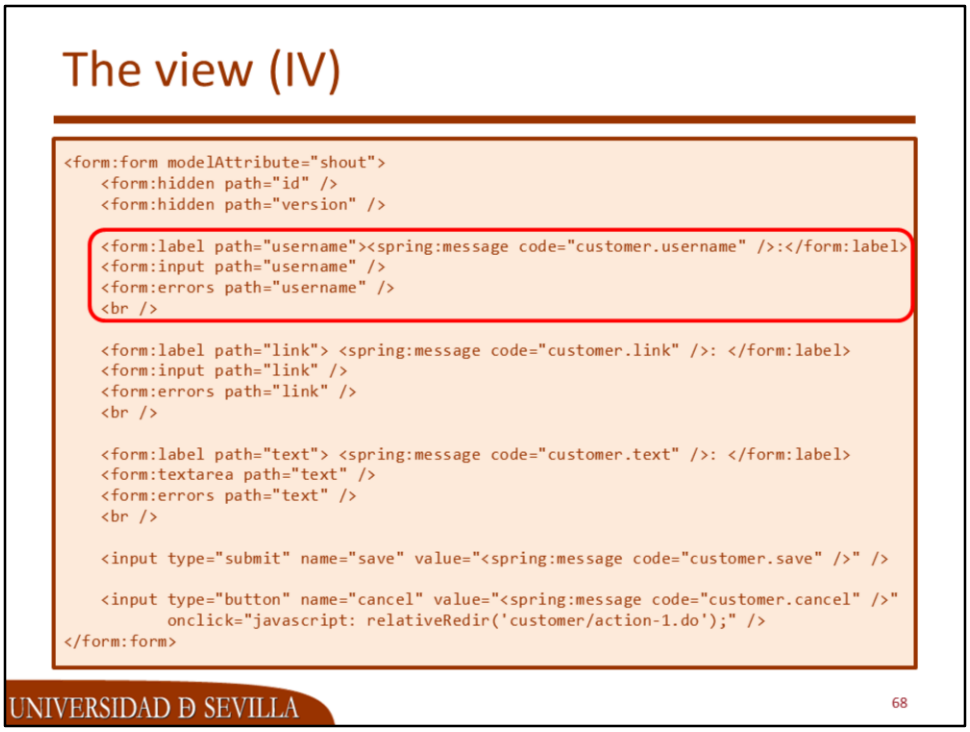
```
<form:form modelAttribute="shout">
    <form:hidden path="id" />
    <form:hidden path="version" />

    <form:label path="username"><spring:message code="customer.username" />:</form:label>
    <form:input path="username" />
    <form:errors path="username" />
    <br />

    <form:label path="link"> <spring:message code="customer.link" />: </form:label>
    <form:input path="link" />
    <form:errors path="link" />
    <br />

    <form:label path="text"> <spring:message code="customer.text" />: </form:label>
    <form:textarea path="text" />
    <form:errors path="text" />
    <br />

    <input type="submit" name="save" value="<spring:message code="customer.save" />" />

    <input type="button" name="cancel" value="<spring:message code="customer.cancel" />"
           onclick="javascript: relativeRedir('customer/action-1.do');" />
</form:form>
```


Since it's an edition form, it starts with a "form:form" tag, which we explored in the previous case study.

The view (III)

```
<form:form modelAttribute="shout">
   <form:hidden path="id" />
   <form:hidden path="version" />

   <form:label path="username"><spring:message code="customer.username" />:</form:label>
   <form:input path="username" />
   <form:errors path="username" />
   <br />

   <form:label path="link"> <spring:message code="customer.link" />: </form:label>
   <form:input path="link" />
   <form:errors path="link" />
   <br />

   <form:label path="text"> <spring:message code="customer.text" />: </form:label>
   <form:textarea path="text" />
   <form:errors path="text" />
   <br />

   <input type="submit" name="save" value="<spring:message code="customer.save" />" />

   <input type="button" name="cancel" value="<spring:message code="customer.cancel" />"
          onclick="javascript: relativeRedir('customer/action-1.do');" />
</form:form>
```


The difference with regard to the previous case study is we're now editing a domain entity. Such entities have an "id" attribute and a "version" attribute that must be stored in the form, but not edited. Such attributes are introduced in the form by means of tag "form:hidden". Simply put, the form stores its values so that they are sent back to your system when the form is submitted, but they are not rendered on the screen, and they cannot be edited by the user.



Then come the attributes to be edited, which are introduced by means of blocks with the following structure:

- A “form:label” tag that indicates the message to be used as a header for the corresponding edition control.
- A “form:input” or a “form:textarea” tag that provides an edition control with a single line or multiple lines, respectively.
- A “form:errors” tag that the system uses to render validation errors automatically.
- A “br” tag to start a new line.

In the slide, we show the block to edit the username.

The view (V)

```
<form:form modelAttribute="shout">
    <form:hidden path="id" />
    <form:hidden path="version" />

    <form:label path="username"><spring:message code="customer.username" />:</form:label>
    <form:input path="username" />
    <form:errors path="username" />
    <br />

    <form:label path="link"> <spring:message code="customer.link" />: </form:label>
    <form:input path="link" />
    <form:errors path="link" />
    <br />

    <form:label path="text"> <spring:message code="customer.text" />: </form:label>
    <form:textarea path="text" />
    <form:errors path="text" />
    <br />

    <input type="submit" name="save" value="<spring:message code="customer.save" />" />

    <input type="button" name="cancel" value="<spring:message code="customer.cancel" />"
           onclick="javascript: relativeRedir('customer/action-1.do');" />
</form:form>
```


There's another similar block to edit the link.

The view (VI)

```
<form:form modelAttribute="shout">
    <form:hidden path="id" />
    <form:hidden path="version" />

    <form:label path="username"><spring:message code="customer.username" />:</form:label>
    <form:input path="username" />
    <form:errors path="username" />
    <br />

    <form:label path="link"> <spring:message code="customer.link" />: </form:label>
    <form:input path="link" />
    <form:errors path="link" />
    <br />

    <form:label path="text"> <spring:message code="customer.text" />: </form:label>
    <form:textarea path="text" />
    <form:errors path="text" />
    <br />

    <input type="submit" name="save" value="<spring:message code="customer.save" />" />

    <input type="button" name="cancel" value="<spring:message code="customer.cancel" />"
           onclick="javascript: relativeRedir('customer/action-1.do');" />
</form:form>
```


And a similar block to edit the text.

The view (VII)

```
<form:form modelAttribute="shout">
    <form:hidden path="id" />
    <form:hidden path="version" />

    <form:label path="username"><spring:message code="customer.username" />:</form:label>
    <form:input path="username" />
    <form:errors path="username" />
    <br />

    <form:label path="link"> <spring:message code="customer.link" />: </form:label>
    <form:input path="link" />
    <form:errors path="link" />
    <br />

    <form:label path="text"> <spring:message code="customer.text" />: </form:label>
    <form:textarea path="text" />
    <form:errors path="text" />
    <br />

    <input type="submit" name="save" value="<spring:message code="customer.save" />" />

    <input type="button" name="cancel" value="<spring:message code="customer.cancel" />"
           onclick="javascript: relativeRedir('customer/action-1.do');" />
</form:form>
```

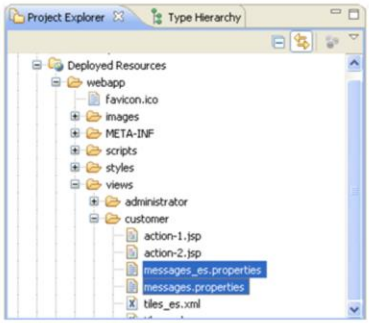

Then comes the submit button, with which you should be familiar now.

The view (VIII)

```
<form:form modelAttribute="shout">
    <form:hidden path="id" />
    <form:hidden path="version" />

    <form:label path="username"><spring:message code="customer.username" />:</form:label>
    <form:input path="username" />
    <form:errors path="username" />
    <br />

    <form:label path="link"> <spring:message code="customer.link" />: </form:label>
    <form:input path="link" />
    <form:errors path="link" />
    <br />

    <form:label path="text"> <spring:message code="customer.text" />: </form:label>
    <form:textarea path="text" />
    <form:errors path="text" />
    <br />

    <input type="submit" name="save" value="<spring:message code="customer.save" />" />

    <input type="button" name="cancel" value="<spring:message code="customer.cancel" />"
           onclick="javascript: relativeRedir('customer/action-1.do');" />
</form:form>
```


And the cancel button, which differs from the previous one. Note that the “Save” button submits the form to your system; conversely, the “Cancel” button simply redirects your browser to the listing of shouts using the built-in JavaScript function called “relativeRedir”.

The i18n&l10n bundles (I)



They're files that help translate keys into messages in Spanish or English

And last, but not least, it's time to explore the i18n&l10n bundles that help us translate keys into Spanish and English messages by means of the "spring:message" tag.

The i18n&l10n bundles (II)

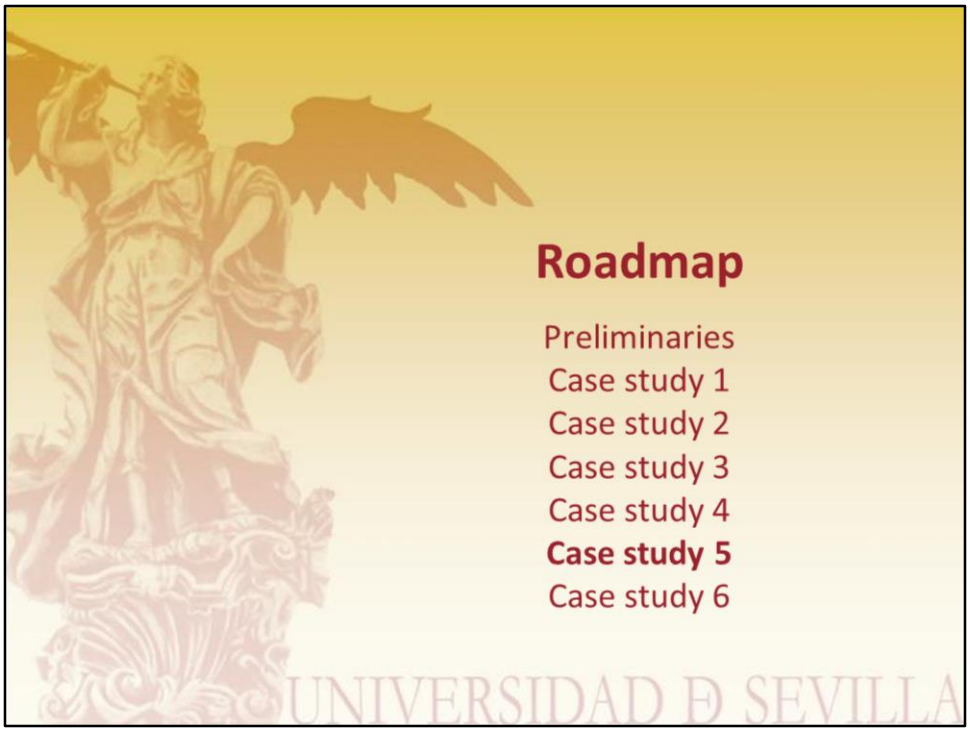
```
customer.action.2 = Esta es la acción 2 del cliente

customer.username = Usuario
customer.link = Enlace
customer.text = Texto

customer.save = Salvar
customer.cancel = Cancelar
```


This is the bundle that provides the Spanish translations.

The i18n&l10n bundles (III)

```
customer.action.2 = This is the customer's action 2

customer.username = Username
customer.link = Link
customer.text = Text

customer.save = Save
customer.cancel = Cancel
```

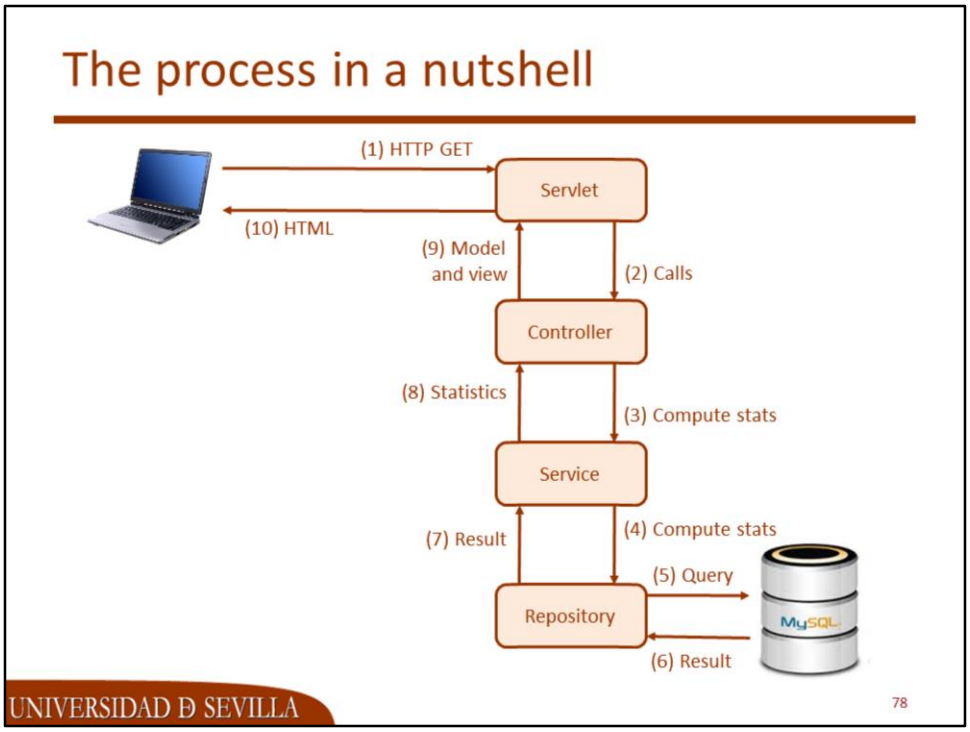

And this is the bundle that provides the English translations.



Great! So far, we've learnt how to list and edit form objects and domain entities. It's time to learn something new.

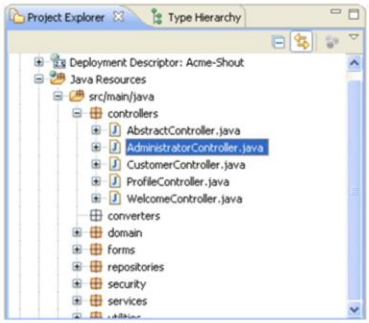
One picture and 1 000 words

This slide shows how action 1 in the administrator's menu looks like. It's a dashboard with three indicators: the count of all shouts, the count of short shouts (shouts with at most 25 characters), and the count of long shouts (shouts with more than 25 characters).



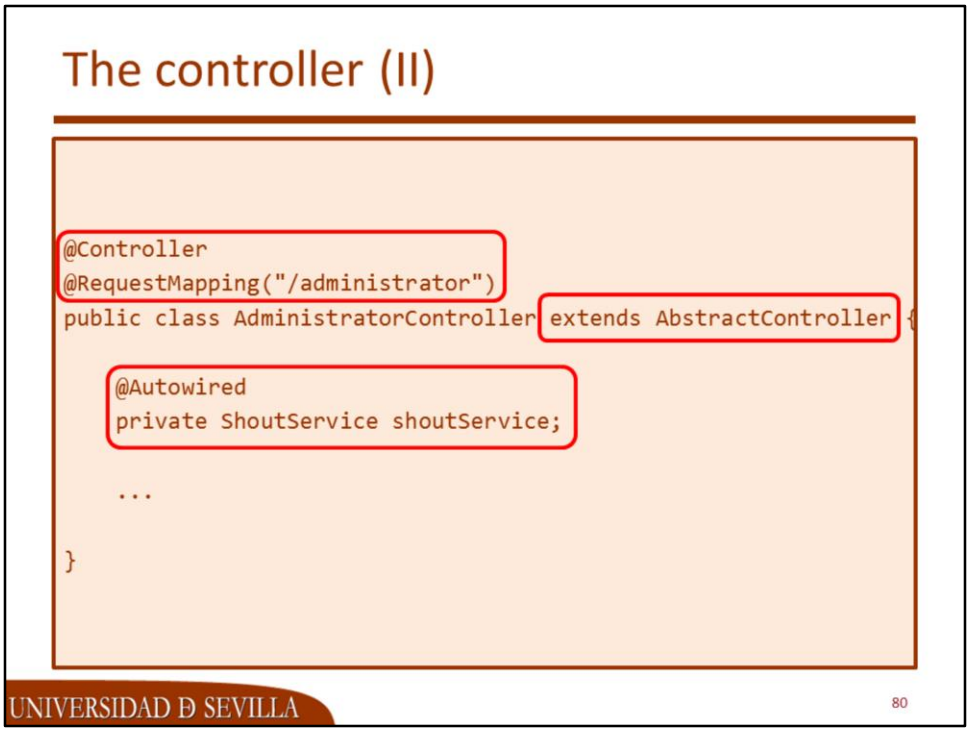
This slide shows the process in a nutshell. Interactions (1) and (2) must sound very familiar to you now. Interaction (3) is initiated by the controller when it requests the service to compute the indicators (which are collectively referred to as statistics); interaction (4) simply delegates computing the indicators to the repository, which (5) issues several queries to the database. The database (6) returns results to the repositories, which are, in turn, (7) returned to the service, which computes the statistics and passes them on to the controller. The controller, as usual, (9) creates a model and a view that are used by the servlet to create the HTML document that is returned to the user's browser.

The controller (I)



It's a Java class with a method that serves requests to /administrator/action-1.do by calling a service to compute some stats, then creating a model, and selecting the appropriate view to render them

Let's start with the controller, which is a Java class with a method that serves requests to "/administrator/action-1.do" by calling a service to compute some statistics, then creating a model, and finally selecting the appropriate view to render them.



There's little new regarding the controller: it's a regular class with the "@Controller" and the "@RequestMapping" annotations, it extends class "AbstractController", and has an autowired private attribute that allows its methods to have access to the shout service.

The controller (III)

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

    statistics = this.shoutService.computeStatistics();

    result = new ModelAndView("administrator/action-1");
    result.addObject("statistics", statistics);

    return result;
}
```


This is the method that serves requests to “action-1”. As usual it’s prepended with a “@RequestMapping” annotation that should be very familiar to you.

The controller (IV)

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

    statistics = this.shoutService.computeStatistics();

    result = new ModelAndView("administrator/action-1");
    result.addObject("statistics", statistics);

    return result;
}
```

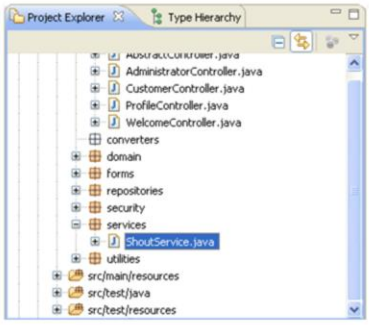

As expected, the method just delegates the computation of the statistics to the corresponding service…

The controller (V)

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

    statistics = this.shoutService.computeStatistics();

    result = new ModelAndView("administrator/action-1");
    result.addObject("statistics", statistics);

    return result;
}
```

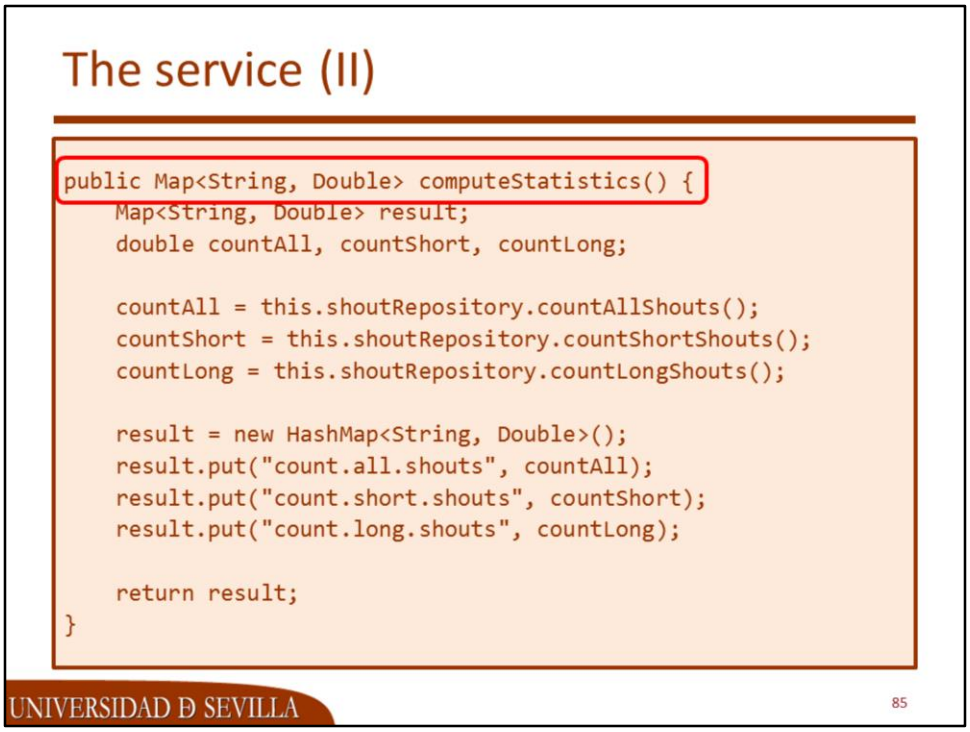

… and then creates the appropriate “ModelAndView” object, which is returned.

The service (I)



It's a Java class with a method that uses the repository to compute some statistics

The service's a bit more interesting.



It has a method called “computeStatistics” that returns the statistics as a map in which the indicators are identified by name and mapped onto their corresponding values.

The service (III)

```
public Map<String, Double> computeStatistics() {
    Map<String, Double> result;
    double countAll, countShort, countLong;

    countAll = this.shoutRepository.countAllShouts();
    countShort = this.shoutRepository.countShortShouts();
    countLong = this.shoutRepository.countLongShouts();

    result = new HashMap<String, Double>();
    result.put("count.all.shouts", countAll);
    result.put("count.short.shouts", countShort);
    result.put("count.long.shouts", countLong);

    return result;
}
```

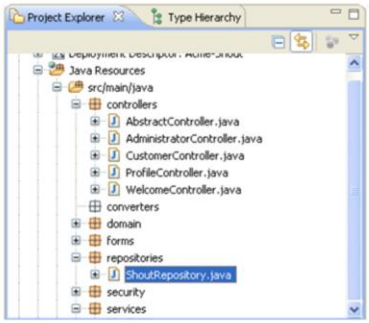

The service first uses the shout repository to compute the values of the indicators.

The service (IV)

```
public Map<String, Double> computeStatistics() {
    Map<String, Double> result;
    double countAll, countShort, countLong;

    countAll = this.shoutRepository.countAllShouts();
    countShort = this.shoutRepository.countShortShouts();
    countLong = this.shoutRepository.countLongShouts();

    result = new HashMap<String, Double>();
    result.put("count.all.shouts", countAll);
    result.put("count.short.shouts", countShort);
    result.put("count.long.shouts", countLong);

    return result;
}
```

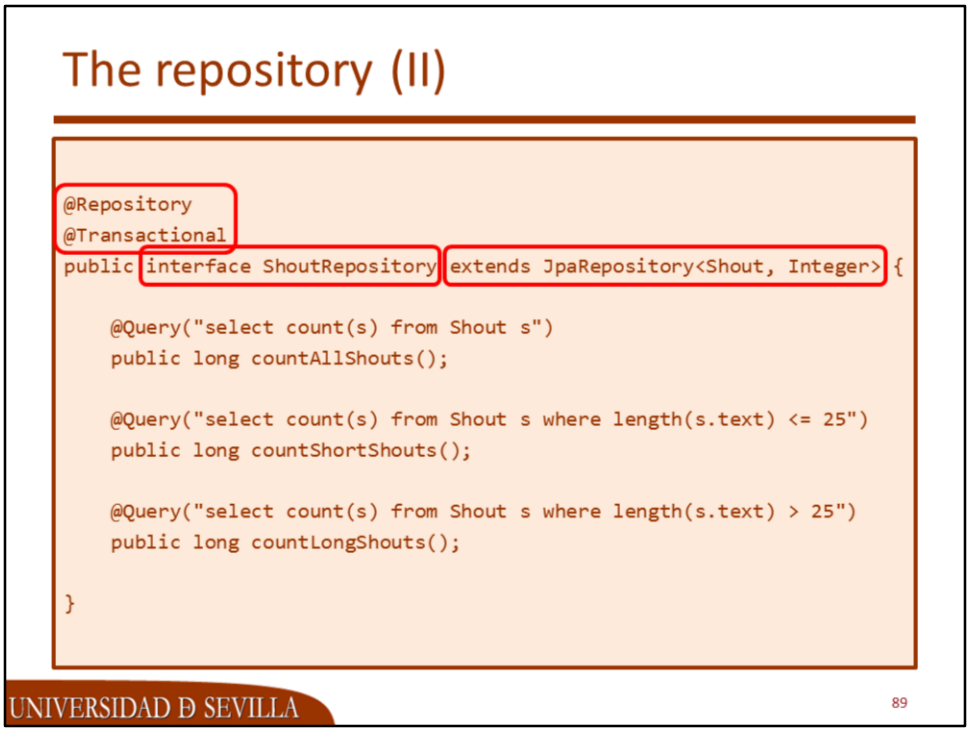

And then stores them in the map that is returned.

The repository (I)



It's a Java interface with some custom methods to execute queries that compute the indicators

The key in this case study is the repository, which provides some custom methods to execute queries that compute the indicators.



Here is it. So far, you must be familiar with the "@Repository" and the "@Transactional" annotations, with the fact that the repository is implemented as an interface instead of a class, and the "JpaRepository" that it extends.

The repository (III)

```
@Repository
@Transactional
public interface ShoutRepository extends JpaRepository<Shout, Integer> {

    @Query("select count(s) from Shout s")
    public long countAllShouts();

    @Query("select count(s) from Shout s where length(s.text) <= 25")
    public long countShortShouts();

    @Query("select count(s) from Shout s where length(s.text) > 25")
    public long countLongShouts();

}
```


The new stuff are the custom methods. For instance, “countAllShouts” is a custom method that counts all of the shouts in the database; note that the query that must be executed by the method is introduced by means of an “@Query” annotation. The language used to write the queries is JPQL, which is very similar to SQL when dealing with simple queries like the ones required to compute our indicators.

The repository (IV)

```
@Repository
@Transactional
public interface ShoutRepository extends JpaRepository<Shout, Integer> {

    @Query("select count(s) from Shout s")
    public long countAllShouts();

    @Query("select count(s) from Shout s where length(s.text) <= 25")
    public long countShortShouts();

    @Query("select count(s) from Shout s where length(s.text) > 25")
    public long countLongShouts();

}
```

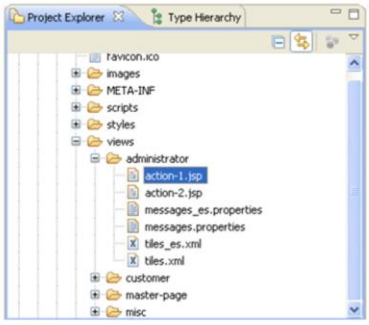

The second query counts the number of shouts whose text has less than or exactly 25 characters.

The repository (V)

```
@Repository
@Transactional
public interface ShoutRepository extends JpaRepository<Shout, Integer> {

    @Query("select count(s) from Shout s")
    public long countAllShouts();

    @Query("select count(s) from Shout s where length(s.text) <= 25")
    public long countShortShouts();

    @Query("select count(s) from Shout s where length(s.text) > 25")
    public long countLongShouts();

}
```


And the third query counts the number of shouts whose text has more than 25 characters.

The view (I)



It's a JSP specification of a user interface used to render some stats to the users

The view to render the statistics is a JSP specification of a table. Read on.

The view (II)

```
<p><spring:message code="administrator.action.1" /></p>

<table>
    <tr>
        <th><spring:message code="administrator.indicator" /></th>
        <th><spring:message code="administrator.value" /></th>
    </tr>
    <tr>
        <td><spring:message code="administrator.count.all.shouts" /></td>
        <td><jstl:out value="${statistics.get('count.all.shouts')}" /></td>
    </tr>
    <tr>
        <td><spring:message code="administrator.count.short.shouts" /></td>
        <td><jstl:out value="${statistics.get('count.short.shouts')}" /></td>
    </tr>
    <tr>
        <td><spring:message code="administrator.count.long.shouts" /></td>
        <td><jstl:out value="${statistics.get('count.long.shouts')}" /></td>
    </tr>
</table>
```


The table has a header row in which we use the “spring:message” tag to write the labels.

The view (III)

```
<p><spring:message code="administrator.action.1" /></p>

<table>
    <tr>
        <th><spring:message code="administrator.indicator" /></th>
        <th><spring:message code="administrator.value" /></th>
    </tr>
    <tr>
        <td><spring:message code="administrator.count.all.shouts" /></td>
        <td><jstl:out value="${statistics.get('count.all.shouts')}" /></td>
    </tr>
    <tr>
        <td><spring:message code="administrator.count.short.shouts" /></td>
        <td><jstl:out value="${statistics.get('count.short.shouts')}" /></td>
    </tr>
    <tr>
        <td><spring:message code="administrator.count.long.shouts" /></td>
        <td><jstl:out value="${statistics.get('count.long.shouts')}" /></td>
    </tr>
</table>
```


And it then has three rows to show the indicators. Note that the model includes variable “statistics”, which is a map that is injected by the controller. Like every other map, it has a method called “get”, which given a key returns its corresponding value. The first row then shows the counter of all shouts.

The view (IV)

```
<p><spring:message code="administrator.action.1" /></p>

<table>
    <tr>
        <th><spring:message code="administrator.indicator" /></th>
        <th><spring:message code="administrator.value" /></th>
    </tr>
    <tr>
        <td><spring:message code="administrator.count.all.shouts" /></td>
        <td><jstl:out value="${statistics.get('count.all.shouts')}" /></td>
    </tr>
    <tr>
        <td><spring:message code="administrator.count.short.shouts" /></td>
        <td><jstl:out value="${statistics.get('count.short.shouts')}" /></td>
    </tr>
    <tr>
        <td><spring:message code="administrator.count.long.shouts" /></td>
        <td><jstl:out value="${statistics.get('count.long.shouts')}" /></td>
    </tr>
</table>
```

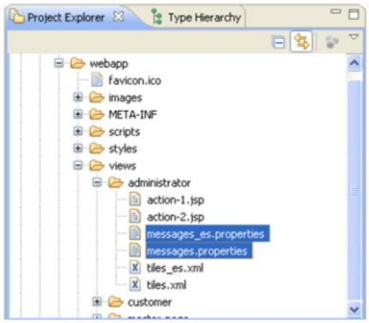

The second row shows the counter of short shouts.

The view (V)

```
<p><spring:message code="administrator.action.1" /></p>

<table>
    <tr>
        <th><spring:message code="administrator.indicator" /></th>
        <th><spring:message code="administrator.value" /></th>
    </tr>
    <tr>
        <td><spring:message code="administrator.count.all.shouts" /></td>
        <td><jstl:out value="${statistics.get('count.all.shouts')}" /></td>
    </tr>
    <tr>
        <td><spring:message code="administrator.count.short.shouts" /></td>
        <td><jstl:out value="${statistics.get('count.short.shouts')}" /></td>
    </tr>
    <tr>
        <td><spring:message code="administrator.count.long.shouts" /></td>
        <td><jstl:out value="${statistics.get('count.long.shouts')}" /></td>
    </tr>
</table>
```


And the third row shows the counter of long shouts.

The i18n&l10n bundles (I)



They're files that help associate user-defined keys with messages written in Spanish or English

There's little new regarding the i18n&l10n bundles.

The i18n&l10n bundles (II)

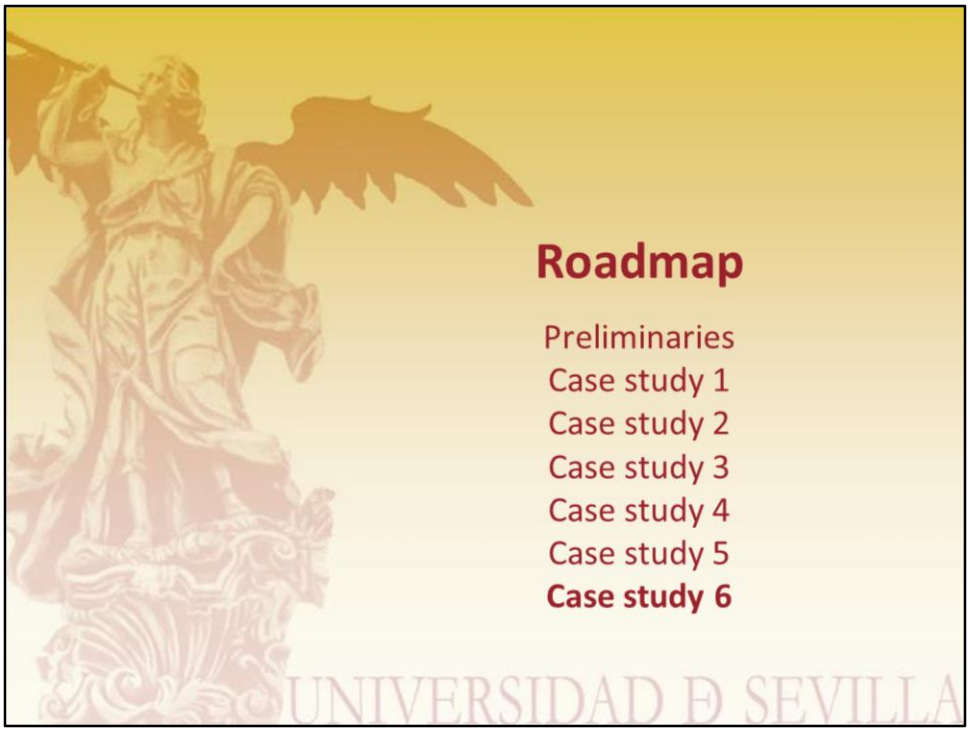
```
administrator.action.1 = Esta es la acción 1 del administrador

administrator.report = Indicators
administrator.indicator = Indicador
administrator.value = Valor
administrator.count.all.shouts = Todos los gritos
administrator.count.short.shouts = Gritos cortos
administrator.count.long.shouts = Gritos largos
```


This is the Spanish bundle.

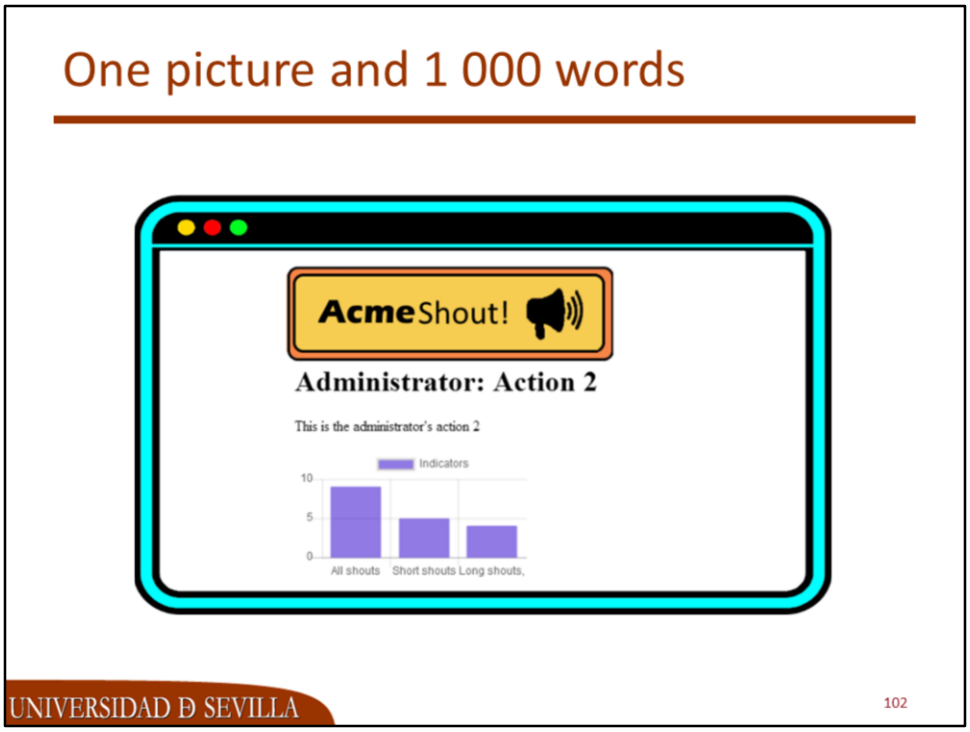
The i18n&l10n bundles (III)

```
administrator.action.1 = This is the administrator's action 1

administrator.report = Indicators
administrator.indicator = Indicator
administrator.value = Value
administrator.count.all.shouts = All shouts
administrator.count.short.shouts = Short shouts
administrator.count.long.shouts = Long shouts
```


And this is the English bundle.



Case study 5 was very interesting, wasn't it? Let's go on to the sixth case study… which is… different!



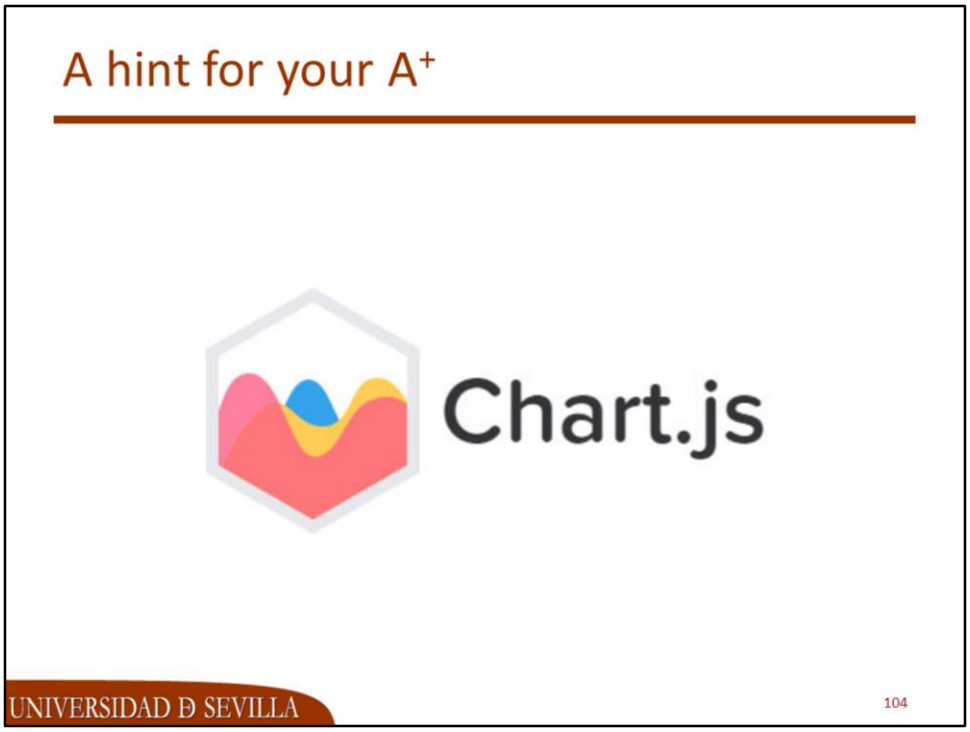
The goal is to extend our system so that it shows the statistics in a simple vertical bar chart.

This is an A+

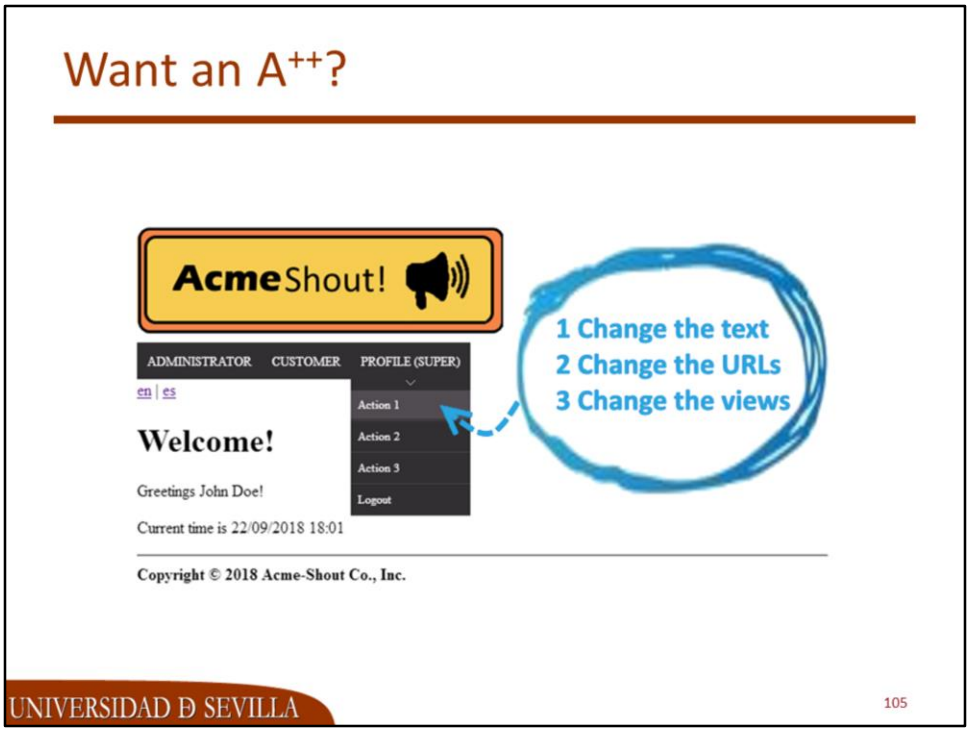
UNIVERSIDAD Đ SEVILLA

103

The difference is that this case study is an A+. That means that it requires you to do a little research without your lecturer's supervision. They'll just provide you with a hint, and you must produce a solution.



The hint for this A+ that you should not miss the opportunity to take a look at a component called Chart.js. Search the Web for information about this component, fetch some tutorials, learn how to use it with a hello-world project, put it to practice, and earn your A+.

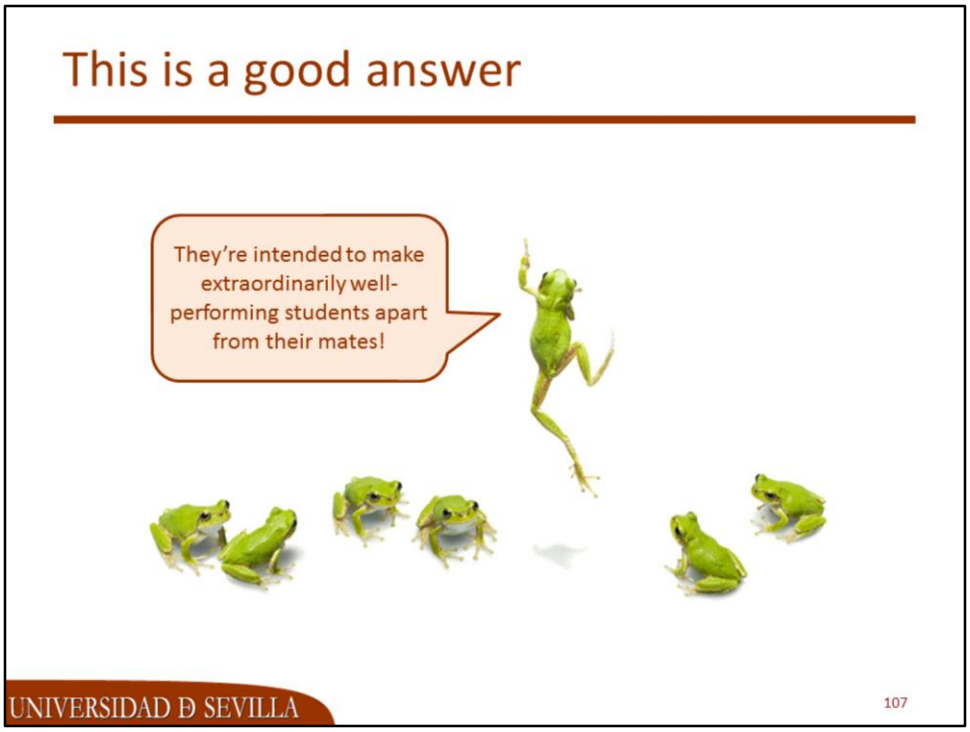


By the way: what about an A++. A+s are great, so A++s must be da bomb! Your A++ in this lesson consists of:

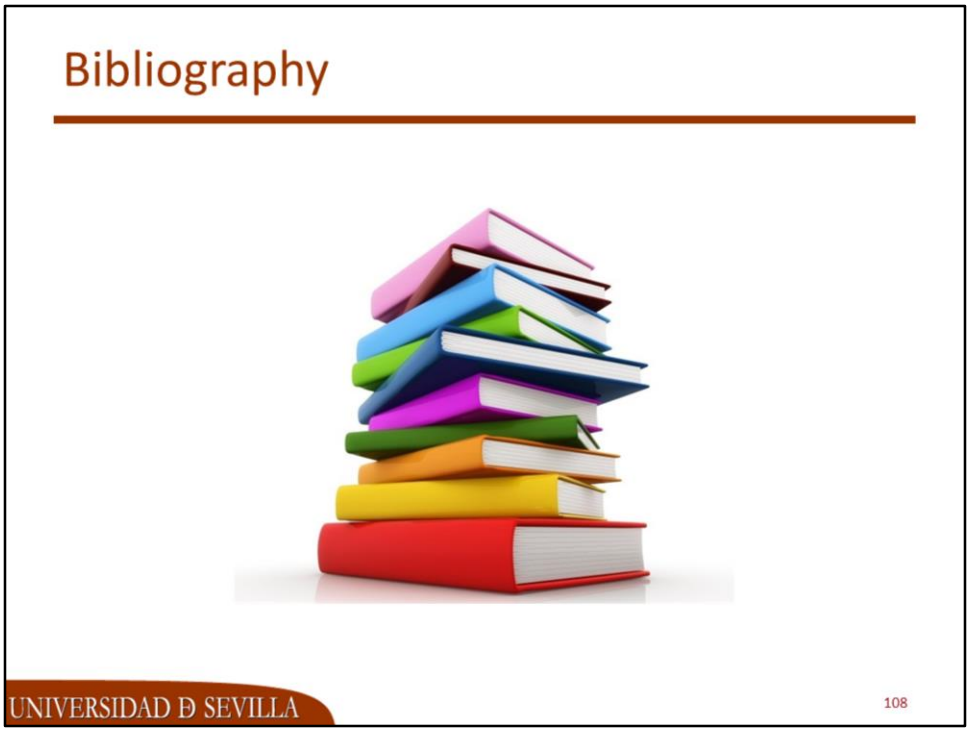
1. Changing the text in the menus into more intuitive messages. For instance "Profile/Action 1" might change as "Profile/Inspiring quotes" and "Customer/Action 2" might change as "Customer/Enter a shout".
2. Changing the corresponding URLs so that they are more intuitive. For instance "/profile/action-3.do" might become "/profile/quotes.do" and "/administrator/action-2.do" might become "/administrator/chart.do".
3. Changing the names of the views so that they are easier to understand. For instance, view "profile/action-3" might be renamed as "profile/oops" and view "administrator/action-1" might be renamed as "administrator/dashboard".

It's not difficult to implement these requirements, but you'll be alone in the wild. Don't expect any hints by your lecturers. Generally speaking: don't event expect your lecturers to propose A++s. If you're interested in an A++, please contact your lecturer.

This is a good question. There's no record of A^{++}s in the syllabus. That's true. So, what are they exactly intended for?



Simply put. They're intended to make extraordinarily well-performing students apart from their mates. Work on as many A^{++}s as you can and learn as much as you can from them.



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.