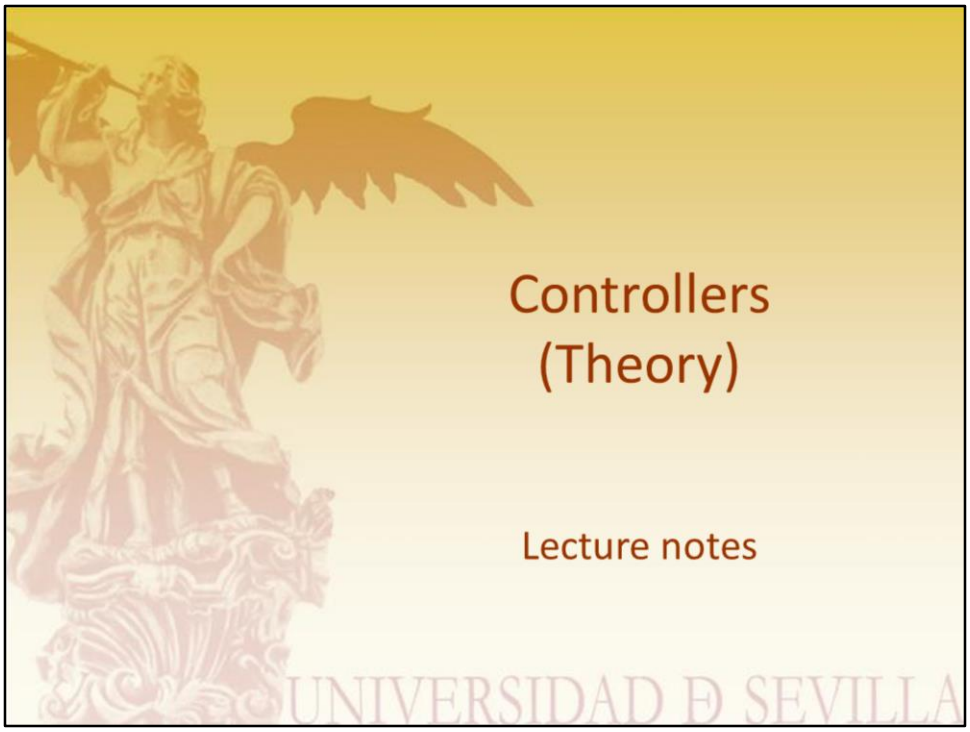


Welcome to this lecture! Our goal's to provide you with the theory that you require to implement controllers.

--

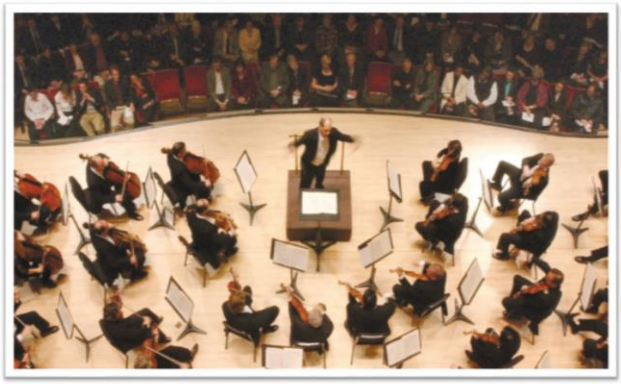
What are controllers?

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 they co-ordinate some services to do so.

How are they devised?

UNIVERSIDAD Ð SEVILLA

4

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

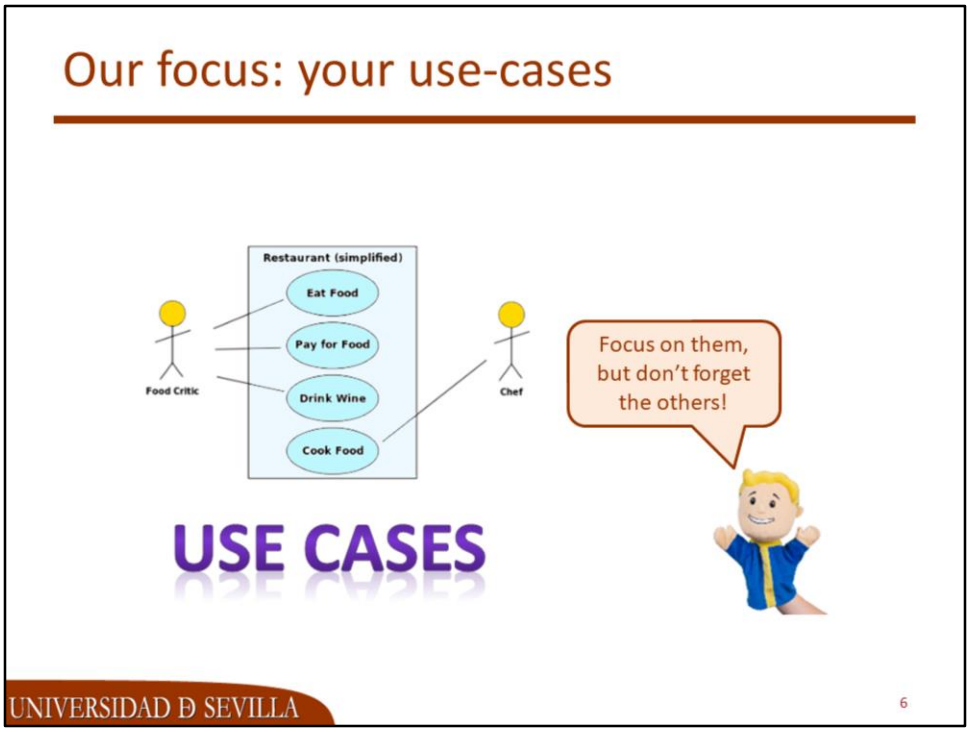
Starting point: your requirements

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

UNIVERSIDAD Ð SEVILLA

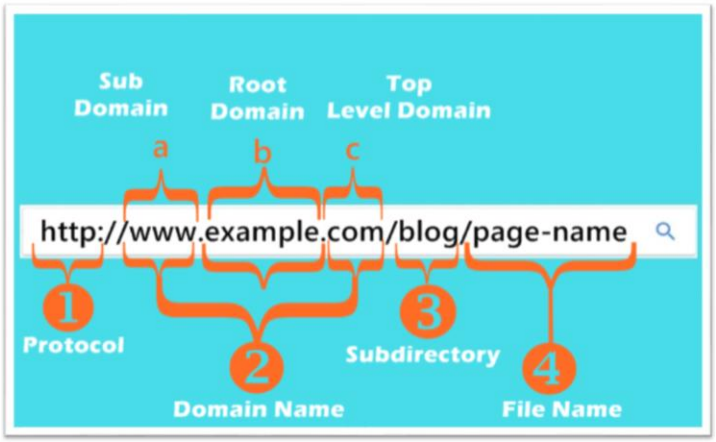
5

We don’t think this should be surprising at all now: the starting point to devise 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 the structure of URLs

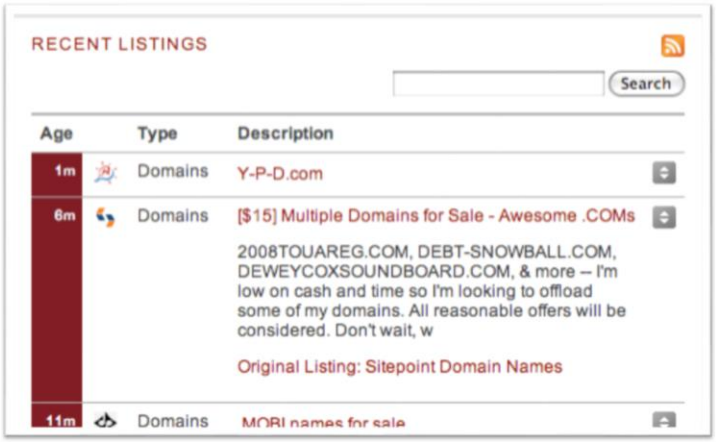


UNIVERSIDAD Ð SEVILLA

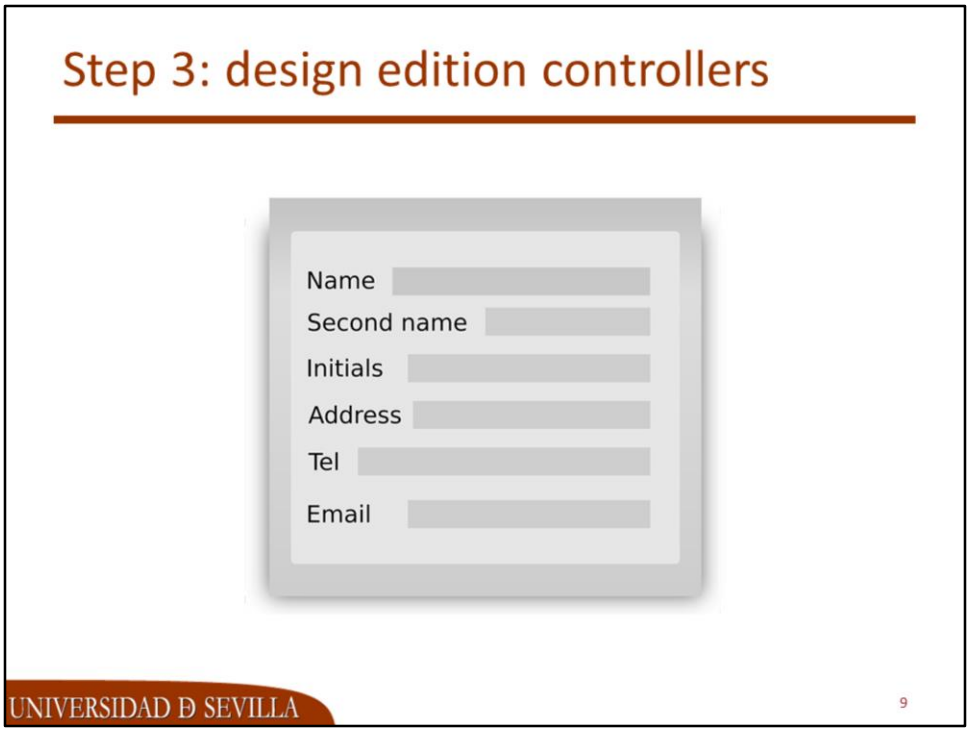
7

The first step is to understand the structure of the URLs of the requests that your controllers must orchestrate.

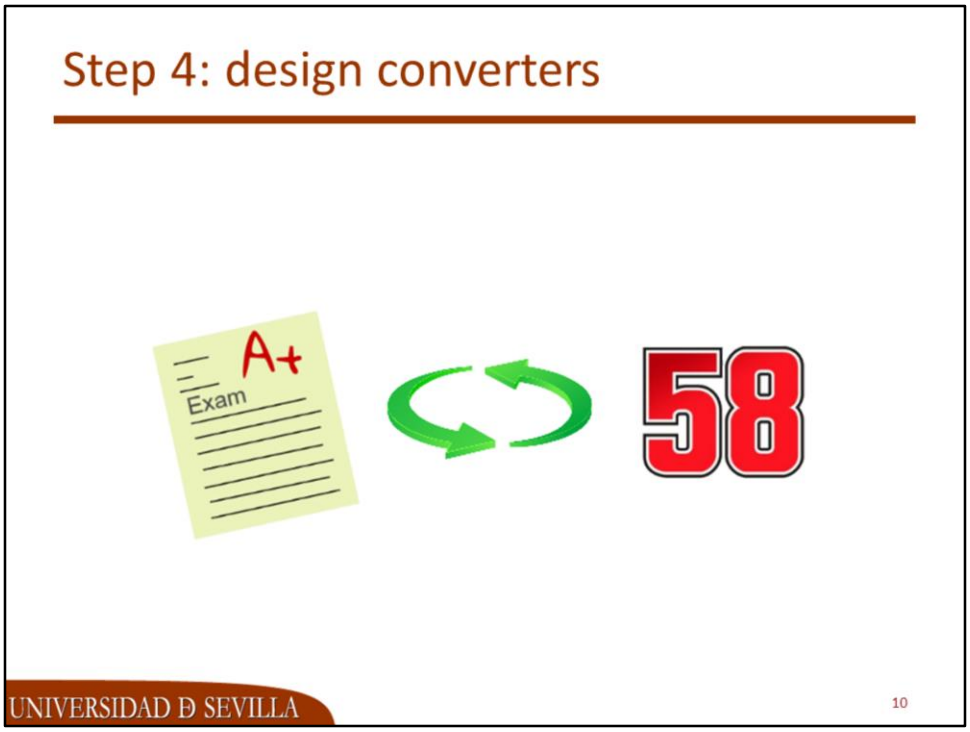
Step 2: design listing controllers



The next step is to design your listing controllers.



And then the edition controllers.



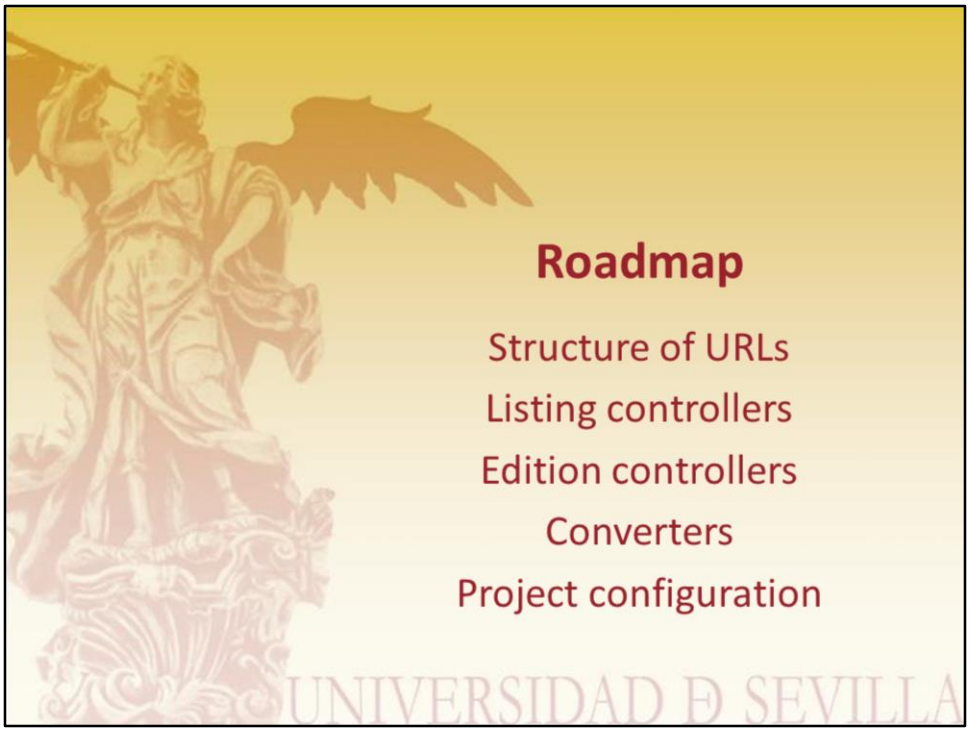
The fourth step consists in creating a number of converters, which basically care of converting objects into strings and vice versa. We mentioned these components in the first lesson, but we haven't resorted to them so far; in this lesson, they play a very important role.

Step 5: configure your project

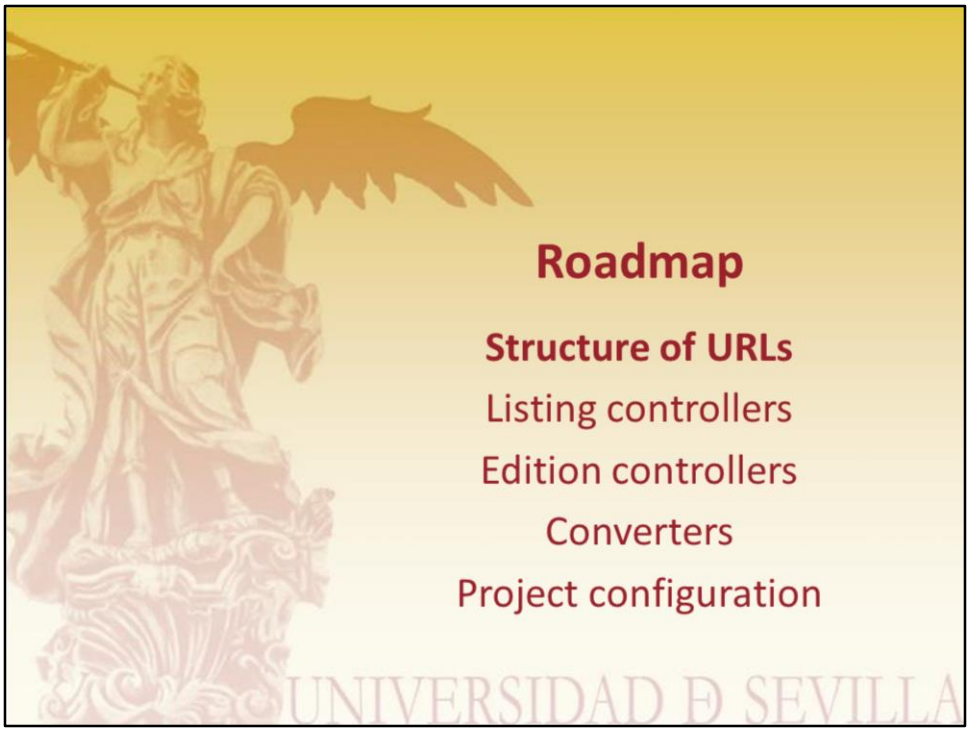
UNIVERSIDAD Ð SEVILLA

11

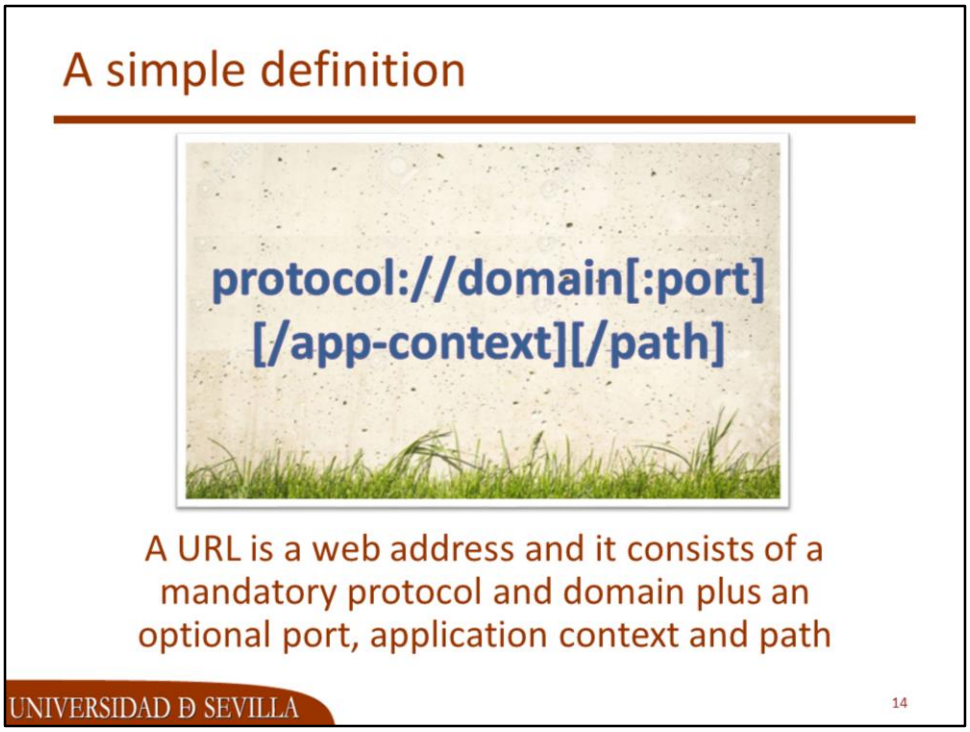
And the final step is to configure your project so that it can handle your controllers.



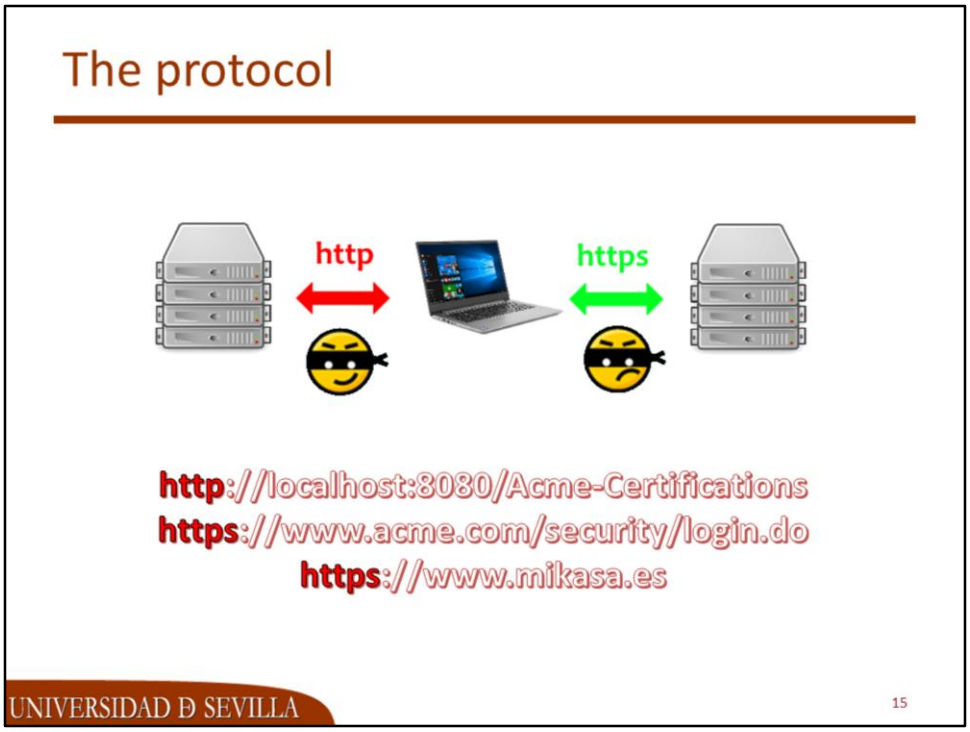
So, this is our roadmap for today's lecture. We'll first present the structure of the URLs, then on how to design a listing controller, then on how to design an edition controller, next on converters and, finally, on how to configure your project.



Let’s start reporting on the structure of the URLs.



This slide shows a simple definition: a URL is a web address and it consists of a mandatory protocol and domain plus an optional port, application context and path. Read on to learn about each component.



The first part of the URL is the protocol. In a web information system it's either HTTP or HTTPS. HTTP is the default protocol, which is appropriate as long as the data exchanged with the server is public; in other words: you don't care if a third party can see them. HTTPS is the secure version of HTTP; this protocol encrypts the data that are transmitted so that the communications are confidential, which makes it the choice for requests that involve passwords, orders, personal data, and the like. For instance, "http://localhost:8080/Acme-Certifications/" is a URL that uses the plain HTTP protocol, "https://www.acme.com/security/login.do" uses the secure HTTPS protocol, and "https://www.mikasa.es" uses the HTTPS protocol, as well.

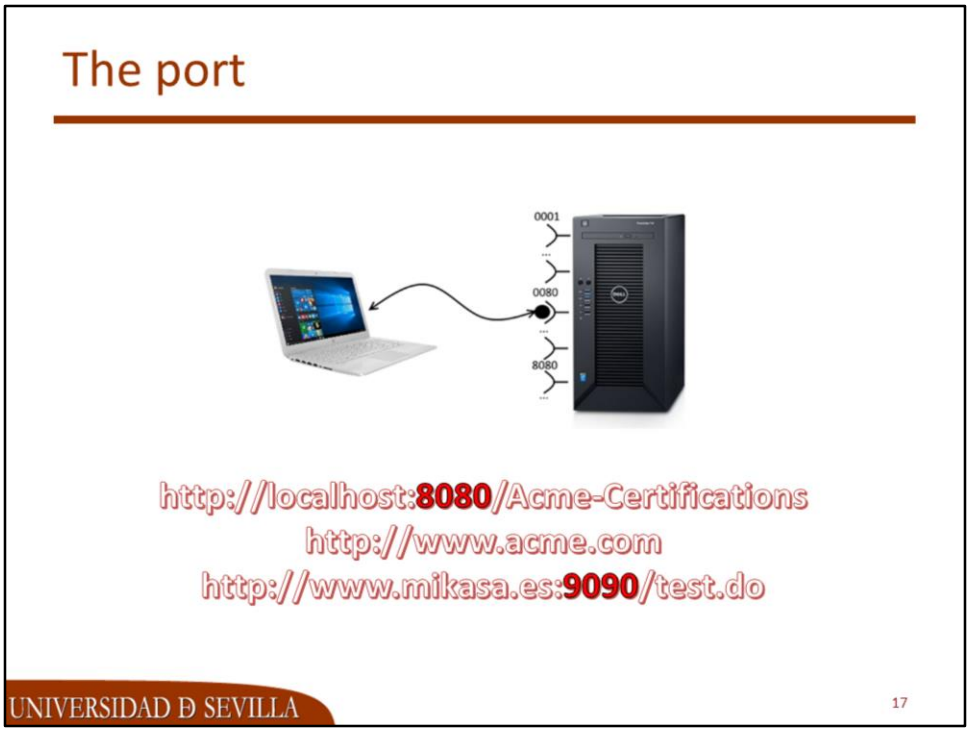
The domain

http://**localhost**:8080/Acme-Certifications
http://**www.acme.com**/images/logo.png
https//**www.mikasa.es**

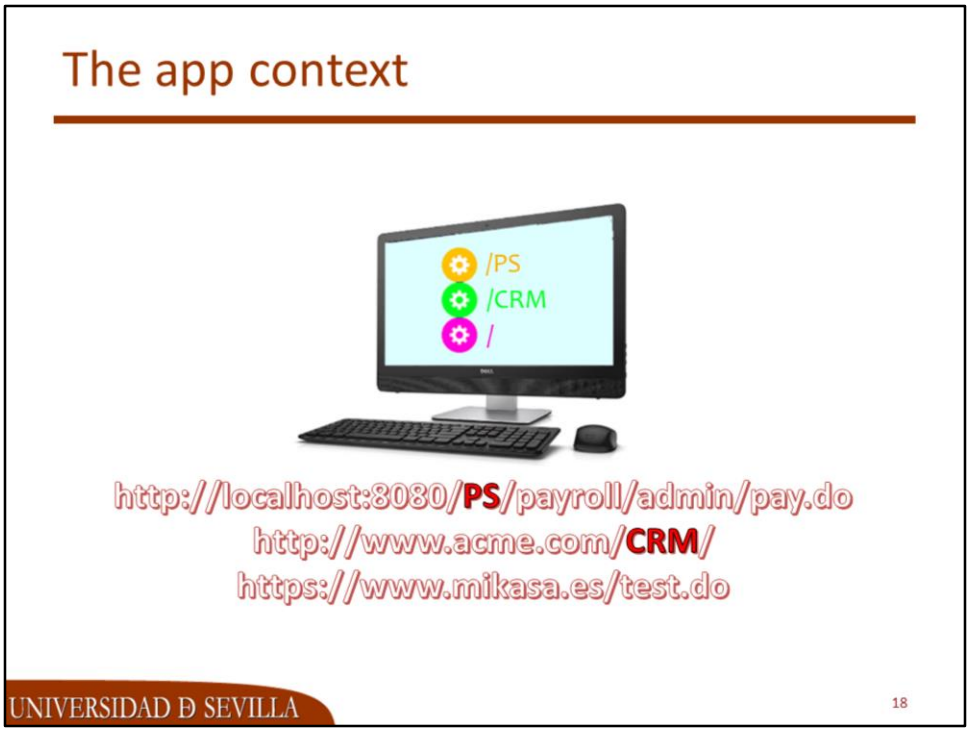
UNIVERSIDAD Ð SEVILLA

16

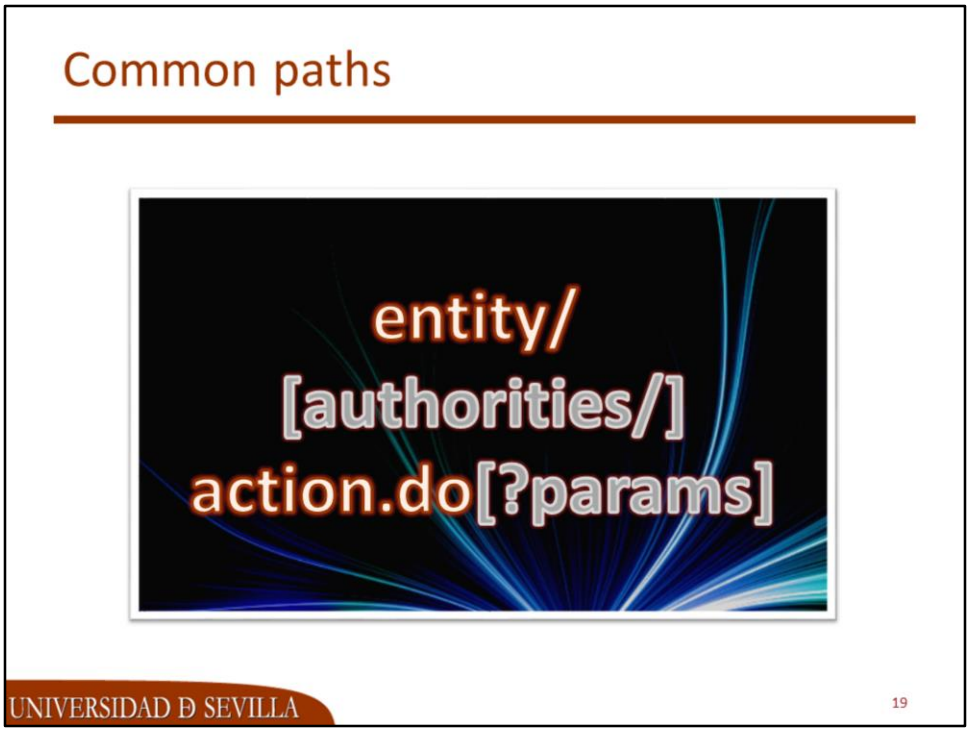
The domain is a symbolic name for the IP address of your server. For instance, the domain is “localhost” in URL “http://localhost:8080/Acme-Certifications”, it’s “www.acme.com” in URL “http://www.acme.com/images/logo.png”, and it’s “www.mikasa.es” in URL “http://www.mikasa.es”. In this subject, we use “localhost” for development purposes and “www.acme.com” to simulate an actual customer’s domain.



The port refers to the TCP/IP port used to transfer data from or to your application server. By default, it's "8080" in development environments and "80" in production environments; some companies use port "9090" for testing purposes in production environments. There's no difference; in theory, any port might be used, but "8080" and "80" are quite standardised; port "9090" is also frequent, but far from a standard. For instance, "http://localhost:8080/Acme-Certifications" requests the welcome page through port "8080", "http://www.acme.com" requests it through port "80", and "http://www.mikasa.es:9090/test.do" requests document "test.do" through port "9090". Note that port "80" is the default port, so "http://www.acme.com" and "http://www.acme.com:80" are the same URL.

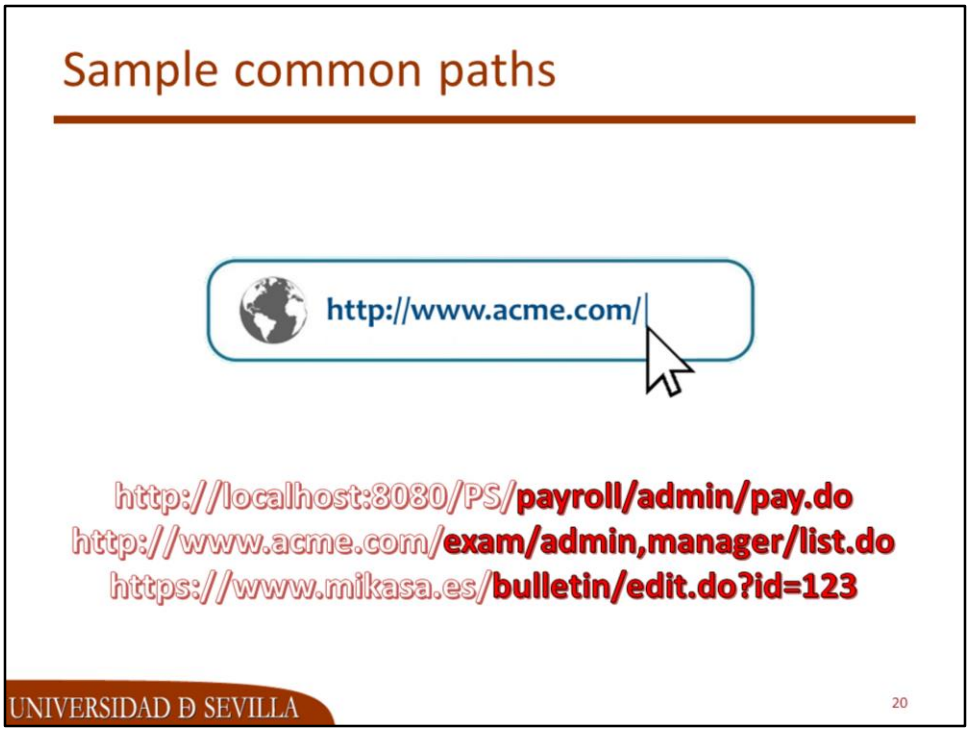


The application context helps install several applications on the same server and make independent requests to each of them. The default application context depends on your working environment, namely: when working on a developer's environment, it's typically the name of your project; when working on a production environment, it's typically the root. For instance, "http://localhost:8080/PS/payroll/admin/pay.do" refers to an application context called "PS" (Payroll System), "http://www.acme.com/CRM" refers to an application context called "CRM" (Customer Relationship Management), and "https://www.mikasa.es/test.do" refers to the root application context.

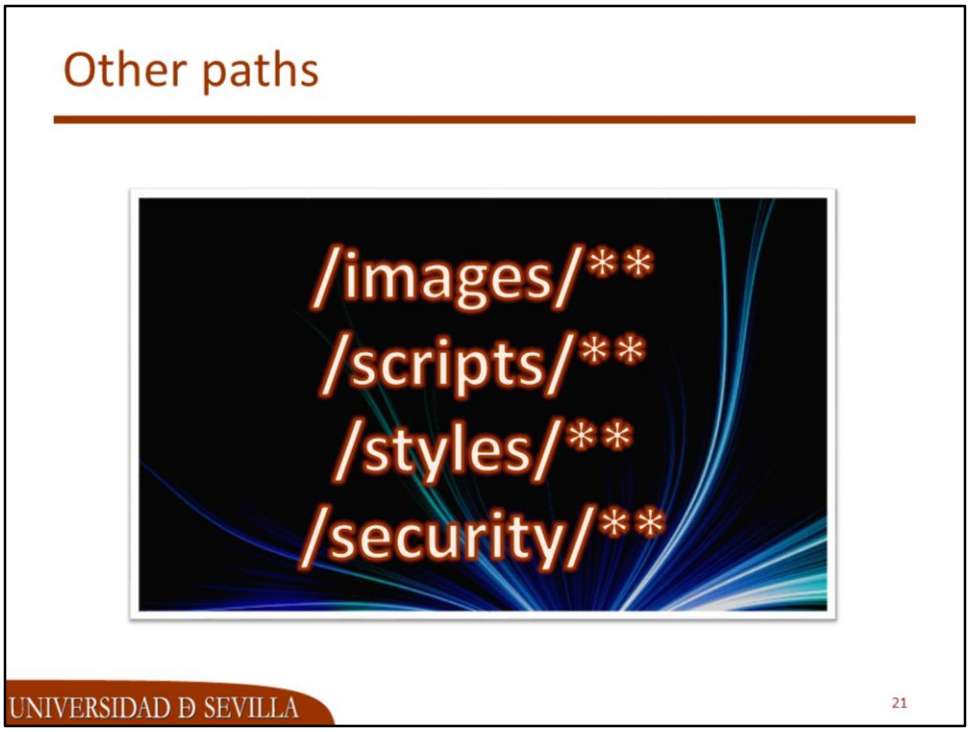


In this slide, we show the structure of most common paths. They are composed of the following parts:

- "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": this refers to an action to be performed, e.g., "list.do" or "create.do".
- "?params": this refers to some optional parameters, e.g., "?id=23" or "?deadline=2019-01-30T23:59:00.000".

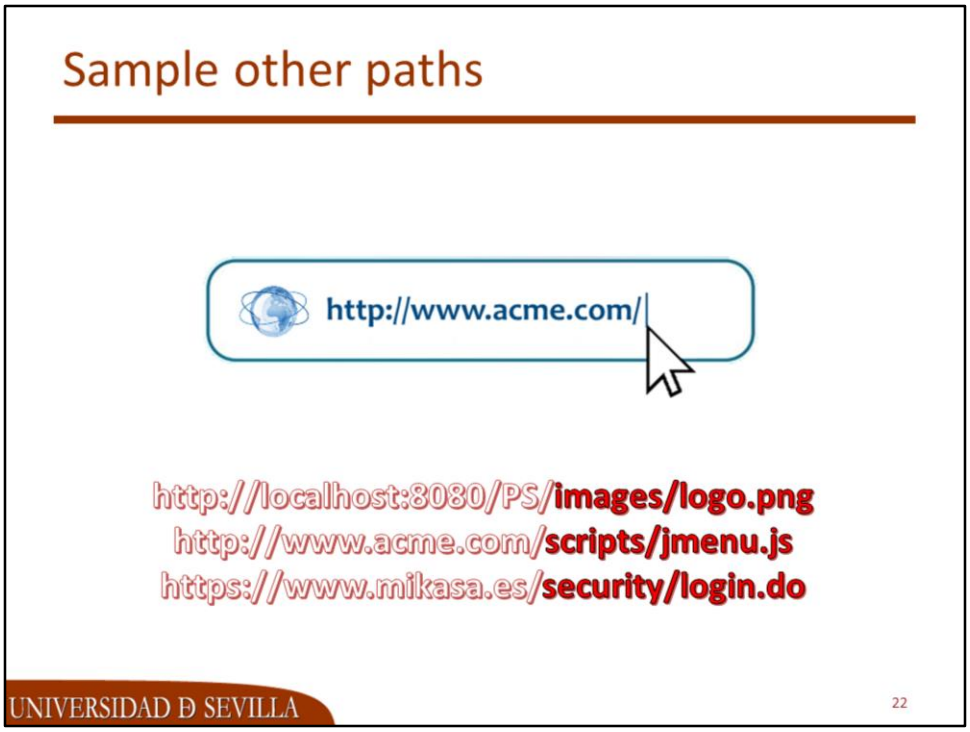


In this slide, we show some URLs with a few common path examples. For instance, “http://localhost:8080/PS/payroll/admin/pay.do” requests action “pay.do” on an entity named “payroll”, which is available to users with role “admin”; URL ”http://www.acme.com/exam/admin,manager/list.do” requests action “list.do” on an entity called “exam”, which is available to users with role “admin” or “manager”; finally, URL “https://www.mikasa.es/bulletin/edit.do?id=123” requests action “edit.do” on an entity called “bulletin” whose identifier is “123” (note that every user may request this action).

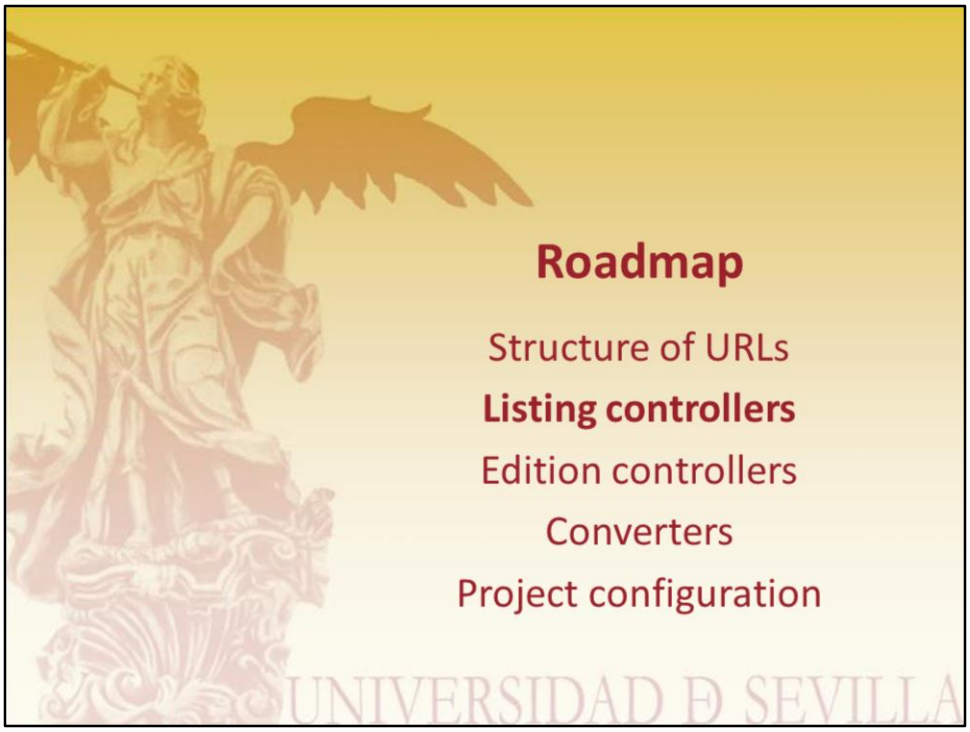


There are a few paths that do not adhere to the pattern in the previous slide, namely:

- "/images/**": it allows to retrieve the images of the system.
- "/scripts/**": it allows to retrieve the Java scripts;
- "/styles/**": it allows to retrieve the CSS styles;
- "/security/**": it provides a couple of pages to log in or to show login errors.

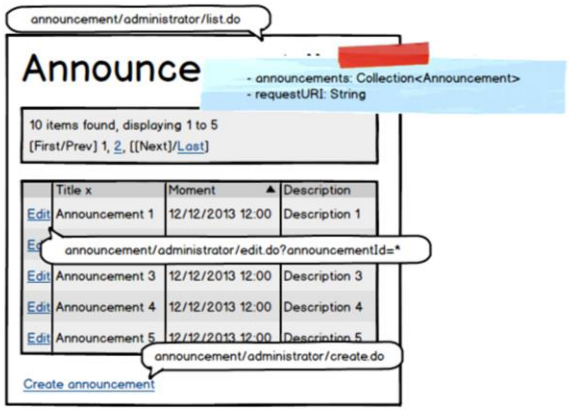


In this slide, we show some URLs with a few other path examples. For instance, URL ”http://localhost:8080/PS/images/logo.png” requests an image called “images/logo.png”; URL “http://www.acme.com/scripts/jmenu.js” requests script “scripts/jmenu.jsp”; and URL “https://www.mikasa.es/security/login.do” requests action “security/login.do”.



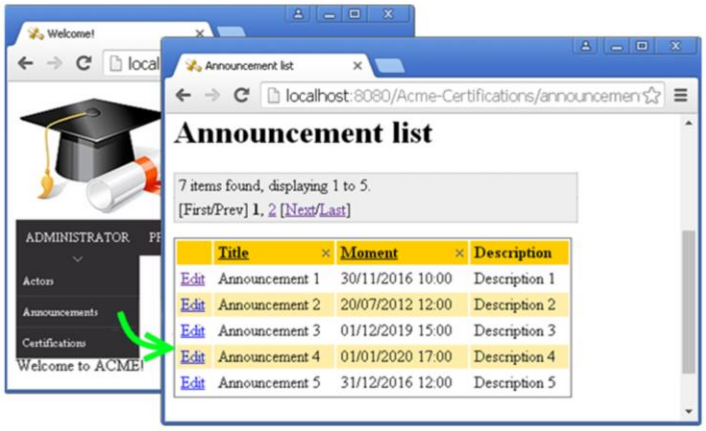
That's enough about URLs. Let's now report on the listing controllers.

The listing view specification

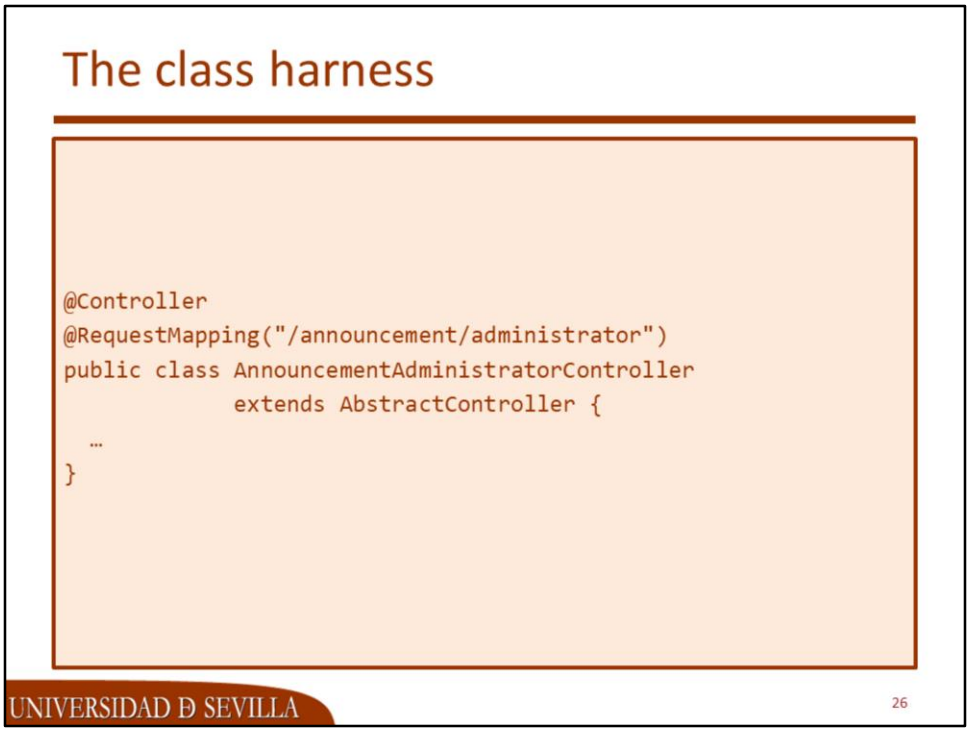


To start working, we need a mock-up with its corresponding model and link specification.

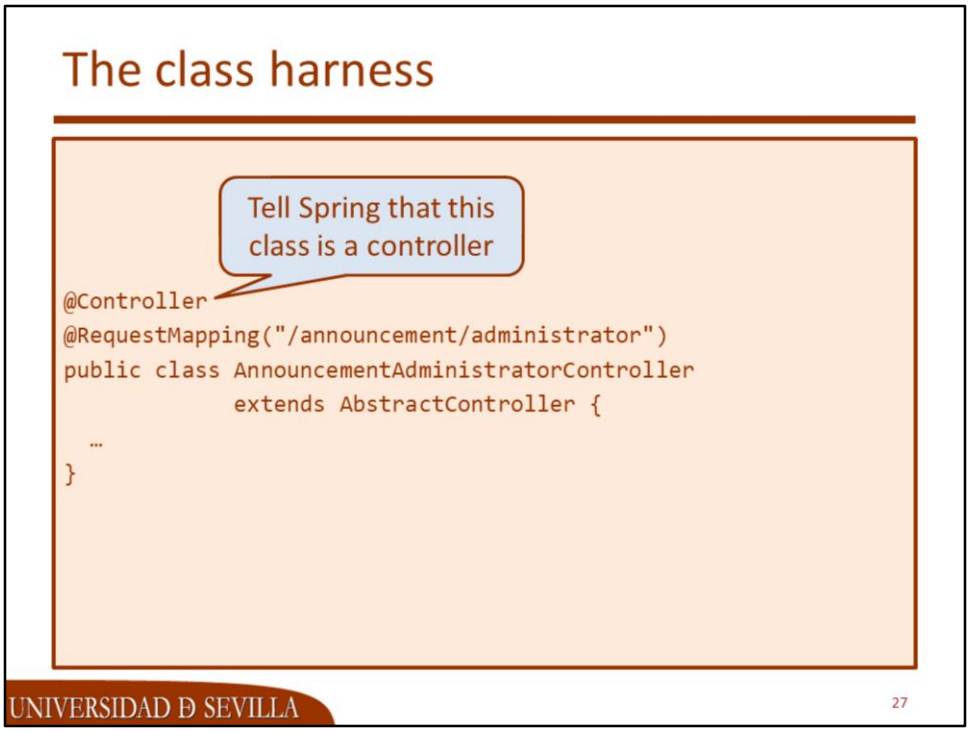
This is what it looks like



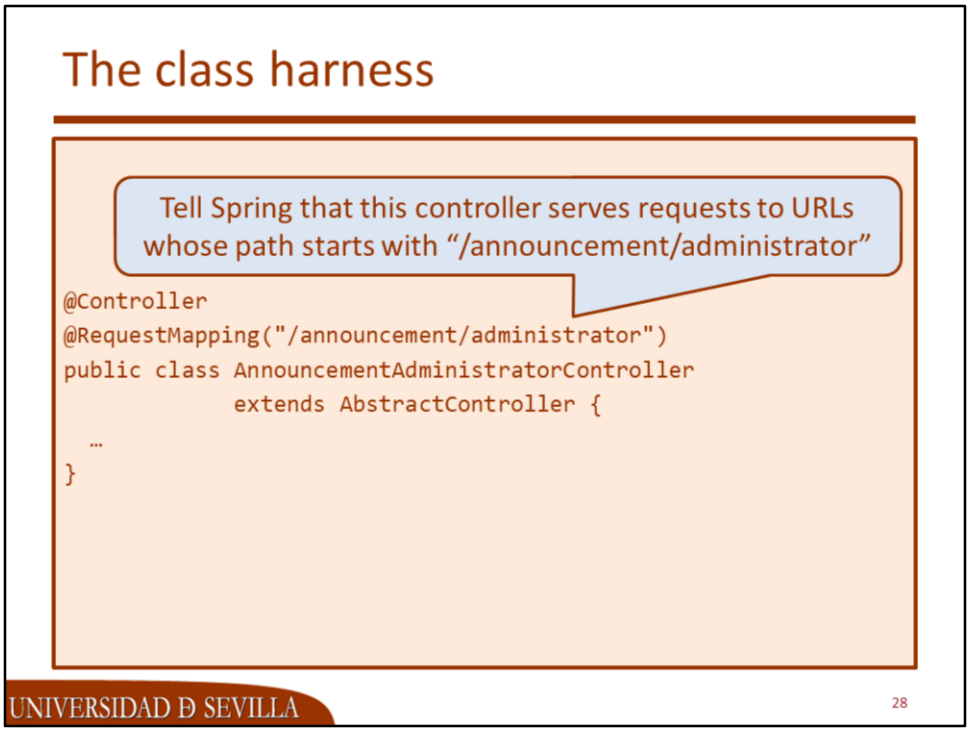
The listing controller must be invoked every time a user needs to display the listing of announcements. Note that there's a pagination bar on top of the grid, that you may sort the grid by clicking on the first or the second columns, and that each announcement has an "Edit" link; there's also a "Create announcement" link at the bottom. Nice, isn't it?



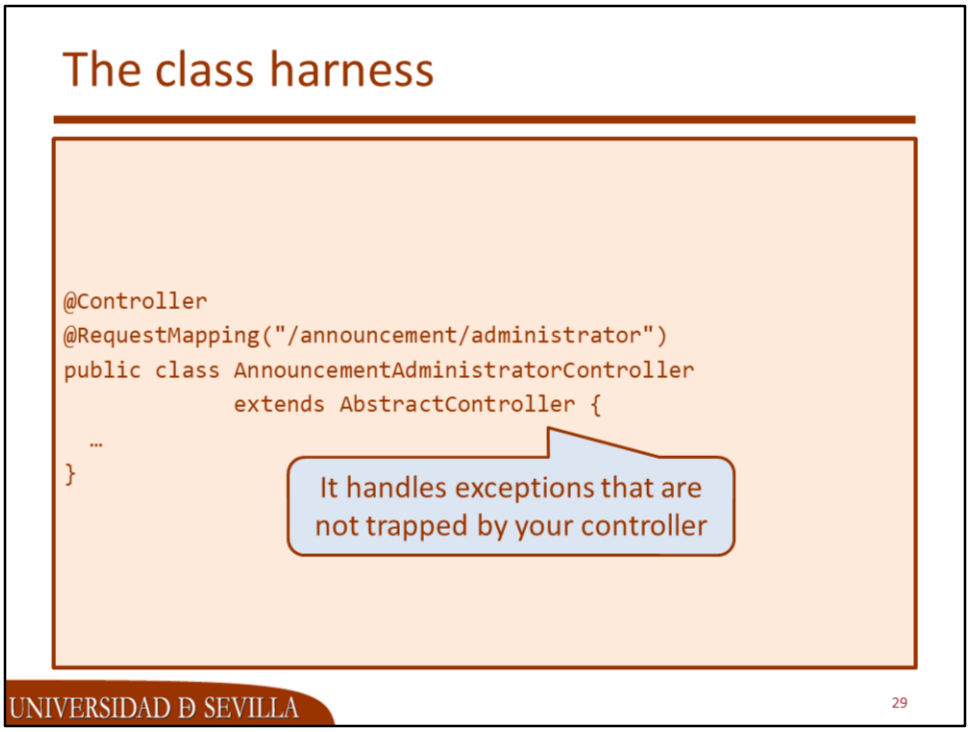
To implement the controller, we have to write a regular class with a number of features that we explain in the following slides.



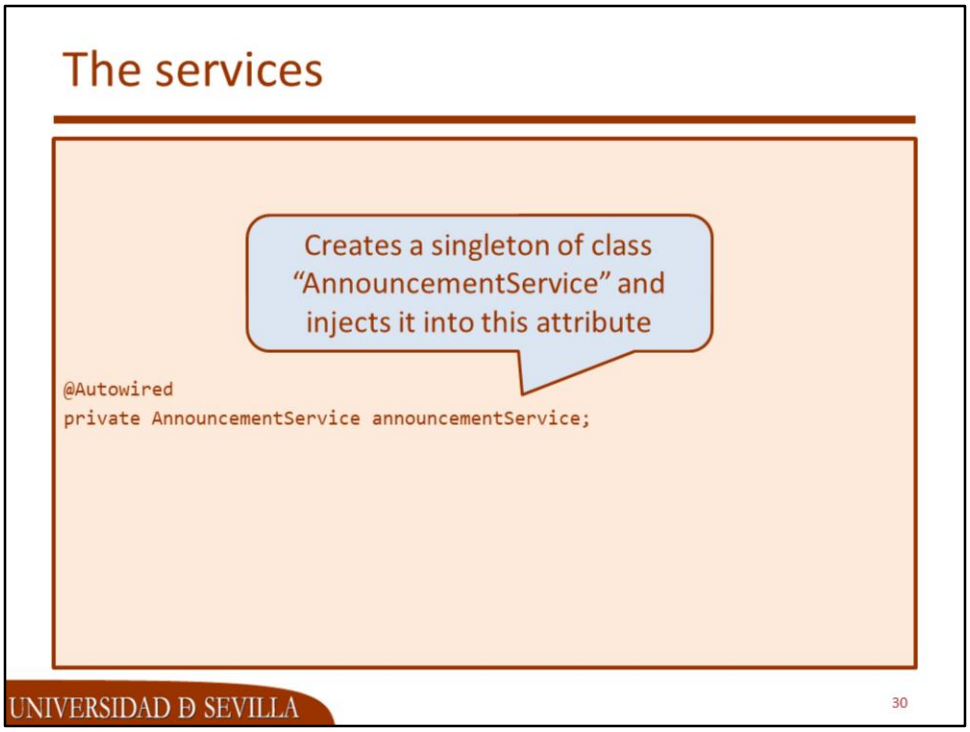
The controller class must have annotation "@Controller", which tells Spring that this class implements a controller. As was the case for repositories and services, Spring instruments this class so as to inject the "@Autowired" dependencies and to create a single object.



The next annotation's "@RequestMapping", which lets Spring know about the prefixes of the paths that this controller serves. In our example, it serves requests to URLs regarding announcements that are handled by administrators; for instance: "http://localhost:8080/Acme-Certifications/announcement/administrator/list.do" or "https://www.acme.com/announcement/administrator/edit.do?announcementId=123".



The controller class must also extend class “AbstractController”, which handles exceptions that are not trapped by your controller.

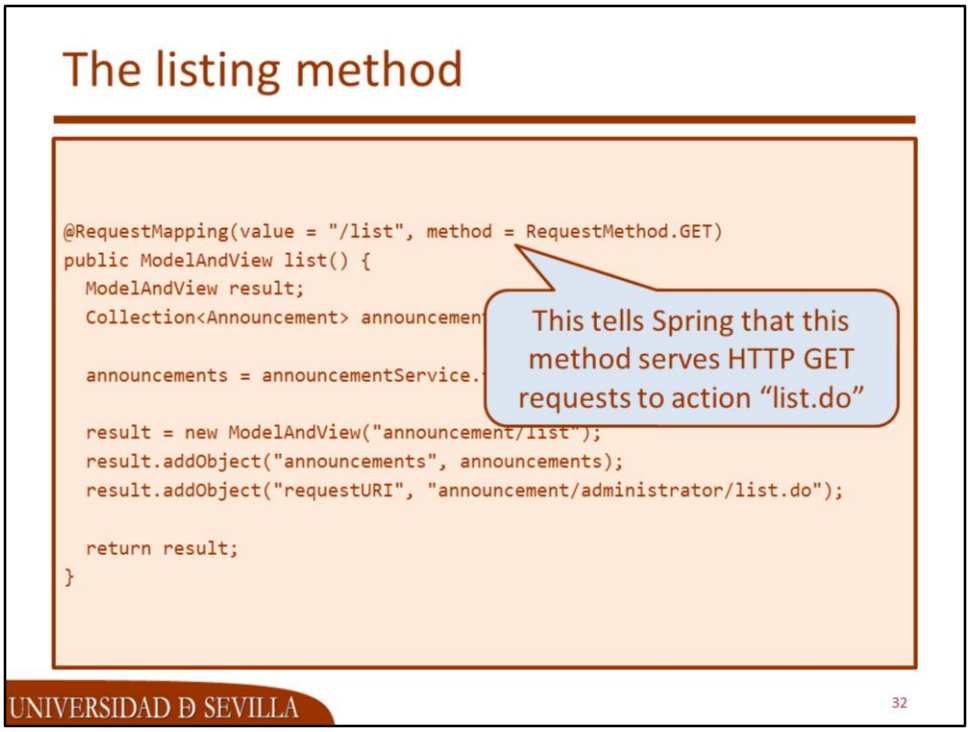


Controllers do typically require some services to implement their logic. They are included by declaring some private attributes that hold references to the services. Note that, we need to put an "@Autowired" annotation in front of each attribute so that Spring injects the appropriate instances into them.

The listing method

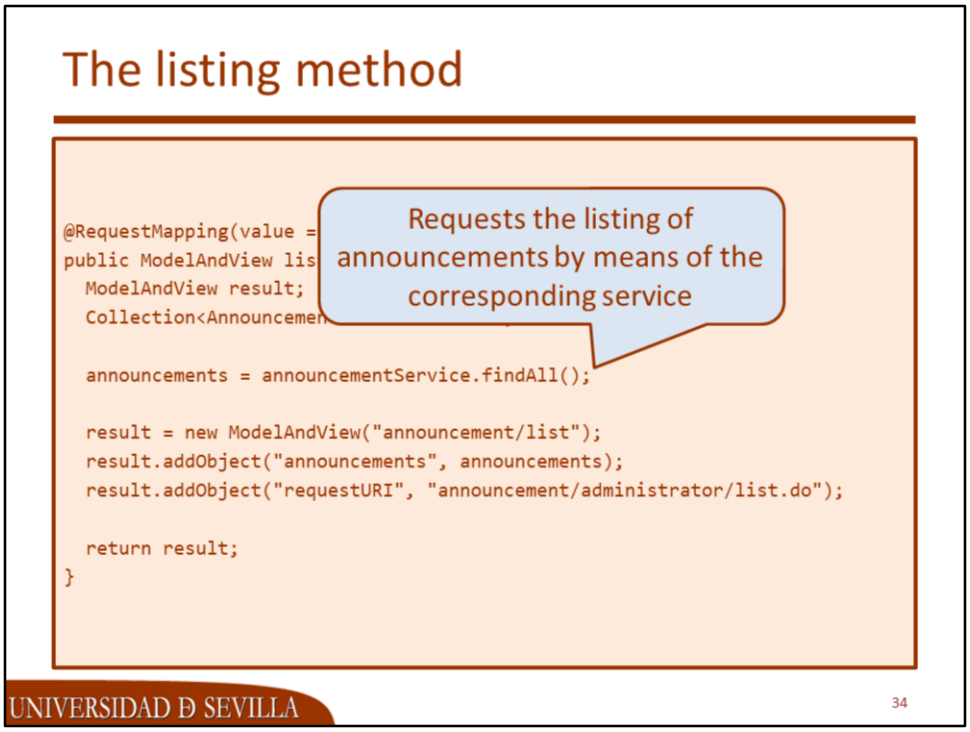
```
@RequestMapping(value = "/list", method = RequestMethod.GET)
public ModelAndView list() {
  ModelAndView result;
  Collection<Announcement> announcements;

  announcements = announcementService.findAll();

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

  return result;
}
```


And, finally, it comes the method that implements the listing controller.

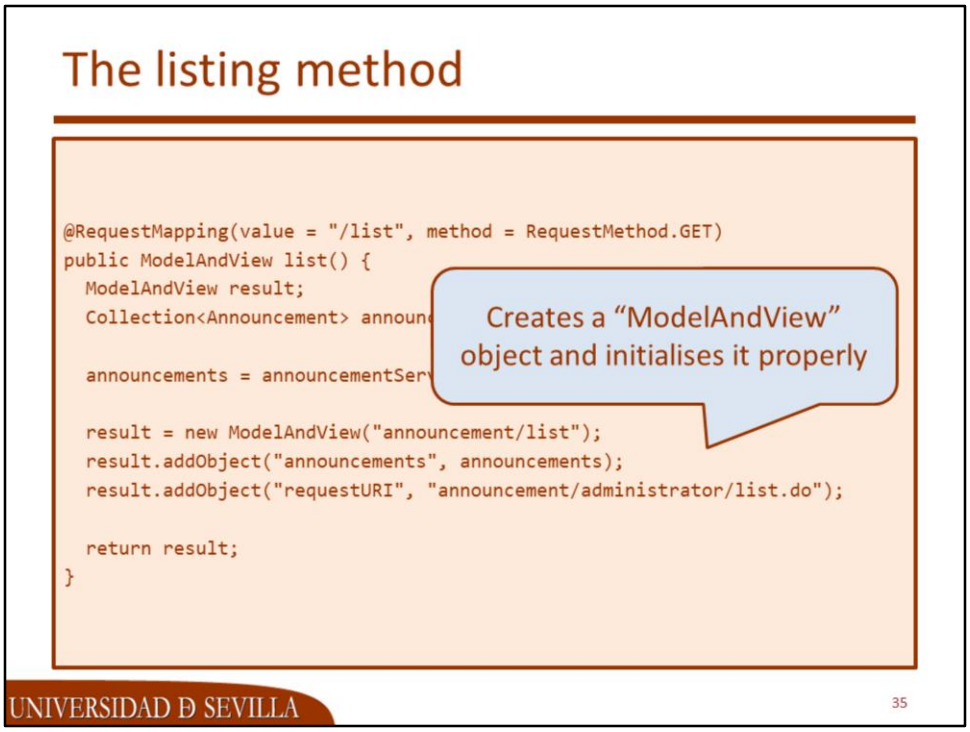


Note that it must be prepended with annotation "@RequestMapping", which tells Spring that this method serves HTTP GET requests to action "list.do".

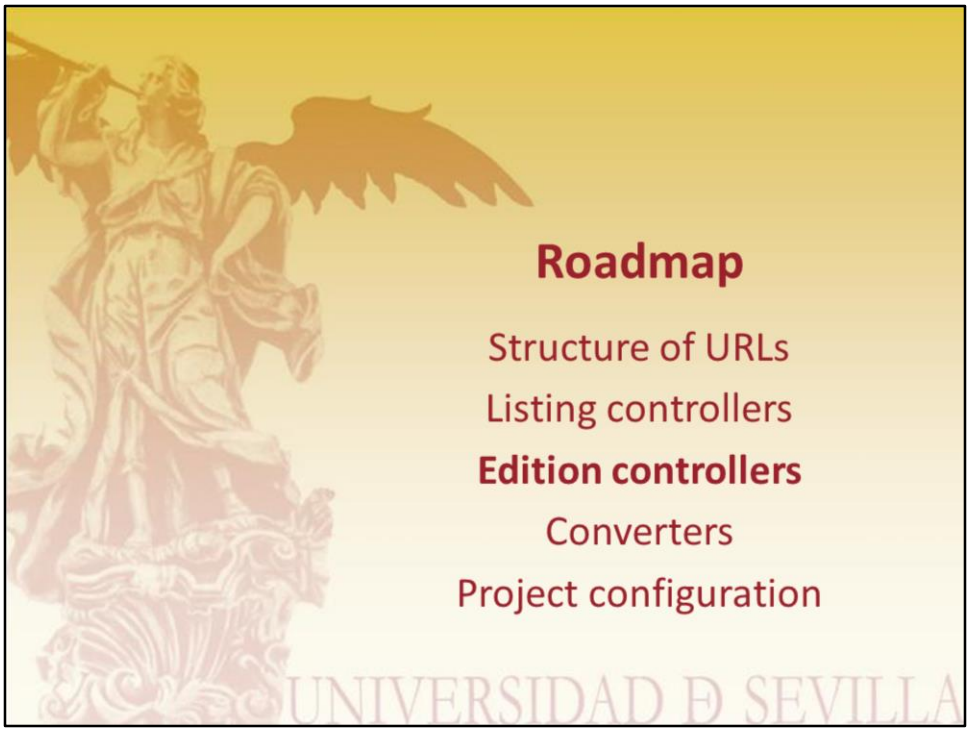
Then comes the definition of the method. Note that it doesn't get any parameters and returns an object of class "ModelAndView", which, obviously embeds a model that maps some variables onto their corresponding values and a view to render them.



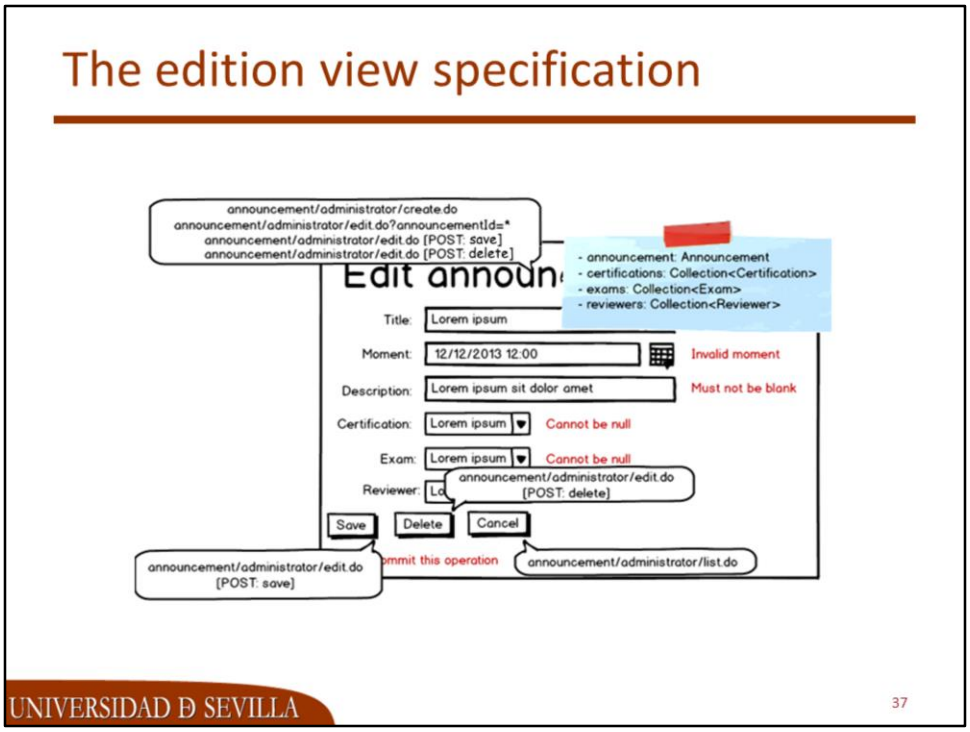
The business logic is very simple. First, the method resorts to the announcement service to retrieve all of the announcements in the database.



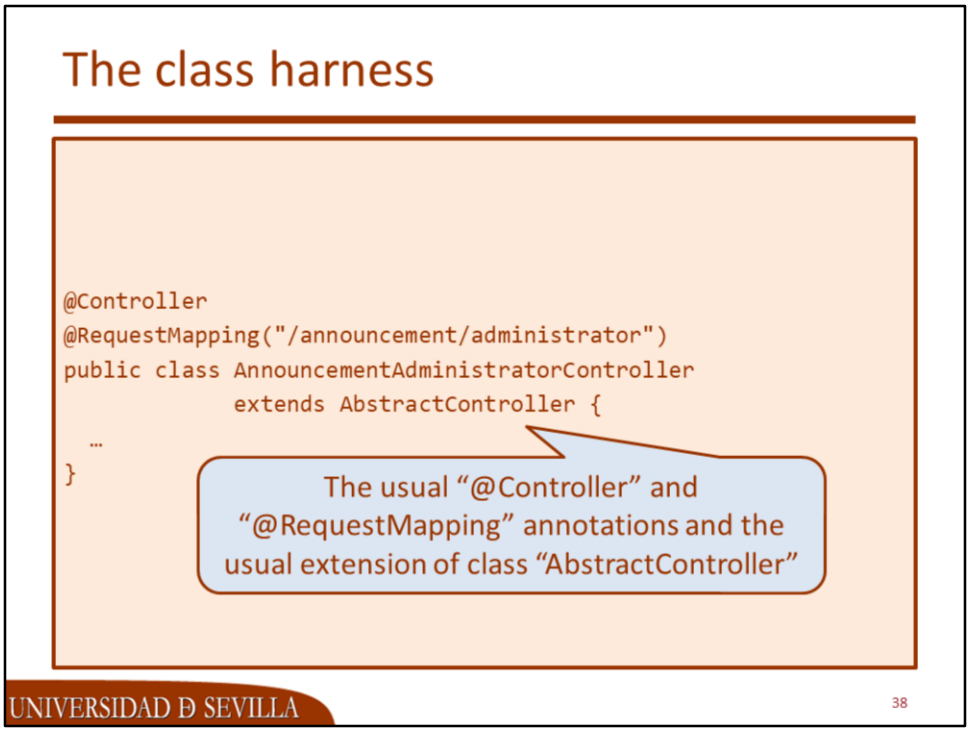
Next, it creates a "ModelAndView" object. Note that the constructor of this class gets a string parameter in which you have to specify the name of the view that must be instantiated. We call that view "announcement/list". The following lines simply instantiate the variables in the model by calling method "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".



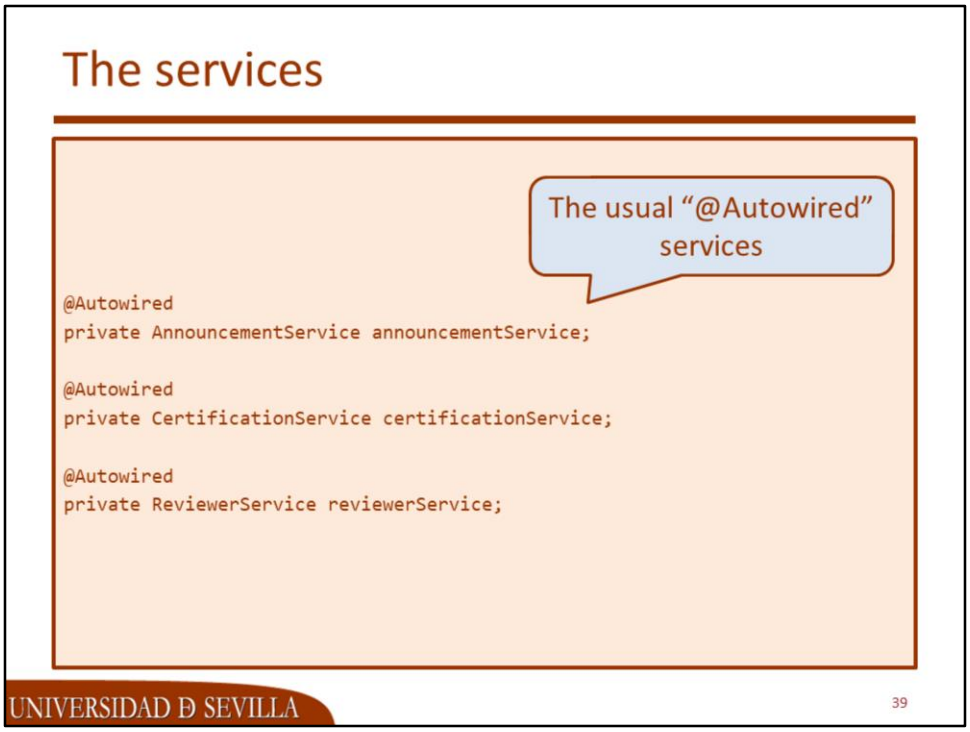
Listing controllers were not difficult at all, were they? Let's now report on the edition controllers, which are a little more involved.



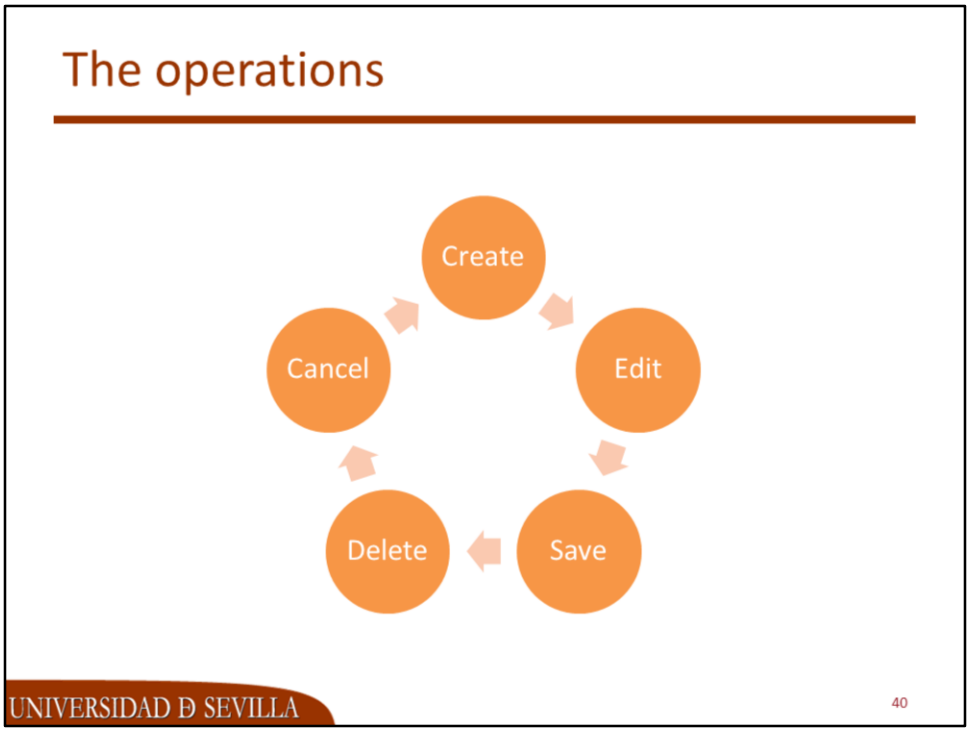
This slide shows the view specification that we've designed to edit an announcement, including the model, and the link specification.



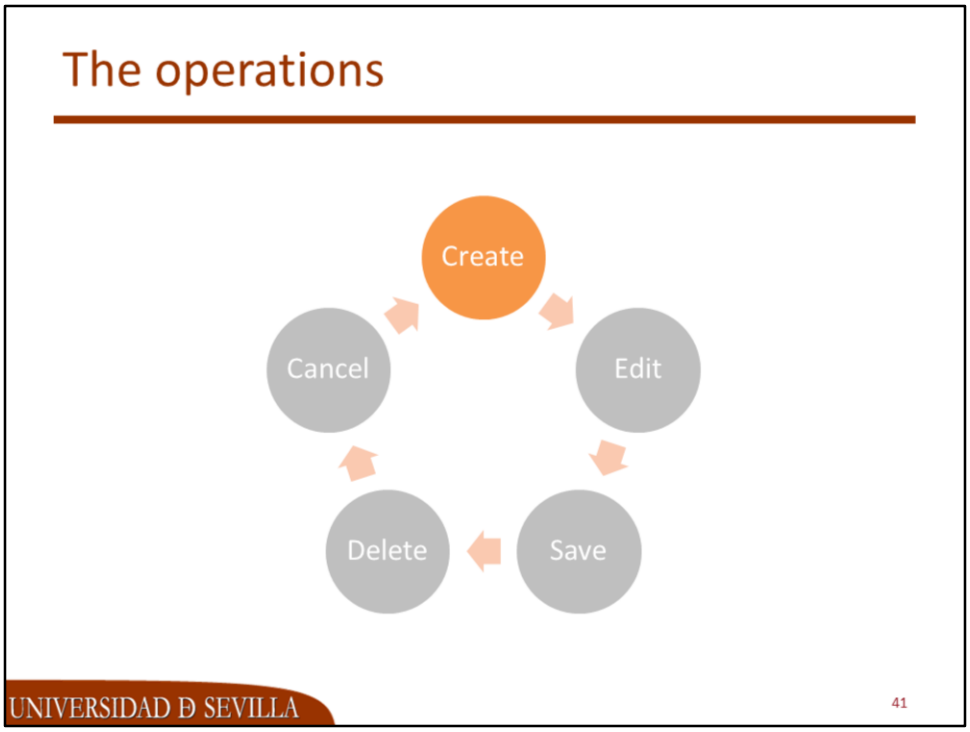
The controller class is very similar to the previous case. It starts with a "@Controller" annotation that lets Spring know that this class implements a controller, then comes the "@RequestMapping" annotation that lets Spring know of the prefixes of the paths that are served by this controller, and finally comes the extension of the "AbstractController" class, which handles unexpected exceptions.



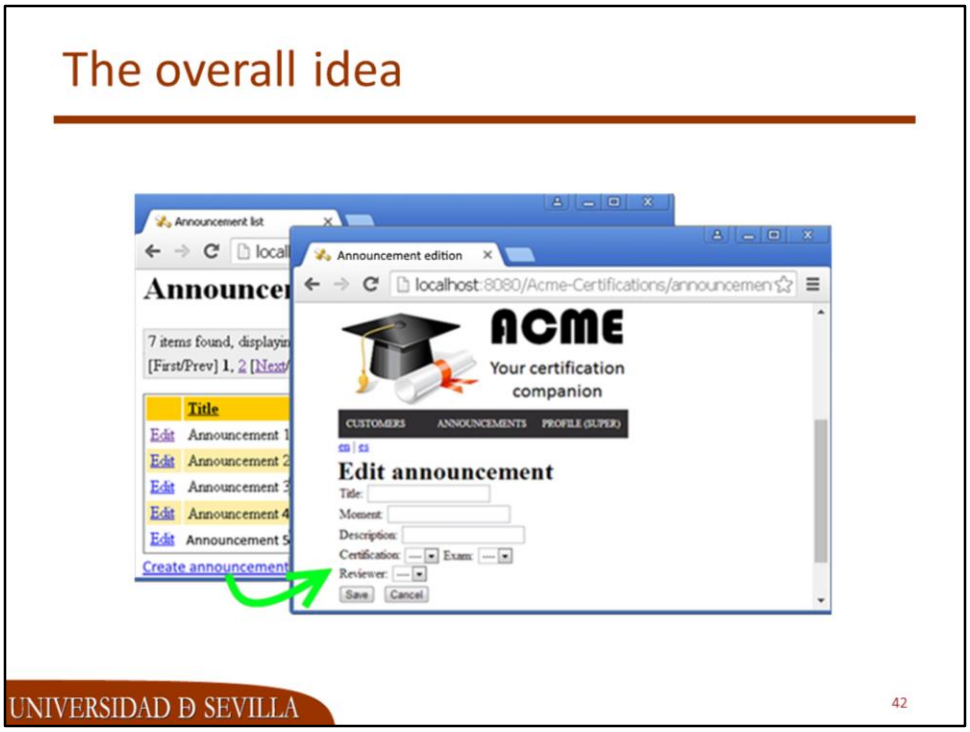
As usual, we need to reference a number of services by means of auto-wired attributes. In this case, we need references to the announcement service, the certification service, and the reviewer service.



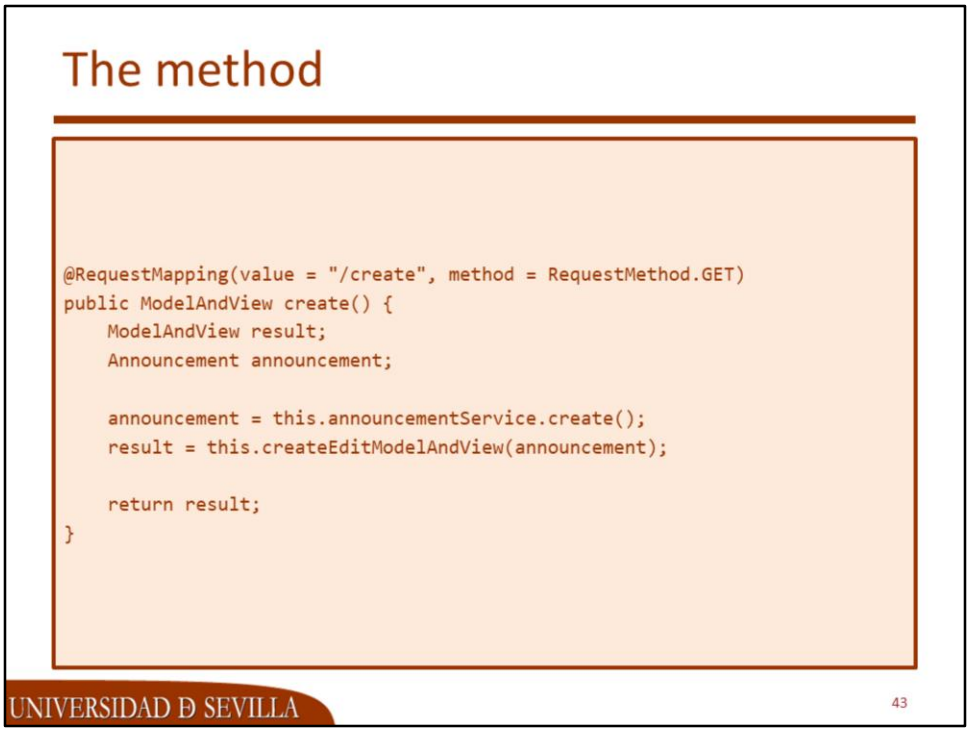
The operations are a little more involved in the case of edition forms, since we need to implement different methods to create, edit, save, or delete entities; cancelling an edition is a different matter.



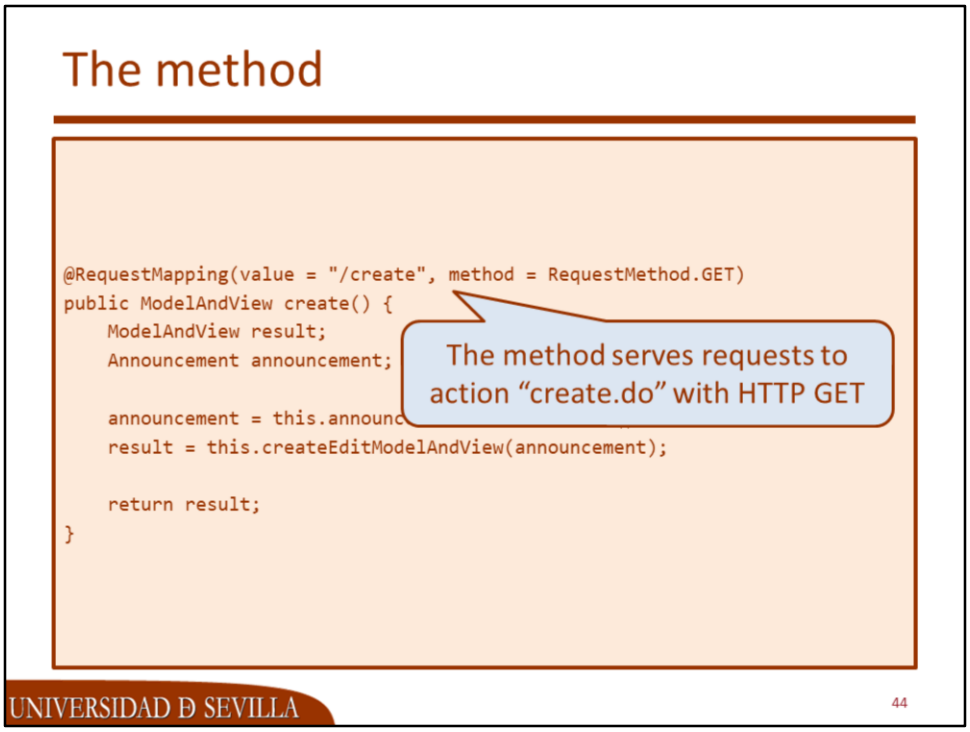
Let’s start with the method to create a new entity.



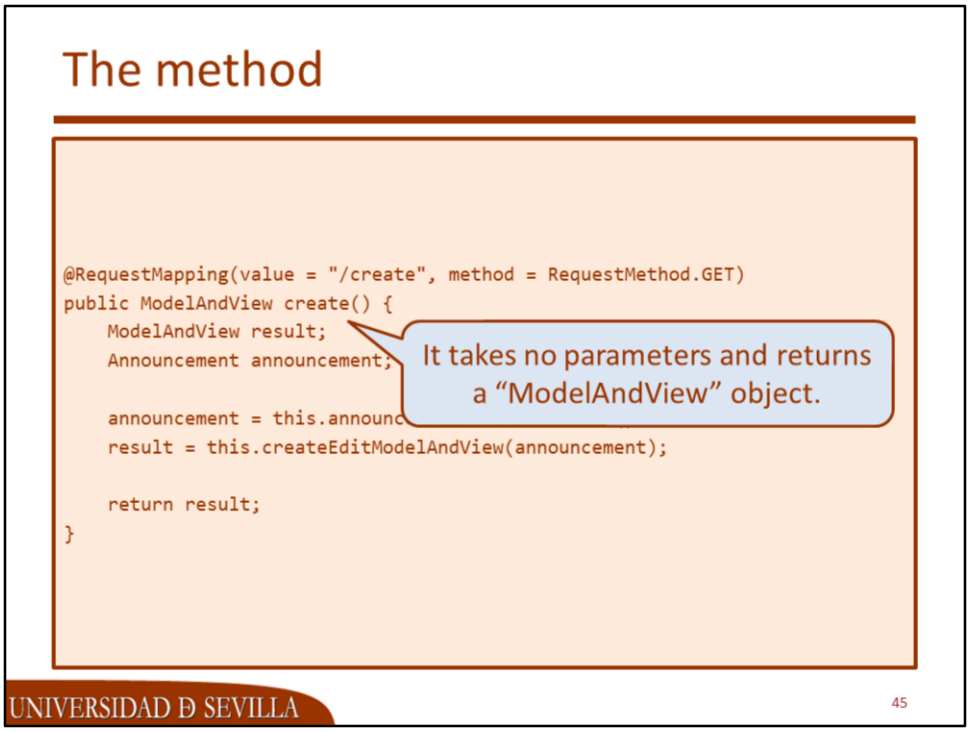
The creation form is requested when the user clicks on the “Create announcement” in the listing form. This is how it looks like: there’s an empty form, the user enters some data in the input boxes, and then hits the “Save” button.



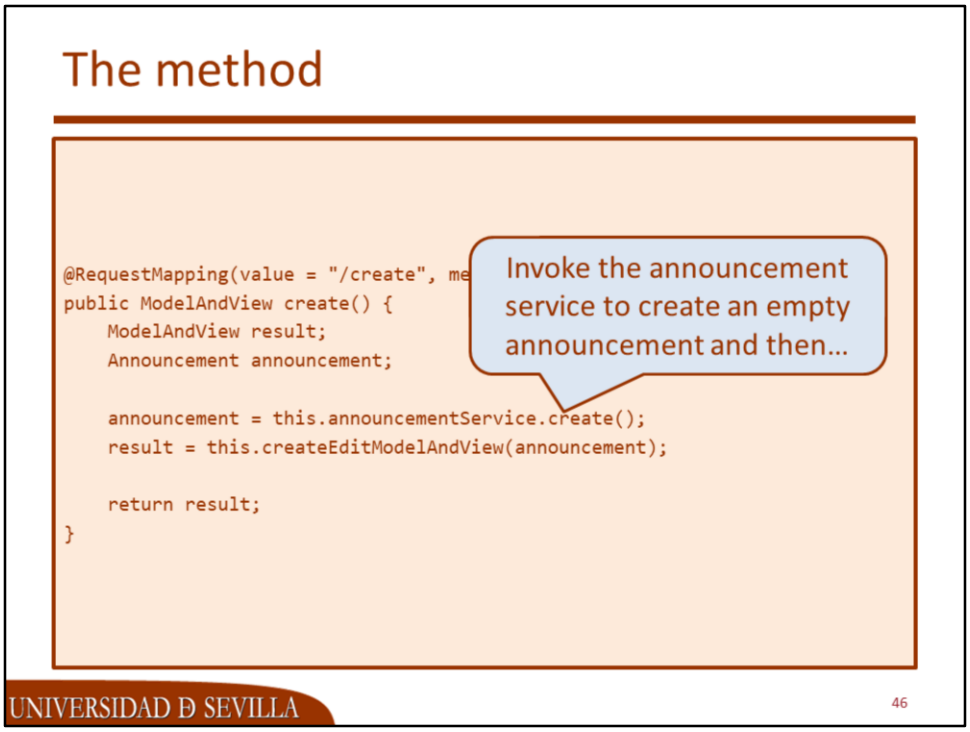
And this is the controller method, which is executed whenever the user requests a URL of the form “http://localhost:8080/Acme-Certifications/announcement/administrator/create.do”.



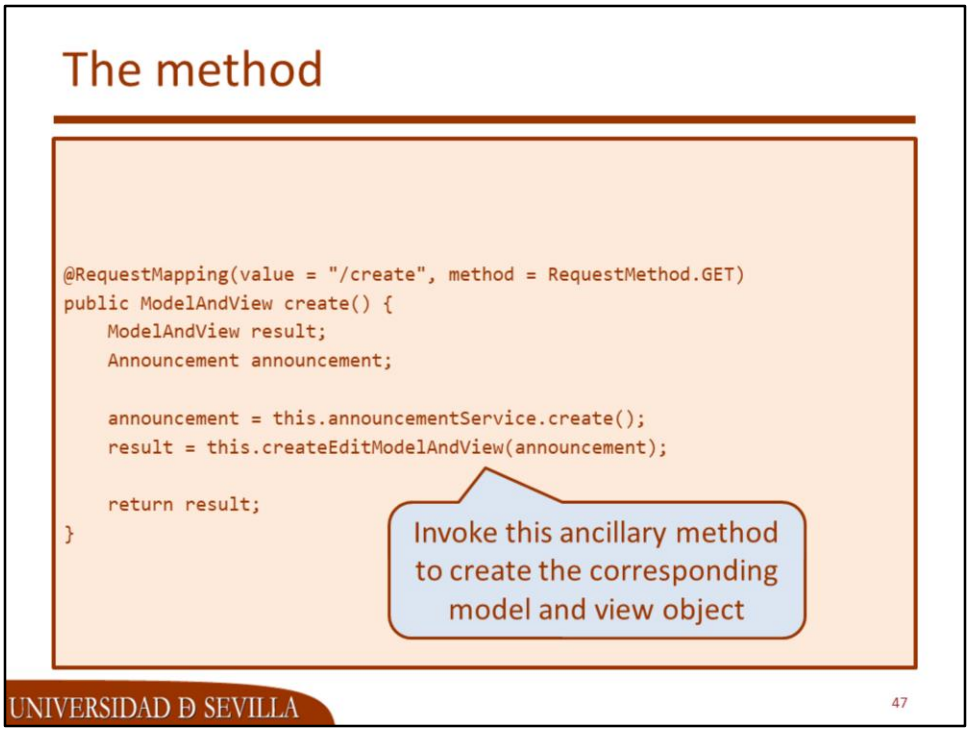
Note that the method must be prepended with a "@RequestMapping" annotation to let Spring know that it serves HTTP GET requests to action "create.do".



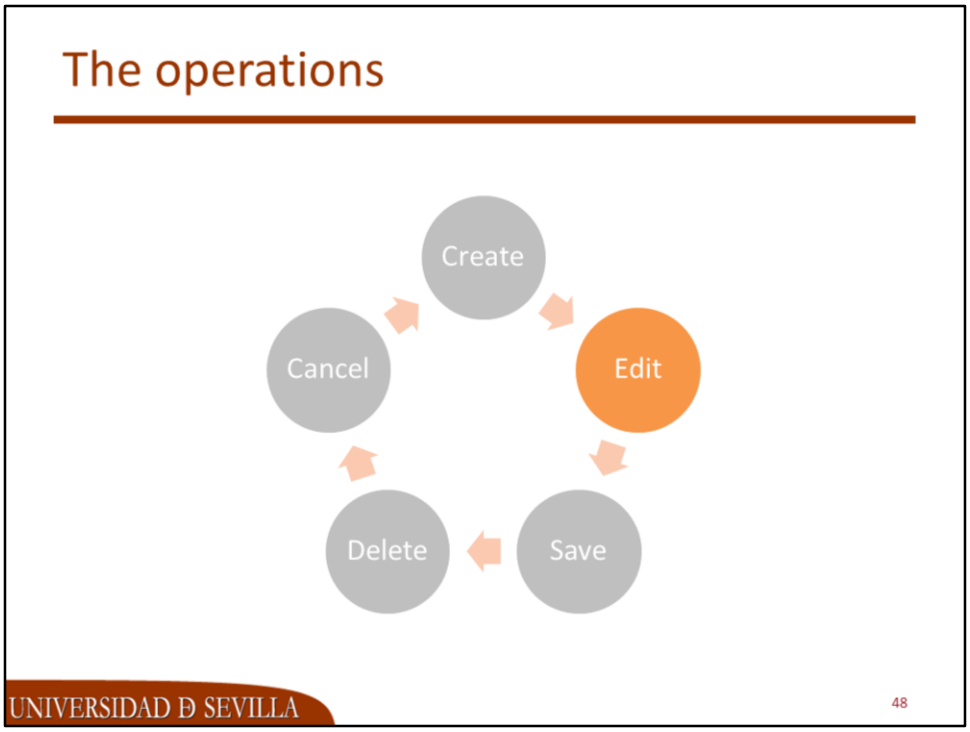
Note, too, that it doesn't take any parameters and returns a "ModelAndView" object as usual.



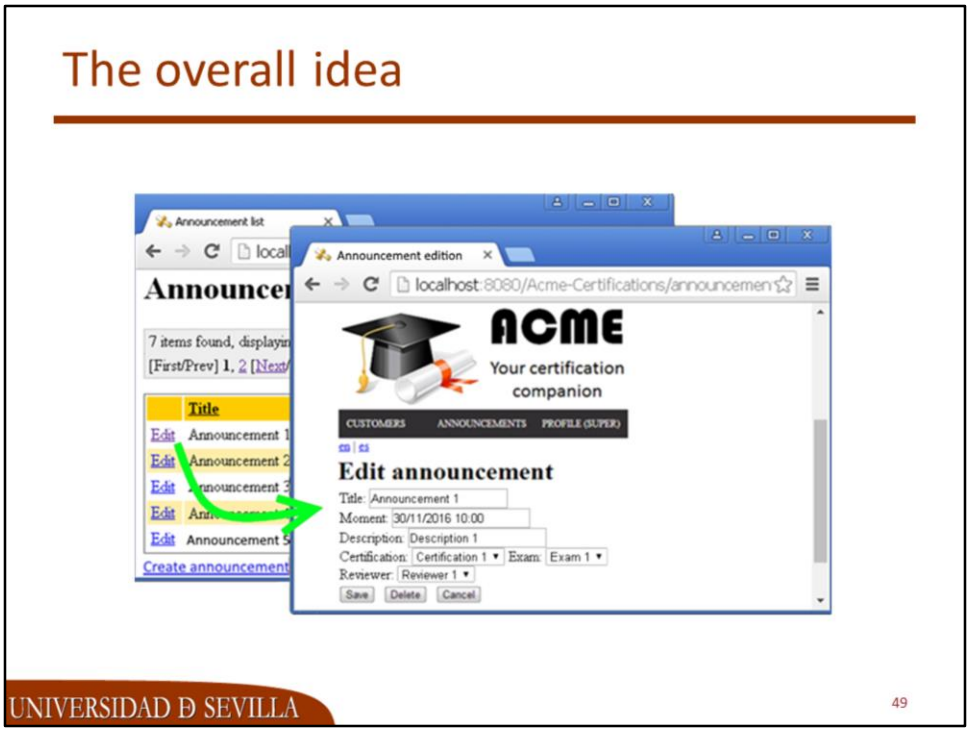
The core of the method is very simple: it first invokes the announcement service to create an empty announcement and then …



…invokes the "createEditModelAndView" ancillary method to create the corresponding model and view object. We'll delve into the details later; so far, it's enough to know that it populates the model that the edition form requires and returns the appropriate view.



It’s now time to delve into the edit operation.

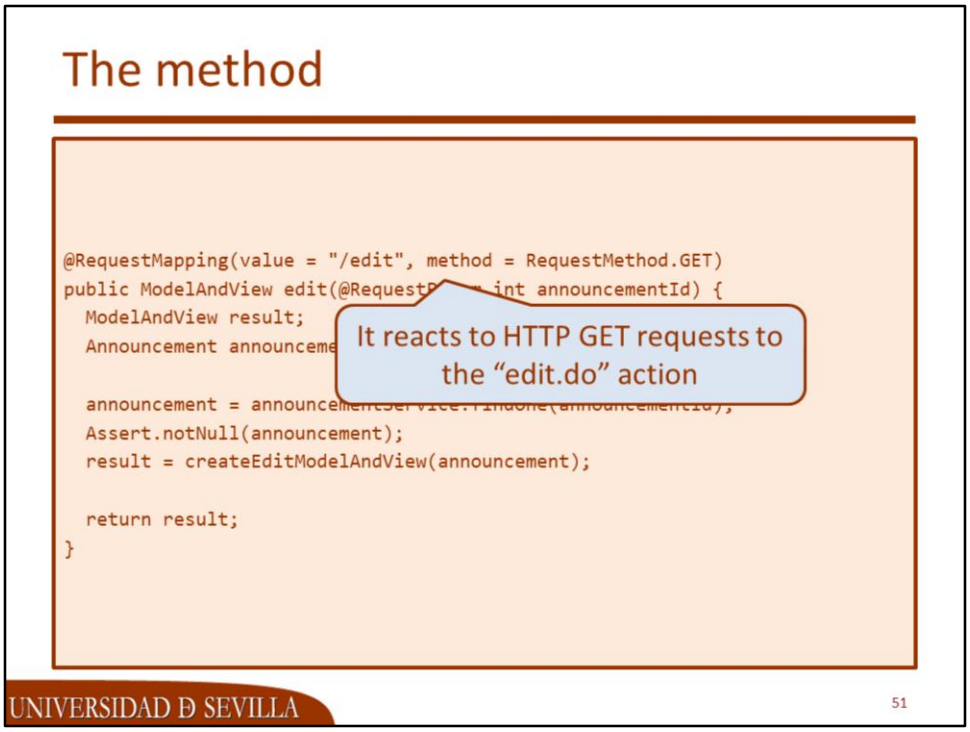


This operation is invoked when the user presses the “Edit” link in the announcement listing. This link takes the user to a form that is preloaded with the values of the attributes of the announcement that was selected to be edited.

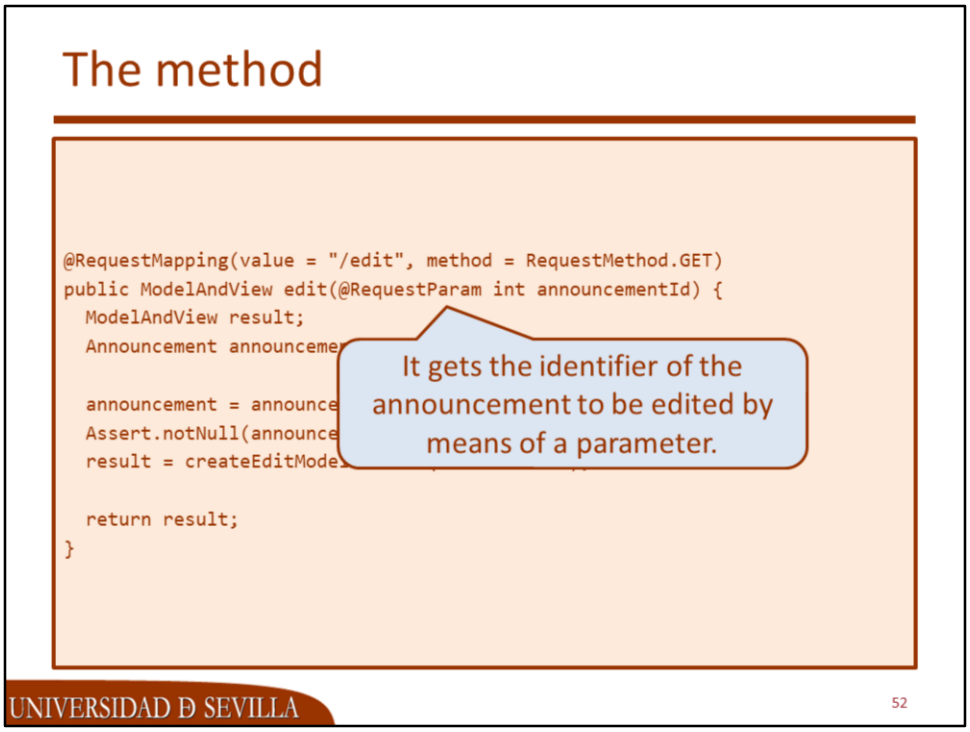
The method

```
@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;
}
```

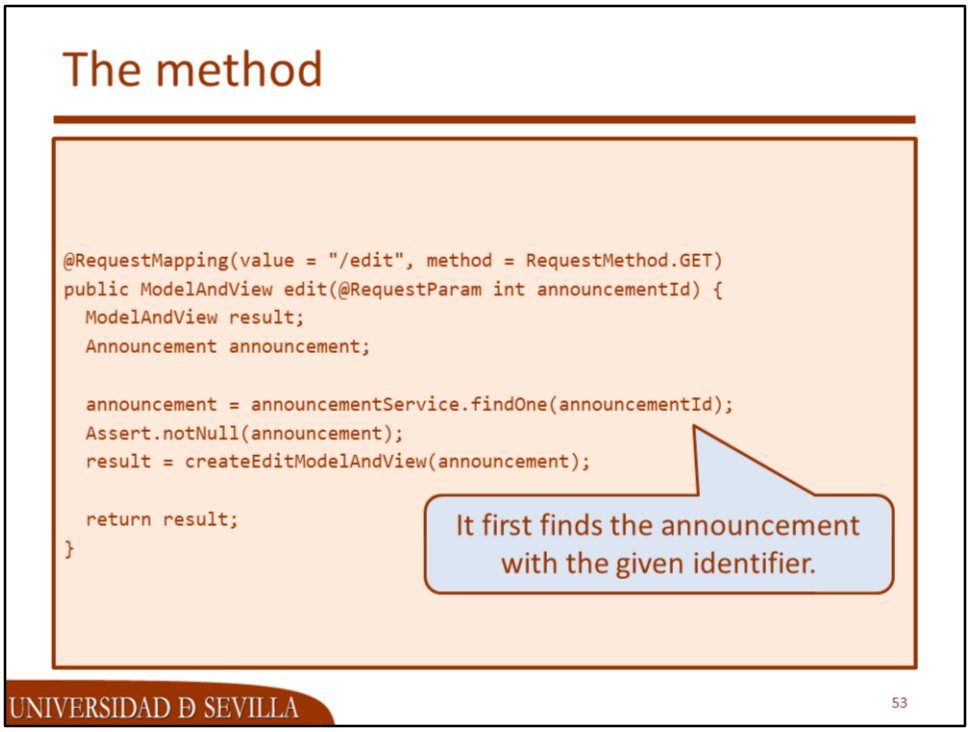

This is the method, which we're going to dissect in the following slides.



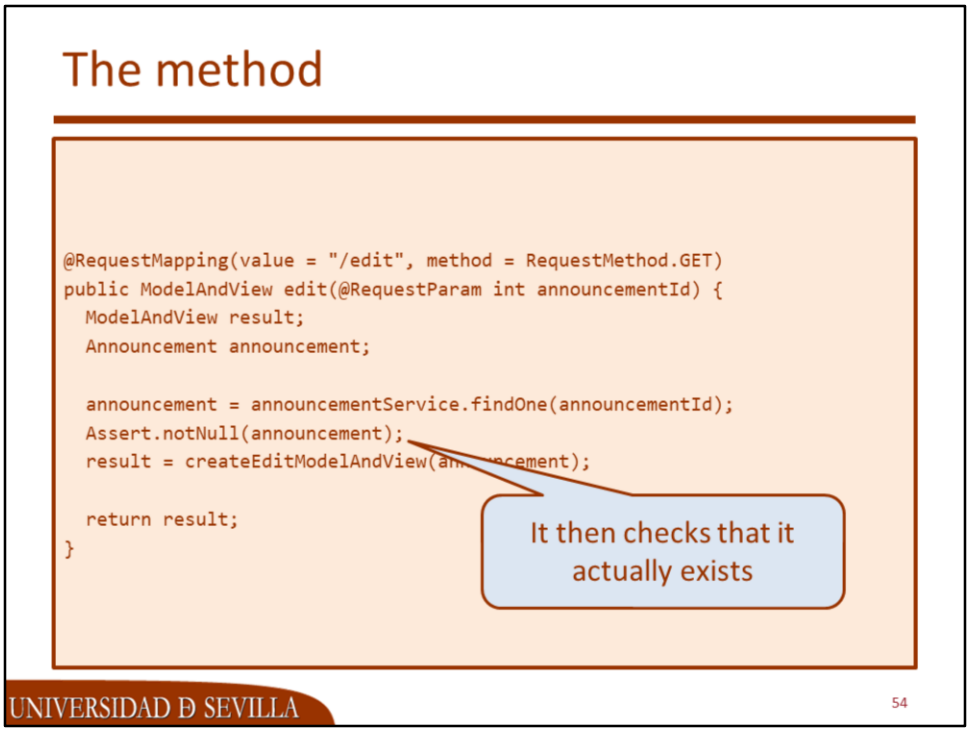
As usual, it's prepended with an "@RequestMapping" annotation that tells Spring that this method must be invoked when the user requests the "edit.do" action using HTTP GET.



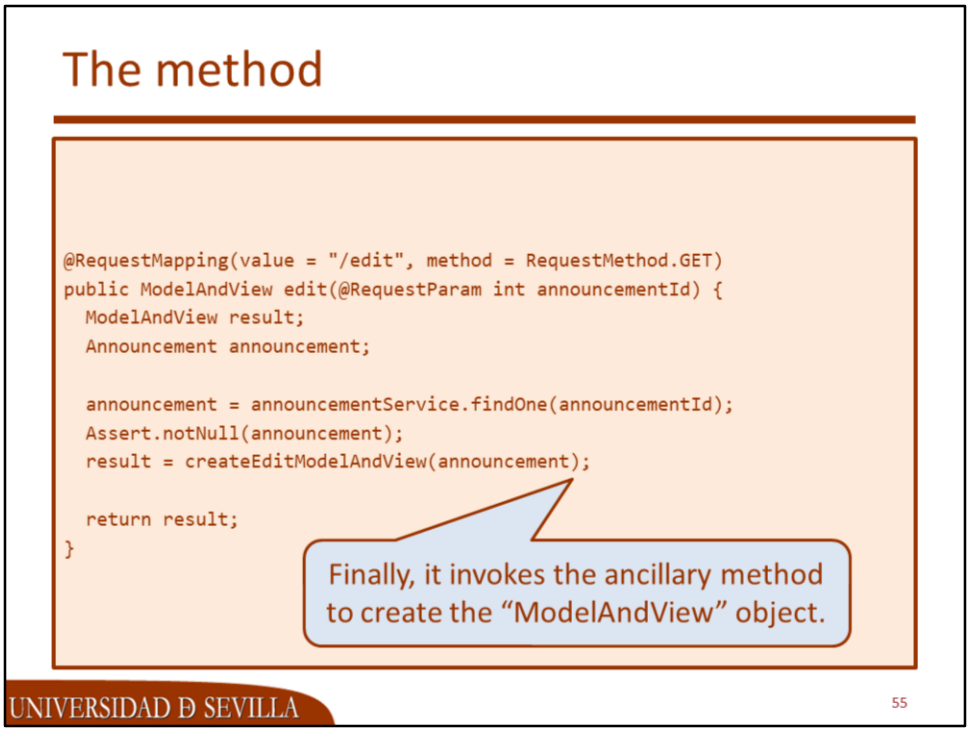
Note that this action requires the request to provide the identifier of the announcement that is going to be edited. For instance, a typical request looks like this: “http://localhost:8080/Acme-Certifications/announcement/administrator/edit.do?announcementId=99”. Note that the value of the URL parameter is captured as a regular method parameter thanks to the “@RequestParam” annotation.



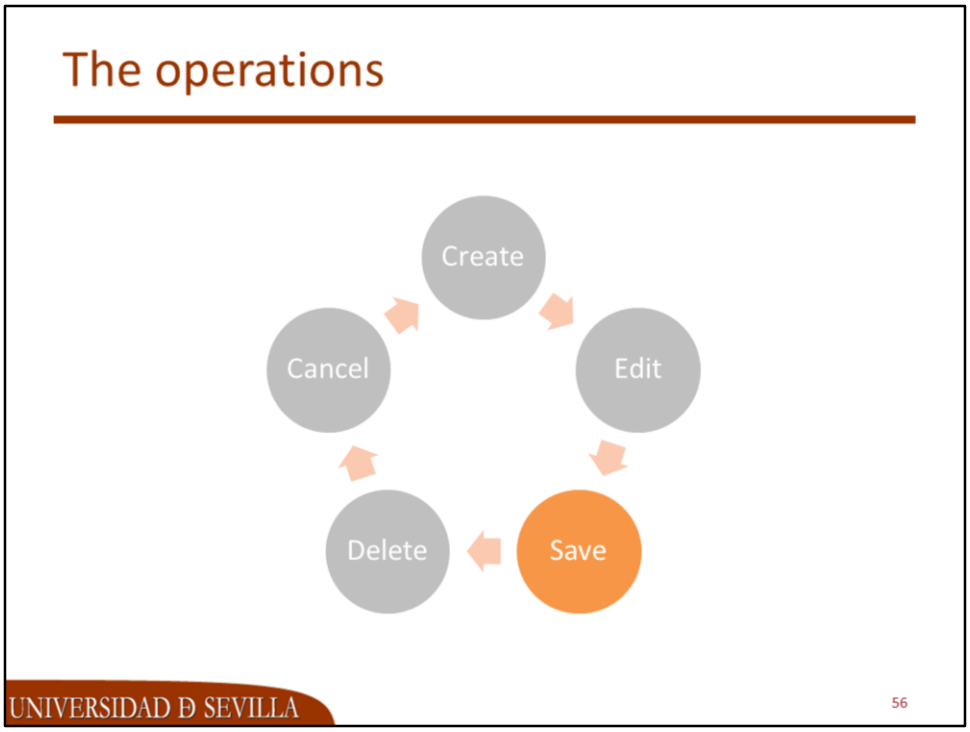
Now, the method uses the announcement service to retrieve the announcement with the given identifier.



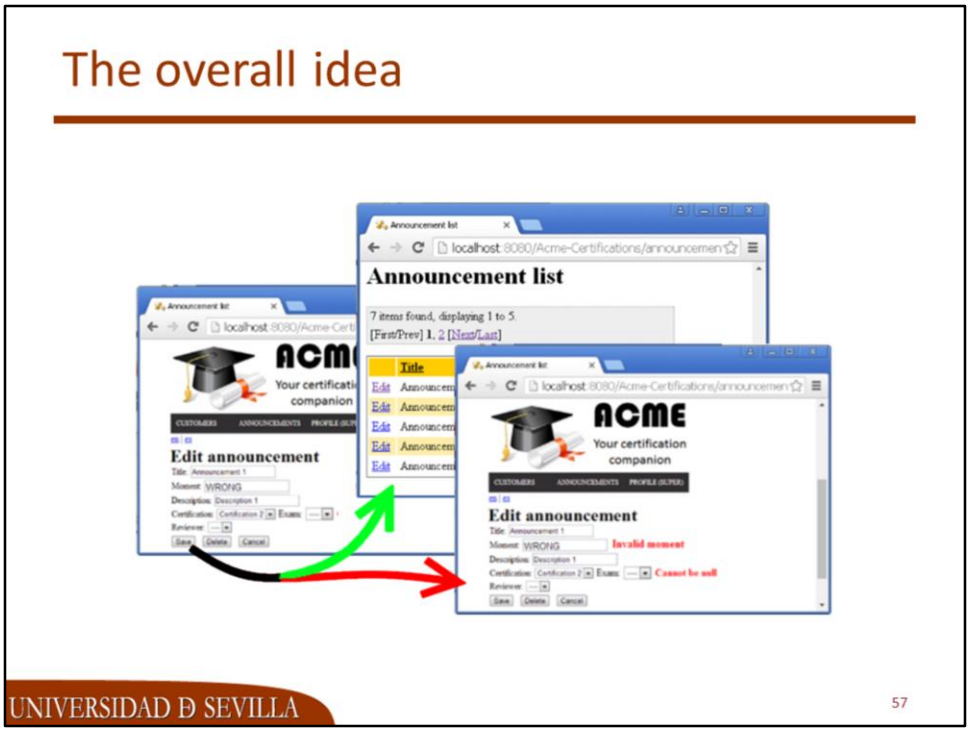
If the announcement exists, then the assert is passed; otherwise it results in an exception that is caught by the “AbstractController” class.



If everything is OK, then it invokes method "createEditModelAndView" to create the appropriate model and view building on the announcement that we've just retrieved.



Let’s now explore how to save an announcement.

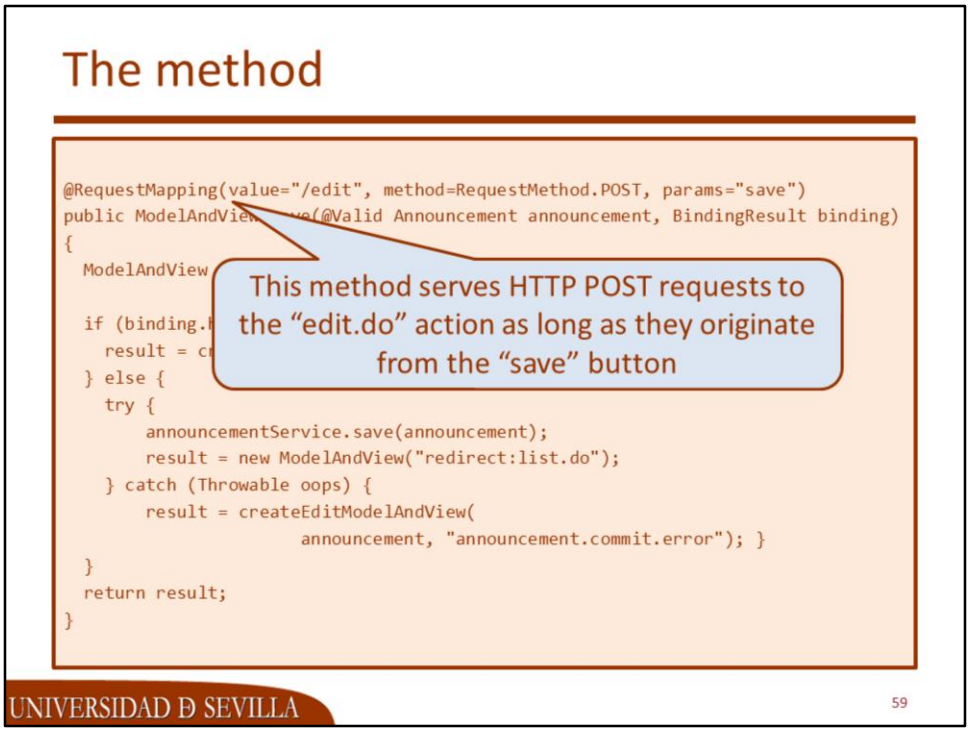


This slide illustrates what may happen when the user presses the “Save” button: it may result in the entity being edited getting saved to the database, in which case the user returns to the listing view, or errors, in which case the edition form is returned and the appropriate error messages are shown on the screen..

The method

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

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

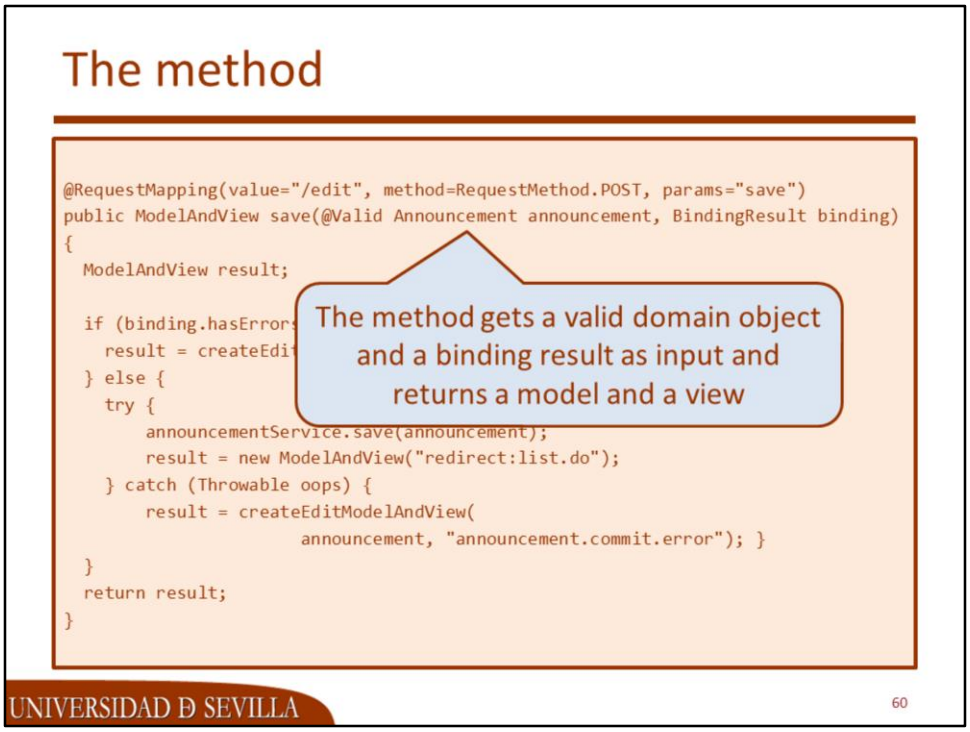

This is the method to save the entity that is being edited.



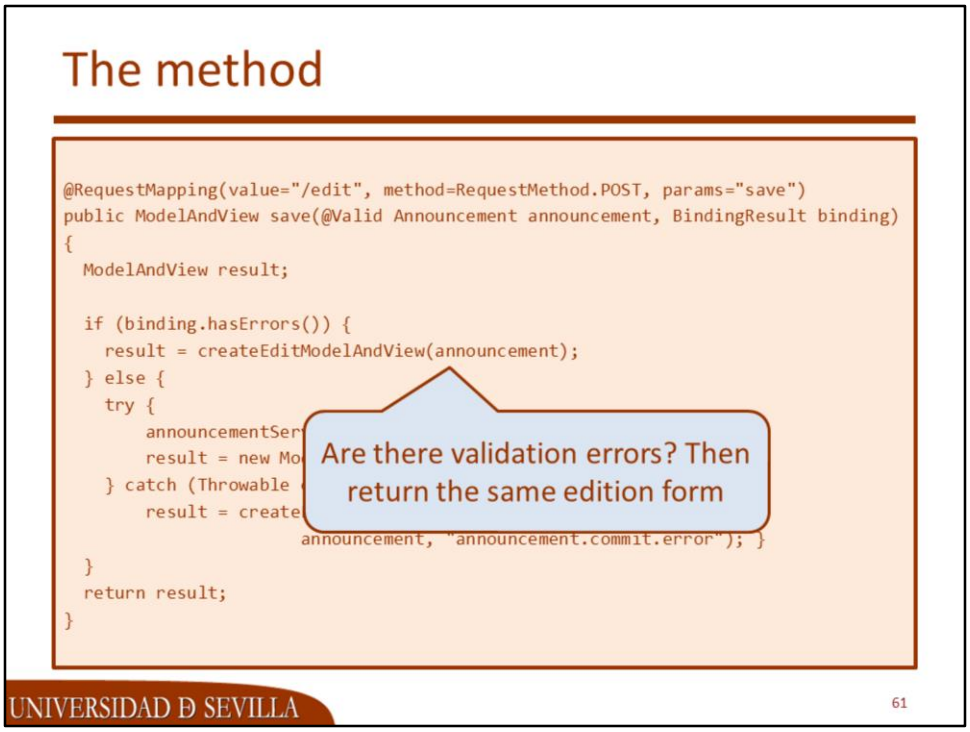
Note the first difference with regard to the methods that we've explored previously: it serves HTTP POST requests to the "edit.do" action, but they must originate from a button whose name is "save". Recall that a typical save button is introduced like this:

```
<input type="submit"  name="save"  value="Save!" />
```

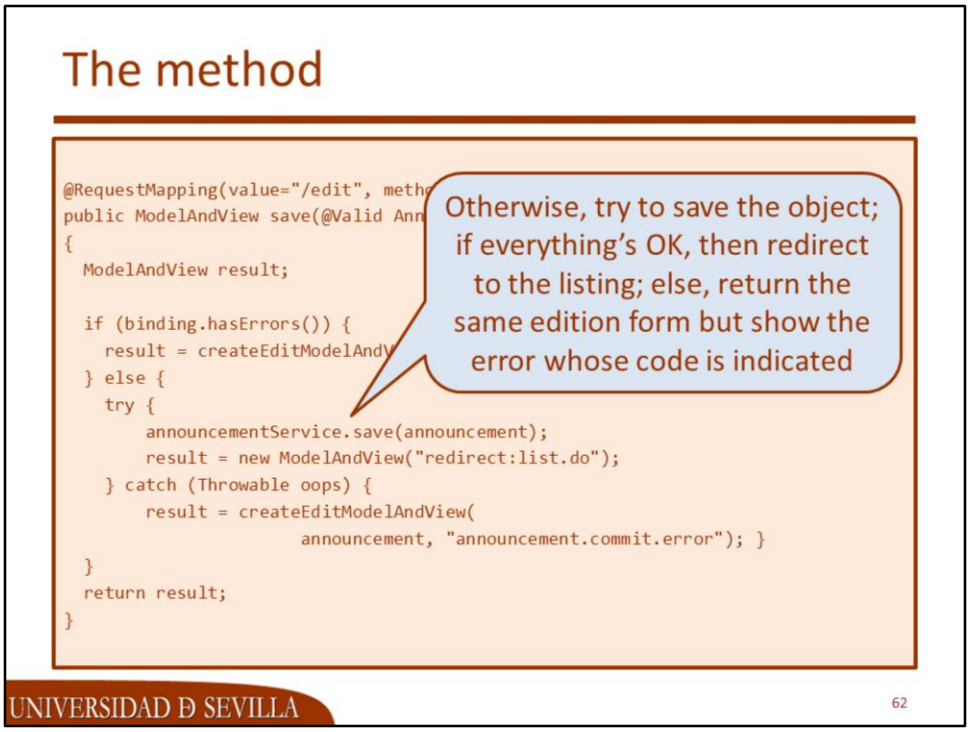
Recall that the "name" attribute has the internal name of the button and the "value" attribute has the label that is shown on top of it. The "params" attribute in the "@RequestMapping" annotation refers to the name of the button.



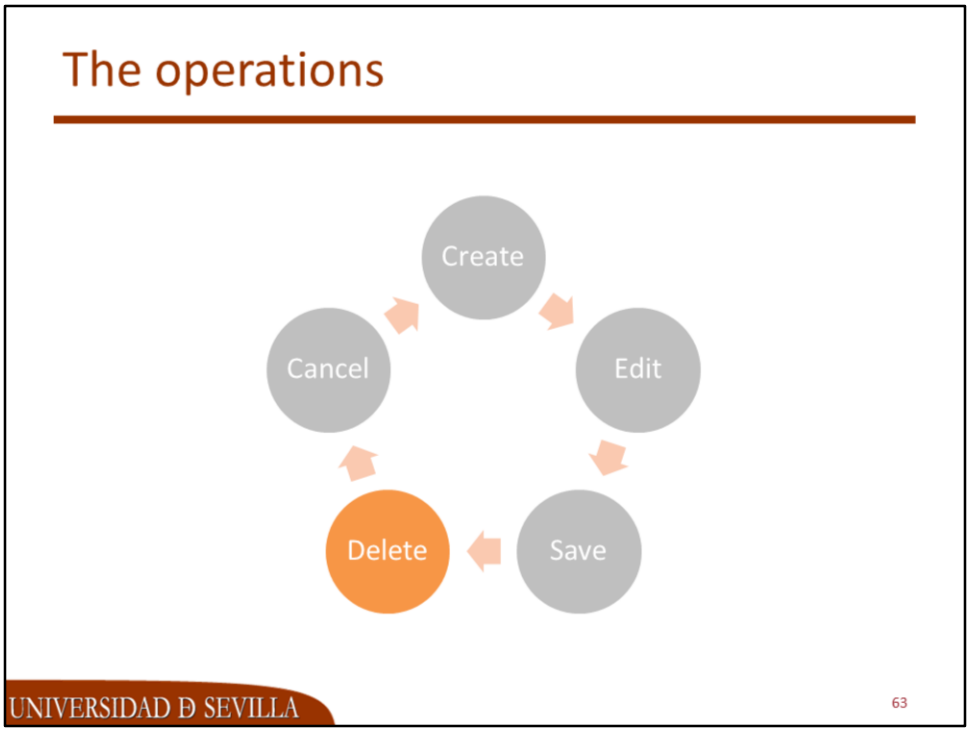
The second difference is that this method gets two parameters, namely: “announcement”, which has a reference to the object being edited, and “binding”, which has a reference to a “BindingResult” object. Note that the “announcement” parameter has an “@Valid” annotation, which requests Spring to check the validation constraints that we defined on the domain entities. The “binding” parameter gets an object in which Spring informs about the validation errors.



If the binding result indicates that there were validation errors, then then method creates a “ModelAndView” object regarding the same announcement; since the binding result has errors, they are shown on the screen automatically.

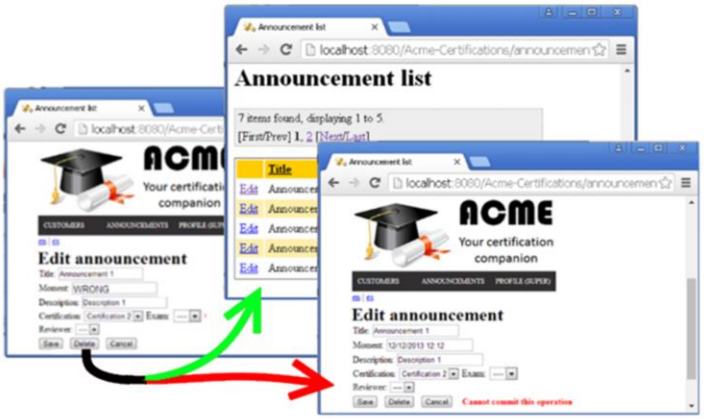


If there are not any errors, then the method tries to save the announcement; if everything's OK, then it redirects to the listing; otherwise, it returns the same edition form but shows the error whose code is indicated.



Let’s now explore the delete operation.

The overall idea



UNIVERSIDAD D SEVILLA

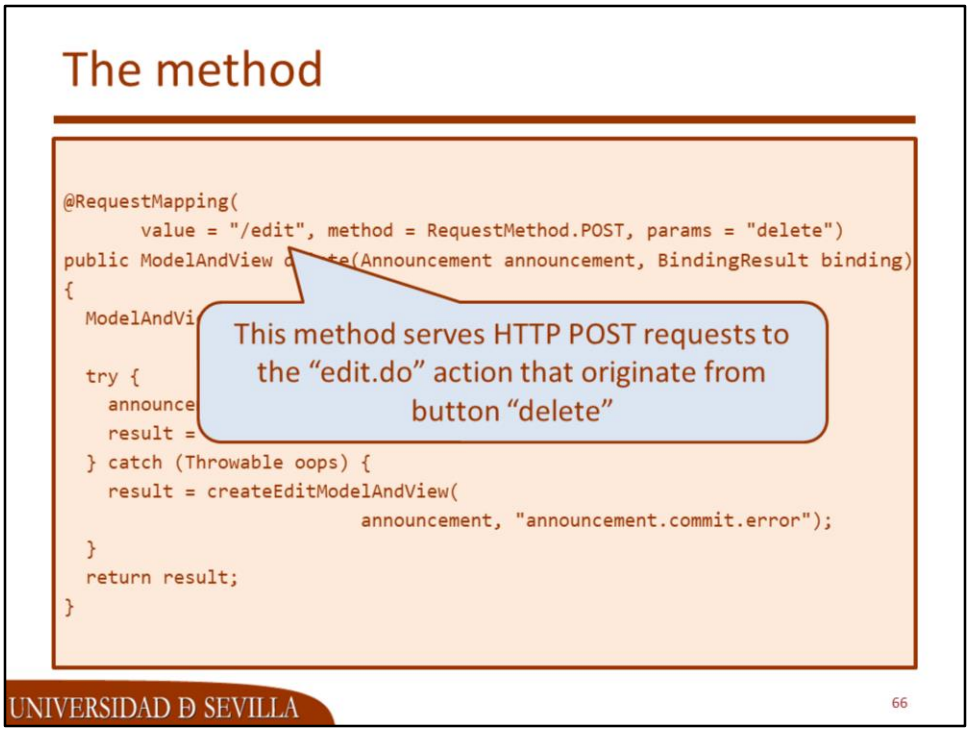
64

This slide shows what happens when the “Delete” button is pressed: the system can either delete the entity being edited from the database, in which case the user returns to the listing view, or an error may occur, in which case an error message is shown on the screen.

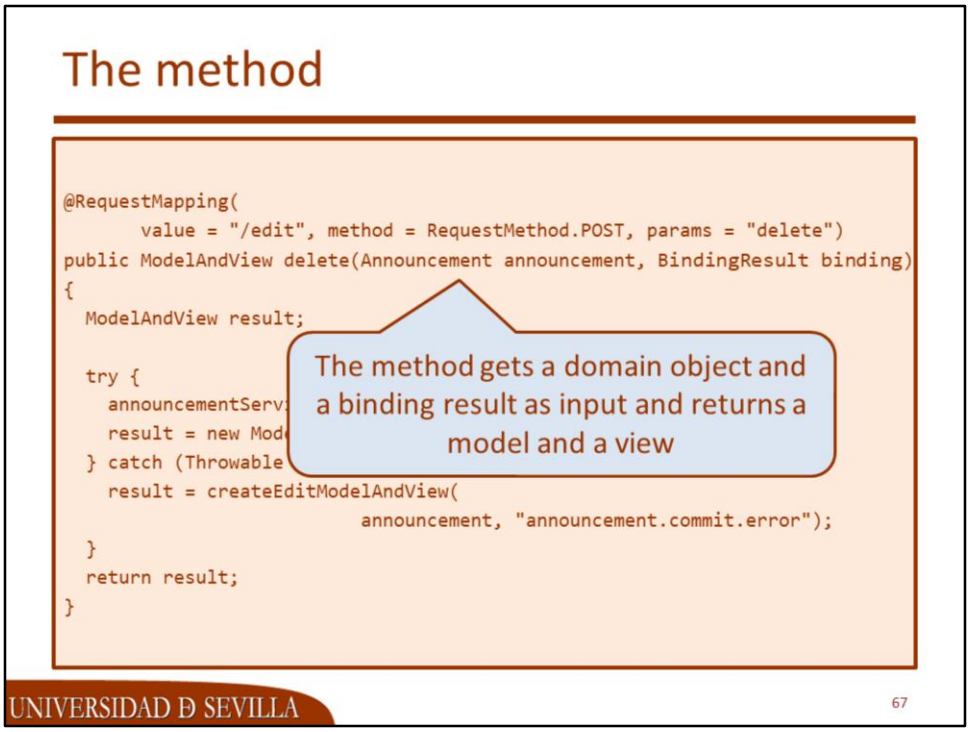
The method

```
@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;
}
```

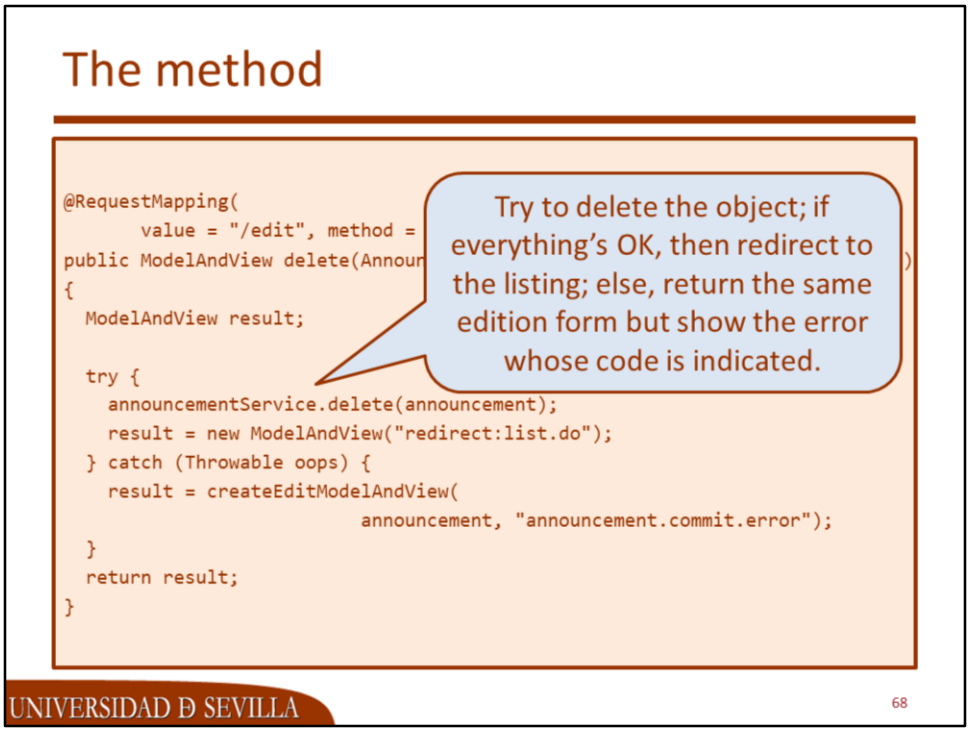

This is the method that implements the delete operation.



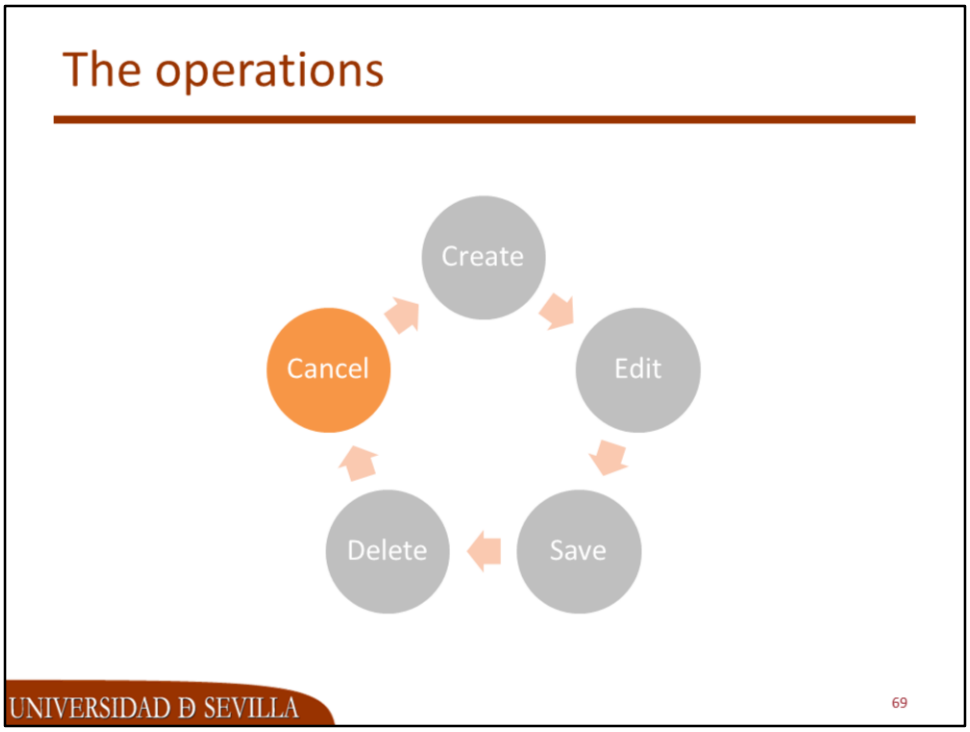
As usual, the method starts with a "@RequestMapping" annotation that indicates that it serves HTTP POST requests to the "edit.do" action as long as they originate from a "delete" button.



The method gets an argument of type “Announcement”, which is not required to be valid. Note, however, that the method needs to get a “BindingResult” parameter, even if you’re not going to use it. The reason why the input object needs not be valid is that they user may have changed an attribute to an invalid value, but that should not prevent us from deleting the object.

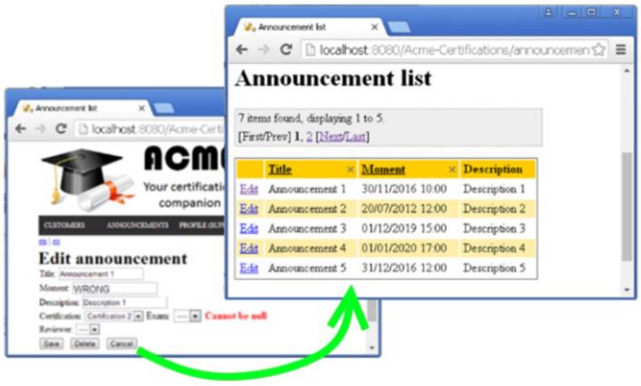


Then the method tries to delete the object from the database. If everything works well, then we return to the listing view; otherwise, we create a "ModelAndView" object that shows the object and an error message.

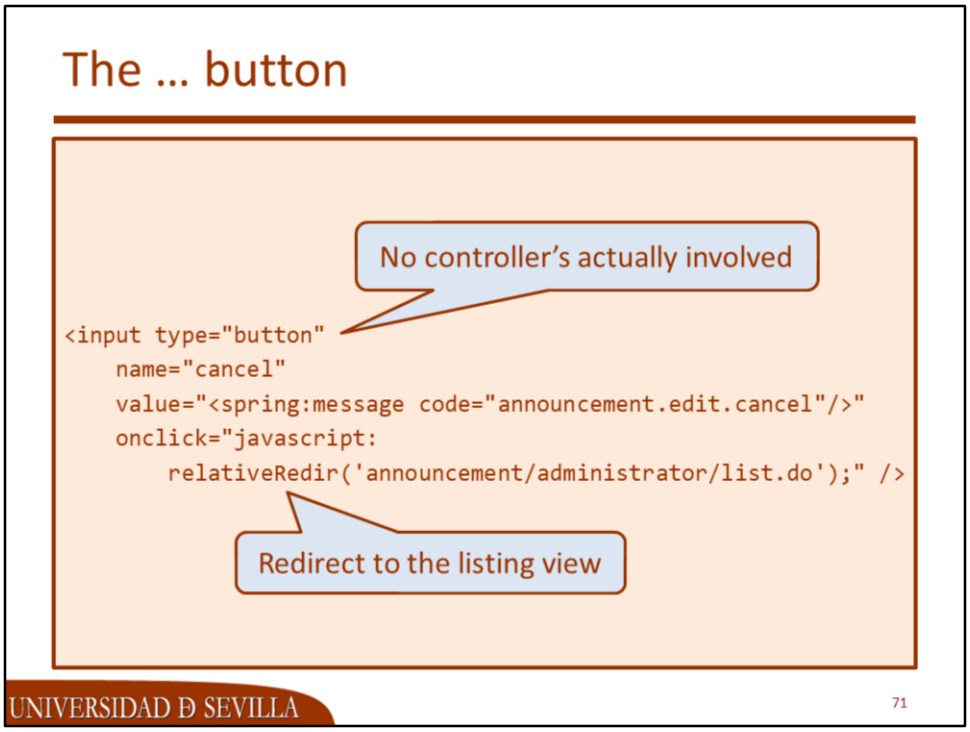


It’s now time to explore the cancel operation, which is a bit different.

The overall idea



The action is invoked when the user presses the “Cancel” button in the edition form, in which case, he or she returns to the listing view.



Note that no controller's involved in cancelling the edition of an entity. Pressing the "Cancel" button just executes a piece of Java Script that redirects the browser to the corresponding listing. This doesn't involve any server-side processing. It just requires the corresponding button to execute a redirection to the listing URL.

The edition forms are a bit more involved than the listings, but not that difficult, right? Ok, it's now time to delve into the details of the ancillary "createEditModelAndView" method, which we have used quite extensively in the previous slides.

First overload: a by-pass

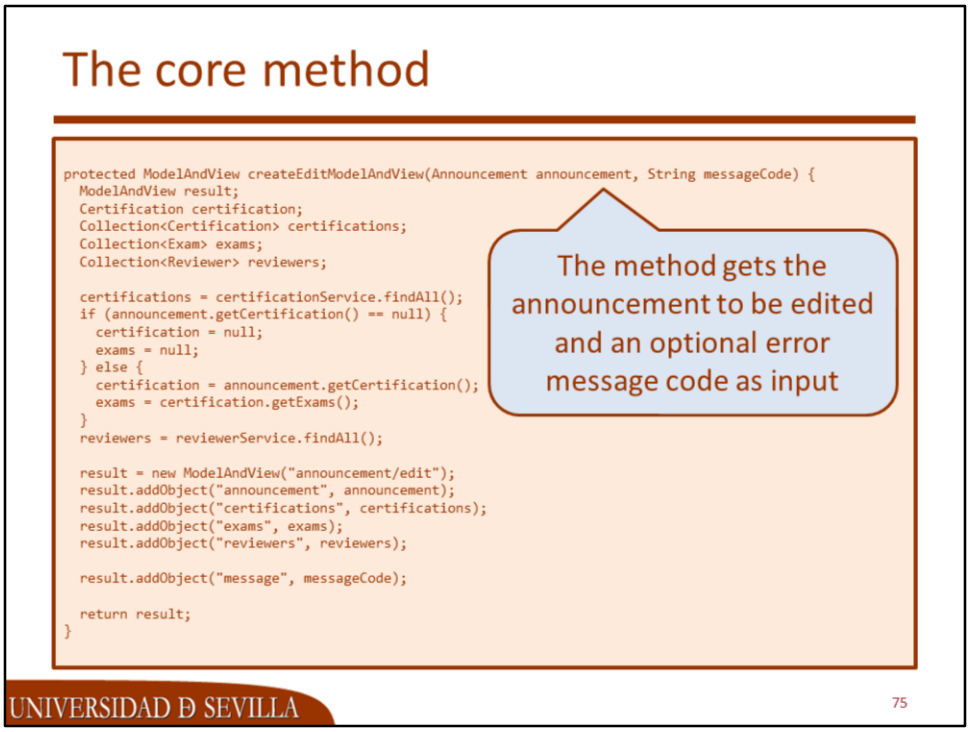
```
protected ModelAndView createEditModelAndView(Announcement announcement) {
  ModelAndView result;

  result = createEditModelAndView(announcement, null);

  return result;
}
```


The first overload of the method is a simple by-pass. It gets an announcement as input and passes it onto the second overload.

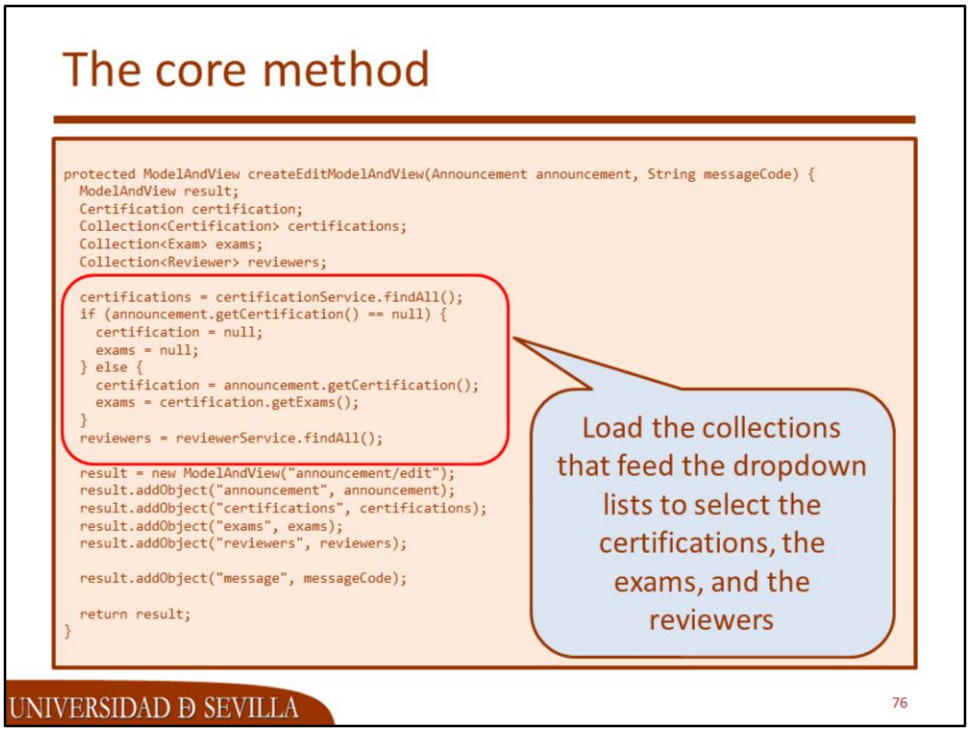
The core method

```
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;
}
```


The second overload implements the core of the method.



It gets an announcement and a message code as input. The announcement is the entity that is going to be edited and the message code references an error message or null.



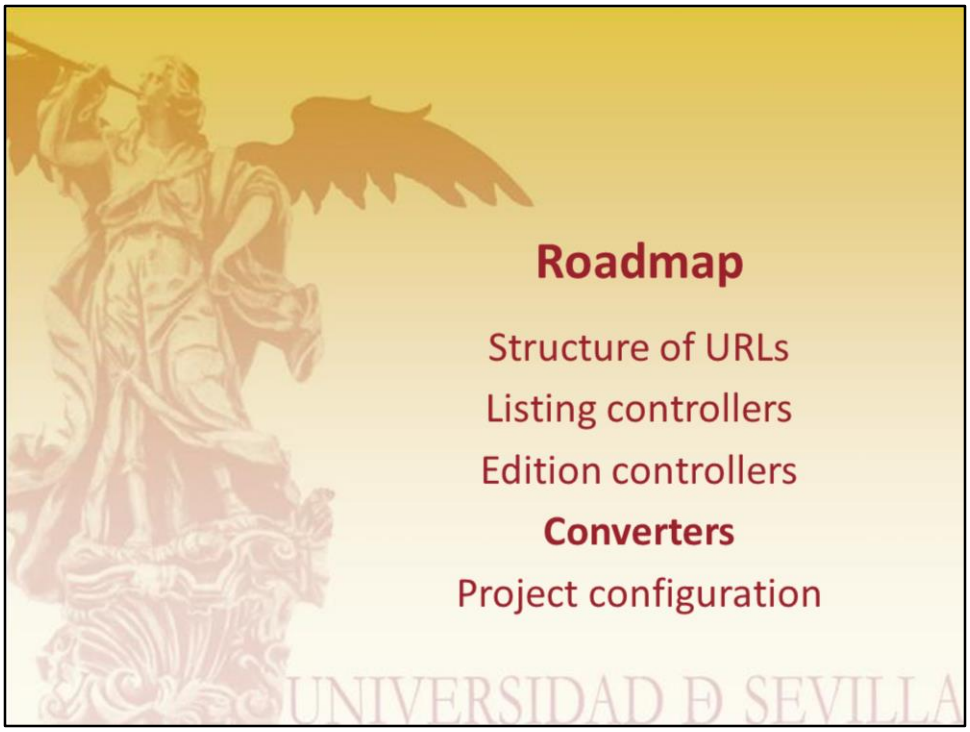
The method then loads the collections that feed the dropdown lists to select the certifications, the exams, and the reviewers.

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;
}
```

Then, create the appropriate "ModelAndView" object

Finally, it creates the appropriate "ModelAndView" object and returns it.

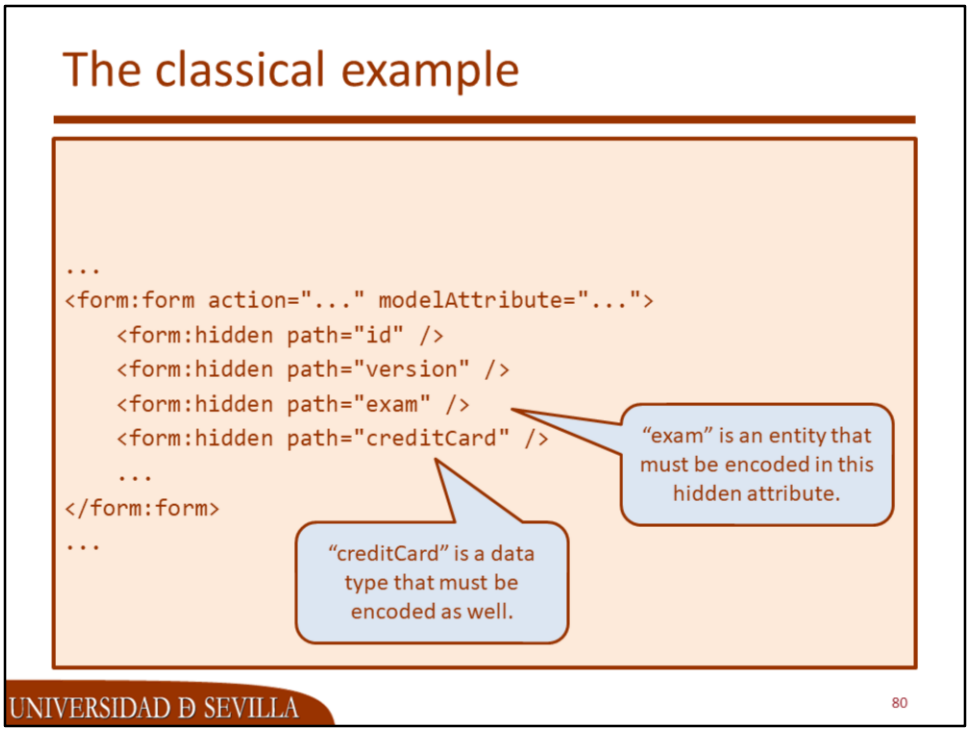


Sure, edition controllers are a bit more difficult to command than listing controllers, but not that difficult; believe us. Let's now deal with the converters.

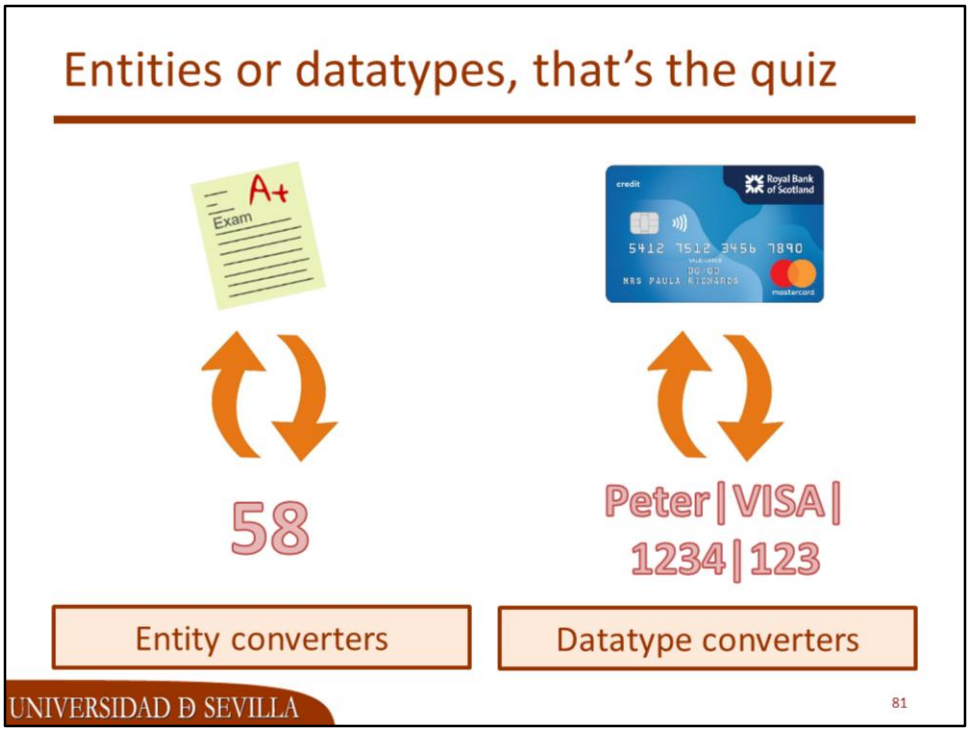
What are converters about?

A converter's a class that transforms an object into a string representation that can be embedded into HTML or vice versa

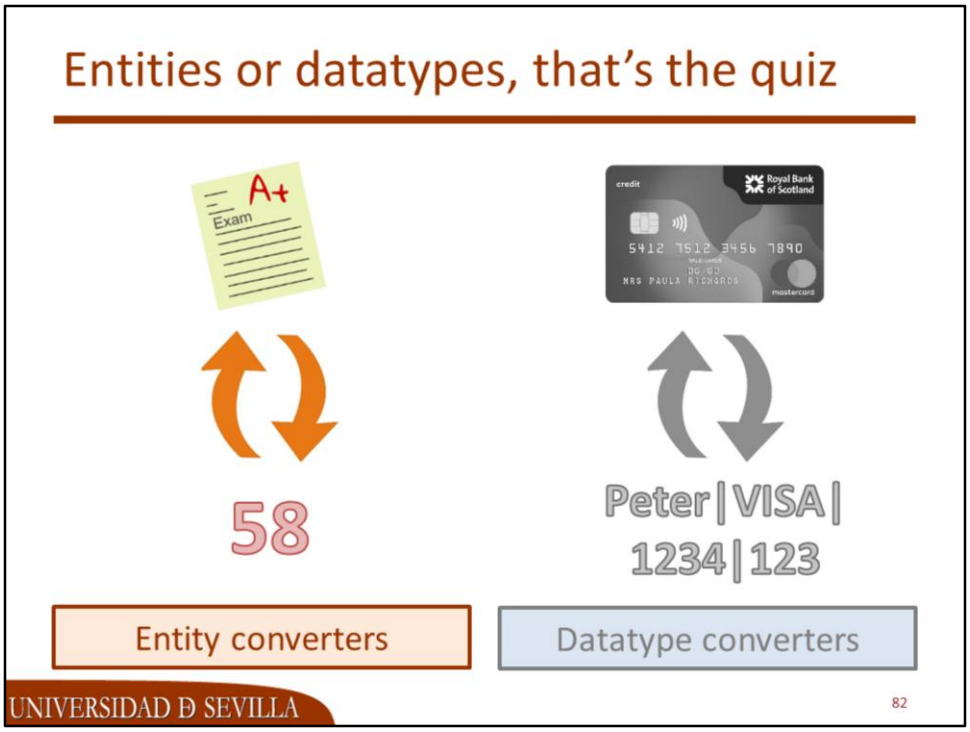
A converter's a class that transforms an object into a string representation that can be embedded into HTML or vice versa.



In this slide, we present an excerpt of a JSP view that provides a form to edit a domain object that involves an exam and a credit card. 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. Similarly, we can't serialise the full object that represents the credit card; we need to encode its attributes appropriately. That's the duty of converters.



We have to make a difference between converters that work on entities and converters that work on datatypes.

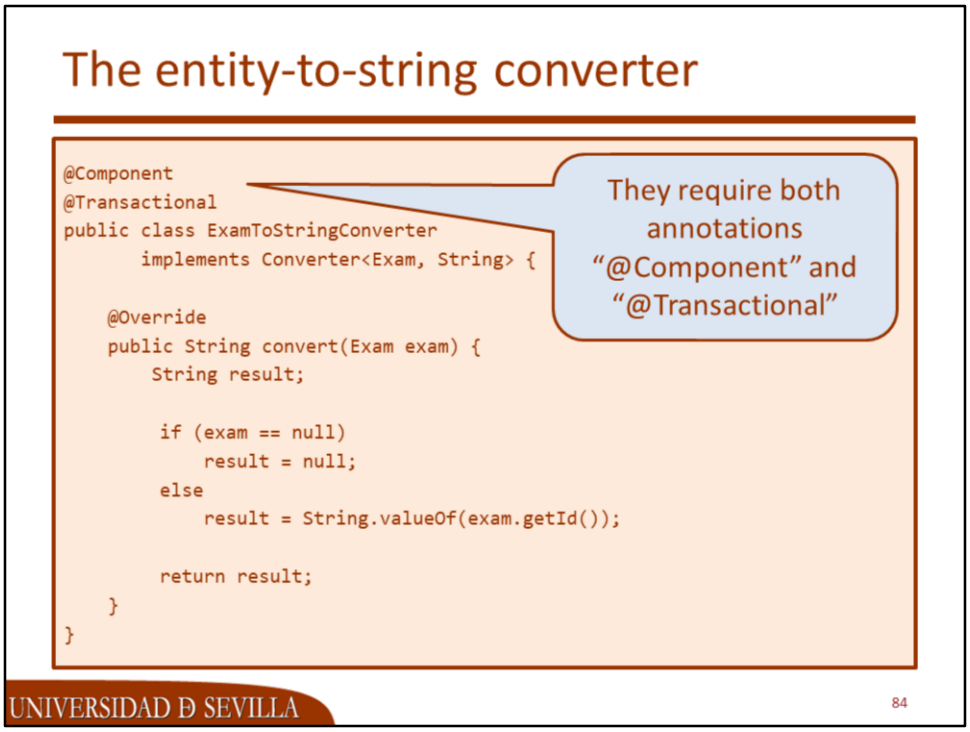


Let's start with the converters that work on entities.

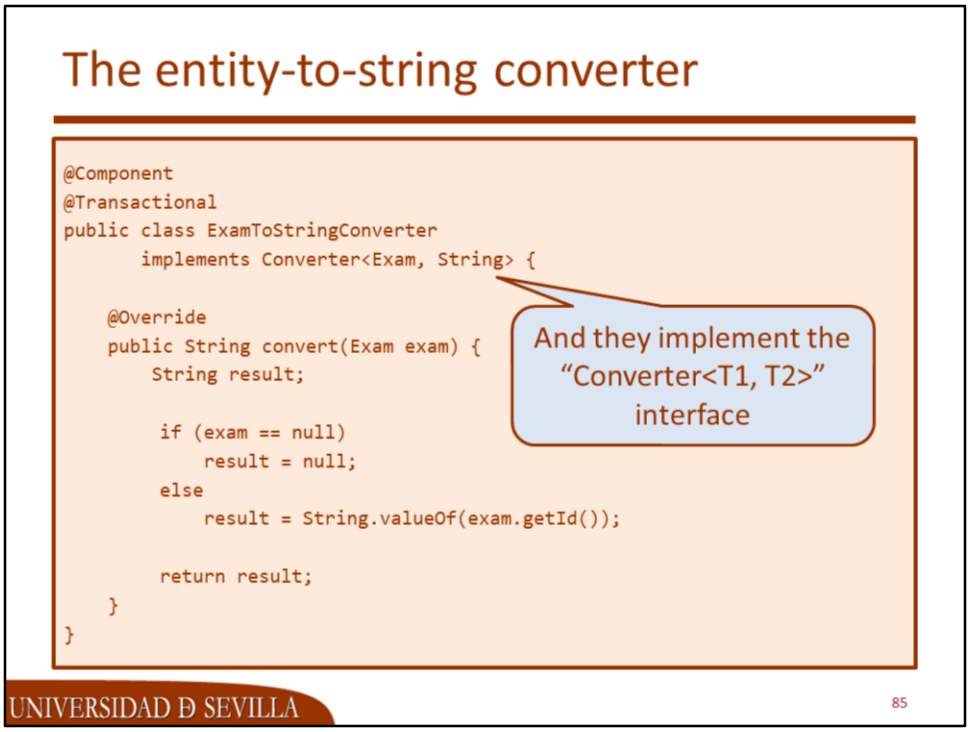
The entity-to-string converter

```
@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 entity-to-string converter looks like.



Realise that they are implemented as regular classes that have annotations "@Component" and "@Transactional". "@Component" is a generic annotation that instructs Spring to instrument this class. (It'd be great to have an "@Converter" annotation, but it doesn't exist.) "@Transactional" is required because converters are likely to be used in the context of transactions.



The class must implement interface "Converter<T1, T2>" and override method "convert", which must convert 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".

The entity-to-string 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;
    }
}
```

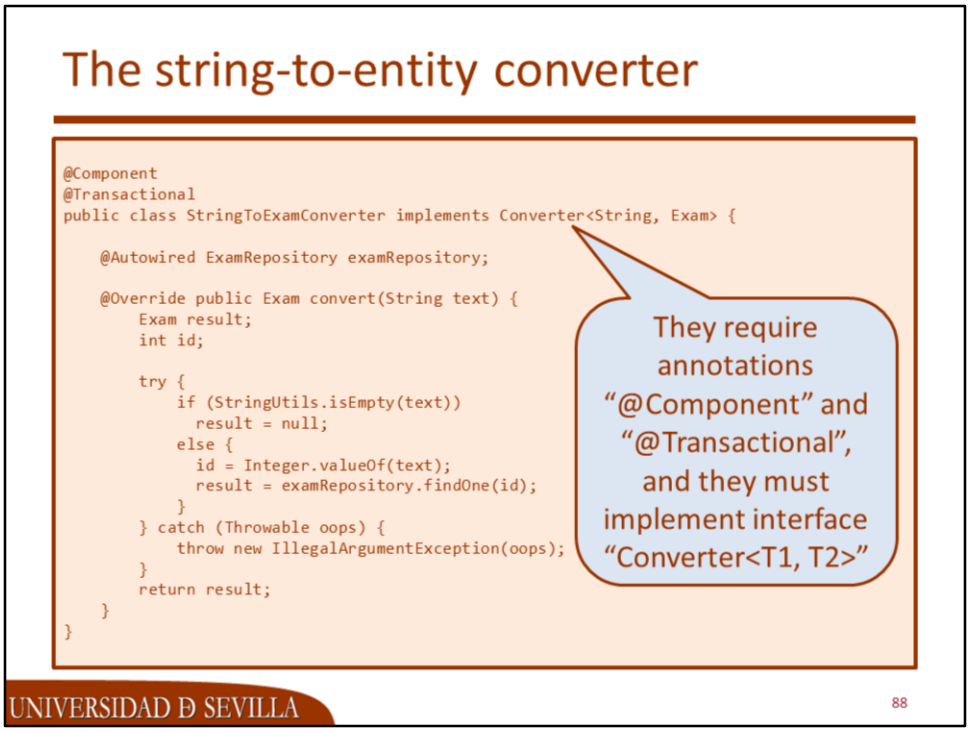
Method "convert" gets an exam and returns its identifier as a string, if any

The "convert" method 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!

The string-to-entity 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;
    }
}
```


And this is how a typical string-to-entity converter looks.



As usual, they require the "@Component" and "@Transactional" annotations and they must implement interface "Converter<T1, T2>".

The string-to-entity 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;
    }
}
```

They require an auto-wired attribute to reference a repository

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 string-to-entity 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;
    }
}
```

Method "convert" tries to convert the input text into an integer id and then searches for the corresponding entity using the repository

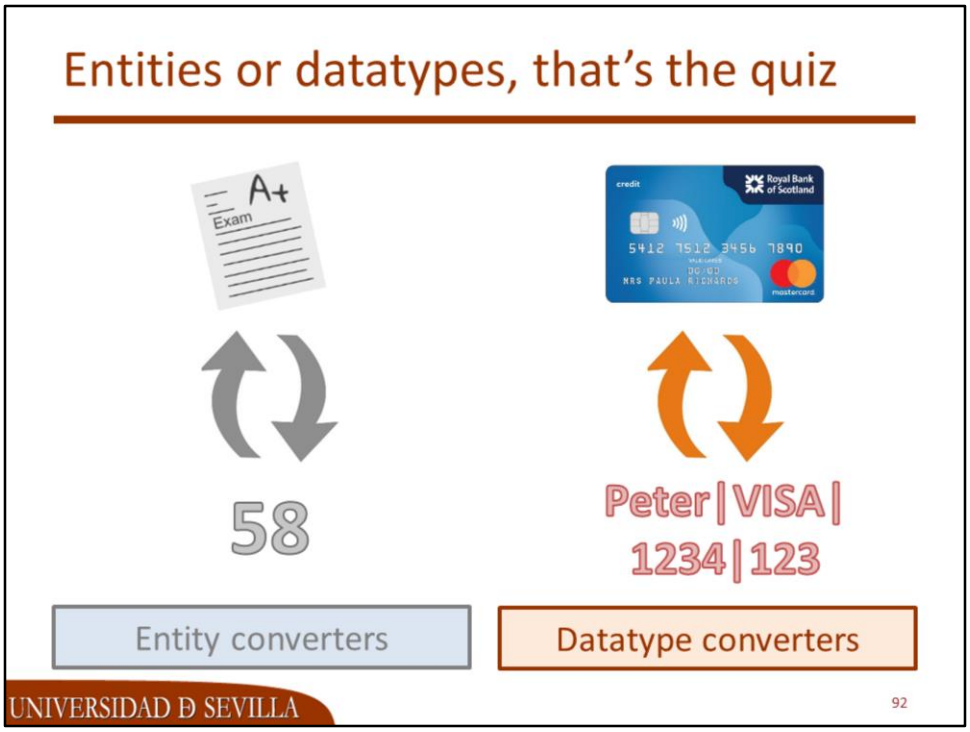
The code of the "convert" method's quite straightforward: it first checks if the result is null; otherwise, it converts the identifier into an integer number and requests the repository to return the corresponding exam.

The string-to-entity 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;
    }
}
```

An exception is thrown if something fails

If something fails, then an exception is thrown, sure.

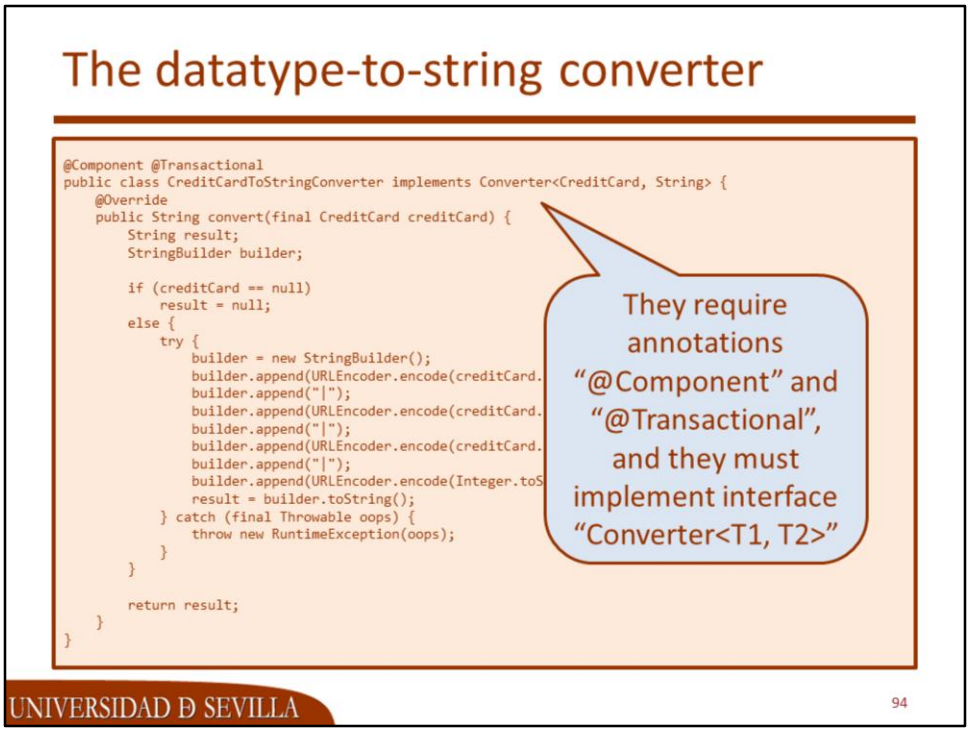


Let's now explore the data type converters.

The 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 datatype-to-string converter that transforms credit cards into strings.



Like the previous converters, it requires annotations "@Component" and "@Transactional" and it must implement interface "Converter<T1, T2>".

The 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;
    }
}
```

Serialise the object as a string of the form "$attr_1|attr_2|\ldots|attr_n$"

The "convert" method is pretty simple: it serialises the object as a string of the form "$attr_1|attr_2|\ldots|attr_n$". We only have to make sure that the attributes are encoded in such a way that they don't have any vertical bars, since that would make decoding them ambiguous. To ensure it, we use class "URLEncoder", which provides an "encode" method that transform special characters like the vertical bar into url-encoded characters. For instance, a string like "My|Card!" is encoded as "My%7CCard%21".

The 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;
    }
}
```

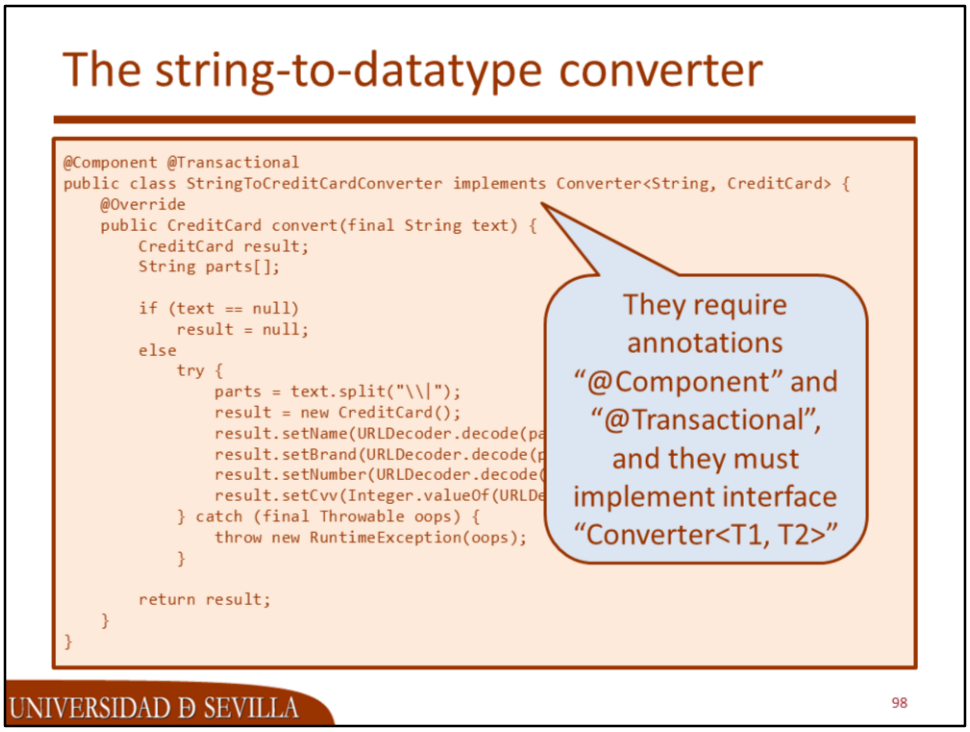
Or throw an exception if something bad happens

If something bad happens, obviously, an exception must be thrown.

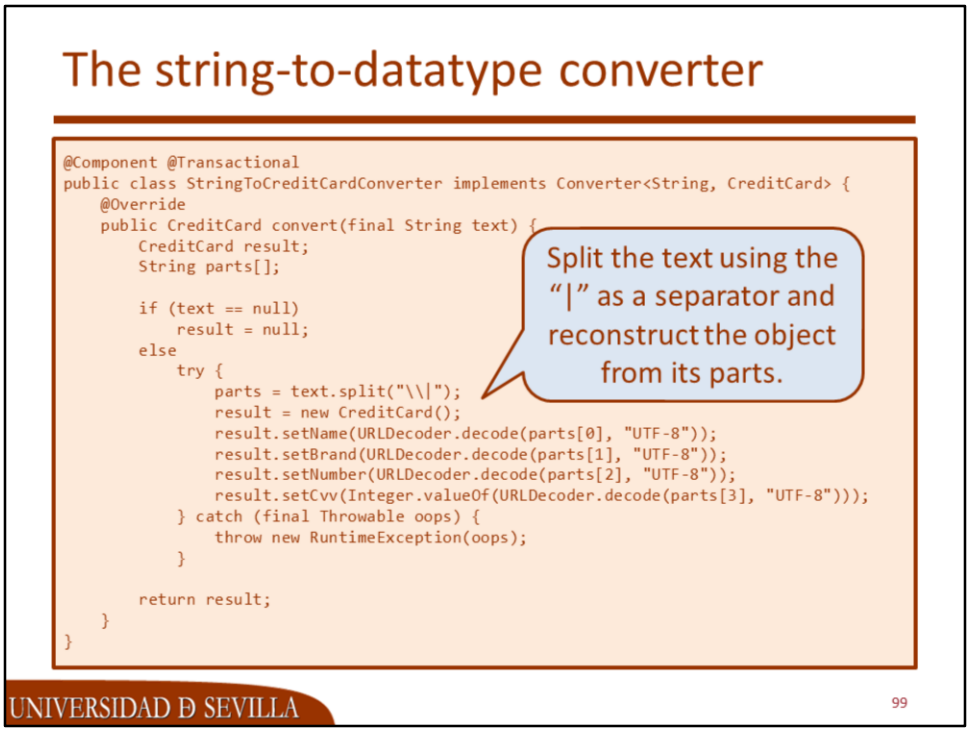
The string-to-datatype converter

```
@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;
    }
}
```


And this is the counterpart method: the string-to-datatype converter.



As usual, the converter requires annotations "@Component" and "@Transactional", and it must implement interface "Converter<T1, T2>"



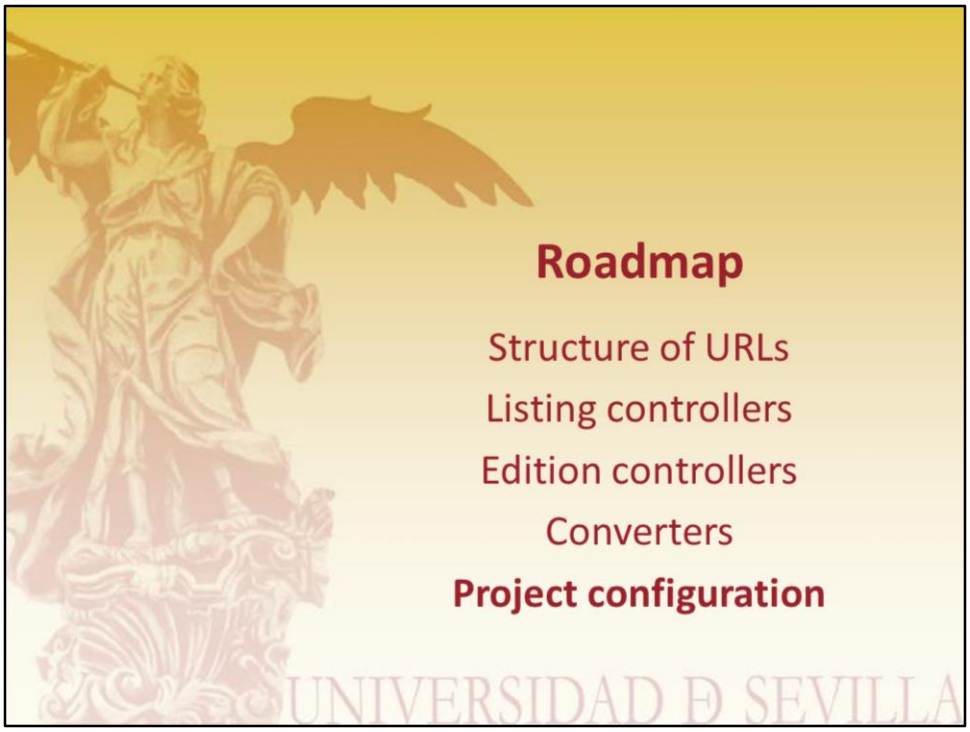
The "convert" method is pretty straightforward: it first splits the input text using the vertical bar as a separator and then reconstructs the original object from the parts. Note that the "URLDecoder" class is used to decode the parts so that every attribute looks as it was in the original object. For instance, string "My%7CCard%21" is decoded back as "My|Card!".

The string-to-datatype converter

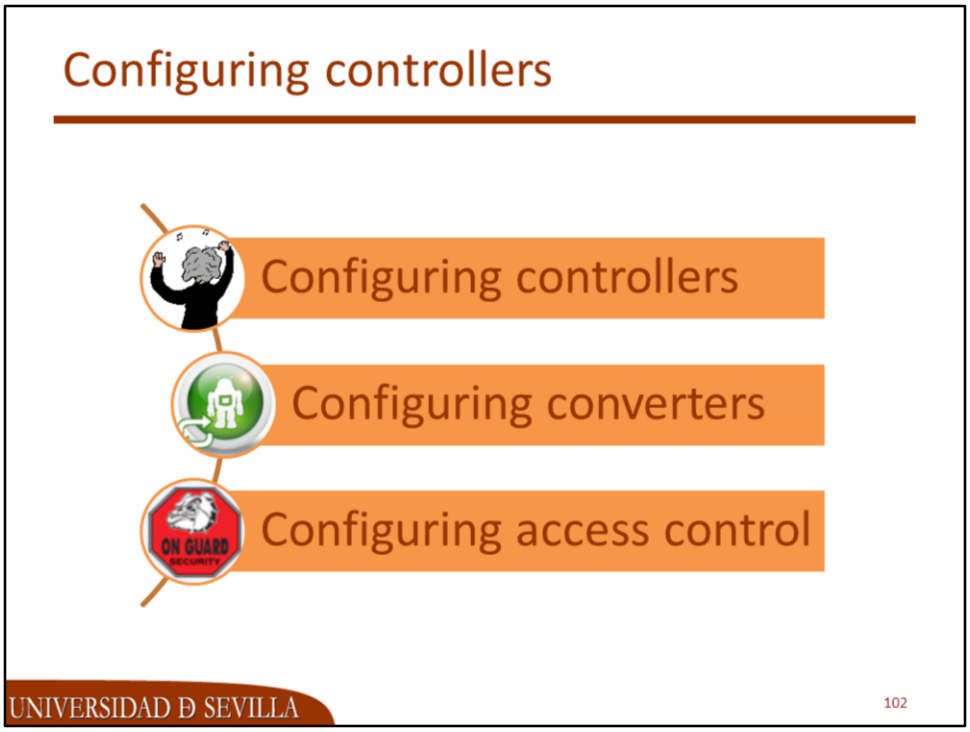
```
@Component @Transactional
public class StringToCreditCardConverter implements Converter<String, CreditCard> {
    @Override
    public CreditCard convert(final String text) {
        CreditCard result;
        String parts[];

        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;
    }
}
```

Or throw an exception if something bad happens

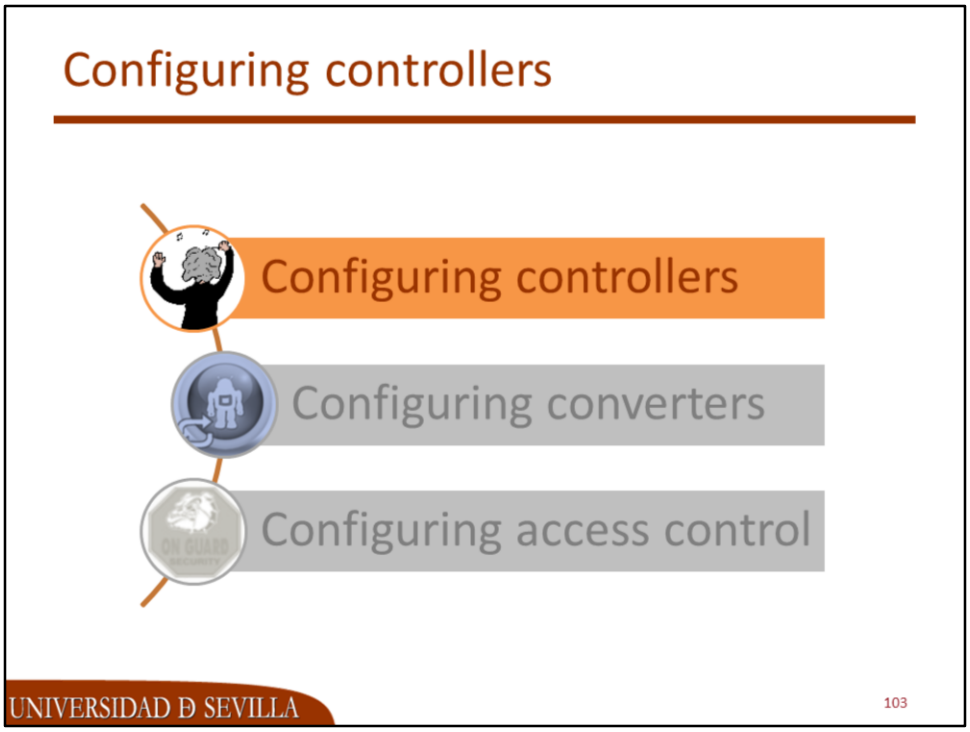
As usual, if something bad happens, then an exception is thrown.



Finally, it's time to report on how you have to configure your project.

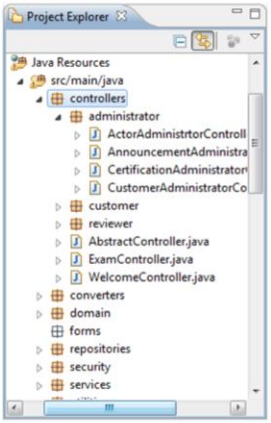


We have to study how to configure the controllers, the converters, and the access control.



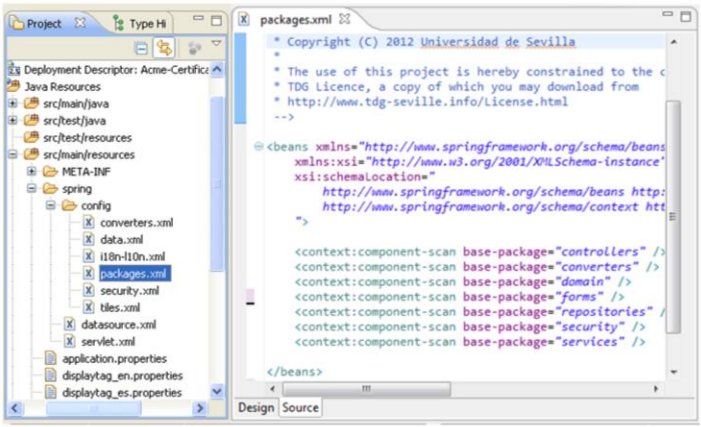
Let’s start with configuring controllers.

The controllers package

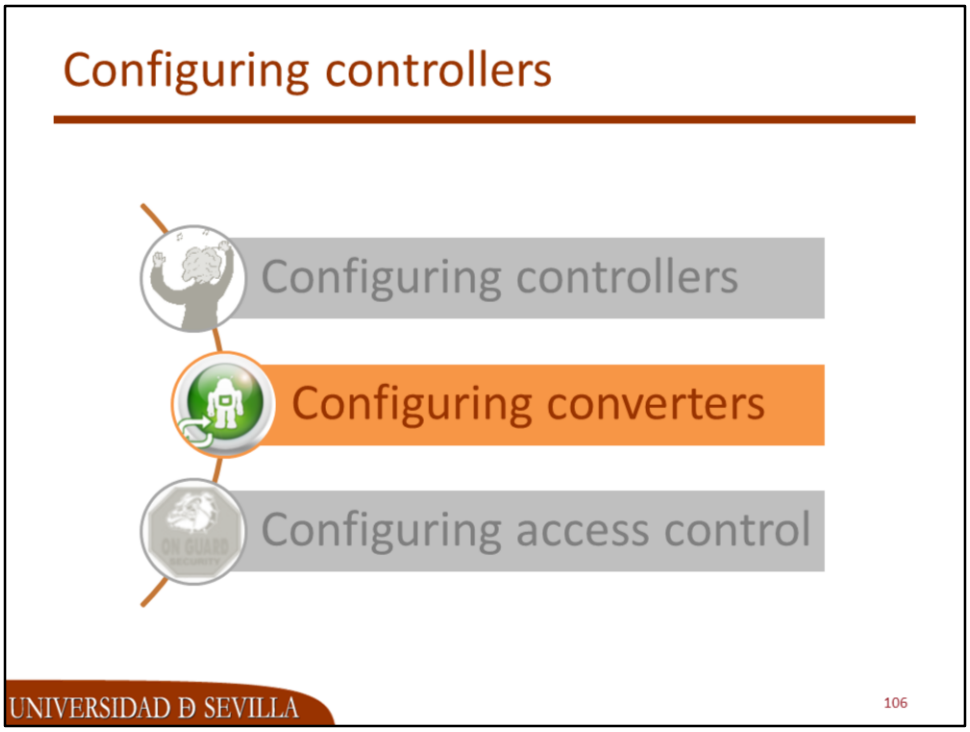


You must add your controllers to a package called “controllers”. Please, note that the abstract controller and controllers that serve public URLs are listed directly in this package, whereas the remaining are grouped into sub-packages that are named after the corresponding entities. Fortunately, that’s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

The packages.xml configuration file

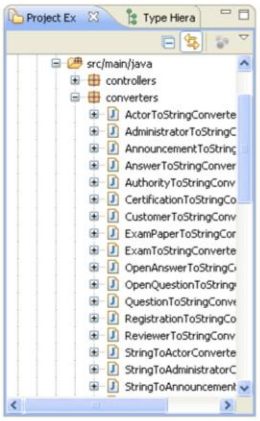


Please, take a look at a configuration file called “packages.xml”. This file specifies the packages that Spring must scan for controllers and other components: converters, domain entities, form objects, repositories, security artefacts, and services. Just learn that we need to list the names of the packages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



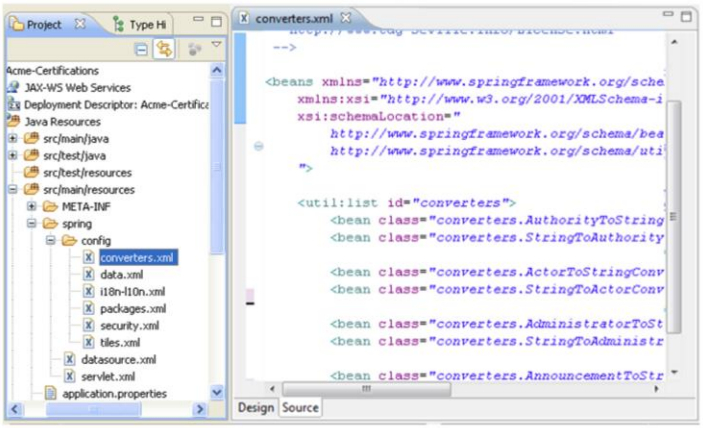
Let’s go on with configuring the converters.

The converters package

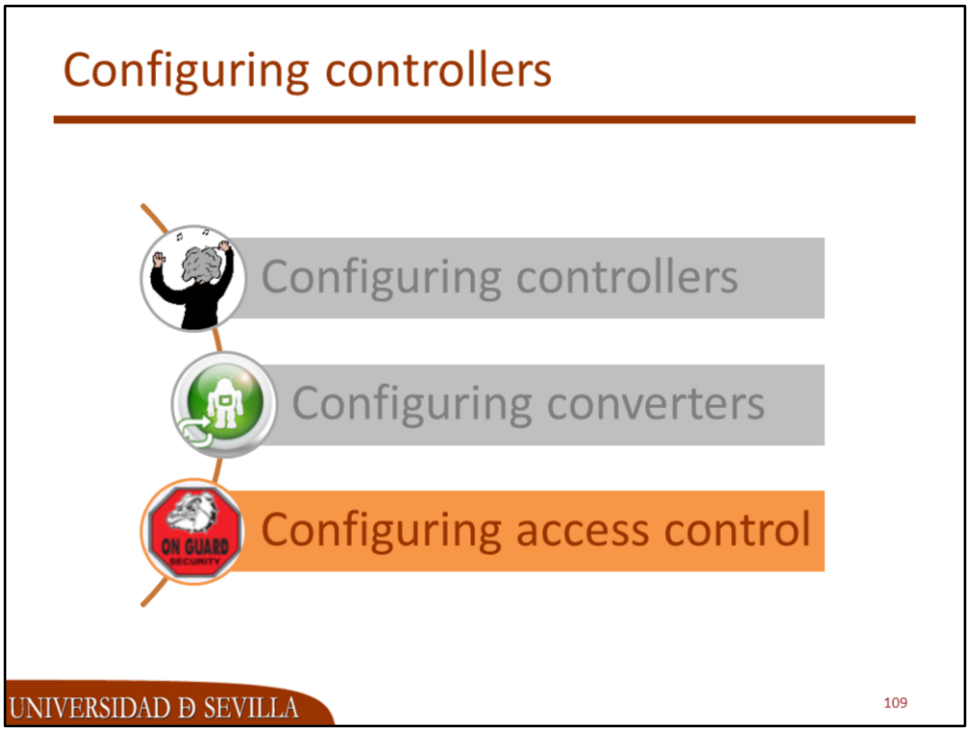


You must add your converters to a package called “converters”. Unfortunately, this isn’t enough; please, keep reading.

The converters.xml file

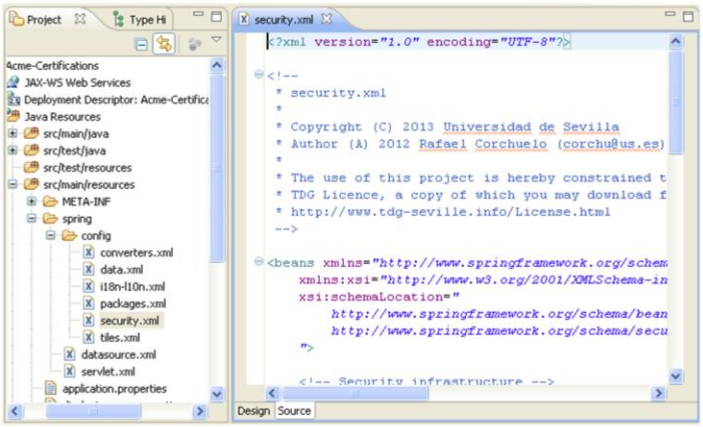


There is a configuration file called “converters.xml” in which you must explicitly list them all, one after the other.



And let's finally explore how to configure the 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.

Then comes a long section in which you must configure the access control to the paths in your system. In this slide, we show the part that configures the access to a number of predefined paths:

- "/": 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.

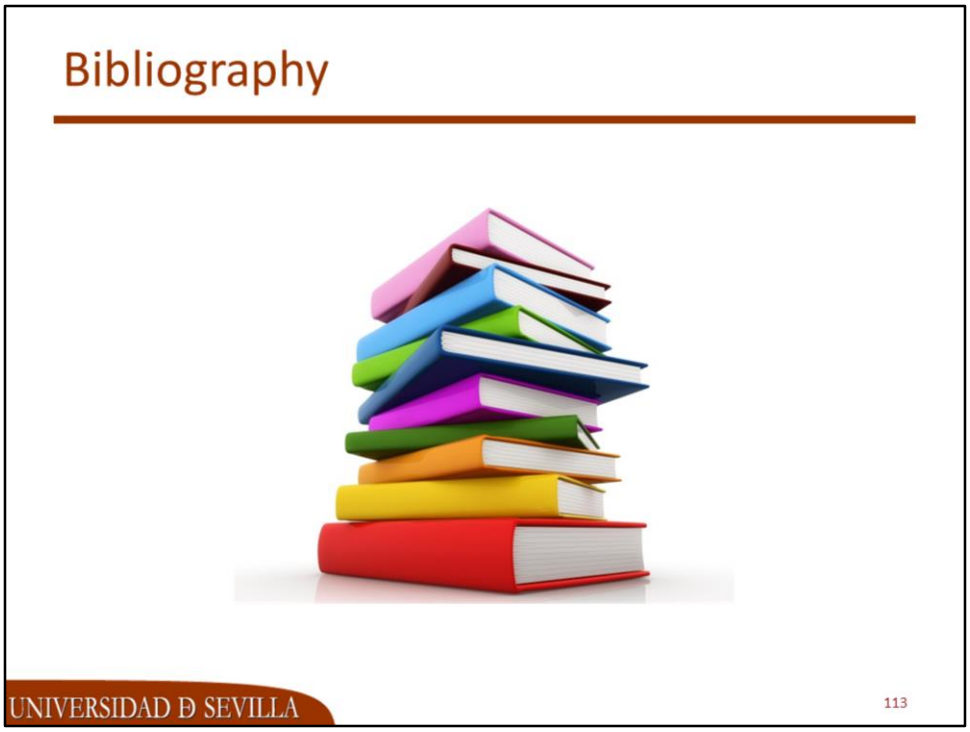
Access control (II)

```
<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>
```


In this slide, we show the application-specific paths. For instance, we specify that only users with authority "ADMIN" can have access to paths 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 path 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 path that hasn't been listed previously. That is, you need to list every path explicitly or, otherwise, no-one will have access to them. This policy is very desirable for security reasons.



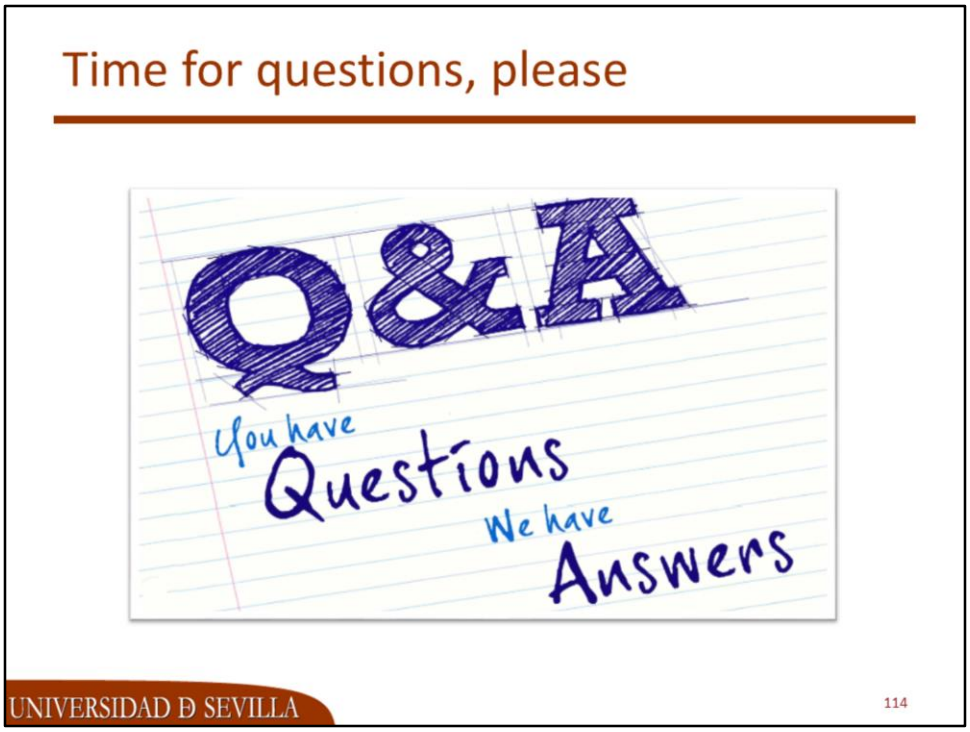
Should you need more information on the topics we've presented, please take a look at any of the following books:

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

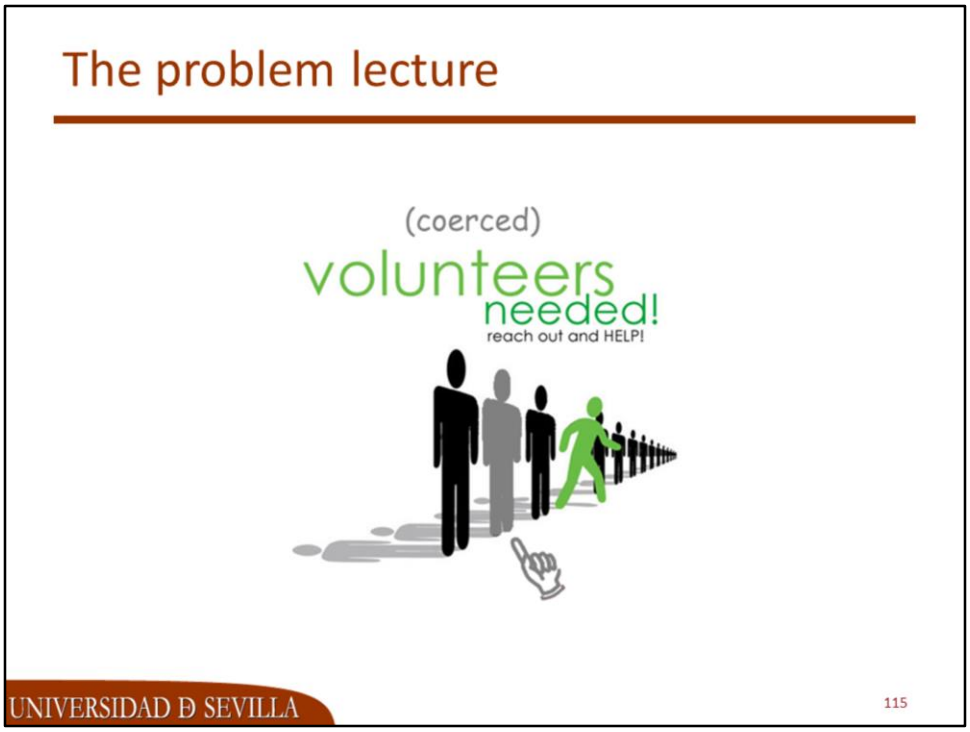
Pro Spring MVC with Web Flow
Marten Deinum, Koen Serneels, Colin Yates, Seth Ladd, Christophe Vanfleteren
Springer, 2012

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.

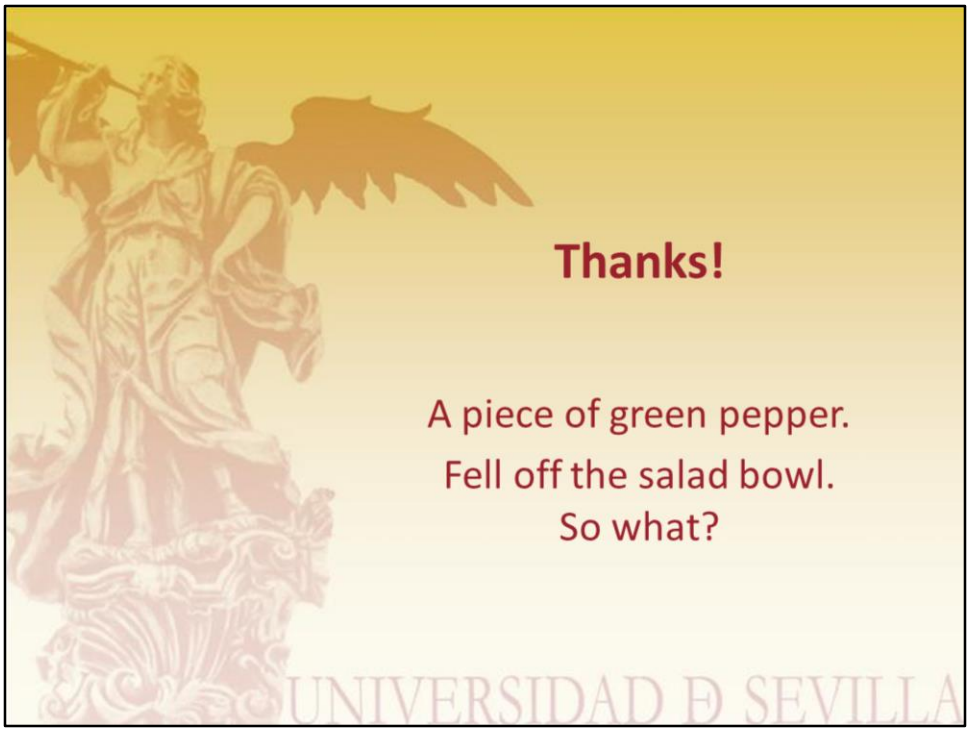
You might also be interested in the electronic documentation provided by Spring Source, the company that manufactures Spring. It's available at http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/htmlsingle/.



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



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



Thanks for attending this lecture!