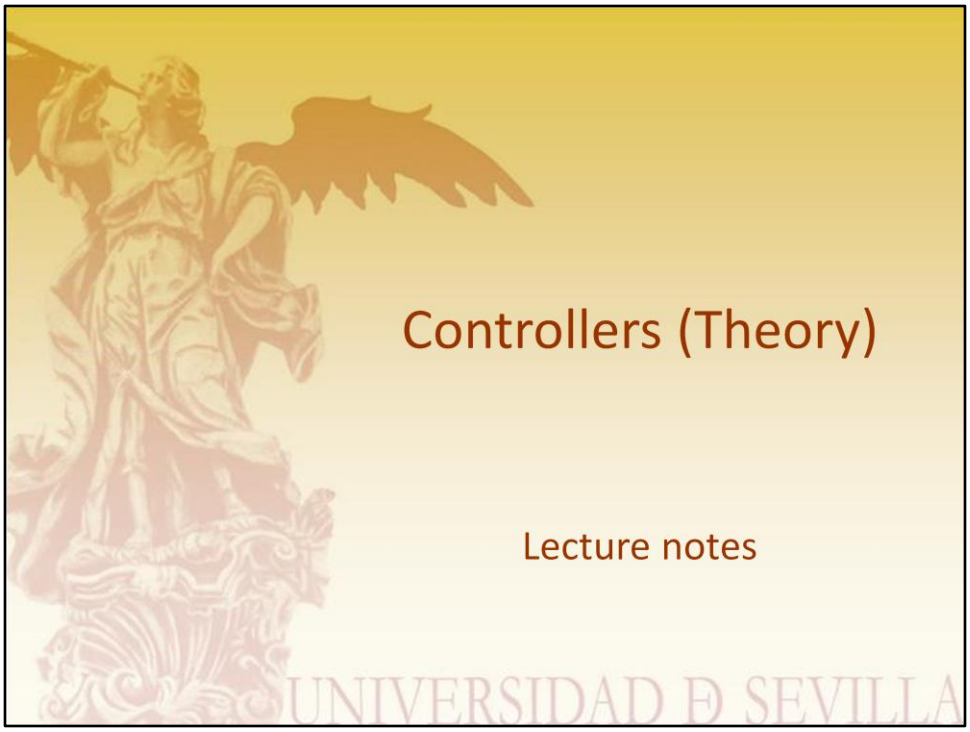


Welcome to this lecture! Our goal's to provide you with the theory that you require to implement controllers.

--

What are controllers?

UNIVERSIDAD Ð SEVILLA

2

As usual, we start with a question. What's a controller? We haven't worked a lot on them, but we've introduced the idea from an intuitive point of view.

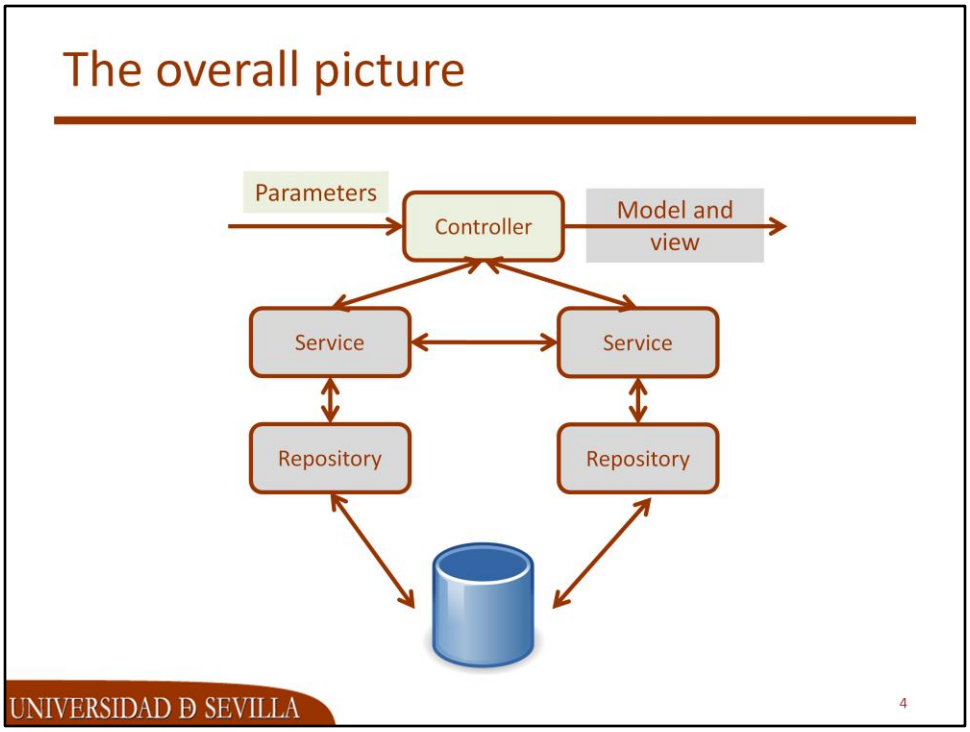
This is a good definition

A controller is a class whose objects orchestrate a number of services in order to serve an HTTP request to a URL

UNIVERSIDAD Ð SEVILLA

3

This is the definition that we're going to use in this subject: a controller is a class whose objects orchestrate a number of services in order to serve an HTTP request to a URL. The name controller suggests that they're kind of orchestrators; that is, they don't do a lot of complex work, but co-ordinate the services to do so.



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 lectures, we studied repositories, services, models, and views; in this lesson, we'll study controllers and their parameters.

How are they devised?

UNIVERSIDAD Ð SEVILLA

5

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

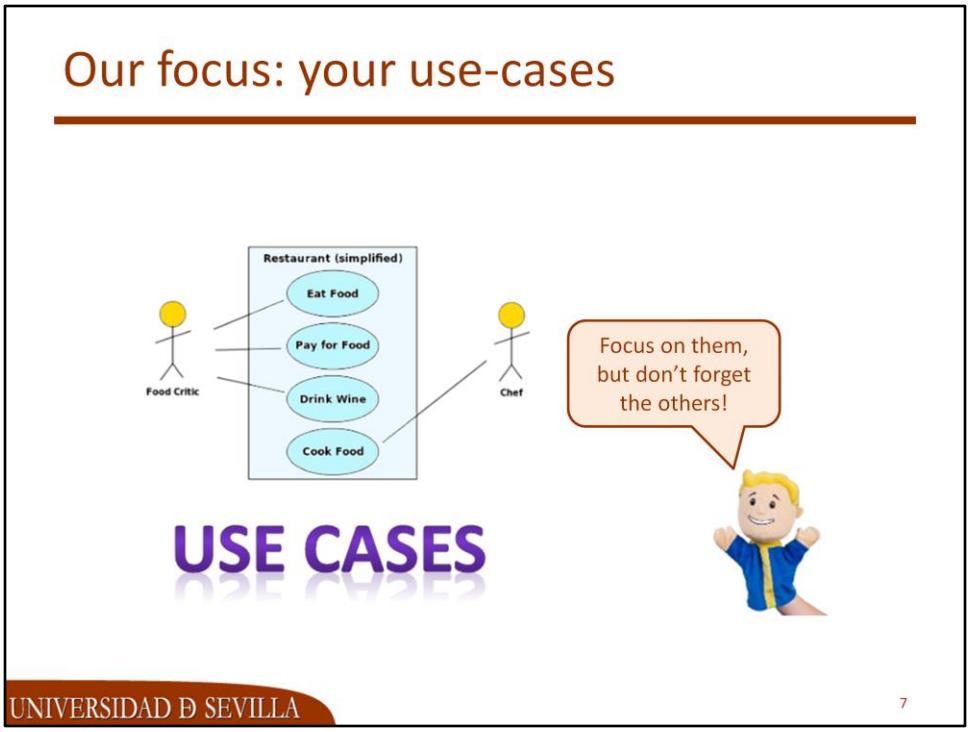
Starting point: your requirements

- 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 your controllers are your requirements.



In particular, we'll focus on the use cases. Please, recall that a use case describes how a user performs a typical operation with a web information system. 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 on your requirements if you wish to produce good controllers.

Step 1: learn a controller's structure

UNIVERSIDAD Ð SEVILLA

8

The first step is to understand the structure of a typical controller, that is, the parts of which it is composed.

Step 2: implement listing controllers

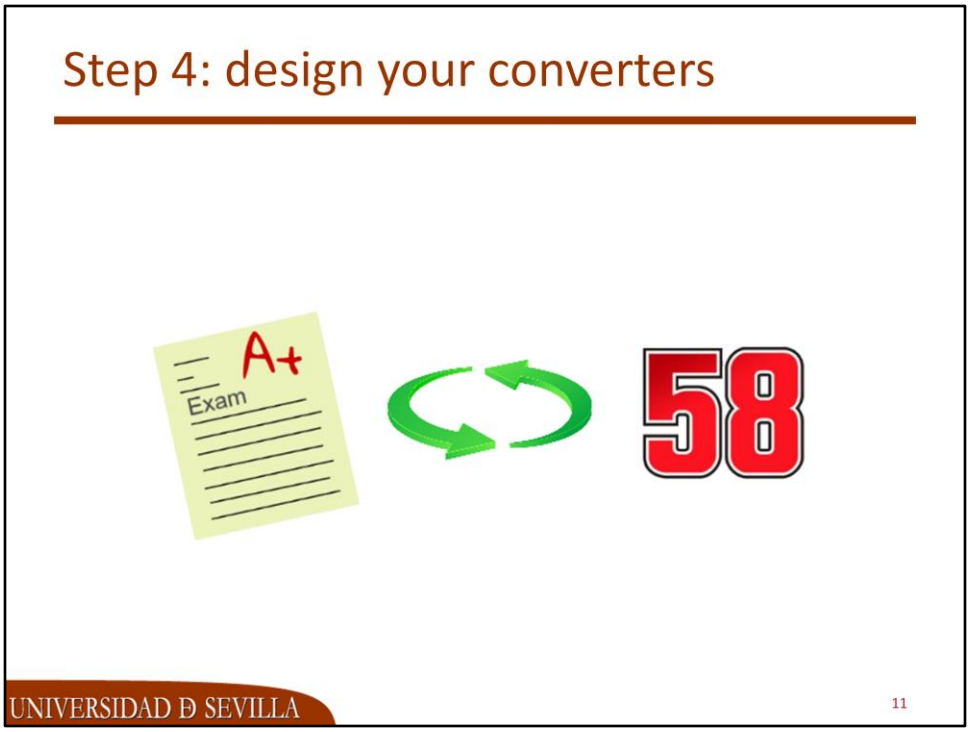
UNIVERSIDAD Ð SEVILLA

9

The next step is to implement your listing controllers.

Step 3: implement edition controllers

And then the edition controllers.



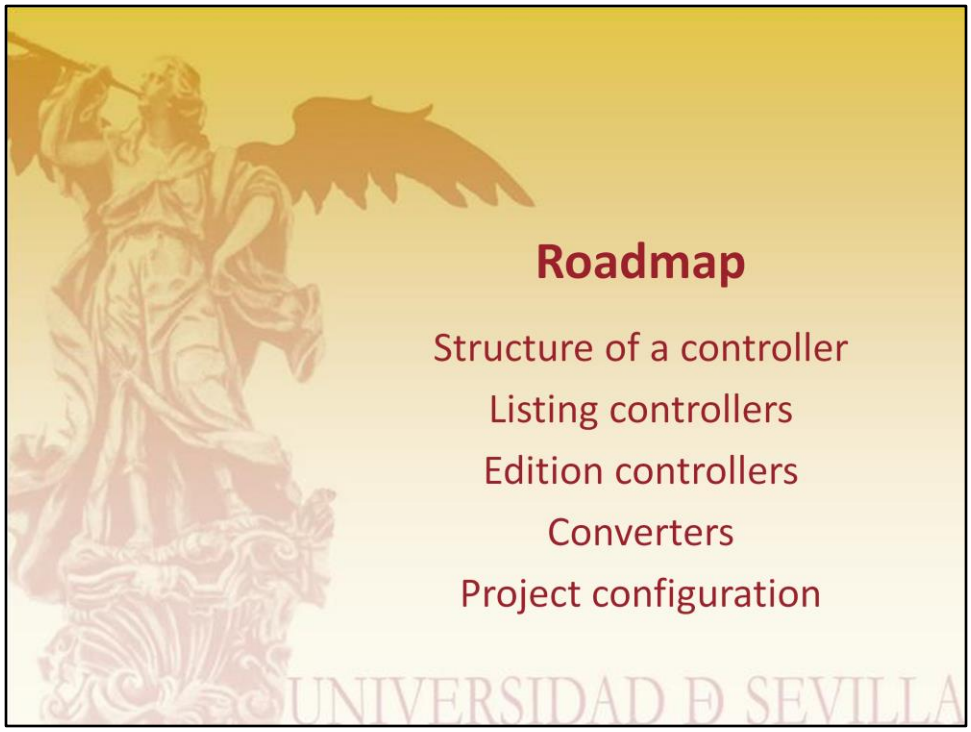
The fourth step consists in creating a number of converters, which basically care of converting objects into strings and vice versa. Please, keep reading to learn about these components.

Step 5: configure your project

UNIVERSIDAD Ð SEVILLA

12

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



So, this is our roadmap for today's lecture. We'll first present the structure of a controller, then will report on how to implement a listing controller and an edition controller, next will report on converters and, finally, on how to configure your project.

Let's start reporting on the structure of a typical controller.

Recalling the definition of controller

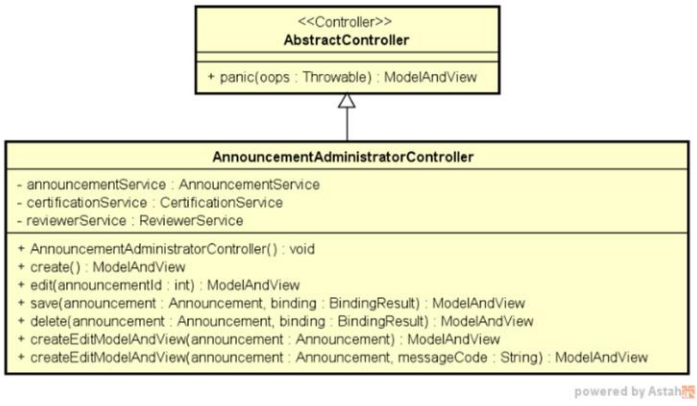
A controller is a class whose objects orchestrate a number of services in order to serve an HTTP request to a URL

UNIVERSIDAD D SEVILLA

15

Before delving into the details, it's important to recall the definition of controller. Please, recall that a controller is a class whose objects orchestrate a number of services in order to serve HTTP requests to some URLs.

The overall design



This slide shows the overall design. Note that our controllers extend a class called "AbstractController", which is provided by our framework, and they have a number of attributes to refer to the services that are orchestrated, and a number of methods that resemble typical CRUD operations. In the following slides, we delve into the details.

The AbstractController class

```
@Controller
public class AbstractController {

   @ExceptionHandler(Throwable.class)
   public ModelAndView panic(Throwable oops) {
      ModelAndView result;

      result = new ModelAndView("misc/panic");
      result.addObject("name", ClassUtils.getShortName(oops.getClass()));
      result.addObject("exception", oops.getMessage());
      result.addObject("stackTrace", ExceptionUtils.getStackTrace(oops));

      return result;
   }

}
```


Class “AbstractController” cares of catching exceptions and showing a panic page when they occur. Can you remember the “panic.jsp” view that we presented in a previous lecture? We mentioned that this is the view that is shown when an uncaught exception is thrown by a controller. This view has three model variables, namely:

- “name”, which stores the name of the exception.
- “exception”, which stores the message of the exception.
- “stackTrace”, which stores the stack trace.

Please, note that the “panic.jsp” view provides a lot of information that you shouldn’t show in a production scenario. It’s intended only for learning purposes, just to facilitate your life as an apprentice.

Your controller class

```
@Controller
@RequestMapping("/announcement/administrator")
public class AnnouncementAdministratorController
             extends AbstractController {
  …
}
```


Your controllers must be declared as regular classes that extend class "AbstractController" and have the following annotations: "@Controller" and "@RequestMapping". Annotation "@Controller" informs Spring that this class implements a controller; annotation "@RequestMapping" instructs Spring to redirect every request to a URL that starts with "protocol://domain[:port]/[app-context/]announcement/administrator" to this controller. Note that you don't have to specify the protocol, the domain, the port, or the application context, only the entity (mandatory) and the authorities (optional). This controller will serve requests to URLs like "http://localhost:8080/Acme-Certifications/announcement/administrator/list.do" in the development configuration or "http://www.acme.com/announcement/administrator/list.do" in the production configuration.

The services

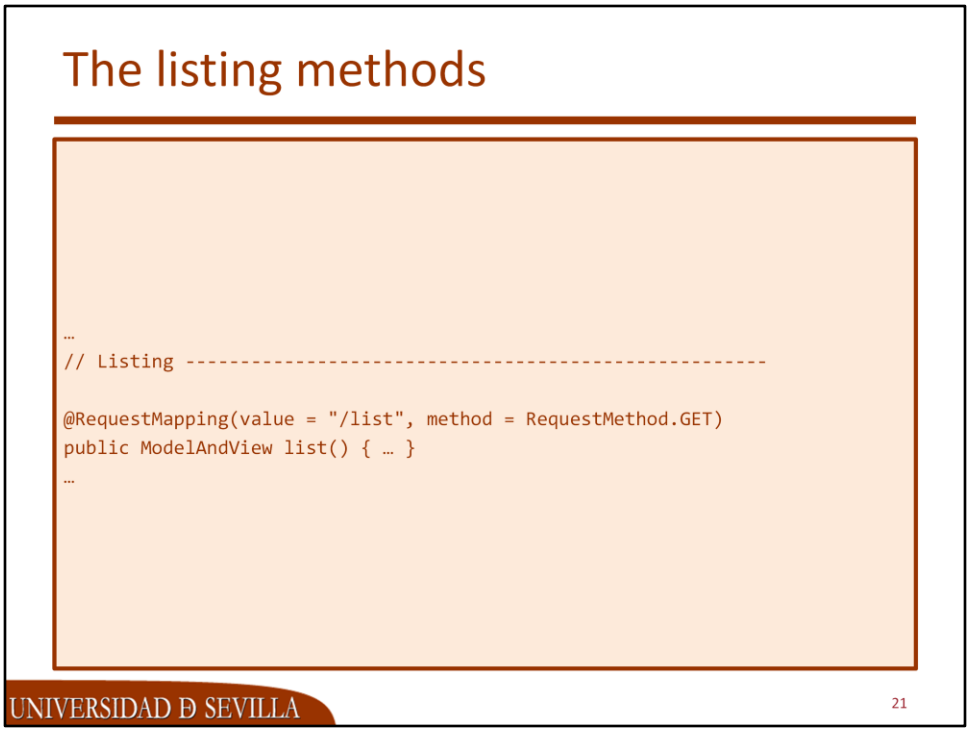
```
…
// Services ------------------------------------------------------

@Autowired
private AnnouncementService announcementService;

@Autowired
private CertificationService certificationService;

@Autowired
private ReviewerService reviewerService;
…
```


Let's now delve into the internals of a controller class. They typically start with a section that we call "Services", in which we declare private attributes that hold references to the services that we need in the controller. Note that, we need to put an "@Autowired" annotation in front of each attribute so that Spring injects the appropriate instances into them.

The constructors

```
…
// Constructors -----------------------------------------------

public AnnouncementAdministratorController() {
  super();
}
…
```


Then comes the constructors section. As usual, you're very unlikely to require a constructor other than the default constructor. They are intended for debugging purposes only, just in case you need to trap the creation of a controller by Spring.



Then comes the section devoted to listing methods. Later, we'll provide additional details on how to implement them. So far, it's enough to know that they must return an object of class "ModelAndView", which is quite a simple object that stores a view name and a model (that is, a map from variables onto objects). Note that the only difference with respect to a regular method is that you must prepend it with an "@RequestMapping" annotation with the following parameters:

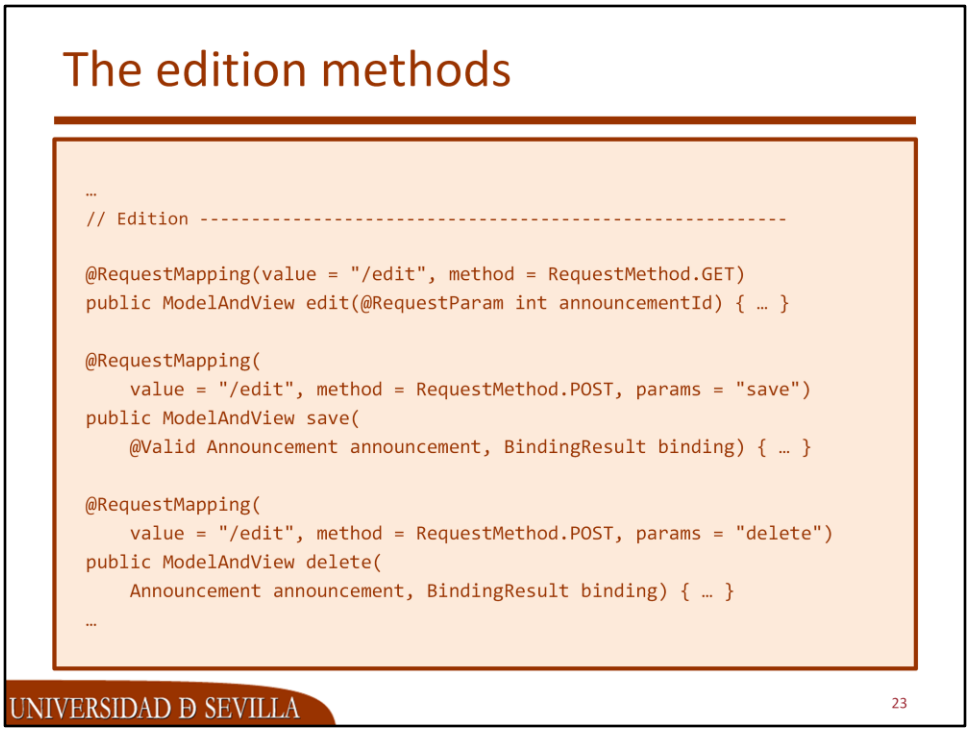
- "value": this specifies the name of the action that this method serves. (Note that the ".do" suffix must be omitted.)
- "method": this specifies the HTTP request method of the action. It can be either "RequestMethod.GET" or "RequestMethod.POST".

The listing method in this slide serves URLs like "http://localhost:8080/Acme-Certifications/announcement/administrator/list.do" in development configurations, but "http://www.acme.com/announcement/administrator/list.do" in production configurations; in both cases, the URL must be requested using the HTTP GET method.

The creation methods

```
…
// Creation ---------------------------------------------

@RequestMapping(value = "/create", method = RequestMethod.GET)
public ModelAndView create() { … }
…
```


Then come the creation methods. Like before, they are regular methods that return a "ModelAndView" object and have an "@RequestMapping" annotation in front of them. The method in this slide, for instance, serves GET requests of the form "http://localhost:8080/Acme-Certifications/announcement/administrator/create.do" in development configurations and "http://www.acme.com/announcement/administrator/create.do" in production configurations.



Then comes the edition section. Editing an entity involves three steps: first, you must click on an "Edit" link that results in a GET request to the server, which, in turn, must return a form to edit an entity; then you must press either a "Save" button or a "Delete" button. To do so, we need three methods, namely:

- The first one serves GET requests to URLs like "http://localhost:8080/announcement/administrator/edit.do?announcementId=99". Note that this URL includes a parameter that is mapped onto a parameter of the method by means of annotation "@RequestParam". The purpose of this method is to find the announcement whose identifier is specified and then return a view with a form to edit it.
- The second one serves POST requests to URLs like "http://localhost:8080/announcement/administrator/edit.do" that result from pressing a "Save" button. Note that if a method serves POST requests, then you must add an optional parameter called "params" to your "@RequestMapping" annotation so as to specify which of the buttons is served by this method. Note too, that a method that serves a POST request gets two parameters: a parameter with the domain object that was edited and a "BindingResult" object that has information about the data entered by the user in the corresponding form and validation errors, if any. We'll provide additional details on this later.
- The third method serves POST requests that originate from a "Delete" button.

The ancillary methods

```
…
// Ancillary methods ----------------------------------------

protected ModelAndView createEditModelAndView(
      Announcement announcement) { … }

protected ModelAndView createEditModelAndView(
      Announcement announcement, String messageCode) { … }
…
```

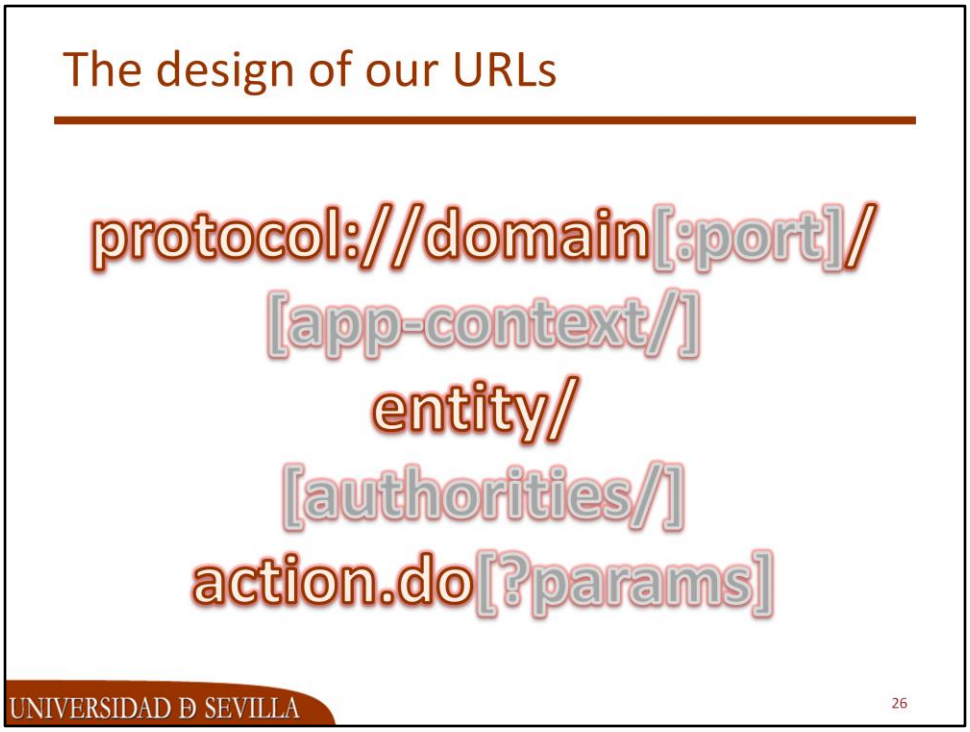

Finally comes an ancillary methods sections. Ancillary methods typically help you factor the creation of “ModelAndView” objects. It’s difficult to foresee the ancillary methods you’ll require in your controller. In this example we present two ancillary methods that are very common. Note that they are regular methods, with absolutely no annotations.

This is a good insight

UNIVERSIDAD Ð SEVILLA

25

Have you realised that there's a pattern in the URLs? Let's analyse it in the following slide.

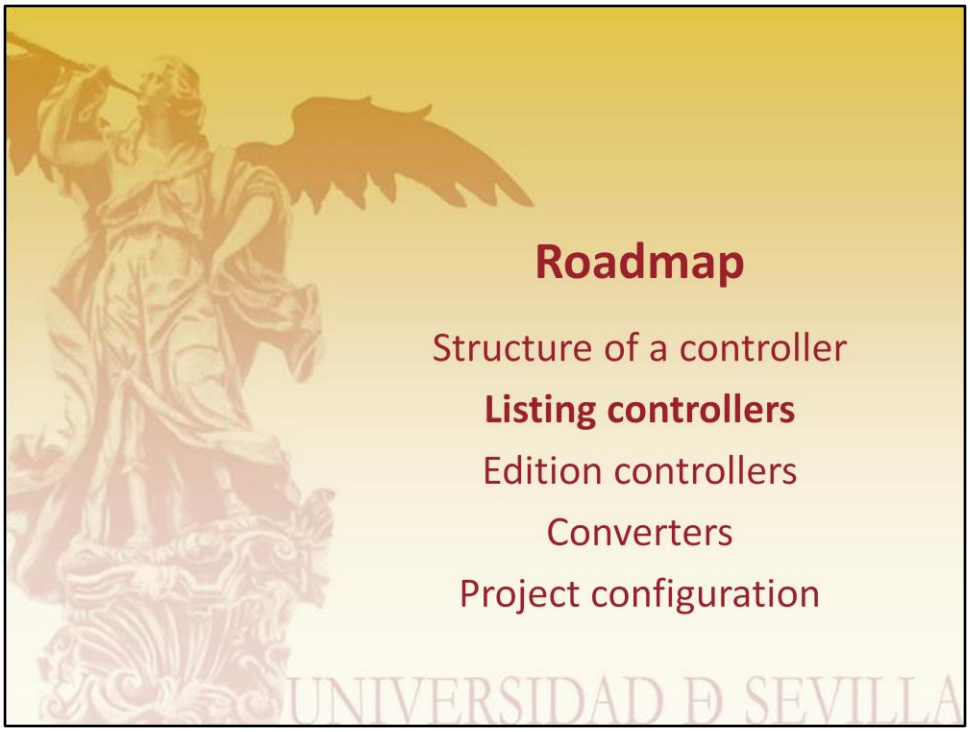


This slide shows the parts of which a typical URL is composed (optional parts are in grey, within squared brackets):

- "protocol://domain[:port]/": this refers to a protocol (generally "http" or "https"), an internet domain (e.g., "localhost" or "www.acme.com"), and a port (e.g., "8080" in the development configuration and "80" in the production one). Note that the port is optional; if it's omitted, then port "80" is the default one.
- "app-context": this refers to the application context, e.g., "Acme-Certification". The application context's mandatory in the development configuration, but it can be easily omitted in the production configuration.
- "entity": this refers to the name of an entity in our application domain, e.g., "announcement" or "certification".
- "authorities": this refers to a number of authorities in our application, e.g., "administrator" or "reviewer,customer". Note that a URL may be accessible to several authorities, in which case, we list them using commas in-between. The authorities part can be omitted, in which case, the URL can be accessed by any authorities, including unauthenticated users.
- "action.do[?params]": this refers to an action to be performed plus some optional parameters, e.g., "list.do" or "register.do?announcementId=23".

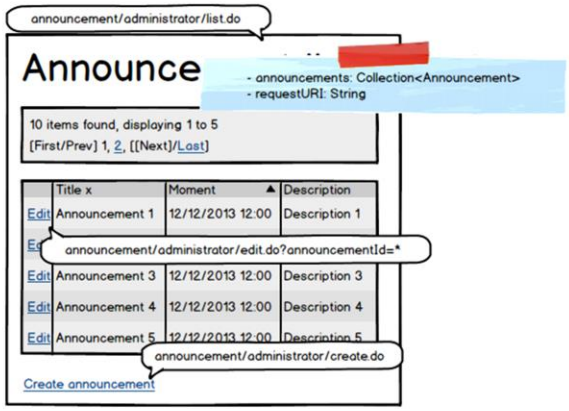
The following are a few sample URLs for the ACME certification system: "http://localhost:8080/Acme-Certifications/certification/customer/list.do",

“http://localhost:8080/Acme-Certifications/announcement/administrator,manager/edit.do?announcementId=123”, “http://www.acme.com/actor/administrator/list.do”, or “http://www.acme.com/exam/findByCertificationId.do? certificationId=987”. Note that there can be some exceptions; for instance, URLs “http://localhost:8080/Acme-Certifications” or “http://localhost:8080/Acme-Certifications/security/login.do” are common and they don’t follow the previous pattern because they are not actually related to business actions.



Let's now report on the listing controllers.

The listing mock-up



To start working, we need a mock-up with its corresponding model and link specification.

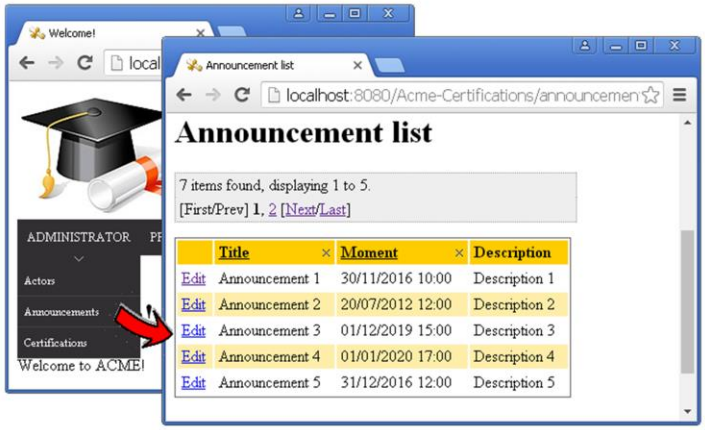
The listing method

```
…
@RequestMapping(value = "/list", method = RequestMethod.GET)
public ModelAndView list() {
  ModelAndView result;
  Collection<Announcement> announcements;

  announcements = announcementService.findAll();

  result = new ModelAndView("announcement/list");
  result.addObject("announcements", announcements);
  result.addObject("requestURI", "announcement/administrator/list.do");

  return result;
}
…
```

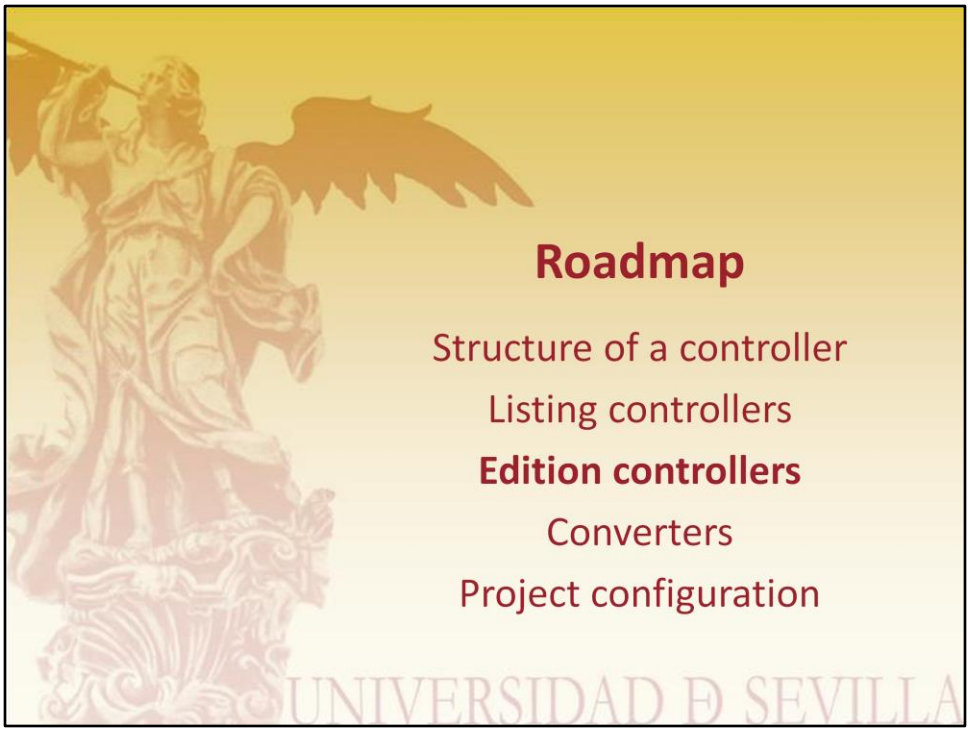

In this slide, we show the method in class “AnnouncementAdministratorController” that supports the previous mock-up. It’s not difficult to understand:

- First, we resort to the announcement service to retrieve all of the announcements in the database.
- Next, we create a “ModelAndView” object. Note that the constructor of this class gets a string parameter in which you have to specify the name of the view that must be instantiated. We call that view “announcement/list”. (Note that we don’t need to create an “announcement/administrator/view”, since we can use a single view with the core and then use “security:authorize” sections to decide which sections are displayed to each user depending on his or her authorities.)
- The following lines simply instantiate the variables in the model by calling method “addObject” on the result “ModelAndView” object. We first instantiate variable “announcements” to the collection of announcements that the service has returned and then variable “requestURI” to “announcement/administrator/list.do”.

This is what it looks like

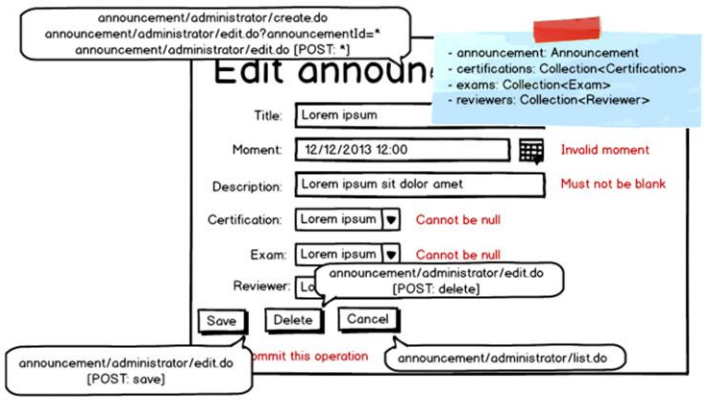


The previous method's enough to produce the output in this slide. Note that there's a pagination bar on top of the grid, that you may sort the grid by clicking on the first or the second columns, and that each announcement has an "Edit" link; there's also a "Create announcement" link at the bottom. Nice, isn't it?



Let's now report on the edition controllers, which are a little more involved.

The edition mock-up



This slide shows the edition mock-up we've designed to edit an announcement, including the model, and the link specification.

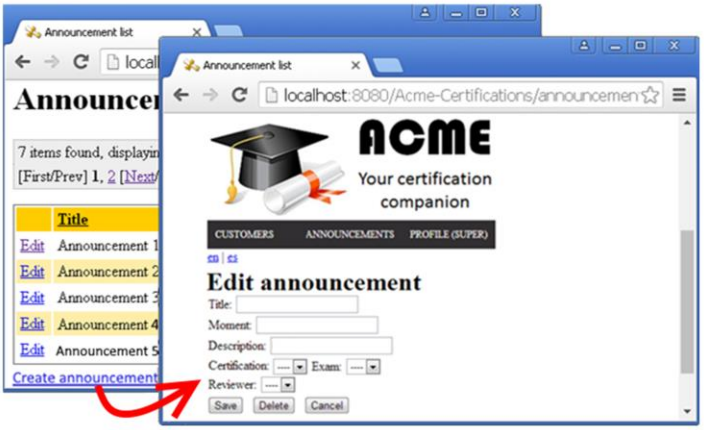
The method to create an entity

```
@RequestMapping(value = "/create", method = RequestMethod.GET)
public ModelAndView create() {
    ModelAndView result;
    Announcement announcement;

    announcement = this.announcementService.create();
    result = this.createEditModelAndView(announcement);

    return result;
}
```


This slide shows the method to create an entity. It's executed whenever the user requests a URL of the form "http://localhost:8080/Acme-Certifications/announcement/administrator/create.do". The method works as follows:

- First, we use the announcement service to create a new announcement with default values for its attributes.
- We then invoke method "createEditModelAndView" to create the appropriate model and view building on the announcement that we've just created. We'll review this method later.
- We return the resulting model and view.

This is what it looks like



This slide shows what the form looks like in reality, so that you can have a better understanding.

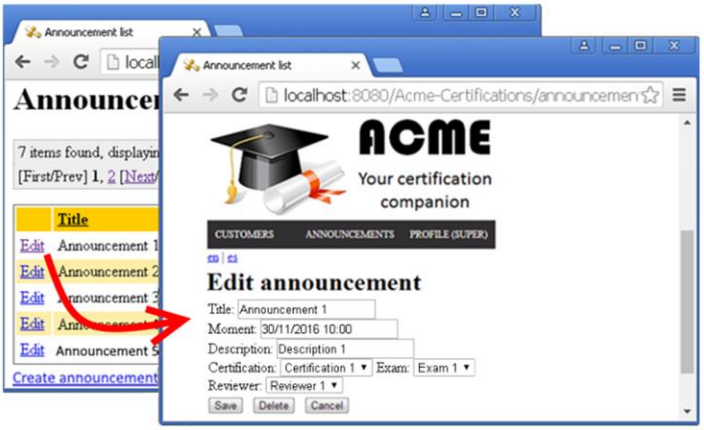
The method to edit an entity

```
@RequestMapping(value = "/edit", method = RequestMethod.GET)
public ModelAndView edit(@RequestParam int announcementId) {
  ModelAndView result;
  Announcement announcement;

  announcement = announcementService.findOne(announcementId);
  Assert.notNull(announcement);
  result = createEditModelAndView(announcement);

  return result;
}
```

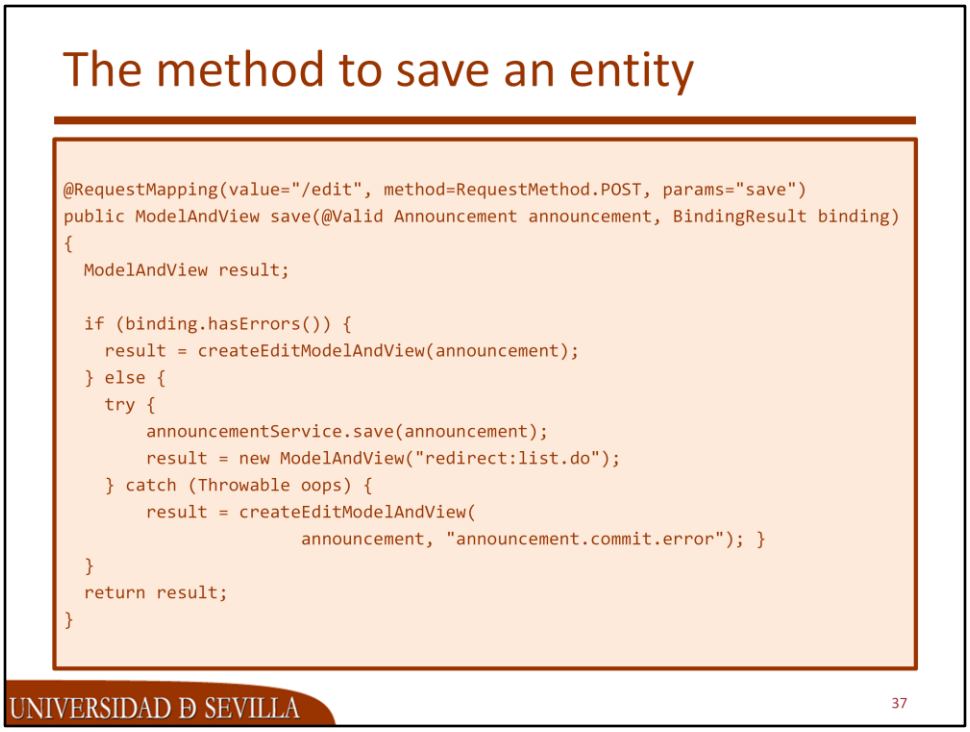

This slide shows the method to edit an entity. It's executed whenever the user requests a URL of the form "http://localhost:8080/Acme-Certifications/announcement/administrator/edit.do?announcementId=99". Note that the parameter in the URL is passed as a regular parameter to the method thanks to the "@RequestParam" annotation. The statements in the method do the following:

- First, we use the announcement service to retrieve the announcement with the identifier requested by means of parameter "announcementId".
- Note that we then check that the announcement is not null using an "Assert.notNull" statement. The reason is that this method is activated when the user requests a URL of the previous form using the GET method. What would happen if the user entered that URL in the address bar to edit an inexistent announcement? What would happen if the user selected an "Edit" link but another administrator deleted that announcement a second before? In both cases, the call to the announcement service would return a "null" reference to indicate that the announcement whose identifier has been indicated doesn't exist. Since this situation is quite exceptional, it makes sense to abort the execution of the controller, which results in a panic page.
- If everything is OK, then we invoke method "createEditModelAndView" to create the appropriate model and view building on the announcement that we've just retrieved. We'll review this method later.

This is what it looks like



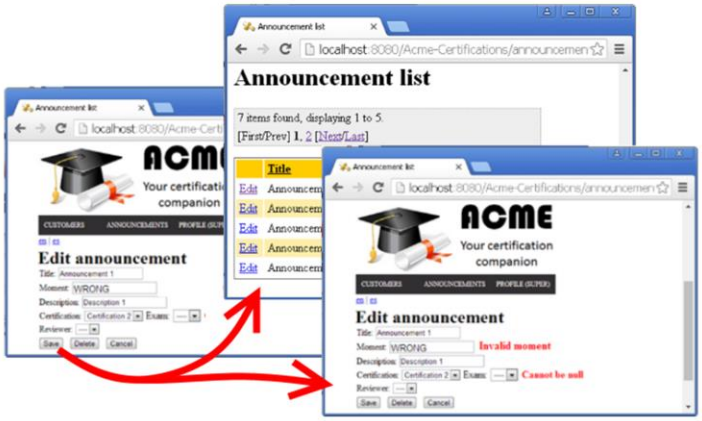
This slide shows what the form looks like in reality, so that you can have a better understanding.



This is the method to save the entity that is being edited. Please, note the following:

- If a method serves a POST request, "@RequestMapping" must include a "params" parameter with the name of the corresponding button.
- The method gets two parameters: "announcement", which has a reference to the object being edited, and "binding", which has a reference to a "BindingResult" object. Note that the "announcement" parameter has an "@Valid" annotation, which requests Spring to check the validation constraints that we defined on the domain entities. The "binding" parameter gets an object in which Spring informs about the validation errors.
- The body of the method consists of an if-then-else statement that works as follows: a) if the entity being edited has validation errors, it then creates a "ModelAndView" object and returns it so that error messages are shown; b) if the binding doesn't have any errors, it then tries to save the announcement, and redirects to the "list.do" action; if an exception happens, then it's caught and a new "ModelAndView" object is created; in this case, the error message with code "announcement.commit.error" is shown.

This is what it looks like



This is what the form looks like if it has validation errors. If there aren't any validation errors, then the listing of announcements is returned.

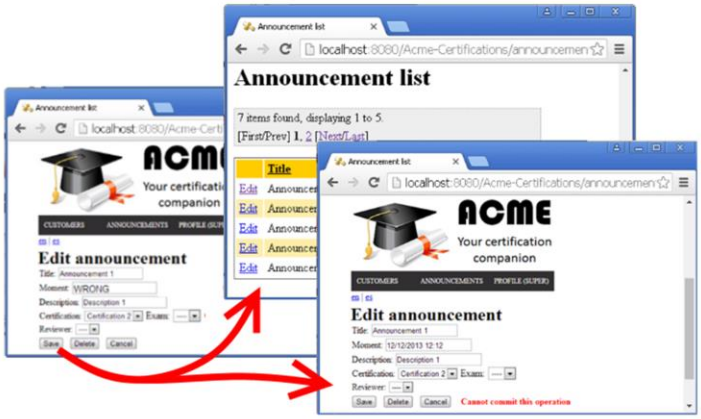
The method to delete an entity

```
@RequestMapping(
       value = "/edit", method = RequestMethod.POST, params = "delete")
public ModelAndView delete(Announcement announcement, BindingResult binding)
{
  ModelAndView result;

  try {
    announcementService.delete(announcement);
    result = new ModelAndView("redirect:list.do");
  } catch (Throwable oops) {
    result = createEditModelAndView(
                           announcement, "announcement.commit.error");
  }
  return result;
}
```


This is the method to delete the object that is being edited. Note the following:

- Once again, please, realise that we need to add a “params” parameter to the “@RequestMapping” annotation to inform Spring that this is the method that serves the POST requests that originate from the “Delete” button.
- Realise, too, that this method gets an argument of type “Announcement” and that we don’t need to add an “@Valid” annotation; if the user’s made changes and the entity’s invalid that shouldn’t prevent us from deleting it. Note, however, that the method needs to get a “BindingResult” parameter, even if you’re not going to use it.
- Then the method tries to delete the object from the database. If everything works well, then we return to the listing view; otherwise, we create a “ModelAndView” object that shows the object and an error message.

This is what it looks like



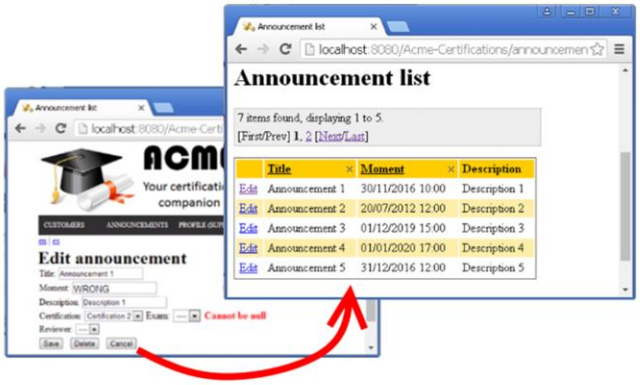
This slide shows what the screen looks like if the delete operation cannot be committed. Typically, the reason is that there are some related object which would become orphan if the entity being edited is deleted. If the operation can be committed, then the listing view is returned.

The method to get back

```
<input type="button"
    name="cancel"
    value="Cancel"
    onclick="javascript:
        relativeRedir('announcement/administrator/list.do');" />
```


Note that no controller's involved in cancelling the edition of an entity. Pressing the "Cancel" button just executes a piece of ECMA Script that redirects the browser to the corresponding listing.

This is what it looks like



And this is the result: we get back to the listing view.

Finally, the ancillary methods

UNIVERSIDAD Ð SEVILLA

43

Before concluding this section, we'd like to report on the two ancillary methods that we've used previously.

A simple by-pass

```
protected ModelAndView createEditModelAndView(Announcement announcement) {
  ModelAndView result;

  result = createEditModelAndView(announcement, null);

  return result;
}
```


The first one is “createEditModelAndView”. Its responsibility’s to create a “ModelAndView” object that’s appropriate for the announcement that’s passed as a parameter. It’s very simple, since it simply by-passes the call to the second method.

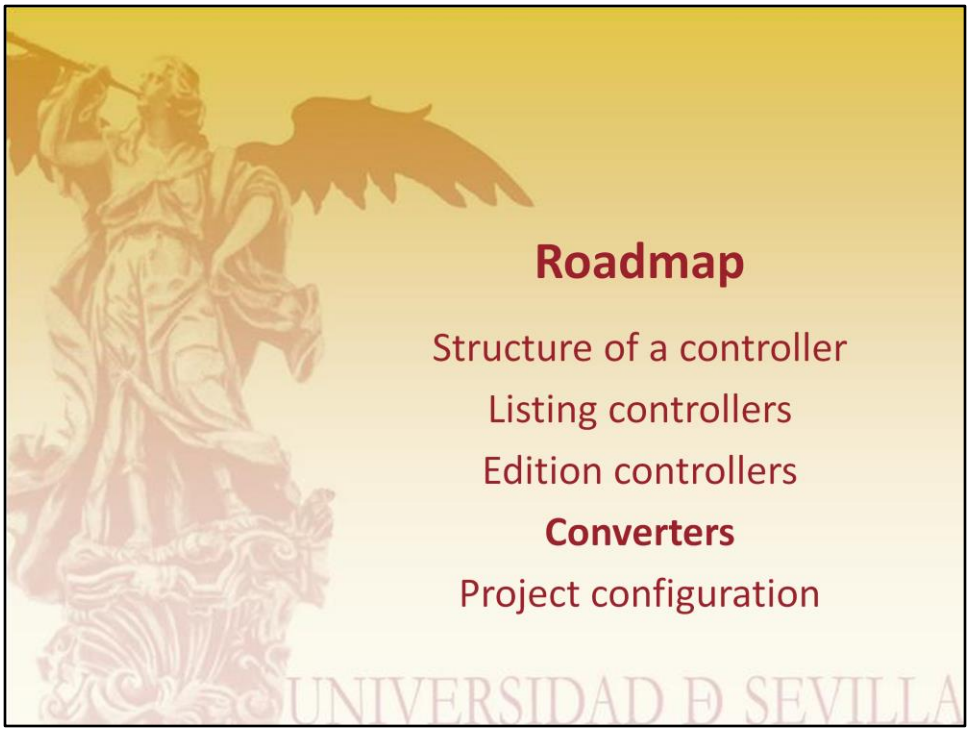
The core method

```
protected ModelAndView createEditModelAndView(Announcement announcement, String messageCode) {
  ModelAndView result;
  Certification certification;
  Collection<Certification> certifications;
  Collection<Exam> exams;
  Collection<Reviewer> reviewers;

  certifications = certificationService.findAll();
  if (announcement.getCertification() == null) {
    certification = null;
    exams = null;
  } else {
    certification = announcement.getCertification();
    exams = certification.getExams();
  }
  reviewers = reviewerService.findAll();

  result = new ModelAndView("announcement/edit");
  result.addObject("announcement", announcement);
  result.addObject("certifications", certifications);
  result.addObject("exams", exams);
  result.addObject("reviewers", reviewers);

  result.addObject("message", messageCode);

  return result;
}
```


This is the core ancillary method. It takes an additional parameter, which is a string that represents the code of a message to be displayed if an error has occurred. The logic in this method isn't difficult at all:

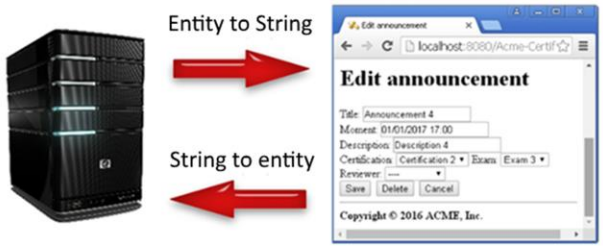
- It first retrieves the certifications in the database by means of the certification service.
- It then checks if the announcement being edited has a certification or not. If it doesn't have any certifications, then the view will get a null certification and a null list of exams; otherwise, the view will get the certification and the collection of exams that is associated with that certification.
- Then the list of all reviewers available is retrieved.
- Once we have the data that we need, the next steps just create a "ModelAndView" object and set the variables in the model to the appropriate values.

Note that you can include a variable called "message" in any view and set it to a message that is defined in an i18n&l10n bundle. This is commonly used to render error messages when an operation cannot be committed.



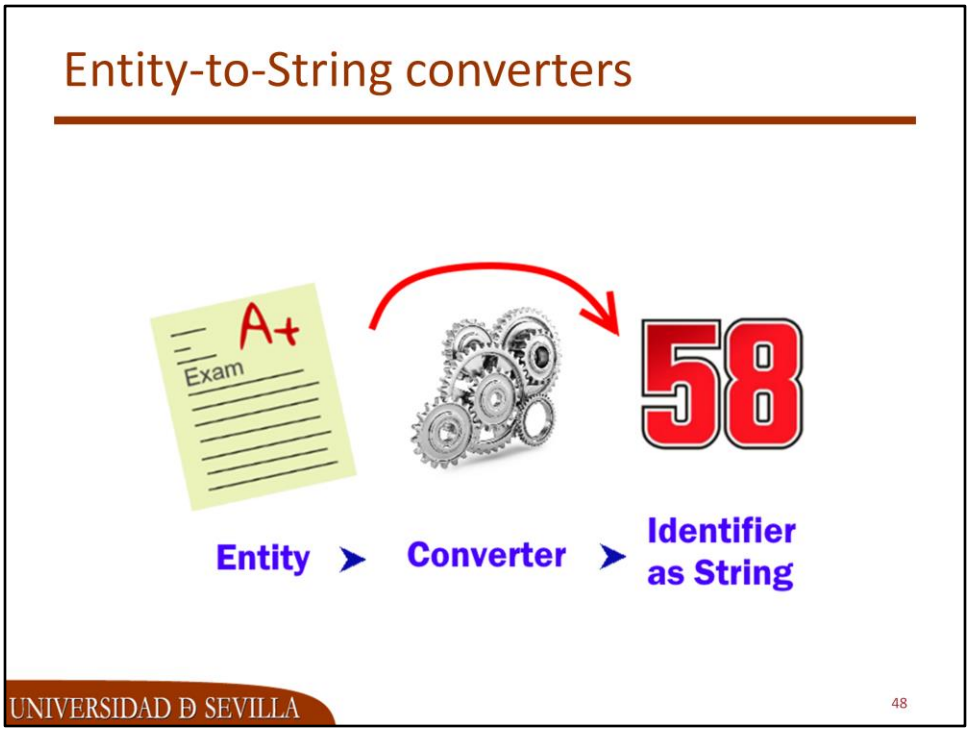
Let's now deal with the converters.

What are converters about?



A converter's a class that helps transform an entity into a string representation of its identifier that's easy to embed into HTML or vice versa

A converter's a class that helps transform an entity into a string representation of its identifier that's easy to embed into HTML or vice versa.



Let's start with the Entity-to-String converters. The idea is very simple: given an object that represents, say, an exam with identifier "58", there must be a converter to transform it into a string that represents this identifier.

Required to encode hidden attributes

```
...
<form:form action="..." modelAttribute="...">
    <form:hidden path="id" />
    <form:hidden path="version" />
    <form:hidden path="exam" />
    ...
</form:form>
...
```

"exam" is a Java object that must be referenced by this hidden attribute.

In this slide, we present an excerpt of a JSP view that provides a form to edit a domain object that involves an exam. Note that the exam is stored in a hidden field because it's not going to be edited, but needs to be stored so that a controller can reconstruct the original object when the form is submitted to the server. We can't serialise the full object that represents the exam here because that would add a lot of useless data to the form; it suffices to store the identifier of the exam. That's why we need a converter.

And selection lists

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

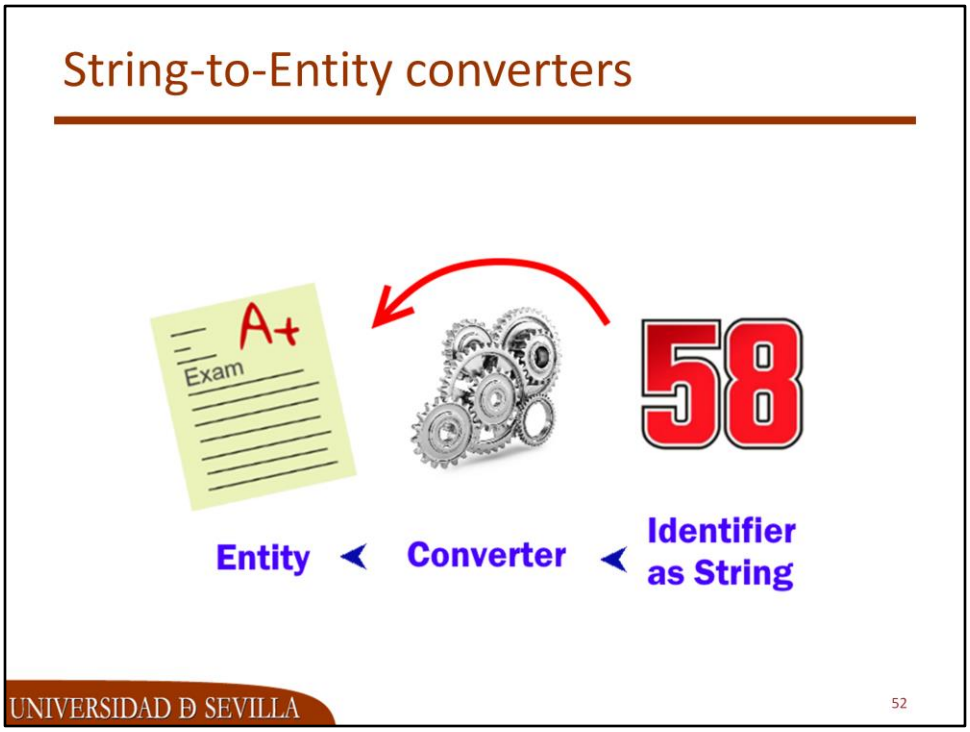
"exams" is a collection of Java objects that must be referenced from this list

In this slide, we present an excerpt of a JSP view that provides a form to edit an announcement. Please, take a look at the drop-down list that is introduced by means of tag "form:select": note that it introduces a number of options that are read from variable "exams"; note, too, that "exams" is a variable that references a collection of exams, that is, Java objects. Obviously, serialising the full objects here doesn't make sense; as was the case before, we just need to reference the exams by means of their identifiers. We also use converters for this task.

The code to the converter

```
@Component
@Transactional
public class ExamToStringConverter
       implements Converter<Exam, String> {

    @Override
    public String convert(Exam exam) {
        String result;

        if (exam == null)
            result = null;
        else
            result = String.valueOf(exam.getId());

        return result;
    }
}
```


In this slide, we show how a typical Java-to-String converter looks like. Just realise that they are implemented as regular classes that have annotations “@Component” and “@Transactional” and implement the Spring “Converter<T1, T2>” interface. Annotation “@Component” requests Spring to instrument this class and annotation “@Transactional” requests it to make its methods transactional. The “Converter<T1, T2>” interface requires the objects that implement it to provide a “convert” method that converts objects of type “T1” into objects of type “T2”. In our projects, “T1” must be an entity of our domain model and “T2” must be “String”. Note that method that implements the conversion is fairly simple: it gets an announcement as a parameter; if it is null, then “null” is returned; otherwise, the identifier is returned as a string. That simple!



Let's now look at the String-to-Entity converters. The idea is very simple: given a string that represents the identifier of an entity, it must consult the database and return that entity.

Required to decode HTTP requests

```
POST /ACME/announcement/edit.do HTTP 1.1
Host: localhost:
Content-Type: application/x-www-form-urlencoded
Content-Length: 118
id=21&version=0&title=Announcement%20X&
moment=12/12/2020%2010:10&description=Description%20X&
certification=123&exam=58
```

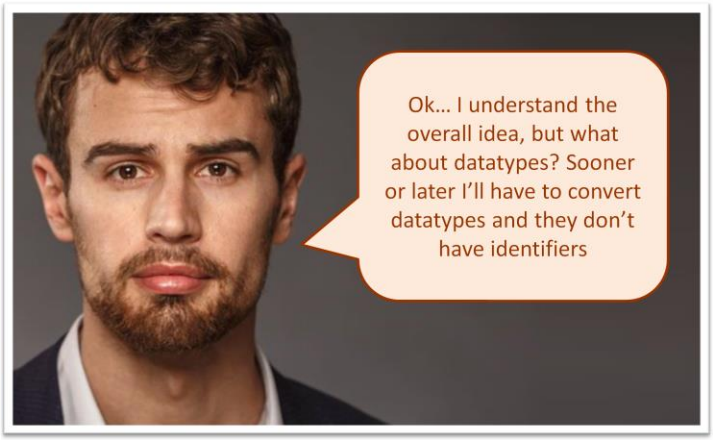

Recall from one of the earliest lessons that submitting an HTML form amounts to making an HTTP POST request to the server. The request encodes the fields of the form as shown in this slide. In this example, we show how a form to edit an announcement looks like when it's submitted to the server. Note that the "title", the "moment", and the "description" properties are encoded using plain strings; note, too, that the "certification" and the "exam" properties are encoded as identifiers. We forget about the properties of the certification or the exam; we just record their identifiers.

The code to the converter

```
@Component
@Transactional
public class StringToExamConverter implements Converter<String, Exam> {

    @Autowired ExamRepository examRepository;

    @Override public Exam convert(String text) {
        Exam result;
        int id;

        try {
            if (StringUtils.isEmpty(text))
              result = null;
            else {
              id = Integer.valueOf(text);
              result = examRepository.findOne(id);
            }
        } catch (Throwable oops) {
            throw new IllegalArgumentException(oops);
        }
        return result;
    }
}
```


In this slide, we show the String-to-Entity converter. As was the case for the previous converter, its definition must be pre-pended by the "@Component" and the "@Transactional" annotations, whose meaning we already know; it must also implement interface "Converter<T1, T2>"; in our systems "T1" is always "String" and "T2" is a domain entity. Recall that this converter has to transform a string with an identifier into an entity, which means that it must perform a query to the database. This is the reason why it declares an autowired exam repository. The code of the "convert" method's quite straightforward: it first checks if the result is null; otherwise, it converts the identifier into an integer number and requests the repository to return the corresponding exam.

This is a very good question



UNIVERSIDAD Ð SEVILLA

55

The overall idea regarding converters is pretty simple, right? But there a good question: what about datatypes? They don't have identifiers, which means that they don't have an identity in the database; their values are embedded in the entities in which they are used.

And this is a very good answer!

The answer's simple: you've got to convert them using a custom method.

A sample datatype-to-string converter

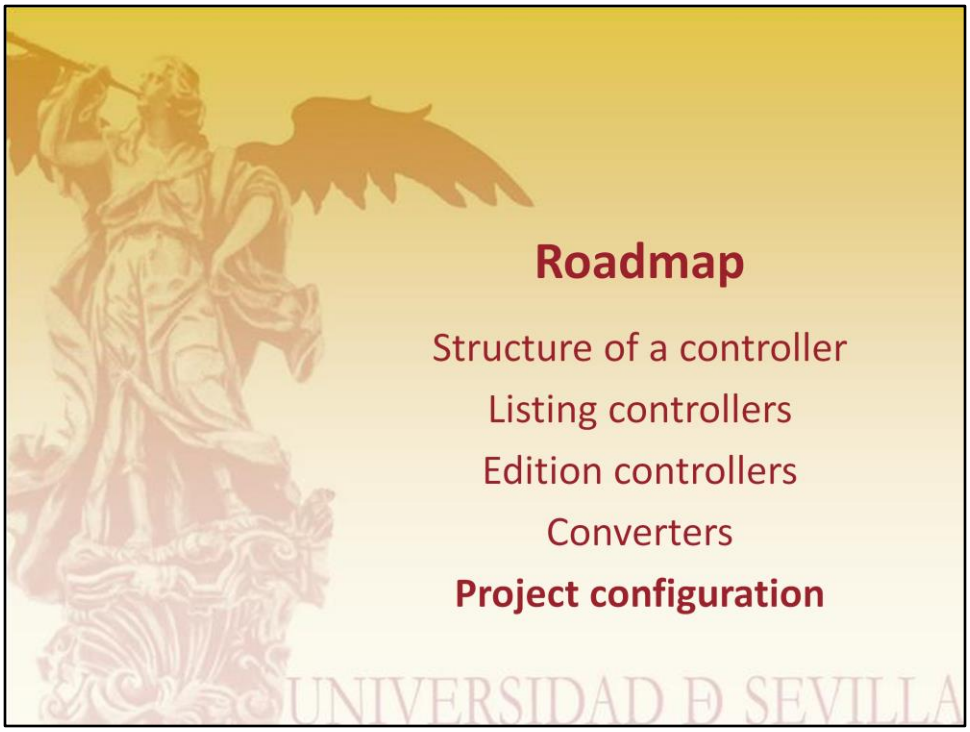
```
@Component @Transactional
public class CreditCardToStringConverter implements Converter<CreditCard, String> {
    @Override
    public String convert(final CreditCard creditCard) {
        String result;
        StringBuilder builder;

        if (creditCard == null)
            result = null;
        else {
            try {
                builder = new StringBuilder();
                builder.append(URLEncoder.encode(creditCard.getName(), "UTF-8"));
                builder.append("|");
                builder.append(URLEncoder.encode(creditCard.getBrand(), "UTF-8"));
                builder.append("|");
                builder.append(URLEncoder.encode(creditCard.getNumber(), "UTF-8"));
                builder.append("|");
                builder.append(URLEncoder.encode(Integer.toString(creditCard.getCvv()), "UTF-8"));
                result = builder.toString();
            } catch (final Throwable oops) {
                throw new RuntimeException(oops);
            }
        }

        return result;
    }
}
```


In this slide, we show a sample datatype-to-string converter that transforms credit cards into strings. It gets a credit card as input; if it is “null”, then it returns “null”; otherwise, it uses a string builder to transform it into a string of the form “name|brand|number|number|cvv”, where each part is URL encoded to prevent interpretation problems. For instance, an object like “CreditCard{name = “Pedro Pérez”, brand = “VISA|2”, number = “4111111111111111”, cvv=123}” is converted into string “Pedro%20P%C3%A9rez|VISA%7C2|4111111111111111|123”.

A sample string-to-datatype converter

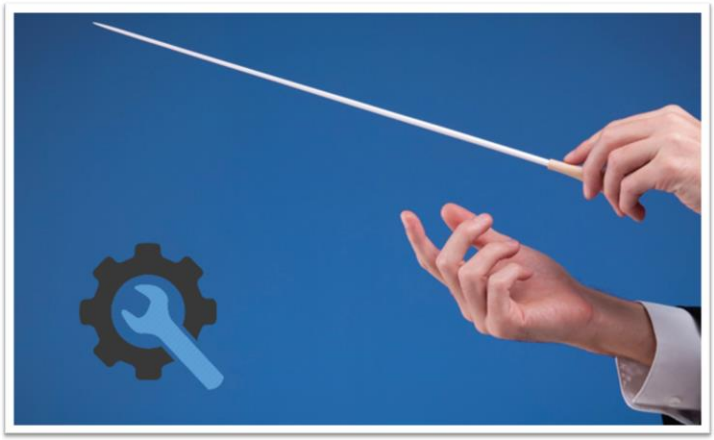
```
@Component @Transactional
public class StringToCreditCardConverter implements Converter<String, CreditCard> {
    @Override
    public CreditCard convert(final String text) {
        CreditCard result;
        String parts[];

        if (text == null)
            result = null;
        else
            try {
                parts = text.split("\\|");
                result = new CreditCard();
                result.setName(URLDecoder.decode(parts[0], "UTF-8"));
                result.setBrand(URLDecoder.decode(parts[1], "UTF-8"));
                result.setNumber(URLDecoder.decode(parts[2], "UTF-8"));
                result.setCvv(Integer.valueOf(URLDecoder.decode(parts[3], "UTF-8")));
            } catch (final Throwable oops) {
                throw new RuntimeException(oops);
            }

        return result;
    }
}
```


In this slide, we show the counterpart, that is, a converter that takes a string as input and reconstructs a credit card from it. It simply splits the input string using the vertical bar as a separator, and then reconstructs a credit card from the resulting parts. For instance, given string "Pedro%20P%C3%A9rez|VISA%7C2|4111111111111111|123", it returns object "CreditCard{name = "Pedro Pérez", brand = "VISA|2", number = "4111111111111111", cvv=123}".



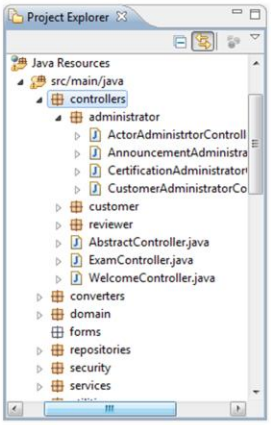
Finally, it's time to report on how you have to configure your project so that your controllers and your converters are properly recognised by Spring; we'll also report a little on how to control de access to specific URLs in your system.

Configuring controllers



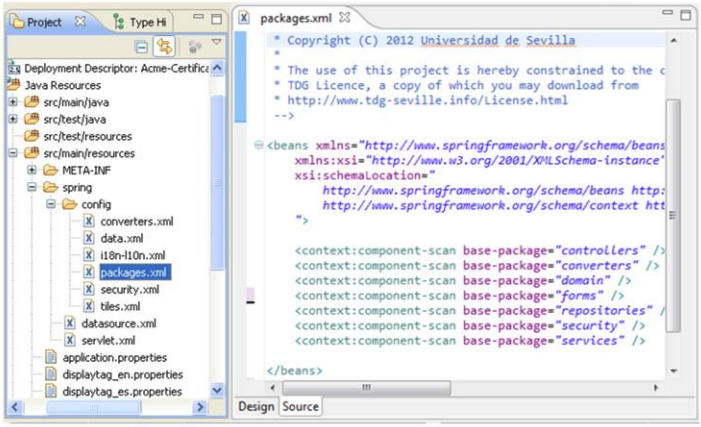
Let’s start with configuring controllers.

The `controllers` package



You must add your controllers to a package called “controllers”. Please, note that the abstract controller and controllers that serve public URLs are listed directly in this package, whereas the remaining are grouped into sub-packages that are named after the corresponding entities. Fortunately, that’s enough to use them; but, please, keep reading.

The packages.xml configuration file

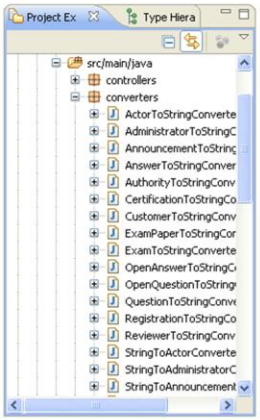


Please, take a look at a configuration file called “packages.xml”. This file specifies the packages that Spring must scan for controllers and other components: converters, domain entities, forms (which we haven’t used thus far), repositories, security artefacts, and services. Just learn that we need to list the names of the packages where our components reside, not the individual names of the controllers, as was the case for converters.

Configuring converters

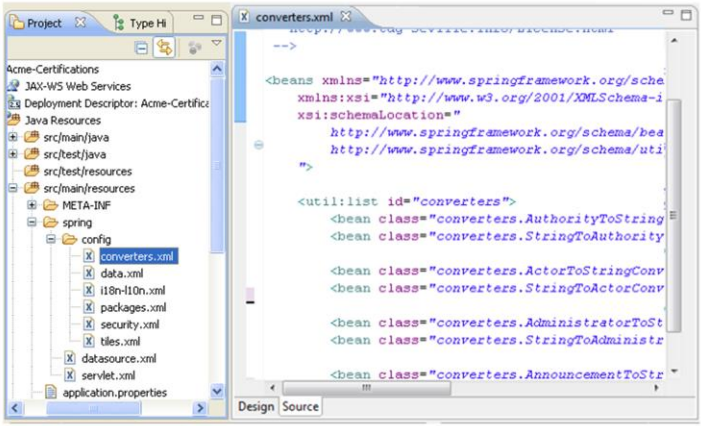
Let’s now learn how to configure converters.

The converters package



You must add your converters to a package called “converters”. Unfortunately, this isn’t enough; please, keep reading.

The converters.xml file

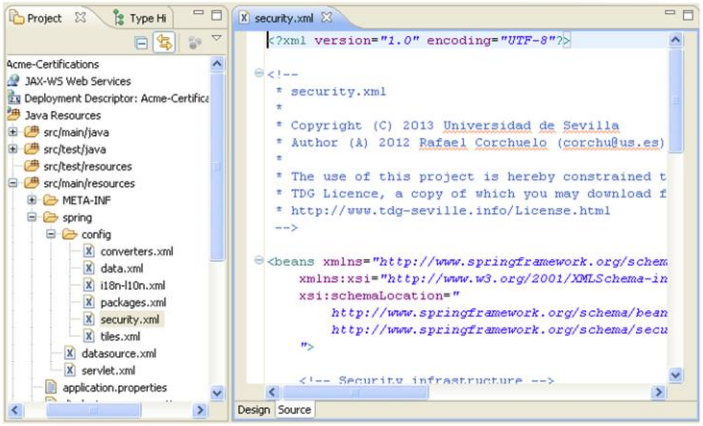


There is a configuration file called “converters.xml” in which you must explicitly list them all, one after the other.

Configuring URL access control

Please, recall that controllers serve HTTP requests that are targeted to a number of URLs. Obviously, you don't want everybody to have access to a URL like "http://www.acme.com/announcement/administrator/edit.do?announcementId=99" ; only administrators should be able to have access to a URL like this. The last step to configure your controllers is to configure the URL access control.

The `security.xml` configuration file



To configure the URL access control, you have to edit a file called “security.xml”. This file has several parts that we analyse in the following slides.

Access control (I)

```
<security:http auto-config="true" use-expressions="true">
  <security:intercept-url pattern="/" access="permitAll" />
  <security:intercept-url pattern="/favicon.ico" access="permitAll" />
  <security:intercept-url pattern="/images/**" access="permitAll" />
  <security:intercept-url pattern="/scripts/**" access="permitAll" />
  <security:intercept-url pattern="/styles/**" access="permitAll" />
  <security:intercept-url pattern="/views/misc/index.jsp"
                          access="permitAll" />
  <security:intercept-url pattern="/security/login.do"
                          access="permitAll" />
  <security:intercept-url pattern="/security/loginFailure.do"
                          access="permitAll" />
  …
</security:http>
```


Then comes a long section in which you must configure the access control to your URLs. We've split this section into three parts to ease the presentation. The first part's shown in this slide. It simply declares that every user has access to a number of predefined URLs:

- "/": this is the root path in your application.
- "/favicon.ico": this is the favicon of your application.
- "/images/**": this refers to the images in your application.
- "/scripts/**": this refers to the ECMA Scripts on which your application relies.
- "/styles/**": this refers to the CSS style files of your application.
- "/views/misc/index.jsp": this is the JSP document that redirects your application to the welcome screen.
- "/security/login.do" and "/security/loginFailure.do": these are the URLs that handle authentication and authentication failures.

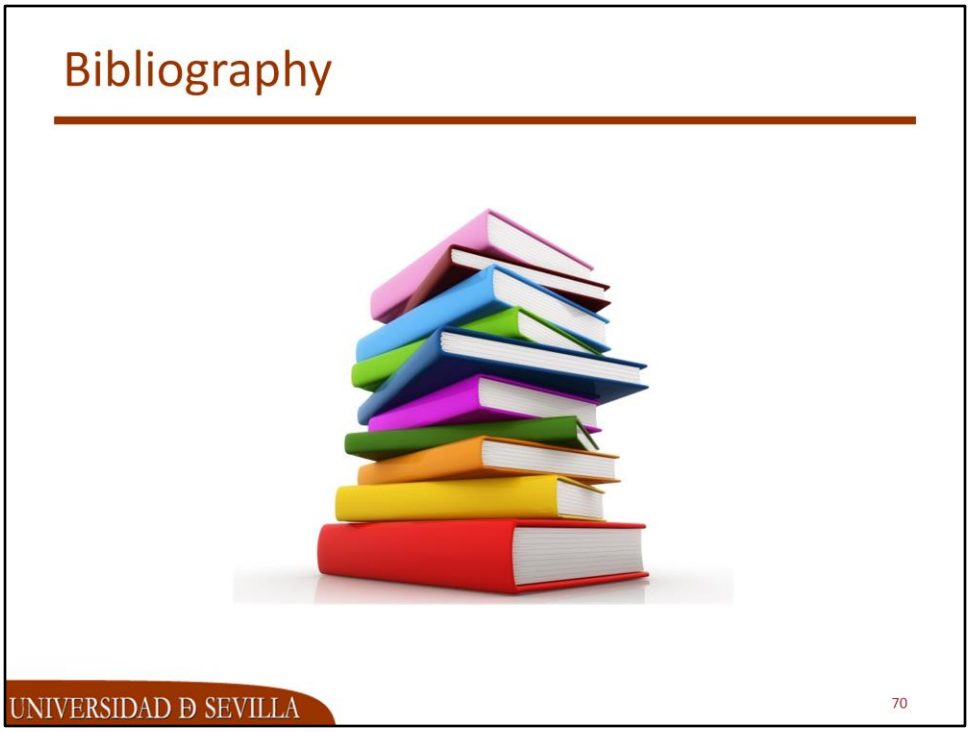
Access control (II)

```
<security:http auto-config="true" use-expressions="true">
  …
  <security:intercept-url pattern="/actor/administrator/**"
                          access="hasRole('ADMIN')" />
  <security:intercept-url pattern="/announcement/administrator/**"
                          access="hasRole('ADMIN')" />
  <security:intercept-url pattern="/announcement/customer/**"
                          access="hasRole('CUSTOMER')" />
  <security:intercept-url pattern="/certification/administrator/**"
                          access="hasRole('ADMIN')" />
  <security:intercept-url pattern="/certification/customer/**"
                          access="hasRole('CUSTOMER')" />
  …
  <security:intercept-url pattern="/**" access="hasRole('NONE')" />
  …
</security:http>
```


The second part deals with application specific URLs, like the ones in this slide. For instance, we specify that only users with authority "ADMIN" can have access to URLs like "/actor/administrator/**", "/announcement/administrator/**", and the like. We don't think you should have any problems at interpreting these elements. We'd only like to attract your attention to the last element: note that we must list every URL explicitly and state the access control we wish to enforce. It's highly recommended that the last element be the following one:

```
<security:intercept-url pattern="/**" access="hasRole('NONE')" />
```

This element prevents every user from having access to a URL that hasn't been listed previously. That is, you need to list every URL explicitly or, otherwise, no-one will have access to them. This policy is very desirable for security reasons.



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, 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/.

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.

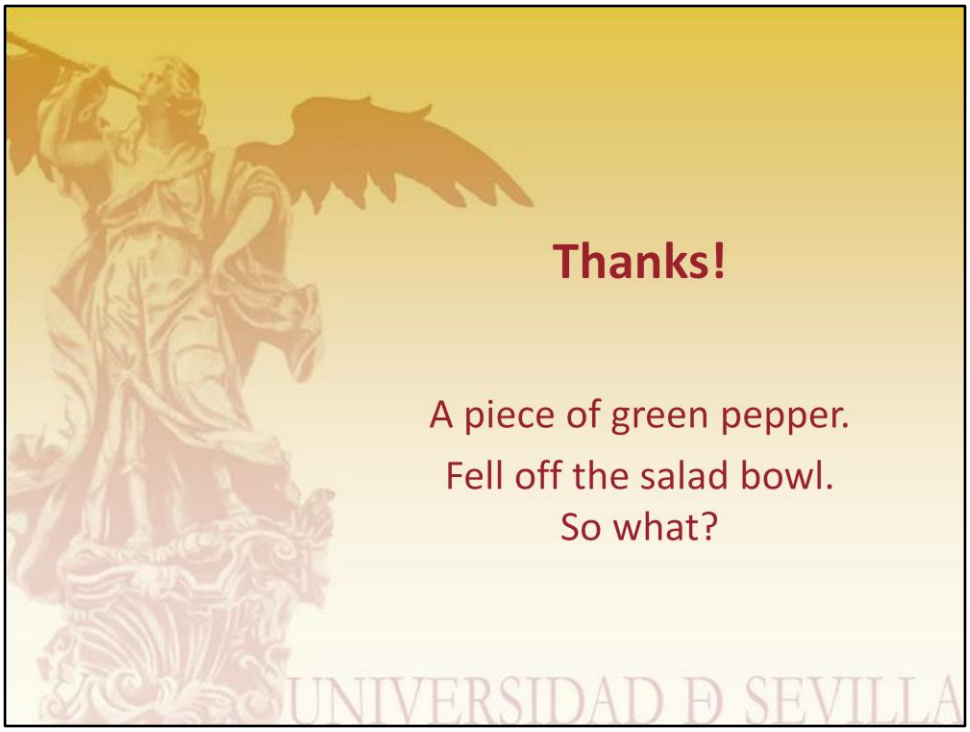
The next lecture

- 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

UNIVERSIDAD Ð SEVILLA

72

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.



Thanks for attending this lecture!