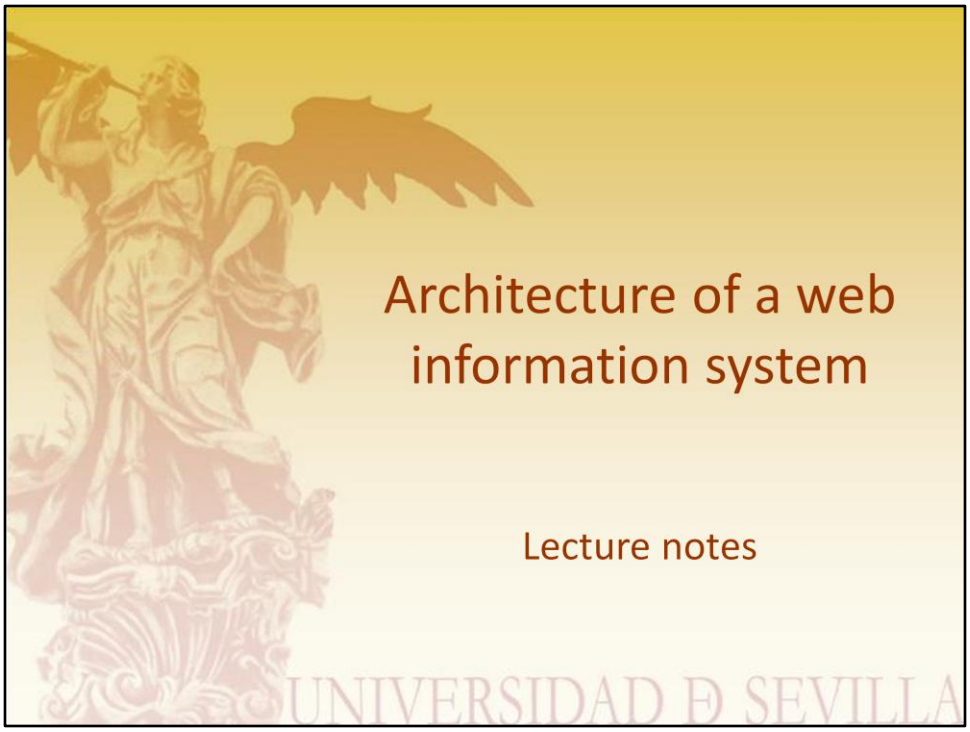


Welcome to this lecture. Today, our goal's to provide a foundation on the architecture that we are going to use in this subject.

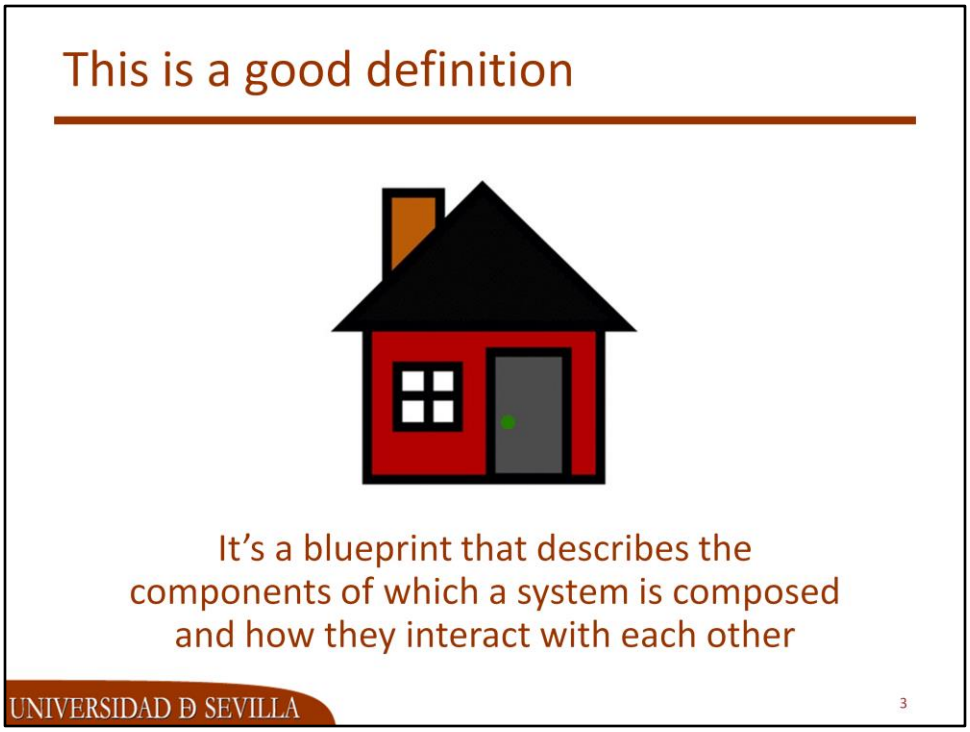
--

What's an architecture?

UNIVERSIDAD Ð SEVILLA

2

It's then a good idea that you try to produce a definition before going on. Please, make a point of defining what an architecture is before peeking at the following slides.



This is the definition of architecture that we're going to use in this subject: it's a blueprint that describes the components of which a system is composed and how they interact with each other. For instance, this slide presents a simplistic house blueprint; it's not an actual house, but your may easily recognise some of the components of which every house is composed. The architecture of a web information system is somewhat similar: it describes a generic web information system, not a particular one.

NOTE: terms "architecture" and "component" are quite polysemous in Software Engineering. That means that the same term is used to refer to a variety of different artefacts. For instance, in this lecture, we use term component to refer to architectural components; in the following lecture, we'll use it to refer to software components. It's important that you get accustomed to the vocabulary and that you can easily make a difference between the many different meanings of these words depending on the context.

How are they devised?

UNIVERSIDAD Ð SEVILLA

4

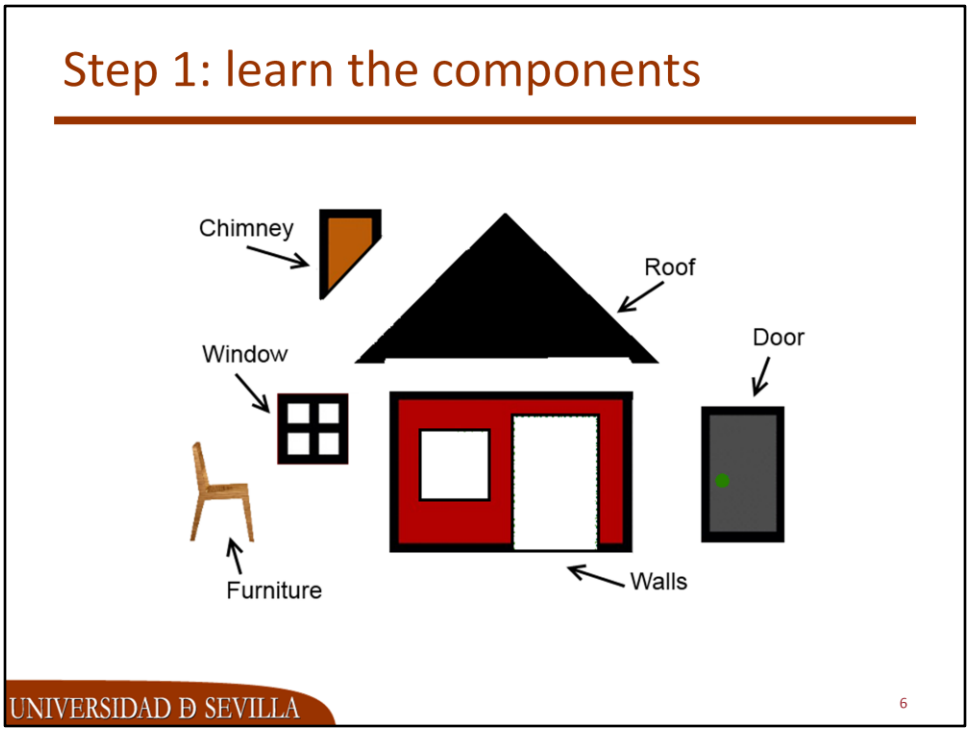
It’s now time to ask yourself how an architecture may be devised. Please, try to answer this question without peeking at the following slides.

Starting point: others' experience

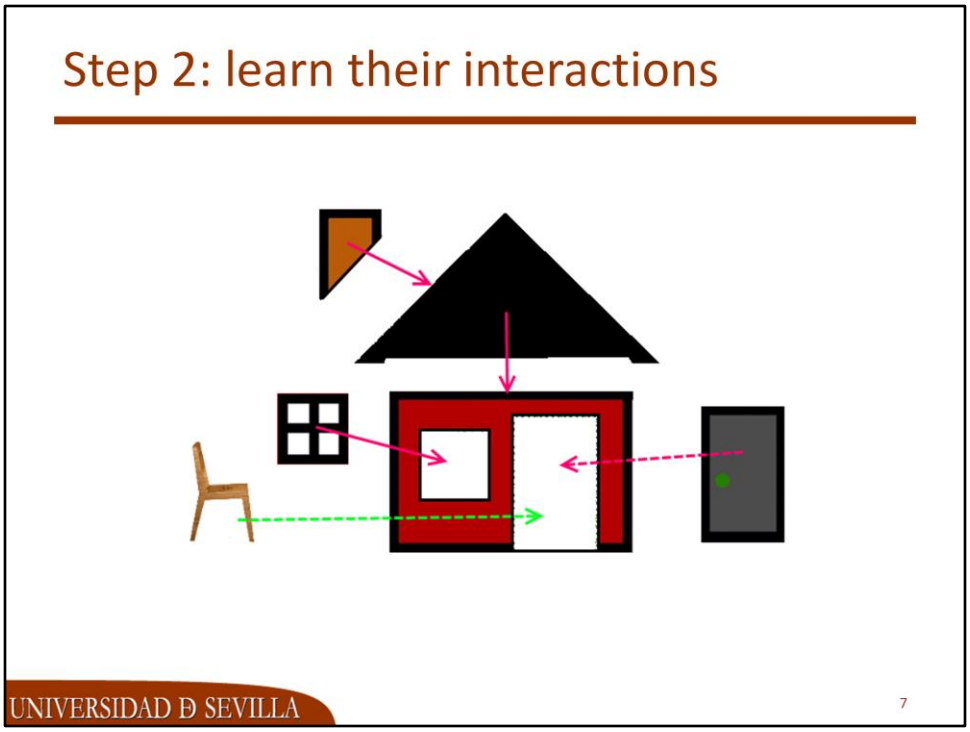
UNIVERSIDAD D SEVILLA

5

The starting point to devise a good architecture is other people's experience. Many software engineers before us have faced the problem of building a web information system, and they have explored a lot of paths that don't work and a few paths that work. We can easily leverage their experience and build on proven architectures to create our own web information systems. Furthermore, good architectures are typically backed by software frameworks that help even more in the task of building a web information system.



We don't think this should be surprising to you: the first step to command an architecture is to understand the components that it defines. For instance, this slide shows a simplistic blueprint of a house. We said it before: it's not a house, but you can easily identify walls, doors, windows, roofs, chimneys, and some furniture, which are the typical components of which every house is composed. The components of a web information system are far different, but they play the same role.



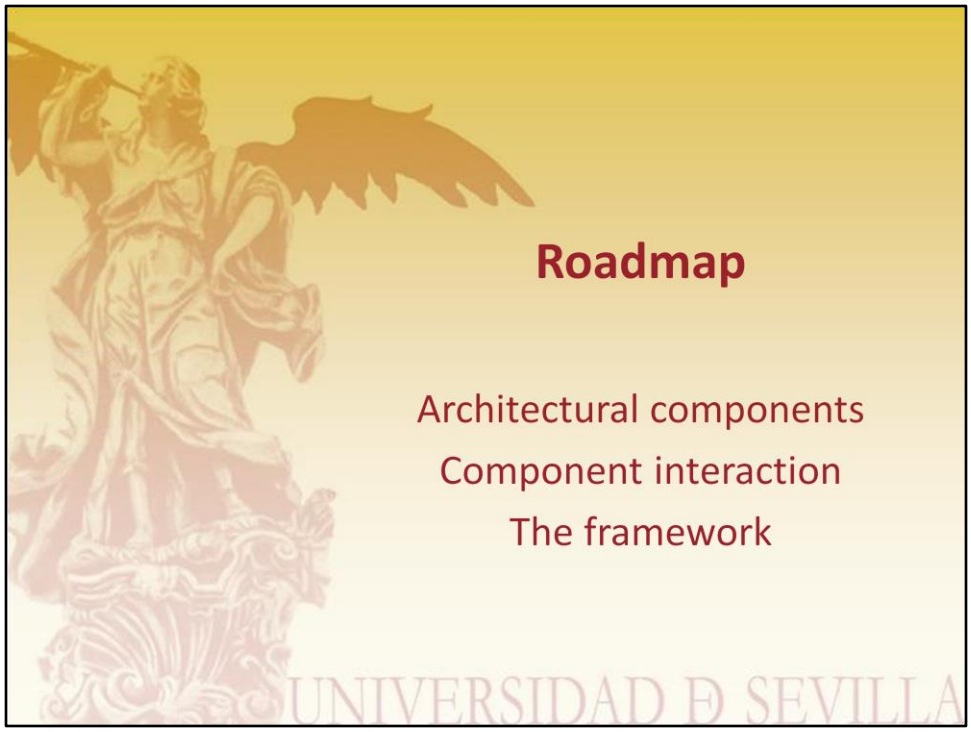
The second step is to learn how the components interact. In our example, for instance, we know that the furniture must be placed inside the house, that the roof must be placed atop the walls, that the chimney must be placed above the roof, and that the door and the windows must be placed in the walls. In a web information system, the interactions are far more complex, but they play the same role.

Step 3: learn a framework

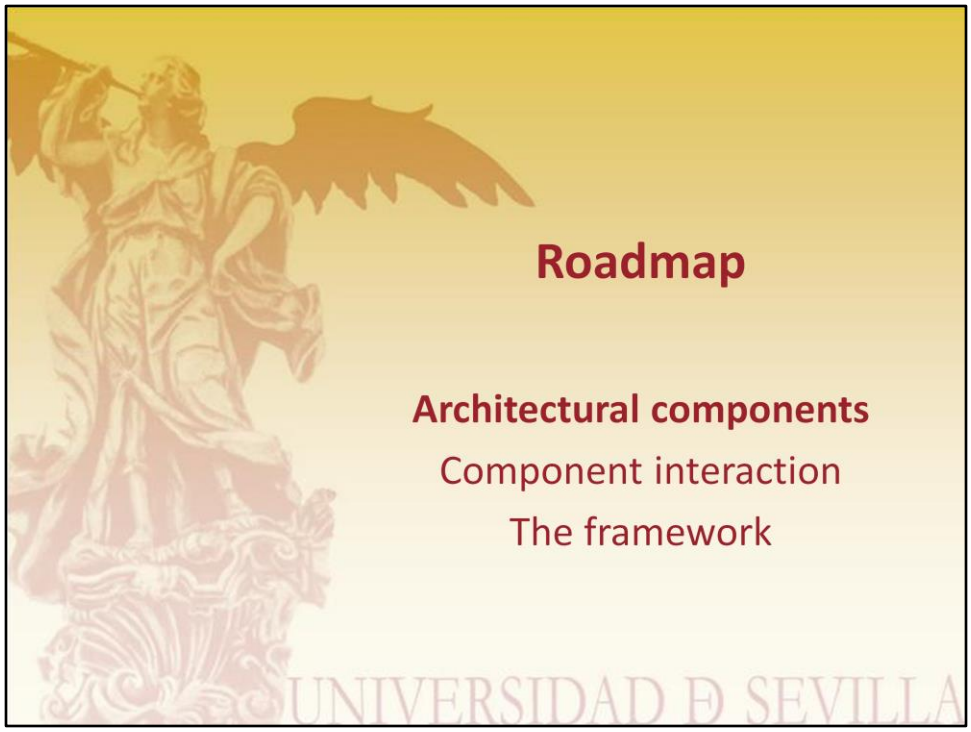
UNIVERSIDAD Ð SEVILLA

8

The third step is to instantiate the architecture, that is, to materialise it so that we can build an actual web information system. We can do that from scratch, but it's not a good idea. A lot of software engineers must have already implemented web information systems that build on the architecture that we're going to study in this lecture. That means that someone must have already devised kind of a template that we can use as a starting point. Such a template is usually called a framework. For instance, the house in the slide's a framework: it's not a complete house, but it provides an instantiation of many of the components in the architecture: it provides wooden walls, aluminium windows, a security door… the user only has to provide a chimney and some furniture so that the house is complete. Having a good framework will obviously save us from a lot of repetitive work.

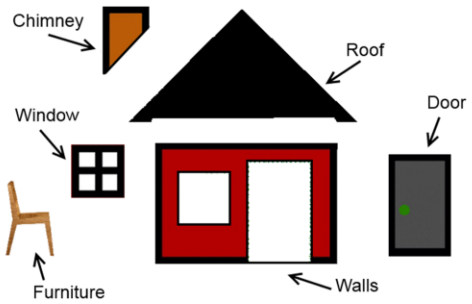


The rest of the lecture is organised as follows: we first will delve into typical architectural components, then into how they interact, and, finally, we'll present some details about the framework that supports the architecture that we're going to use in this subject.



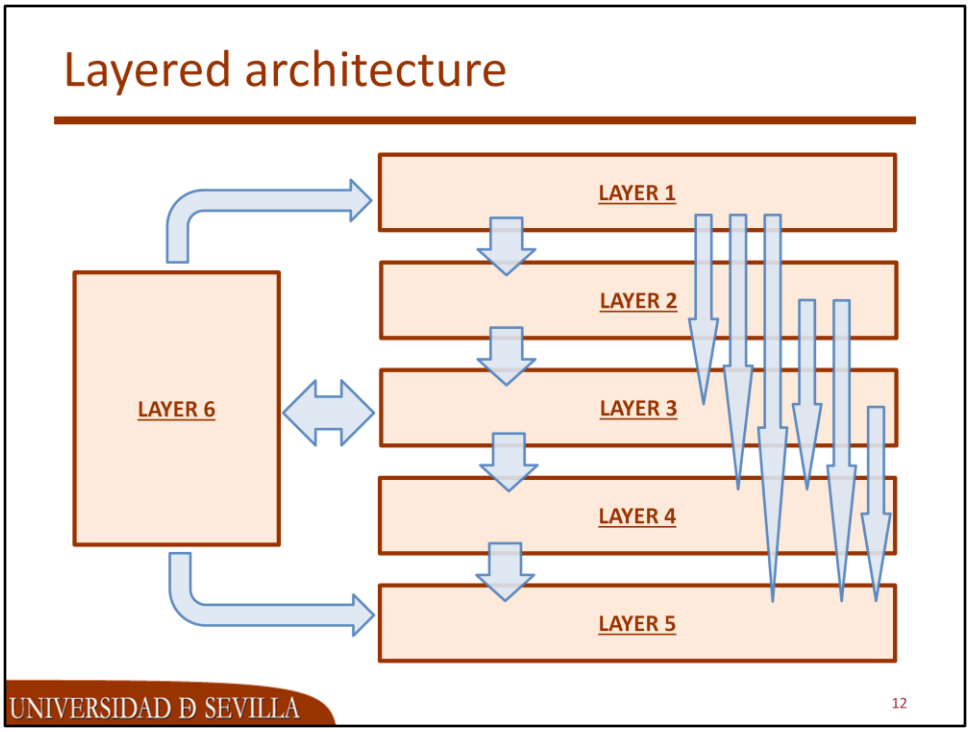
Let's start with a description of our architectural components.

What are them?

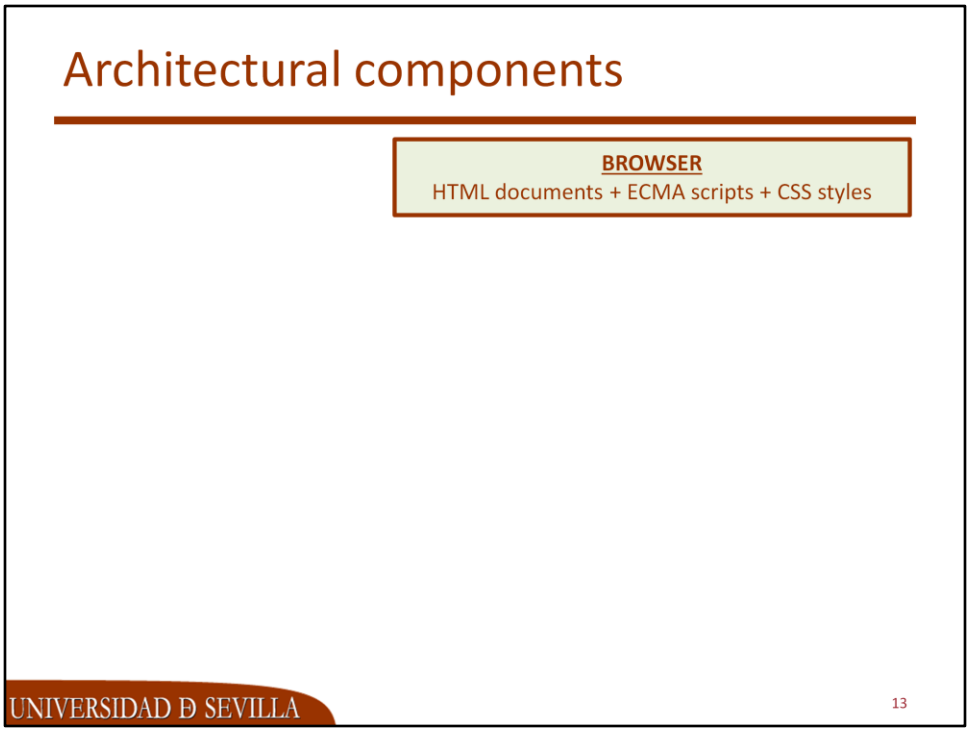


They are the pieces of which a web information system is composed

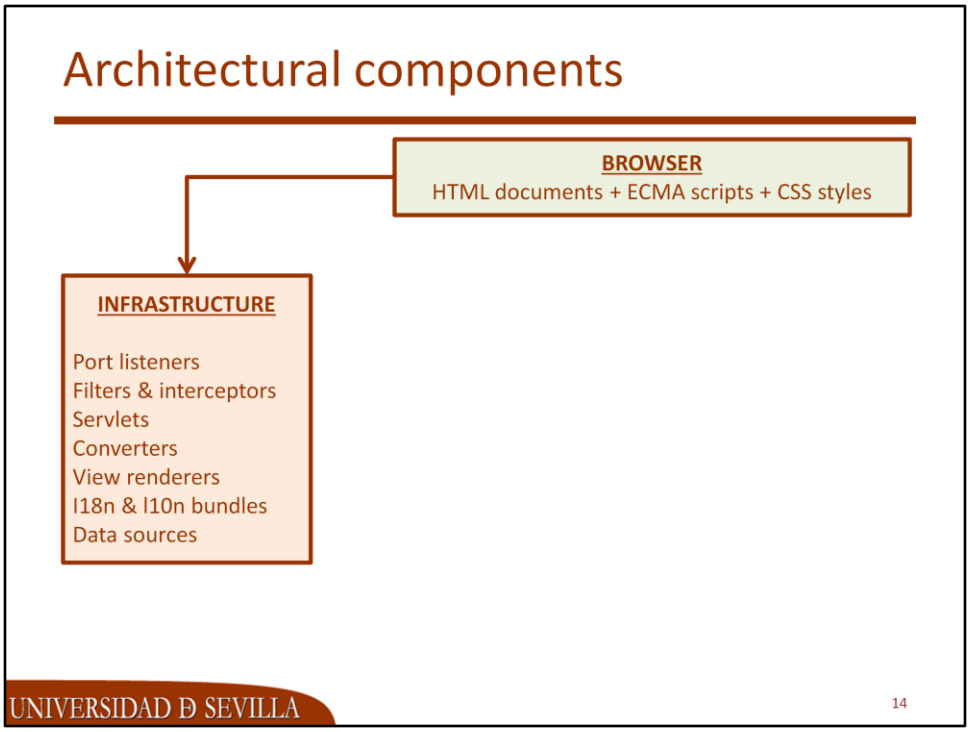
Recall from the introduction to the lecture that the architectural components are the pieces of which a web information system is composed. Unfortunately, our architecture has a lot more components than this toy house and they are far more complex. In other words, we need a means to organise them logically so that we can understand them more easily.



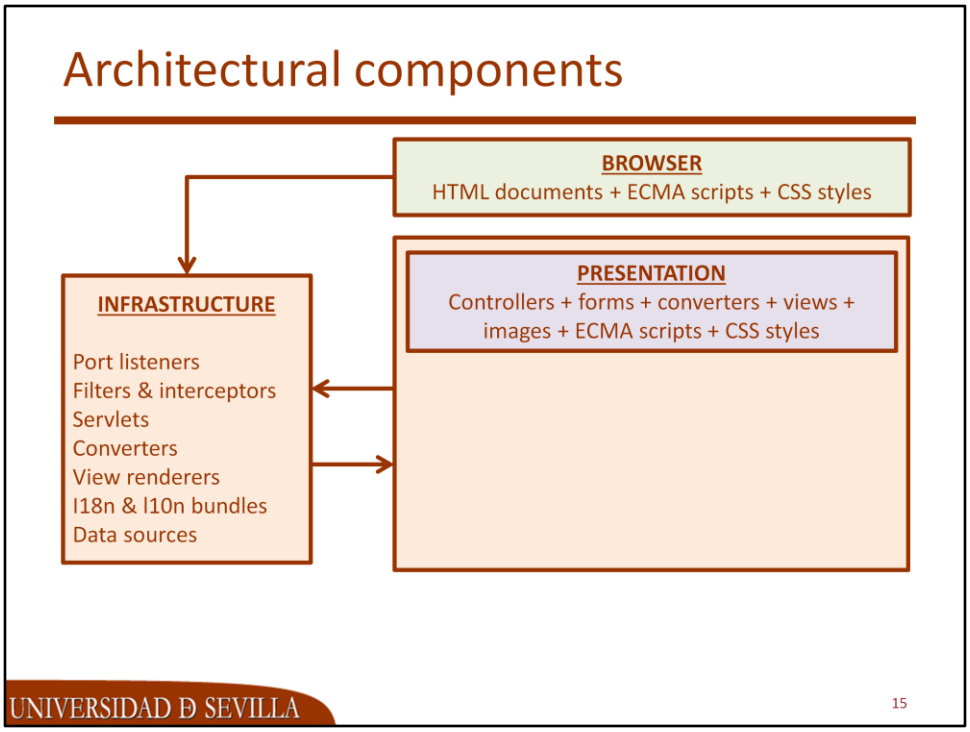
It's common to organise the architectural components into layers. A layer is kind of a package that helps us organise a number of artefacts that are related to each other. Artefact is a term that is commonly used in Software Engineering to refer to anything related to software: a class, a script, an image, or a diagram, to mention a few examples. Typically, upper layers have access to the artefacts in the layers below them, but not vice-versa. Sometimes, there are so-called side layers that act as intermediaries between other layers. In the next slides, we'll delve into the layers of our architecture.



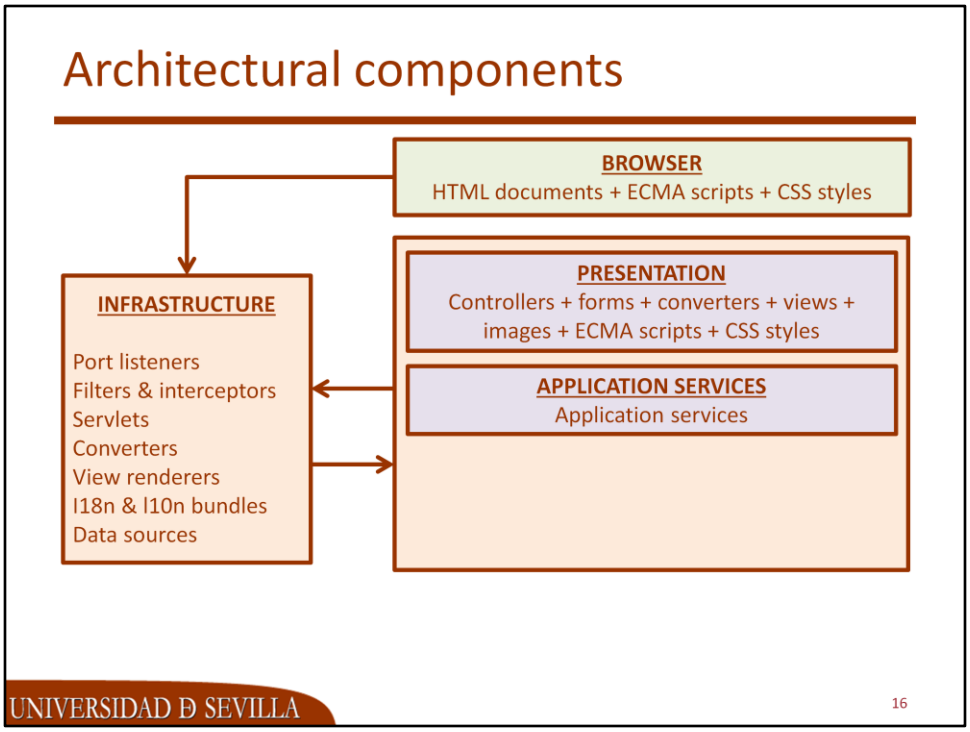
The first layer is the browser layer. The artefacts in this layer are HTML documents, ECMA scripts (this is the official name of JavaScript), and CSS styles. These artefacts provide a browser with a description of a user interface. Such user interface has links and buttons, which constitute the basic mechanism to interact with a web information system.



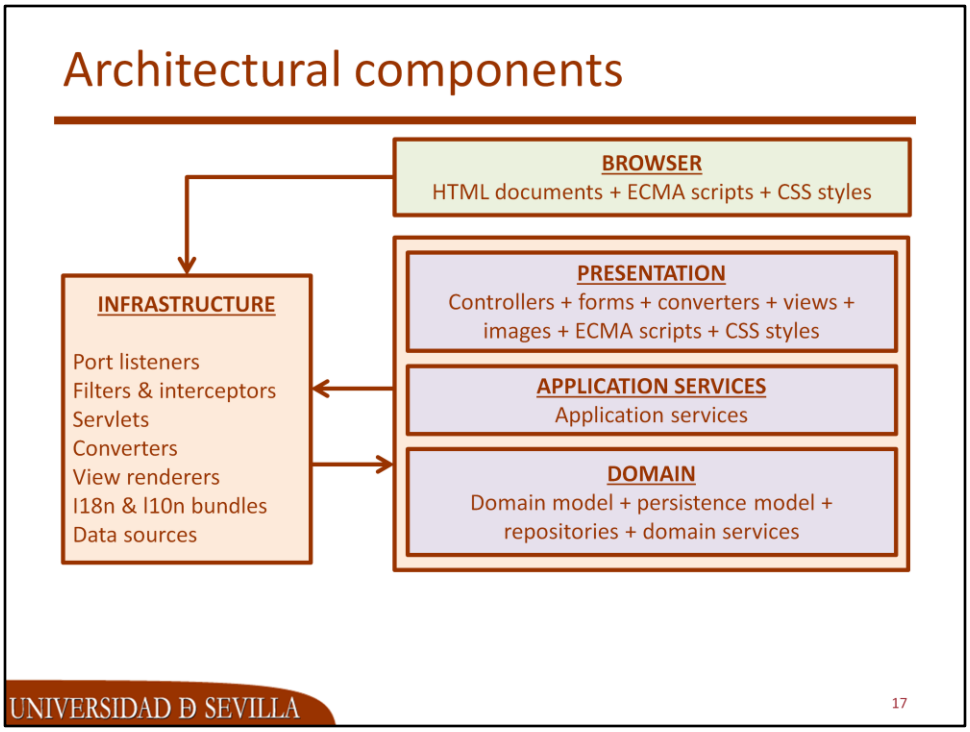
The requests that originate from the browser (clicking on a link or a button) are intercepted by one or more artefacts in the so-called infrastructure layer. The key feature of this layer is that its artefacts are provided by the supporting framework; developers only have to configure them if necessary, but they don't usually have to write much code in this layer.



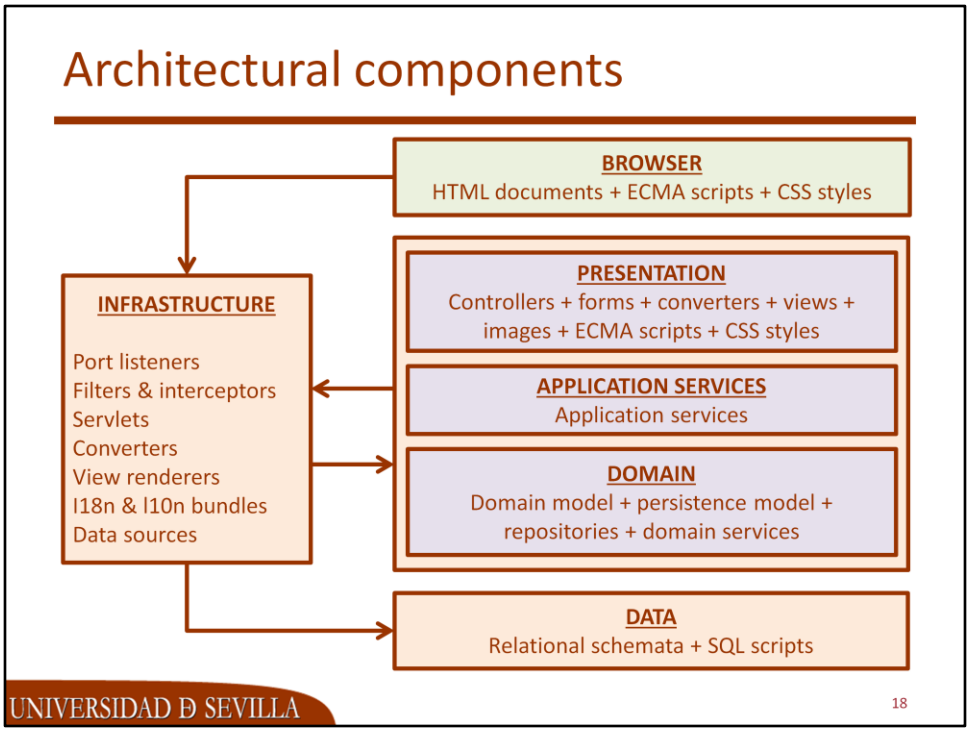
The artefacts in the infrastructure layer interact with the artefacts in the presentation, the application services, and the domain layers. The presentation layer has many artefacts that are related to generating user interfaces; this includes controllers, forms, converters, views, images, ECMA scripts, and CSS styles. We'll provide additional details on these components later; so far it is enough to know that a controller is a class that provides a number of methods that are executed when the browser requests a given URL; forms are objects that represent the data that is edited in an HTML form; converters are classes that transform objects of one type, e.g., strings, into objects of another type, e.g., domain objects like exams or customers; and views are templates that indicate how some data must be rendered, e.g., in a list or in a form. Images, ECMA scripts, and CSS styles are the typical resources that make up a user interface and they are commonly transferred verbatim to the browser layer.



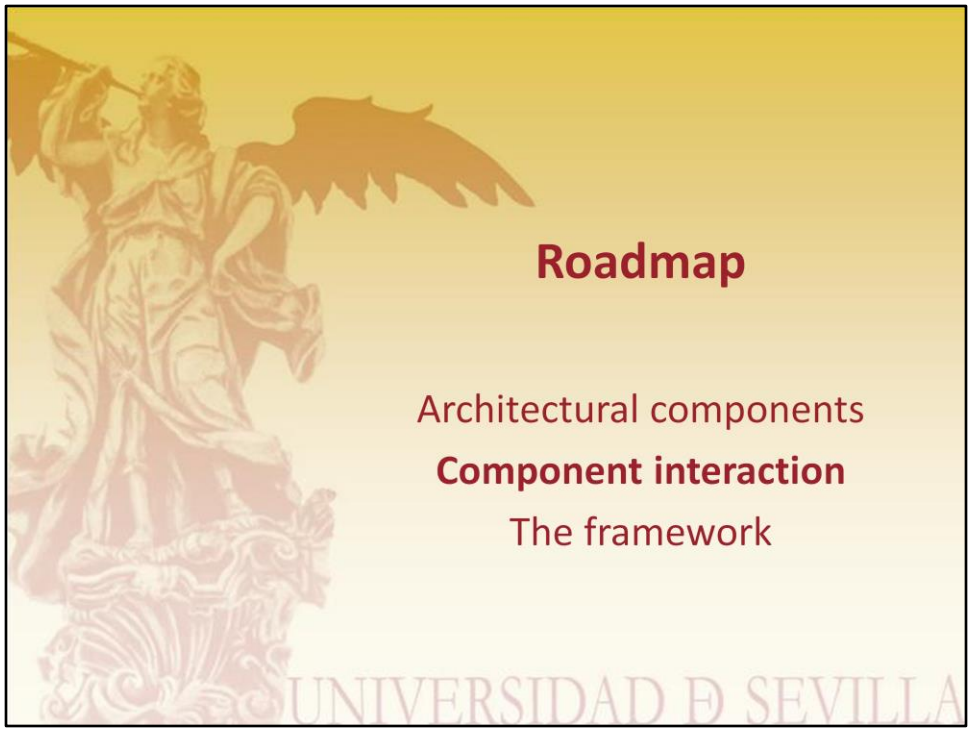
The application services layer is composed exclusively of application services, which are classes that provide methods that implement business logic. For instance, think of a typical use case in which a user can buy something in an electronic mall; the artefacts that are related to creating and handling the user interface are in the presentation layer; the artefacts that implement functionality like adding an item to the shopping cart or checking out are in the application services layer.



The domain layer has a number of artefacts that represent the data that is manipulated by a web information system plus repositories and domain services. The data is represented in two models that are called the domain model and the persistence model; the former provides classes that represent business objects like a cart, an item, or a product; the latter provides a specification of how these classes must be mapped onto a relational database. The domain services are business services that are intimately related to the domain model. Please, do not try to draw a clear line to separate them from the application services; it does not exist. If you aren't sure if some business services should be in the application services layer or the domain layer, put them in the former layer.

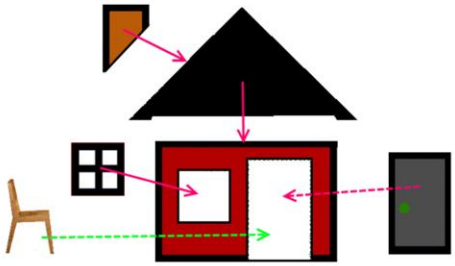


The final layer is the data layer, which is composed of relational schemata and SQL scripts, if any. The good news are that the artefacts in this layer can be generated automatically in quite an efficient manner if our persistence model is good enough. As a conclusion we won't have to worry a lot about the artefacts in this layer.



So far we know a little about the components of our architecture. It's now time to explore how they interact with each other.

What are the interactions about?

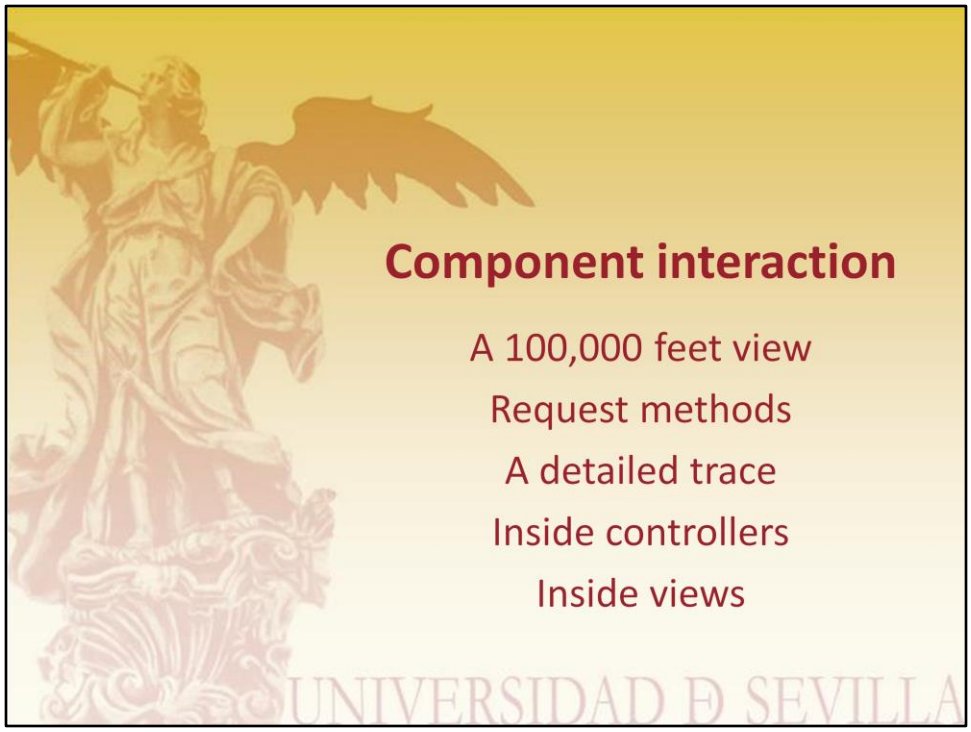


Interactions are about how the components co-ordinate with each other to provide some business functionality

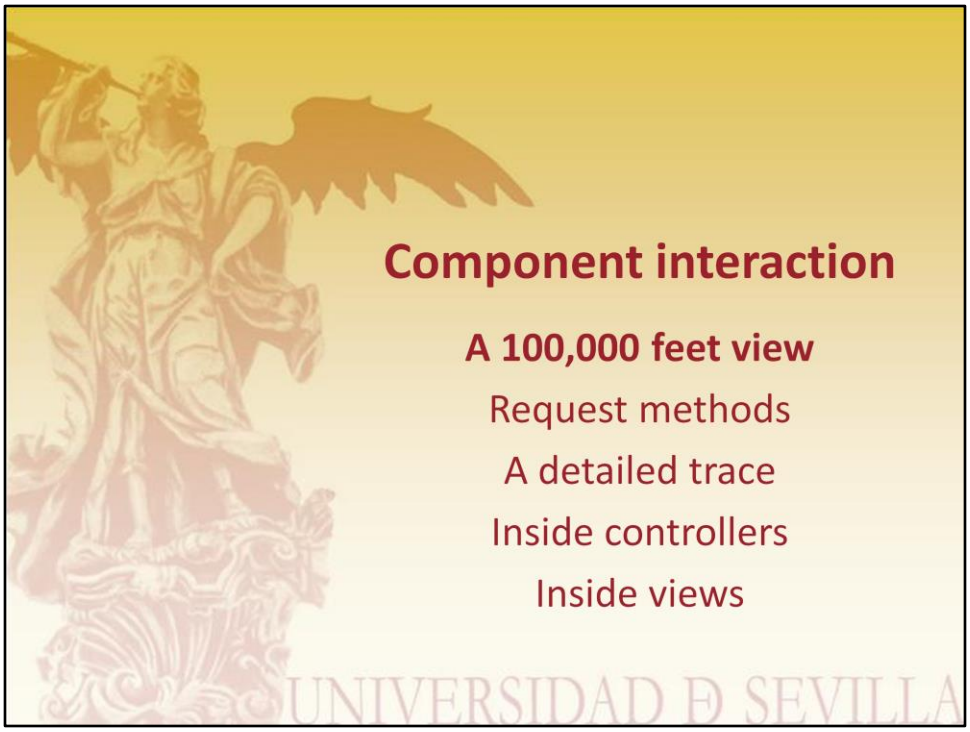
UNIVERSIDAD Đ SEVILLA

20

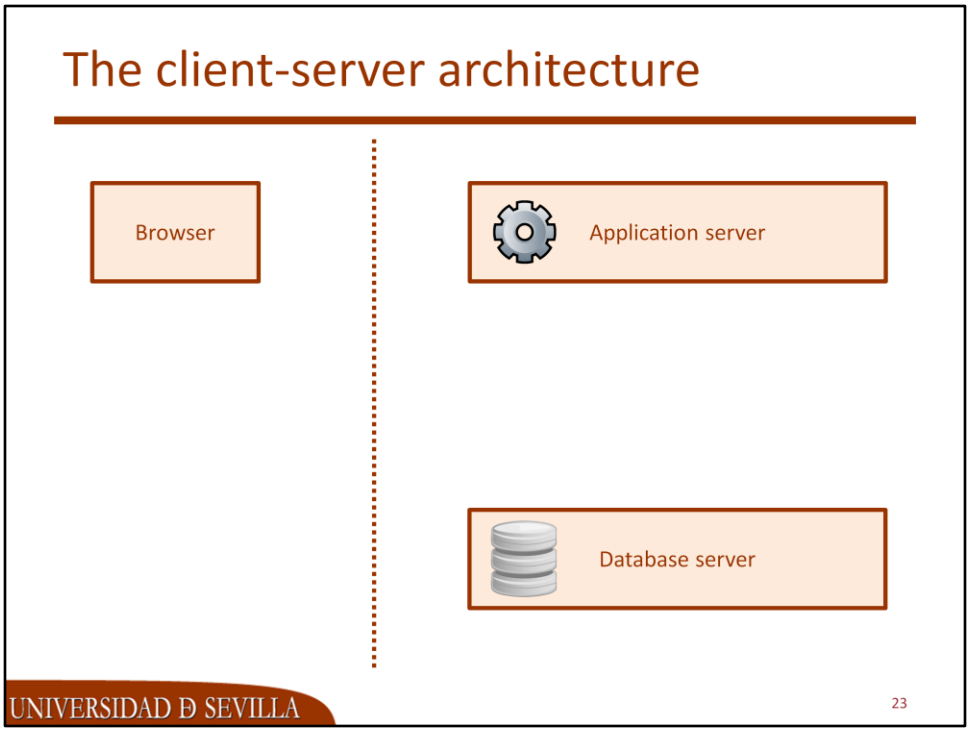
The interactions between the components of our toy house are very simple. The interactions amongst the components that we've presented in the previous section are far more involved. We have explored several ways to present this information to you, and our conclusion was that the easiest way to understand the interactions amongst such a variety of components is to trace what happens when you click on a link or a button in a web page until the server responds with an HTML page, an image, or a similar resource.



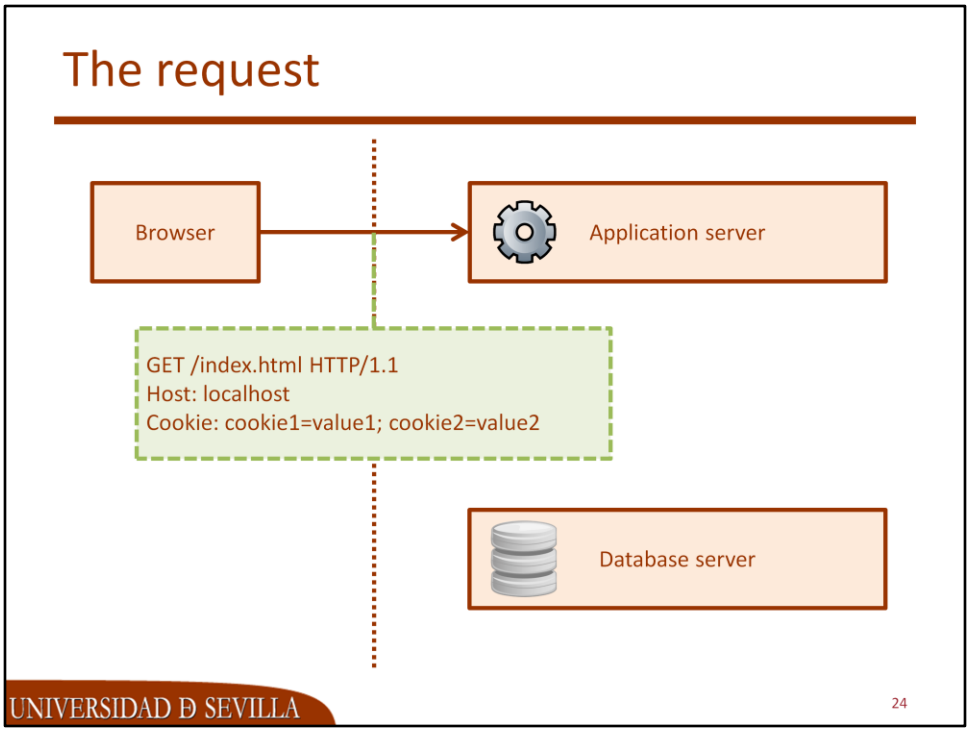
We'll start with a 100,000 feet view of the process; then, we'll report on HTML request methods and we'll delve into every tiny detail to understand how a request is processed; we'll pay special attention to the controllers and the views.



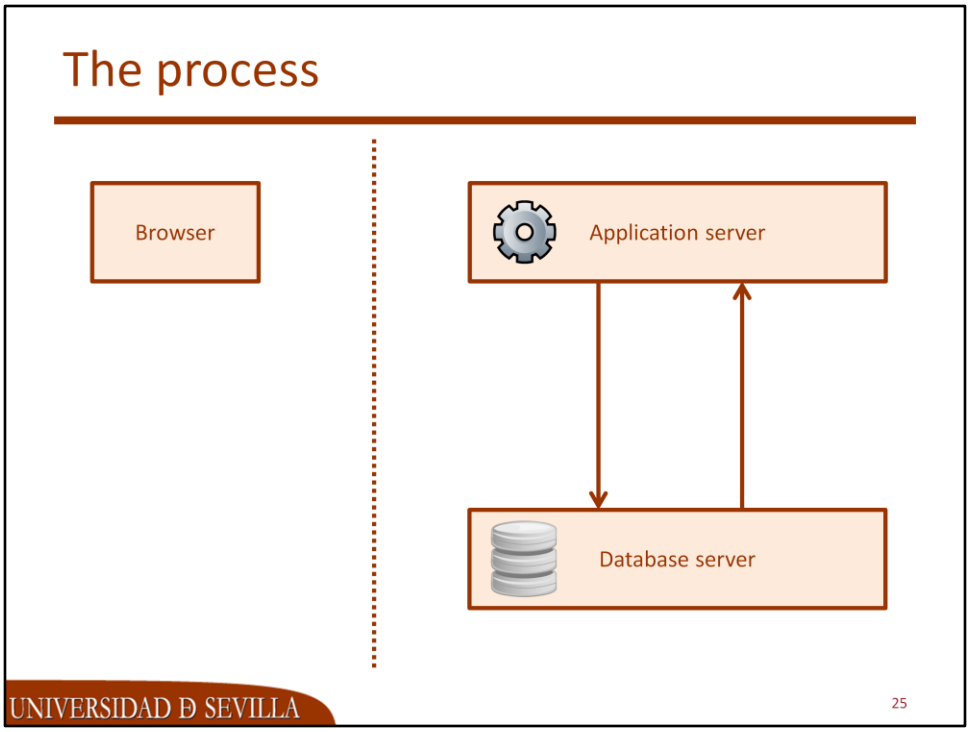
Let's start with a 100,000 feet view of how this stuff works.



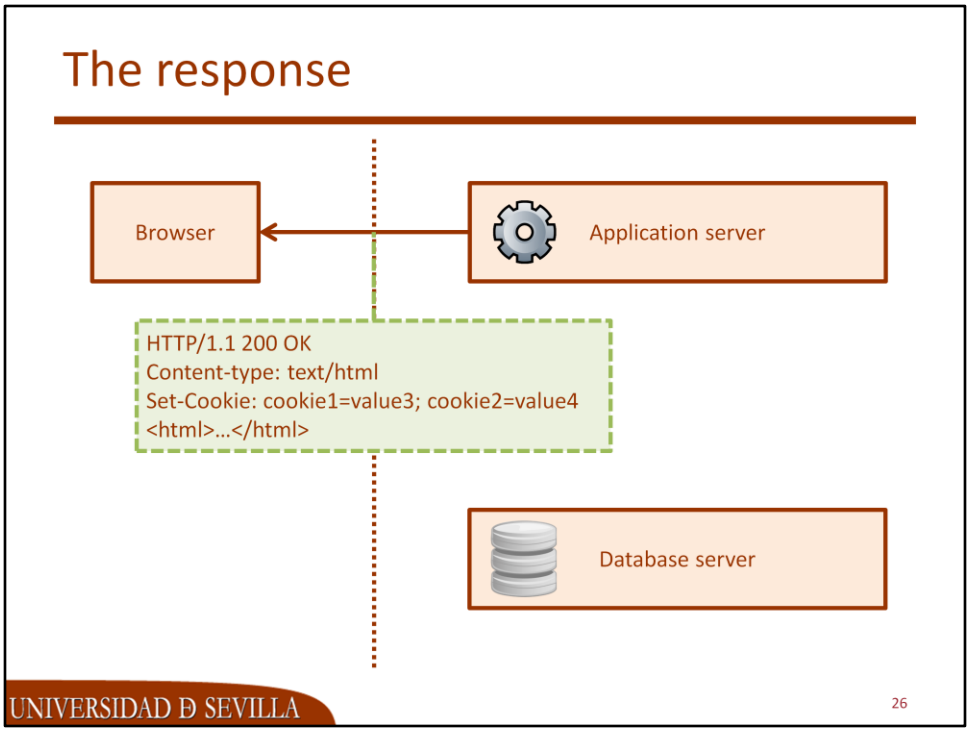
We're pretty sure that you've heard that web information systems rely on a client-server architecture. Don't worry. Earlier in this lesson, we mentioned that "architecture" is quite a polysemous term. Client-server architecture means that a typical web information system involves three subsystems: a browser (the client), which runs on a user's computer, an application server and a database server (the servers), which run on a company's data centre or a cloud. A data centre or a cloud is a collection of computers; the difference is that data centre usually refers to computers that are owned by the company that owns the web information system and cloud refers to computers (usually virtual computers) or platforms (execution engines and servers) that are outsourced from other companies, e.g., Amazon, Microsoft, Cloud Foundry, and a long so-on.



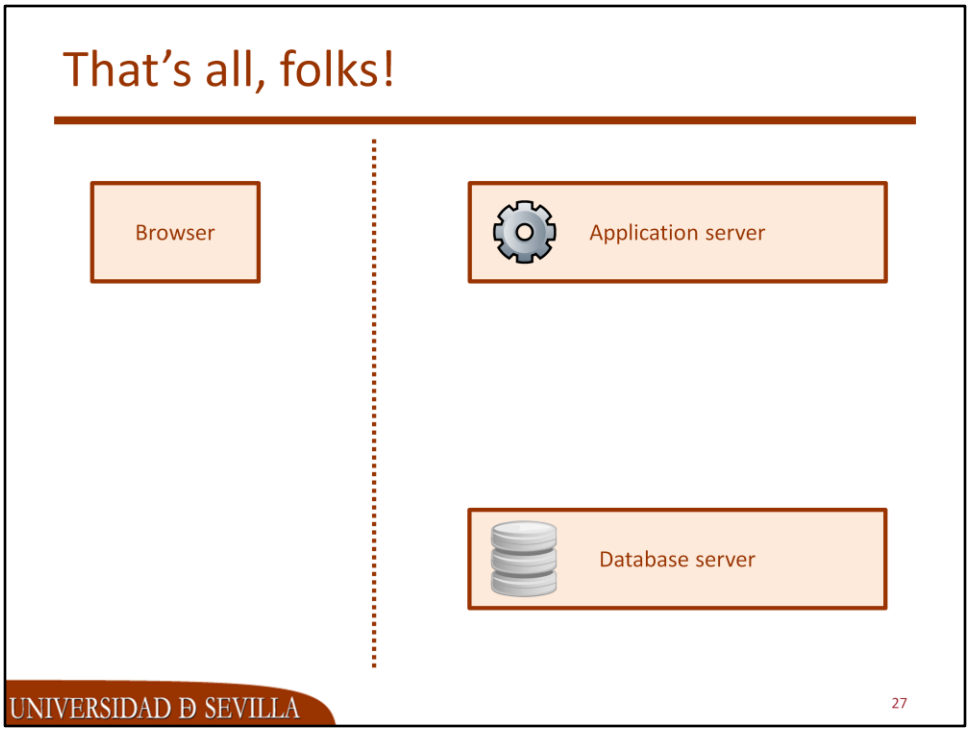
When a user clicks on a link or a button on a web page or keys in a URL in an address bar, the browser generates a request to the server. This request's encoded using a protocol called HTTP. If the user clicks on a link, then the request is similar to the one in this slide: "GET", then the path to a document, then "HTTP/1.1", then the host to which the request is routed, and, optionally, a collection of cookies. You must have heard of cookies: they are small chunks of data that a server sends to a browser, and the browser returns them whenever it makes a request to the server. Doesn't make sense to you? Don't worry; we'll report a little more on this later. The request if the user presses a button is very similar. We'll provide more details on such requests later.



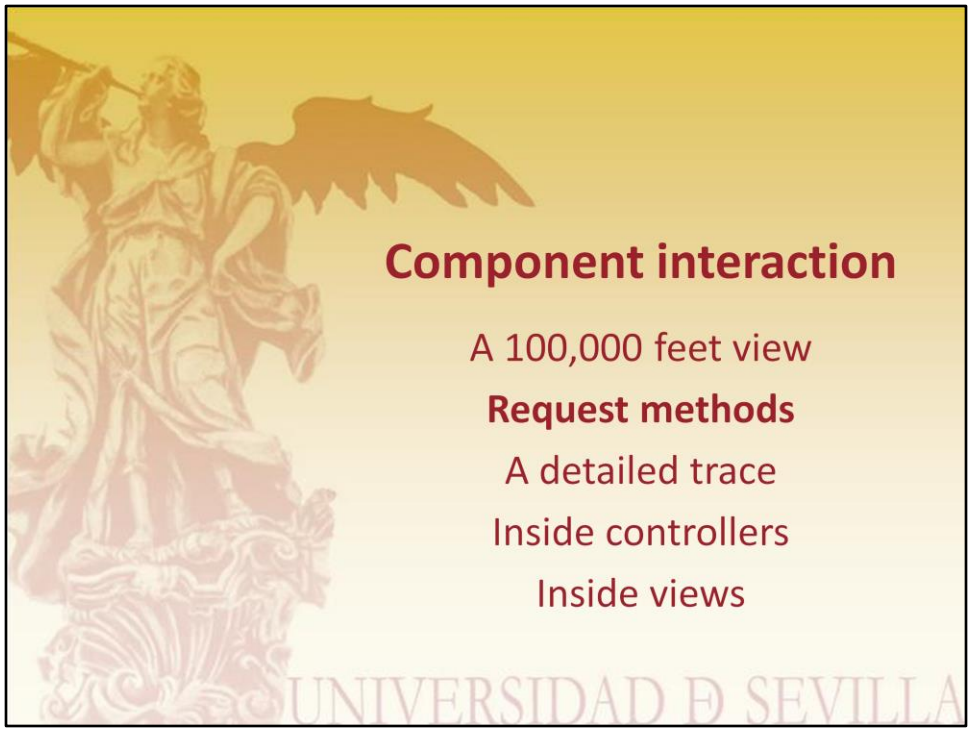
When the request arrives at the application server, it must instantiate a number of objects to serve it. These objects include, at least, a servlet and a controller. We'll provide more details on this later. What matters now is that the application server may request data from the database server, transform them, merge them with data that comes from the request, and perform other complex manipulations, but it must end up producing a response.



The response is typically an HTML document, which is represented in HTTP as shown in this slide. Note that the server may also return a number of cookies that will be stored by the browser; when a new request is made from the browser, the cookies will be sent back to the browser. You may wonder why the server needs to send cookies; why doesn't the server store the cookies in memory or in a file? It's easy. Think of a typical e-commerce server, e.g., Amazon.com. How many browsers do you think are making concurrent requests to Amazon's servers? Do you still think that it makes sense to store that information in memory or local files at the server side? Yes, using cookies is a matter of scalability. You may have heard of something called "server state" or "server sessions" to avoid using cookies; please, forget about that. We won't use those concepts at all since they would prevent us from producing scalable web information systems.



And when the response is delivered to the browser, it renders the user interface and the cycle re-starts when he or she clicks on a new link or button.



Note that the previous view is a 100,000 feet view of what's going on behind the scenes. We still don't know a lot about the details, but we know that a request involves a request method.

HTTP request methods

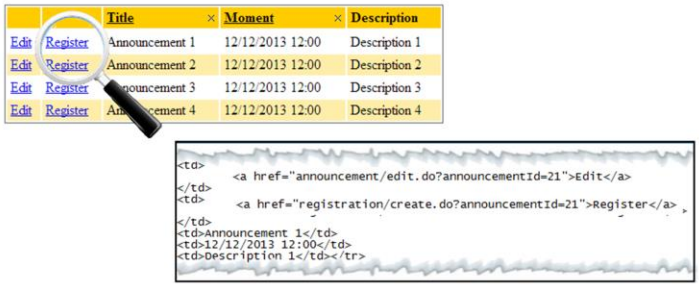
- CONNECT
- DELETE
- GET
- HEAD
- POST
- PUT
- TRACE

In the example before, we used the GET request method; we didn't mention it explicitly, but when a user clicks on a button, that usually results in a request that uses the POST method. There are many other methods available: CONNECT, DELETE, HEAD, PUT, and TRACE. If you're interested, please, take a look at the official RFC that is available at http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html. For the purposes of this subject, it's enough to know about GET and POST.

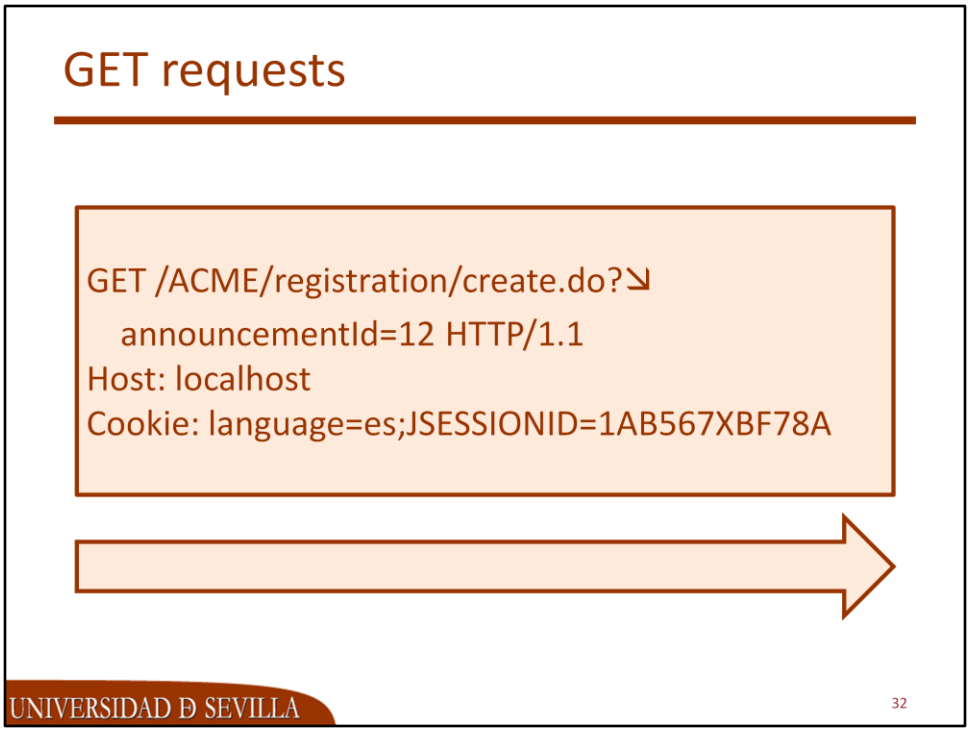
NOTE: a lot of people refer to the HTTP request methods as verbs since they somewhat indicate what the requester wishes to do; the object on which the verbs act are specified differently depending on the verb.

We'll restrict our attention to the GET and the POST methods since they are the only ones that are supported by HTML. We are sorry, no other methods are available natively in HTML, although advanced developers can use ECMA script to implement them. Fortunately, GET and POST is enough in the majority of cases, so this won't be a serious limitation.

The GET method



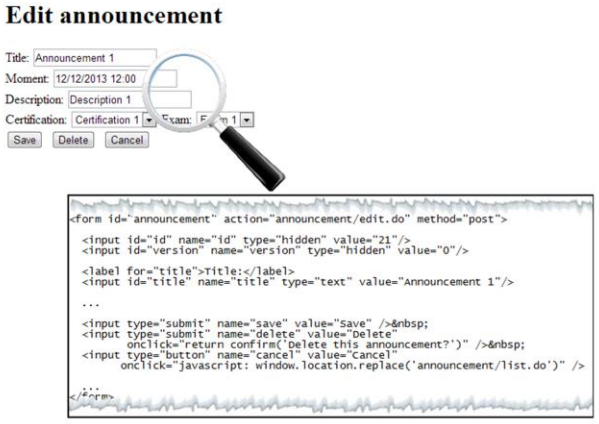
The GET method is the most common one. GET requests typically originate from hyper-links and address bars. They may also originate from a button, but this is not recommended. In this slide, we show an excerpt of a simple web page. It's a table in which the first two columns are hyper-links, whose HTML code is also shown.



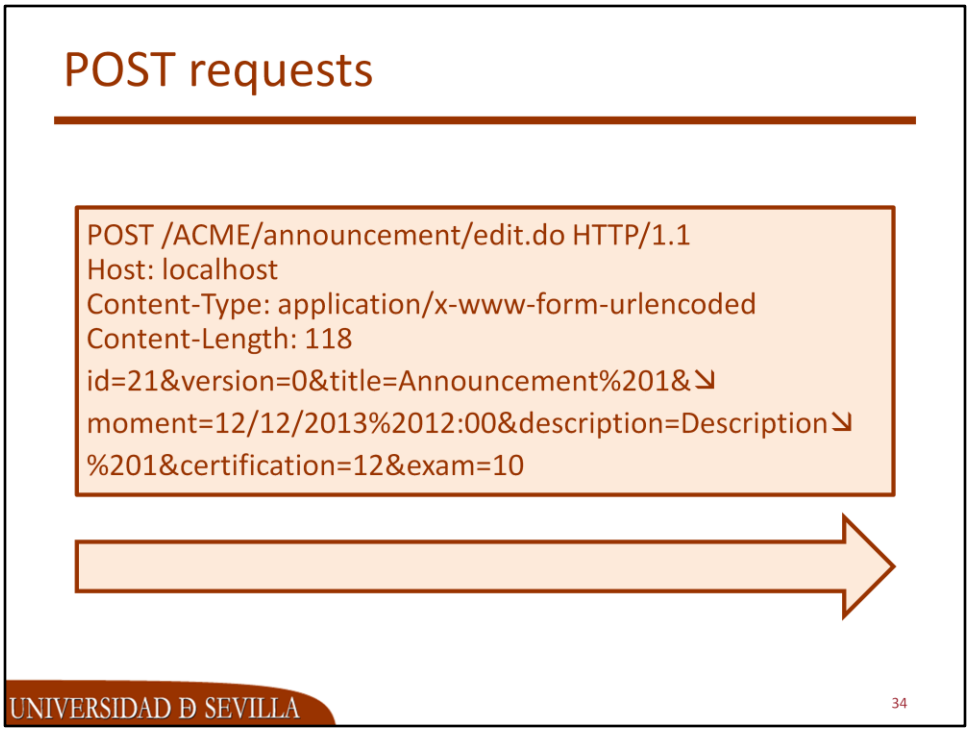
When the user clicks on a hyper-link, the browser creates a request to the server. This request's like the one we analysed before: "GET", the path to a resource, "HTTP/1.1", the host, and a collection of cookies. Just realise that the hyper-link in the slide before reads "<a href="registration/create.do?announcementId=12">", but the path in the request reads "/ACME/registration/create.do?announcementId=12". This is because relative paths must be made absolute before they are sent in a request. Note, too, we have added a parameter in this example, which comes after the "?" symbol in the path.

NOTE: We use the "⇲" symbol to indicate that a line continues, but there's not enough space on the slide. There aren't any actual blanks after the "?".

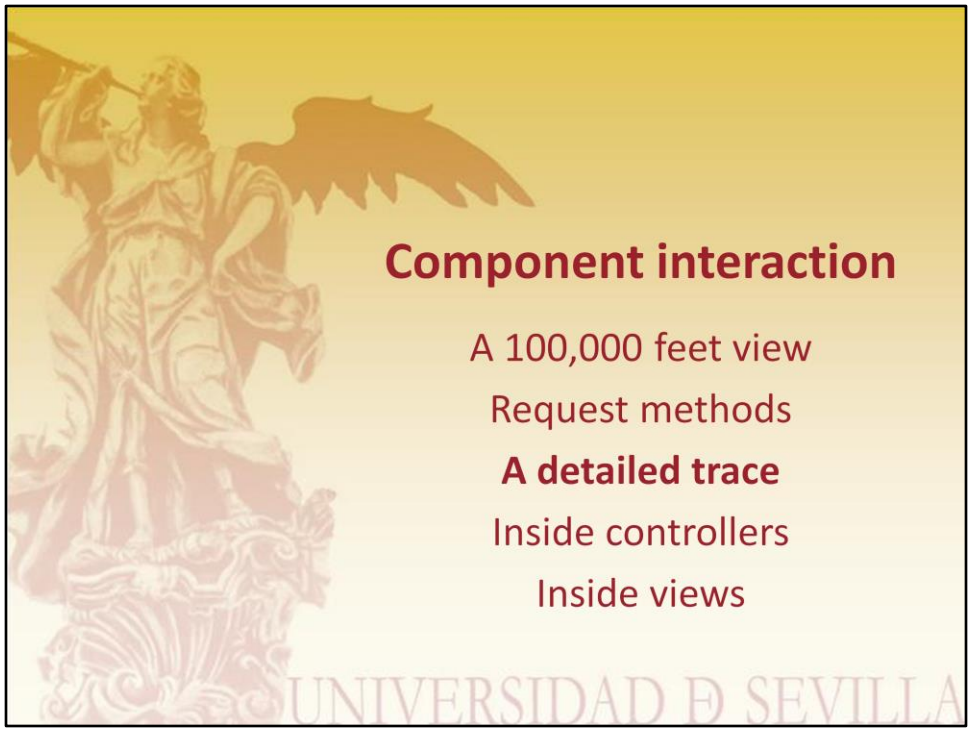
The POST method



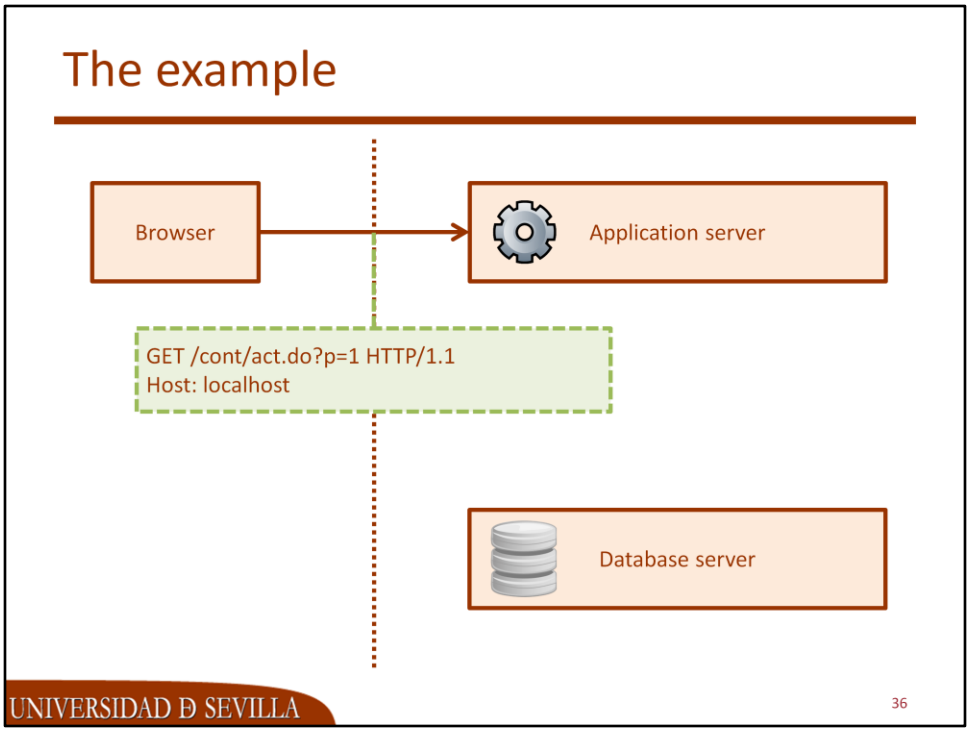
POST requests typically originate from buttons, which, in turn, submit forms. (Please, recall that a button may also result in a GET request, but this is not recommended.) In the slide, we show a simple form and an excerpt of its HTML code. When the user clicks on the “Save” or the “Delete” button, the form is submitted to the server, which results in a request like the one in the following slide.



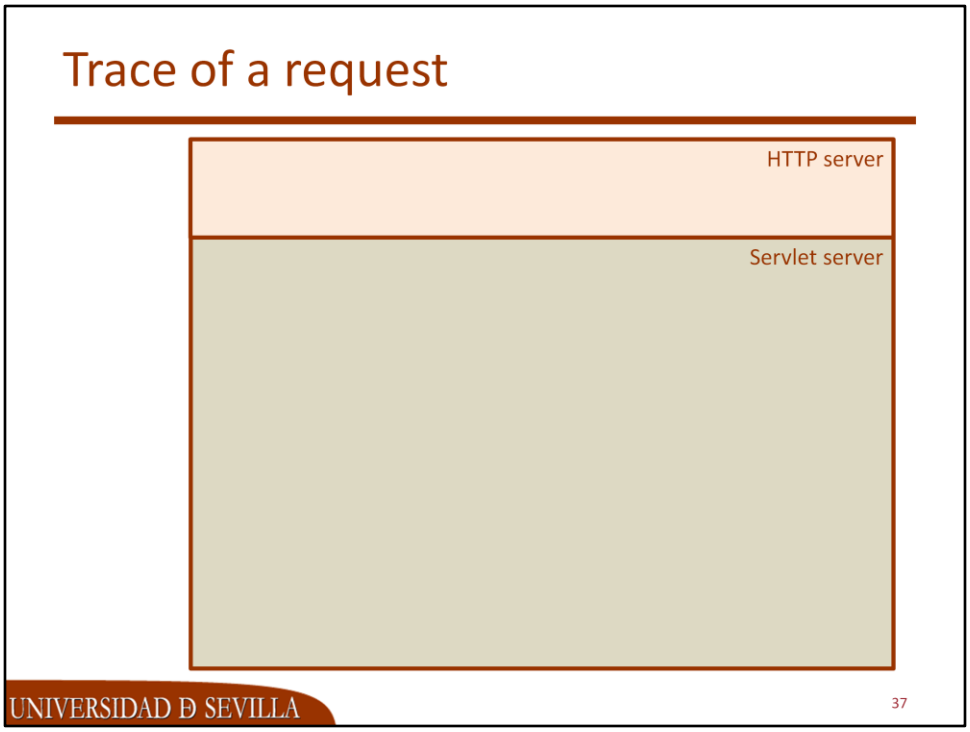
The main differences with regard to the GET request are that the method is “POST”, which requires two additional attributes: “Content-Type”, to indicate how the data that is being submitted to the server is encoded, and “Content-Length”, to indicate the length of the data in bytes. The data in the form come after these attributes. The reason why submitting a form using the GET method is not recommended is that the length of the data to be submitted may be arbitrarily long, and most operating systems put a limitation of a few hundreds characters on the length of a URL.



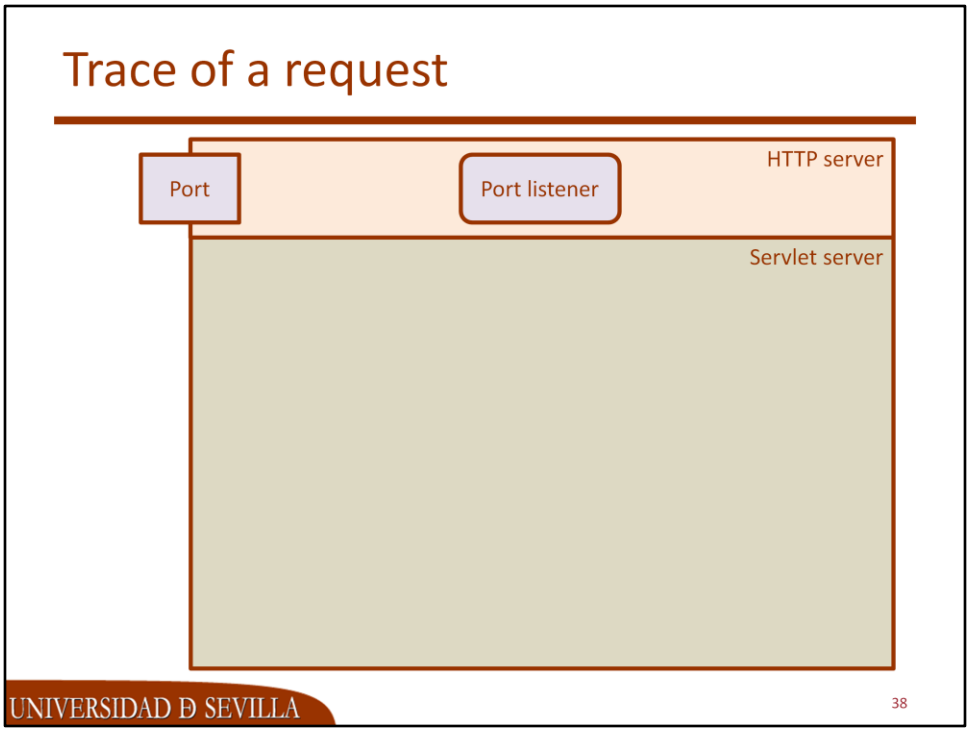
Ok. Now, we have a 100,000 feet view of how a request to a server works, and we know a little about the GET and the POST request methods. It's now time to delve into the details.



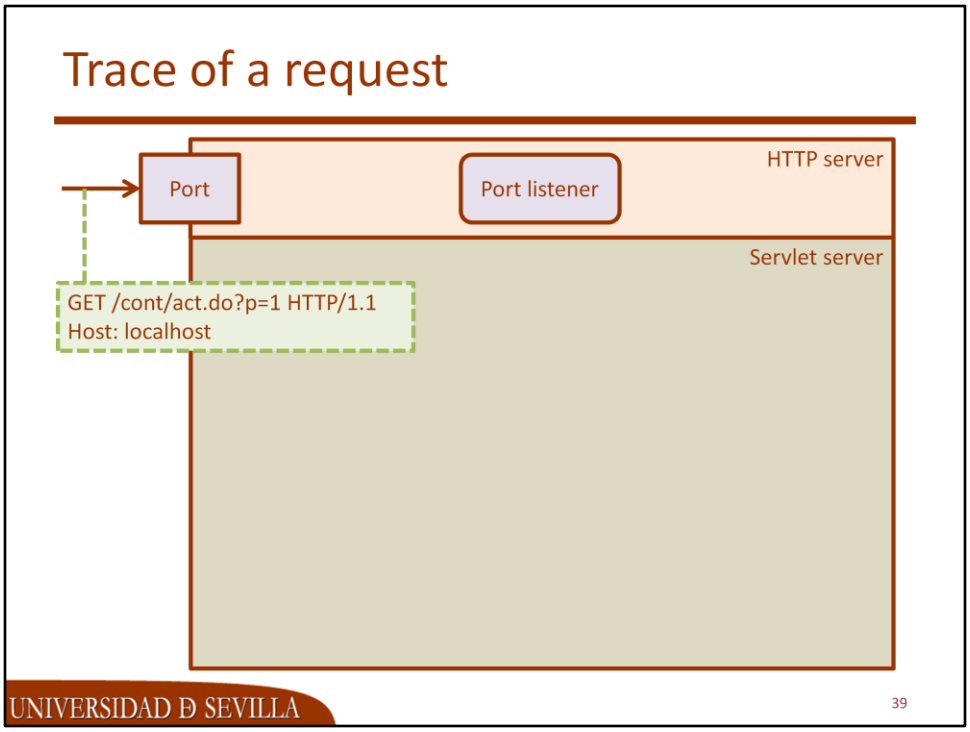
We'll work on the following sample request. Assume that the user clicks on a hyper-link whose URL is "/cont/act.do?p=1". The ".do" suffix indicates that this request is not to an image or other passive resources, but to an active resource. We'll now trace this request until the server produces a response.



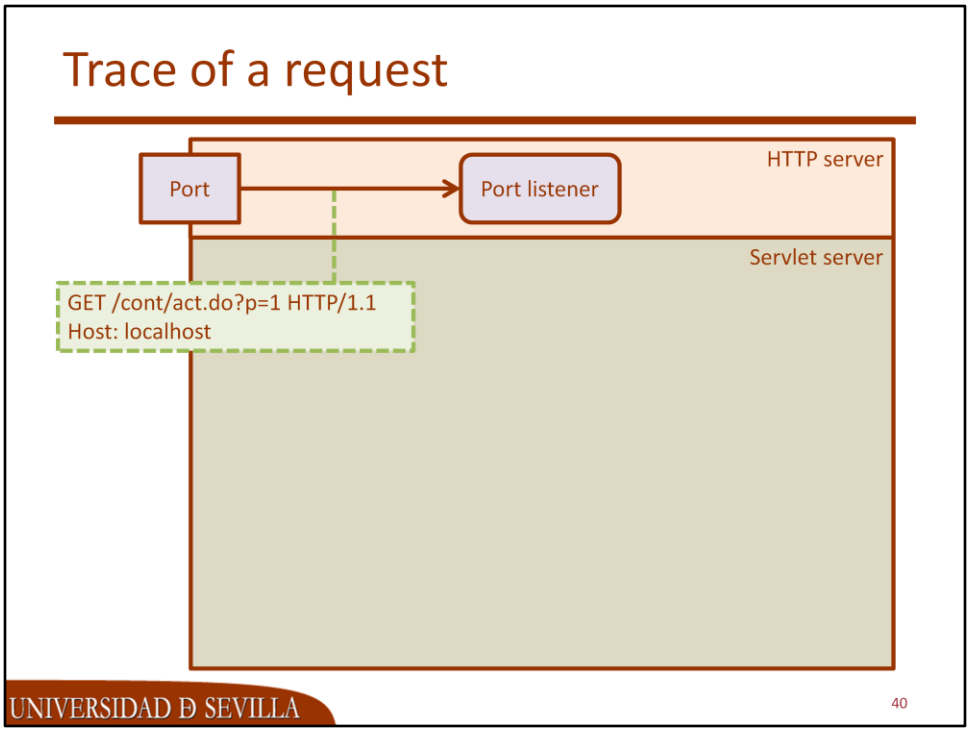
This slide shows the structure of a typical application server. They are usually composed of an HTTP server, which cares of handling the HTTP protocol, and a servlet server, which cares of implementing the business logic that must produce a response to every request. In this subject, the application server is Apache Tomcat; its HTTP server is called Coyote and its servlet server is called Catalina.



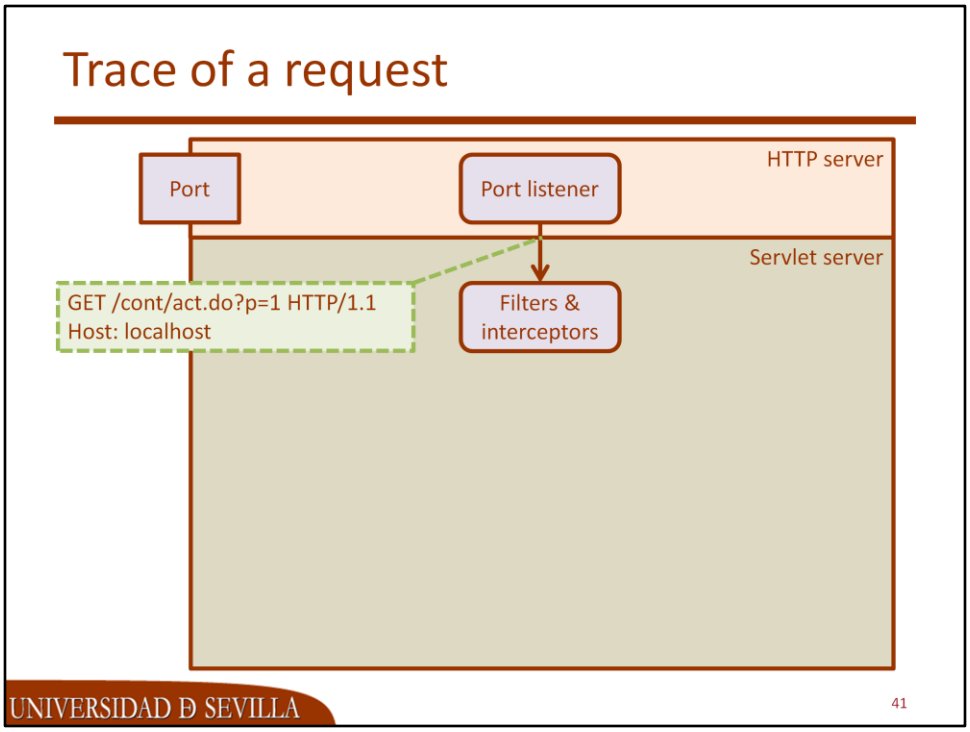
The HTTP server's basically composed of a port and a port listener.



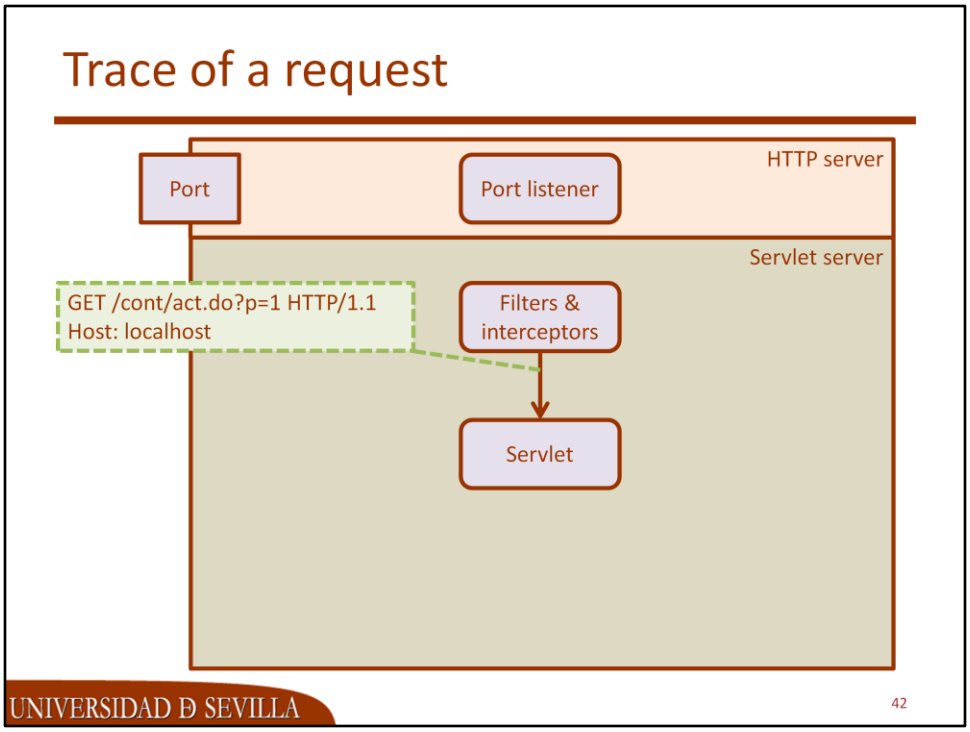
The port is an operating-system artefact that allows to read messages from the network interface of a computer and filters them using an IP address and a port number. HTTP requests are usually read through port 8080 in development configurations and port 80 in production configurations.



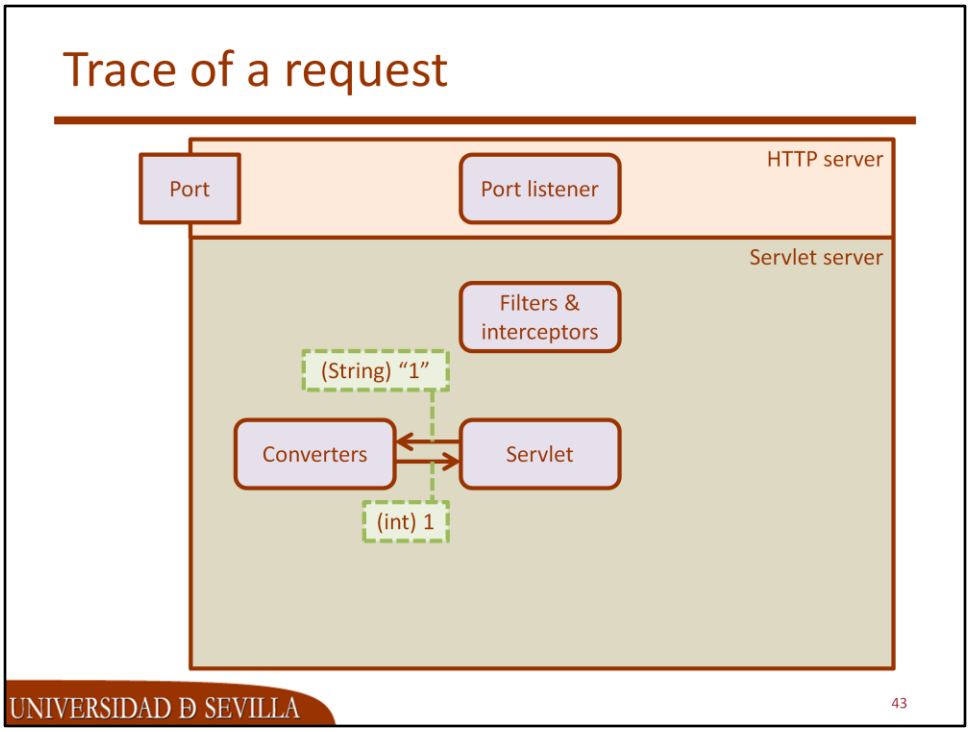
The task of the port is very simple: it simply passes the HTTP request that it reads from the network interface on to a port listener.



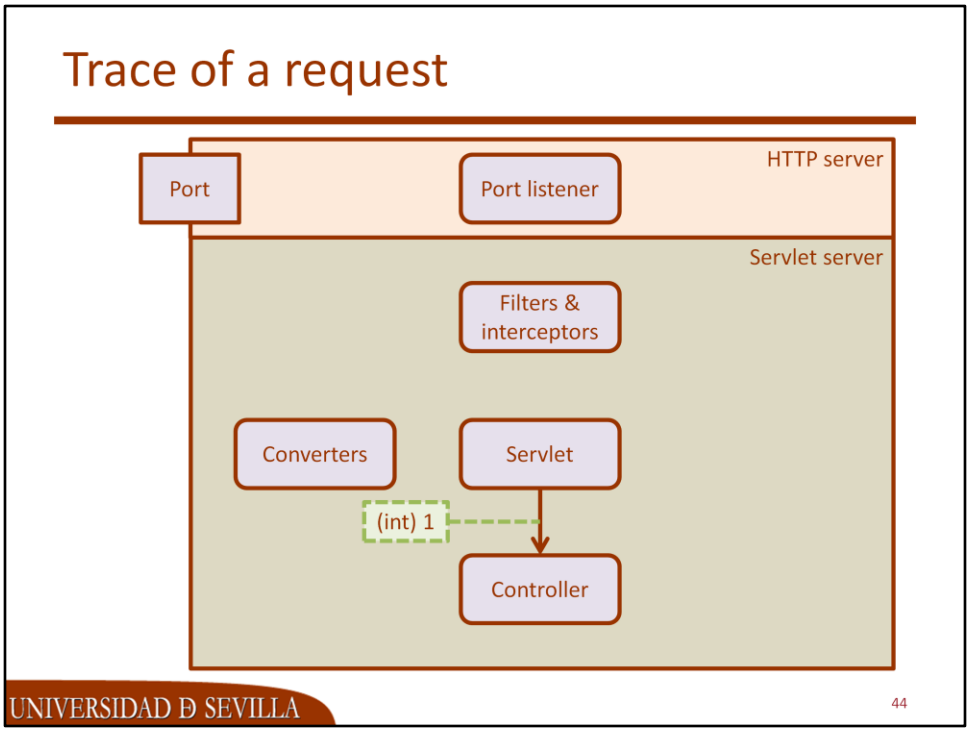
Neither is it too difficult the task of the port listener, since it typically passes the request onto a chain of filters and interceptors. They implement pre-processing or post-processing steps like checking if the request is authenticated and can thus be served or not, if the request includes a parameter to change the locale (the language in which the response will be produced), and so on. A filter or an interceptor might also be used to log the requests made to a server in a log file. Conceptually, filters and interceptors are the same thing; the only difference is that filters are a standard in the Java world, whereas interceptors are part of the framework we'll use to implement the architecture that we're describing.



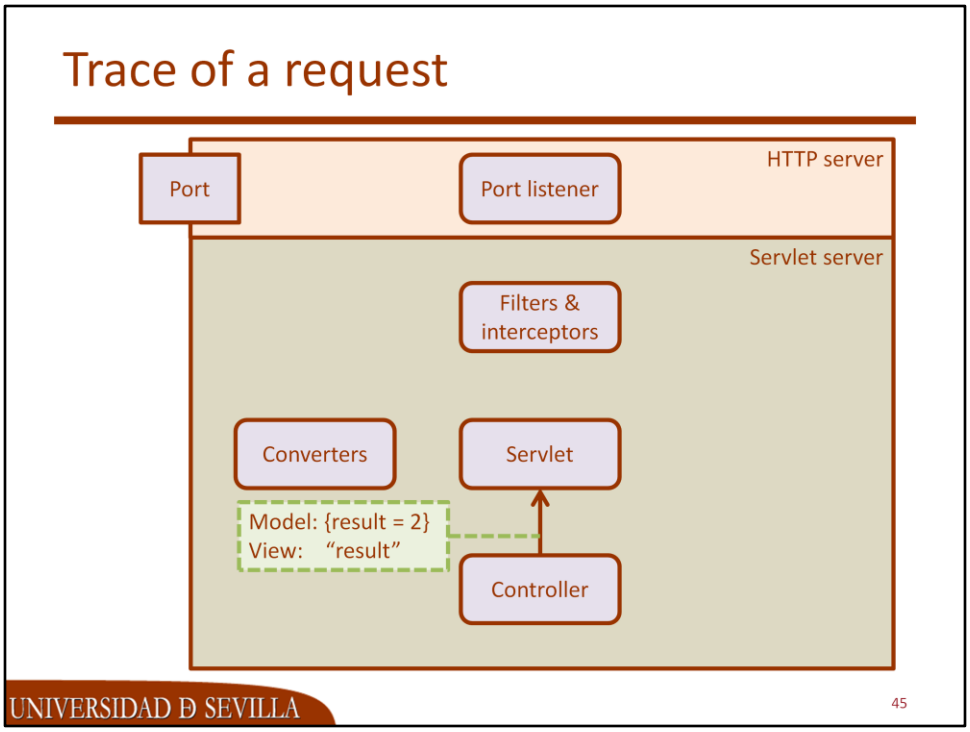
If no filter or interceptor drops the request, which might happen if it is not properly authenticated, the request is then routed to a so-called servlet. A servlet is an object that starts processing the request and orchestrates the following steps.



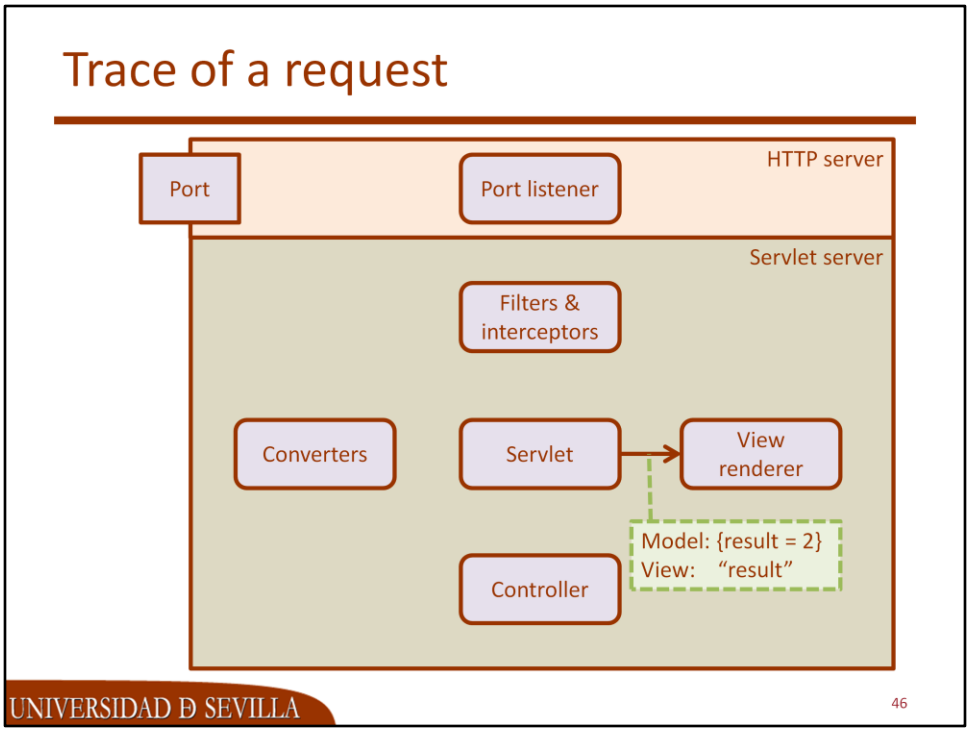
The first thing it does is to transform the parameters or the form attributes in the request into objects of the appropriate type. Please, note that the parameters or the form attributes in a request are just strings of characters. For instance, the parameter "p=1" or the attribute "moment=12/12/2013%2012:00", which we've already seen in the previous examples, might seem to be of type integer and date, but they are plain strings of characters. The servlet transforms these strings into objects of the appropriate types by means of converters. The framework that we use to implement our architecture provides us with many simple converters; we, as developers, will often have to design new converters that are specifically tailored to our project-specific types. For instance, we'll typically have to create a converter to transform an integer that represents an object identifier into a domain object, e.g., an exam or a customer, by looking up the integer in a database.



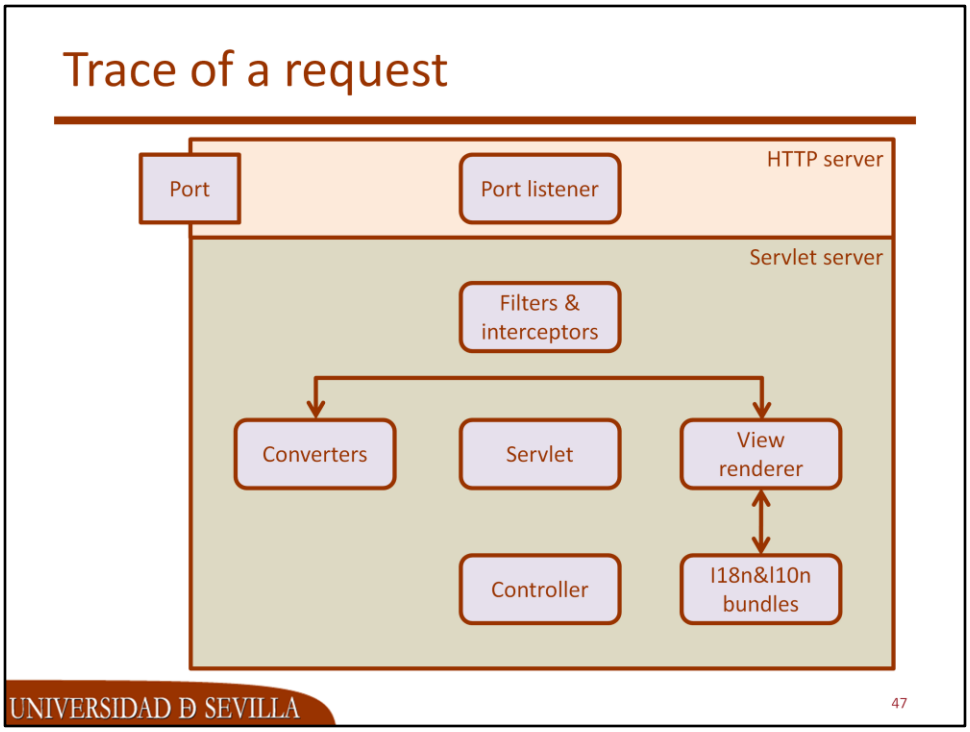
Once the servlet has converted the parameters or the form attributes into objects of the appropriate types, it delegates serving the request to a controller. A controller is an object that serves request that start with a given prefix (for instance "/cont" in our example); these objects provide a number of methods, each of which serves a given suffix (for instance "/act.do" in our example). Think of the controllers as if they were objects that provide a number of methods that help serve a number of requests that are logically related. For instance, it's very common that a controller serves request to list, create, edit, or delete entities of a given type, e.g., exams, customers, carts, and the like. We'll delve into the details behind the controllers later; so far this idea is enough to get the overall picture.



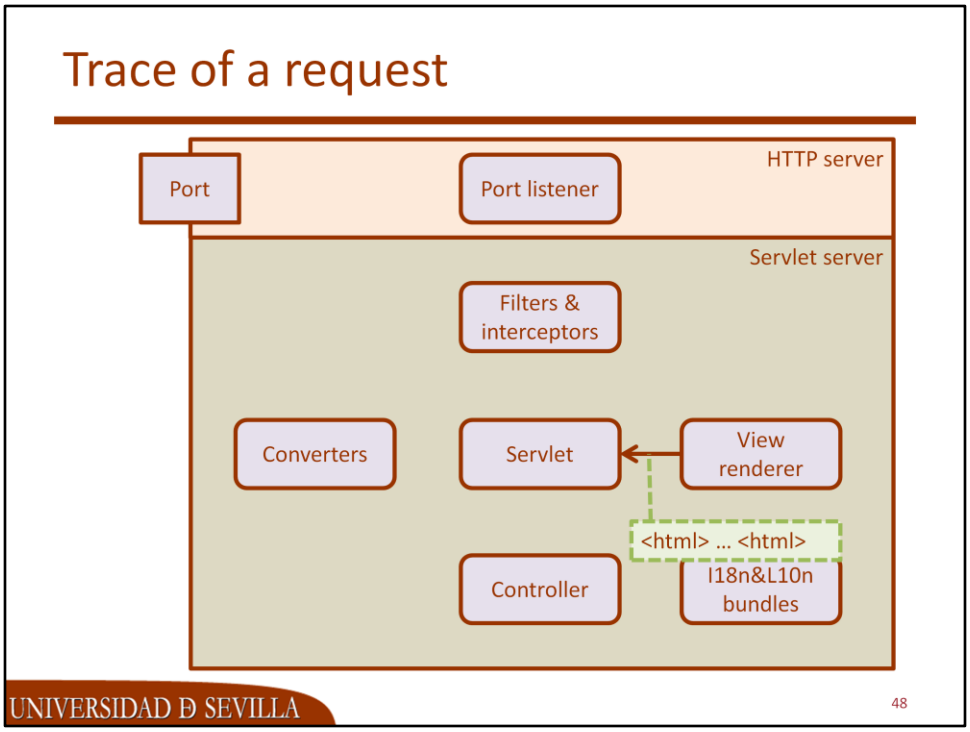
When the controller finishes working, it returns a model and the name of a view. The model is a map in which it associates variables with values; for instance "{result = 2}". The view is a piece of HTML in which the values of the variables in the model will be inserted. Think of a view as a template that has placeholders to insert the values of the variables in a model and processing instructions that endow the template with limited algorithmic capabilities.



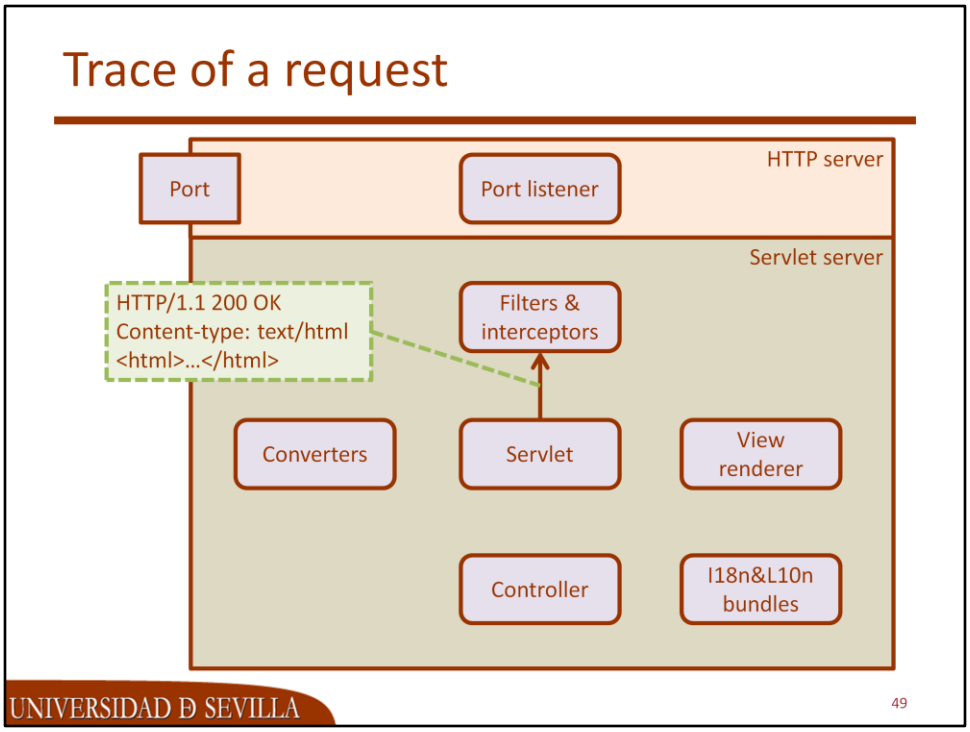
When the servlet gets the model and the name of the view from the controller, it passes them both on to a view renderer, which is an object that interprets the view, that is: it executes the processing instructions in the template and inserts the values of the variables in the model in the appropriate placeholders. The view renderer used by Tomcat is called Jasper.



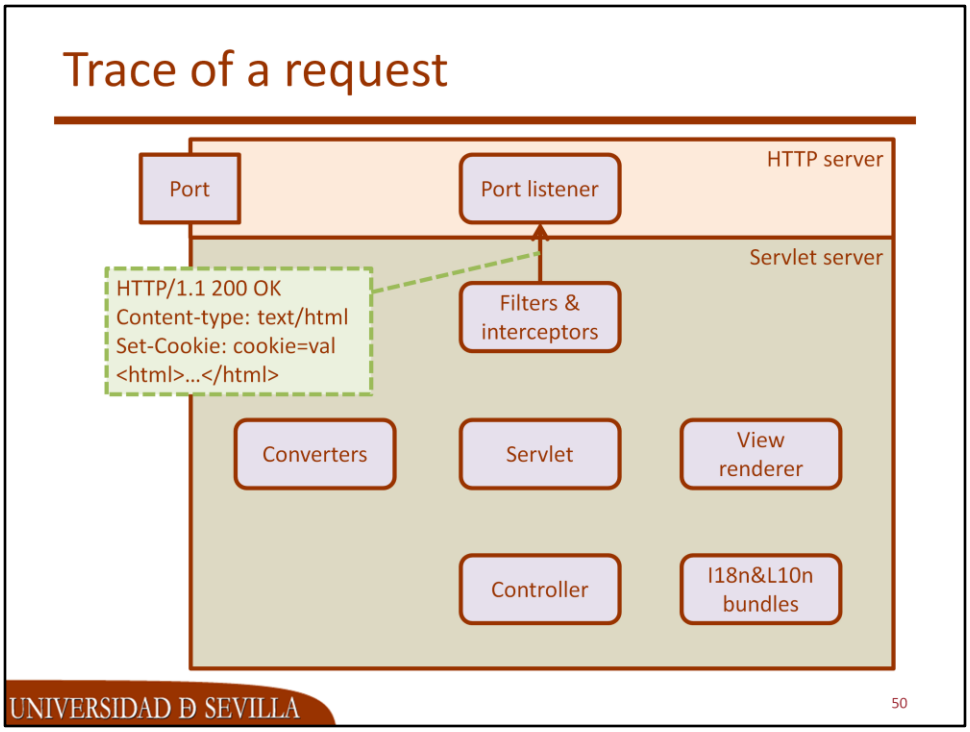
The view renderer relies on the converters and i18n & l10n bundles. Such a bundle is an object that cares of internationalisation and localisation issues. Basically, this object transforms some keys in the view into messages in a given language. For instance, instead of writing the string “Welcome to this wonderful application” literally in a view, we’ll insert a key like “welcome.message”. The i18n & l10n bundle is responsible for translating this key into “Welcome to this wonderful application” when our system is working in English or “Bienvenidos a esta maravillosa aplicación” when it runs in Spanish. The language in which an application runs is commonly known as the locale of the application.



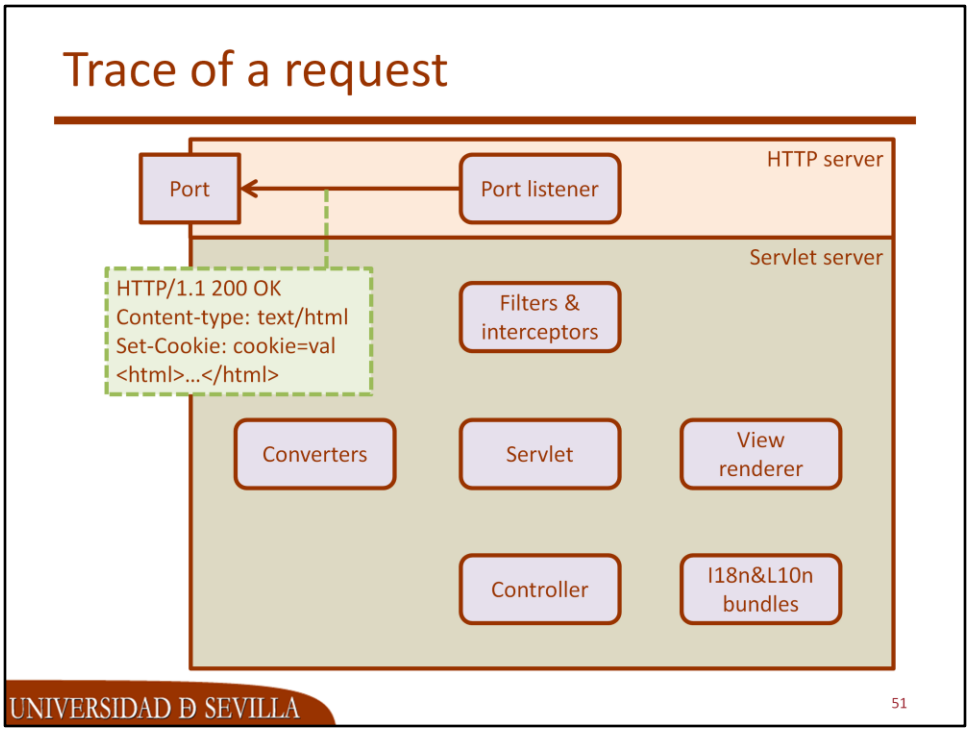
In the end, the view rendered produces a piece of HTML code that is returned to the servlet.



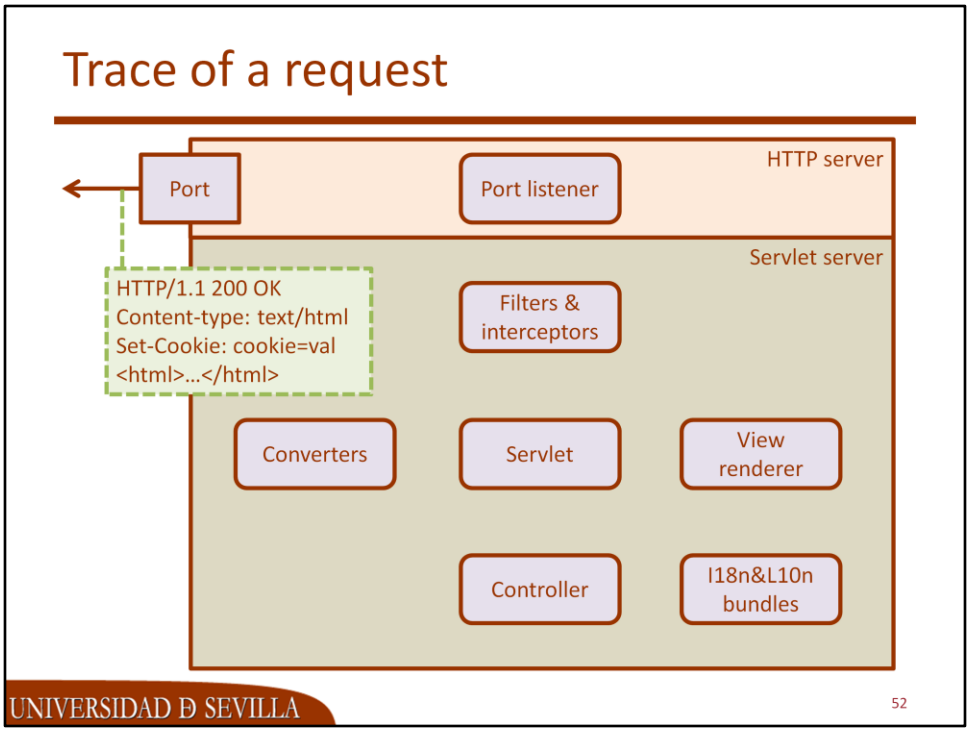
The servlet, in turn, returns it to the chain of filters and interceptors. (Note that the HTML has been packaged into an HTTP response message.)



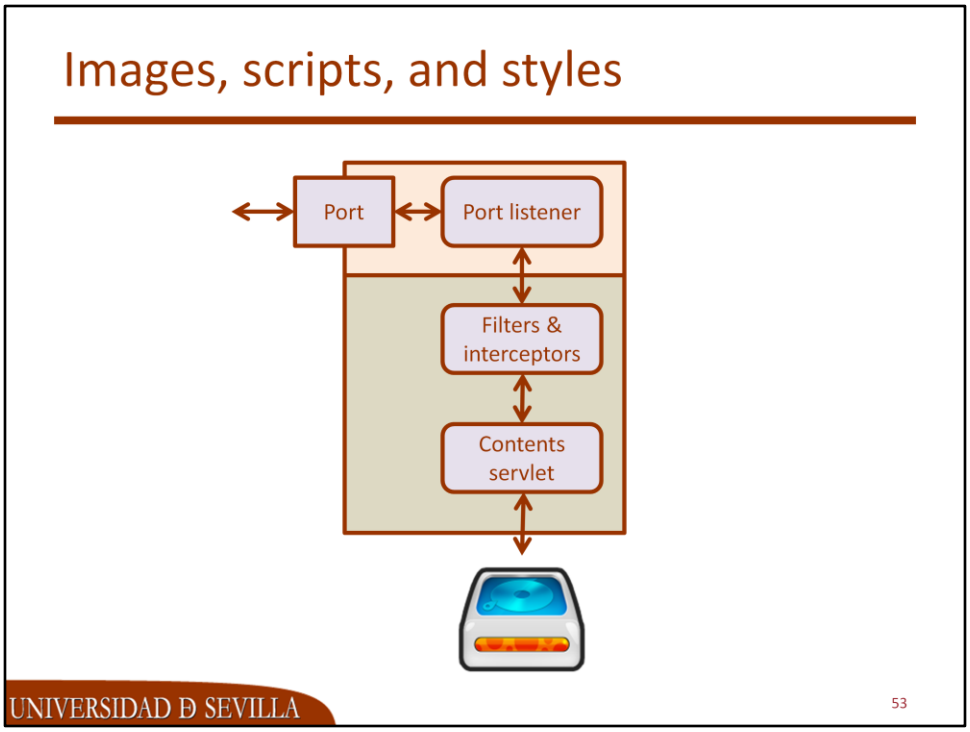
The filters and interceptors perform some post-processing, e.g., adding a cookie or logging, and return the result to the port listener.



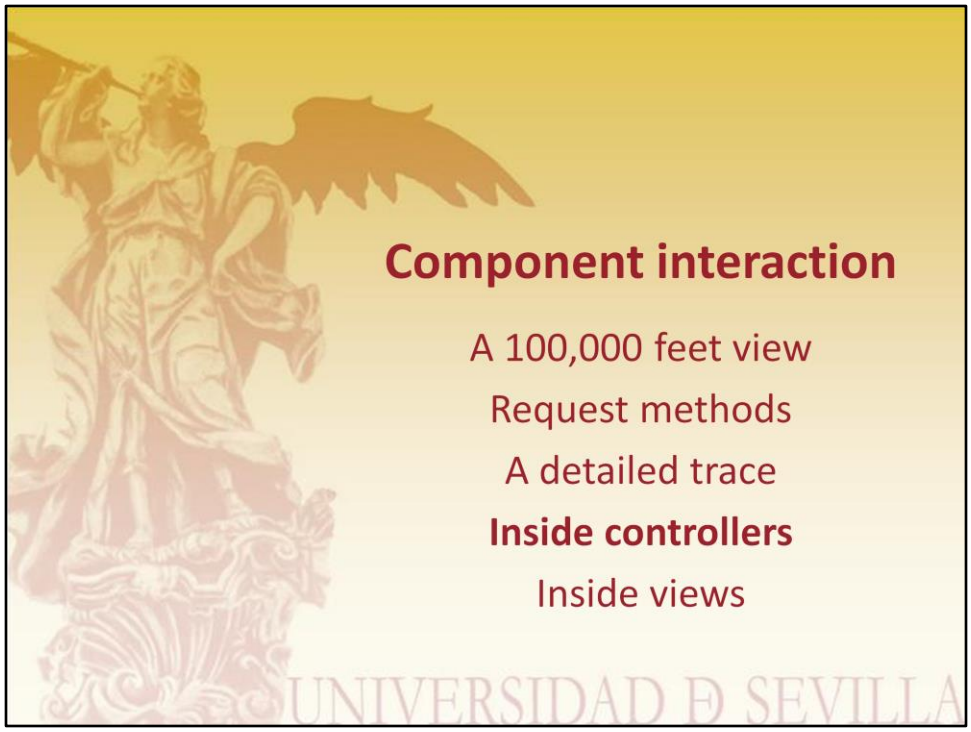
This in turn, returns the response to the port.



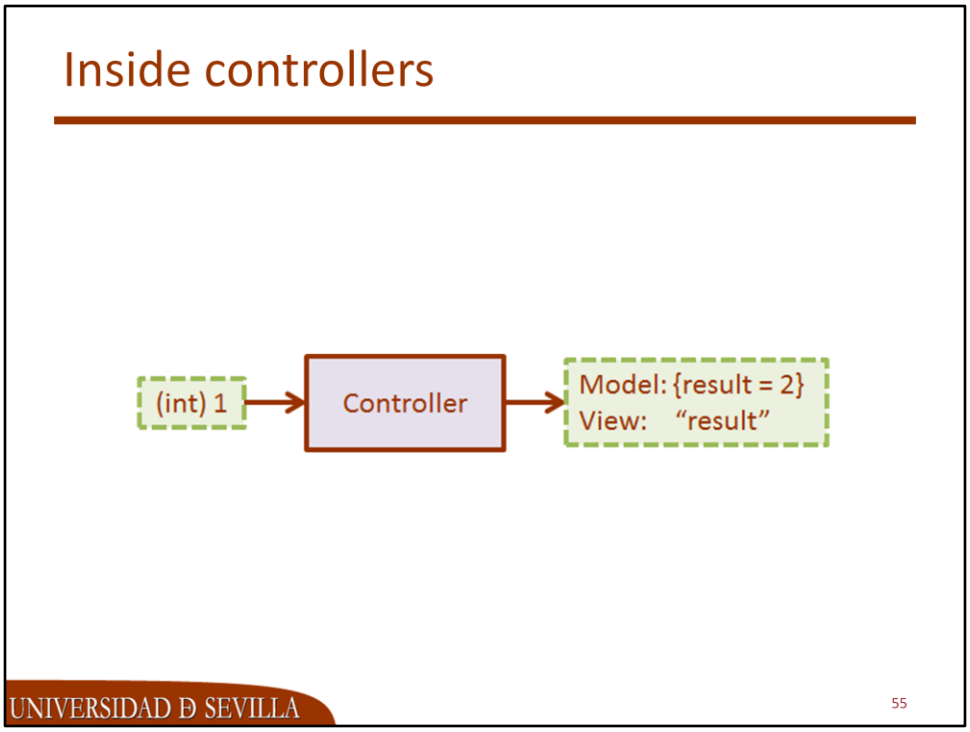
And, finally, the port interacts with the network interface and sends the response to the network.



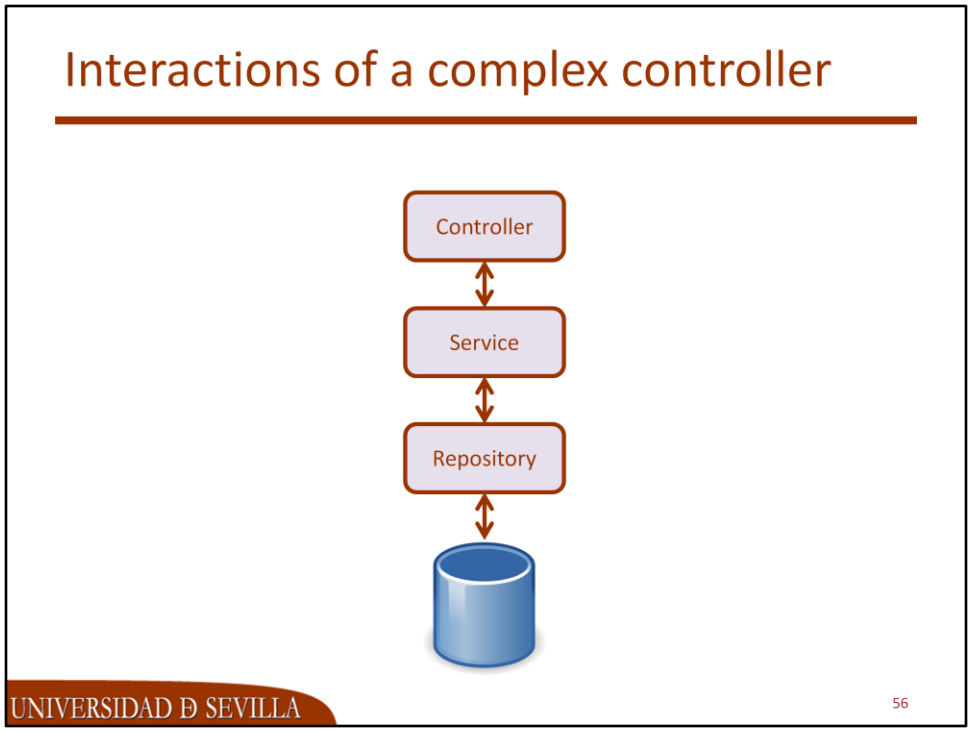
The process that we've presented in the previous slides traces a ".do" request, that is, a request that results in a business action. Recall that these requests typically result into HTML documents; in order for the browser to render these documents, it'll have to make additional requests to retrieve ECMA script files, CSS style files, and images. The process of serving such requests is slightly different. In such cases, the request goes through a port, a port listener, and a chain of filters and interceptors, but then a special servlet takes control. This servlet is commonly referred to as the contents servlet, since it simply fetches the requested resources from disk and returns their contents.



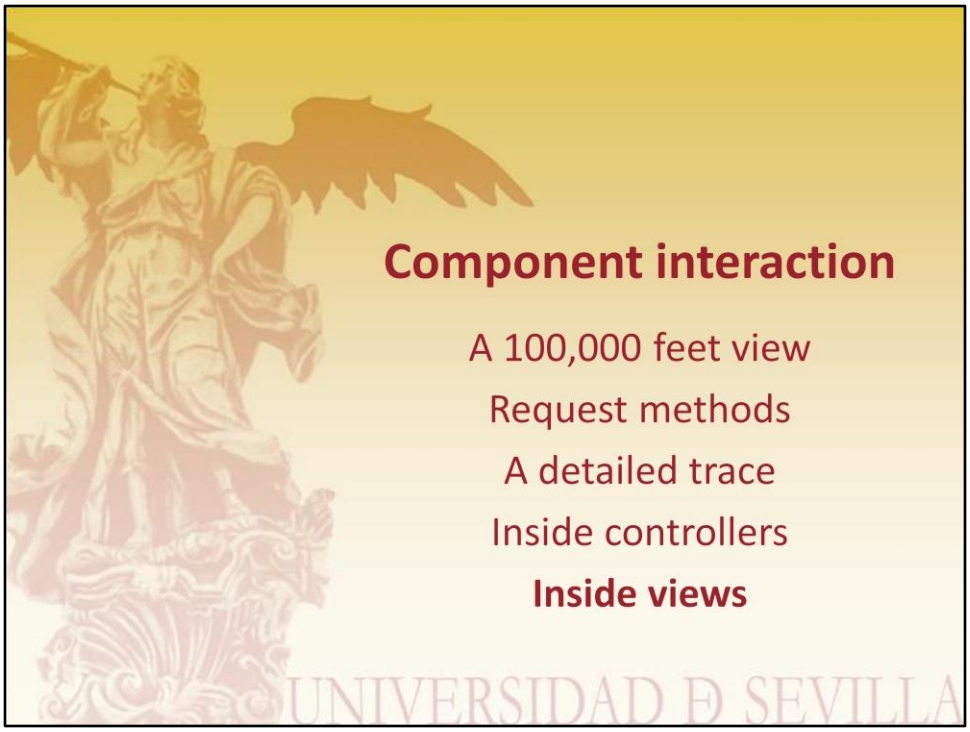
In the previous section, we've learnt a lot about how a request is served. By now, you should be familiar with how things go on behind the scenes. Note, however, that we've provided very little insight into the controllers. It's now time to delve into their details. Not many, but enough to have a better understanding.



So far, we know that a controller is an object that takes objects that come from the request, processes them, and outputs a model and a view name.



Typical controllers in a web information system rely on services, which, in turn, rely on repositories. Services implement business rules and logic and repositories retrieve, create, update, or delete data from a database. This design helps assign responsibilities very well: the repositories care, exclusively, of interacting with the database: they transform objects into relational tuples and vice-versa; services implement business rules and logic in an abstract manner; and controllers care of passing data onto the services so that they can process them, creating models with the results, and selecting the appropriate views to render them.



As we promised, we have not delved into a lot of details. Just enough to have a better understanding, but we still have to work a lot more on them. It's now time to peek at the views.

Inside views

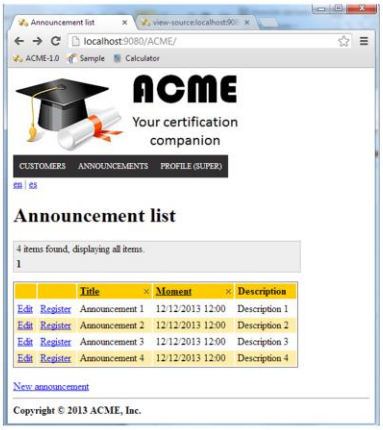
```
<%--
 * binary-result.jsp
 *
 * Copyright (C) 2012 Universidad de Sevilla
 *
 * The use of this project is hereby constrained to the conc
 * TDG Licence, a copy of which you may download from
 * http://www.tdg-seville.info/License.html
 --%>

<%@page language="java" contentType="text/html; charset=ISO-
    pageEncoding="ISO-8859-1"%>

<%@taglib prefix="spring" uri="http://www.springframework.or
<%@taglib prefix="form" uri="http://www.springframework.org/
<%@taglib prefix="jstl" uri="http://java.sun.com/jsp/jstl/co

<p>${x} ${operator} ${y} = ${result}</p>

<jstl:if test="${exception == true}">
    <p style="color: red">
        <spring:message code="calculator.exception" />
    </p>
</jstl:if>
```

UNIVERSIDAD D SEVILLA

58

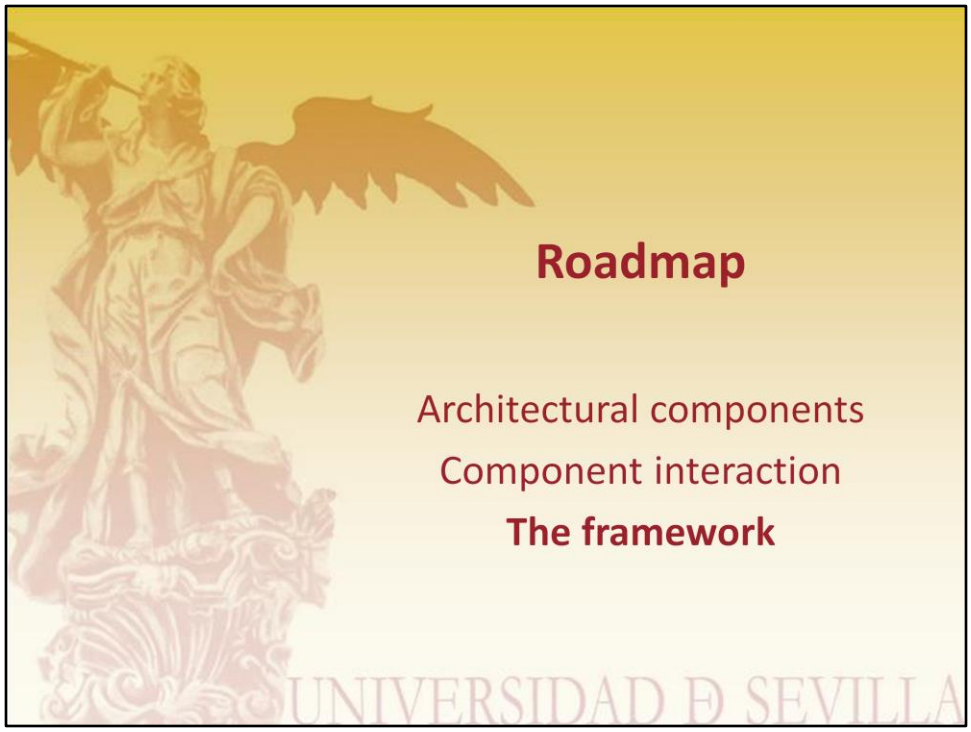
As we told you before, a view is basically an HTML document with placeholders and processing instructions. In the example in this slide, the placeholders are “${x}”, “${operator}”, and ${result}, and the processing instruction is “<jstl:if test="${exception == true}"> …</jstl:if>”. Recall that the controller must return a model (a map from variables onto objects) and a view name to the servlet, and that it, in turn, passes these data onto a view renderer that inserts the values of the variables in the model into the appropriate placeholders, and then executes the processing instructions. As you might expect, there are assignment, loop, multi-choice, and a few other processing instructions, but, please, bear in mind that the language is very limited. It’s not easy at all to build a complex interface using these simple processing instructions.

User controls



Fortunately, there are hundreds of components that provide advanced user controls like menus, data grids, autocomplete input boxes, calendars, and so on. That means that we'll usually only have to search for the appropriate component, not to create a complex interface building exclusively on HTML and the limited set of processing instructions that we've presented before. We'll learn a little more about them in a forthcoming lesson. So far, this is enough.

NOTE: in this slide, we don't use term "component" to refer to architectural components, but to software components; that is, we use it to refer to libraries that provide advanced tags and processing instructions.



So far, you should have a good understanding of the architecture that we're going to take as a starting point for our web information systems. Please, recall that we had a piece of good news when we started this lecture: commonly, architectures are supported by software frameworks that save us from a lot of work. It's now time to delve into the one that we're going to use in this subject.

What's a framework?

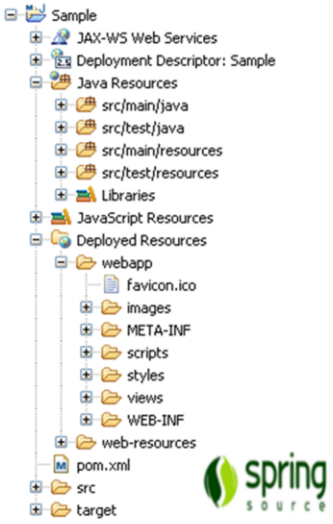
It's a semi-instantiated architecture that provides a number of components and placeholders to plug specific logic in

UNIVERSIDAD Ð SEVILLA

61

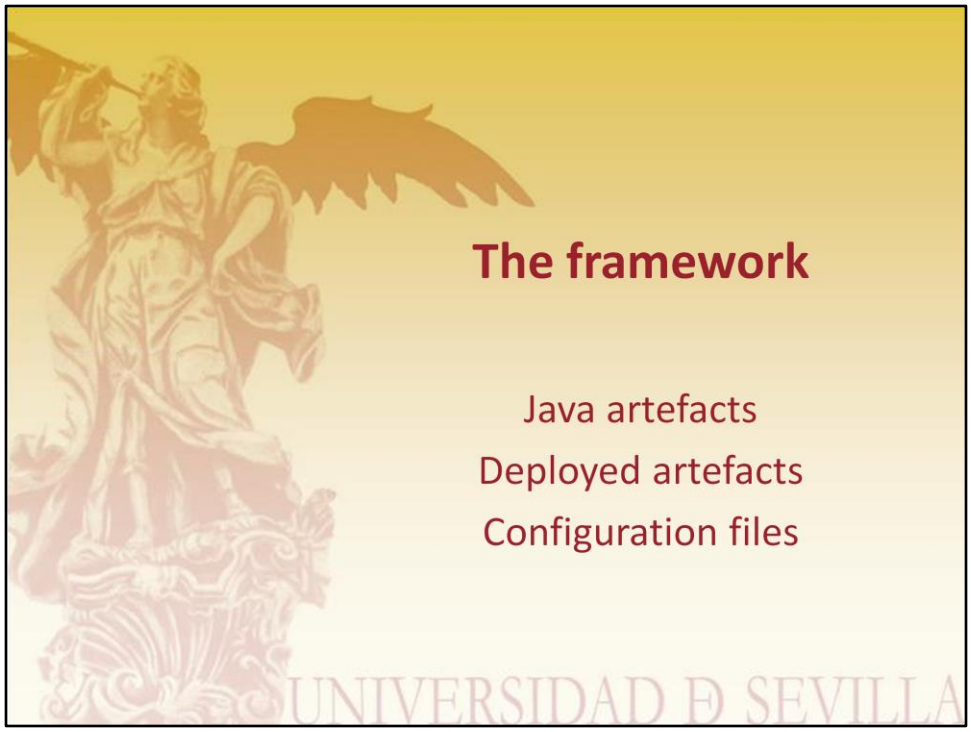
Recall from the introduction to this lecture that a framework is kind of a semi-instantiated architecture. It provides a number of classes that implement the components that we've studied previously, plus a number of placeholders or interfaces that we need to implement in order to provide project-specific classes. For instance, most frameworks provide a servlet or a view renderer implementation, since these components are absolutely project-independent; contrarily, it's your responsibility as a developer to provide the framework with classes that implement the domain objects (for instance, exams, customers, companies, accounts, and so on) plus a persistence model that maps them onto relational databases. That's like providing a piece of furniture or a chimney to the pre-built house in this slide.

Our framework

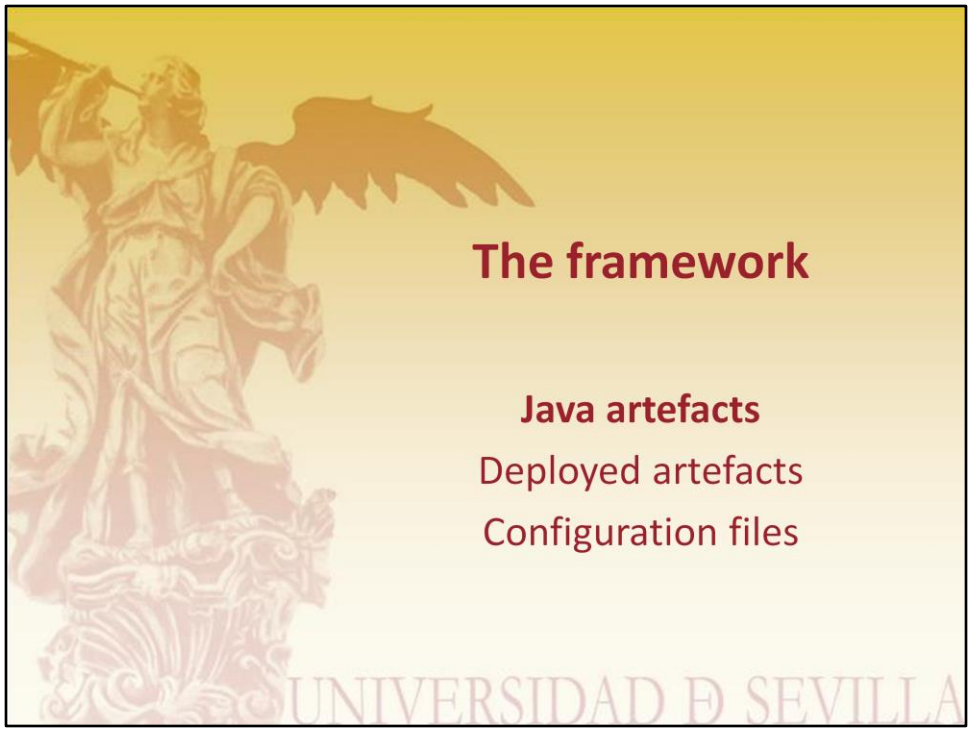


- Java artefacts
  - Java classes
  - Java resources
- Deployed artefacts
  - Client resources
  - Server resources
- Configuration files

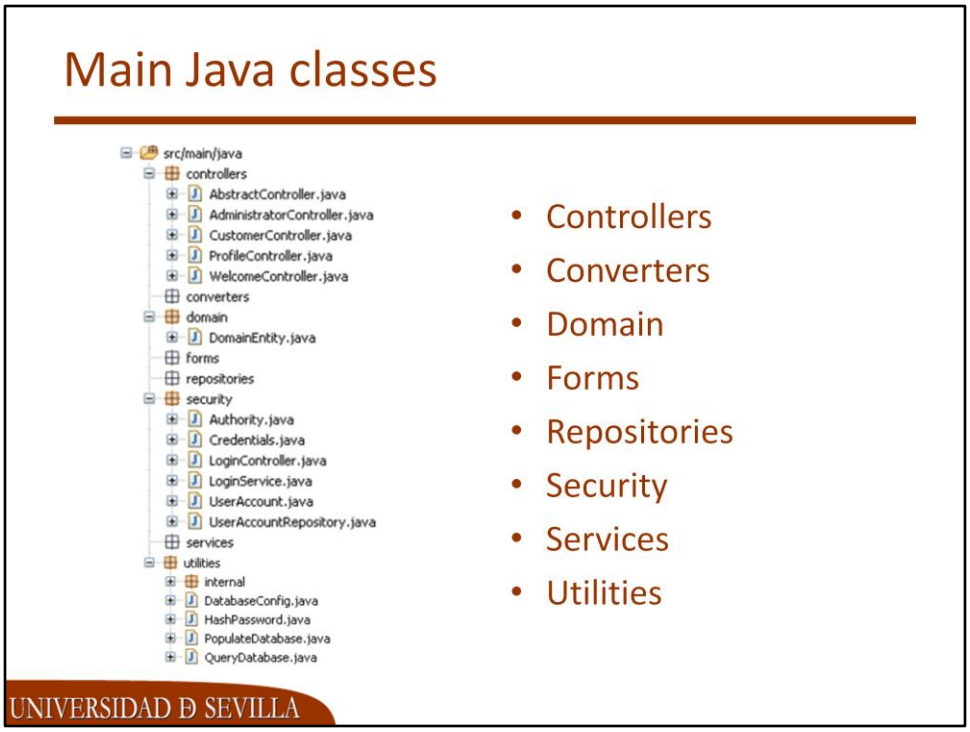
The framework that we're going to use in this subject is based on the Spring Framework. We delivered it as a project template in the previous lecture. So far, we'd just like to comment on its structure, that is, on how we've mapped the architecture that we've described in the previous sections onto an actual project that you can use as a starting point to develop new projects. The framework consists of Java artefacts (Java classes and Java resources), deployed artefacts (client resources and server resources), and a number of configuration files.



In the following subsections, we'll provide additional details on these artefacts.

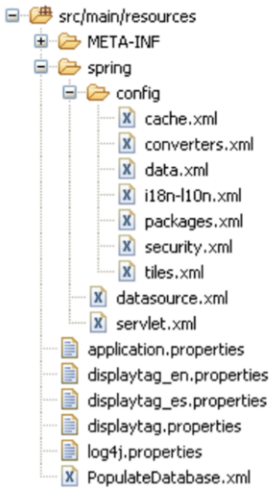


Let's start with the Java artefacts.



The main Java classes are organised into a number of packages, most of which, should now be familiar to you: “controllers”, “converters”, “domain”, “forms”, “repositories”, “security”, “services”, and “utilities”. So far, we haven’t talked about forms, but they are not difficult to understand: these are classes that are intended to model the information in a web form in cases in which there’s not an appropriate domain object (don’t worry if you don’t get the idea; you’re very likely not to have to use any forms during the first semester). Neither have we talked about security or utilities, but they are easy to explain: the “security” package has all of the classes that are related to keeping the access to your web information systems under control; the “utilities” package provides three utilities to hash passwords, populate databases, and to query them. Note that most packages provide a number of classes that instantiate a part of our architecture; you must complement them with classes that are specific to your projects.

Main Java resources



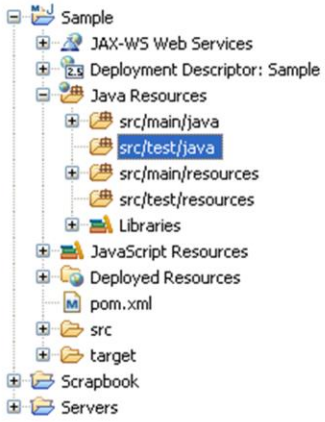
- META-INF
- Spring configuration
- Property files
- Population file

Java resources are not Java classes, but configuration files that are required so that your Java classes can work, namely:

- "META-INF": this folder contains a file called "persistence.xml" that provides information on how to connect to a database (driver, URL, user, password, and a few more properties).
- "spring": this folder contains many configuration files that are related to the Spring Framework.
- "*.properties": these files provide some properties that are required to configure a couple of components, namely: DisplayTag and Log4J.
- "PopulateDatabase.xml": these files provide a description of a number of sample objects that will be used to populate the database with initial data.

Note that the majority of resources are actually configuration files that we'll explore later.

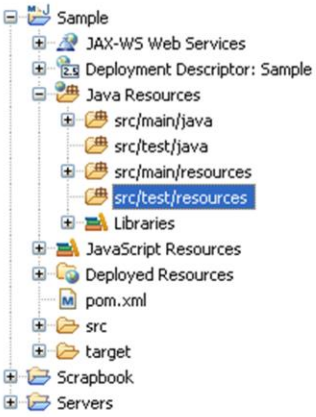
Test Java classes



UNIVERSIDAD Ð SEVILLA

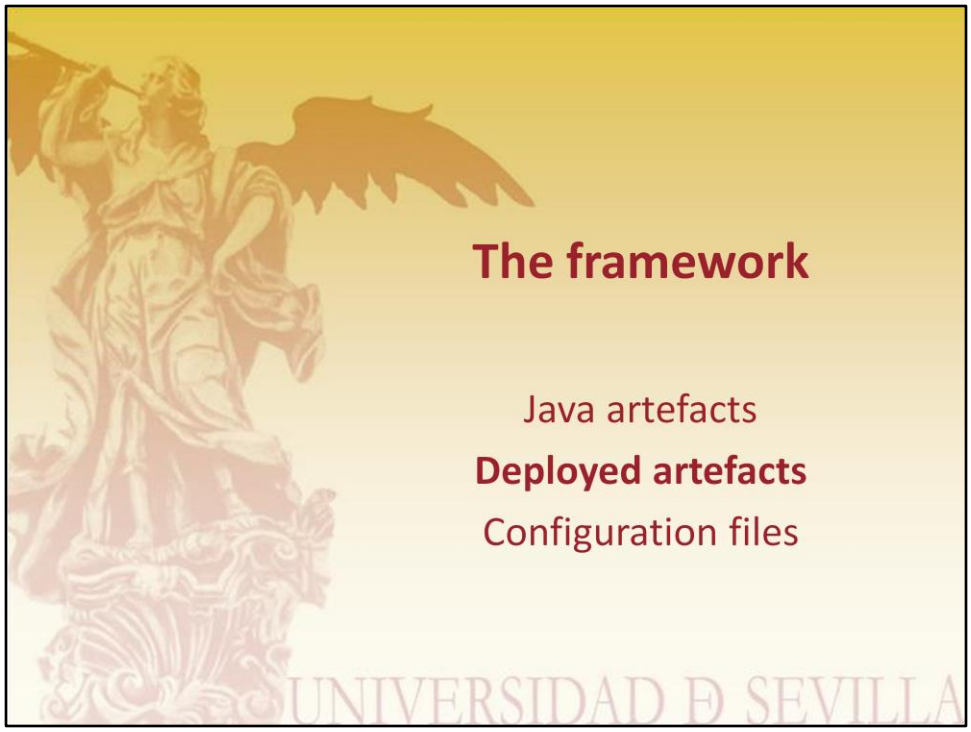
In addition to the main Java classes, we'll also have to provide a number of test classes, that is classes that implement scripts that will help us find bugs in our web information systems. These classes have little to do with the architecture of the system, and we won't deal with this topic until the spring semester. Thus, for the time being, we won't provide any test classes.

Test Java resources



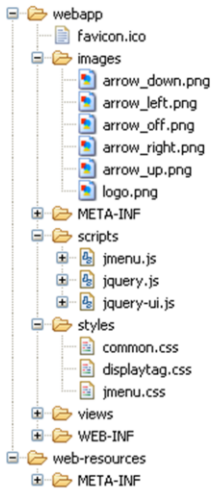
UNIVERSIDAD Ð SEVILLA

We'll have test classes in the Spring semester; we'll also need test resources. But, for the time being, we won't have any.



Let's now report on the deployed artefacts, which can be client artefacts and server artefacts.

Client artefacts

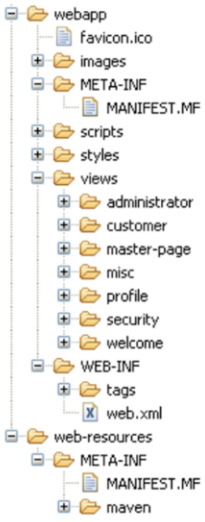


- Favicon
- Images
- Scripts
- Styles

UNIVERSIDAD Ð SEVILLA

The client artefacts are resources that are sent to the user's browser verbatim, that is: the favicon, images, scripts, and styles. Don't you know what a favicon is? It's a small icon that your browser shows on your browsing tabs; think of it as a small logo that helps users identify the contents of a tab more easily.

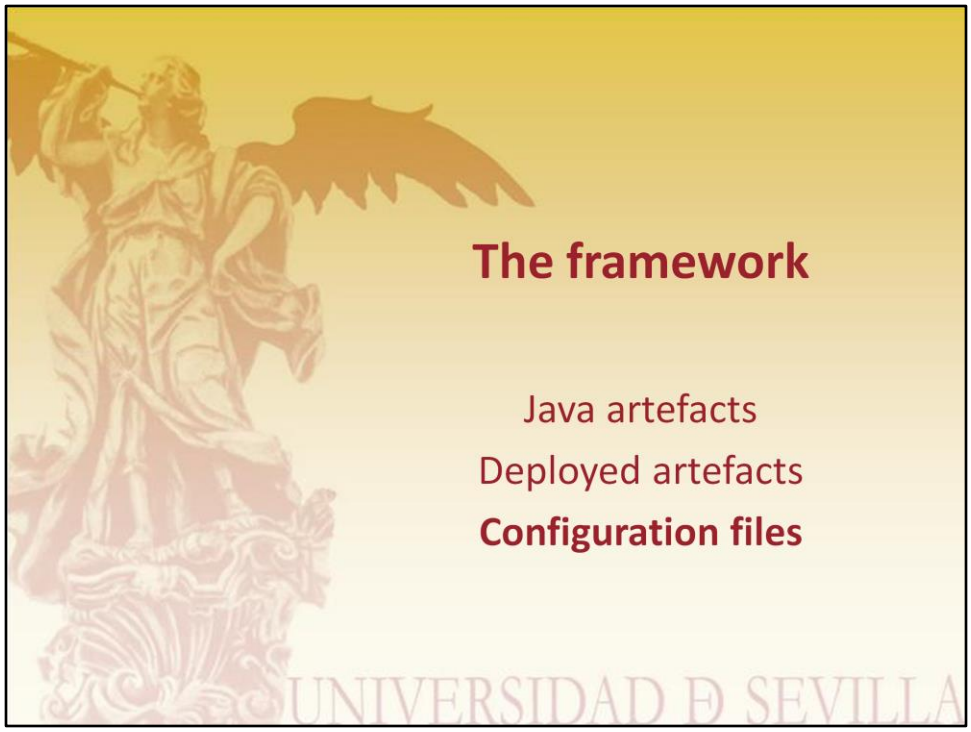
Server artefacts



- META-INF
- WEB-INF
- Views

UNIVERSIDAD D SEVILLA

The server artefacts are resources that the server processes, but are never sent to the user's browser. They include a few files in folders "META-INF" and "WEB-INF" and a folder with the views of our project. In the previous section, we presented the views as HTML templates; soon, we'll learn that they require a few more artefacts, including the i18n & l10n bundles.



Finally, let's take a quick look at the configuration files in the framework.

Be very careful!

UNIVERSIDAD Ð SEVILLA

73

Before going ahead, it's very important that you know this: changing configuration files requires a lot of expertise; creating them requires a lot more expertise. Our goal's that you can understand what they are intended for, not that you command them.

pom.xml

```
<!--
  * pom.xml
  *
  * Copyright (C) 2012 Universidad de Sevilla
  *
  * The use of this project is hereby constrained to the cor
  * TDG Licence, a copy of which you may download from
  * http://www.tdg-seville.info/License.html
 -->

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 ht

    <modelVersion>4.0.0</modelVersion>

    <!-- Artifact identification -->

    <groupId>Sample</groupId>
    <artifactId>Sample</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>war</packaging>

    <!-- Indexing information -->

    <name>Sample</name>
    <url>http://www.example.com</url>
    <description>This is a sample template project</descript

    <!-- Build commands -->

    <build>
```


"pom.xml" is Maven's configuration file. It provides artefact information, indexing information, build commands, repository information, and dependencies.

web.xml

```
<!--
 * web.xml
 *
 * Copyright (C) 2012 Universidad de Sevilla
 *
 * The use of this project is hereby constrained to the con
 * TDG Licence, a copy of which you may download from
 * http://www.tdg-seville.info/License.html
 -->

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instan
    xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="ht
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee h
    id="WebApp_ID" version="2.5">

    <display-name>Sample</display-name>

    <!-- Context configuration -->

    <listener>
        <listener-class>org.springframework.web.context.Con
    </listener>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/servlet.xml
        </param-value>
    </context-param>

    <!-- Dispatcher servlets -->
```

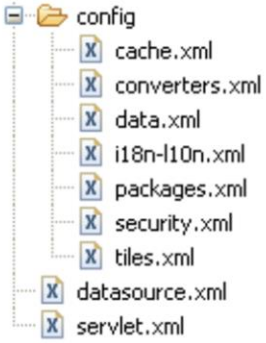

This file describes your servlet to the application server. It provides information on the context configuration, the class that implements your servlet, filters, welcoming, and error pages. Please, note that this file does not describe the internal configuration of your servlet; it only informs your application server that there's a new servlet in the neighbourhood; the internals are described in a separate file called "servlet.xml".

servlet.xml

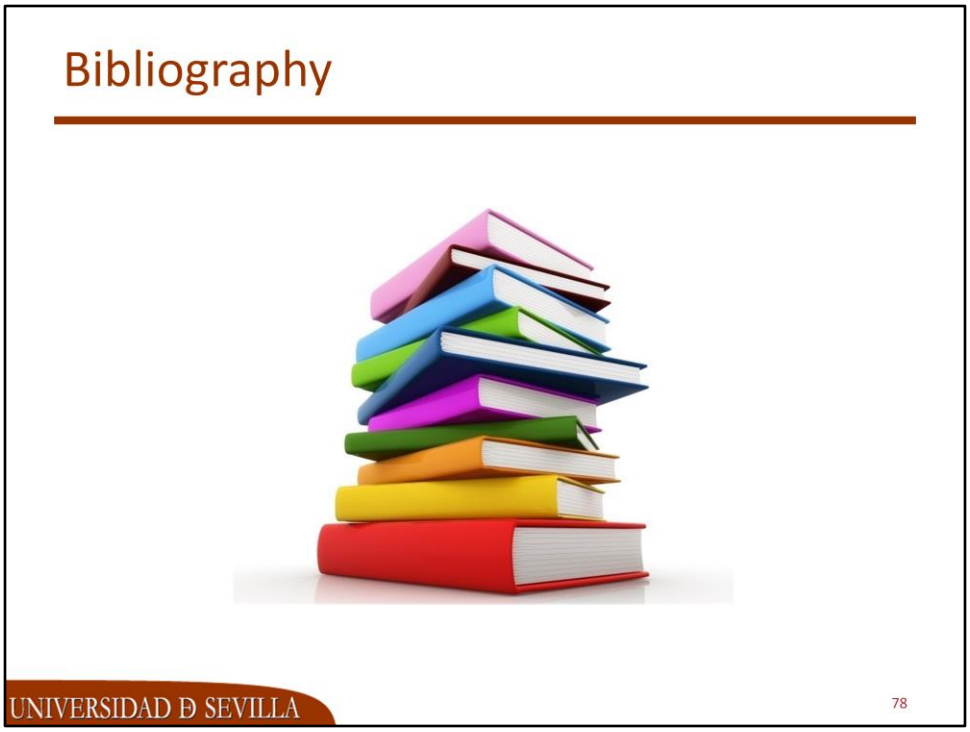
```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xm
    xmlns:mvc="http://www.springframework.org/schema/mvc" xm
    xmlns:tx="http://www.springframework.org/schema/tx" xmln
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://
        http://www.springframework.org/schema/context http:/
        http://www.springframework.org/schema/mvc http://www
        http://www.springframework.org/schema/data/jpa http:
        http://www.springframework.org/schema/tx http://www.
        http://www.springframework.org/schema/security http:
    ">

    <!-- Context -->

    <import resource="config/packages.xml" />

    <context:annotation-config />

    <mvc:annotation-driven validator="validator"
        conversion-service="conversionService" />

    <!-- Converters -->

    <import resource="config/converters.xml" />

    <bean id="conversionService"
        class="org.springframework.format.support.Formatting
        <property name="converters" ref="converters" />
    </bean>
```


This file is a harness that describes the converters, the data sources, the i18n & l10n bundles, security, and other architectural components. Note that it's a harness: that means that it's a template that provides the configuration that is common to many projects; you have to provide project-specific configuration in the "*.xml" files in folder "config".

config/*.xml



These are the configuration files that provide project-specific information regarding our web information systems. It's here that we must configure the converters, the data sources, the i18n & l10n bundles, and the security configuration. There are a couple of additional configuration files called "packages.xml" and "tiles.xml": the former lists the packages in which we provide our controllers, converters, domain classes, forms, repositories, security classes, and services. So far, it's enough to have an overall idea. We'll delve into the details of these configuration files in forthcoming lectures.



Should you need more information on this lecture, please, take a look at the following bibliography:

Pro Spring MVC
Marten Deinum, Koen Serneels, Colin Yates, Seth Ladd, Christophe Vanfletere
Apress, 2012

Web information systems
David Taniar, Johanna W. Rahayu
Idea Group Publishing, 2004

Enterprise software architecture and design
Dominic Duggan
Wiley, 2012

This bibliography is available in electronic format for our students at the USE's virtual library. If you don't know how to have access to the USE's virtual library, please, ask our librarians for help.

Time for questions, please

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

Thanks for attending this lesson!