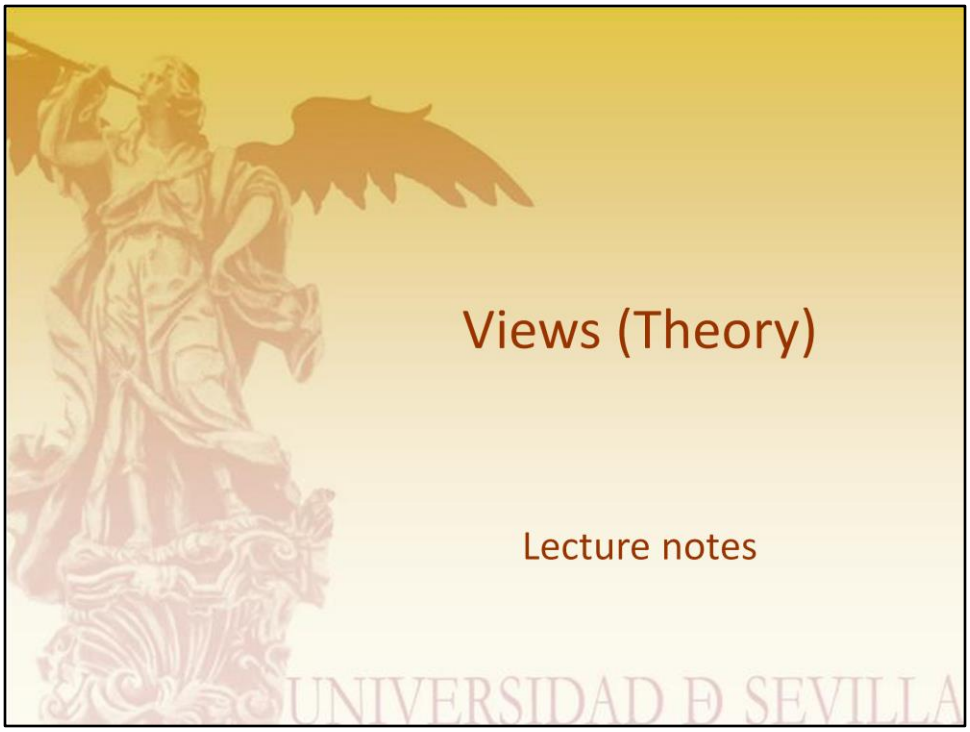


Welcome to this lecture! Our goal's to report on the theory that you need to create views.

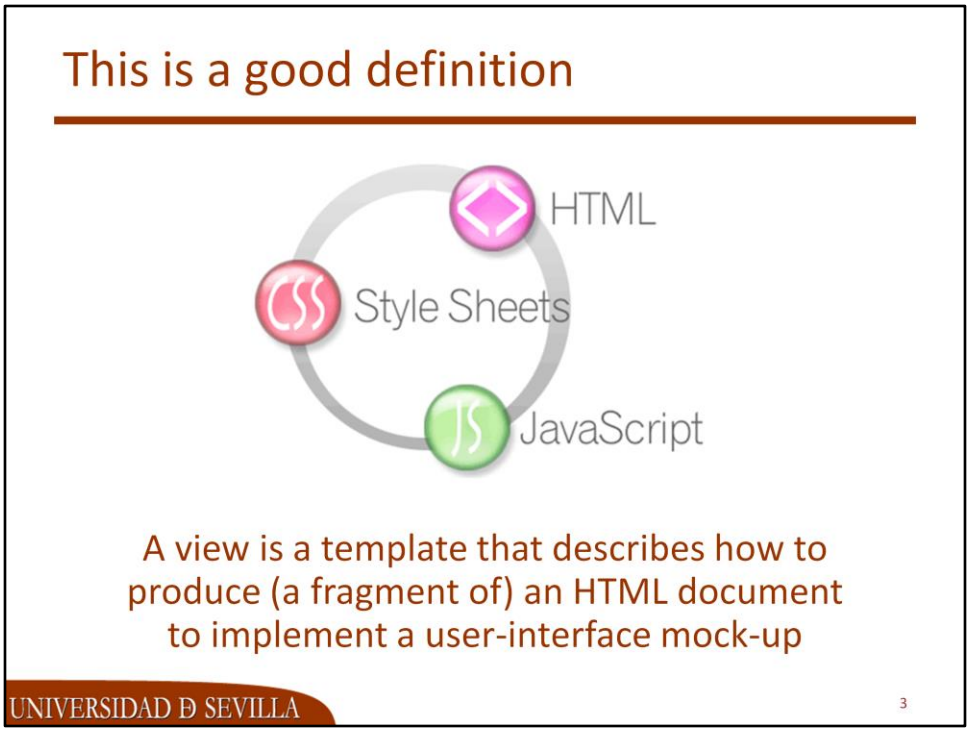
--

What are views?

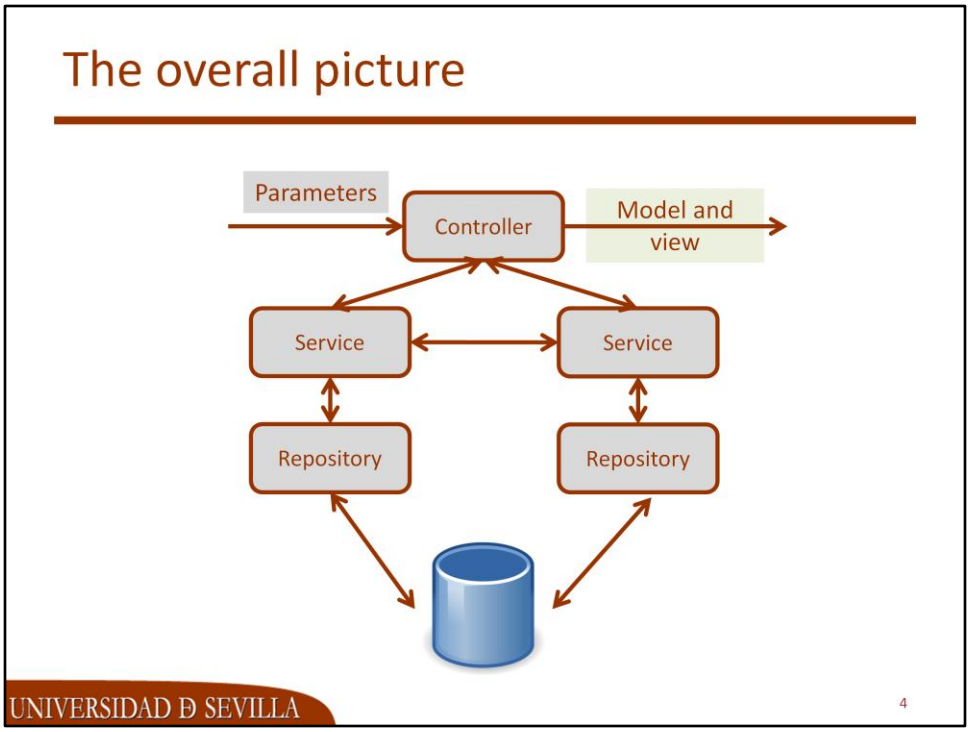
UNIVERSIDAD Ð SEVILLA

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 mock-up, or simply mock-up. 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 ECMAScript scripts (which endow HTML documents with algorithmic capabilities).



This slide provides an overall picture of a part of our architecture. Please, recall from lesson "L01 – Introduction" that sooner or later every request to a web information system is passed on to a controller, which gets the parameters of the request and is expected to return a model and a view name. In the previous lecture, we studied repositories and services, which support controllers; in this lesson, we are going to delve into the views; in the next lesson, we'll finally study controllers.

How are they devised?

UNIVERSIDAD Ð SEVILLA

5

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

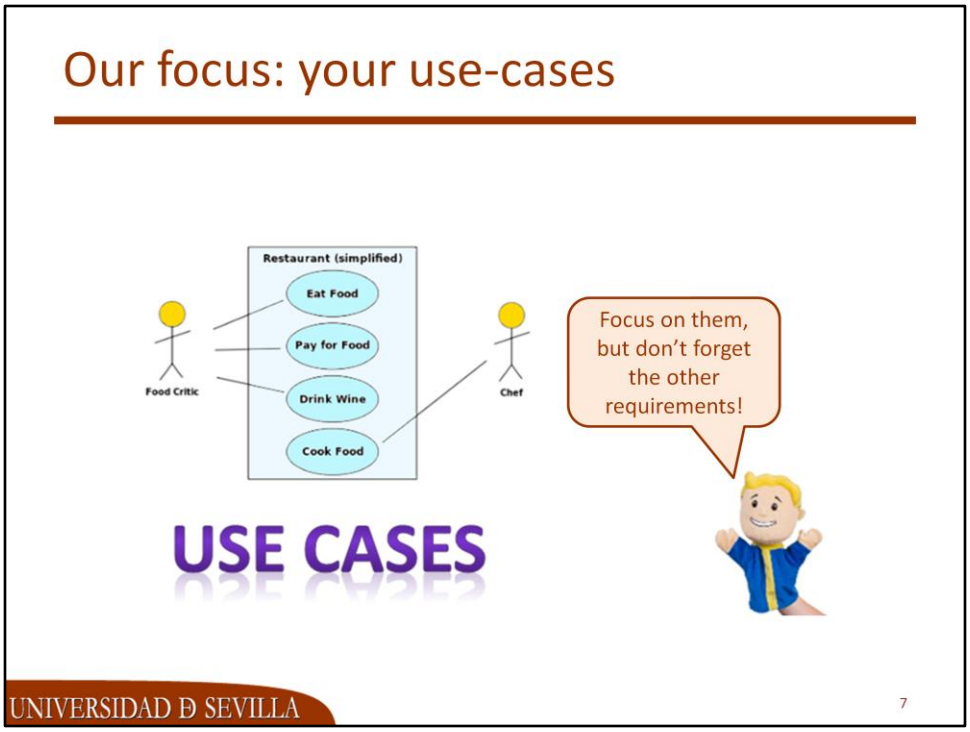
Starting point: your requirements

- 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

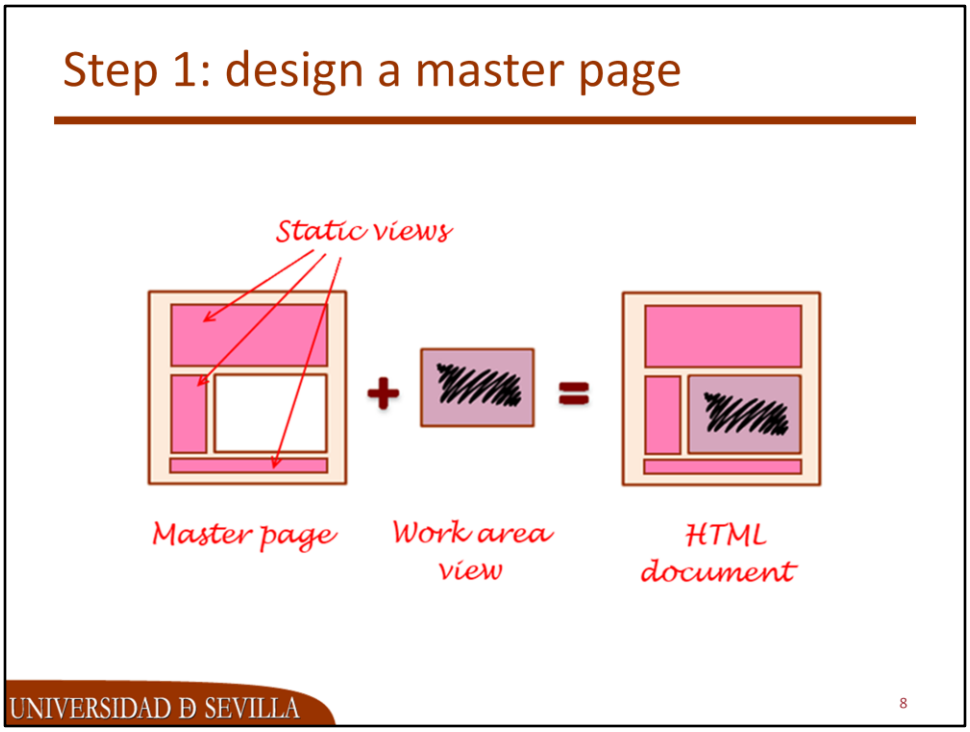
UNIVERSIDAD Ð SEVILLA

6

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

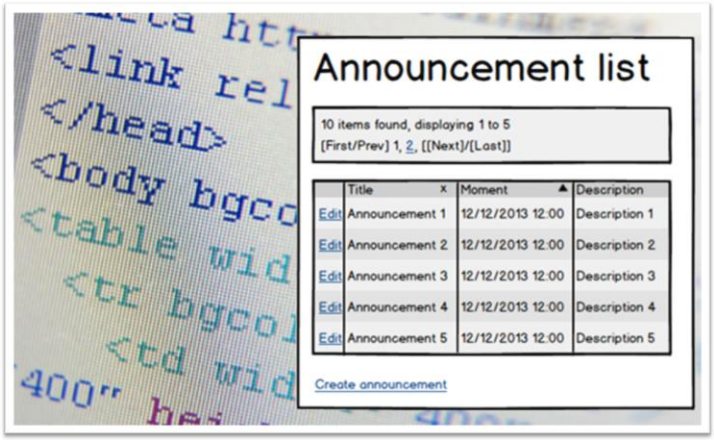


In particular, we'll focus on your use cases. But, please, remember that having a focus doesn't mean forgetting about the other sections. Don't forget about the illities and the information requirements since you need to have a holistic perspective of your requirements if you wish to produce good views.



The first step is to create a master page, which is a template that provides a common look and feel to the pages in our system. They provide logos, menus, copyright messages, and so on, which are provided by static views; we only need to plug a number of views into the master page to create the HTML documents that will be sent to your user's browser.

Step 2: implement your views



The second step is to implement your views, building on the mock-ups.

Step 3: 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!

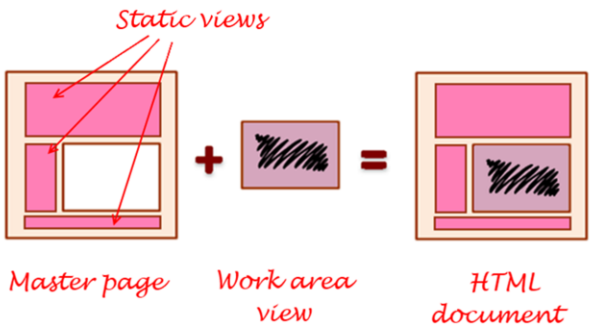
Unfortunately, there's not a fourth step regarding checking the views.

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, then we'll delve into implementing views, and finally into how to configure your project.

Let’s start with the master pages.

What's a master page?



It's a template that describes the parts that are common to every page and has holes into which the views are plugged in

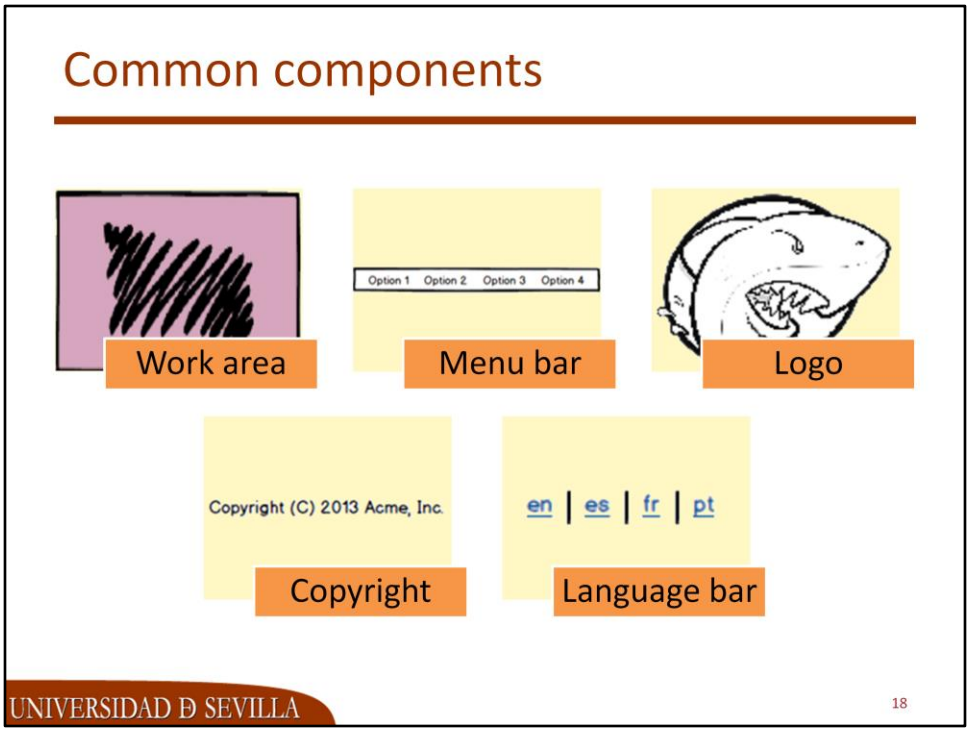
UNIVERSIDAD D SEVILLA

15

A master page is a template that describes the components that are common to every page in your web information systems; note that it must have holes into which more specific views are plugged in. Some views are static and help you provide a homogeneous look and feel to your web information systems; such views commonly describe the menu bars, the logo, the copyright message, the language bar, and other such common components of typical web user interfaces. The work area refers to the part of a page that is specific to an action in a use case. Note that views must be plugged into the master page to produce HTML documents.

In the following slides, we'll first explore a few common designs that are related to master pages and then we'll delve into the technology that we're going to use to implement them.

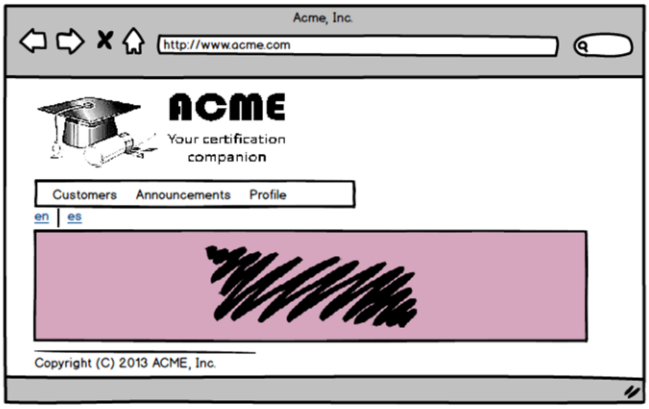
Let’s first delve into the common designs that are related to master pages.



In this slide, we show the most common components of a which a master page is composed, namely:

- A work area, aka body, which is the hole into which we'll plug views that depend on the action that the user is performing.
- A menu bar, which allows the user who is using the system to select the options that it offers.
- A logo, which identifies the company that owns the system.
- A copyright message, which is legal information.
- A language bar, which allows to change the language in which the system works.

A common master page layout

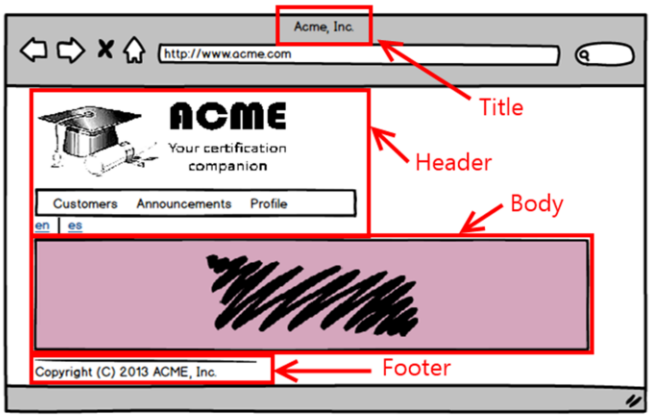


This slide shows how the previous components are typically laid out on a typical master page design. The work area is in the middle, the logo, the menu bar, and the language bar are at the top, and the copyright message is at the bottom. This is quite a simple layout, but effective enough for the purposes of this subject.

Let's now delve into the technology that we're going to use to implement master pages.

It is known as Apache tiles. It's a solid technology that has been around for years.

Apache Tiles in a nutshell



UNIVERSIDAD D SEVILLA

In a nutshell, Apache Tiles is used to specify how to combine the different components of which a web page is composed. Next, we're going to delve into the details.

The master page definition file

```
<?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 a master page configuration 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.

A regular page definition file

```
<?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>
```


Building on the master page, we can create regular configuration files like the one in this slide. In this example, we define two views, that is, two web pages that rely on our master page, namely:

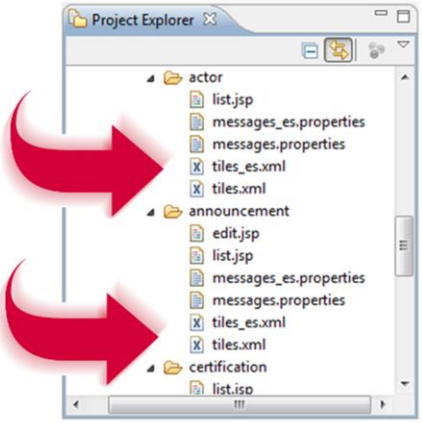
- "announcement/list": it extends the previous master page and sets the "title" and the "body" attributes to appropriate values. Note that Apache Tiles uses attribute "value" to provide the value of an attribute, independently from whether it is a piece of text, like in the case of the "title" attribute, 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.

NOTE: please, bear in mind that attribute "name" specifies a name. Although it might seem that "announcement/list" or "announcement/edit" are paths, they are not; they are just identifiers by means of which we'll refer to these views in our code.

What about i18n & l10n?

Wait a minute! What about i18n & l10n? Please, recall that i18n stands for internationalisation and that l10n stands for localisation. That is, adapting an application to the local culture of the person who is using it. Generally speaking, i18n & l10n amounts to displaying the messages of our application in different languages, e.g., English and Spanish, or variations of those languages, e.g., Peninsular Spanish, Argentinian Spanish, British English, or American English. The problem's that if you take a look at the slides in which we've presented some Apache Tiles documents before, the messages were written in English, e.g., "Announcement list" and "Edit announcement".

The key, the name of your tile files

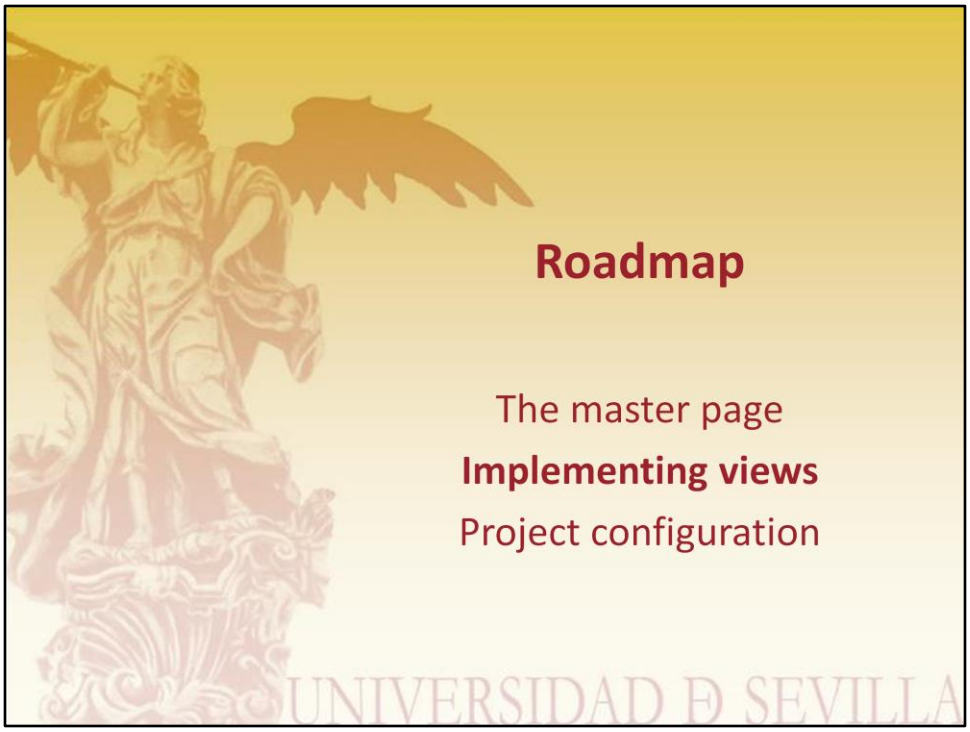


It all depends on the name of the Apache Tiles documents. By convention, all tiles documents are called “tiles.xml”. It’s a common mistake that you call your tiles document “master-page.tiles”, “announcement-list.xml”, “announcement-edit.tiles”, or something like that. By default, a tiles document whose name’s “tiles.xml” is assumed to be in English; if you whish to produce a version that is localised to, say, Spanish then you must produce another file with name “tiles_es.xml” in which the attributes are in Spanish; similarly, the name of the file must be “tiles_fr.xml” if you’re localising your file to French, “tiles_it.xml” if you prefer Italian, or “tiles_pt.xml” if you prefer Portuguese, and so on. The suffix you must use 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.

An i18d & l10d definition file

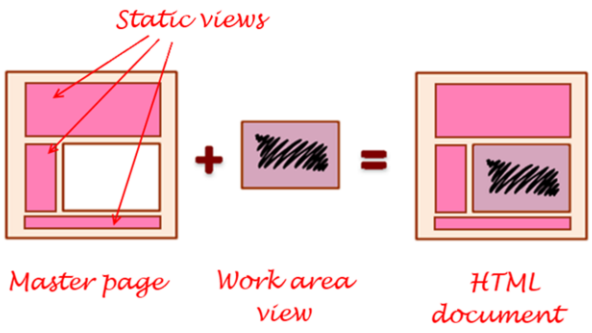
```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE tiles-definitions PUBLIC
       "-//Apache Software Foundation//DTD Tiles Configuration 2.0//EN"
       "http://tiles.apache.org/dtds/tiles-config_2_0.dtd">

<tiles-definitions>

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

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

</tiles-definitions>
```


Here you have an example of an internationalised and localised definition file; note that we only need to change the values of the attributes. Yes, we agree with you: you have to duplicate a whole definition file just to change a couple of attributes. That's true, it's boring and error prone, but this is how technology works and you must learn it.



The theory regarding master pages was easy, wasn't it? Let's now delve into how to implement views, which you know are plugged into master pages to create actual HTML documents.

What's a view?

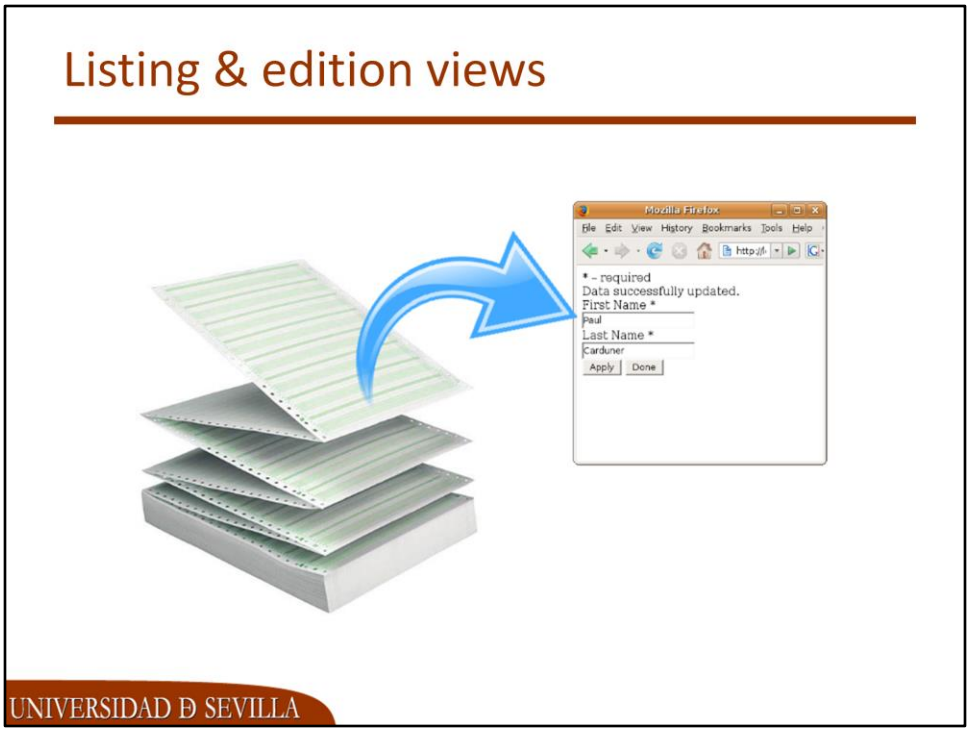


A view is a specification of the contents of a component of a master page

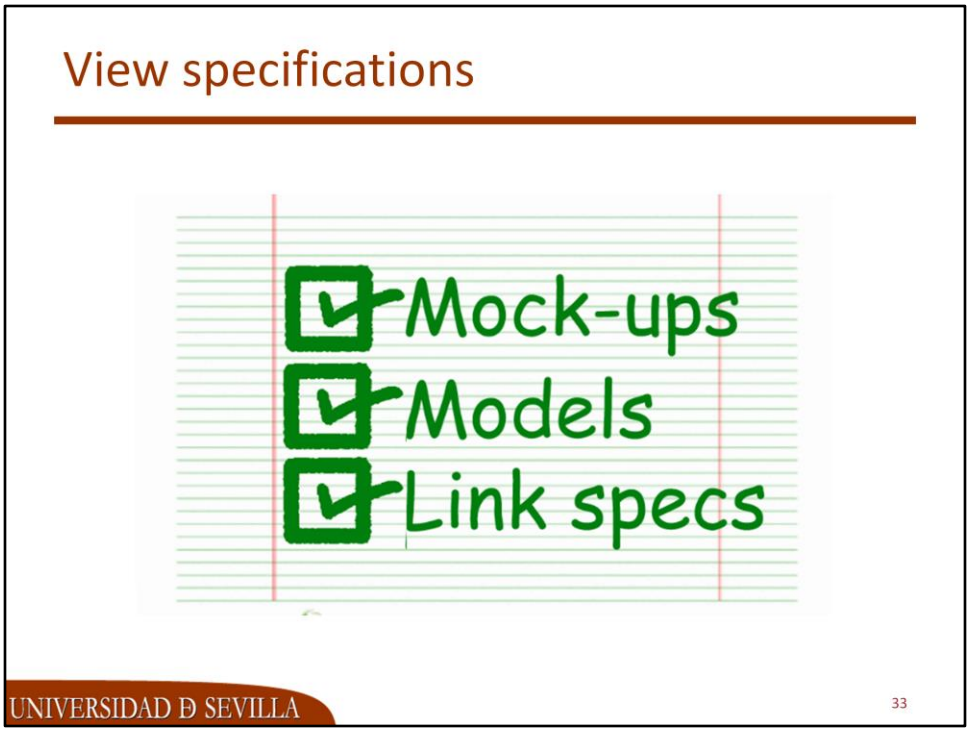
As usual, we start with a question and a definition. What's a view? It's a specification of the contents of a component of a master page. This is quite a simple definition, but very often professional software engineers also use term view to refer to the resulting HTML document, as well; you just must get accustomed to this terminology and avoid confusions.

We'll first explore a few common designs and then will delve into the technology used to implement them.

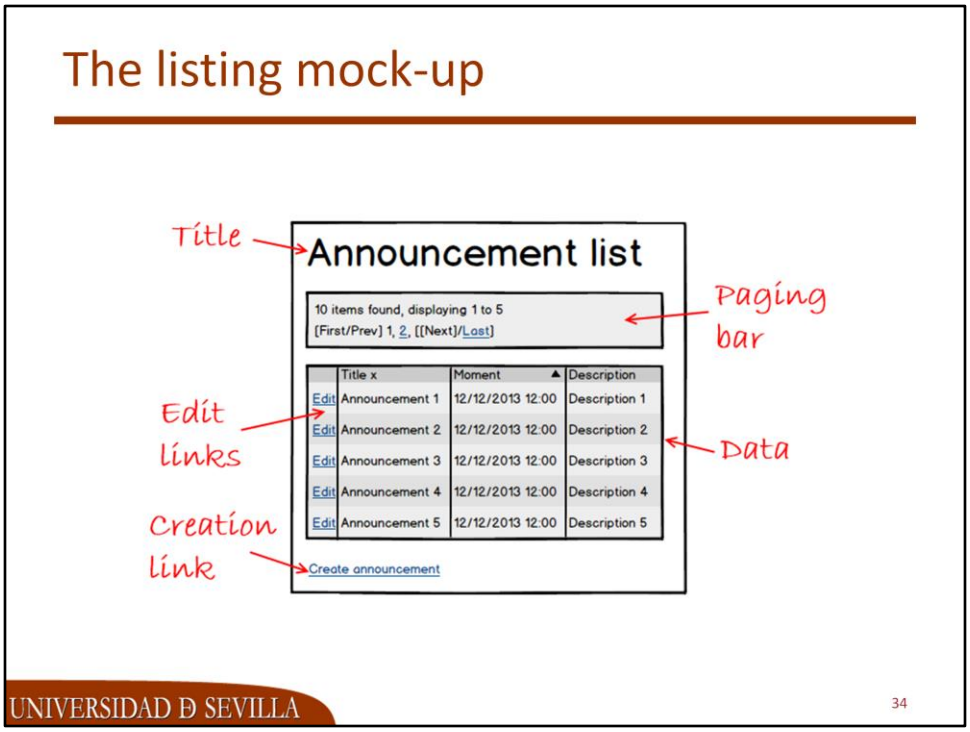
Let’s start with the common designs.



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. In the following slides, we'll present the designs that we're going to use in this subject.



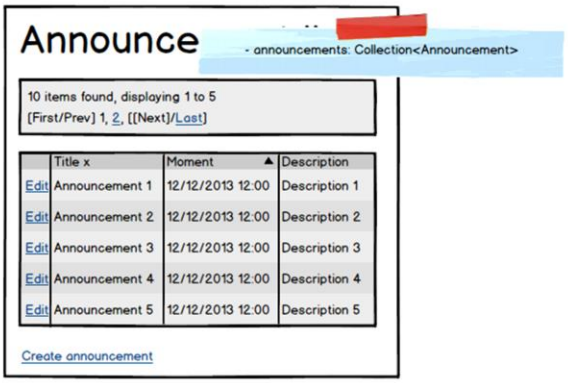
To fully specify a view, so that a programmer can start implementing it, you need to have a mock-up, a model, and a link specification. Very likely, you'll get the mock-ups from your requirements engineer, but it's you who must work on the models and the link specification. Please, keep reading to learn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



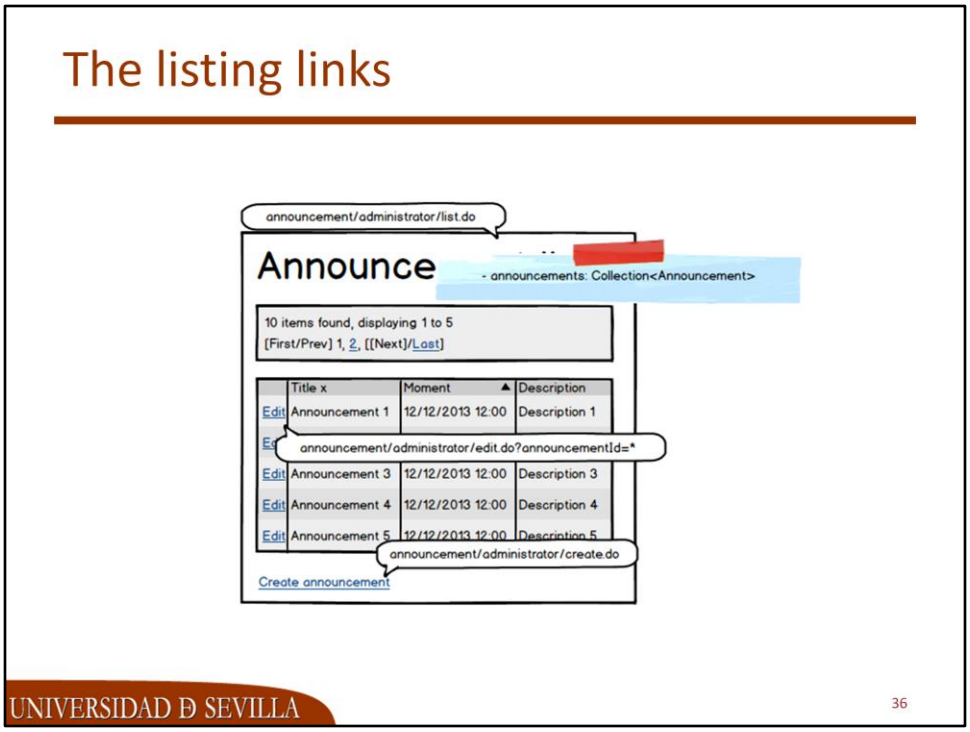
This slide shows the design of the listing mock-up that we're going to use. Its components are the following:

- 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 more often than not presented in tabular format.
- The edit links, which allow to edit individual entities.
- The creation link, which allows to create a new entity.

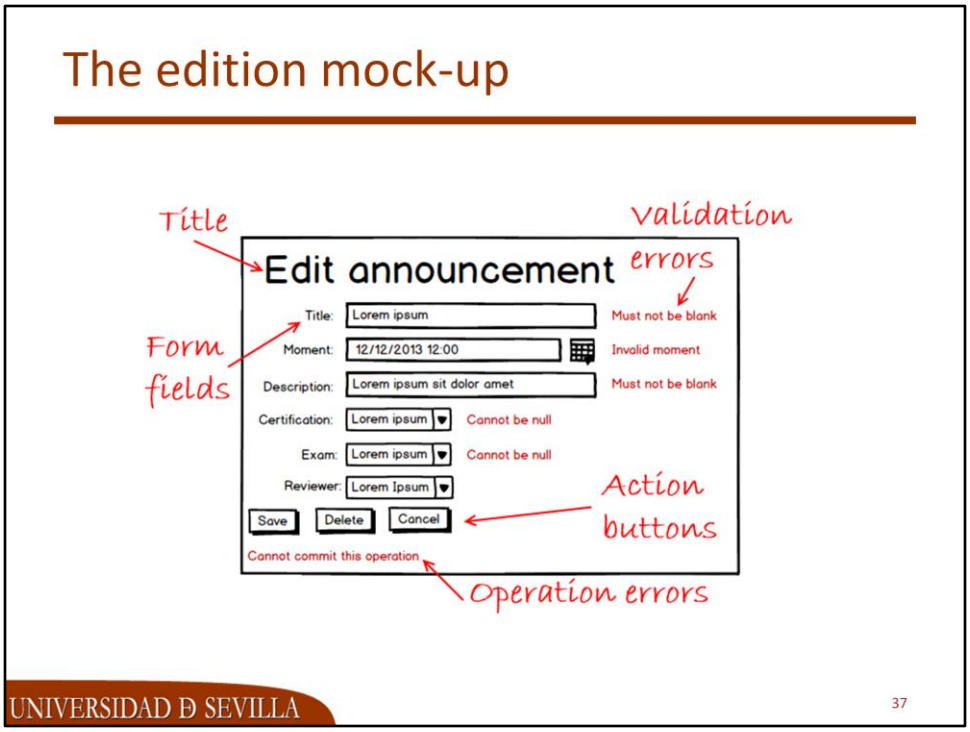
The listing model



You may wonder where the data that is shown comes from. It's easy, it comes from a model. Please, recall from the introduction to this lecture 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.



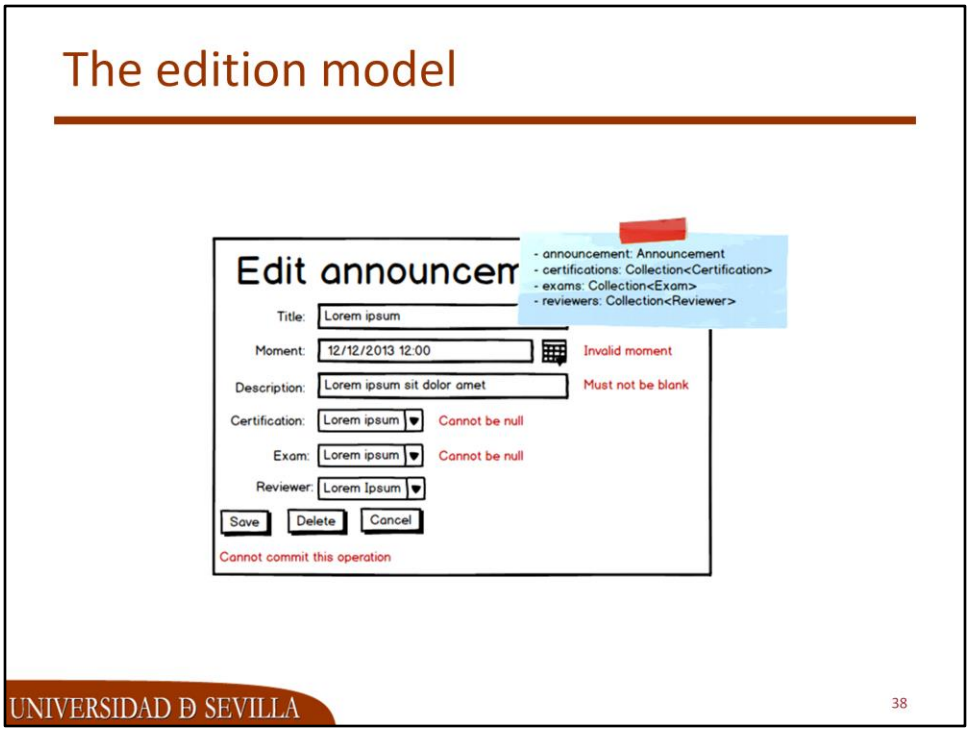
Finally, we need to specify the URLs of the links in the mock-ups. This slide shows the links in the listing mock-up. Please, note that they all have the same structure: first comes the entity name, then the authority, then an action name with a ".do" suffix, and, optionally, some parameters. The values 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.



This slide shows the edition mock-up we've designed to edit an announcement. These mock-ups are also known as forms. Its components are the following:

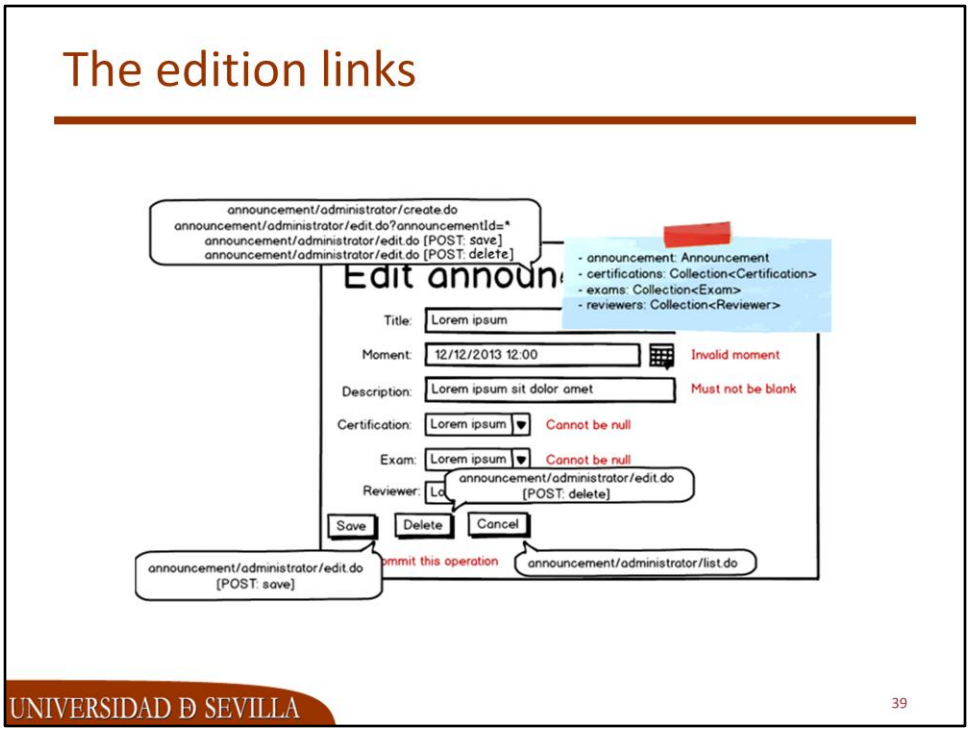
- A title, which provides a description of the entity being edited.
- Form fields, which have a label that describes the field and an input box.
- Validation errors, which are shown only when the data entered is invalid according to the constraints in the domain model.
- Action buttons, which allow the 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.

NOTE: the master page that we provide in our project template can handle a model variable called "message"; if this variable is set to a string, then its contents are shown as an error message below your forms.

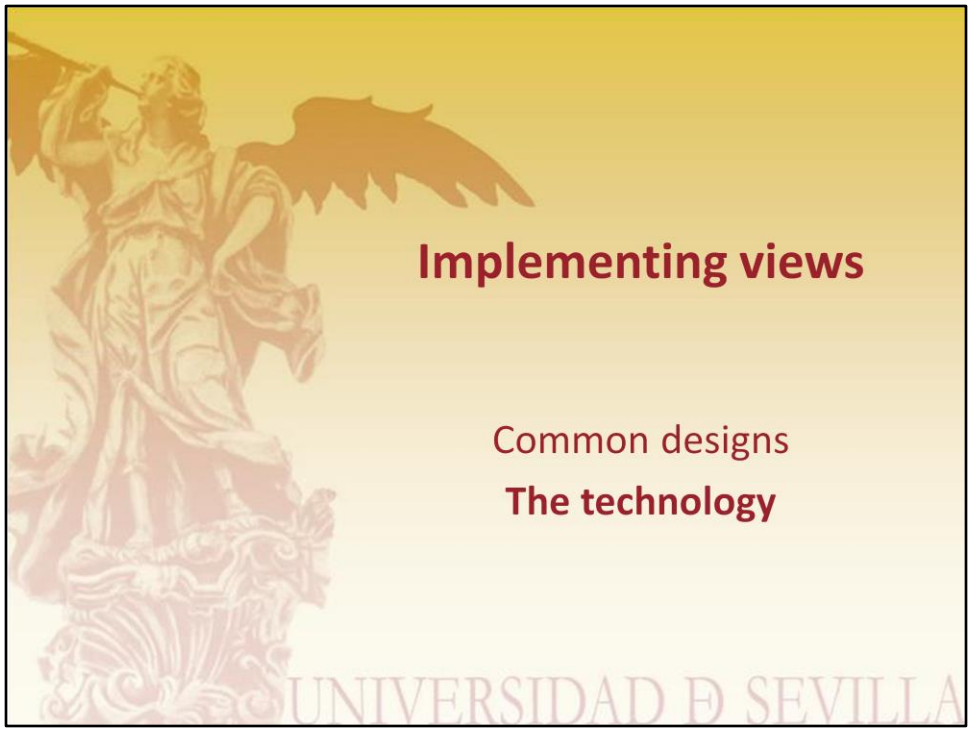


As usual, we'll describe the model with a sticky cyan note. In this case, the model consists of the following variables:

- "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.



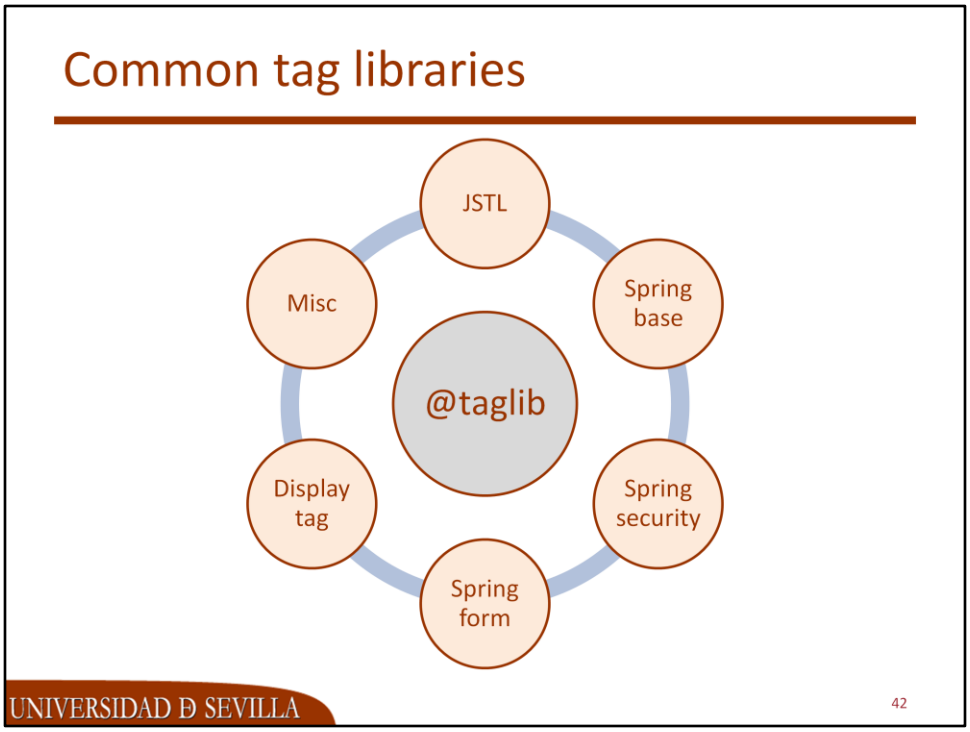
Finally, we need to describe the links in the mock-up. The edition mock-up is a little more involved than the listing mock-up since it can be requested by means of GET and POST requests. In the former case, the user has clicked on a link and is requesting to edit a given announcement, so the URL includes 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.



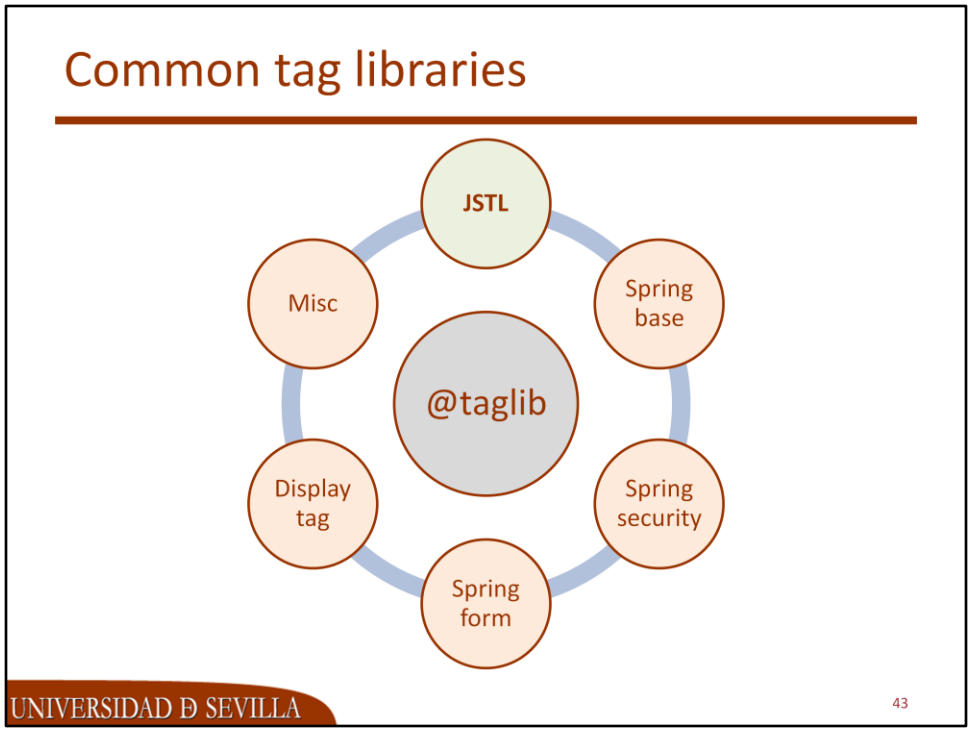
It’s now time to delve into the technology that we’re going to use to implement views.

This is our technology

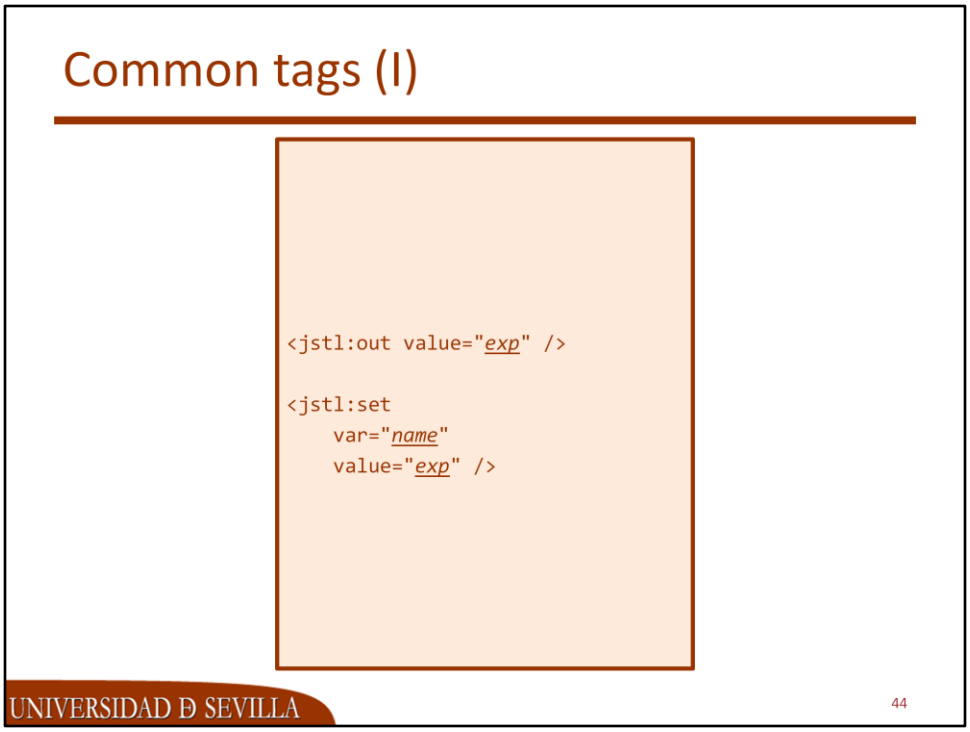
It's JSP, which stands for Java Server Pages. It's also quite a solid technology that has also been around for years.



Now, we'll spend a little time at providing a foundation on the most common tag libraries.



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.

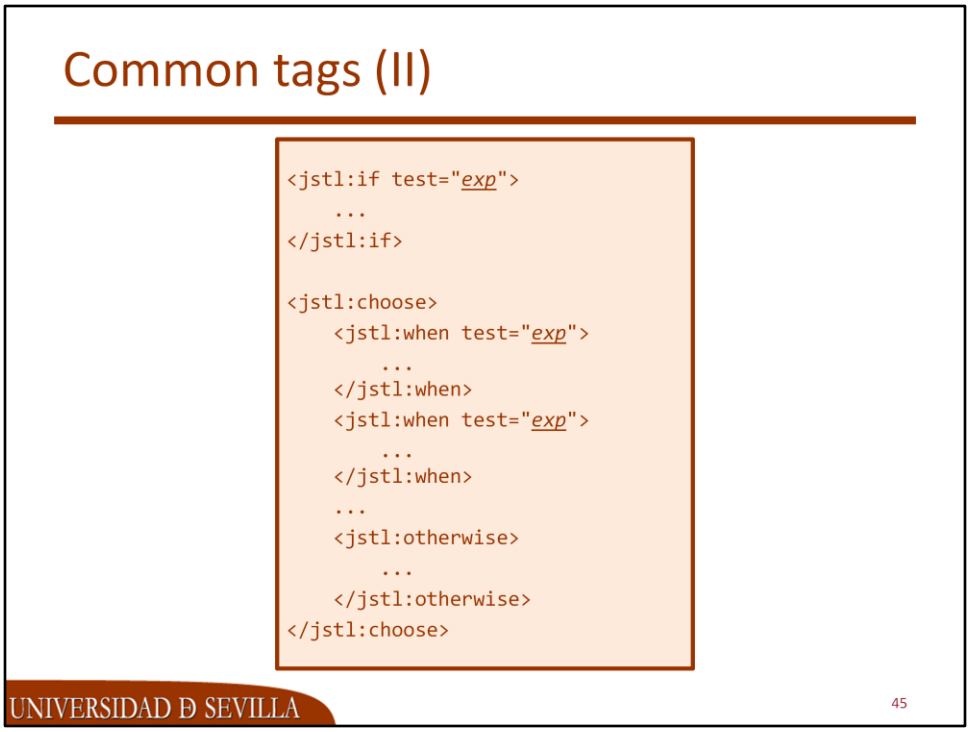


The JSTL tag library provides processing instructions by means of which you can implement very simple algorithms. In this slide and the following ones, we'll provide an overall view of the tags available. In the explanations, we use "name" to refer to a variable name like "i" or "counter" and "exp" to 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.

The first group of tags includes "out" and "set":

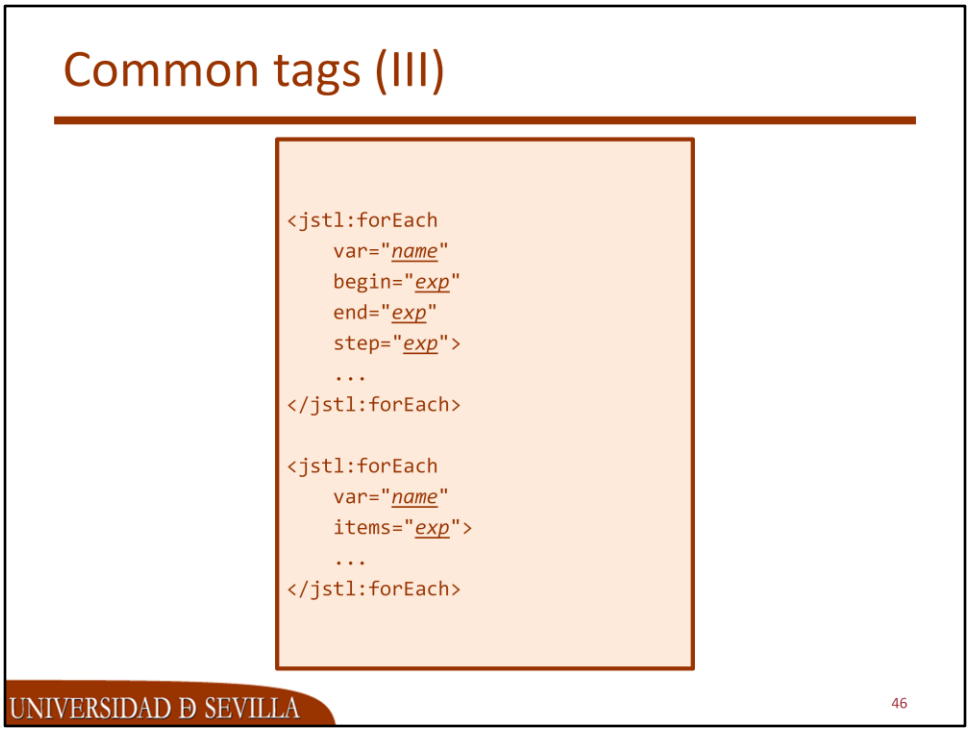
- "out": outputs the result of evaluating "exp". Note that instead of this tag, you can simply write "exp", but the result's not the same: "out" cares of translating characters like "<" or "&" into their HTML equivalent entities, namely, "<" and "&"; so, you should always use "out" unless you're absolutely sure that your expression doesn't result in any special characters.
- "set": sets variable "name" to the result of evaluating "exp". Note that the variable is stored in the model, which implies that you might accidentally override a variable and this would result in a very difficult to debug error. Use this instruction very carefully.

NOTE: in the sequel, we write the attributes that you have to specify in slanted, underlined font; we write the fixed values of some attributes in normal font. See later some examples.



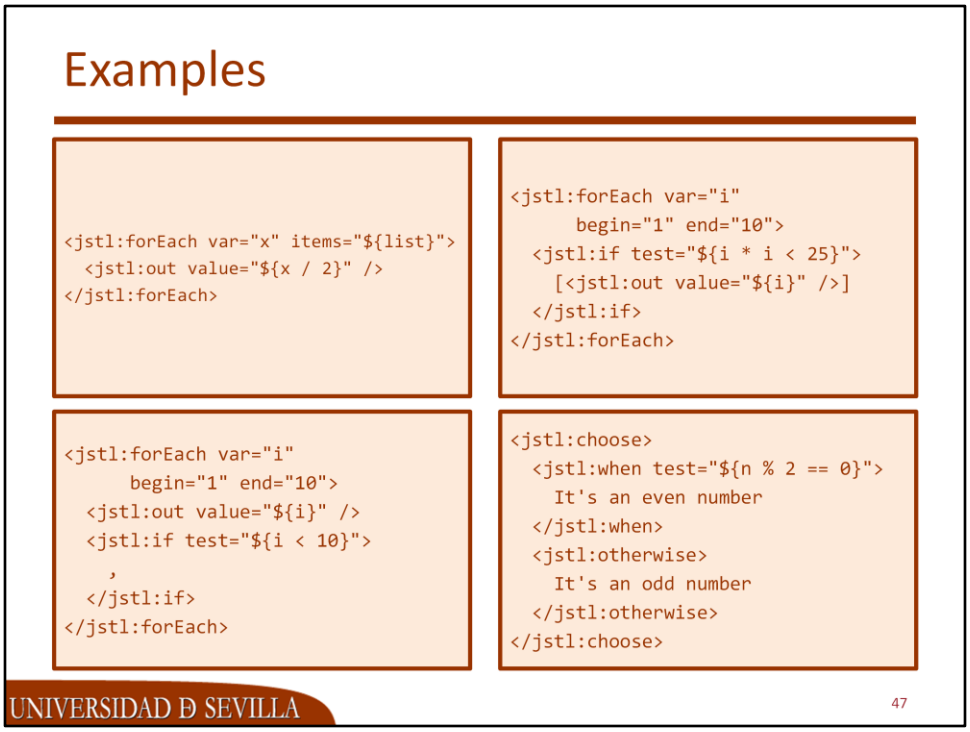
The second group of tags includes “if” and “choose”:

- “if”: if “exp” evaluates to “true”, then this tag outputs 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 “exp” 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.



The third group includes the “forEach” tag, which creates an iterator called “name” and outputs its body repeteadly, assigning a different value to the iterator in each iteration. This tag has two variants, namely:

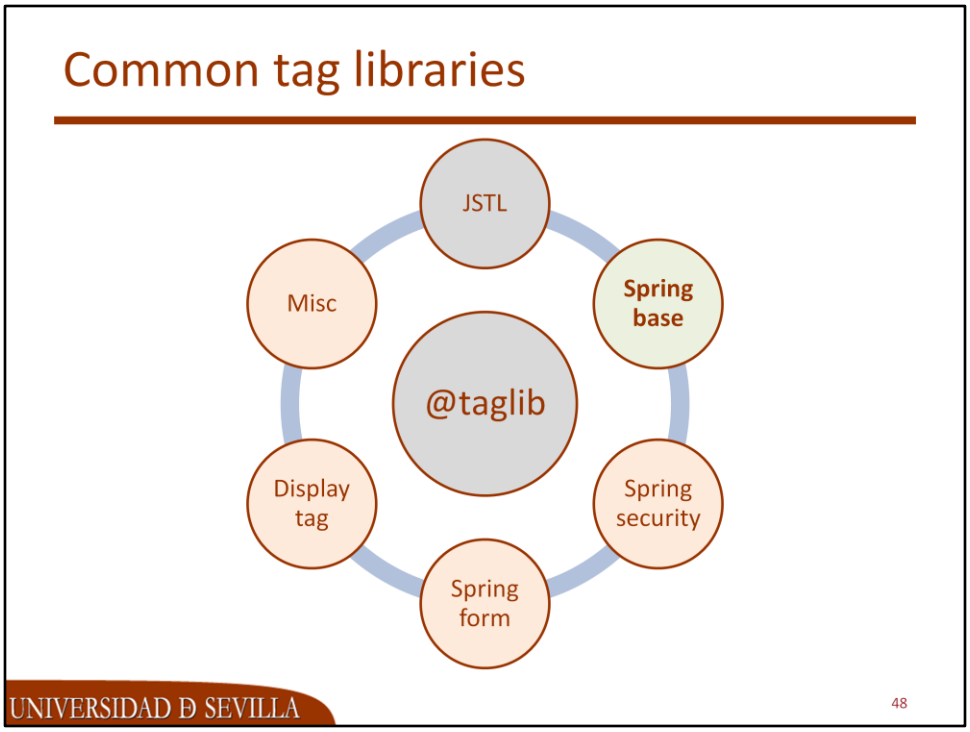
- In its simplest form, the variable iterates from the integer value indicated by attribute “begin” until the integer value indicated by attribute “end”, according to the increment indicated by attribute “step”; if no step is indicated, then “1” is the default. (Please, recall that you can write “1” or “${1}” since this is a literal.)
- In its more advanced form, it iterates over the elements of a collection that is indicated by means of attribute “items”.



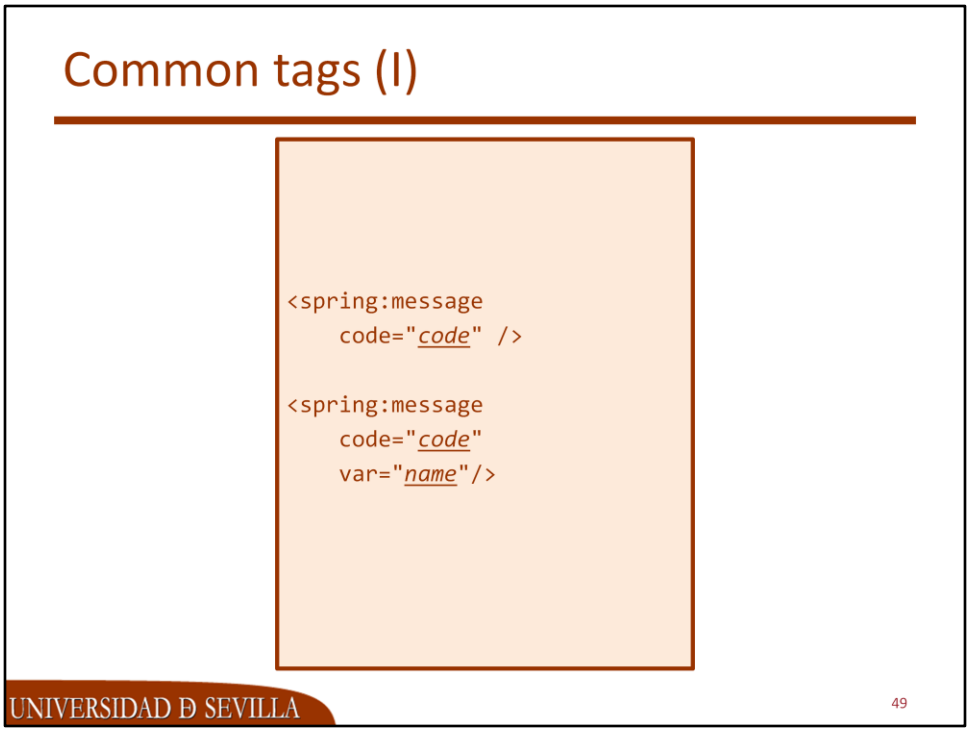
This slide shows a few examples. From left to right, from top to bottom:

- 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 number; 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.



This tag library provides only a couple of interesting tags. The first one is “message”, which will definitely help you internationalise and localise your JSP documents. It has a mandatory attribute called “code” that refers to a code in an i18n&l10n bundle; that code is assumed to be associated with a message in a given language. The tag has two variants, namely:

- The first one outputs the message immediately.
- The second one stores it in the variable indicated by attribute “var”. No output is produced. (Please, recall that you must be very careful when assigning a value to a variable; if there’s already a variable in the model with the same name, you may easily get in trouble.)

An i18n&l10n bundle is a text file like the ones in this slide. The one on the left's an excerpt of an English bundle and the one on the right's an excerpt of a Spanish bundle. Note that the bundle consists of a list of lines, each of which is of the form "code = text". The name of the file matters: it must be "messages.properties" or "messages_XX.properties", where "XX" denotes an ISO 639 language code, e.g., "en" is English, "es" is Spanish, "fr" is French, "it" is Italian, "pt" is Portuguese, or "se" is Swedish. If no code is specified, then "en" is the default. There's more information on these codes available at http://en.wikipedia.org/wiki/ISO_639, for instance.

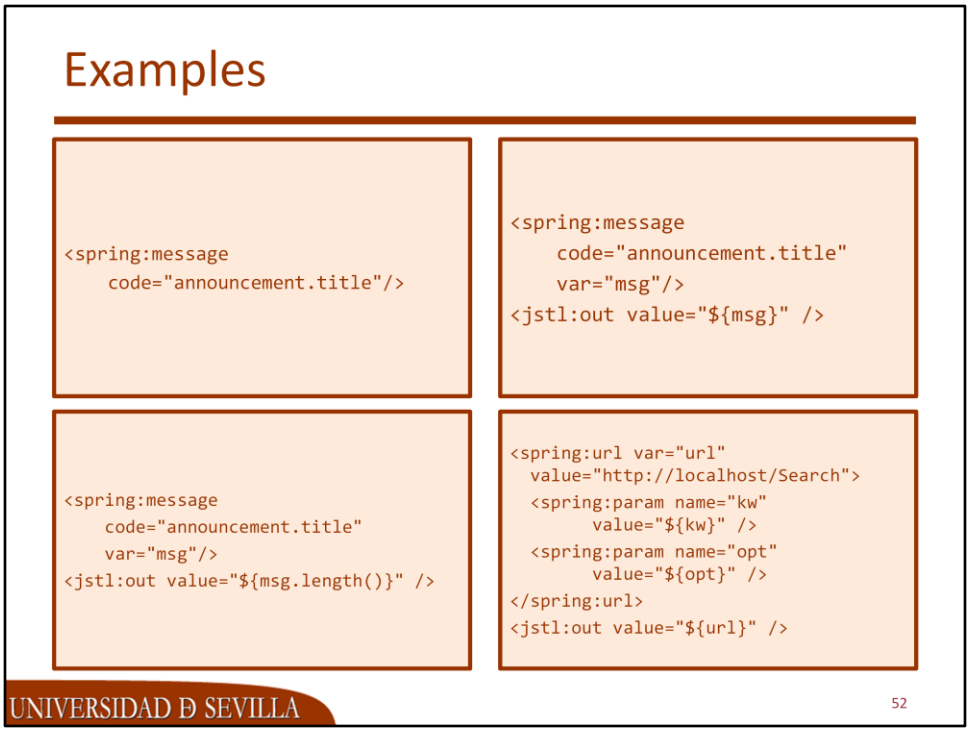
NOTE: Due to space limitations, we use the "⇲" symbol to indicate that a line continues. In general, the text that corresponds to each code's written in a single line; if you require to break a piece of text into two physical lines, then write "\⇙" and continue on the following line, where "\⇙" denotes a back slash followed by a carriage return. For instance, the following defines a code with four lines:

```
long.text=Somewhere in La Mancha, in a place whose name I do not care \⇙
    to remember, a gentleman lived not long ago, one of those who \⇙
    has a lance and ancient shield on a shelf and keeps a skinny nag and \⇙
    a greyhound for racing.
```

Common tags (II)

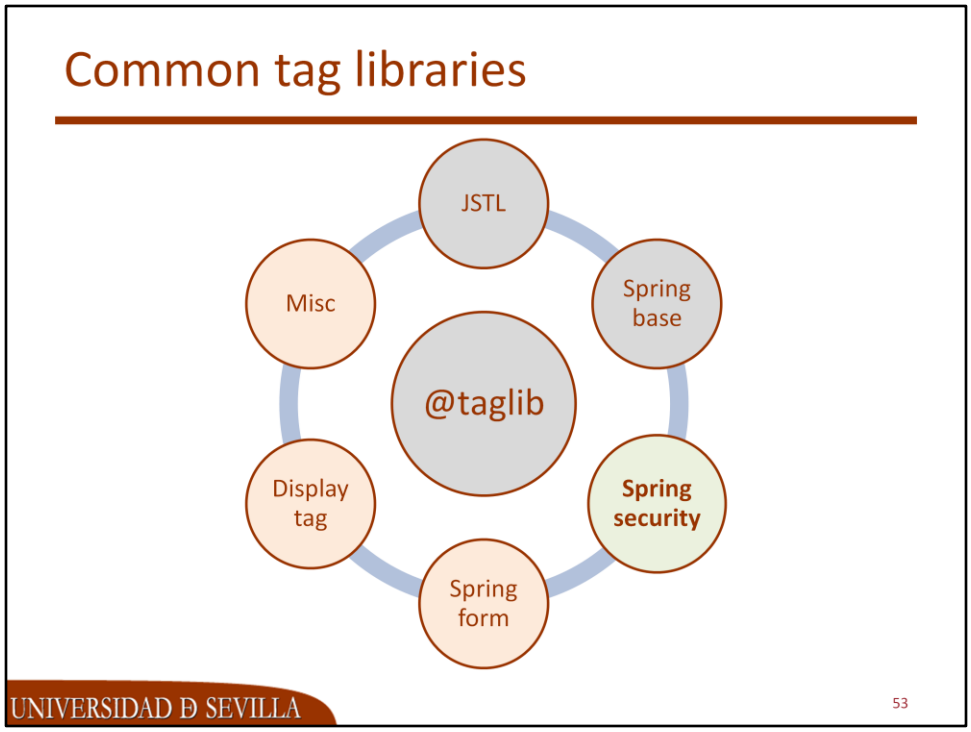
```
<spring:url
    var="name"
    value="url">
    <spring:param
        name="name"
        value="exp"/>
    ...
    <spring:param
        name="name"
        value="exp"/>
</spring:url>
```


The other interesting tag in the Spring base tag library is “url”. This tag constructs a URL and stores the result in the variable specified by attribute “var”; the parameters to the URL are specified by means of “param” tags.

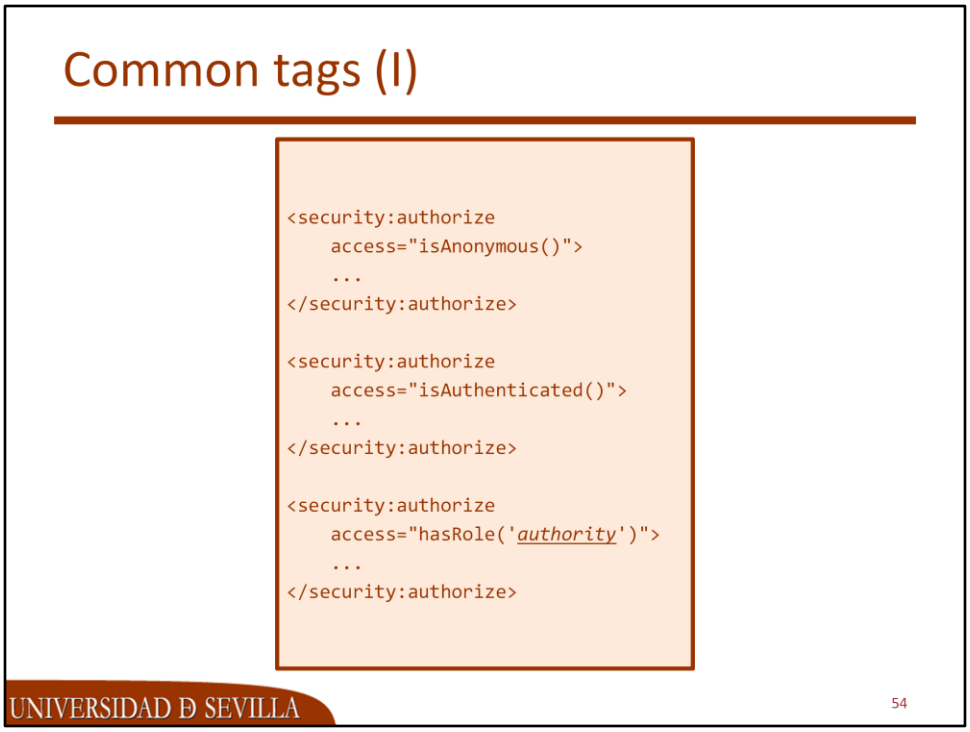


This slide shows a few examples. From left to right, from top to bottom:

- 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.

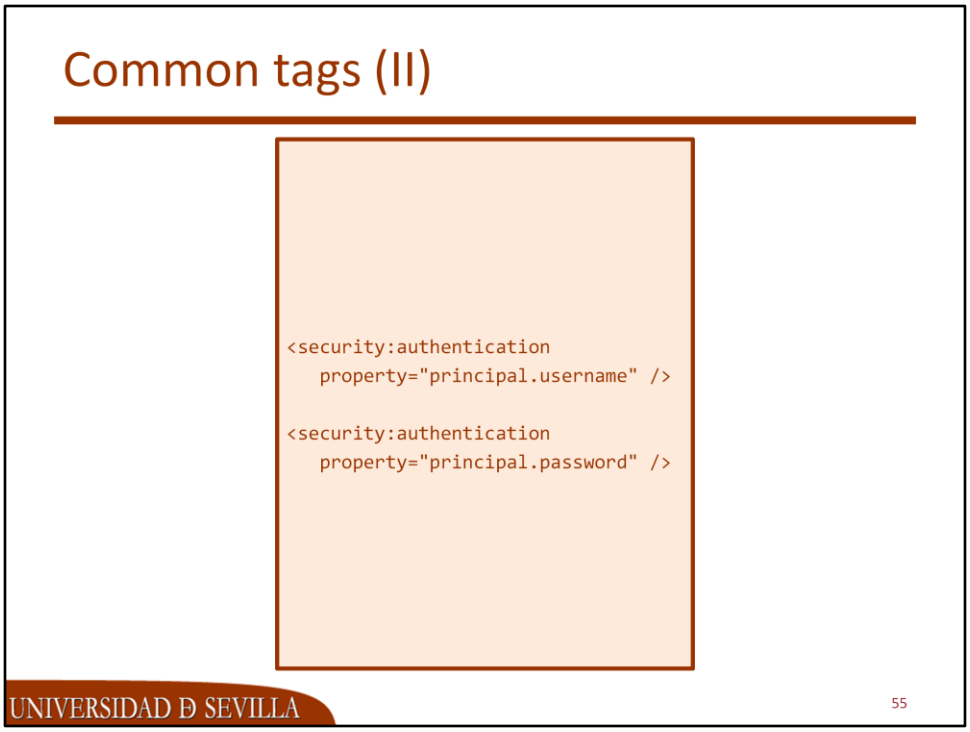


It's now time to explore the Spring security tag library.



The Spring security tag library provides a couple of tags. One of them is "authorize", which has a mandatory attribute called "access" in which you can write expressions of the following form:

- "isAnonymous()": this outputs the body of the tag only if the principal isn't authenticated. Please, recall that "principal" refers to the person for whom this code is being executed.
- "isAuthenticated()": this outputs the body of the tag only if the principal's authenticated, independently from the authorities he or she has.
- "hasRole('authority')": this interprets the body of the tag only if the principal's authenticated and his or her user account has the indicated authority, e.g., "ADMIN" or "CUSTOMER". Please, remember that an authority refers to a role. Spring's a little incoherent here since you can't write "hasAuthority", but "hasRole". It's technology!



The other interesting tag in this library is “authentication”. It’s used to retrieve information about the user account of the principal, as long as he or she’s authenticated. This tag has a mandatory attribute called “property” by means of which you can specify the information you wish to retrieve from the user account, namely:

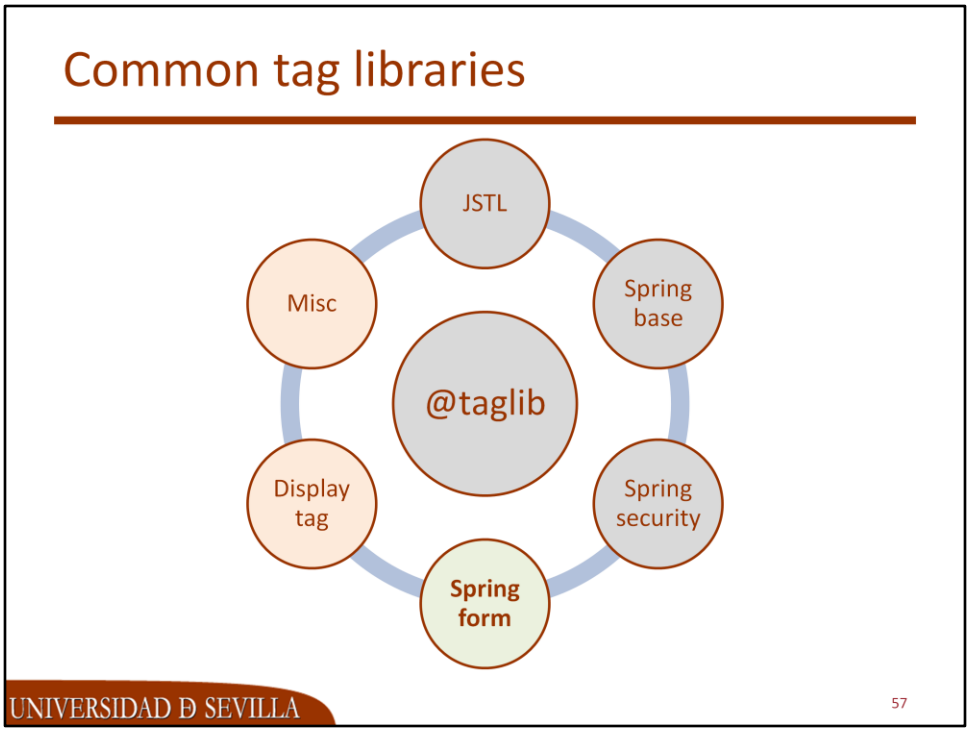
- “principal.username” to retrieve the username.
- “principal.password” to retrieve the hash of the password. (Please, recall that you should never store passwords in a database, but their hashes.)

Unfortuntely, no other choices are available. Note that your application may store additional information, e.g., an email or a phone number, but this information isn’t stored in the user account, but in application-specific entities to which the security system doesn’t have access.

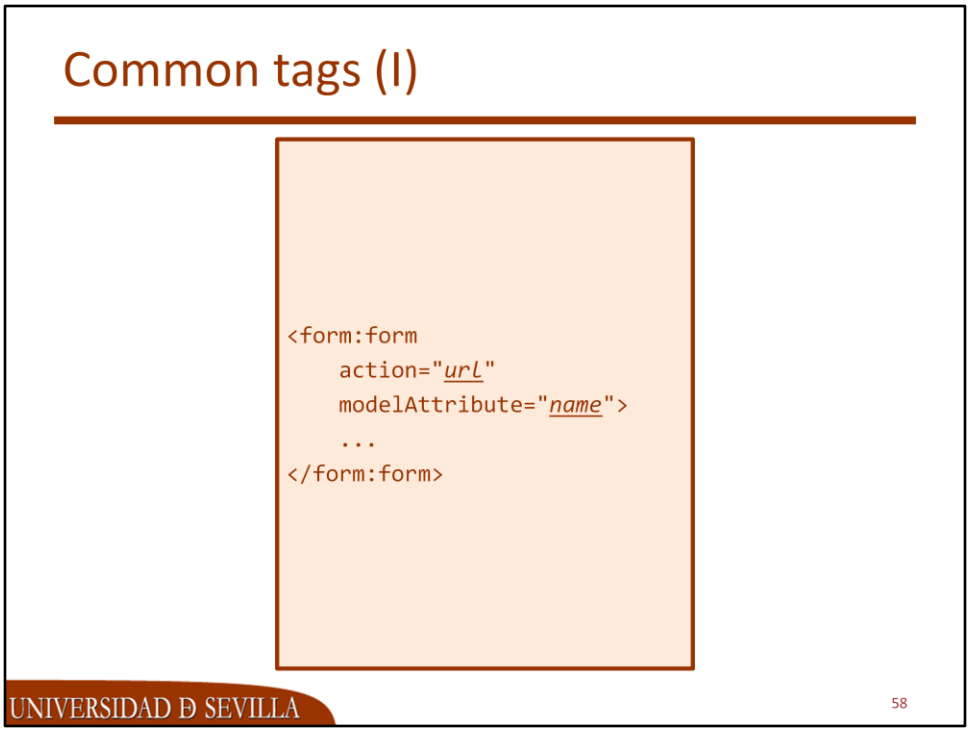
Examples

```
<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.



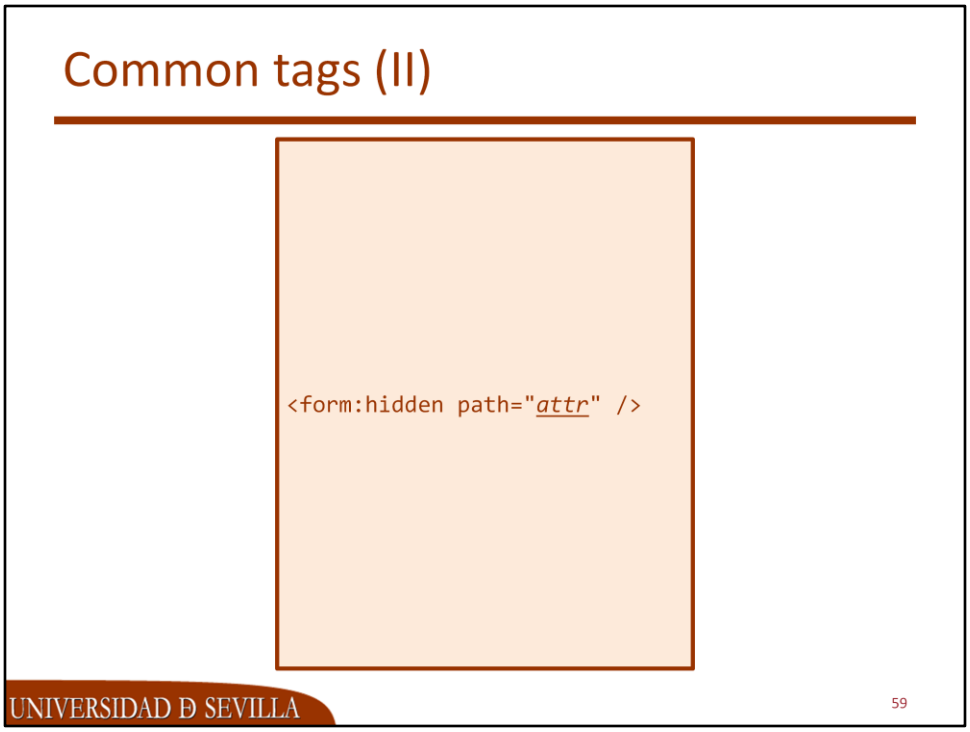
Let's now analyse the Spring form tag library. This is more complex than the previous ones, but quintessential to implementing typical use cases.



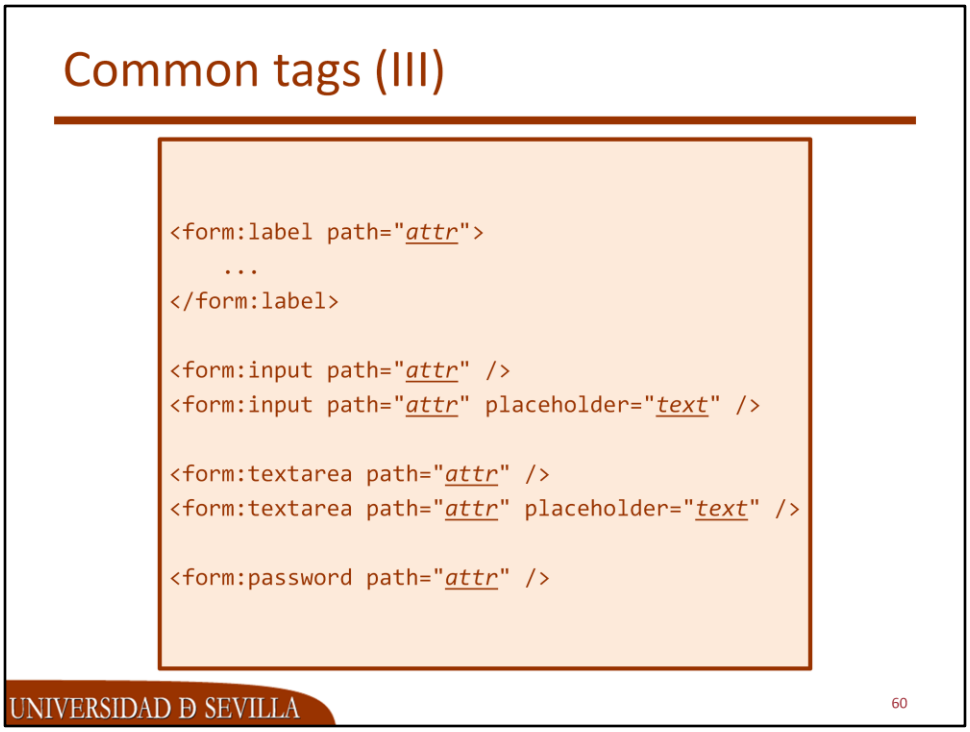
The Spring form tag library provides a number of tags to work with HTML forms, which are the primary HTML mechanism to edit domain entities. The most important one is “form”, which represents an HTML form. It has two mandatory attributes, namely:

- “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.)
- “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.

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



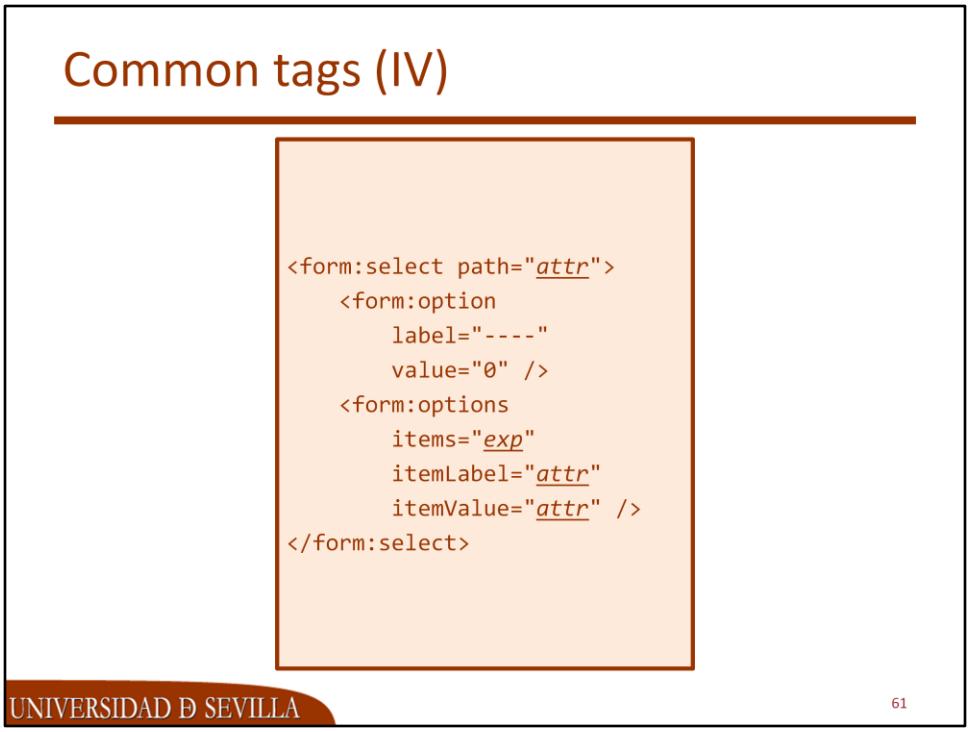
The first one goes unnoticed to the users since it doesn't produce any outputs, but it's extremely important. It's "hidden", which is a tag that holds the value of an attribute of the domain entity being edited, but doesn't display it. (The name of the entity's attribute is indicated by means of attribute "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!



The next group of tags includes “label”, “input”, and “password”, which provide a foundation to edit simple attributes like a title, a name, a moment, or a password.

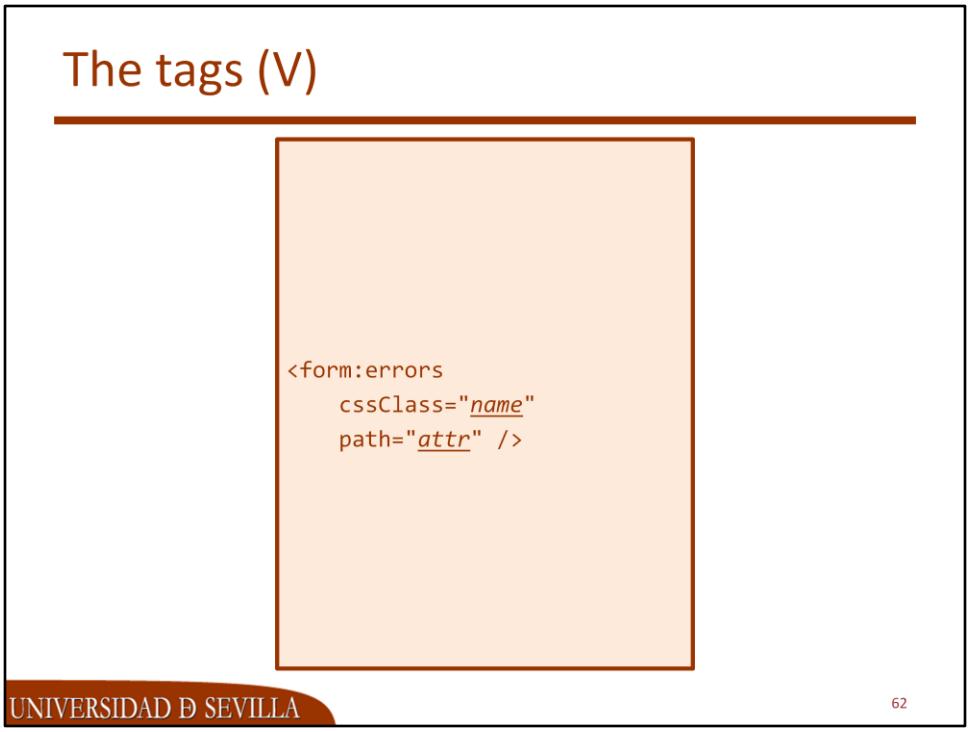
- “label”: this tag introduces a label for a textbox or a dropdown list. The label’s usually introduced by means of a “spring:message” tag, so that it can be easily localised to several languages.
- “input”: this tag introduces a single-line textbox in which the user can key in the value of an attribute.
- “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” and the “textarea” tags have a “placeholder” attribute that allows to specify a tip that help 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.

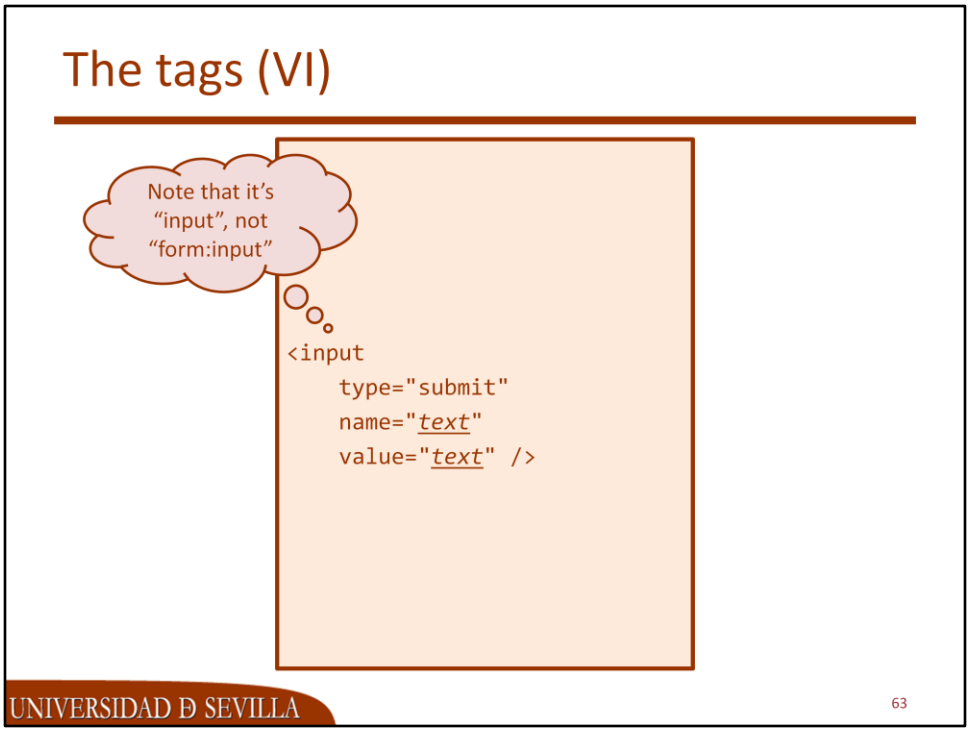


This slide reports on the “select” tag, which is used to display a dropdown list. The list of options is provided by means of tags “option” and “options”, namely:

- “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.
- “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, e.g., “title” or “name”; and “itemValue”, which references the attribute with the identifier of the entities in the list (its value is always “id” since this is the name of the identifier attribute in our entities).



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.



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.)

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.

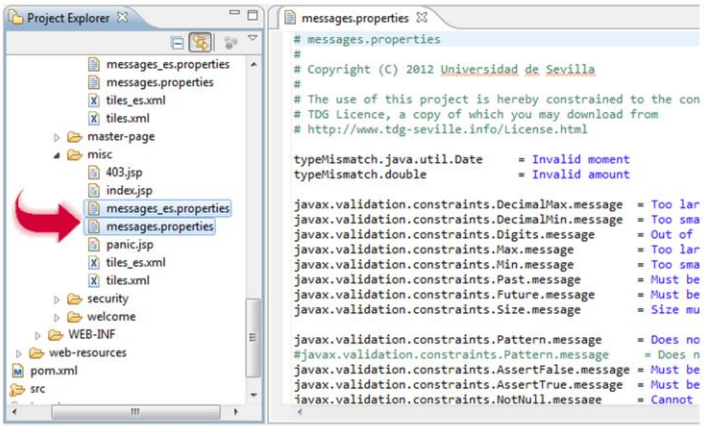
A note on the errors

UNIVERSIDAD Đ SEVILLA

66

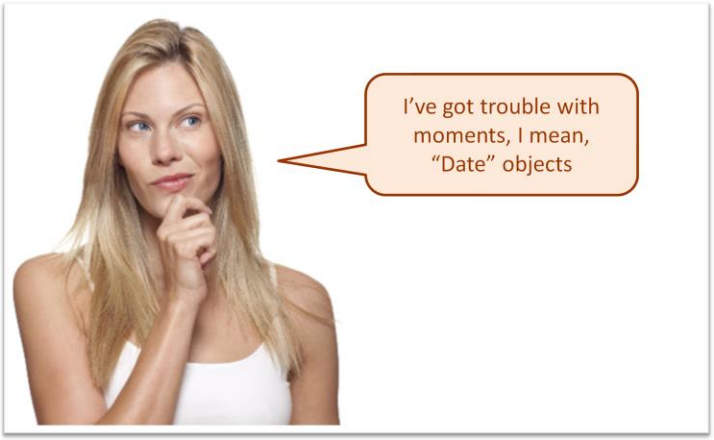
We're sure you're asking this: where's the text of the error messages? When we presented the "errors" tag, we mentioned a CSS class and an attribute name, but we didn't specify a message. So, where are messages "Invalid moment", "Must not be blank", or "Cannot be null"? More than that: how can we internationalise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

Localising error messages

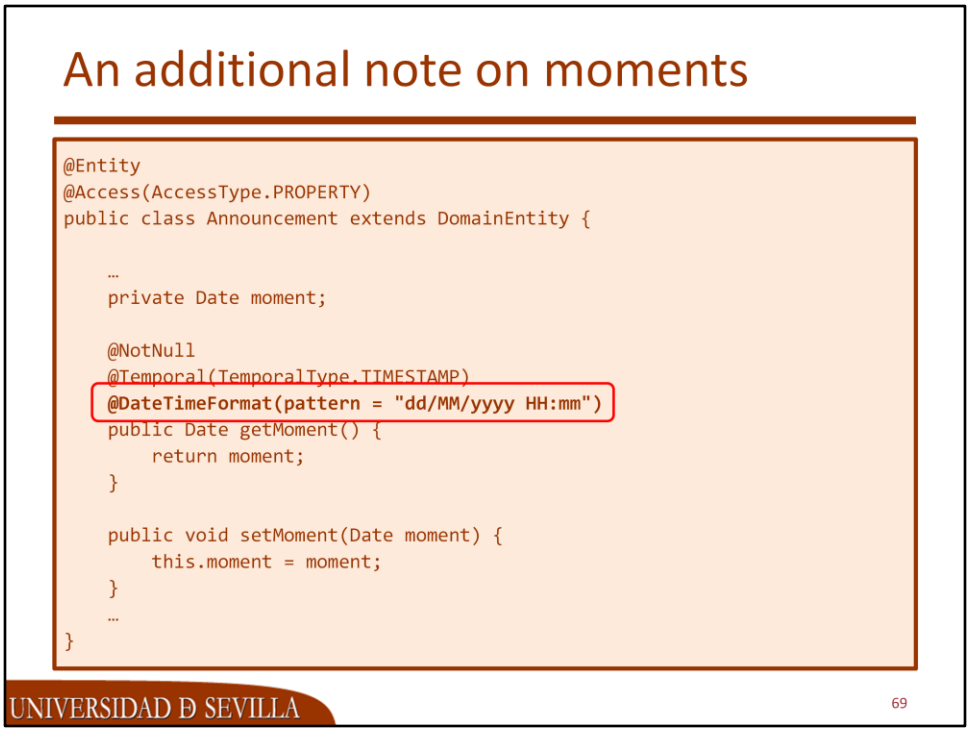


Do you remember the project template? In the “Deployed resources” virtual folder, there’s another folder called “misc” that contains several miscellaneous files, including “messages.properties” and “messages_es.properties”. These files have the messages that tag “form:errors” displays. They are i18n&l10n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

A note on the moments

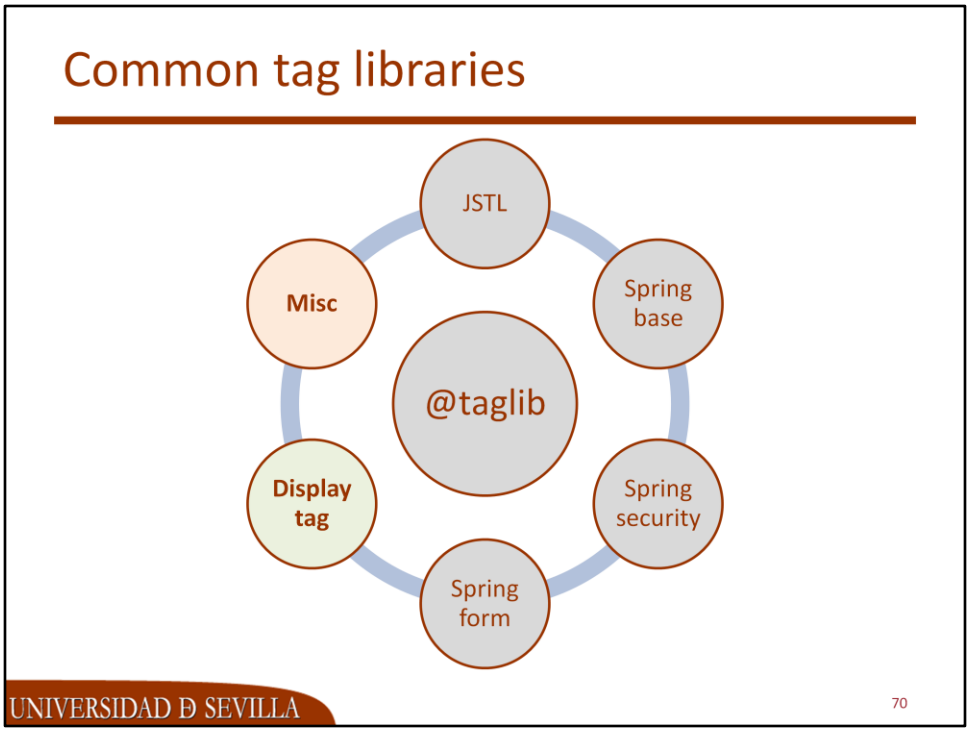


Before concluding this section, we must mention an additional problem with displaying and editing dates.

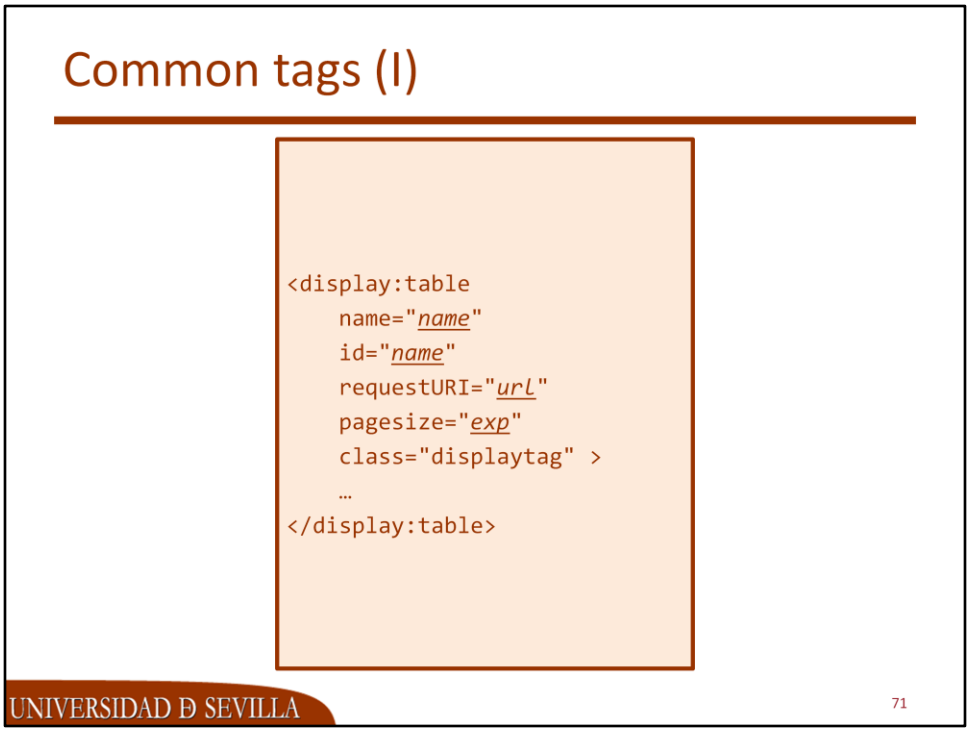


You know that you can use a "form:input" tag to edit an attribute of a simple type, which includes strings, integers, doubles, and also dates. The problem with dates is that there are too many formats to represent them; for instance a date like "01/02/03" may refer to "February 1, 2003", to "January 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!



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.

Inside the “table” tag, you can put as many columns as you wish. Please, keep reading.

Common tags (II)

```
<display:column>
    …
</display:column>

<display:column
    property="name"
    title="exp"
    sortable="exp"
    format="pattern"/>
```


Columns are specified by means of the “column” tag, which has two variants:

- A free column: in this case, you write “<display:column>…</display:column>” and the text you write inside is interpreted and output. For instance, you may use this format to display an “Edit” link.
- A property column: in this case, you must write 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.

The format attribute relies on the Java message formats that are documented at https://docs.oracle.com/javase/7/docs/api/java/text/MessageFormat.html. We’ll commonly use it to provide a format for dates; for instance, a pattern like “{0,date,dd/MM/yyyy HH:mm}” displays a date in format “day/month/year hours:minutes”.

Example

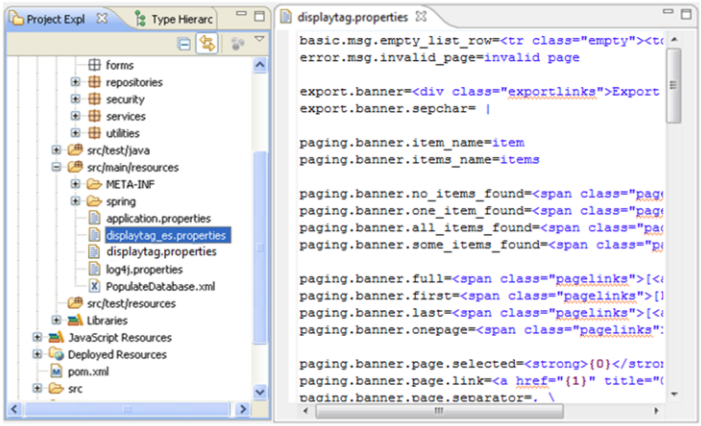
```
<display:table name="actors" id="row"
    requestURI="actor/administrator/list.do"
    pagesize="5" class="displaytag" >

  <spring:message code="actor.name" var="nameHeader" />
  <display:column property="name" title="${nameHeader}" sortable="true" />

  <spring:message code="actor.email" var="emailHeader" />
  <display:column property="email" title="${emailHeader}" sortable="true" />

  <spring:message code="actor.phone" var="phoneHeader" />
  <display:column property="phone" title="${phoneHeader}" sortable="true" />

  <spring:message code="actor.address" var="addressHeader" />
  <display:column property="address" title="${addressHeader}"
                  sortable="false" />

</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. Note that we localise the title of the columns by means of “spring:message” tags that retrieve them from the appropriate i18n&l10n bundles. It’s necessary to save the message to a variable and then use it in the “column” tag; in this case, you can’t nest the tags because they both are Spring tags.

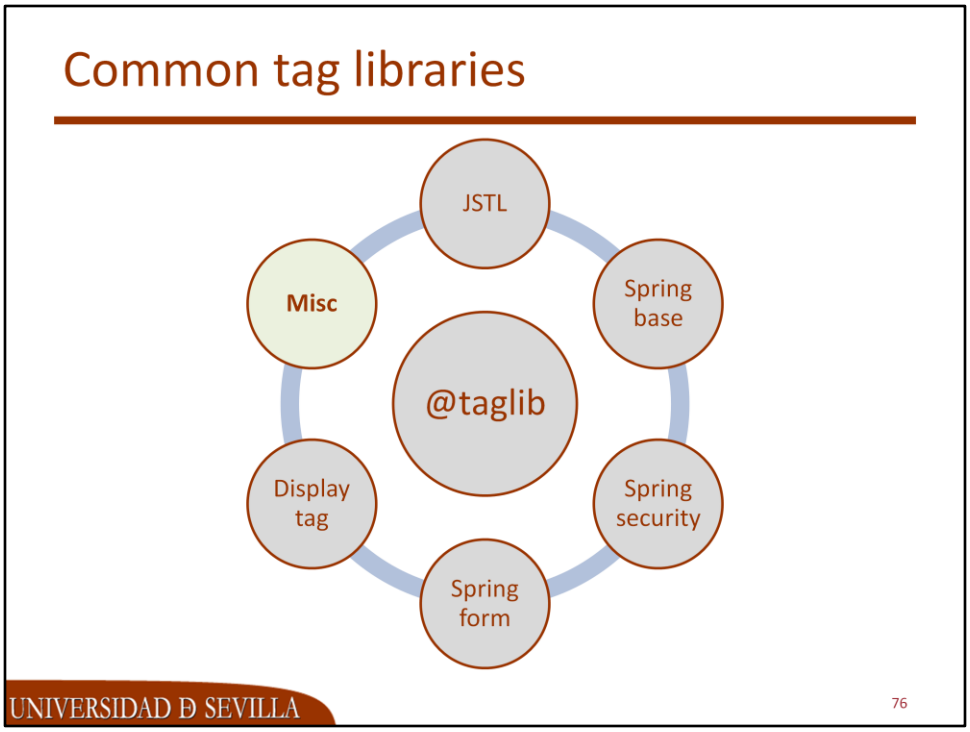
What the example looks like

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

What about i18n & l10n?



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. Displaying all items.” 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.

Common tags (I)

```
<tiles:insertAttribute name="attr" />
```


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

Common tags (II)

```
<jsp:useBean
    id="name"
    class="java-class" />
```


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”.

Common tags (III)

```
<fmt:formatDate
    value="exp"
    pattern="pattern" />

<fmt:formatNumber
    value="exp"
    pattern="pattern" />
```

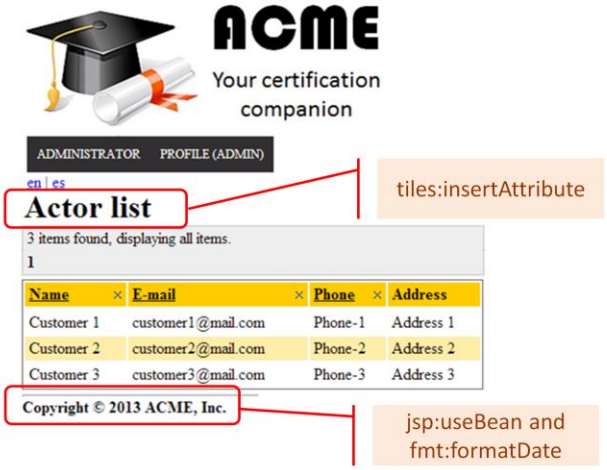

The last miscellaneous tags are “fmt:formatDate”, which allows to format a date according to a pattern, and “fmt:formatNumber”, which allows to format a number according to another pattern. The formatting patterns are documented at https://docs.oracle.com/javase/7/docs/api/java/text/MessageFormat.html.

Example

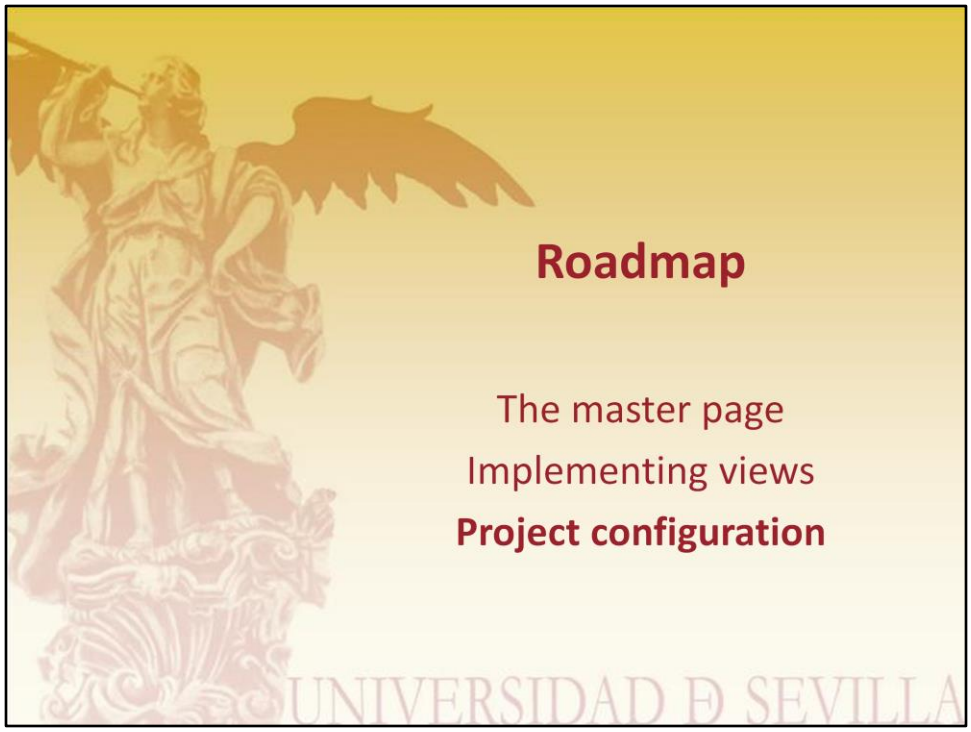
```
…
<h1> <tiles:insertAttribute name="title" />  </h1>
…
<b>
   <jsp:useBean id="date" class="java.util.Date" />
   Copyright &copy;
   <fmt:formatDate value="${date}" pattern="yyyy" />
   ACME, Inc.
</b>
…
```


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

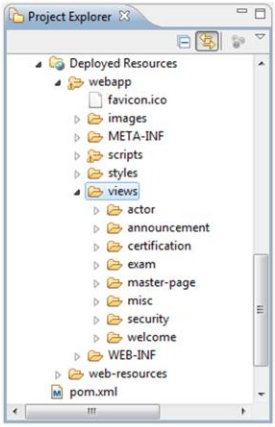


This is how the example looks like in a typical page.



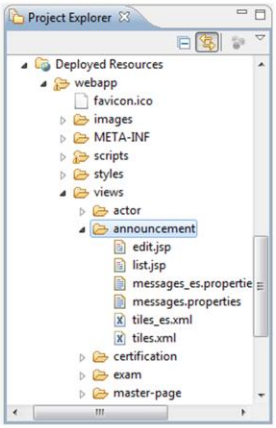
Let's move on! Let's now talk on the configuration you must perform to your projects so that they can handle your views.

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.

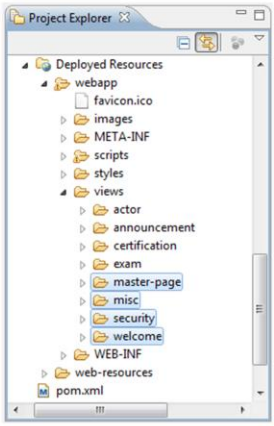
Structure of a subfolder



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

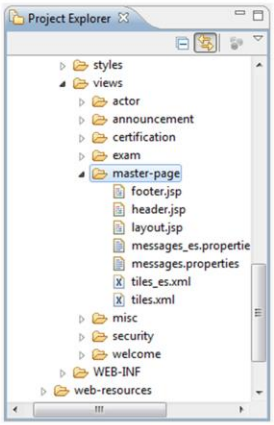


UNIVERSIDAD D SEVILLA

85

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.

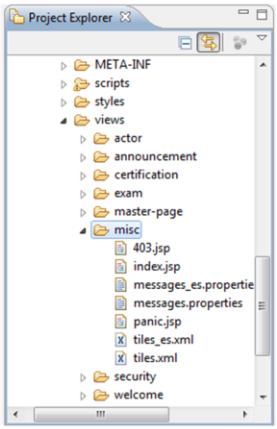
The `master-page` subfolder



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.

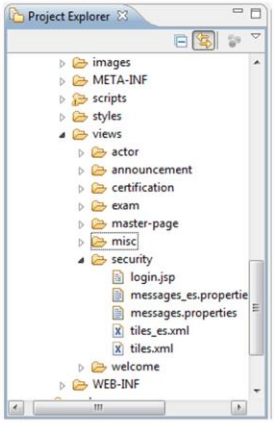
The `misc` subfolder



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

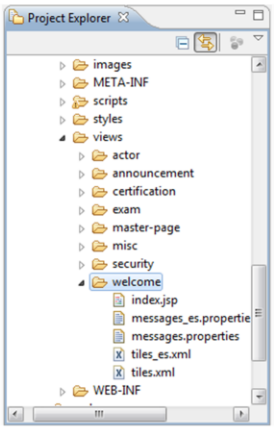
- “403.jsp” : this is a view that displays the following message when a user attempts to have access to a URL that is forbidden to him or her: “Oops! It seems that you don't have access to this resource.” You’re not likely to change this view in this subject.
- “index.jsp”: this view redirects to “/welcome/index.do”, 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



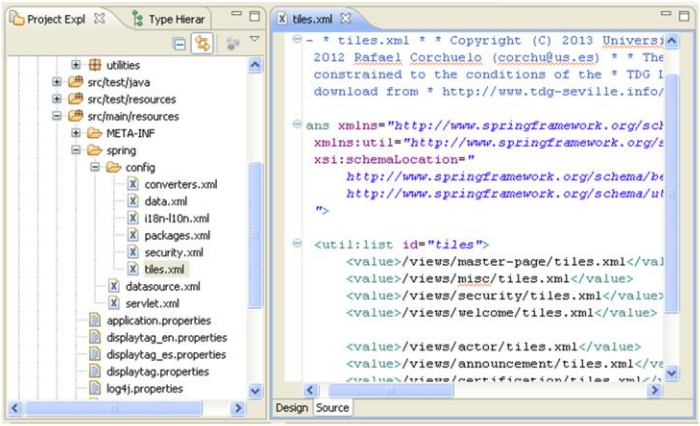
The “security” subfolder contains the files required to display a login screen: “login.jsp”, which defines the view, “messages_es.properties” and “messages.properties”, which are the i18n&l10n bundles, and “tiles_es.xml” and “tiles.xml”, which define the login web page building on the master page.

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.

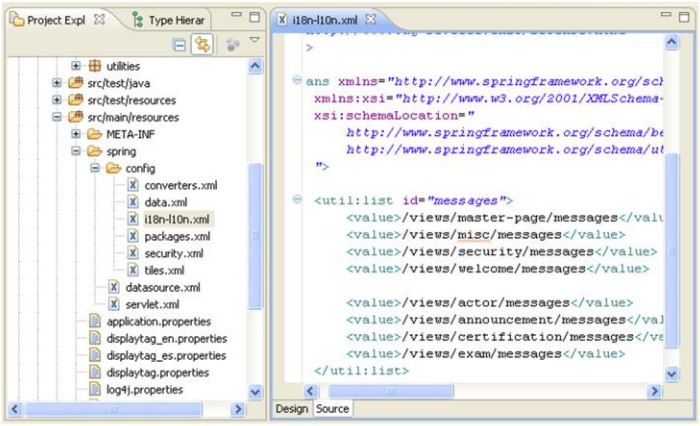
The `tiles.xml` configuration file



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.

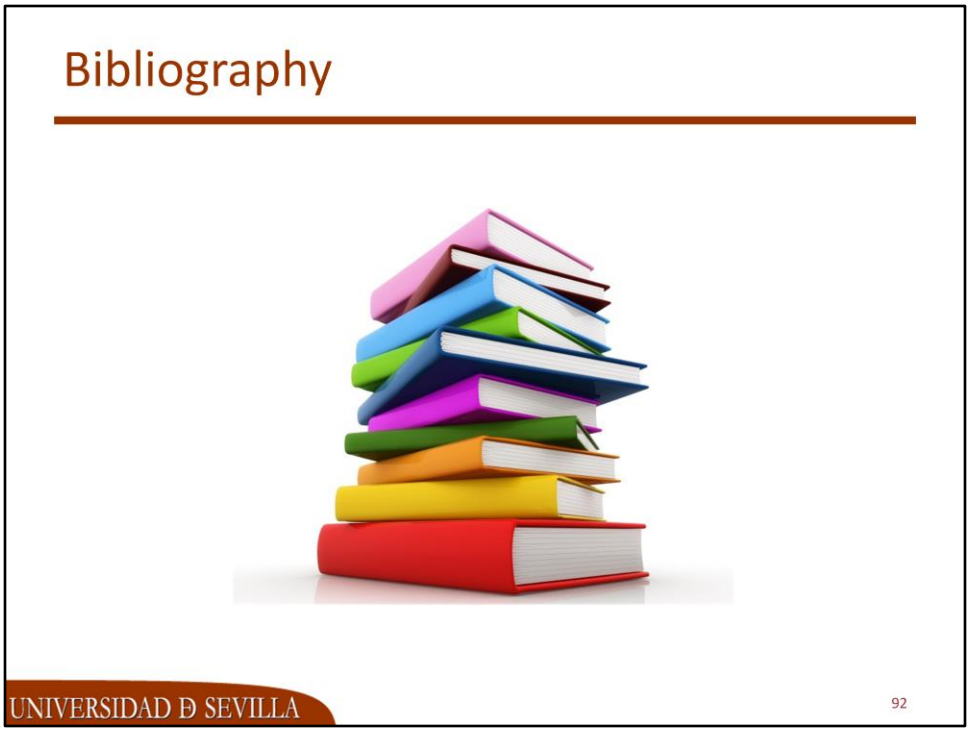
NOTE: please, note that you must provide the name of the English Tiles documents only, e.g., “/views/announcement/tiles.xml”. Spring locates the Spanish or whatever other variant automatically; you mustn't list it in this file. If you list it, you’ll get an error that is very difficult to understand,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

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.

The next lecture

- We need (coerced) volunteers
- Volunteer collaboration is strongly advised
- Produce a solution and a presentation
- Rehearse your presentation at home!
- Each presentation is allocated 100/*N* min
- Presentations must account for feedback

The next lecture is a problem lecture. We need some volunteers, who are expected to collaborate to produce a solution and a presentation. Please, rehearse your presentation at home taking into account that you have up to 100/*N* minutes per problem, including feedback, where *N* denotes the number of problems.

Quite an intense lesson, wasn't it? Please, bear in mind that it's a key lesson. Please, study it very carefully. Thanks for attending this lecture!