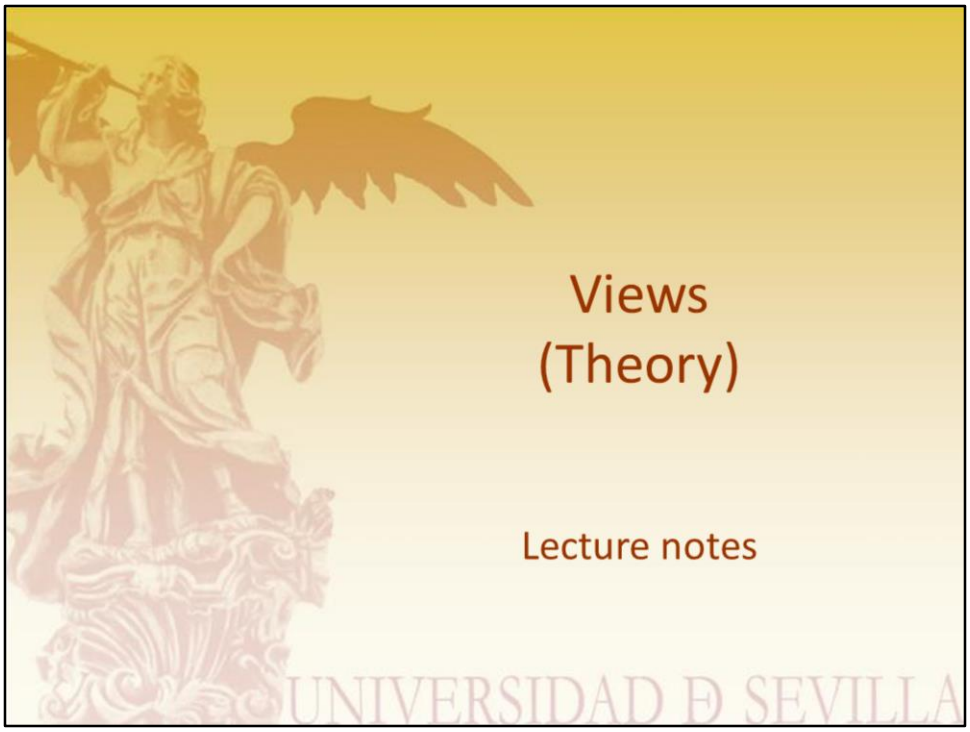


Welcome to this lecture! Our goal's to report on the theory that you need to create views.

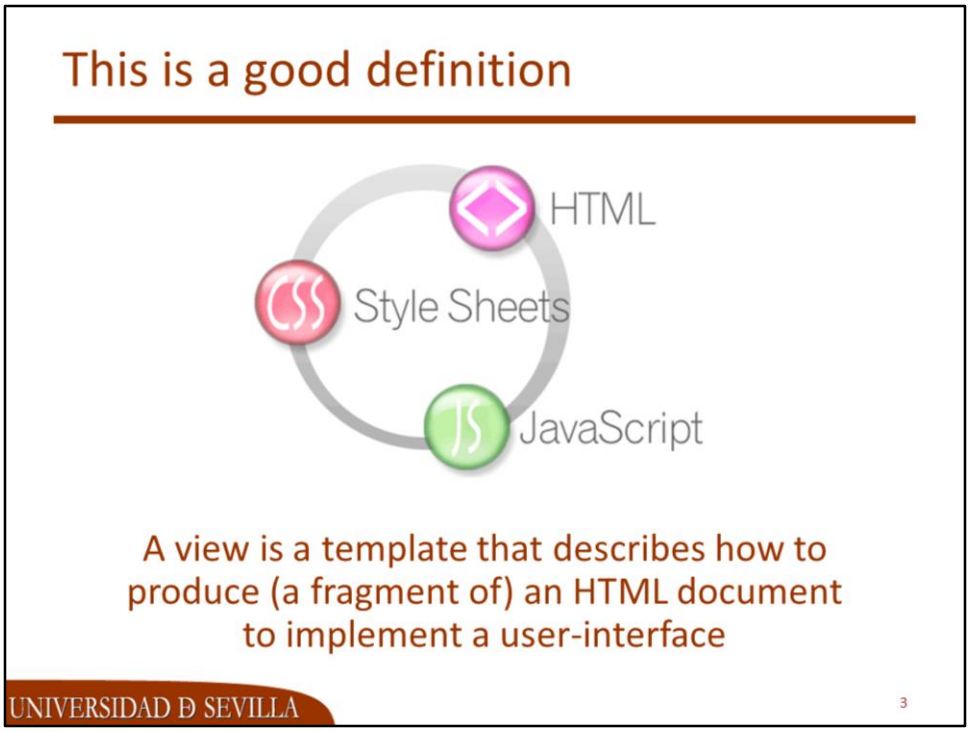
--

What are views?

UNIVERSIDAD Ð SEVILLA

2

As usual, we'll start with a definition: what are views? You should have an intuitive understanding since we introduced this concept in Lesson "L01 – Introduction".



Here, we provide the definition that we're going to use in this subject: it's a template that describes how to produce an HTML document (or a fragment of an HTML document) so that we can implement a user-interface. You know that HTML is the universal language on which web pages rely, and that HTML documents are commonly accompanied by CSS style sheets (which provide hints on how to render the elements of an HTML document regarding positions, colours, line styles, and so on) and JavaScript scripts (which endow HTML documents with algorithmic capabilities).

How do you think views are devised? As usual, please, try to produce your own answer before taking a look at the following slides.

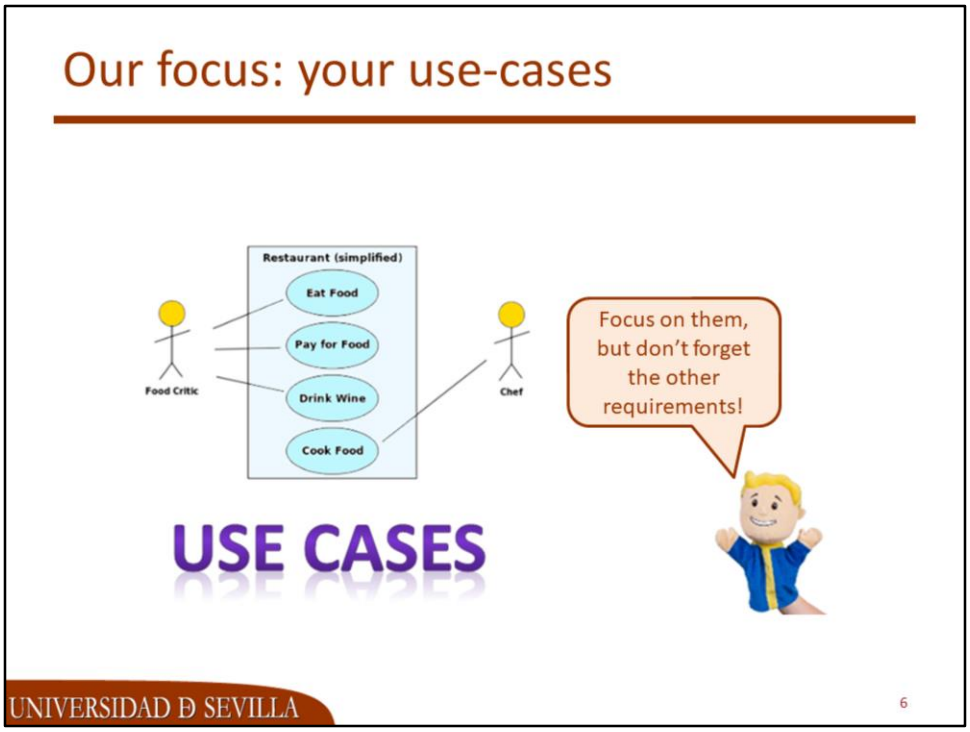
Starting point: your requirements

- Ilities
  - The non-functional aspects of a system
- Information
  - The data a system has to process
- Use cases
  - The tasks the actors can perform on the system

UNIVERSIDAD Ð SEVILLA

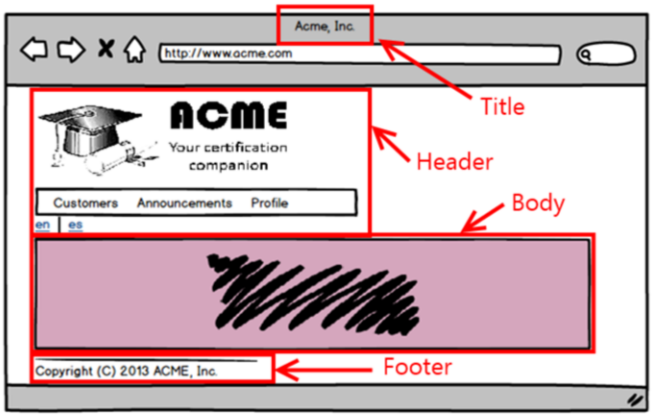
5

We don’t think this should be surprising at all now: the starting point to devise the views of your project are your requirements.



In particular, we'll focus on your use cases. But, please, remember that having a focus doesn't mean forgetting about the other sections. Don't forget about the illities and the information requirements since you need to have a holistic perspective of your requirements if you wish to produce good views.

Step 1: design a master page



UNIVERSIDAD D SEVILLA

7

The first step is to create a master page using a technology called Apache Tiles. The master page's a template that provides a common look and feel to the pages in your system.

Step 2: specify listing & edition views

The second step is to specify your views, which entails designing mock-ups, models, and link specifications.

Step 3: implement the views

```
<html>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>

<f:view>
    <body>
    <h:form>
        <h3>Please enter your name and password.</h3>
        <table>
            <tr>
                <td>Name:</td>
                <td><h:inputText value="#{user.name}" /></td>
            </tr>
            <tr>
                <td>Password:</td>
                <td><h:inputSecret value="#{user.password}" /></td>
            </tr>
        </table>
        <p><h:commandButton value="Login" action="login" /></p>
    </h:form>
    </body>
</f:view>
</html>
```


The third step is to implement your views using a technology called JSP.

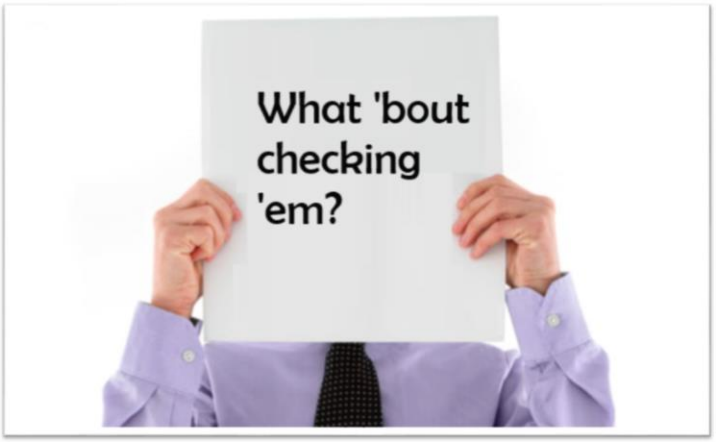
Step 4: configure your project

UNIVERSIDAD Ð SEVILLA

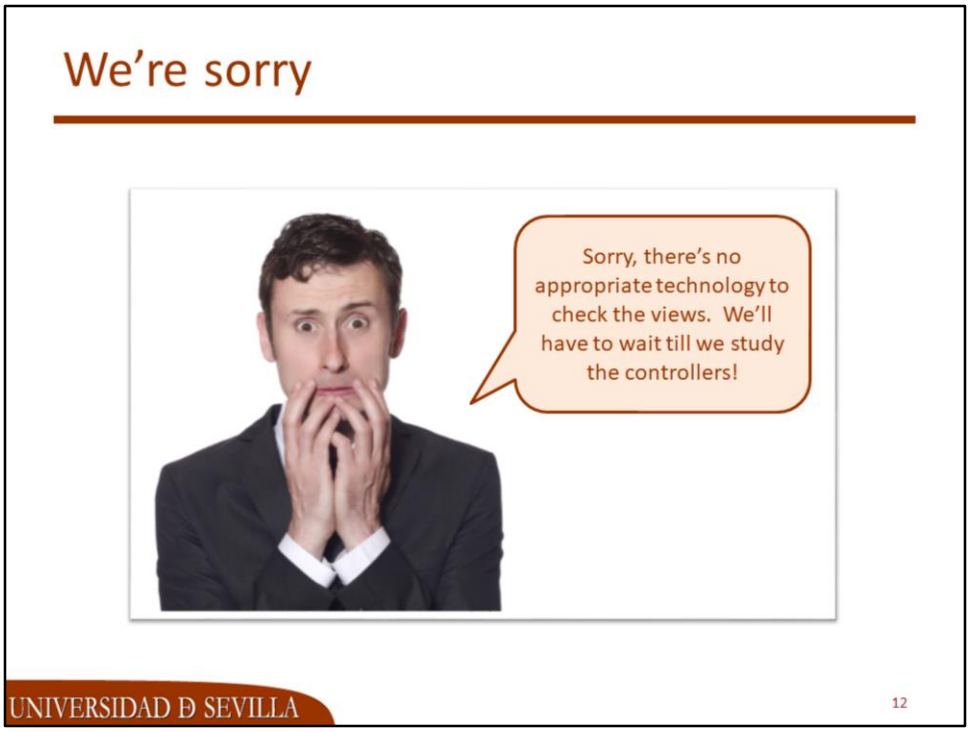
10

And the final step is to configure your project so that it can handle your views.

That's a good question!



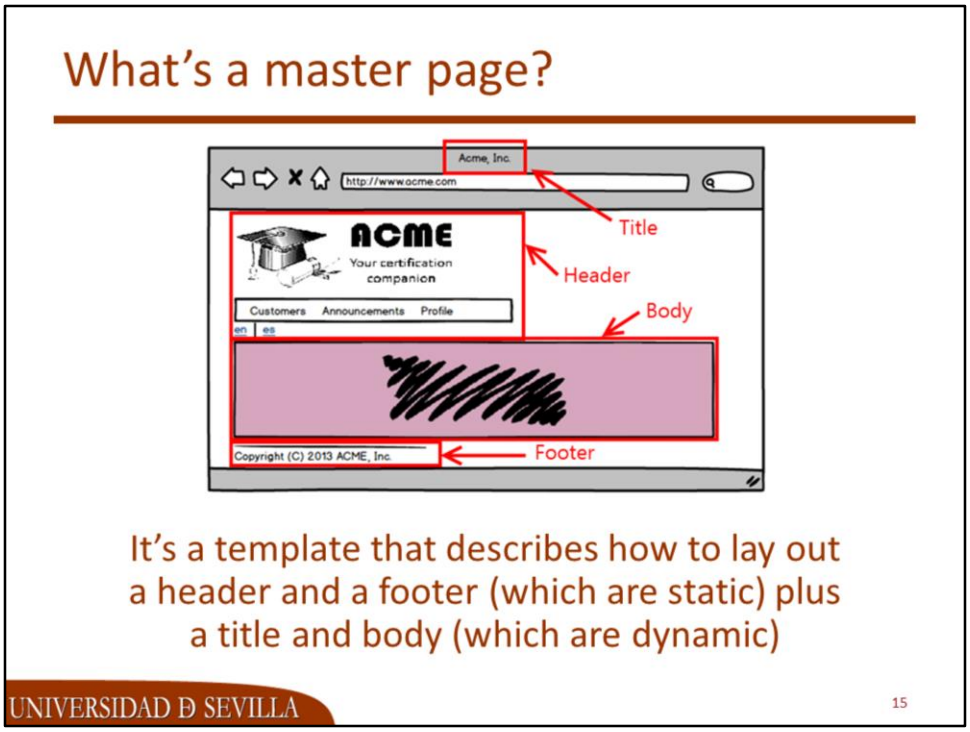
Unfortunately, there's not a fourth step regarding checking the views.



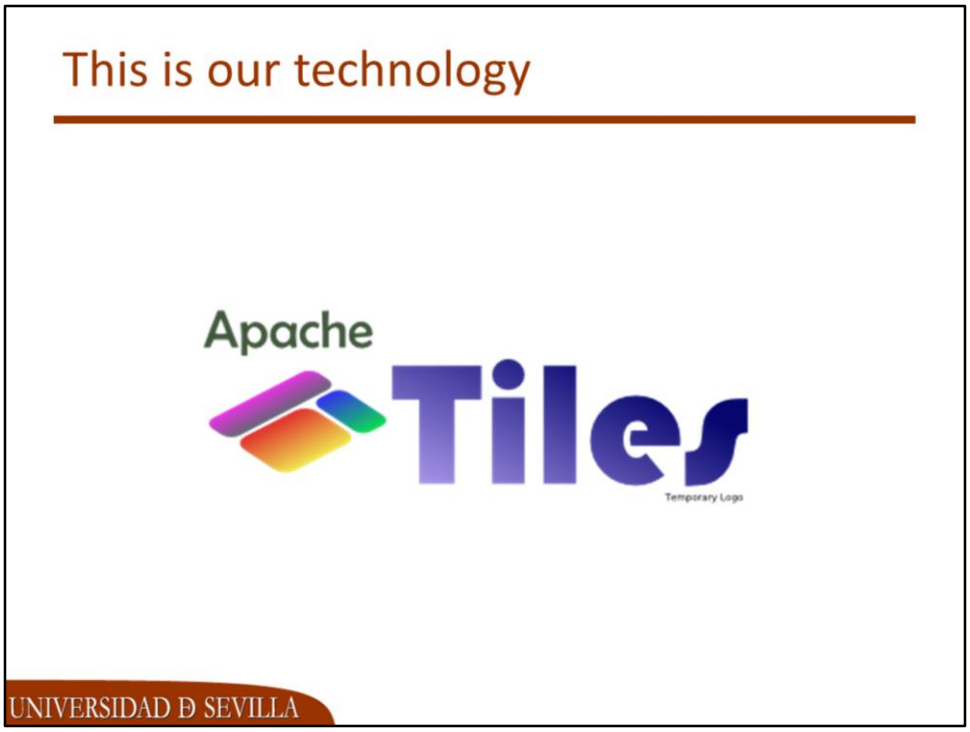
We're sorry, but we haven't found any good technologies to check your views. That means that we'll have to work on our views, but we won't be able to check them until the next lesson, when we start working on the controllers.

This is then our roadmap for today's lecture: we'll start with some details regarding master pages, next we'll delve into specifying views, then into implementing them, and finally into how to configure your project.

Let's start with the master pages.

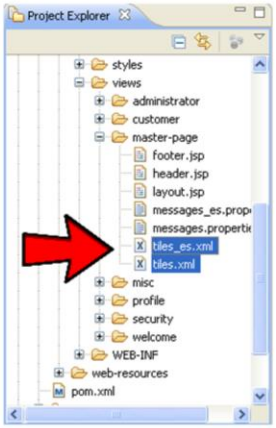


A master page's a template that describes how to lay out a header and a footer (whose contents are the same in every page) plus a title and a body (whose contents vary from page to page). The header typically contains a logo and a menu bar; the footer typically contains a copyright message; the title provides a short description to your pages; and the body is the work area of the page, which is basically used to display listings and edition forms. Note that technically speaking, all of the components of the master page are views, but more often than not we'll use term "view" to refer to a page that extends the master page with a title and body or to the body itself.



Apache tiles is the technology that we're going to use in order to implement our master pages.

The master page definition file (I)



The definition file of your master page is located in folder “webapp/views/master-page”. There are two independent files: “tiles.xml”, which defines the master page in English, and “tiles_es.xml”, which defines it in Spanish. In our project template, both files are identical since the master page is the same in both languages. The suffix is defined by a standard called ISO 639; you may find additional information on this standard at http://en.wikipedia.org/wiki/ISO_639, for instance.

The master page definition file (I)

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE tiles-definitions PUBLIC
       "-//Apache Software Foundation//DTD Tiles Configuration 2.0//EN"
       "http://tiles.apache.org/dtds/tiles-config_2_0.dtd">

<tiles-definitions>
  <definition name="master.page" template="/views/master-page/layout.jsp">
      <put-attribute name="title" value="" />
      <put-attribute name="header" value="/views/master-page/header.jsp" />
      <put-attribute name="body" value="" />
      <put-attribute name="footer" value="/views/master-page/footer.jsp" />
  </definition>
</tiles-definitions>
```


This slide shows the contents of the “tiles.xml” file. The definition of the master page is provided in a “definition” element that is contained within a “tiles-definitions” element. Note that this element has two attributes: “name”, which we’ll use everywhere to reference this master page, and “template”, which indicates the path to a file in which we provide a description of the master page. Then come a number of attributes that are used within the previous file, namely: “title”, “header”, “body”, and “footer”. Note that attributes “header” and “footer” are set by default to a couple of files that provide the contents of these attributes (these files are JSP documents); “title” and “body” are however blank, since we can’t provide a title and a body until we instantiate the master page with a specific view.

Extending the master page (English)

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE tiles-definitions PUBLIC
       "-//Apache Software Foundation//DTD Tiles Configuration 2.0//EN"
       "http://tiles.apache.org/dtds/tiles-config_2_0.dtd">

<tiles-definitions>

  <definition name="announcement/list" extends="master.page">
    <put-attribute name="title" value="Announcement list" />
    <put-attribute name="body" value="/views/announcement/list.jsp" />
  </definition>

  <definition name="announcement/edit" extends="master.page">
    <put-attribute name="title" value="Edit announcement" />
    <put-attribute name="body" value="/views/announcement/edit.jsp" />
  </definition>

</tiles-definitions>
```


You can easily extend the master page to create your own definitions. In this example, we define two views in a “titles.xml” file, that is, two web pages that rely on our master page and are assumed to be internationalised and localised to English, namely:

- “announcement/list”: it extends the previous master page and sets the “title” and the “body” attributes to appropriate values. Note that Apache Tiles uses attribute “value” to provide the value of an attribute, independently from whether it is a piece of text, like in the case of the “title” attribute, or a reference to a JSP document. This is very confusing at first, because there’s a single attribute with two different meanings. But… hey, that’s technology! Whatcha expect?
- “announcement/edit”: it also extends the previous master page and sets the values of attributes “title” and “body” to the appropriate values.

Extending the master page (Spanish)

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE tiles-definitions PUBLIC
       "-//Apache Software Foundation//DTD Tiles Configuration 2.0//EN"
       "http://tiles.apache.org/dtds/tiles-config_2_0.dtd">

<tiles-definitions>

  <definition name="announcement/list" extends="master.page">
    <put-attribute name="title" value="Announcement list" />
    <put-attribute name="body" value="/views/announcement/list.jsp" />
  </definition>

  <definition name="announcement/edit" extends="master.page">
    <put-attribute name="title" value="Edit announcement" />
    <put-attribute name="body" value="/views/announcement/edit.jsp" />
  </definition>

</tiles-definitions>
```


And this slide shows the corresponding definitions in Spanish. Recall that these definitions are expected to be provided in a file called “titles_es.xml”.

Let's now work on how to specify the views.

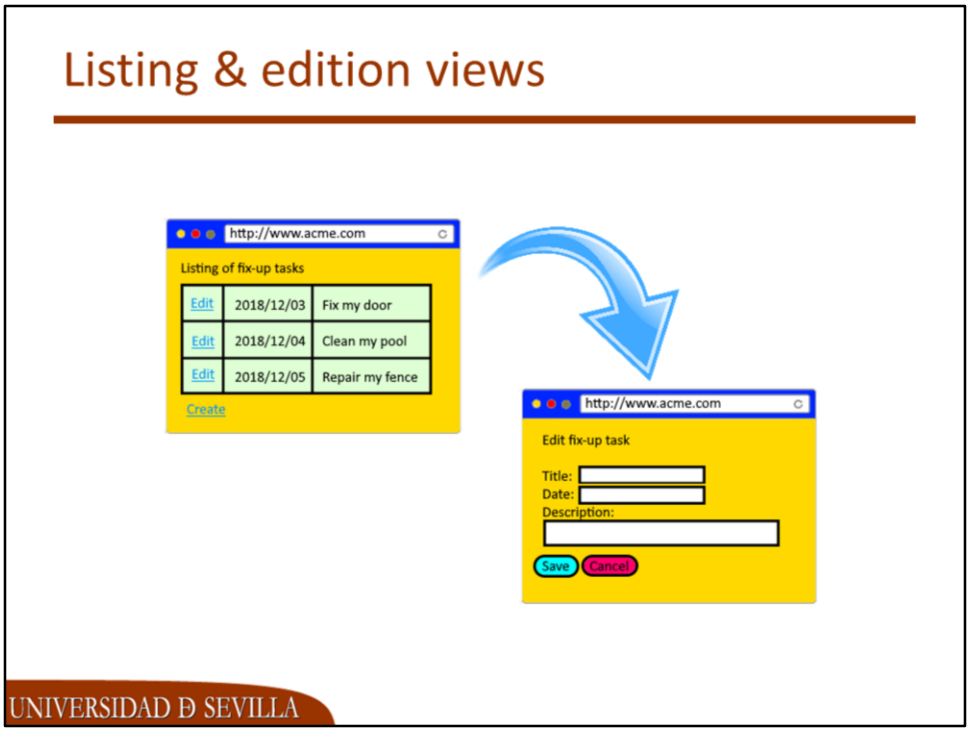
What's a view specification

A view specification consist of a mock-up, a model, and a link specification

UNIVERSIDAD Ð SEVILLA

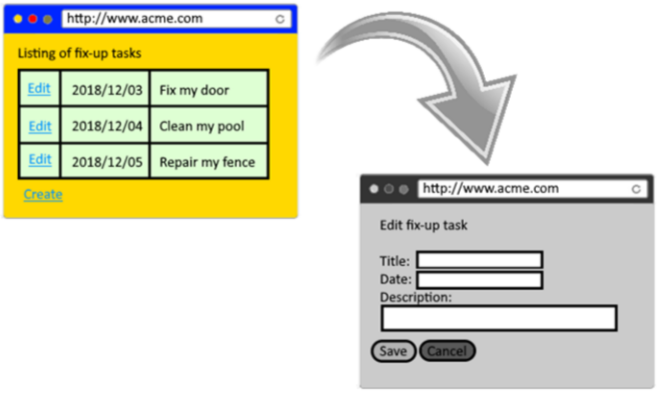
22

To fully specify a view, so that a programmer can start implementing it, you need to have a mock-up, a model, and a link specification. Very likely, you'll get the mock-ups from your requirements engineer, but it's you who must work on the models and the link specification. Please, keep reading to learn more about this stuff.



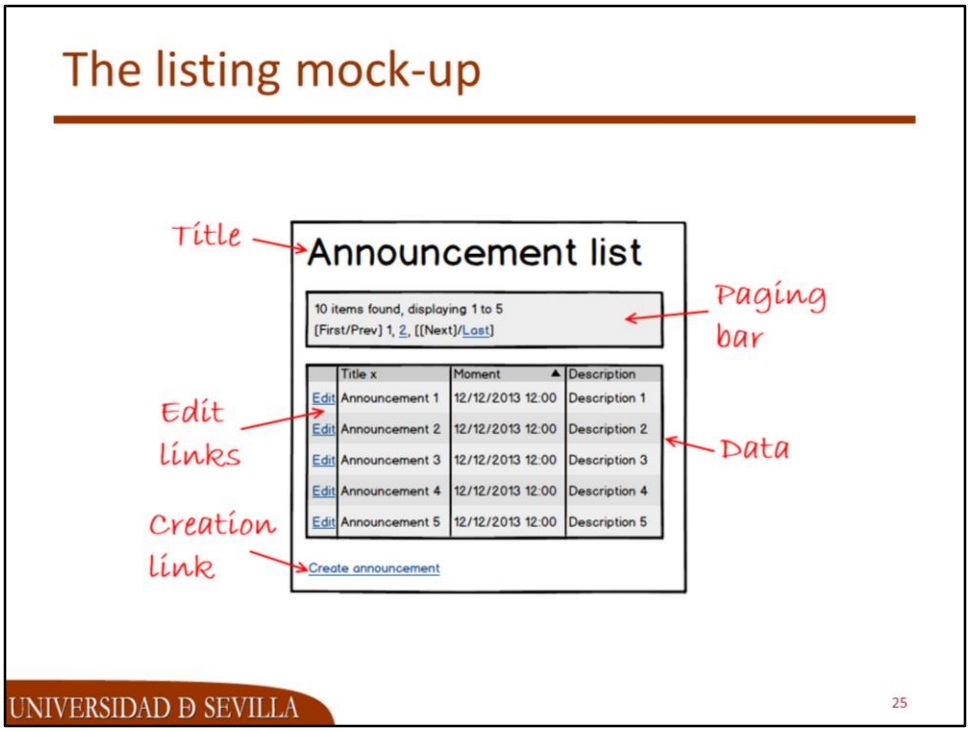
In a typical web information system, the majority of views are listing and edition views. The former present listings of domain entities and the latter allow to create new objects, to update existing ones, or to delete them. Very commonly, users will first have access to a listing view and then to an edition view.

Listing & edition views



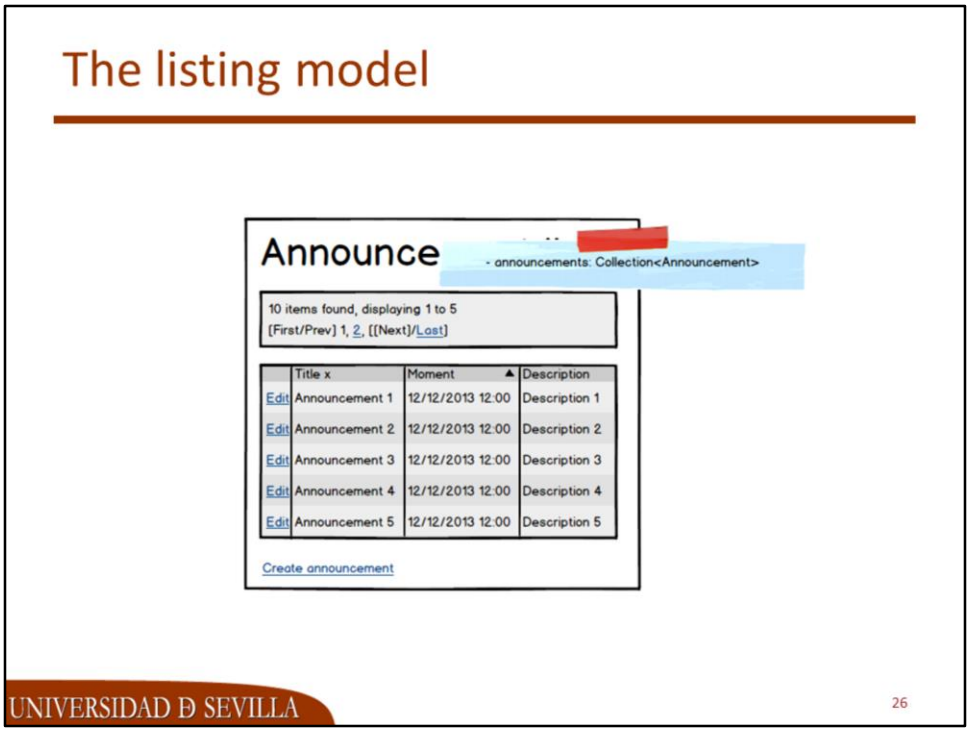
UNIVERSIDAD D SEVILLA

Let's start with the listing views.

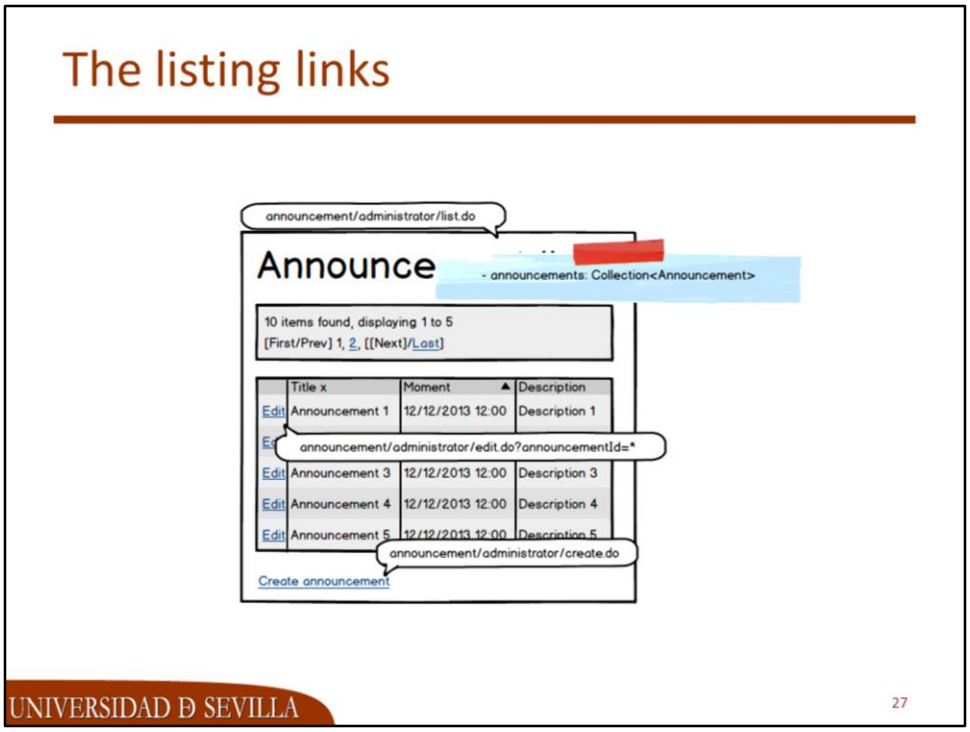


This slide shows the design of the listing mock-up that we're going to use. Its components are the following:

- A title, which provides a description of the entities being listed.
- A paging bar, that allows to scroll through the listing.
- The data, which is presented in tabular format.
- The edit links, which allow to edit individual entities.
- The creation link, which allows to create a new entity.

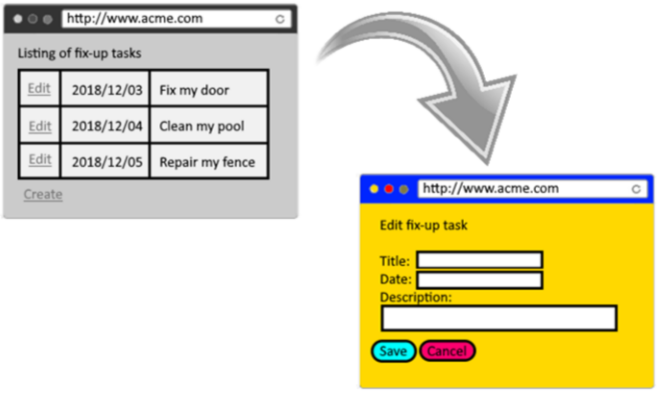


You may wonder where the data that is shown comes from. It's easy, it comes from a model. Please, recall that a controller must produce a model and a view name: the view name is the name of the view to be rendered, and the model's a map from variables onto objects that provides the data that must be injected in the view. In the example in this slide, for instance, we've used a cyan note to represent the model that feeds this view; in this case, it consists of a variable called "announcements" that provides the collection of announcements to be displayed.



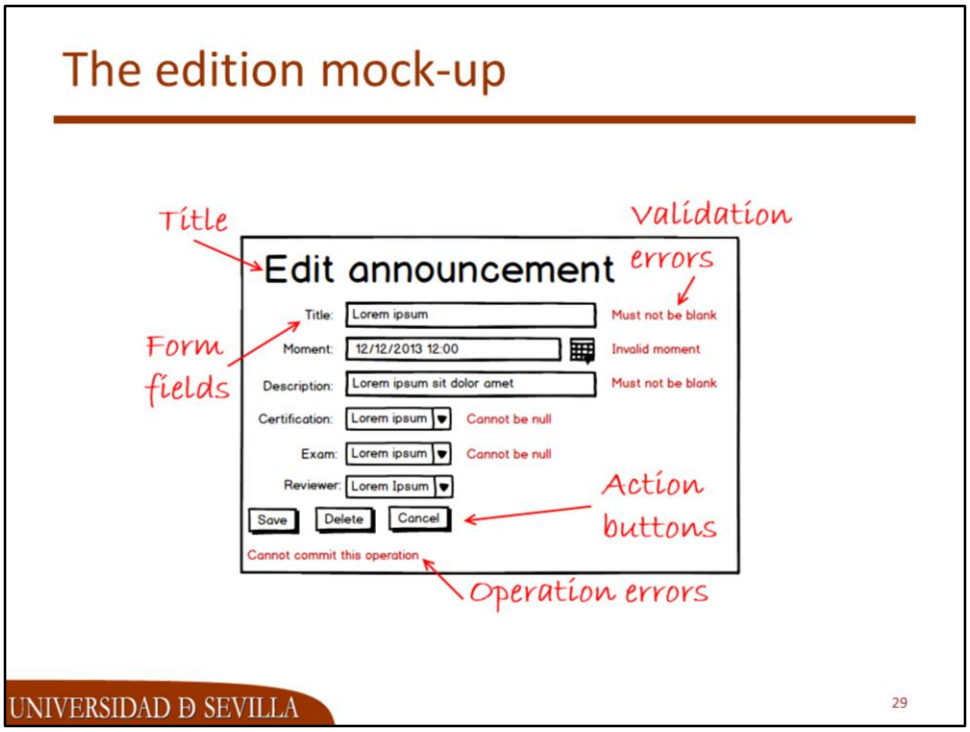
Finally, we need to specify the URLs of the links in the mock-ups. This slide shows the links in the listing mock-up. Please, note that they all have the same structure: first comes the entity name, then the authority, then an action name with a ".do" suffix, and, optionally, some parameters. The values of the parameters are indicated with an "*", which is a placeholder that must be replaced by the corresponding identifier. This is enough for the majority of cases; in cases in which there are other more complex parameters, you must add notes to your link specification.

Listing & edition views



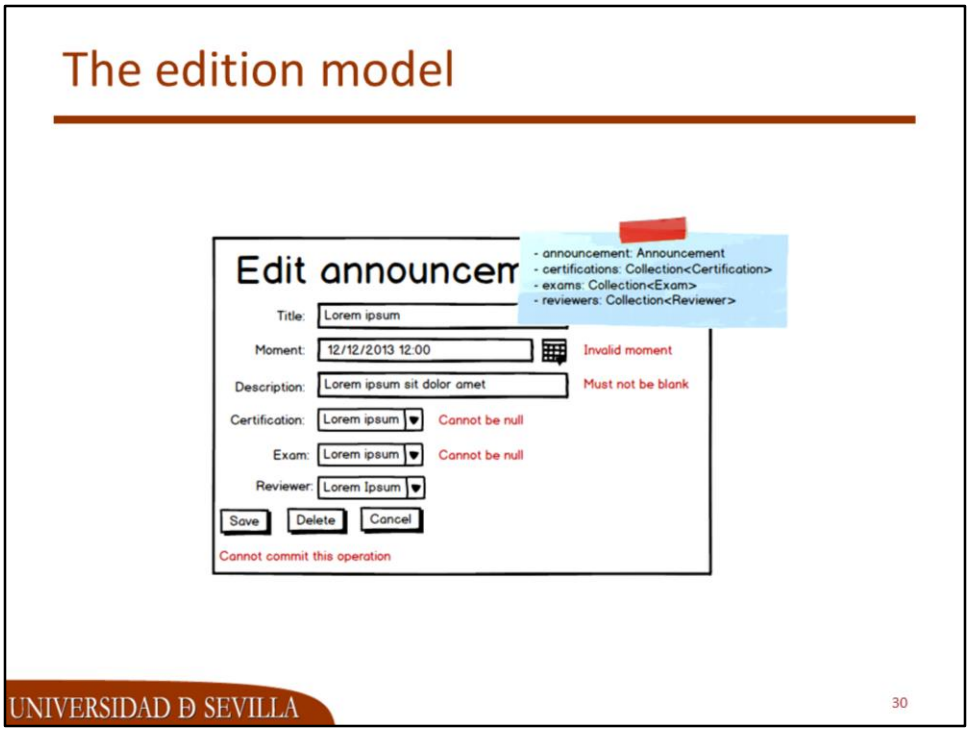
UNIVERSIDAD D SEVILLA

Let's now explore the edition forms.



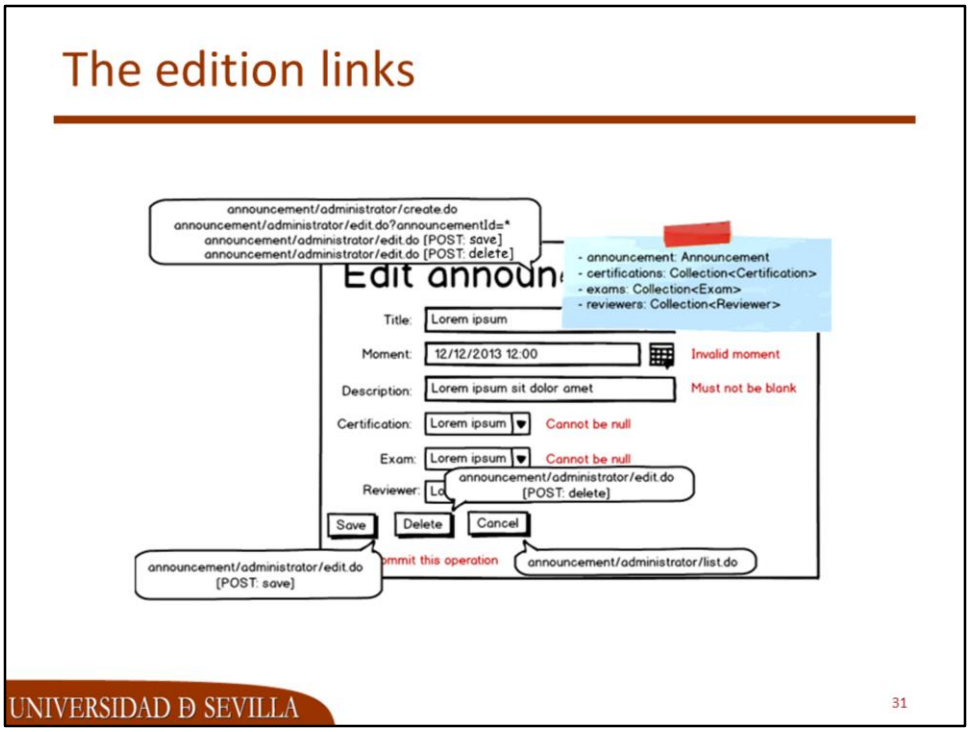
This slide shows the edition mock-up we've designed to edit an announcement. Its components are the following:

- A title, which provides a description of the entity being edited.
- Form fields, which have a label that describes the field and an input box.
- Validation errors, which are shown only when the data entered is invalid according to the constraints in the domain model.
- Action buttons, which allow the user to save the entity being edited, to delete it, or to get back to the listing.
- Operation errors, which are messages that indicate that an error happened when the user pressed the "Save" or the "Delete" button. The master page that we provide reads the message to be shown in the operation errors from a model variable called "message".



As usual, we'll describe the model with a sticky cyan note. In this case, the model consists of the following variables:

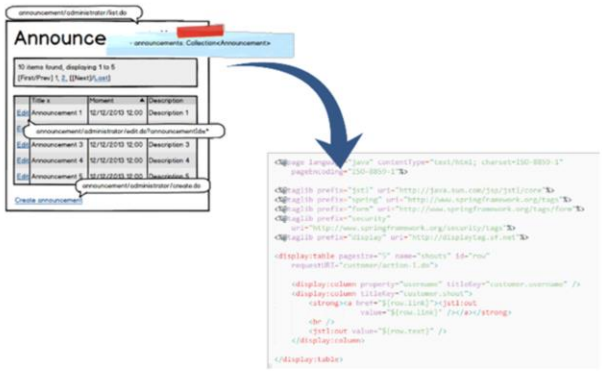
- "announcement": it references the object to be edited.
- "certifications": it provides the whole list of certifications, which will be used to feed the dropdown list in which the user selects a certification for an announcement.
- "exams": it provides the whole list of exams that are available for the selected certification.
- "reviewers": it provides the whole list of reviewers that may be selected to review an announcement.



Finally, we need to describe the links in the mock-up. The edition mock-up is a little more involved than the listing mock-up since it can be requested by means of GET and POST requests. In the former case, the user has clicked on a link and is requesting to edit a given announcement, so the URL includes an “announcementId” parameter. In the second case, the user has clicked on the “Save” or the “Delete” buttons. Note that in the cases in which a link results in a POST request, we indicate it in squared brackets, including the name of the corresponding button.

Let's now delve into how to implement views.

What's view implementation about?

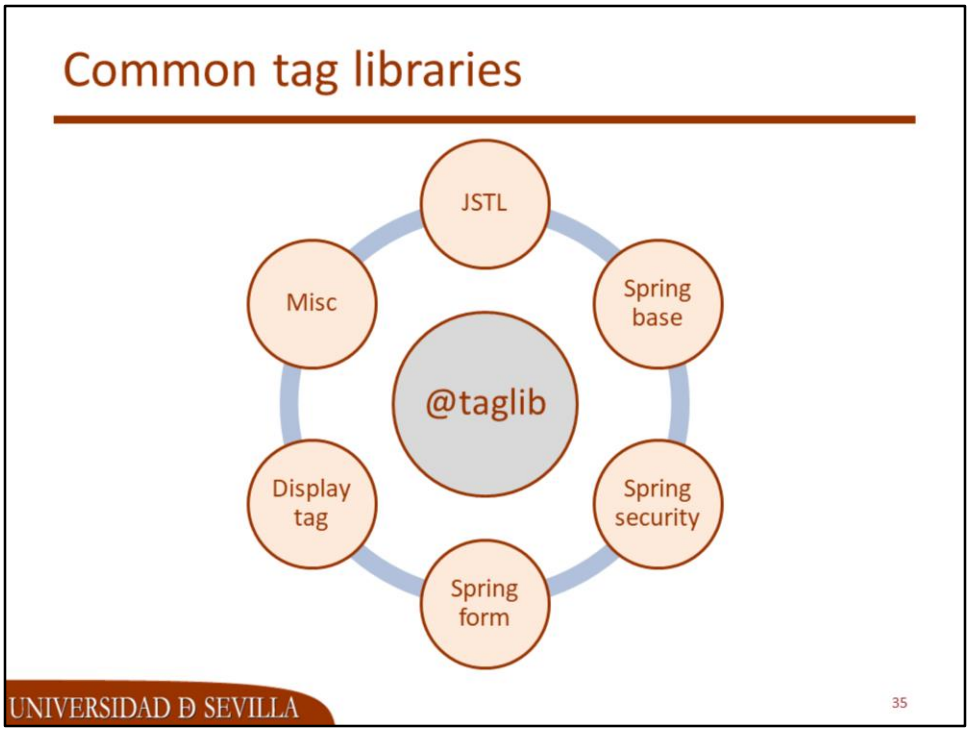


It's about implementing a view specification using the JSP technology

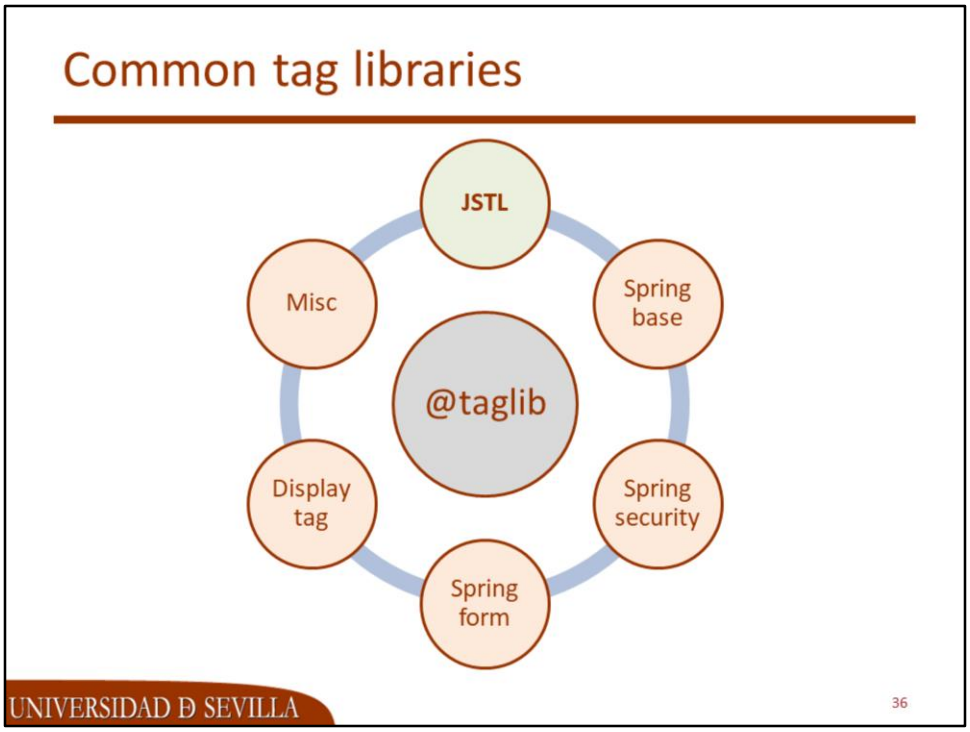
View implementation is about implementing a view specification using the JSP technology.

A sample, please

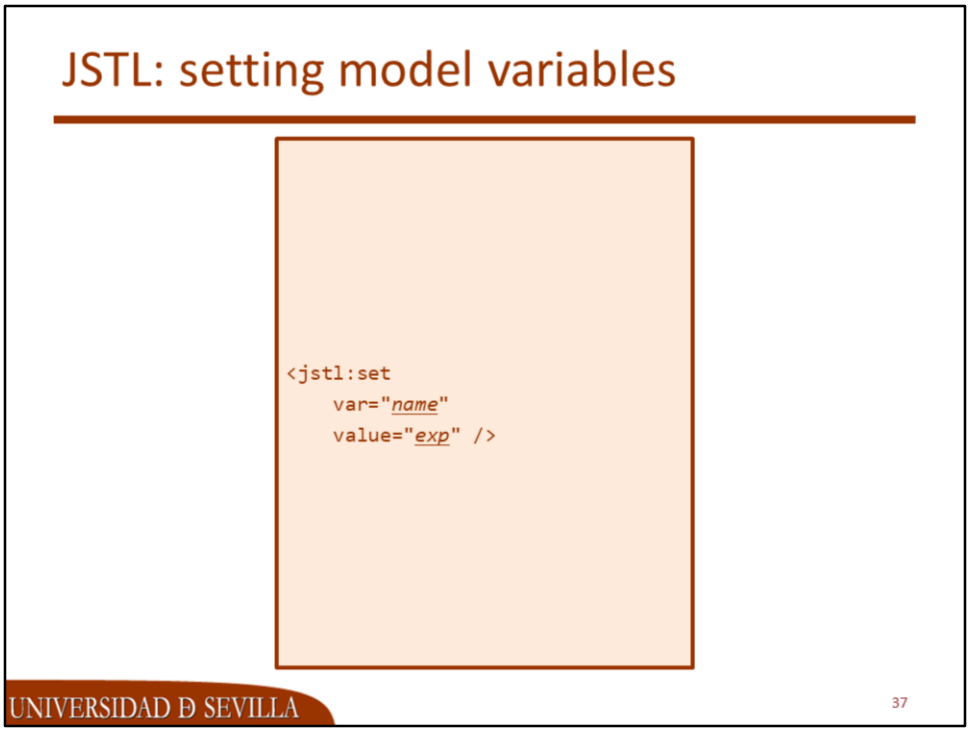
This slide provides a sample JSP view. It has a header with a “page” element and a series of “taglib” elements. The “page” element provides information about the language used in the project, which is Java in our case, the type of contents that the view generates, which is HTML text with the ISO-8859-1 charset, and the page encoding, which is also ISO-8859-1. The “taglib” elements allow to import libraries that provide tags that simplify implementing the mock-ups. The body of the view basically consists of tags and HTML code.



Obviously, the core of JSP are the tab libraries. In this slide, we show the most important ones.



Let's start with the JSTL tag library. JSTL stands for "Java Standard Template Library", which is the reason why some developers refer to this tag library as the core tag library. The JSTL tag library provides processing instructions by means of which you can implement very simple algorithms.

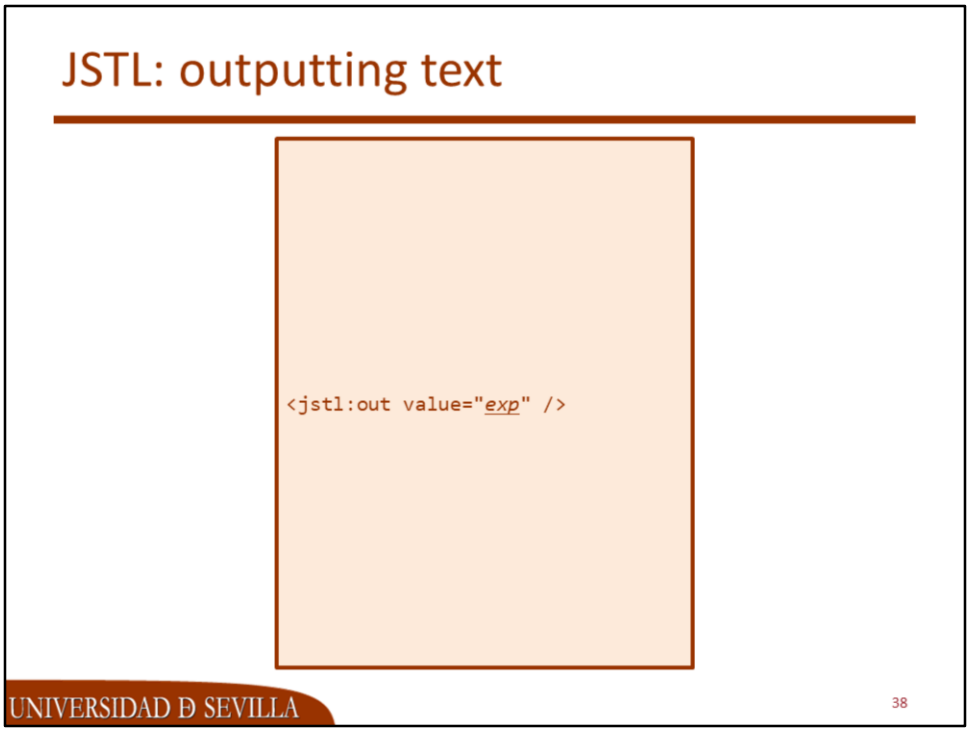


The first tag is “set”, which sets model variable “name” to the result of evaluating “exp”. For instance,

<jstl:set var=“secret” value=“${1000 + 200 + 30 + 4}” />

sets a variable called “secret” to value “1234”.

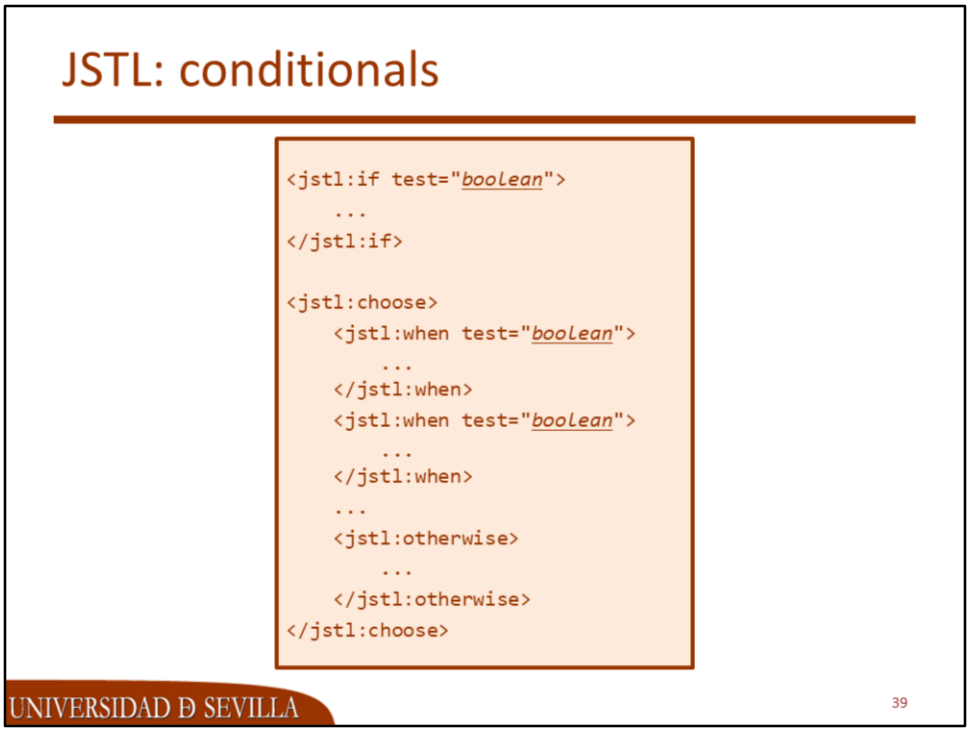
NOTE: in the sequel, we use “name” to refer to a variable name like “i” or “counter” and “exp” to refer to an expression like “${10}”, “${counter}”, “${message != null}”, or “${message.length()}”. If an expression is a literal, then it can be written literally; for instance “${10}” and “10” are the same and “ ${‘hello’} “ is the same as “ ‘hello’ ”.



The next tag's "out", which outputs the result of evaluating "exp". Note that instead of this tag, you can simply write "exp", but the result's not the same: "out" cares of translating characters like "<" or "&" into their HTML equivalent HTML entities, e.g., "<" and "&". Therefore, you should always use "out" unless you're absolutely sure that your expression doesn't result in any special characters. For instance,

<jstl:out value="${message}" />

outputs string ""My <<first>&gt message"" assuming that there's a variable in the model called "message" whose value is "\"My <<first>> message\"".



The next tags help implement conditionals, namely:

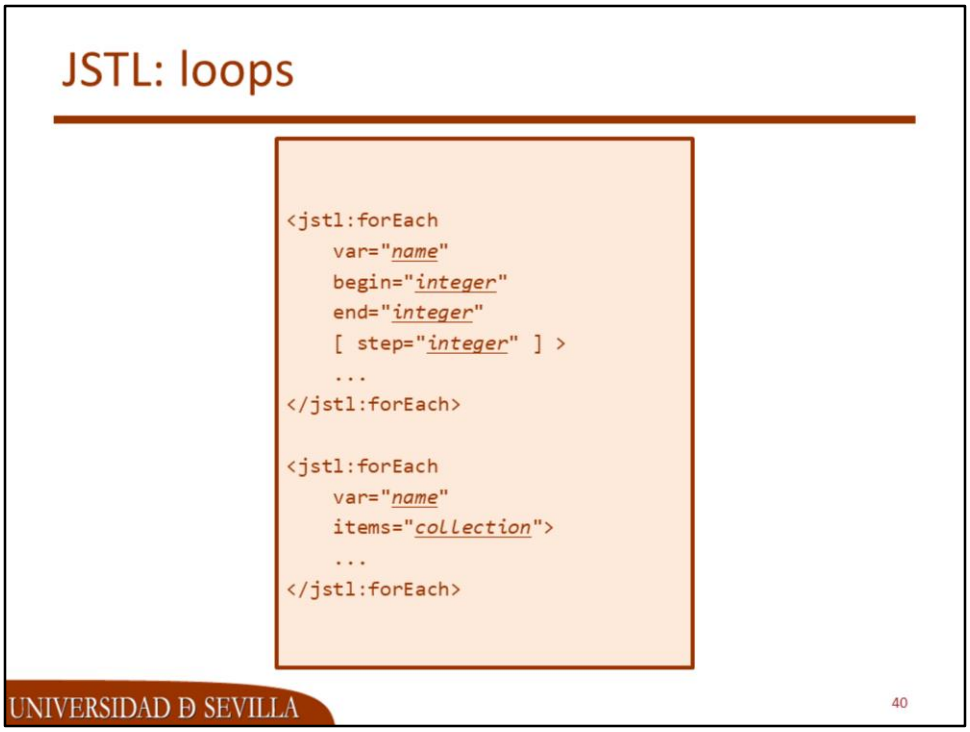
- "if": if "boolean" evaluates to "true", then this tag executes its body; otherwise it's ignored. There's no if-then-else tag.
- "choose": this is a multi-branch if statement; it interprets the body of the first "when" branch whose "boolean" evaluates to "true"; if more than one branch evaluates to "true", only the first one is interpreted; if none evaluates to "true", then the "otherwise" branch is interpreted, if any.

For instance,

```
<jstl:choose>
  <jstl:when test="${x >= 0 && x <= 1000}">
     This is a small number
  </jstl:when>
  <jstl:when test="${x >= 1000}">
     This is a large number
  </jstl:when>
  <jstl:otherwise>
     This is a negative number
  </jstl:otherwise>
</jstl:choose>
```

will output "This is a small number" if there's a model variable whose value is a number in range zero up to one thousand.

NOTE: in the sequel, we use "boolean" to refer to an expression that yields "true" or "false". For instance, "${message.length() > 25}", or "${!p || q}". If an expression is a literal, then it can be written literally; for instance "${true}" and "true" are the same and "${false}" is the same as "false".



The JSTL library also provides a "forEach" tag that helps implement loops. It has two variants:

- In its simplest form, the variable iterates from the integer value indicated by attribute "begin" until the integer value indicated by attribute "end", according to the increment indicated by attribute "step"; if no step is indicated, then "1" is the default.
- In its more advanced form, it iterates over the elements of a collection that is indicated by means of attribute "items".

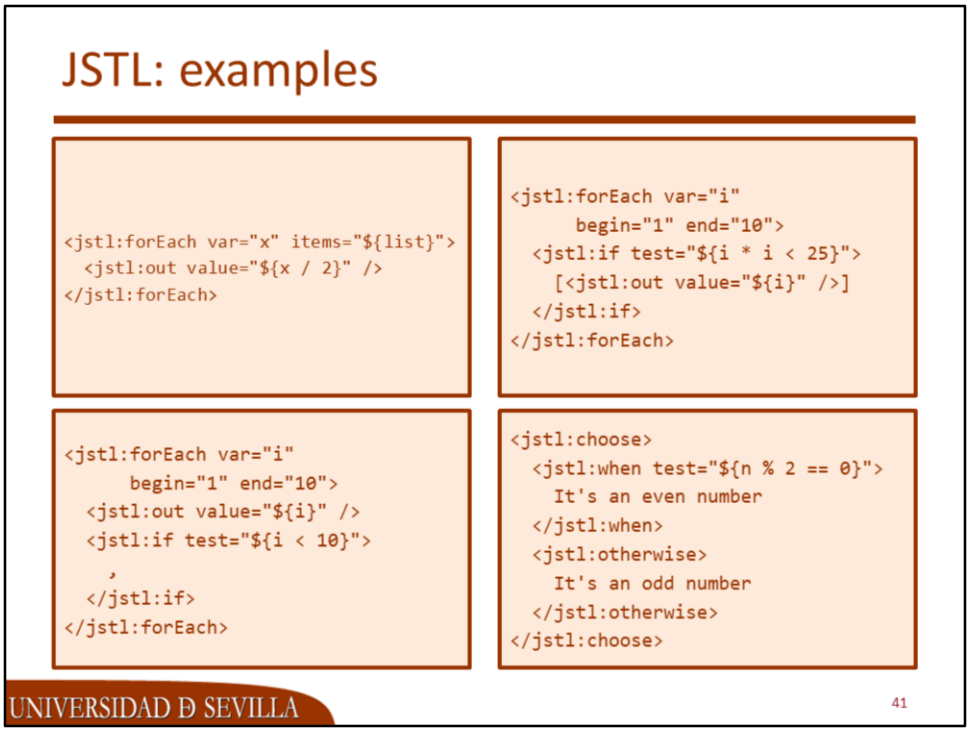
For instance,

```
<jstl:forEach var="i" begin="1" end="10"
   <jstl:out value="${i * i}" />
</jstl:forEach>
```

outputs the square of the integers in range one up to ten.

NOTE: in the sequel, we use "integer" to refer to an expression that yields an integer number, e.g., "${message.length()}", or "${x + 1}", and "collection" to refer to an expression that returns a collection of objects, e.g., "${certifications}" assuming that "certifications" is a variable of type "Collection<Certification>".

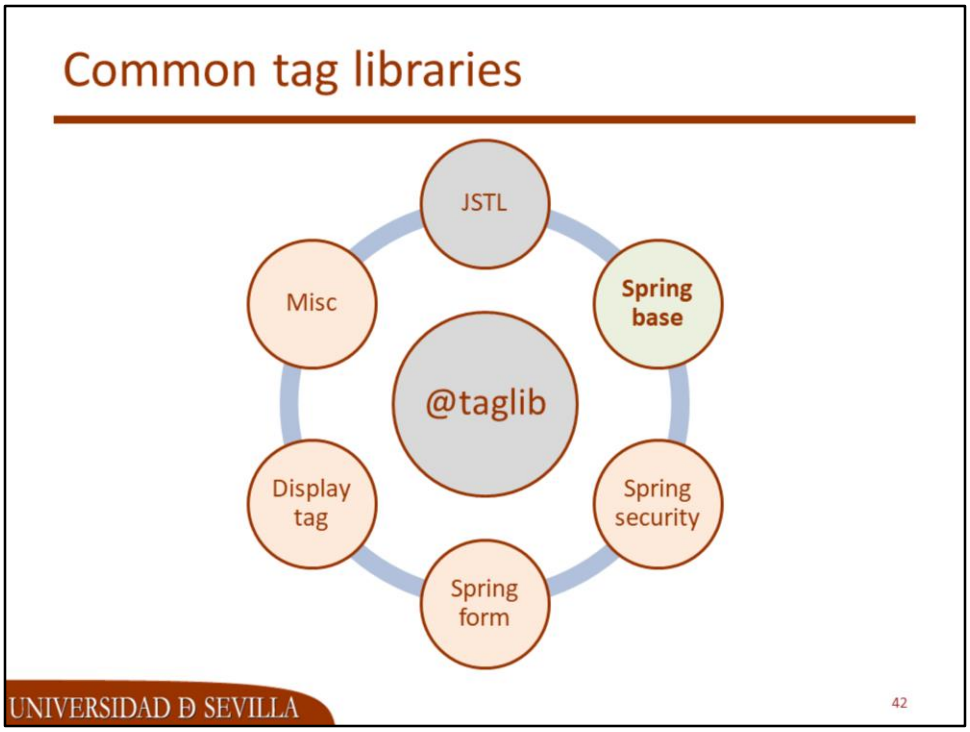
NOTE: in the sequel, we use square brackets to denote optional attributes of the tags; for instance, the "step" attribute in the first variant of the "forEach" tag is optional.



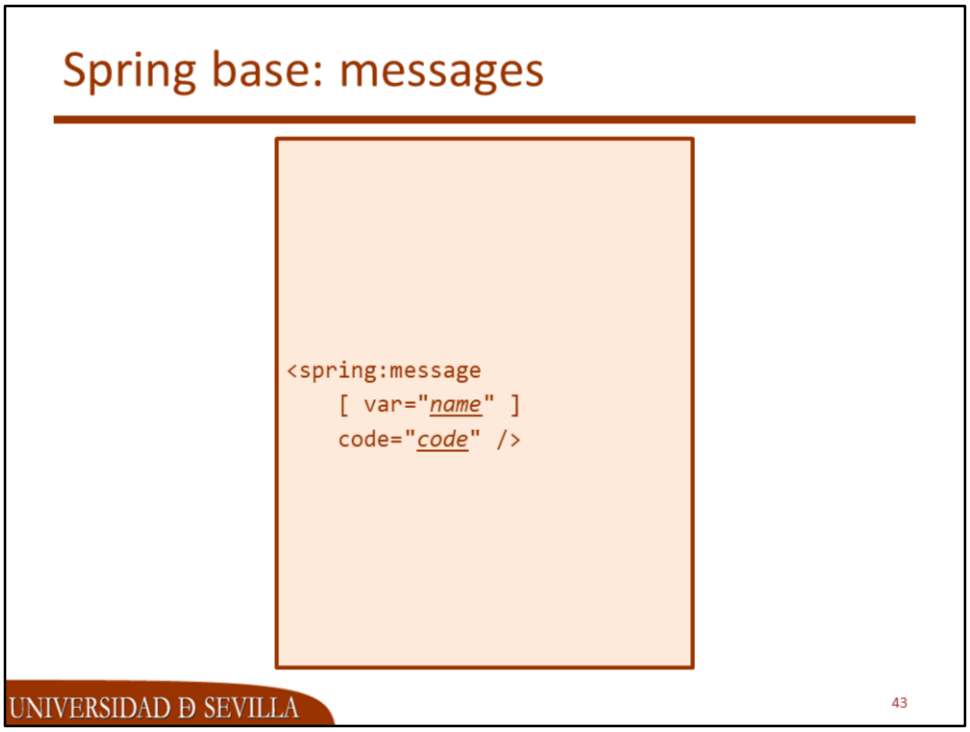
This slide shows a few examples. From left to right, from top to bottom:

- The first one assumes that the model with which the view is instantiated contains a variable called “list” that holds a collection of numbers. It uses a “forEach” tag to iterate through this collection; in each iteration it outputs one of the numbers in “list” divided by two.
- The second one iterates from 1 to 10 and outputs the numbers such that their square is less than 25.
- The third one loops from 1 to 10 and prints out the corresponding numbers; note that a comma is added unless the iterator’s reached its limit.
- The forth one assumes that there’s a variable “n” in the model and that it was set to a number; the tag outputs whether that number’s even or odd.

Please, note that the JSTL tags are very simple and that they aren’t very useful to implement complex algorithms, only simple ones like in these examples.



Let's now report on the Spring base library, which is one of the simplest ones, but very useful to internationalise and localise your views.

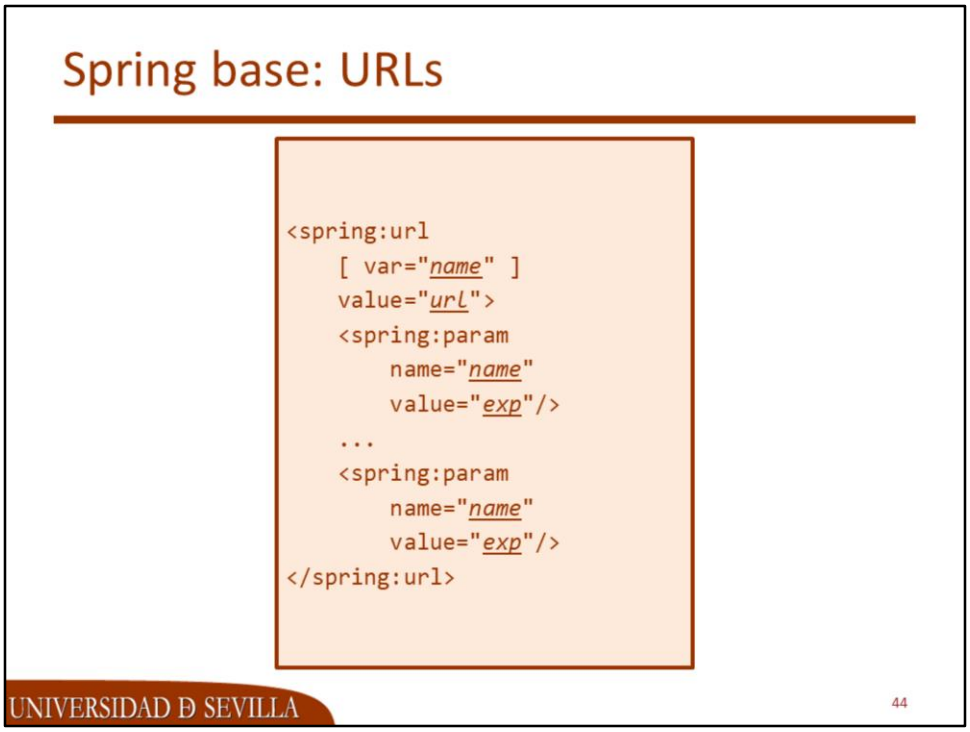


The first tag is “message”, which will definitely help you internationalise and localise your JSP documents. It has an optional attribute called “var” that allows to specify the name of a variable that will hold the message that is indicated by means of attribute “code”, which refers to a code in an i18n&l10n bundle. If the “var” attribute is omitted, then the result of executing the tag is output. For instance,

<spring:message code=“welcome.greeting” />

will output string “Welcome to ACME!” or “¡Bienvenidos a ACME!” assuming that there’s an internationalisation and localisation bundle that defines code “welcome.greeting” as “Welcome to ACME!” ” in English and ““¡Bienvenidos a ACME!” in Spanish.

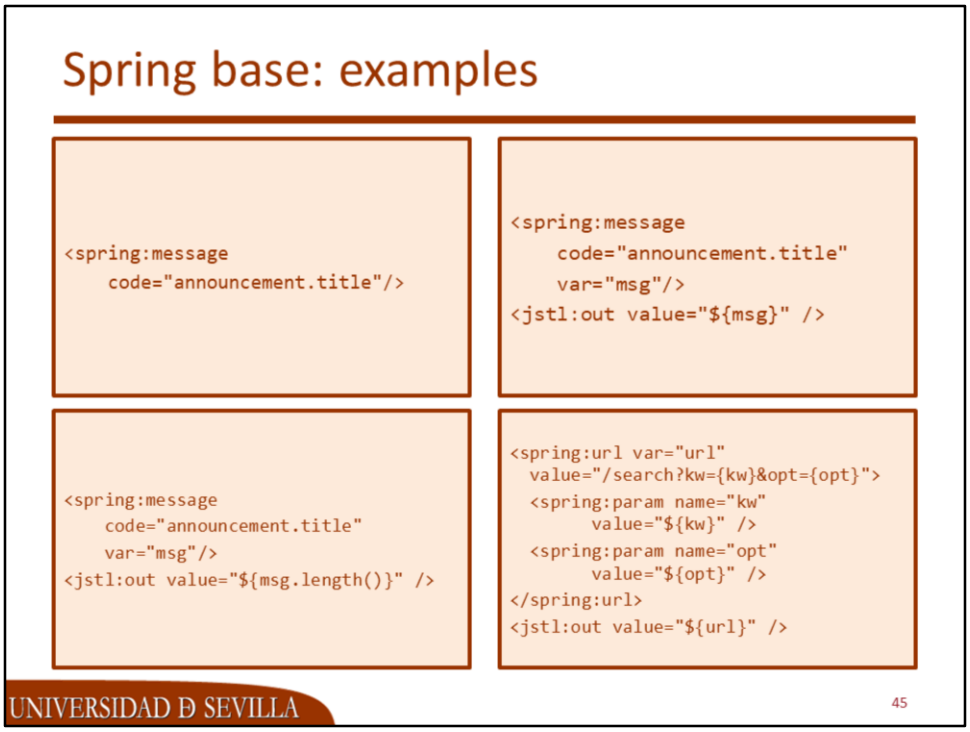
NOTE: in the sequel, we assume that “code” refers to an internationalisation and localisation code.



The next tag is “url”, which constructs a URL and stores the result in the variable specified by attribute “var”; if no variable is specified, then the result is output. The parameters to the URL are specified by means of “param” tags. For instance,

```
<spring:url
   value=“http://www.acme.com/welcome.do?name={name}">
   <spring:param name=“name" value=“${username}" />
</spring:url>
```

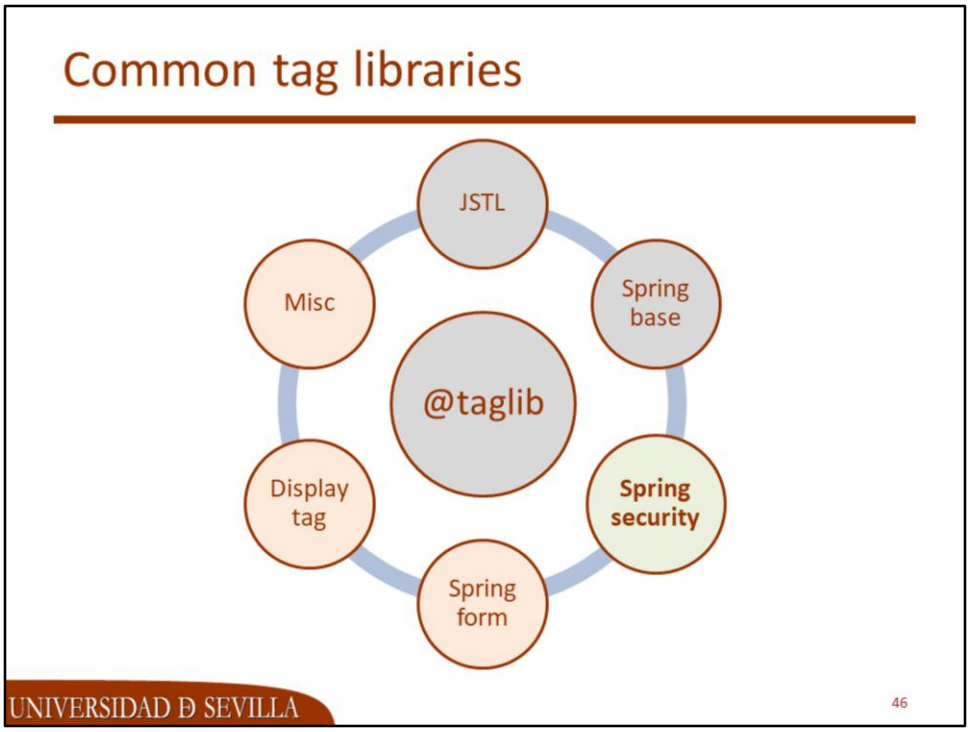
will output “http://www.acme.com/welcome.do?name=Jerôme+Citro*ën*” assuming that there’s a variable in the model called “username” whose value is “Jerôme Citroën”.



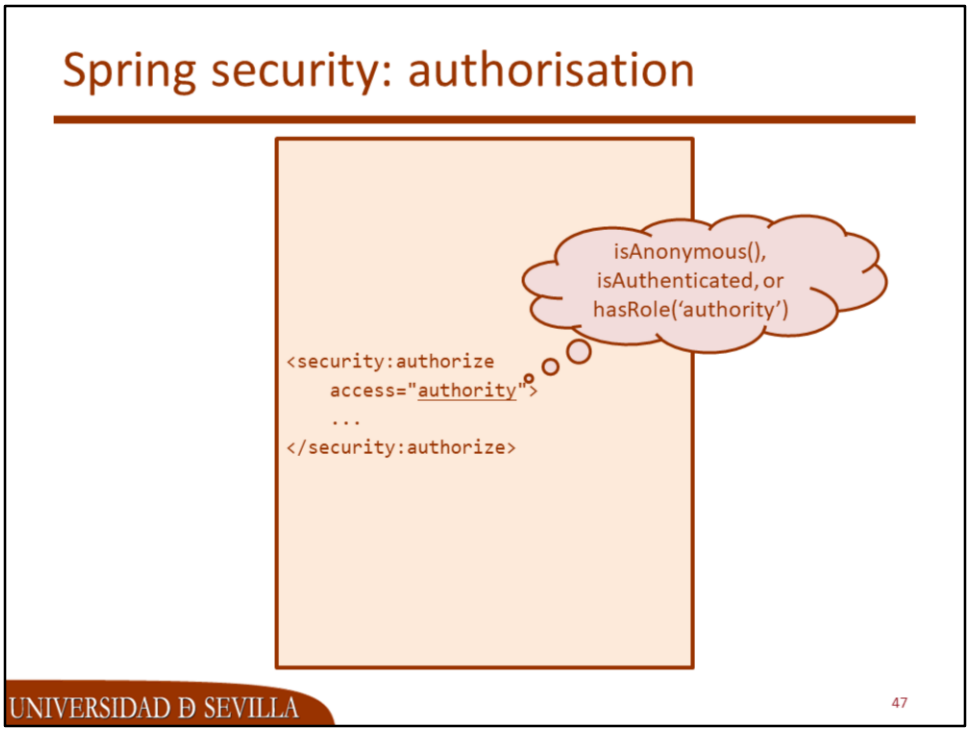
This slide shows a few examples. From left to right, from top to bottom:

- The first one outputs the message that is identified with code "announcement.title" in the active i18n&l10n bundle. If the system's working in English, this results in string "Title", but in string "Título" if it's working in Spanish.
- The second one's similar, but stores the message in variable "msg" and then outputs it using a "jstl:out" tag.
- The third example is also similar, but instead of printing out the message itself, it prints out its length.
- The fourth example outputs a URL of the following form: "http://localhost/search?kw=${kw}&opt=${opt}". For instance, if "kw" has value "Hi there!" and "opt" has value "A&B", then the result is "http://localhost/search?kw=Hi%20there!&opt=A%26B". Note that special characters like spaces or ampersands are encoded appropriately so that they can be used in a URL.

WARNING: there seems to be an bug in tag "url" that prevents it from encoding the "&" character correctly. We hope that bug's fixed in forthcoming versions.



It’s now time to explore the Spring security tag library.

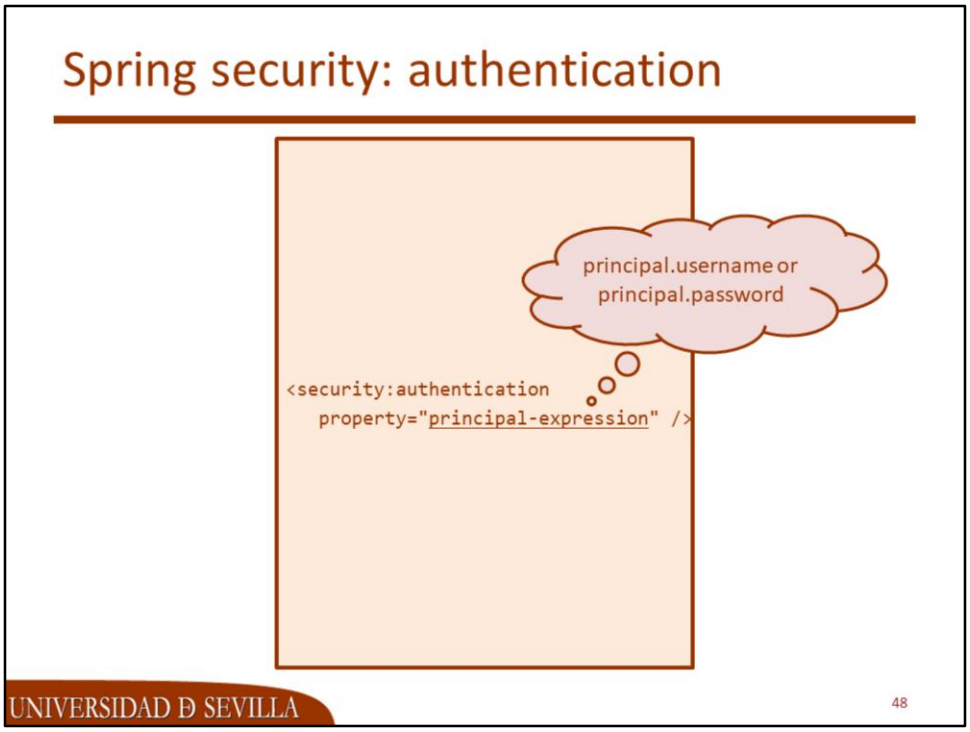


The first tag's "authorize", which has a mandatory attribute called "access" in which you can write an authority expression. For instance:

```
<security:authorize access="hasRole('ADMIN')">
   Wow, you're an admin!
</security:authorize>
```

outputs "Wow, you're an admin" as long as the principal is authenticated and has authority "ADMIN"; otherwise, nothing is output.

NOTE: in our framework, the authority expressions can be of the following form: "isAnonymous()", which evaluates to "true" as long as the principal's not authenticated; "isAuthenticated()", which evaluates to "true" as long as the principal's authenticated; and "hasRole('authority')", which evaluates to "true" as long as the principal's authenticated and has the given authority. Note that Spring's a bit inconsistent since "hasRole" doesn't check if the principal has a role, but an authority.



The next tag's "authentication". It's used to retrieve information about the user account of the principal, as long as he or she's authenticated. For instance,

```
<security:authorize access="isAuthenticated()">
   <security:authentication property="principal.username" /> |
   <security:authentication property="principal.password" />
</security:authorize>
```

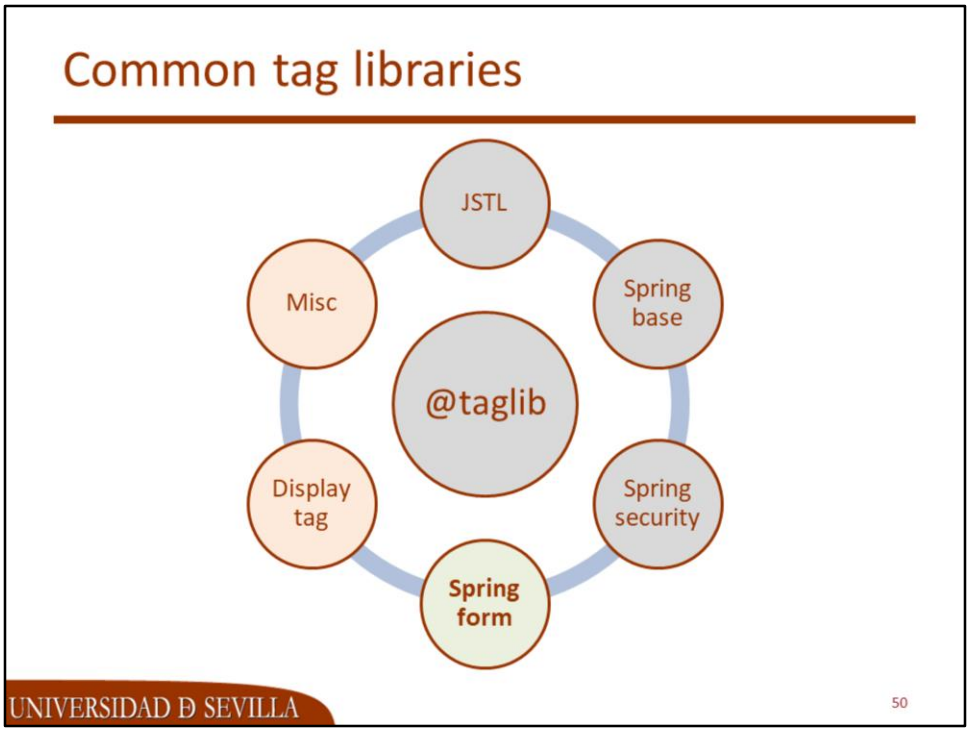
outputs "super | 1b3231655cebb7a1f783eddf27d254ca" if the principal is authenticated as user "super" and his or her password has hash "1b3231655cebb7a1f783eddf27d254ca".

NOTE: in our framework, a "principal-expression" can be either "principal.username", which provides the principal's username, or "principal.password", which provides the hash of his or her password.

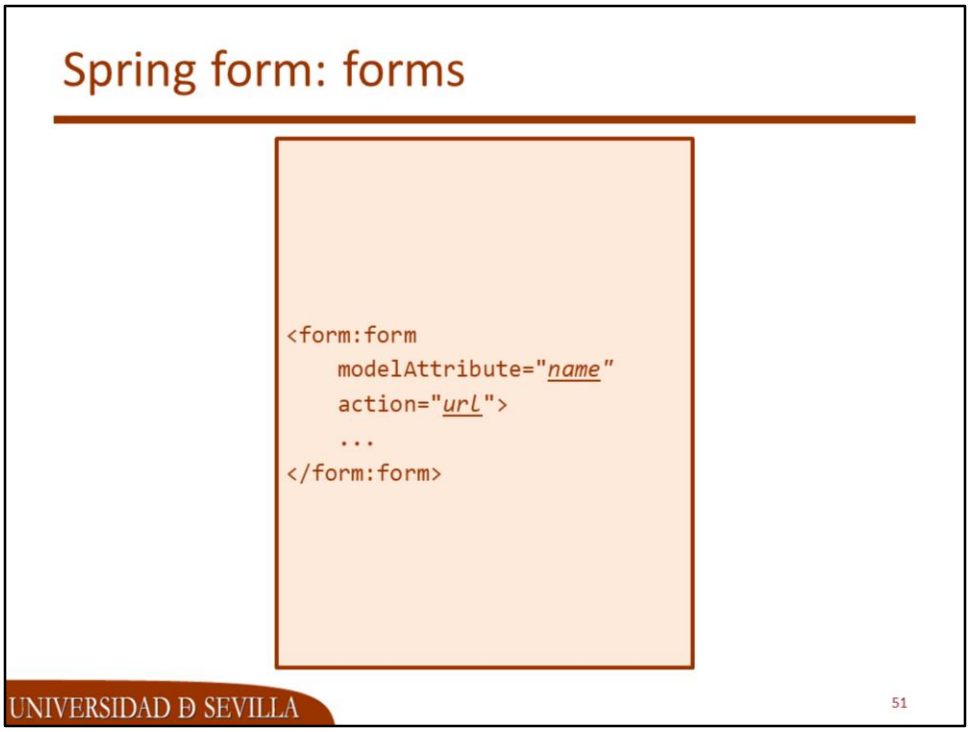
Spring security: examples

```
<security:authorize access="hasRole('ADMIN')">
  <display:column>
    <a href="announcement/administrator/edit.do?announcementId=${row.id}">
      <spring:message code="announcement.edit" />
    </a>
  </display:column>
</security:authorize>

<security:authorize access="hasRole('ADMIN')">
  <div>
    <a href="announcement/administrator/create.do">
      <spring:message code="announcement.create" />
    </a>
  </div>
</security:authorize>
```


This slide presents a couple of examples. The first example checks if the principal's authenticated and his or her user account has authority "ADMIN"; if this condition is met, then a link to edit an announcement is output. The second element's similar, but shows a link to create a new announcement. Obviously, no user other than an administrator should be presented these links.



Let's now analyse the Spring form tag library. This is more complex than the previous ones, but quintessential to implementing typical use cases.



The most important tag is “form”, which represents an HTML form. It has two mandatory attributes, namely:

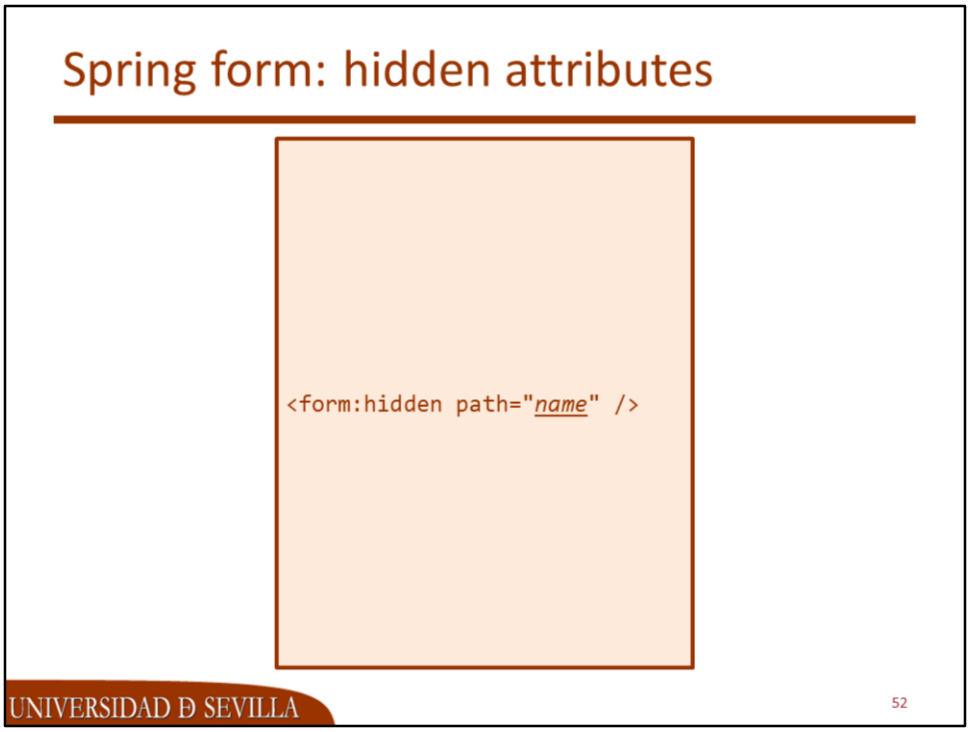
- “modelAttribute”: it indicates the name of the variable that holds the domain entity on which the form’s going to work. That variable must be available in the model with which the view’s instantiated.
- “action”: it specifies the URL to which the form must be submitted. (Submitting or posting a form means making an HTTP request to the server using the POST method.)

The tags that we’re going to present in the following slides must be used inside the body of a “form” tag. For instance,

```
<form:form modelAttribute="announcement“
action="announcement/administrator/edit.do">
  …
</form:form>
```

Introduces a form to edit an object that is referenced by means of variable “announcement” in the model; when the user presses the “save” button, the form will be posted to the “announcement/administrator/edit.do” relative URL.

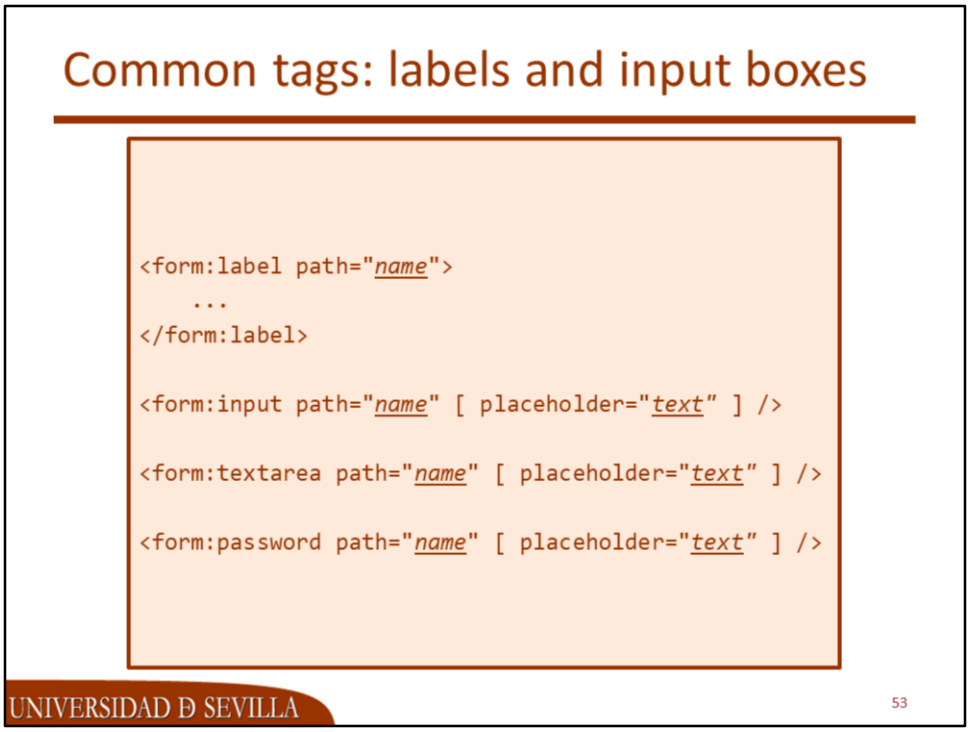
NOTE: in the sequel, “url” refers to an expression that results in a valid URL.



The following tag's "hidden", which helps hold the value of an attribute of the domain entity being edited, but doesn't display it. The name of the entity's attribute is indicated by means of attribute "path". At a first glance, that mightn't make sense to you, but it's very useful. Can you remember class "DomainEntity"? It endows every entity with two attributes called "id" and "version". The former's a unique identifier and the second's a counter that indicates how many times an entity has changed since it was stored in the database for the first time. These attributes must be stored in every form using a "hidden" tag. The reason is very simple: otherwise, the server won't be able to figure out which entity you're editing; if you simply post a bunch of attributes, but don't provide a clue on the identifier and the version of the entity with which they are associated, then the server won't be able to do anything. Did you understand that? Don't go ahead unless you understand that! For instance,

```
<form:form modelAttribute="announcement"
           action="announcement/administrator/edit.do">
  <form:hidden path="id" />
  <form:hidden path="version" />
  …
</form:form>
```

completes the form in the previous slide with a couple of hidden fields that store the values of attributes "announcement.id" and "announcement.version".



The next group of tags includes the following labels:

- "label": this tag introduces a label for an input, a text area, or a password.
- "input": this tag introduces a single-line textbox.
- "textarea": this tag introduces a multiple-line textbox.
- "password": it's similar to "input", but the user can't see the contents; generally asterisks or big dots are shown on the screen.

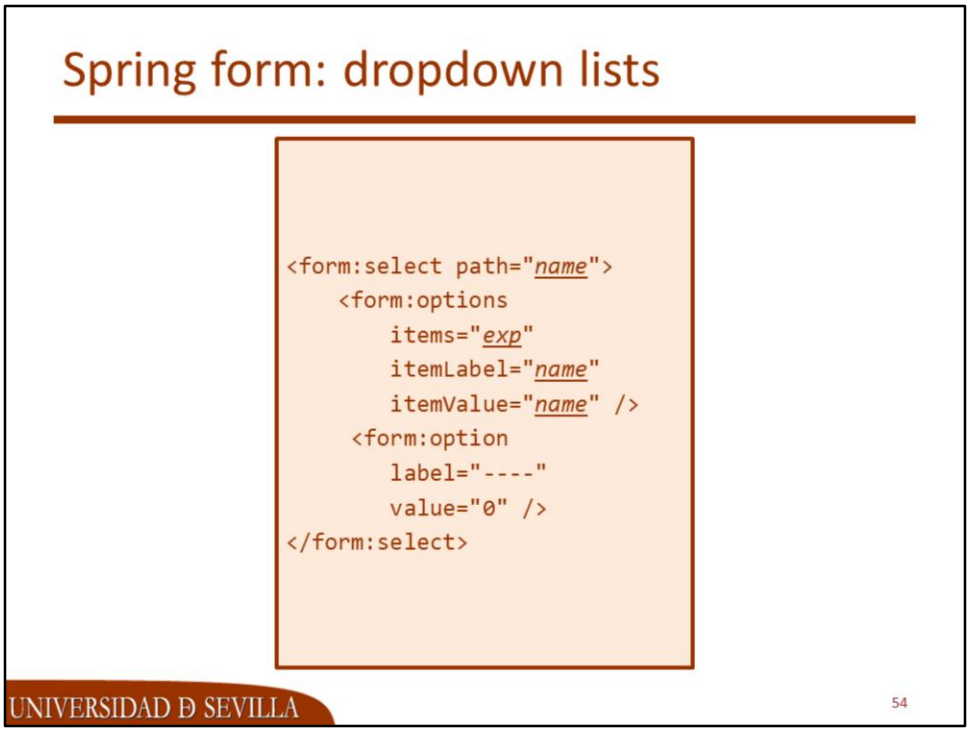
Note that the "input", the "textarea", and the "password" tags have an optional "placeholder" attribute that allows to specify a tip that helps know the format of the data expected. For instance, if you use a "form:input" tag to enter a moment, it's a good idea to use a placeholder like "dd/mm/yyyy hh:mm" to indicate the format of the date to be entered. If no placeholder is specified, then no tip is shown.

For instance,

```
<form:form modelAttribute="announcement"
                action="announcement/administrator/edit.do">
  ...
  <form:label path="text">
    <spring:message code="announcement.description" />:
  </form:label>
  <form:textarea path="description" />
  ...
</form:form>
```

completes our running example with text area that allows to edit attribute "announcement.description"; we assume that there's an internationalisation and localisation bundle that provides a definition for code "announcement.description", e.g., "Description" in English and "Descripción" in Spanish.

NOTE: in the sequel, "text" refers to an expression that returns a string.



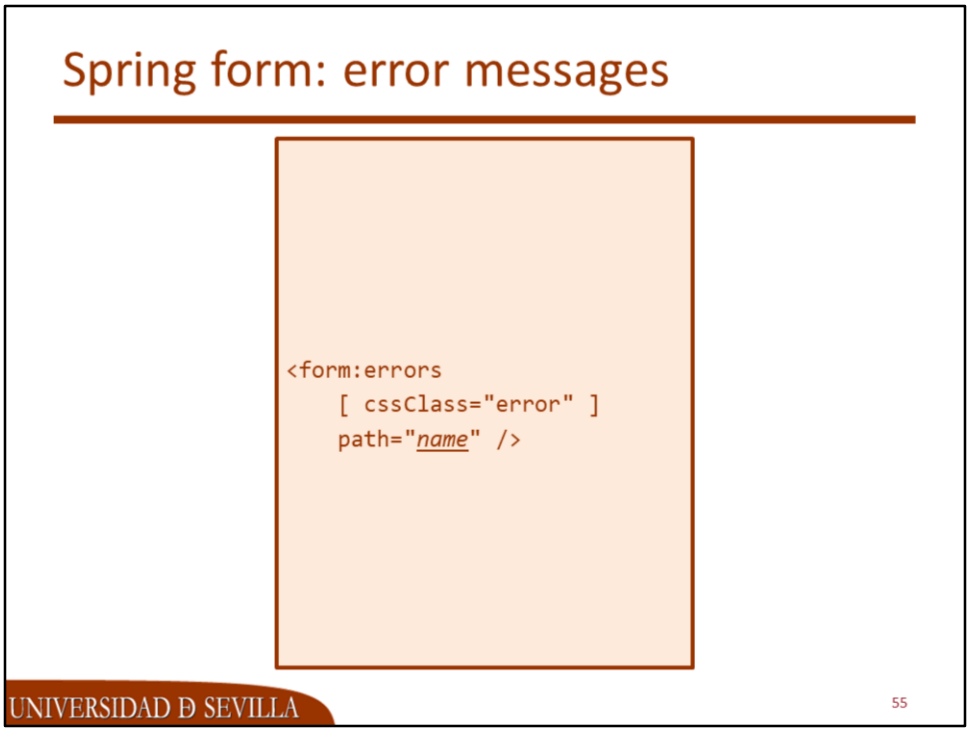
This slide reports on the “select” tag, which is used to display a dropdown list. Its attributes are the following:

- “options”: this tag is used to specify the non-default values. It has the following mandatory attributes: “items”, which must provide a collection of entities, one of which must be used to assign a value to the attribute specified in the “select” tag; “itemLabel”, which references an attribute that provides a meaningful label to the entity, and “itemValue”, which references the attribute with the identifier of the entities in the list.
- “option”: this tag’s used to specify the default value for the dropdown list. This value’s usually represented as “----”, or something similar, and its internal value is “0” since we know that no persistent entity may have this identifier.

For instance,

```
<form:form modelAttribute="announcement“
                action="announcement/administrator/edit.do">
  …
  <form:select id="exams" path="exam">
    <form:options items="${exams}" itemLabel="title" itemValue="id“ />
    <form:option value="0" label="----" />
  </form:select>
  …
</form:form>
```

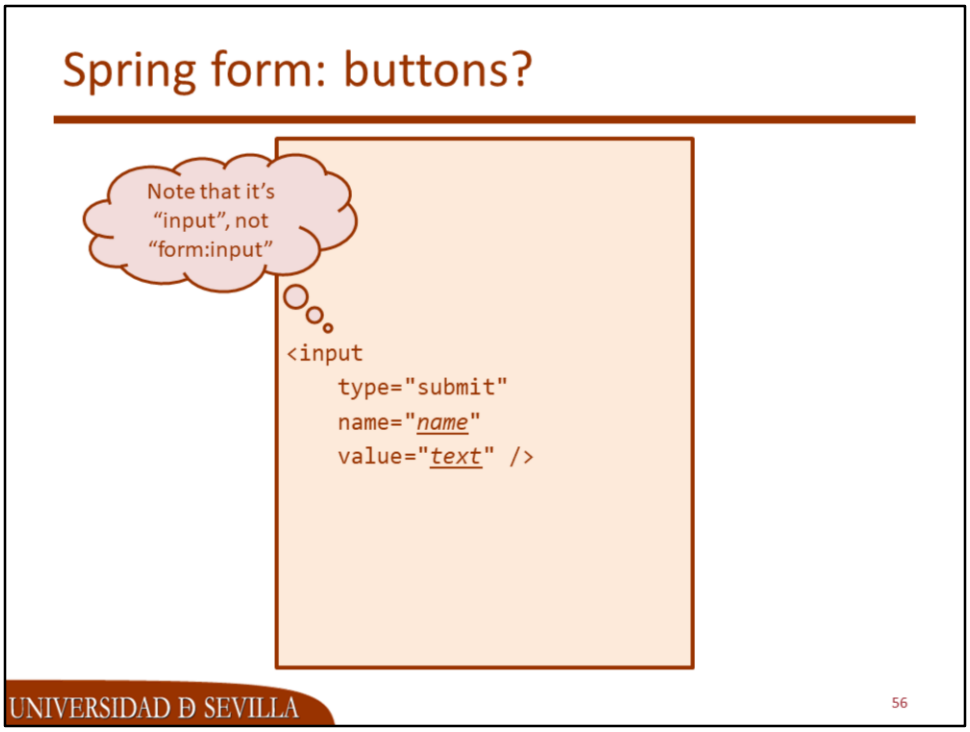
completes our running example with a dropdown list that allows to edit attribute “announcement.exam”; in the example, we assume that the model contains a variable called “exams” that has a collection of exams to show in the dropdown list; we also assume that the exams have an attribute called “title” that provides a label that must be displayed in the list and an attribute called “id” that provides a unique identifier for each exam.



And finally, there's a tag called "errors" that allows to show validation errors. Can you remember the constraints in your Java domain models? We used annotations like "NotBlank" or "Min(0)" to specify that an attribute can't be blank or less than zero. The user, however, can type in whatever he or she likes in an input box, text area, or password box, or he or she can leave a dropdown list to its default value. This tag shows an error message next to an input box, text area, password box, or dropdown list to inform the user that the value he or she's entered is invalid. This tag has an optional attribute called "cssClass" by means of which you can assign a CSS class to endow your error messages with a little colour; our framework provides a default CSS class called "error", which we recommend that you should use all the time. For instance,

```
<form:form modelAttribute="announcement"
                action="announcement/administrator/edit.do">
  …
  <form:label path="title">
    <spring:message code="announcement.title" />:
  </form:label>
  <form:input path="title" />
  <form:errors cssClass="error" path="title" />
  <br />
  …
<form:form/>
```

includes a text box to edit attribute "announcement.title"; if there are any errors, then they are shown next to the input box thanks to the "errors" label.



Unfortunately, the Spring from tag library doesn't provide a "button" tag. We have to resort to plain HTML to introduce a button by means of an "input" tag ("input", not "form:input"). This tag has the following attributes:

- "type": the value must be "submit".
- "name": this is a key that will help us identify the button that submitted the form to the server. Note that the submission of a form results in a POST request to the server. It's important that the request includes the name of the button that was pressed to submit the form or, otherwise, we won't be able to discern if we have to save an entity or to delete it, to mention a typical example.
- "value": it specifies the text that will be displayed on the button; obviously, this text must be retrieved from an i18n&l10n bundle, which implies that you must use a "spring:message" tag, but you already know about this. (Please, pay attention to the example that we're going to present in the next slide; the notation used to internationalise and localise the "value" attribute is a little confusing at first.)

For instance,

```
<form:form modelAttribute="announcement"
                action="announcement/administrator/edit.do">
  …
   <input type="submit" name="save" value="<spring:message
code="announcement.save" />" />
  …
<form:form/>
```

introduces a "save" button in our form. Please, note that the "value" attribute is internationalised by means of an inner "spring:message" tag.

Spring form: example

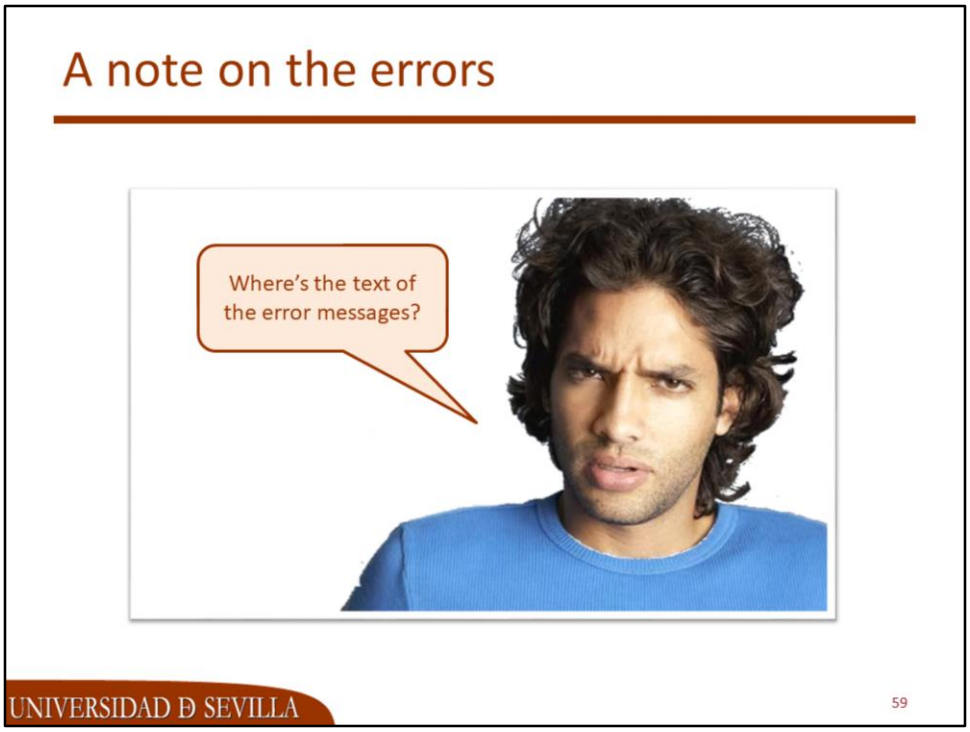
```
<form:form action="announcement/administrator/edit.do"
    modelAttribute="announcement">

    <form:hidden path="id" />
    <form:hidden path="version" />

    <form:label path="title">
      <spring:message code="announcement.title" />:
    </form:label>
    <form:input path="title" />
    <form:errors cssClass="error" path="title" />
    <br />
    …
    <input type="submit" name="save"
       value="<spring:message code="announcement.save" />" />
    …
</form:form>
```


This slide shows an excerpt of a simple form to edit announcements. Below, we provide an explanation of the most interesting parts:

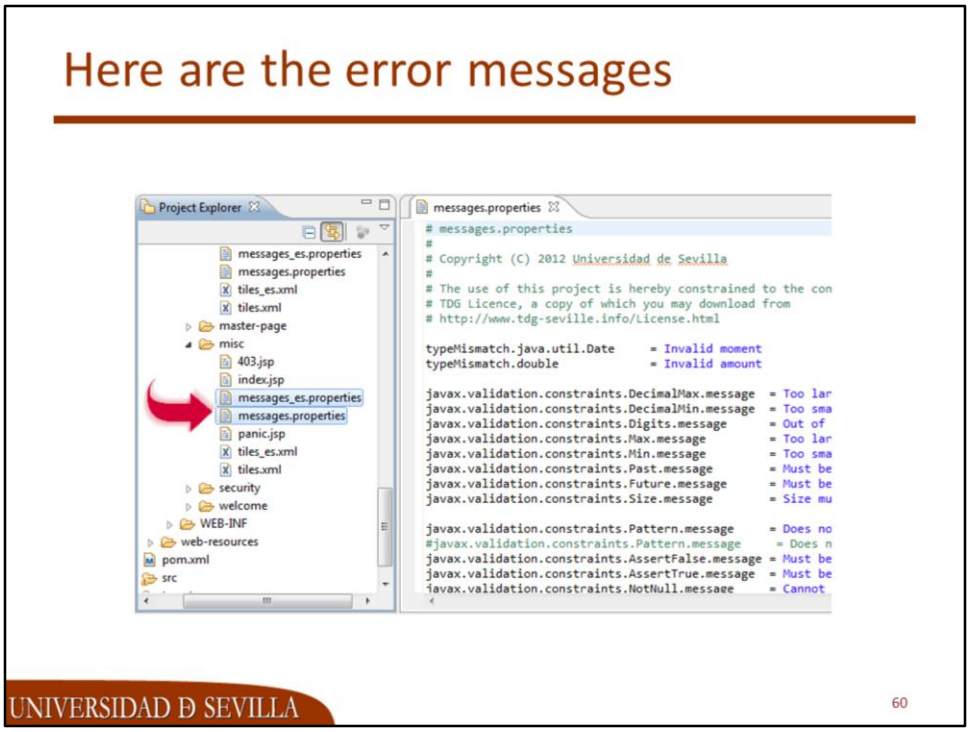
- The form posts to relative URL “announcement/administrator/edit.do”. There must be a controller that serves the requests for this URL. (More on this later.)
- This is a form that uses an object called “announcement” that must be provided by the model used to instantiate the view in which this form was included.
- The form has two “hidden” tags that store the identifier and the version of the object being edited. Note that we must store these values in the HTML page that is sent to the user or otherwise when he or she presses the save button to store the changes, the system won’t be able to know which entity it is.
- Then come some blocks of the form label-input-errors-br. Each such block produces a label (with a message that is localised to the language in which the system’s working), then a textbox or a dropdownlist, and then an error box. In the example, we only illustrate the block that corresponds to the title of the announcement being edited; the blocks for the moment, the description, or the other attributes are very similar.
- After the attributes, we put the buttons to save, edit, or cancel the edition. Here, we only illustrate the save button. Note that the way we provide the text for the save button is a little confusing, since we need to insert a Spring tag inside an HTML tag, but this is the only way to localise the label of a button.

What it looks like

This slide shows how the form looks within the context of our master page. Nice, isn't it? Note that there are some validation errors shown in red because the user entered invalid data.



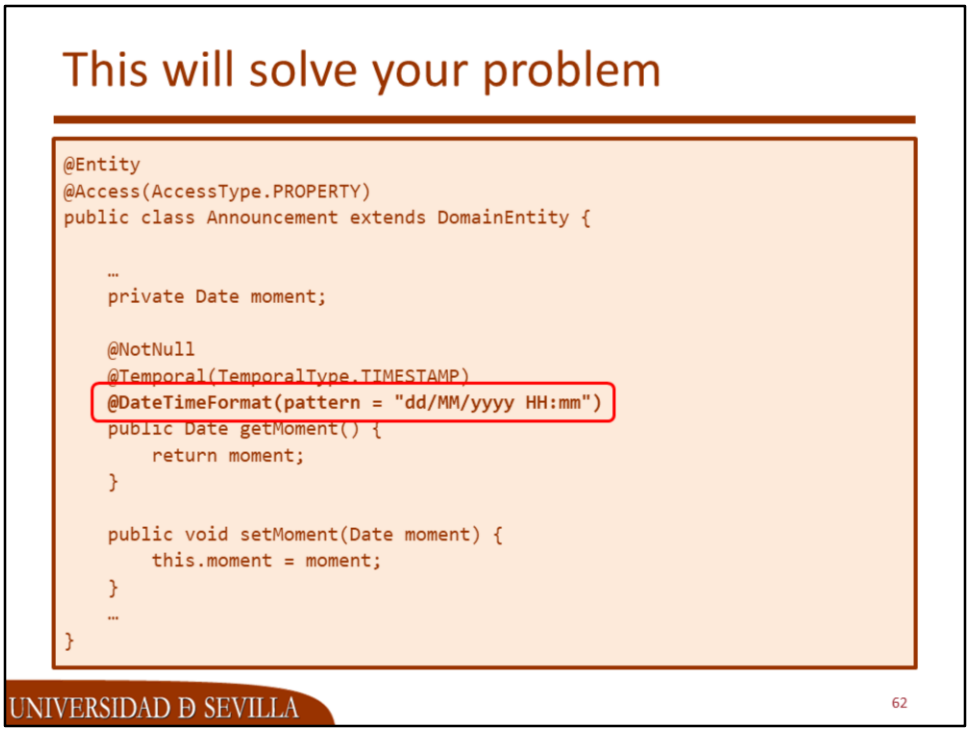
We're sure you're asking this: where's the text of the error messages? When we presented the "errors" tag, we mentioned a CSS class and an attribute name, but we didn't specify a message. So, where are messages "Invalid moment", "Must not be blank", or "Cannot be null"? More than that: how can we internationalise and localise them?



In the “Deployed resources” virtual folder of our project template, there’s another folder called “misc” that contains several miscellaneous files, including “messages.properties” and “messages_es.properties”. These files have the messages that tag “form:errors” displays. They are i18n&l10n bundles, so their structure’s exactly the same as we presented previously when we studied the “spring:message” tag.

A note on the moments

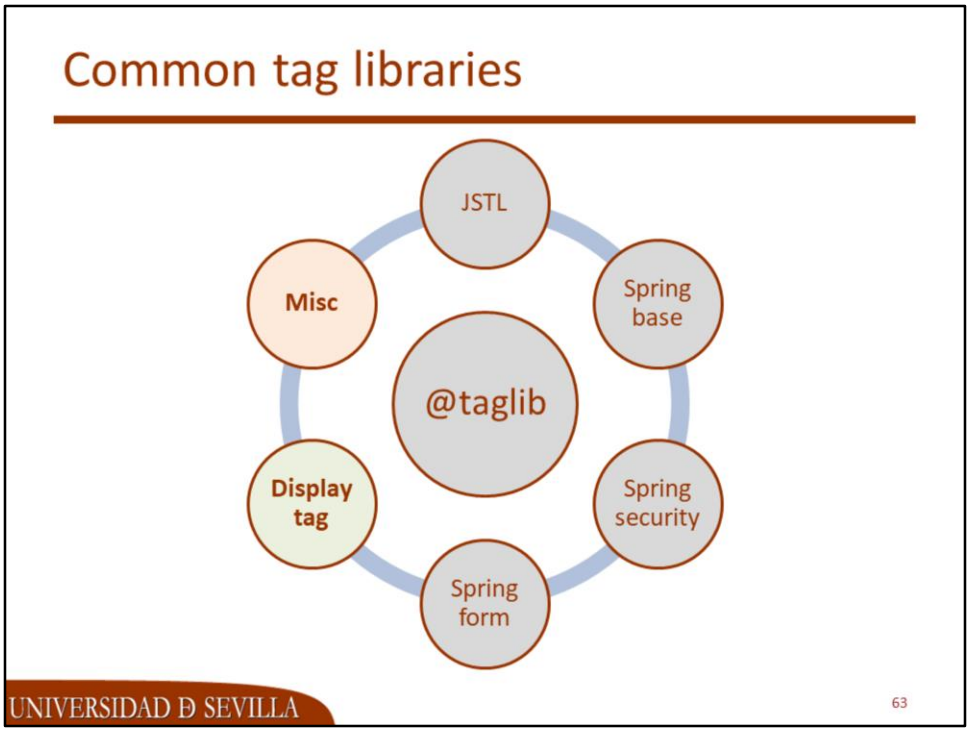
Before concluding this section, we must mention an additional problem with displaying and editing dates.



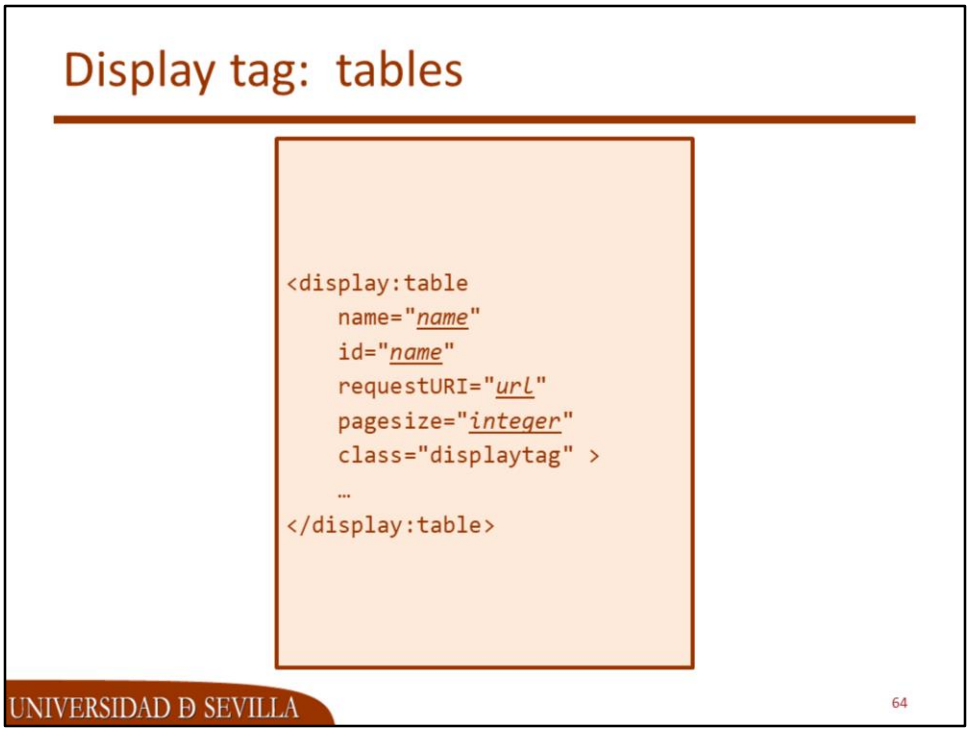
You know that you can use a "form:input" tag to edit an attribute of a simple type, which includes strings, integers, doubles, and also dates. The problem with dates is that there are too many formats to represent them; for instance a date like "01/02/03" may refer to "February 1, 2003", to "January 2, 2003", or to "February 3, 2001", just to mention a few common interpretations. To solve this problem, we must make a small change to our Java domain model and add a "@DateTimeFormat" annotation to every attribute of type "Date". This annotation takes a parameter called "pattern" that indicates how a string that represents a date must be interpreted or how a date object must be rendered as a string. In the slide, we use quite a common format: "day/month/year hours:minutes". Take a look at https://docs.oracle.com/javase/7/docs/api/java/text/MessageFormat.html for further information on the available formats.

WARNING: if you followed our instructions in Lesson L03 - Persistence models, then you entered dates in configuration file "PopulateDatabase.xml" using the following format: "yyyy/MM/dd HH:mm". The problem's that there's an interference if you use this annotation: you must change your dates to use format "dd/MM/yyyy HH:mm" or otherwise your dates will be misinterpreted. We're sorry. Technology is far from perfect!

WARNING: the "@DateTimeFormat" annotation just sets a format. A moment like "32/01/2018 12:00" conforms to this format and it is a valid date; it refers to the first day of February, 2018. This is very confusing at first, but that's technology!



Let's move on! Let's now analyse the Display tag library.



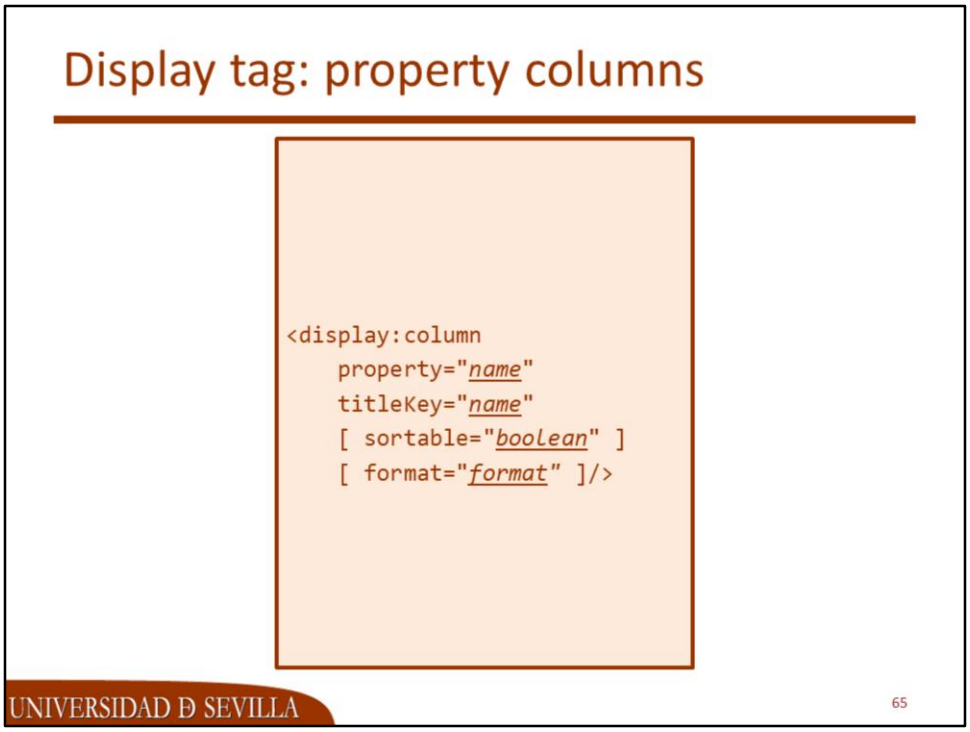
The Display tag library provides a tag called “table” that we can use to present collections of domain entities in a grid. The attributes are the following:

- “name”: this attribute indicates the name of a variable that contains a collection of domain entities to be listed. The variable must obviously be stored in the model that is used to instantiate the view in which this tag’s used. (Note that this attribute is the equivalent to “modelAttribute” in the previous tag library.)
- “id”: this attribute defines a variable that iterates over the domain entities in the collection. Please, don’t mistake it for the “id” attribute in HTML tags! It’s the name of an iterator variable by means of which we can have access to the individual entities in the collection to be displayed.
- “requestURI”: this attribute indicates the URL from which the collection of domain entities must be retrieved when the user clicks on a pagination link. (It’s similar in spirit to the “action” attribute in the previous tag library.)
- “pagesize”: this attribute’s an integer that indicates how many entities per page must be shown.
- “class”: this attribute indicates the name of the CSS class that the grid must use. Please, don’t change the default value of this attribute.

For instance,

```
<display:table name="announcements" id="row“ requestURI="${requestURI}"
                pagesize="5" class="displaytag">
  …
</display:table>
```

Introduces a table that shows the objects in a model variable called “announcements”; it iterates over the collection using a model variable called “row”; when a pagination link is clicked, the table requests more items from the URL in the model variable called “requestURI”; the default page size is five items; and the default CSS class is “displaytag”.



Columns are specified by means of the “column” tag. The first variant of this tag is shown on this slide. It allows to include a column that shows the values of a given attribute. The tag has the following attributes:

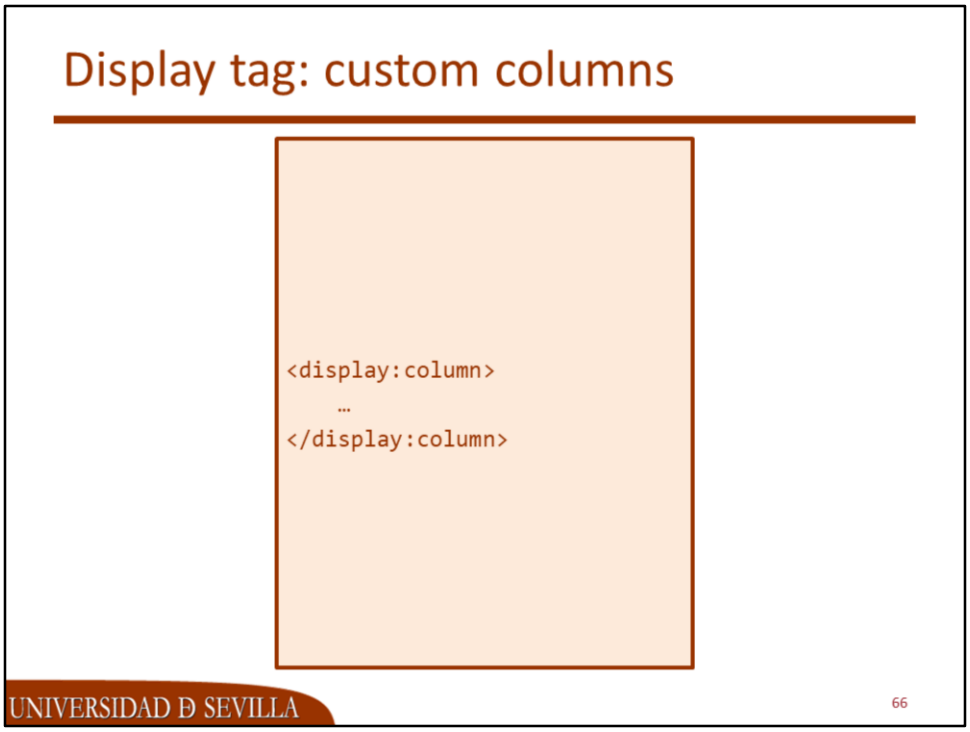
- “property”, which indicates the name of an attribute in the domain entities being listed;
- “title”, which indicates the title of the column;
- “sortable”, which indicates if the column is sortable or not;
- and “format”, which indicates how to format the value of the attribute, if necessary.

For instance,

```
<display:table name="announcements" id="row“ requestURI="${requestURI}"
                pagesize="5" class="displaytag">
  …
  <display:column property="moment" titleKey="announcement.moment"
                  sortable="true" format="{0,date,dd/MM/yyyy HH:mm}" />
  …
</display:table>
```

introduces a column that shows the value of attribute “row.moment”, where “row” is the variable used to iterate over collection “announcements”, the column is sortable, and its format is “day/month/year hours:minutes”.

NOTE: in the sequel, “format” refers to a Java message format. They are documented at https://docs.oracle.com/javase/7/docs/api/java/text/MessageFormat.html.



You may also include custom columns by means of tag “display:column”. This tag let you design what you wish to show in the column. For instance,

```
<display:table name="announcements" id="row“ requestURI="${requestURI}" pagesize="5"
class="displaytag">
  …
  <security:authorize access="hasRole('ADMIN')">
    <display:column>
      <a href="announcement/administrator/edit.do?announcementId=${row.id}">
          <spring:message code="announcement.edit" />
      </a>
    </display:column>
  </security:authorize>
  …
</display:table>
```

introduces a column that shows a link to edit the announcements whose id is in attribute “row.id”, where “row” is the variable used to iterate over collection “announcements”; note that the column is shown as long as the principal’s authenticated as an administrator thanks to the “authorize” label.

Display tag: example

```
<display:table name="actors" id="row"
    requestURI="actor/administrator/list.do"
    pagesize="5" class="displaytag" >

    <display:column property="name" titleKey="actor.name" />
    <display:column property="email" titleKey="actor.email" />
    <display:column property="phone" titleKey="actor.phone" />
    <display:column property="address" titleKey="actor.address" />

</display:table>
```

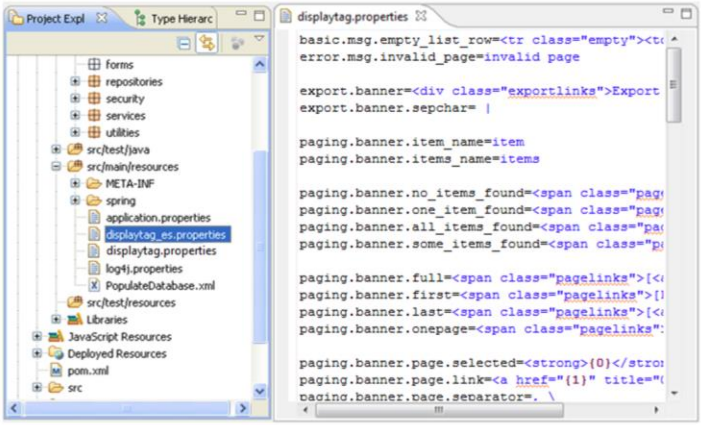

In this slide, we show an example that displays the list of actors of your system. Note that the list is provided by requesting the relative URL “actor/administrator/list.do” and that it’s available to the tag by means of variable “customers” in the model; the individual objects will be available by means of variable “row”, as indicated in the “id” attribute. We’ve set up several attribute columns to display each of the attributes of an actor.

What the example looks like

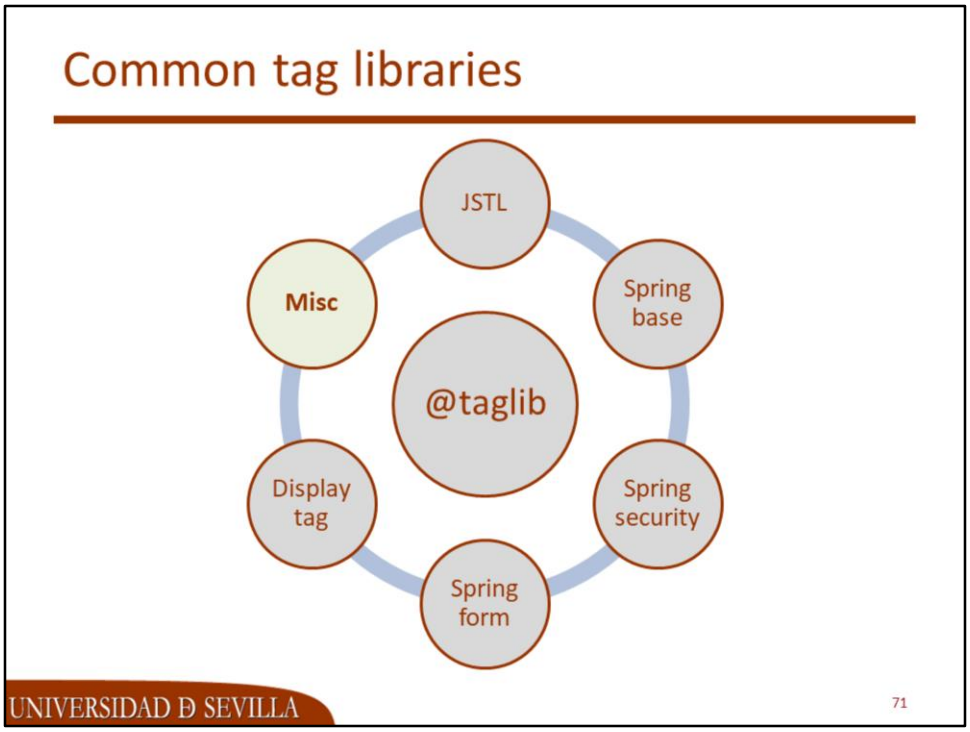
This is what the listing looks like in the context of our master page. Nice, isn't it?

What about i18n & l10n? This is a good question. In the previous slide we presented a sample listing in English. Note that we know how to internationalise and localise the headers of a grid, but what about the messages that the grid itself displays? For instance: “3 items found”.

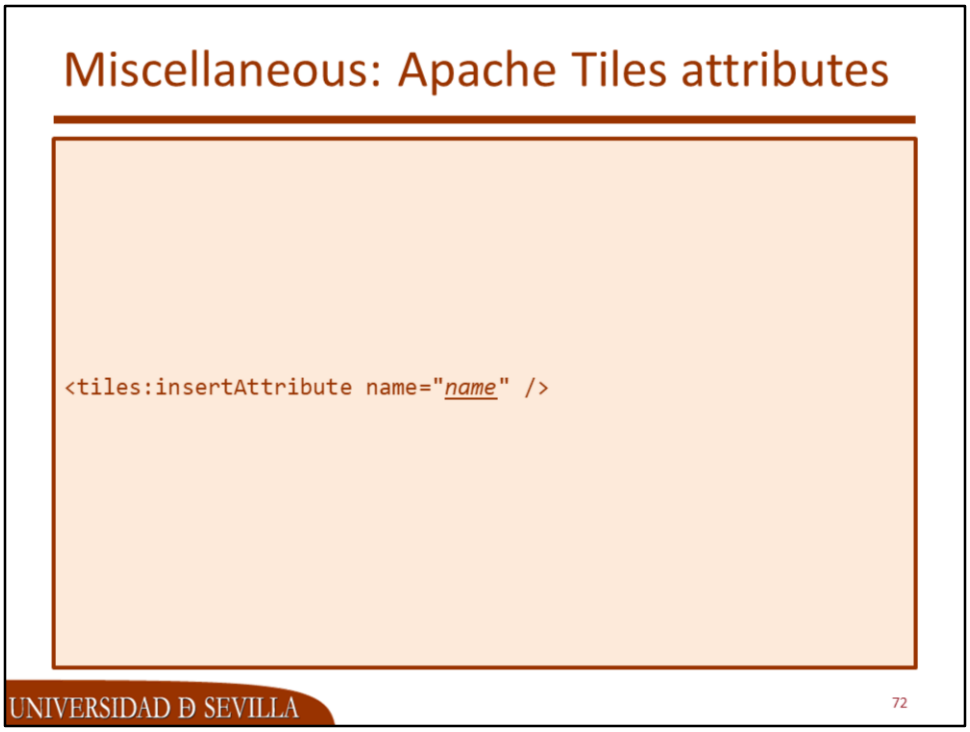
These are the i18n & l10n bundles



The Display tag library uses the i18n&l10n bundles stored in folder "src/main/resources/" whose names are "displaytag.properties" or "displaytag_XX.properties" (where "XX" refers to an ISO 639 code).



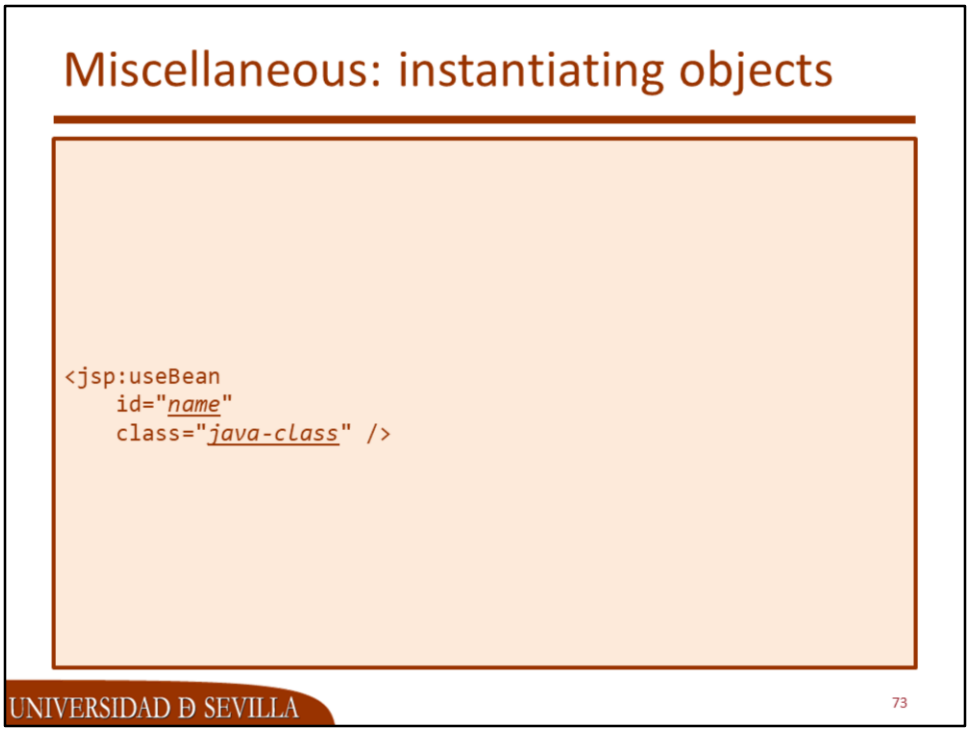
Finally, let's take a look at some miscellaneous tags.



The first miscellaneous tag is “tiles:insertAttribute”, which allows to insert the value of a tiles attribute somewhere in a JSP view. For instance,

```
…
<h1> <tiles:insertAttribute name="title" /> </h1>
…
```

introduces the value of attribute “title” within a first-level header tag.

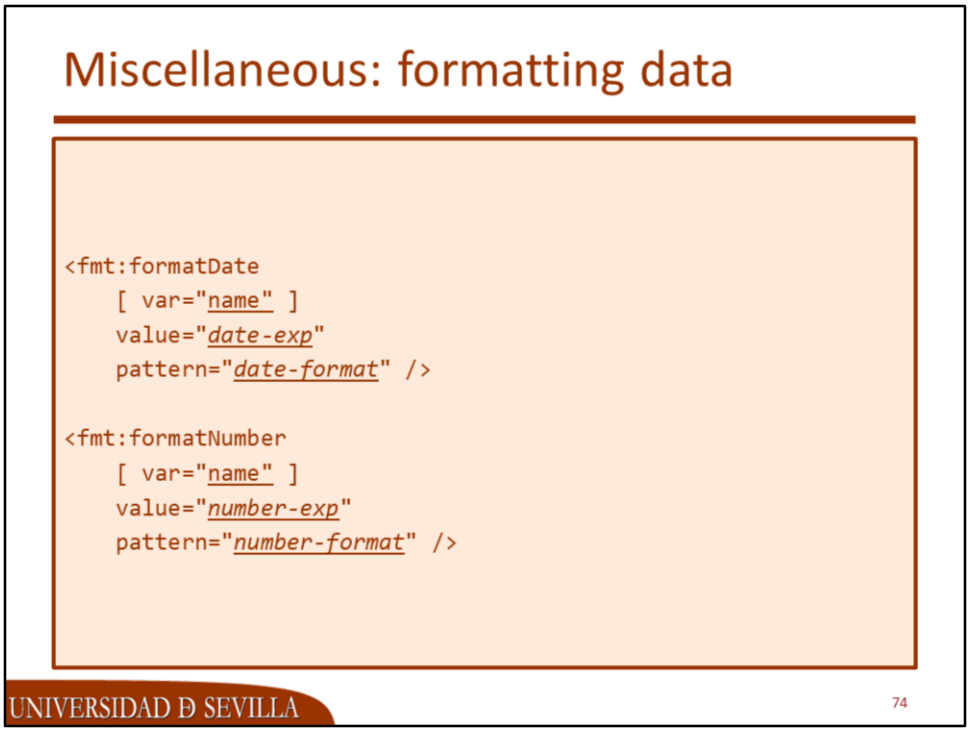


The second one is “jsp:useBean”, which allows to introduce a new variable whose name’s indicated in attribute “id”; the variable’s instantiated to a new object of the class specified in attribute “class”. For instance,

```
<jsp:useBean id="date" class="java.util.Date" />
```

creates an instance of class “java.util.Date” and assigns it to model variable “date”.

NOTE: in the sequel, “java-class” refers to a fully-qualified Java class name.



The last miscellaneous tags are “fmt:formatDate”, which allows to format a date according to a format, and “fmt:formatNumber”, which allows to format a number according to another format. Note that there’s an optional attribute called “var” that allows to store the value of the expression in a model variable; if no value is specified, then the result is output. For instance,

```
<fmt:formatDate value="${date}" pattern="yyyy" />
```

shows the year of a date object called “date”.

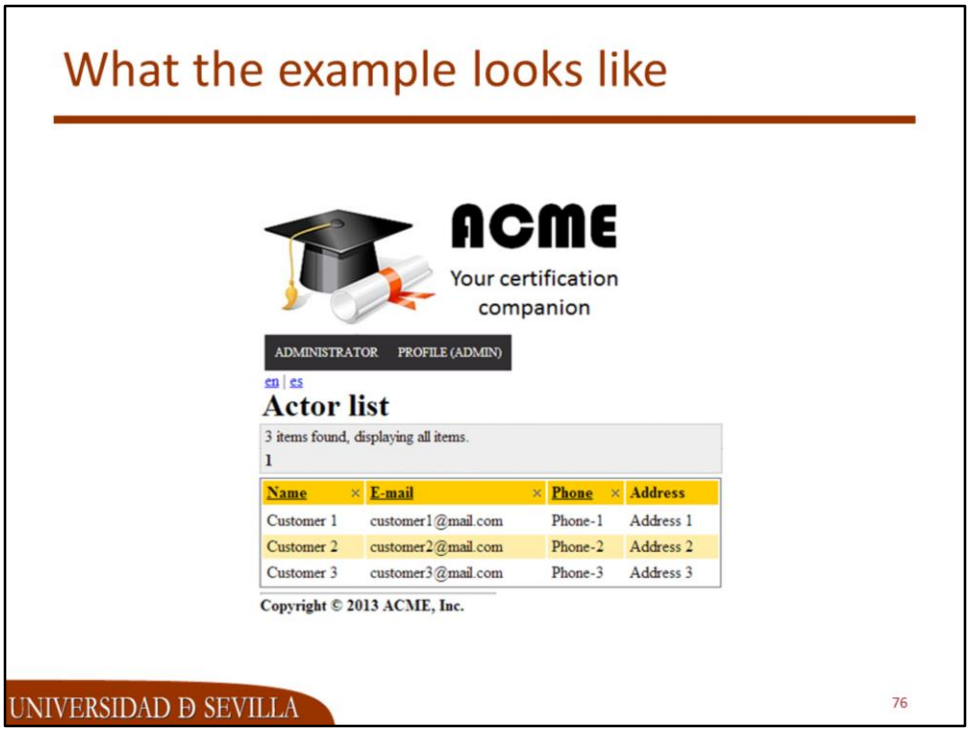
NOTE: in the sequel, we use “date-exp” to refer to an expression that returns an object of type “java.util.Date”, “date-format” refers to one of the standard date formats, “number-exp” refers to an expression that returns a number, and “number-format” refers to one of the standard number formats. The standard formats are available at https://docs.oracle.com/javase/7/docs/api/java/text/MessageFormat.html. Note that they are generally of the form “{index, type, [subtype,] style}”, but only the “style” part is used in tags “formatDate” and “formatNumber”. For instance, to format a date, you must use a date format like “yyyy/MM/dd”, not a date format like “{0, date, yyyy/MM/dd}”.

Miscellaneous: example

```
…
<h1> <tiles:insertAttribute name="title" />  </h1>
…
<b>
   <jsp:useBean id="date" class="java.util.Date" />
   Copyright &copy;
   <fmt:formatDate value="${date}" pattern="yyyy" />
   ACME, Inc.
</b>
…
```


This slide shows a simple example with an excerpt of a view in which we use “tiles:insertAttribute” to introduce its title in an “h1” header and then create a “Date” object, assign it to variable “date”, and format it using pattern “yyyy”, which keeps only the year.

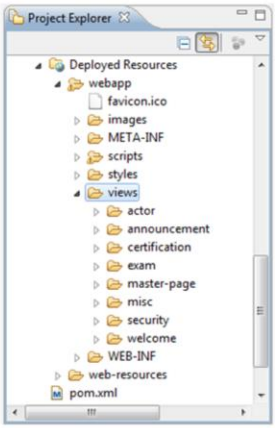
What the example looks like



This is how the example looks like in a typical page.

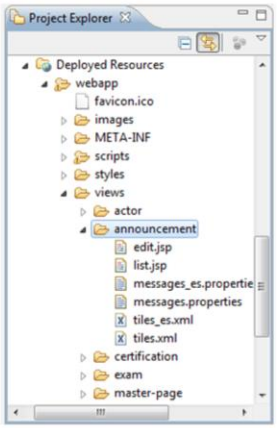
Let's move on! Let's now talk on the configuration that you must carry out so that your projects can use your views.

The views folder



The first thing you must know is that your views are stored in a folder called "webapp/views", to which you can have access through the "Deployed Resources" virtual folder. Note that the "webapp/views" folder is divided into a number of folders, the majority of which have names that correspond to an entity in our domain model.

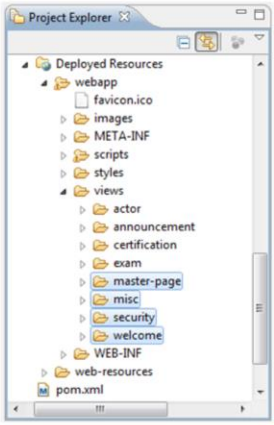
Structure of a subfolder



Each such subfolder contains a number of Apache Tiles documents, JSP documents, and i18n&l10n bundles that are related to an entity in our domain model. Typically, these include the following:

- “edit.jsp”: it’s a JSP document that defines an edition form.
- “list.jsp”: it’s a JSP document that defines a grid to list entities.
- “messages_es.properties” and “messages.properties”: they’re the i18n&l10n bundles required to render the previous documents. The former provides messages in Spanish and the latter in English.
- “tiles_es.xml” and “tiles.xml”: they’re the Apache Tiles documents that extend our master page to define the page in which we list our entities or edit them. The former defines the pages in Spanish and the second in English.

Predefined subfolders

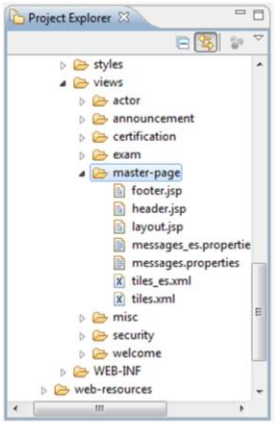


UNIVERSIDAD Đ SEVILLA

80

Our project template provides four predefined subfolders, namely: master-page, misc, security, and welcome. In the following slides, we provide a few more details on them.

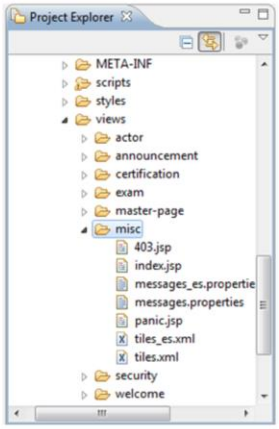
The master-page subfolder



The “master-page” subfolder provides the following files:

- “footer.jsp”: it’s the footer of our master page; typically, you need to change the copyright message only.
- “header.jsp”: it’s the header file of our master page; typically, you need to change the menu bar. Please, find additional information on how to configure the menu bar at http://www.myjqueryplugins.com/jquery-plugin/jmenu.
- “layout.jsp”: it’s the template of our master page; it’s very unlikely that you need to modify it.
- “messages_es.properties” and “messages.properties”: these are the i18n&l10n bundles for the master page; typically, you need to add new message codes to support the options of your menu bar.
- “tiles_es.xml” and “tiles.xml”: these are the Tiles documents that define our master page; it’s very unlikely that you need to modify it.

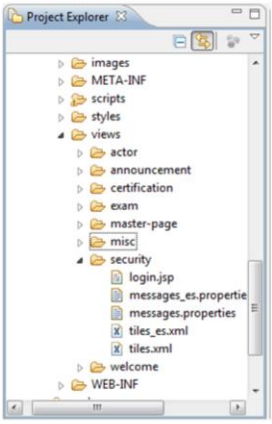
The misc subfolder



The “misc” subfolder has some miscellaneous files, namely:

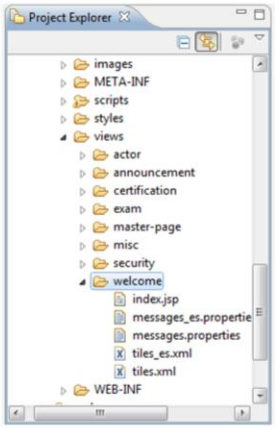
- “403.jsp” : this is a view that displays the following message when a user attempts to have access to a URL that is forbidden to him or her: “Oops! It seems that you don't have access to this resource.” You’re not likely to change this view in this subject.
- “index.jsp”: this view redirects to “/welcome/index.do”, which is the welcome page. Unfortunately, it doesn’t seem possible with the current technology to present a welcome page unless we redirect to it. Another problem with technology, sorry.
- “messages_es.properties” and “messages.properties”: these are the i18n&l10n bundles that define the validation errors that tag “form:errors” displays.
- “panic.jsp”: it’s a view that displays a panic message; a panic message is produced whenever one of your controllers throws an exception. This view displays a sorry message and a stack trace.
- “tiles_es.xml” and “tiles.xml”: these are the Apache Tiles configuration files for the panic web page.

The security subfolder

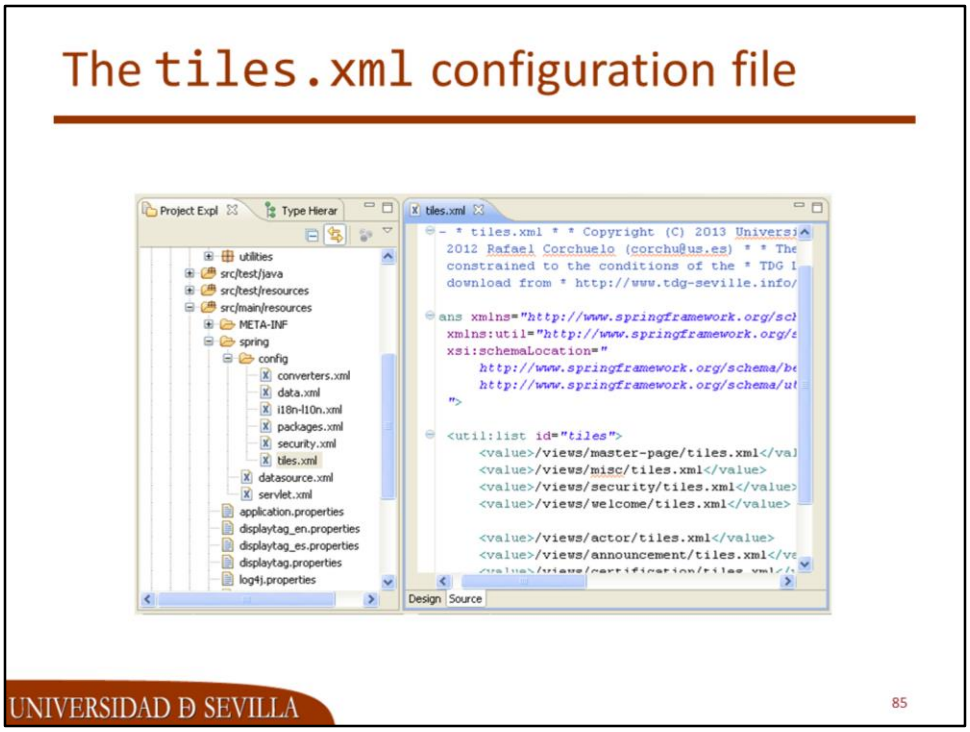


The “security” subfolder contains the files required to display a login screen: “login.jsp”, which defines the view, “messages_es.properties” and “messages.properties”, which are the i18n&l10n bundles, and “tiles_es.xml” and “tiles.xml”, which define the login web page building on the master page.

The welcome subfolder



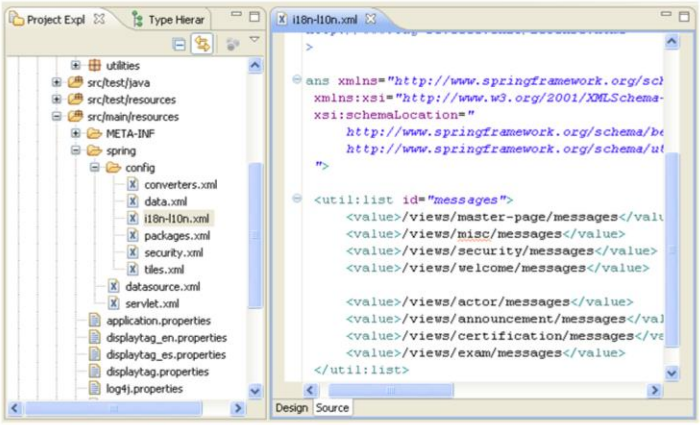
The last predefined subfolder is “welcome”, which contains the files required to display a welcome page in our web information system. It shouldn’t be difficult to you to guess the meaning of these files: “index.jsp”, which defines the view, “messages_es.properties” and “messages.properties”, which are the i18n&l10n bundles, and “tiles_es.xml” and “tiles.xml”, which define the welcome web page building on the master page.



Apart from the previous folders, you also need to know a couple of configuration files in the "src/main/resources/spring/config" folder. The first one is "tiles.xml". This file instructs Spring about where the Apache Tiles documents that define where the web pages of your application reside. Note that the template provides a list with the Apache Tiles document in the predefined subfolders of the "views" folder; you need to add a new value to the list for every new subfolder you create.

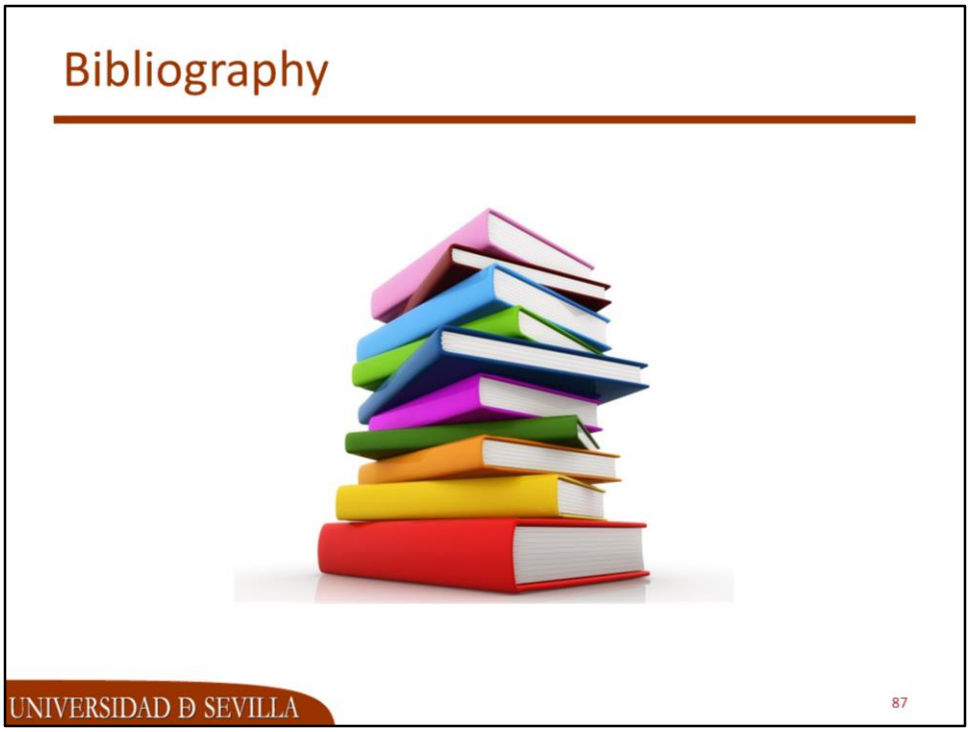
NOTE: please, note that you must provide the name of the English Tiles documents only, e.g., "/views/announcement/tiles.xml". Spring locates the Spanish or whatever other variant automatically; you mustn't list it in this file. If you list it, you'll get an error that is very difficult to understand, so, you're warned!

The i18n-l10n.xml config file



This configuration file's very similar to the previous one, but it provides information about your i18n&l10n bundles. By default, the project template provides a list with the i18n&l10n bundles in the predefined subfolders of the "views" folder; you need to add new entries to this list whenever you add new subfolders.

NOTE: please, read the values of the list carefully. Note that you must write "/views/announcement/messages", for instance, not "/views/announcement/messages.properties" or "/views/announcement/messages_es.properties". This is a common mistake. Unfortunately, the technology is inconsistent. In the previous configuration file, you had to provide the full name of the English Apache Tiles documents; in this configuration file you have to provide the prefixes only, without the extension or the ISO 369 code. Hey, that's technology! Whatcha think?



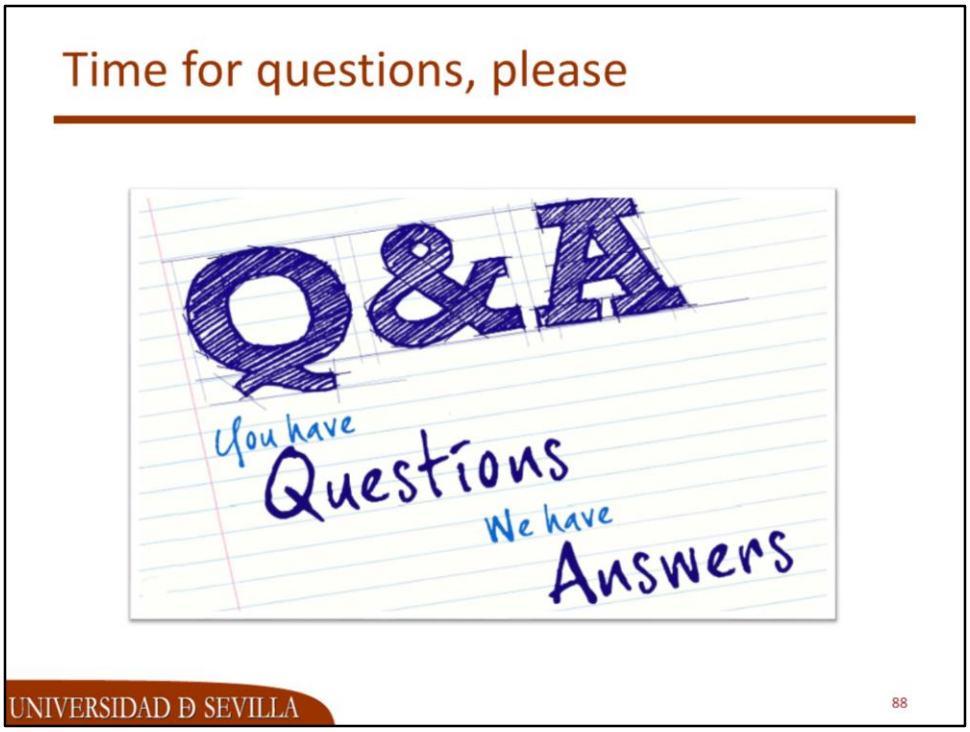
Should you need more information on the topics we've presented, please take a look at any of the following books

GUI bloopers 2.0: common user interface design don't and dos
Jeff Johnson
Morgan Kaufmann, 2008

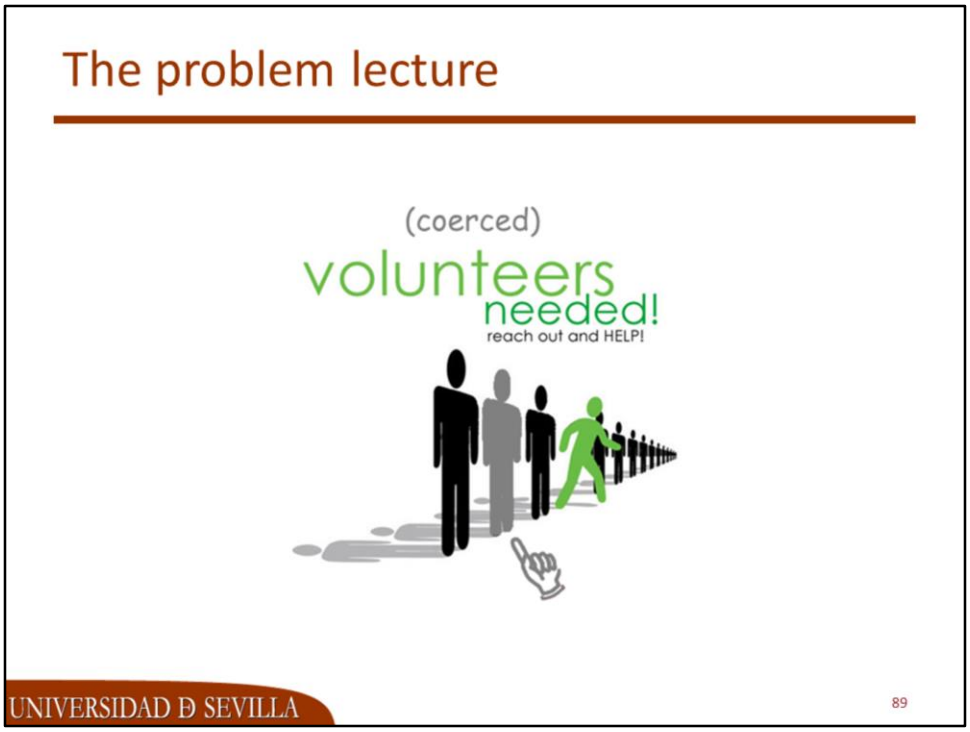
Pro Spring MVC: with Web Flow
Marten Deinum, Koen Serneels, Colin Yates, Seth Ladd, and Christophe Vanfleteren
Springer, 2012

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.

You might also be interested in the electronic documentation provided by Spring Source, the company that manufactures Spring. It's available at http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/htmlsingle/. The following JSP tutorial is also quite interesting: https://www.tutorialspoint.com/jsp/jsp_standard_tag_library.htm.



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.



Next week, we have a problem lecture and we need volunteers! Who's first in line?

Thanks for attending this lecture! See you soon.