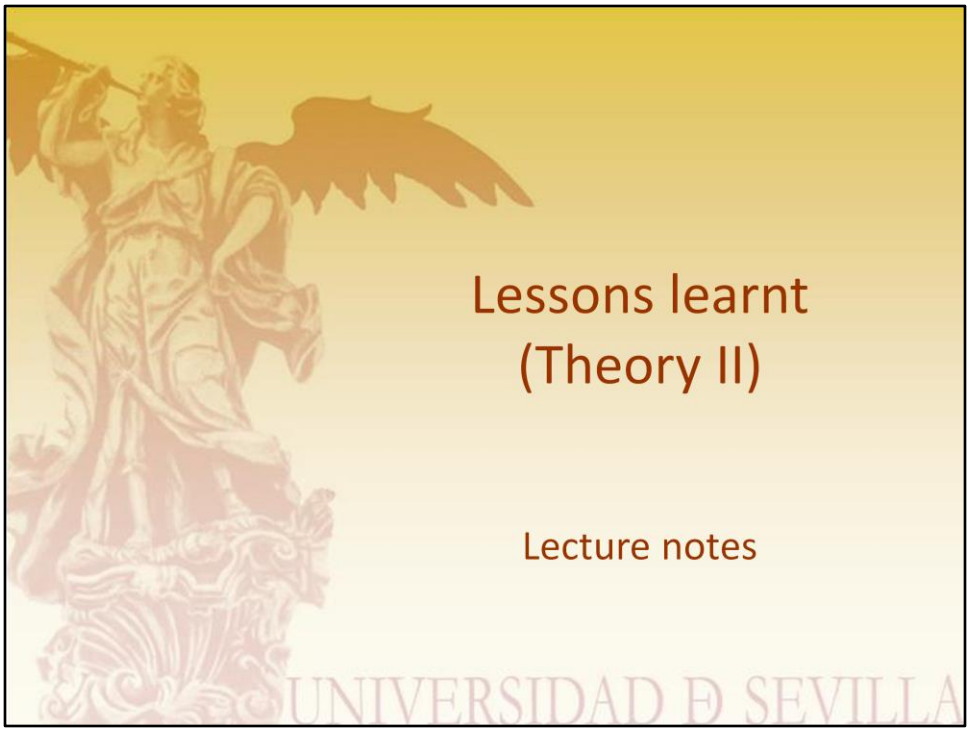
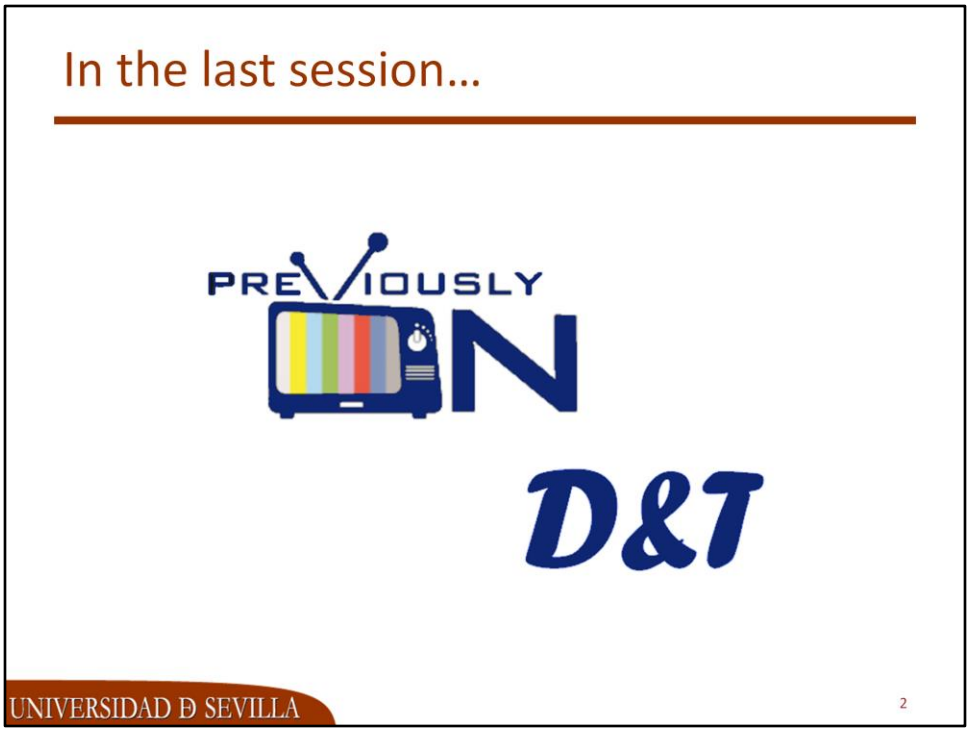


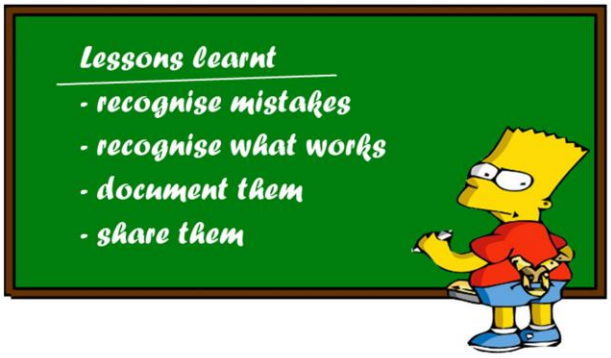
Welcome to this lecture!

--



Today, we're resuming on the theory regarding our lessons learnt. Let's first briefly recall what we presented in the previous lecture.

This is a good definition

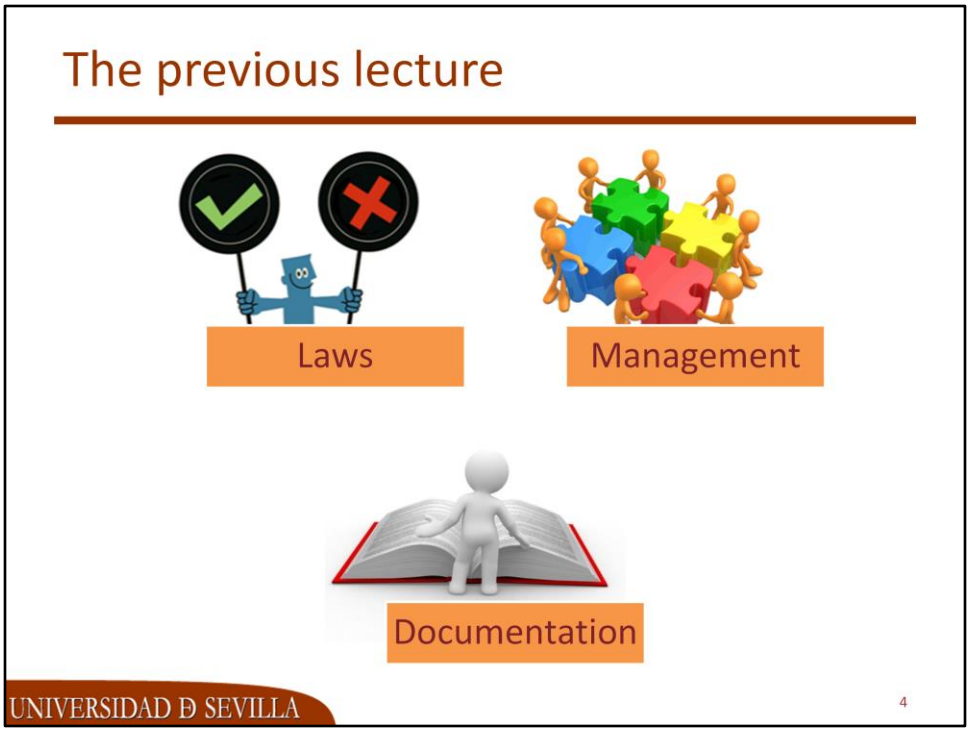


It's a lesson (god or bad) that we've learnt from our experience, that we've documented, and shared with others

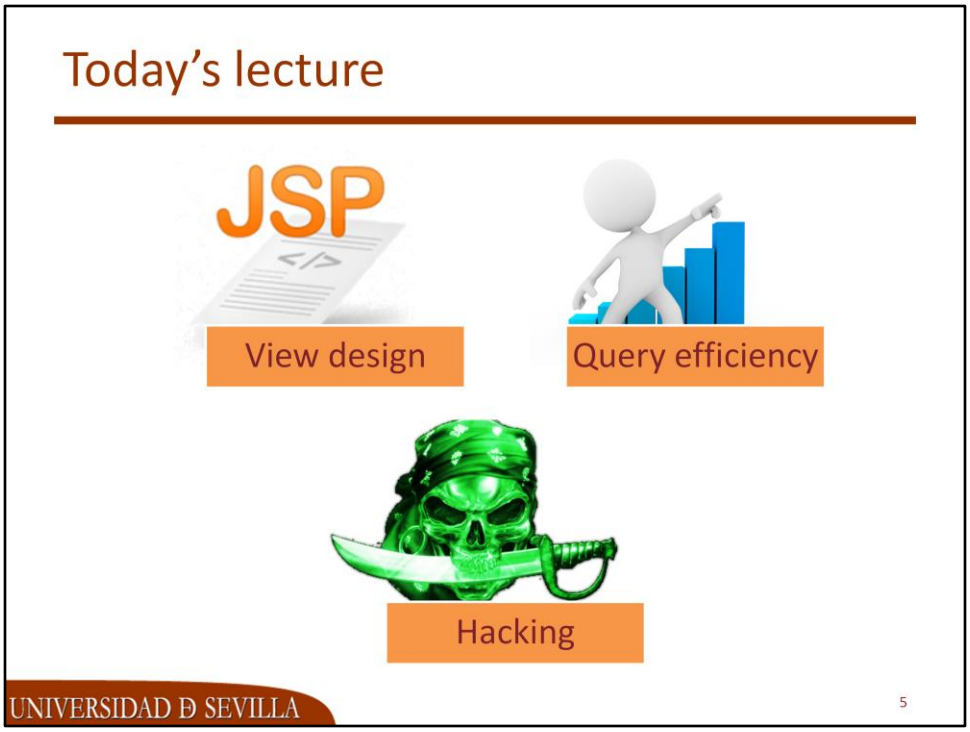
UNIVERSIDAD Ð SEVILLA

3

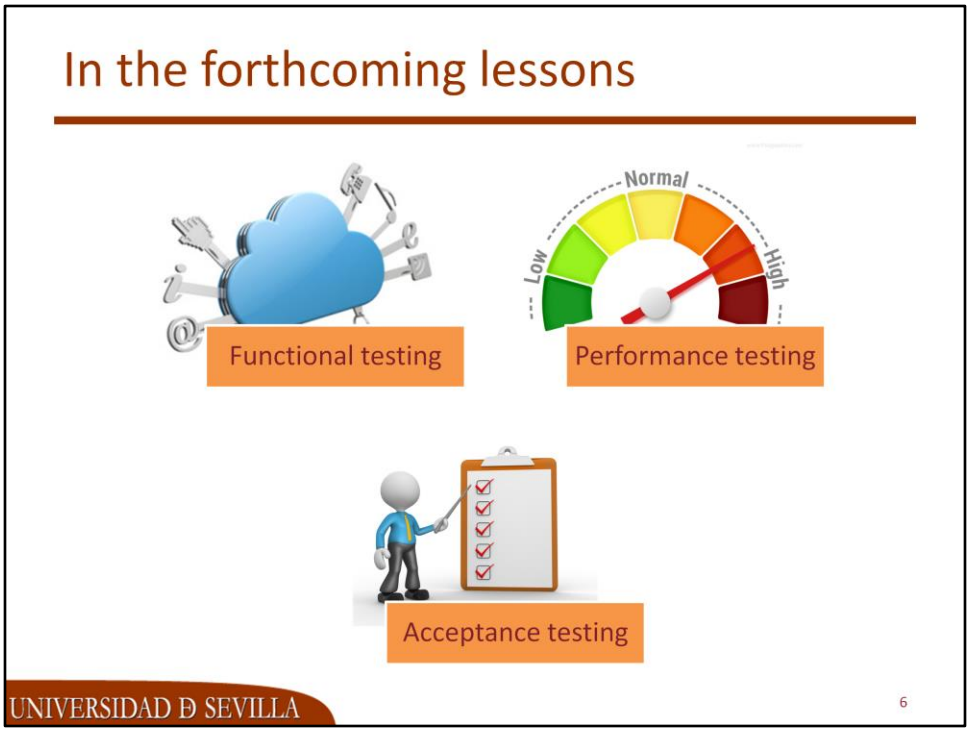
We learnt that a so-called lesson learnt is something (god or bad) that we've learnt from our experience, that we've documented, and shared with others. Recapping on the lessons learnt is the only way to improve.



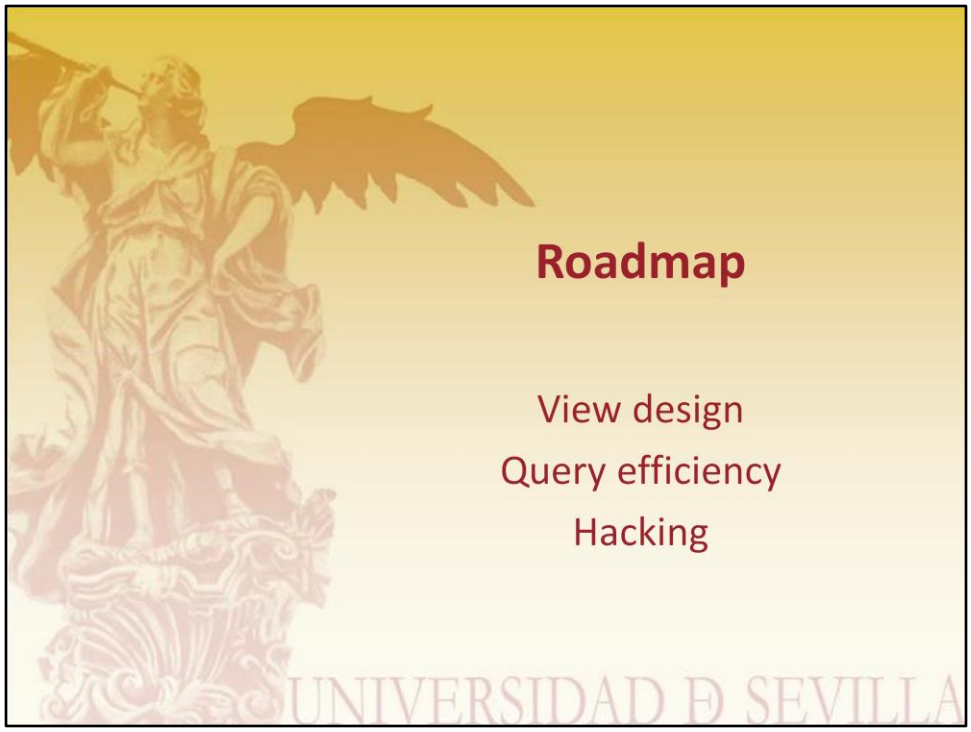
In the previous lecture, we learnt a few lessons about laws, management, and documentation.



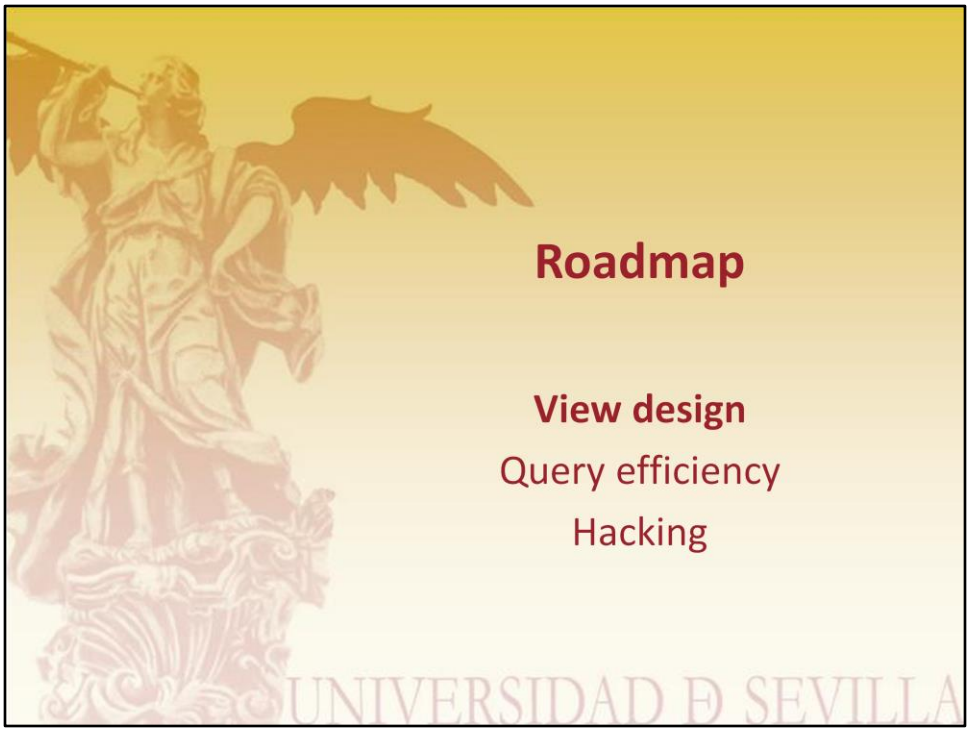
In today's lecture, we're going to delve into some details regarding designing JSP views, JPQL queries, and hacking.



In the forthcoming lessons, we'll explore functional, performance, and acceptance testing.



This roadmap shouldn't be surprising at all, right?



Let's start with a few lessons learnt regarding view design.

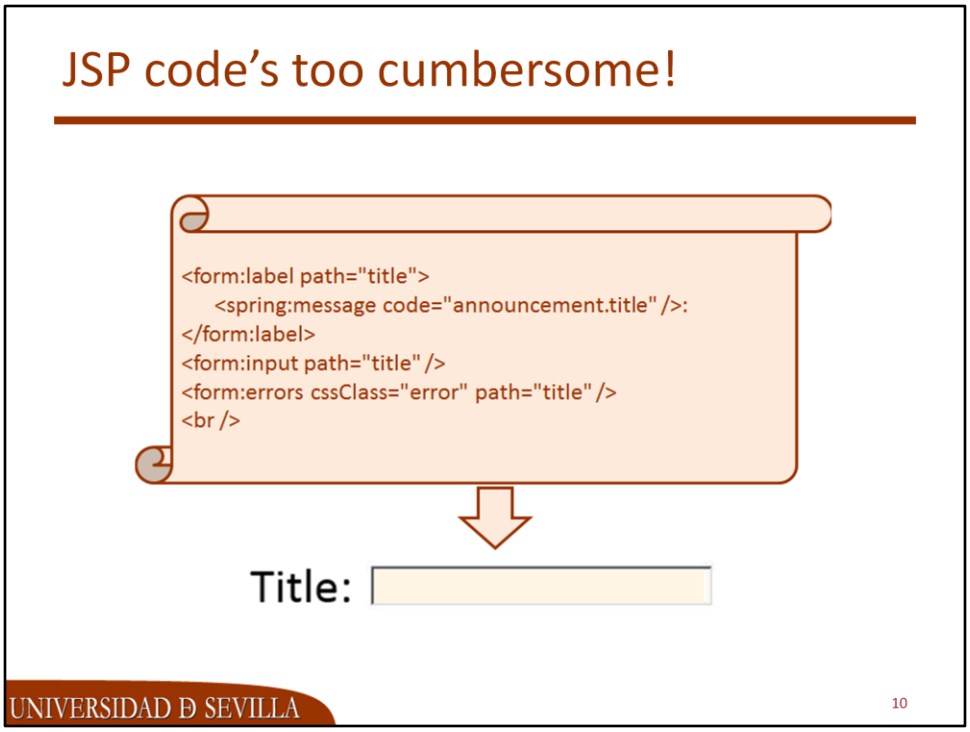
What's view design about?

It's about encoding a mock-up using JSP, Apache Tiles, CSS, HTML, and ECMA Script

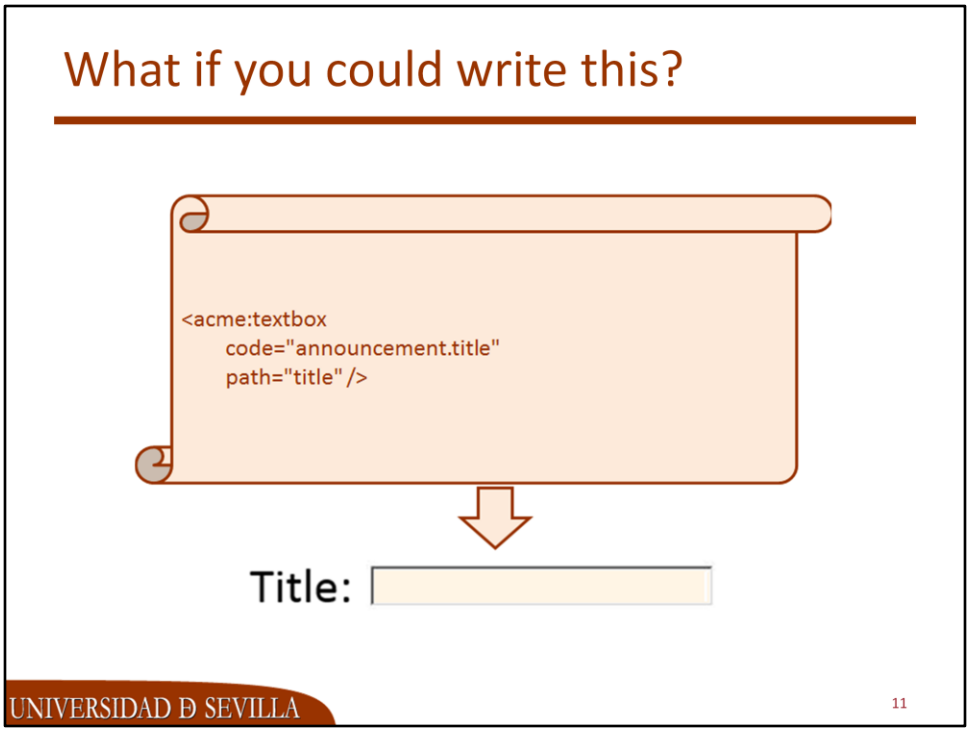
UNIVERSIDAD D SEVILLA

9

View design's about encoding a mock-up using JSP, Apache Tiles, CSS, HTML, and ECMA Script. During the previous semester, we learnt a lot about these technologies and we practiced a lot with the help of several projects.

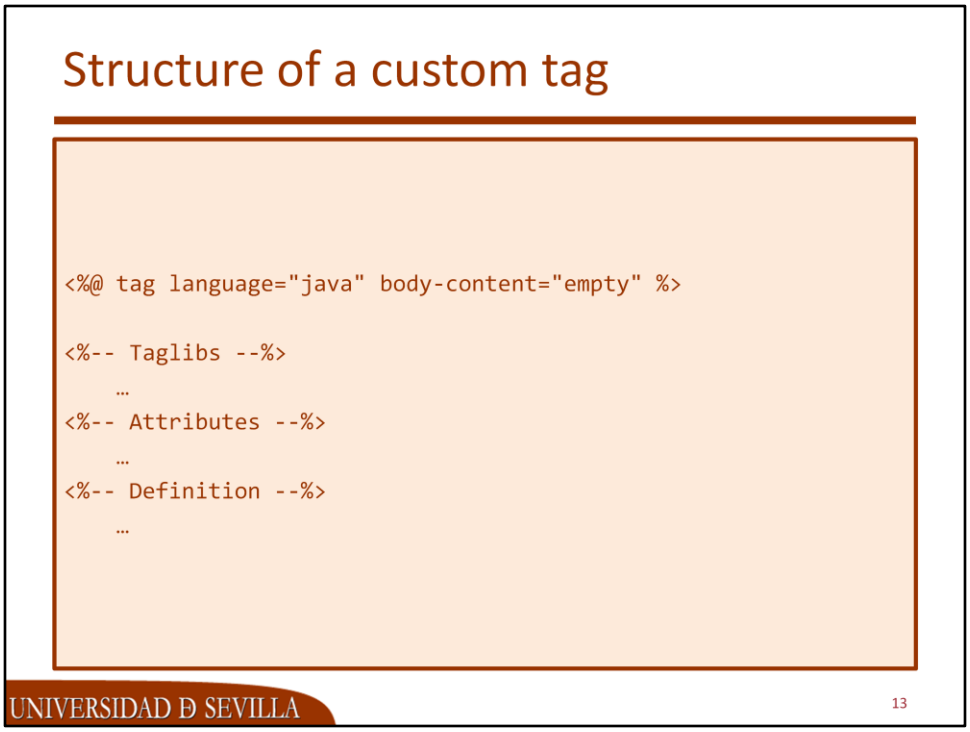


During the autumn semester, we've learnt that JSP code is too cumbersome, which means that it's very error prone. This slide shows an excerpt of a typical edition form. Note that such forms typically consist of many blocks that are composed of a label, an input box, an error message, and a new line; every block's almost identical to the previous one, except for the label to be displayed and the path to be used. Clearly, it'd be a good idea to compact this code a little, that is, to write less instructions, but get the same results.



What do you think about this code. It's a lot easier! It'd be great if you could write this code to get the same result.

Fortunately, this is possible thanks to so-called custom tags. They're not difficult at all to create and they may save you from a lot of repetitive work (and a lot of mistakes). Please, pay attention to the following slides.



This is the general structure of custom tag definition. It starts with a header that reads as follows:

<%@ tag language="java" body-content="empty" %>

And then come the tag libraries on which it relies, the definition of the attributes, and the definition of the custom tag itself. Please, take a look at the following slides, in which we provide more details on each section.

Common tag libraries

```
<%@ taglib prefix="jstl"
       uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt"
       uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="tiles"
       uri="http://tiles.apache.org/tags-tiles" %>
<%@ taglib prefix="spring"
       uri="http://www.springframework.org/tags" %>
<%@ taglib prefix="form"
       uri="http://www.springframework.org/tags/form" %>
<%@ taglib prefix="security"
       uri="http://www.springframework.org/security/tags" %>
```


In this slide, we show the most common tag libraries that we recommend to include in every custom tag definition. We won't provide any additional details on them because we explored them all in lesson "L05 – Views". Please, review the theory lecture notes of that lesson if necessary.

Attributes (I)

```
<%@ attribute name="path" required="true" %>
<%@ attribute name="code" required="true" %>

<%@ attribute name="readonly" required="false" %>

<jstl:if test="${readonly == null}">
       <jstl:set var="readonly" value="false" />
</jstl:if>
```


Attributes are introduced by means of “<%@ attribute>” declarations. For every attribute, we must specify a name, whether it’s required or not, and an optional type (more on attribute types later). For instance, in this slide, we’re defining three attributes: “path” and “code”, which are required, and “readonly”, which isn’t required. Note that the default value of a non-required attribute is “null”; thus, it’s very common in practice that after declaring non-required attributes developers include instructions to endow them with appropriate values. For instance, if the user doesn’t specify a value for attribute “readonly”, we then set it to “false”.

Attributes (II)

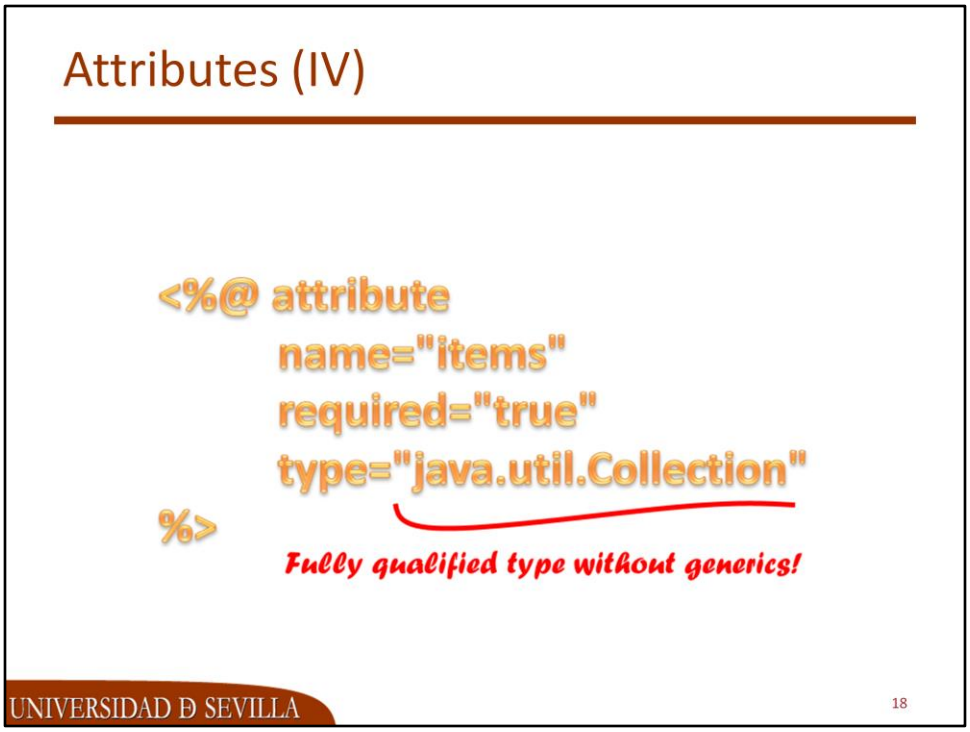
UNIVERSIDAD Ð SEVILLA

16

Note that we didn't declare an explicit type for the attributes that we presented in the previous slide. If you were asked to guess a type for them, you might well think that they are of type "Object". That's a good guess because we're working in a Java context. Unfortunately, it's a wrong guess!

Attributes (III)

Attributes are of type “String”, by default.



If you need to change the default type of an attribute, you must declare it in the "<%@ attribute>" declaration. Note that you must specify the fully qualified type and note, too, that generics aren't allowed. That is, if you need to declare a required attribute called "items" that holds a collection of, say, certifications, you must declare it like in this slide. Just note that the type must be "java.util.Collection", not "java.util.Collection<Certification>", "Collection<Certification>", or just "Collection"; that won't work.

And the custom definition

```
<div>
    <form:label path="${path}">
        <spring:message code="${code}" />
    </form:label>
    <form:input path="${path}" readonly="${readonly}" />
    <form:errors path="${path}" cssClass="error" />
</div>
```


Finally comes the definition of the custom tag itself. It's a piece of JSP code in which you can freely use the attributes that were declared previously as if they were regular model variables.

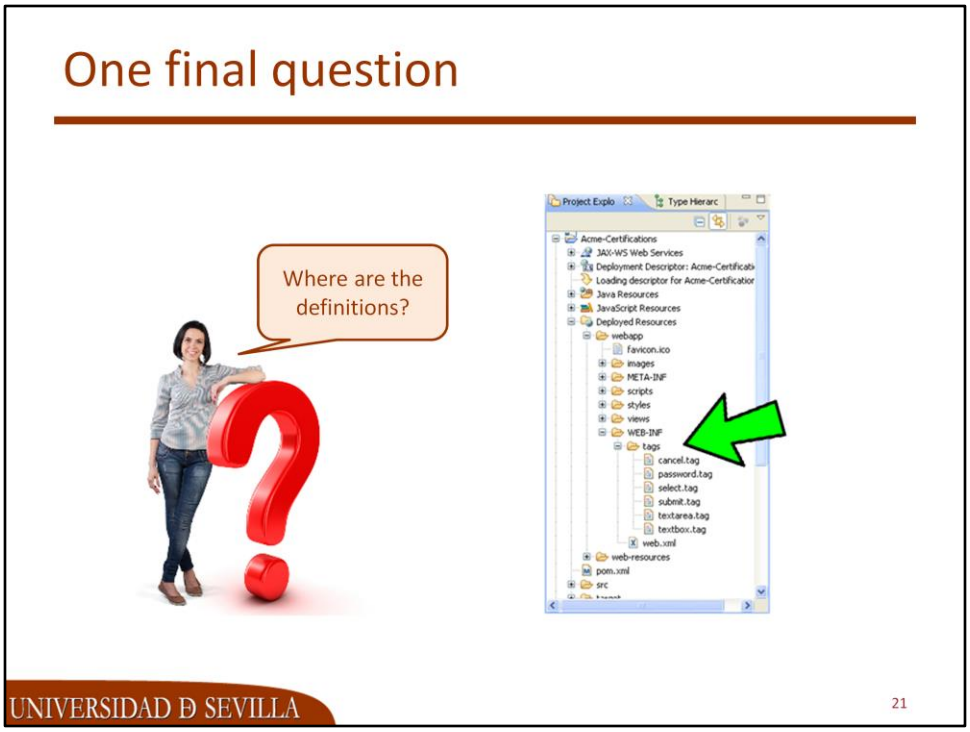
Using custom tags

```
<%@page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
…
<%@ taglib prefix="acme" tagdir="/WEB-INF/tags" %>
…
<acme:textbox code="announcement.title" path="title" />
<acme:textbox code="announcement.moment" path="moment" />
<acme:textarea code="announcement.description" path="description" />
<acme:select code="announcement.certification" path="certification"
    items="${certifications}" onchange="javascript: reloadExams()"
    itemLabel="title" id="certifications" />
…
```

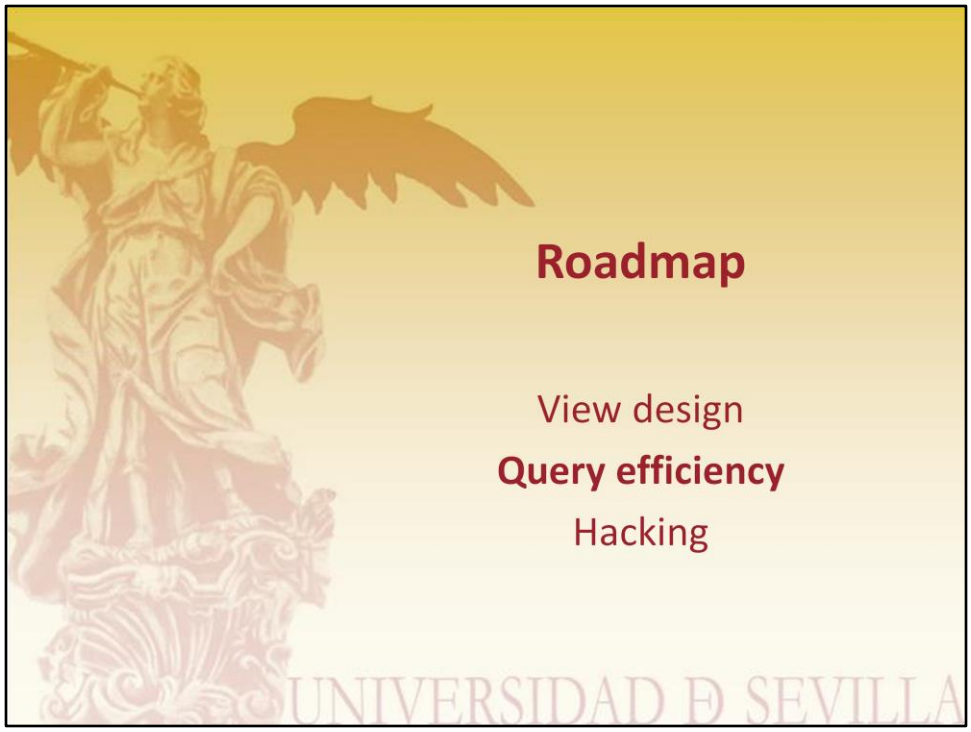

Once your custom tags are ready, you can use them very easily. The only thing you have to do is to add the following instruction to the JSP document in which you wish to use them:

```
<%@ taglib prefix="acme" tagdir="/WEB-INF/tags" %>
```

This instructs the JSP compiler to read your custom tag definitions, so that you can use them as if they came from a regular tag library.



One final question: where must you store the definitions of your custom tags? They must be stored in folder "webapp/WEB-INF/tags". (Note that they are defined in ".tag" files.) By default, our template provides the following custom tags: "acme:textbox", "acme:textarea", "acme:select", "acme:submit", and "acme:cancel"; please, explore them to learn how to use them and create as many as you need in your projects.



In the previous section, you learnt something that's very useful to simplify your views. That's a very good piece of news, isn't it? Let's keep learning a few more things about queries; inefficient queries, to be more precise.

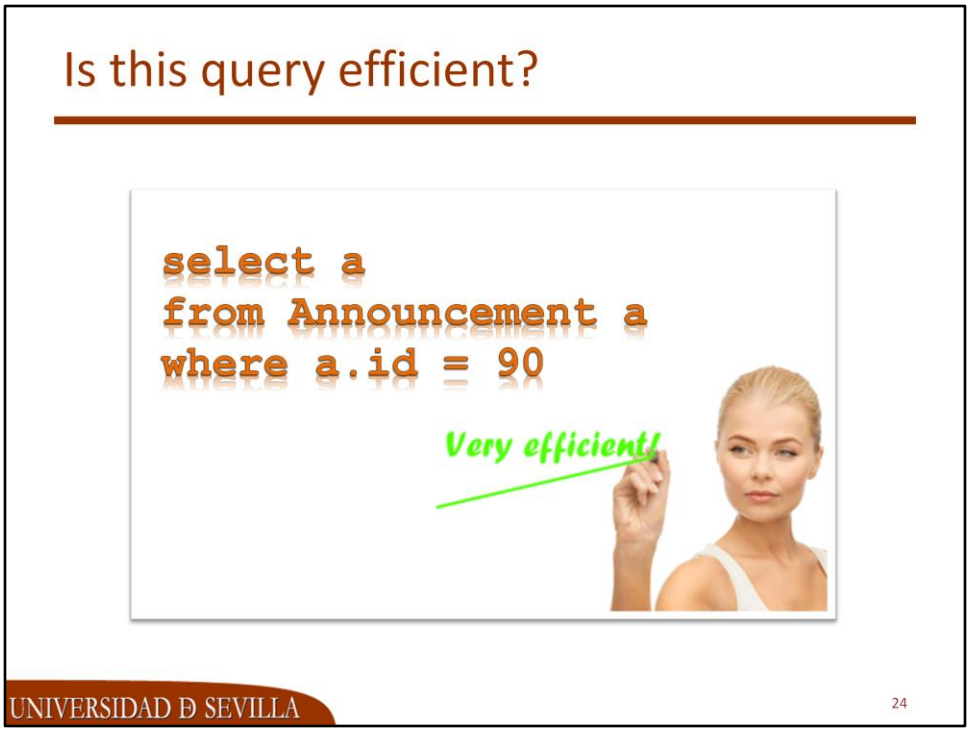
What's query efficiency about?

It's about not wasting computer resources when executing queries, which requires to design them carefully and to fetch their data in O(log *n*) time

UNIVERSIDAD Ð SEVILLA

23

As usual, we start with a question: what's query efficiency about? It's about not wasting computer resources when executing queries, which requires to design them carefully and to ensure that their data can be retrieved in logarithmic time; linear time is ok when working with a small dataset, but it's very inefficient when dealing with medium- or large-sized data; quadratic or higher times are simply unacceptable. To understand the insights of efficiency, we are going to explore a couple of queries.



For instance, let's analyse this simple query that retrieves an announcement with a given identifier or null if such an announcement does not exist. This query's very efficient by default! That is: there's nothing you have to do so that it is run in O(log *n*) time.

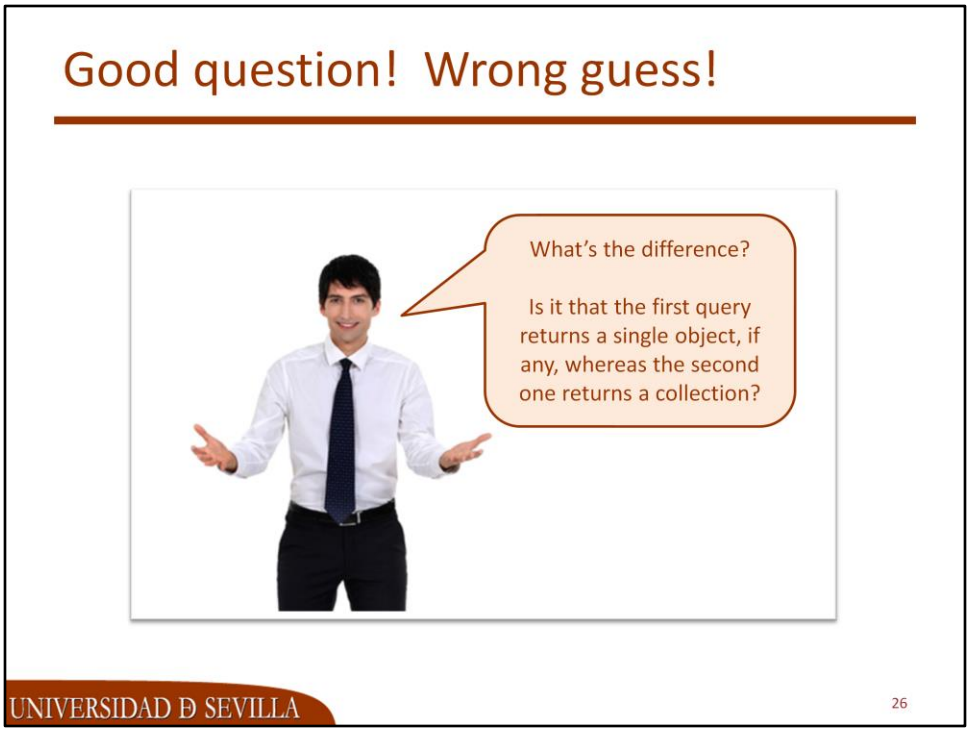
What about this one?

```
select a
from Announcement a
where a.moment >
    '2017/08/10 10:00'
```

UNIVERSIDAD D SEVILLA

25

Let's now analyse this query, which retrieves the announcements that are scheduled after a given date, if any. This query's very inefficient by default! That is: unless you do something else, it will not run in O(log *n*) time.



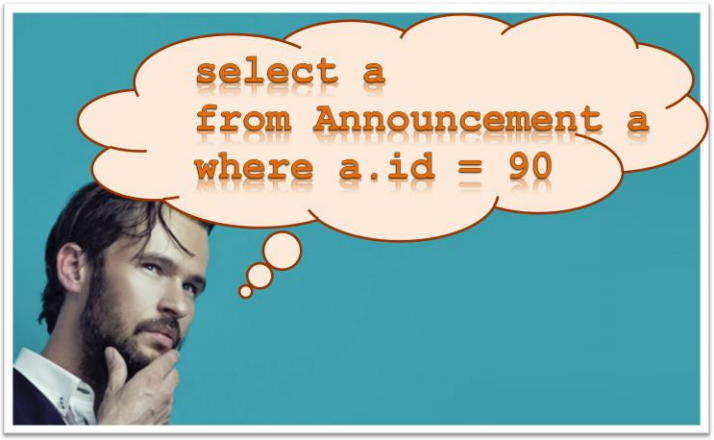
What's the difference between them? This is a good question. You might think that the difference's that the first query returns a single object, if any, whereas the second query returns a collection. Unfortunately, that's not the reason. Wrong guess!

Data looks like this in the database

id	version	description	moment	title	exam_id
69	0	Blah, blah, blah	09/09/2017 10:00	Announcement 01	23
15	1	Blah, blah, blah	10/01/2017 09:30	Announcement 02	22
20	1	Blah, blah, blah	01/12/2016 19:45	Announcement 03	22
110	2	Blah, blah, blah	12/12/2018 08:00	Announcement 04	123
70	3	Blah, blah, blah	13/02/2017 10:00	Announcement 05	9
120	2	Blah, blah, blah	08/04/2016 14:00	Announcement 06	123
125	1	Blah, blah, blah	15/10/2017 10:00	Announcement 07	34
90	1	Blah, blah, blah	12/07/2017 17:15	Announcement 08	54
35	4	Blah, blah, blah	13/02/2018 10:00	Announcement 09	908
43	5	Blah, blah, blah	06/07/2017 10:00	Announcement 10	245
40	1	Blah, blah, blah	30/01/2017 20:30	Announcement 11	65
55	2	Blah, blah, blah	20/09/2018 15:00	Announcement 12	65

In order to understand the reason, it's necessary to analyse how data are stored and how they are located. This slide shows a sample instance of the table in which the announcements are stored. Note that they aren't sorted according to any obvious criteria; initially, they are stored according to the insertion moment, but they soon get untidy when they are updated.

How to run this query on that table?

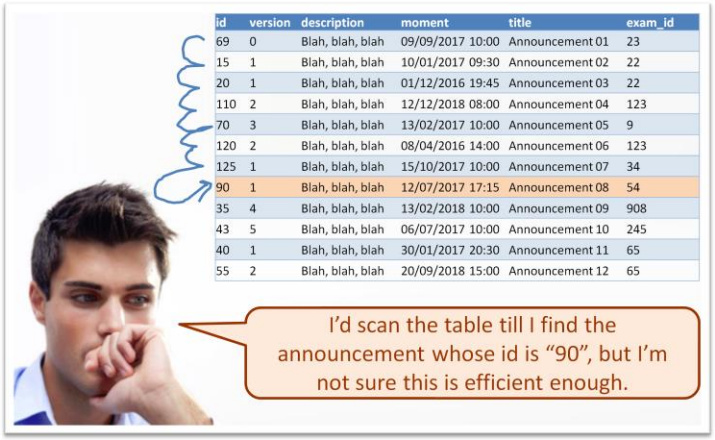


UNIVERSIDAD D SEVILLA

28

Now, imagine that you've got to implement the MySQL query engine and that you have to run the query in this slide on the previous table. What would you do?

Really? It works... inefficiently

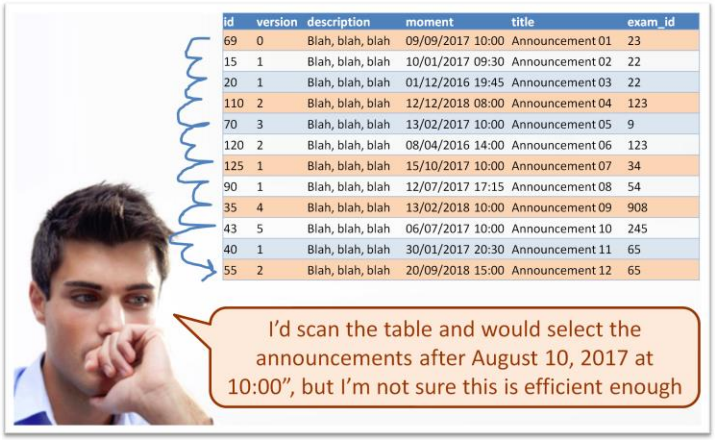


Some of you might answer something like "I'd scan the table until I find the announcement whose id is 90". Ok, that works from a functional point of view, but it's inefficient.

And what about this one?

And what about executing the other query? What would you do?

Really? It works... inefficiently

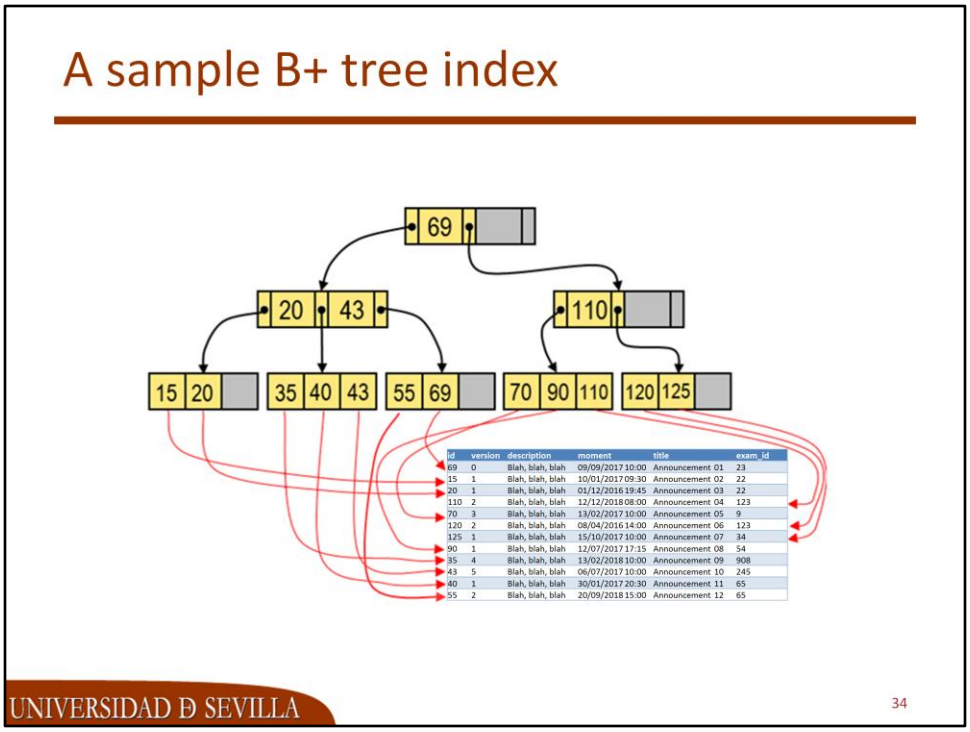


id	version	description	moment	title	exam_id
69	0	Blah, blah, blah	09/09/2017 10:00	Announcement 01	23
15	1	Blah, blah, blah	10/01/2017 09:30	Announcement 02	22
20	1	Blah, blah, blah	01/12/2016 19:45	Announcement 03	22
110	2	Blah, blah, blah	12/12/2018 08:00	Announcement 04	123
70	3	Blah, blah, blah	13/02/2017 10:00	Announcement 05	9
120	2	Blah, blah, blah	08/04/2016 14:00	Announcement 06	123
125	1	Blah, blah, blah	15/10/2017 10:00	Announcement 07	34
90	1	Blah, blah, blah	12/07/2017 17:15	Announcement 08	54
35	4	Blah, blah, blah	13/02/2018 10:00	Announcement 09	908
43	5	Blah, blah, blah	06/07/2017 10:00	Announcement 10	245
40	1	Blah, blah, blah	30/01/2017 20:30	Announcement 11	65
55	2	Blah, blah, blah	20/09/2018 15:00	Announcement 12	65

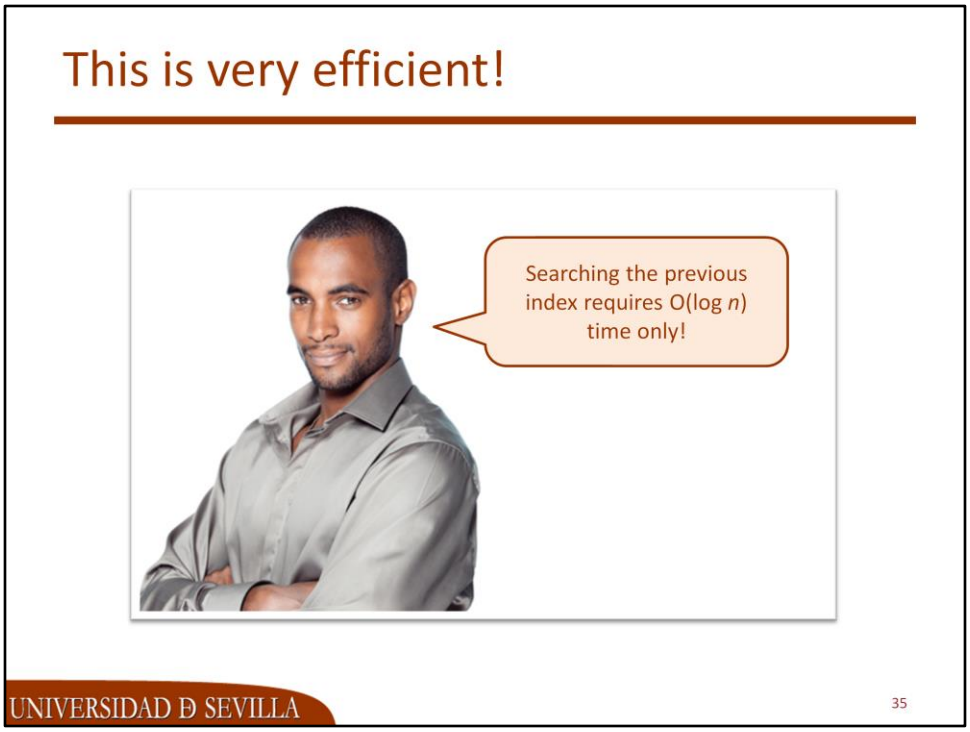
Your answer might well be as follows: "I'd scan the table and would select the announcements after August 10, 2017 at 10:00." Ok, that also works from a functional point of view, but it's not efficient at all.

Unfortunately, the previous approaches to implement the sample queries are efficient because they require O(*n*) time to retrieve their data, where *n* denotes the number of entities in the table being scanned. Unfortunately, linear time is typically too much time when dealing with medium-sized or large datasets. With such datasets, it's a must that the queries can retrieve their data in O(log *n*) time.

Fortunately, the solution to the previous problem's quite simple: use a B+ tree index to scan the table.



This slide shows a B+ tree index for the "id" attribute of the announcements. Note that finding a customer given an identifier involves the following steps: retrieve the root node of the B+ tree index (constant time), navigate to the next level of the tree (logarithmic time regarding the number of data in each node), repeat the navigation until a leaf node is reached (logarithmic time regarding the number of data in the table), and find the pointer to the data within the leave node (logarithmic time regarding the number of data in each leaf node).

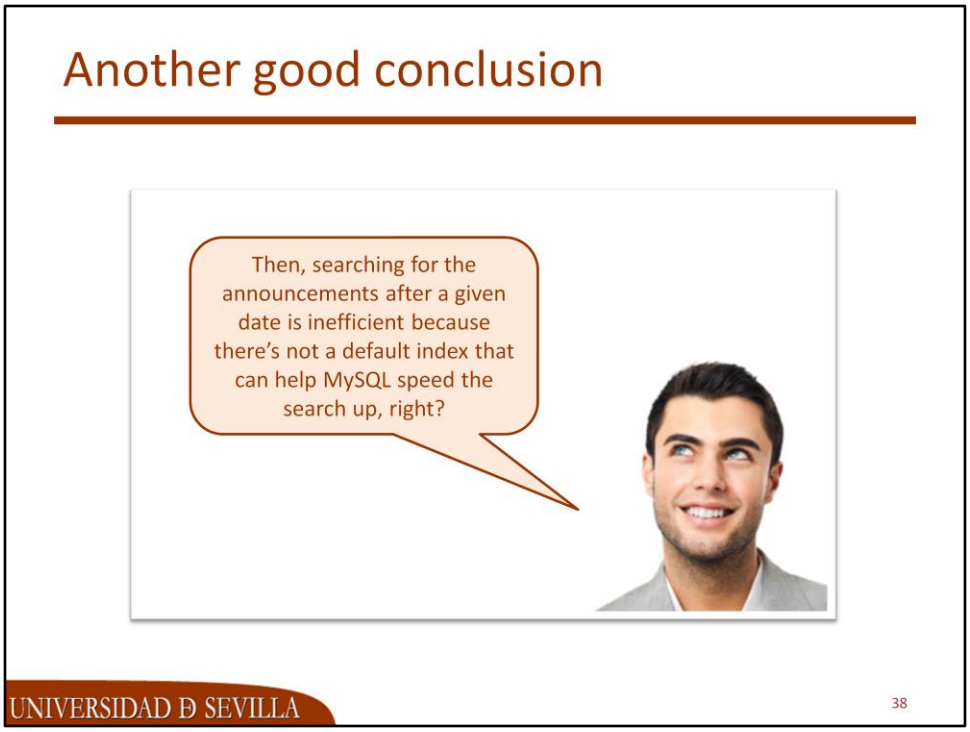


The important piece of news is that searching for information in a B+ tree requires no more than O(log *n*) steps, which is a lot more efficient than scanning the table sequentially. Believe us: as the size of a table grows (think of tables with several GiB of information), the difference between O(*n*) and O(log *n*) becomes more and more evident. (If you don't trust us, create a spreadsheet and simulate the behaviour of both time orders for large values of variable *n.*)

NOTE: Formally, the time complexity is roughly O(*A* + *B* * *L* + *C*), where *A* denotes the time to fetch the root node, *B* denotes the time to find the next node to retrieve, *L* denotes the number of levels of the tree, and C denotes the time to find the pointer to the data in the leave nodes; simply put: O(1 + log *m* * log *n* + log *m*). Note that *m* can be assumed to be very small with regard to *n*, so the conclusion is that the time to retrieve data from a B+ tree index is O(log *n*). Furthermore, the size of the inner and the leaf nodes can be fined tuned so that the overall time is O(1)!

There's another good piece of news: the JPA technology analyses your domain model and creates an index for every @Id attribute and every role attribute it finds. That means that searching for an entity that has a given identifier or fetching an entity that is related to another one by means of a role attribute is very efficient.

A few slides ago, we mentioned that a query like “select a from Announcement a where a.id = 90” is efficient. The reason should be clear now: JPA’s created an index for the “id” attribute, so MySQL can locate an announcement with a given identifier in O(log *n*) time.



We also mentioned that a query like "select a from Announcement a where a.moment > '2017/01/01 10:00'" is inefficient. The reason is that JPA does not define a default index on attribute "moment", so MySQL has to scan the whole announcement table to find the ones that fulfil the query, which requires O(*n*) time.

Fortunately, it's not very difficult to create your own indices. Please, keep reading.

Defining an index

```
@Entity
@Access(AccessType.PROPERTY)
@Table(indexes = { @Index(columnList = "moment") })
public class Announcement extends DomainEntity {
    …
}
```

UNIVERSIDAD D SEVILLA

40

For instance, in this slide, we've defined an index on attribute "moment". Re-compile your project and re-populate your database. That simple annotation may be the difference between an application that takes longer, longer, and longer to fetch the list of announcements that are active, and an application that can deal well with hundreds of thousands of announcements.

Defining a multi-index

```
@Entity
@Access(AccessType.PROPERTY)
@Table(indexes = {
    @Index(columnList = "minimumMark, minimumScore")
})
public class Exam extends DomainEntity {
    …
}
```


There are cases in which you need to define an index on multiple attributes. This slide shows a good example in which the entity “Exam” is endowed with an index on its “minimumMark” and its “minimumScore” attributes; this allows to search efficiently for exams with given required minimum marks and scores. Note that the attributes are listed in a string and separated by commas.

Defining a unique index

```
@Entity
@Access(AccessType.PROPERTY)
@Table(uniqueConstraints =
       { @UniqueConstraint(columnNames = "username") })
public class UserAccount
             extends DomainEntity implements UserDetails {
    …
}
```


If you need to define a unique index, then you must use the annotation that we show in this slide.

NOTE: in a previous lecture, we taught that you had to use an "@Column" annotation to define a unique index; it works pretty well, but we recommend that you should start using the new "@Table" annotation, since it is more general and homogeneous.

Defining a unique multi-index

```
@Entity
@Access(AccessType.PROPERTY)
@Table(uniqueConstraints = {
        @UniqueConstraint(columnNames = {
                "email", "phone"
        })
})
public abstract class Actor extends DomainEntity {
    …
}
```


You can also define a unique multi-index. For instance, this slide shows how to define such an index on attributes “email” and “phone” of entity “Actor”. This index helps speed up queries that involve both attributes and also prevents two different actors from having the same email and phone. Please, note the inconsistency regarding the notation: in this case, the list of attributes is specified as an array. Sorry, that’s technology!

A note on role attributes

```
@Entity
@Access(AccessType.PROPERTY)
@Table(uniqueConstraints =
      { @UniqueConstraint(columnNames =
                          {"announcement_id", "owner_id"}) })
public class Registration extends DomainEntity {
    …
}
```

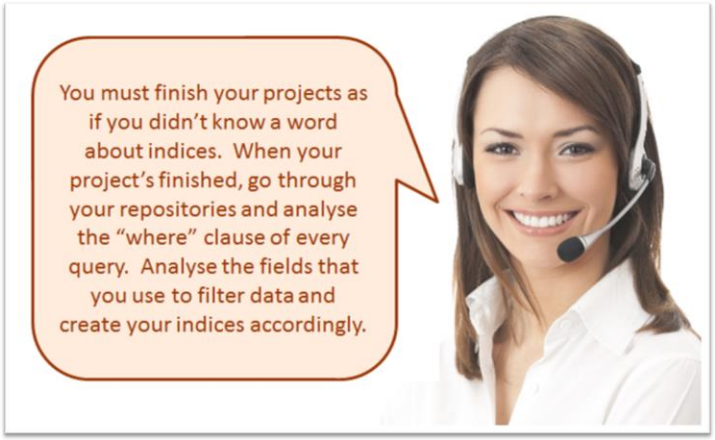

Finally, we should report on an issue regarding role attributes. This slide shows an example in which a unique multi-index is defined on attributes “announcement” and “owner” of entity “Registration”. Please, pay attention to the names of the attributes, since they have a trailing “_id”. The reason is that indices have to do with the relational implementation of a Java domain model; that means that the annotations work on table attributes, not on Java attributes! JPA translates every Java attribute into a table attribute with the same name, except for role attributes, which get an “_id” suffix automatically. If you forget this suffix, then you’ll get a difficult to interpret error, namely:

```
PopulateDatabase
----------------------
Initialising persistence context `Acme-Certifications'...
[PersistenceUnit: Acme-Certifications] Unable to build Hibernate SessionFactory
```

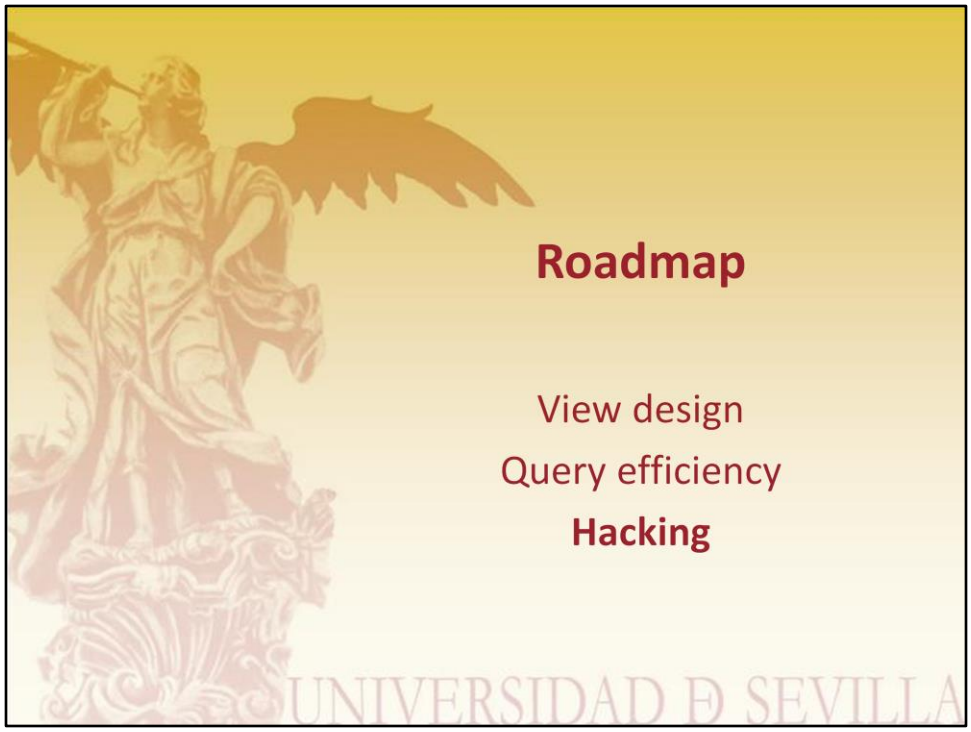
Nothing else. Please, simulate this problem in practice so that you get familiar with it.

Ok, now you know the theory. What should you do to put this theory into practice? That's a good question!

This is a good answer



Our recommendation is that you must finish your projects as if you didn't know a word about indices. When your project's finished, go through your repositories and analyse the "where" clause of every query. Analyse the fields that you use to filter data and create your indices accordingly; just recall that JPA creates indices on your "id" attributes and on every role attribute; your must declare the others explicitly.



Did you know the impact that an index may have on an application? Please, take this very seriously. As seriously as hacking, which is the next section in our roadmap.

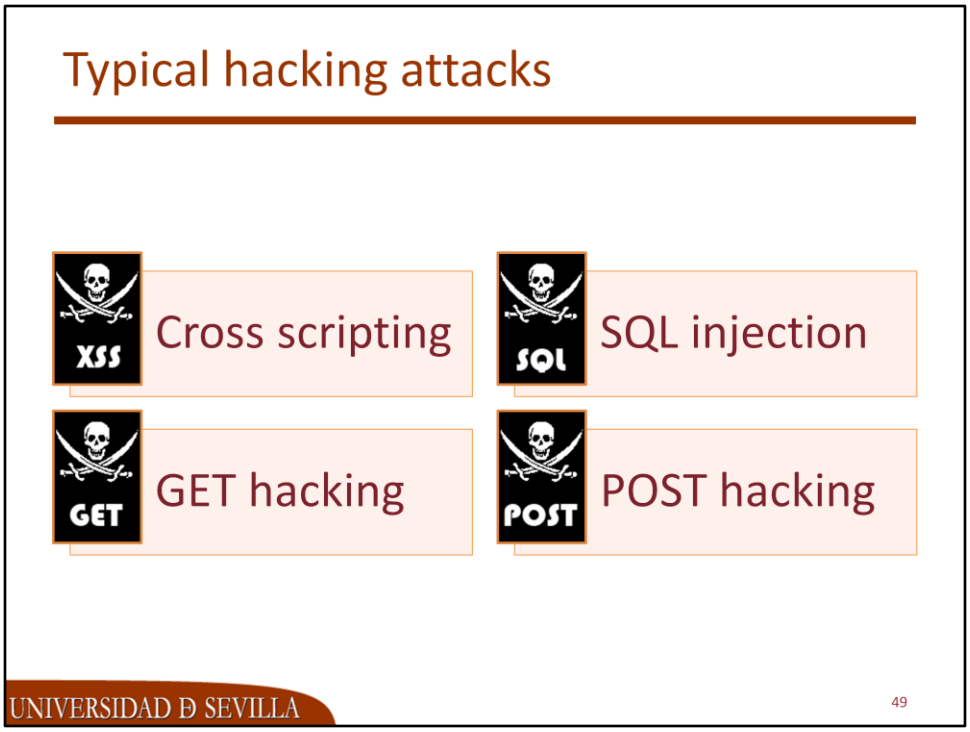
What's hacking about?

It's about doing something you're not authorised to do with an application, independently from whether the authorisation is implicit or explicit

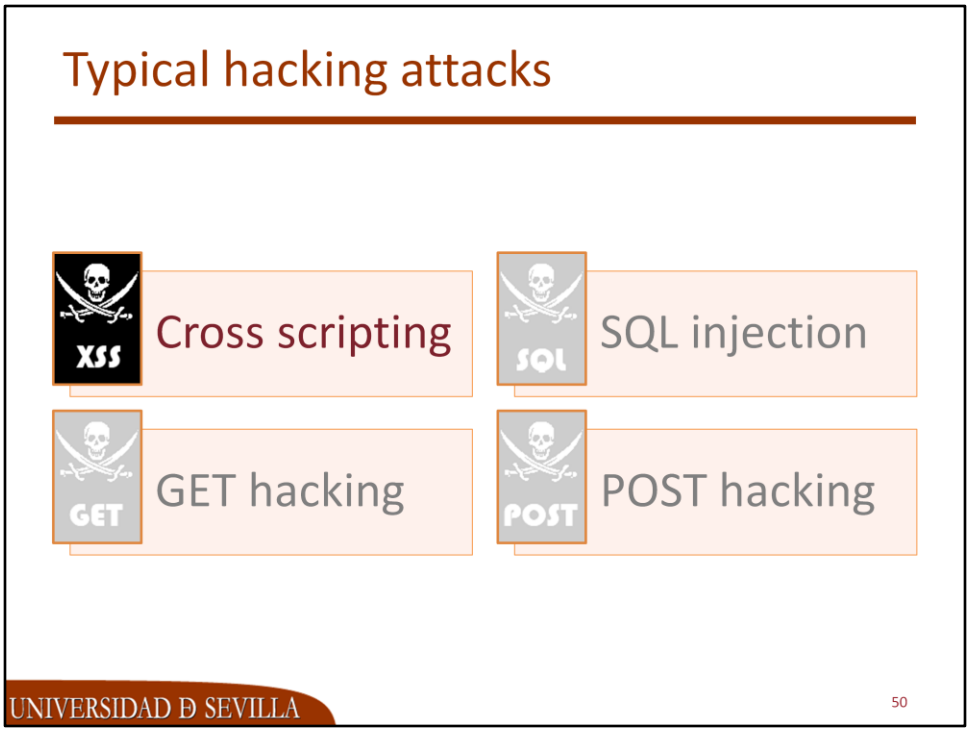
UNIVERSIDAD Đ SEVILLA

48

Hacking's about doing something you're not authorised to do with an application. For instance, you cannot register to an announcement in Acme Certification using the account of another user; neither should you enter a piece of data that uploads a virus to Acme's systems. We don't think you're interested in hacking any systems, but, unfortunately, there are hackers around and we should not ease their lives regarding our systems.



In this slide, we present the most typical hacking attacks that a web information system may suffer: cross scripting, SQL injection, GET Hacking, and POST hacking. We'll delve into the details in the following slides.



Let's start with cross scripting.

What's cross scripting (aka XSS)?

It consists in entering a piece of data that is a piece of malicious ECMA Script

UNIVERSIDAD Ð SEVILLA

51

Cross scripting, or XSS for short, consists in entering a piece of data that's actually a piece of malicious ECMA Script. For instance: assume that Dr. Evil registers to Acme Certification and enters "<script> alarm('Hacked!'); </script>" as his name. What do you think might happen when an administrator lists the customers? Yes, unless we prevent it, the administrator will get a pop-up message that reads "Hacked!". This piece of script is not malicious at all, but it might be. Hackers are aware of the security holes in current browsers, and they never miss an opportunity to use them to hack remote systems.

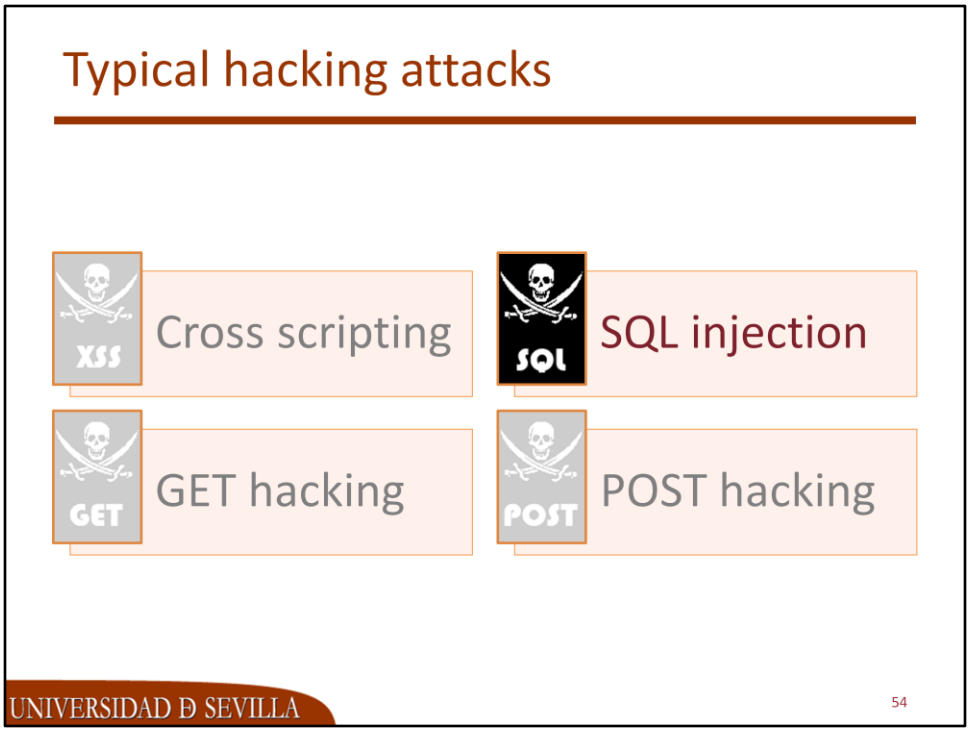
What can you do to prevent cross scripting? That's a good question!

This is a good answer!

UNIVERSIDAD D SEVILLA

53

You must do two things. First, and foremost important: never print any data out using a plain “${data}” JSP expression; that expression outputs the result of evaluating “data” literally; if that value’s a piece of ECMA Script, it will execute on the browser. Don’t forget to use the “jstl:out” instruction, which translates special characters like “<“ or “>” into appropriate HTML entities that prevent the script from executing. For instance, if “data” is a string with value “<script> alarm('Hacked!'); </script>”, then “${data}” outputs “<script> alarm('Hacked!‘); </script>”, whereas “jstl:out” outputs “<script> alarm("Hacked!"); </script>”. Realise the difference, it’s very important. Our second recommendation is that you should explore a validation annotation called “@SafeHtml”; it allows to reject data that is suspicious to be a script; please, play a little with it.



SQL injections are quite a common hacking technique. Let's explore it.

What's SQL injection?

It basically consists in entering a piece of data that is actually a piece of SQL

UNIVERSIDAD D SEVILLA

55

It basically consists in entering a piece of data that is actually a (fragment of a) SQL statement. For instance, assume that a user has access to a login form and enters "admin" as username and "' or 'A' = 'A" as password. Can you foresee any problems? Apparently, this is a wrong password and the application should reject it. Let's, however, take a look at how *many typical* applications work internally. (Please, note that we emphasise "many typical". It's very common in the real-world, but not in the kind of web information systems that we're developing in D&T.)

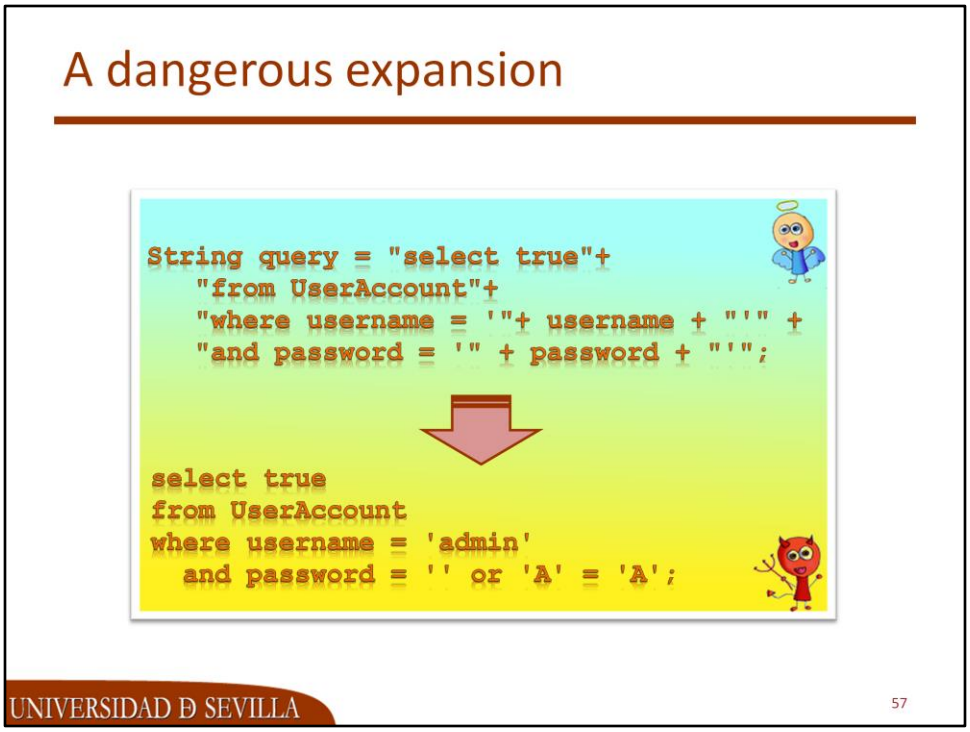
A typical programmer's SQL Query

```
String query = "select true"+
   "from UserAccount"+
   "where username = '"+ username + "'" +
   "and password = '" + password + "'";
```

UNIVERSIDAD Ð SEVILLA

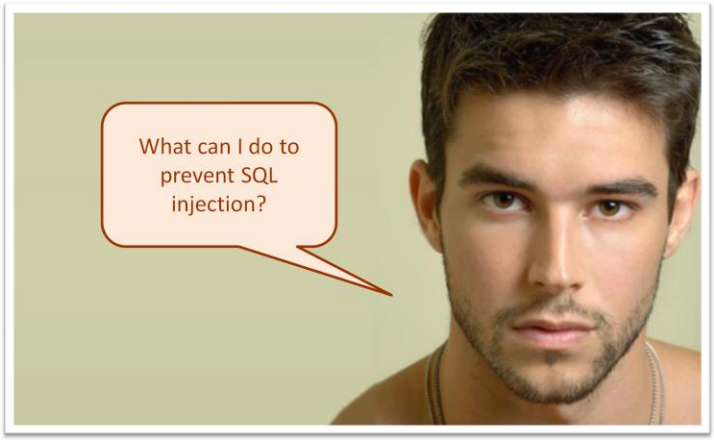
56

This is a typical query to check the credentials provided by a user. If you search the Web, you'll find hundreds of pages with a very similar query. Apparently, it works well, we mean: given a username and a password, it returns "true" if they match. If true's returned, then the application knows that the user must be granted access; if "null" is returned, then the application mustn't grant any access to the user. Pretty simple!



But let's expand the previous query when variable "username" has value "admin" and variable "password" has value "' or 'A' = 'A'". Did you realise the problem? Please, review this slide as many times as necessary; it's important that you understand that the problem's that this query returns "true" independently from the user name or password used. Did you realise the problem now? Realise that "A and B or C" is interpreted as "(A and B) or C"; so it suffices that "C = true" for the whole expression to hold. Substitute "A" for "username = 'admin'", "B" for "password = ''", and "C" for "'A' = 'A'" and the reason why the previous query matches any username and password should be crystal clear.

This is a good question!

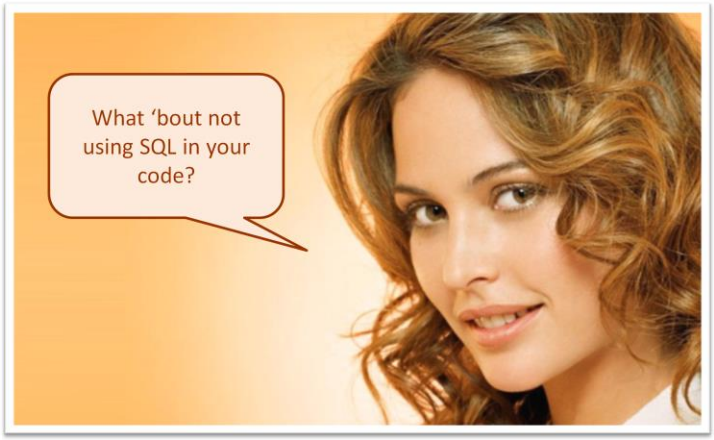


UNIVERSIDAD Đ SEVILLA

58

This is a good question: how can you prevent SQL injection attacks?

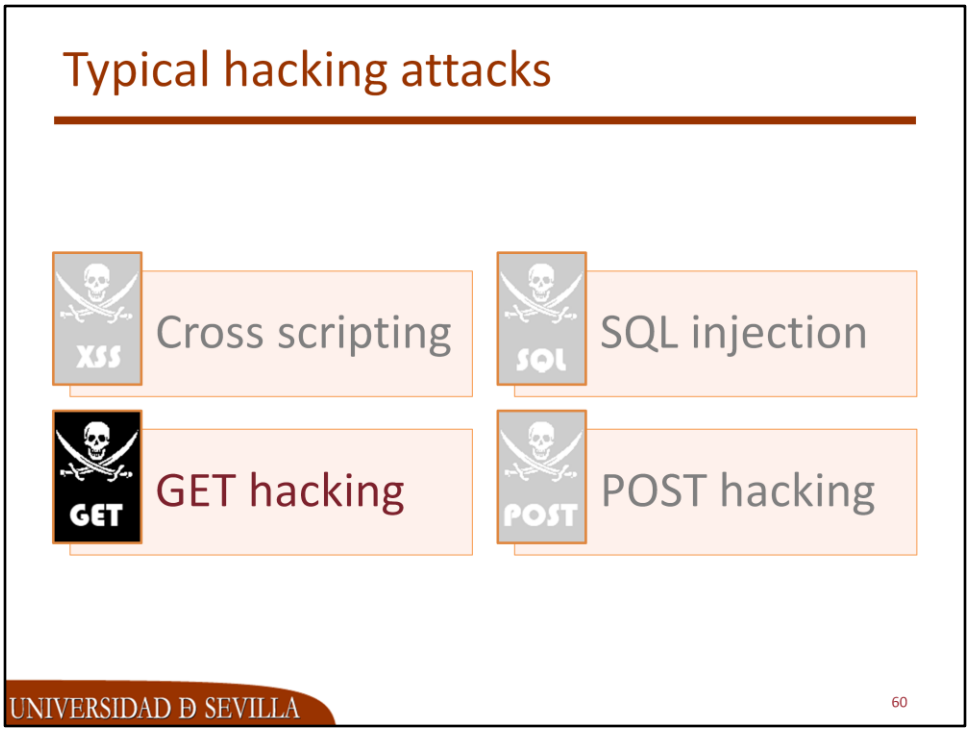
This is a good answer!



UNIVERSIDAD Đ SEVILLA

59

And this is the best answer: don't use SQL in your code! Please, recall that we're using JPQL and repositories; there's absolutely no way to SQL-inject an application that uses these technologies. In cases in which you need to use a SQL query, please, explore your SQL API and search for a means to provide parameters to your queries. Do never create a query by concatenating strings.



SQL injection is very easy to prevent (like cross scripting). Unfortunately, it's difficult to believe how many existing web information systems can be hacked using these simple techniques. Let's now delve into the details of GET hacking.

What's GET hacking?

It's about entering a URL address that is not shown in any web pages, but exist!

UNIVERSIDAD D SEVILLA

61

Simply put, GET hacking's about entering URL addresses that are not displayed in any web pages, but exist! For instance, we're pretty sure that you've made an attempt not to provide any links to "/certification/administrator/edit.do" in a web page that is intended to be browsed by a customer; but this address exists and nothing prevents a hacker from making his or her browser for it.

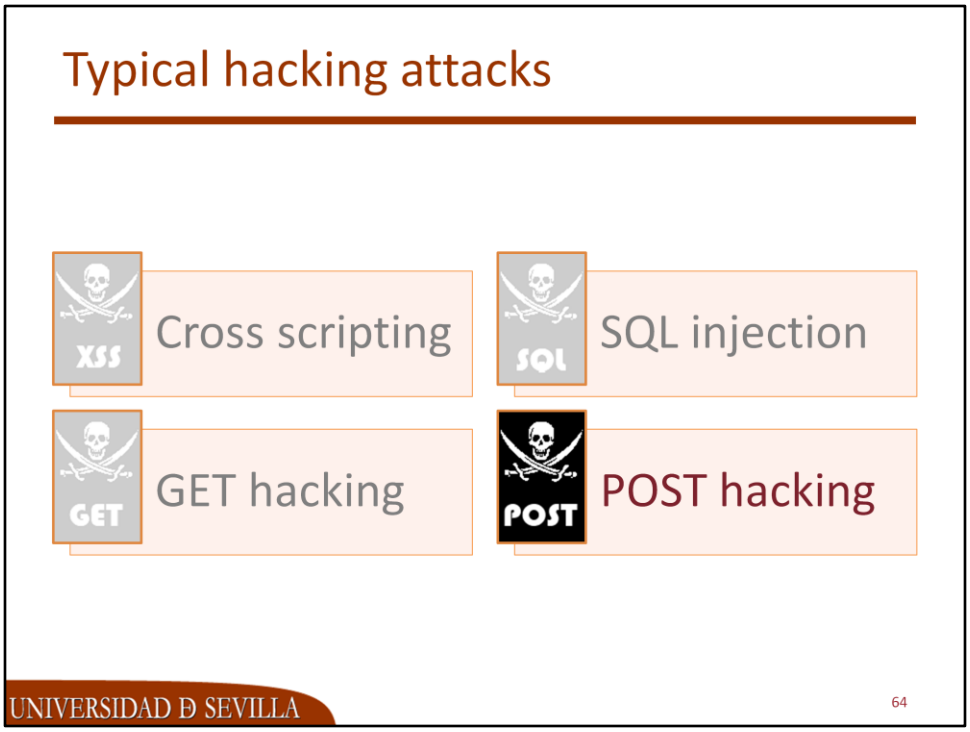
How can you prevent GET Hacking? That's a good question with a pretty simple answer.

This is a good answer!

UNIVERSIDAD Ð SEVILLA

63

First, and foremost important: never forget to check the principal in your services; if a service's intended to be used by, say, an administrator, you must check that the principal's actually an administrator; otherwise, raise an exception and let your system panic. Second, and very important, too: don't forget to configure your "security.xml" file appropriately. These two pieces of advice are very simple to implement, but they may help you prevent a lot of headache.



Finally, let's talk a little about POST hacking. Preventing this kind of hacking's a little more difficult, but not a lot.

What's POST hacking?

It's about posting invalid or malicious data to a URL address

UNIVERSIDAD Ð SEVILLA

65

Simply put, POST hacking's about posting invalid or malicious data to a URL address. Is it similar to GET Hacking? Yes, it is, but it's a little more involved. Preventing this kind of hacking requires to think of the design of your domain model and introducing new kinds of objects in some situations.

Let's examine the JSP edition form

```
<form:form action="registration/customer/edit.do"
           modelAttribute="registration">

    <form:hidden path="id" />
    <form:hidden path="version" />
    <form:hidden path="owner" />
    <form:hidden path="announcement" />
    <form:hidden path="examPaper" />

    <acme:textbox code="registration.moment
                  path="moment" readonly="
    <acme:textbox code="registration.payment.moment"
                  path="payment.moment" />
    <acme:textbox code="registration.payment.amount"
                  path="payment.amount" />

    <acme:submit code="registration.save" name="save" />
    <acme:cancel code="registration.cancel"
                 url="announcement/customer/list.do" />

</form:form>
```

UNIVERSIDAD D SEVILLA

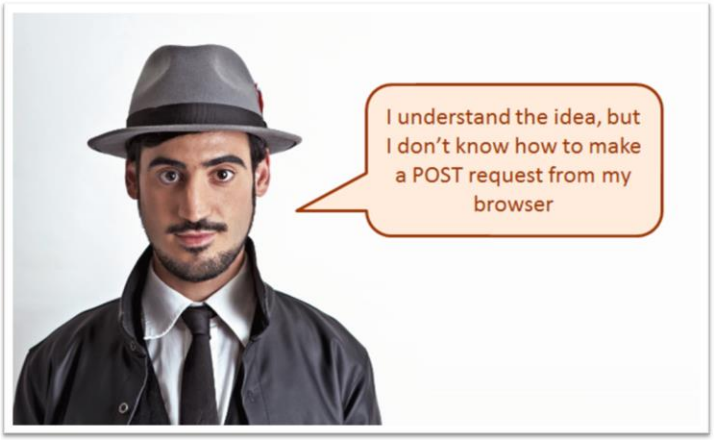
66

Let's now examine a typical JSP form to edit a registration in the context of the Acme Certification project. Basically, a registration basically stores the moment when it was created, but it has a number of role attributes that relate it to a customer, an exam paper, one or more payments, and an announcement. In the previous semester, we taught you that you must serialise every attribute to the form (including role attributes) because, otherwise, the corresponding domain object cannot be fully reconstructed when the form's submitted to the server; we also taught you that attributes that needn't be edited must be serialised using "form:hidden" tags. That's correct, we didn't tell you a wrong word; the problem's that these forms are very easy to hack. Keep reading to learn how.

Let's examine the HTML edition form

This slide shows a piece of HTML that results from the previous JSP form. Note that every "form:hidden" tag results in an "input" HTML tag that contains a value. For instance, there's a hidden input called "owner" that has value "89", a hidden input called "announcement" with another value, and a hidden input called "examPaper" whose value must be "" (please, recall that you have to pay prior to attending an announcement, so you can't have an exam paper when you register). A hacker might easily try the following: what would happen if he or she changed the value of input "owner" to "34"? Would this add a new registration to that customer, whoever he or she is? What would happen if he or she changed the value of input "announcement" to "10"? Would this allow him or her to register to an announcement that isn't active? What if he or she changed the value of the "examPaper" input to "678"? Would this allow him or her to have produced an exam paper, even if he or she didn't attend the announcement? The answer to the previous questions is yes, unless the developers make it impossible.

Don’t you know how?

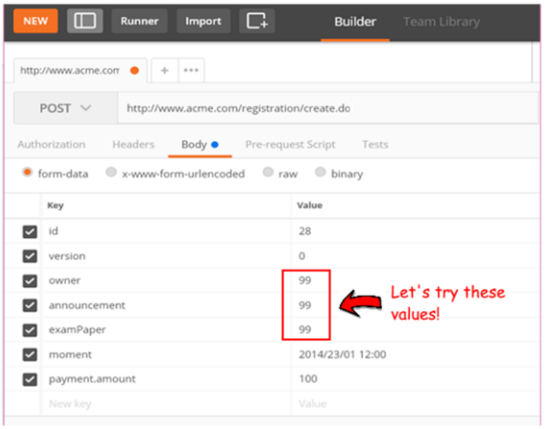


UNIVERSIDAD Ð SEVILLA

68

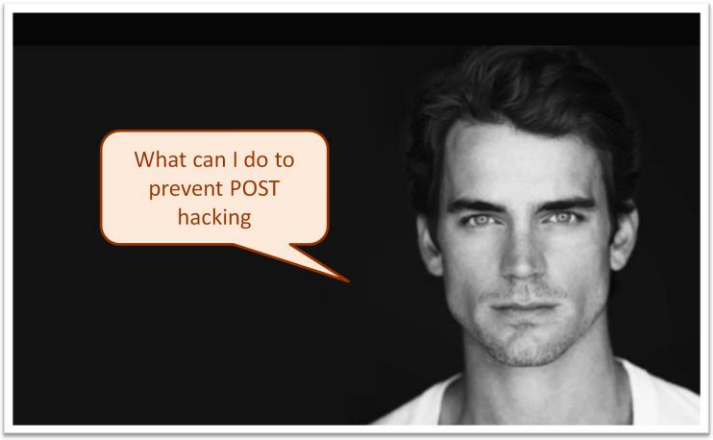
We hope that the previous explanations were enough to get the idea across. If you didn’t understand the problem, please, don’t go ahead. Go through the previous slides again and make sure that you understand this problem before you go on. You might wonder how to make a POST request; we assume that you’re accustomed to GET requests because this is the kind of request that you create whenever you enter a URL at the address bar of your browser. Typical browsers don’t provide a similar means to create a POST request, but there are lots of browser plug-ins and standalone tools around.

Pleased to introduce you to POSTMAN



For instance, try a Chrome plug-in called POSTMAN, which allows to create GET or POST requests very easily. Please, play with it for a little and you'll easily learn to hack your applications… or somebody else's applications.

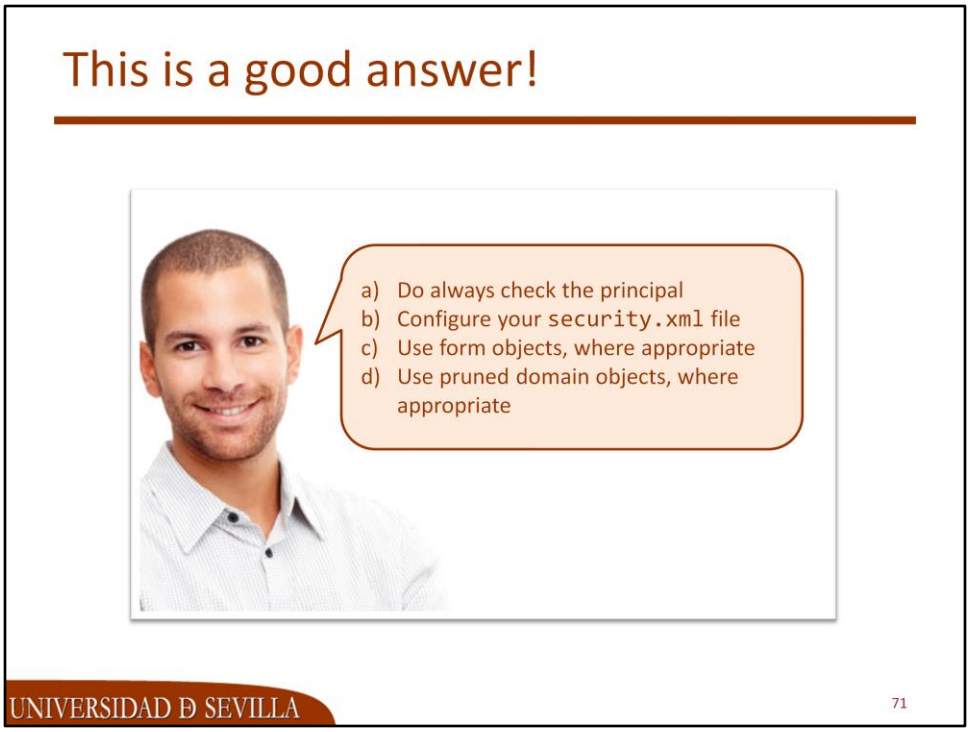
This is a good question!



UNIVERSIDAD Ð SEVILLA

70

This is a good question: what can you do to prevent POST hacking?



And this is a very good answer: a) don't forget to check the principal in your services to make sure that he or she's allowed to edit the objects that they're trying to edit; b) configure your "security.xml" file to prevent unauthorised POST requests; c) use so-called form objects where appropriate; and d) use pruned domain objects where necessary.

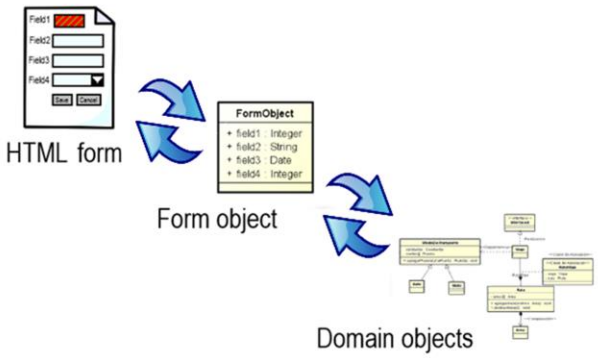
Form object? Pruned domain object?

UNIVERSIDAD Ð SEVILLA

72

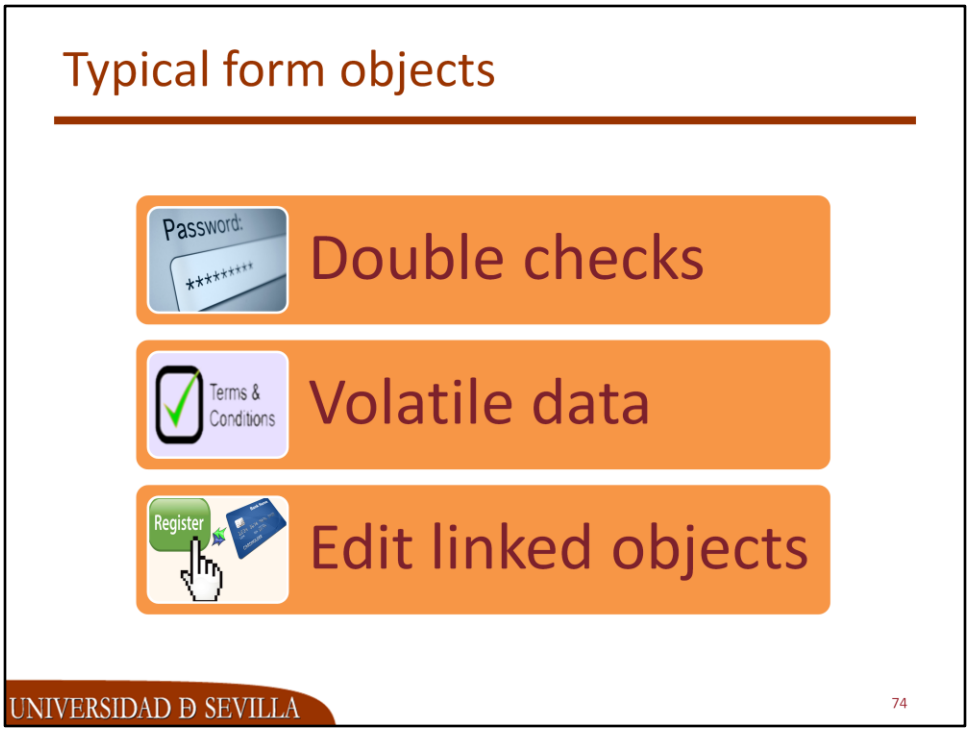
Ok, we know that you've raised an eyebrow regarding the last two items in the previous slide. What's a form object? What's a pruned domain object? Don't worry, these ideas are quite simple. Please, keep reading.

A definition of form object



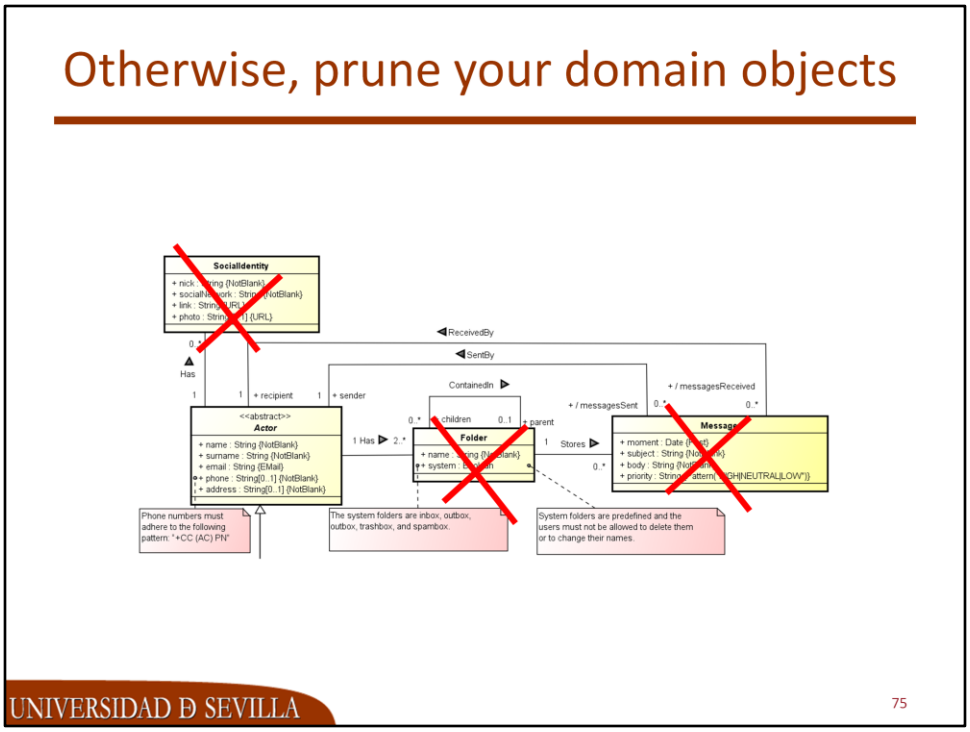
It's an object that provides the exact attributes that are required in an HTML form, independently from the domain objects

Let's start with form objects: a form object is an object that provides the exact attributes that are required in an HTML form, independently from the domain objects. Using form objects helps keep the data that is serialised to the HTML forms to a minimum, which helps prevent POST attacks.

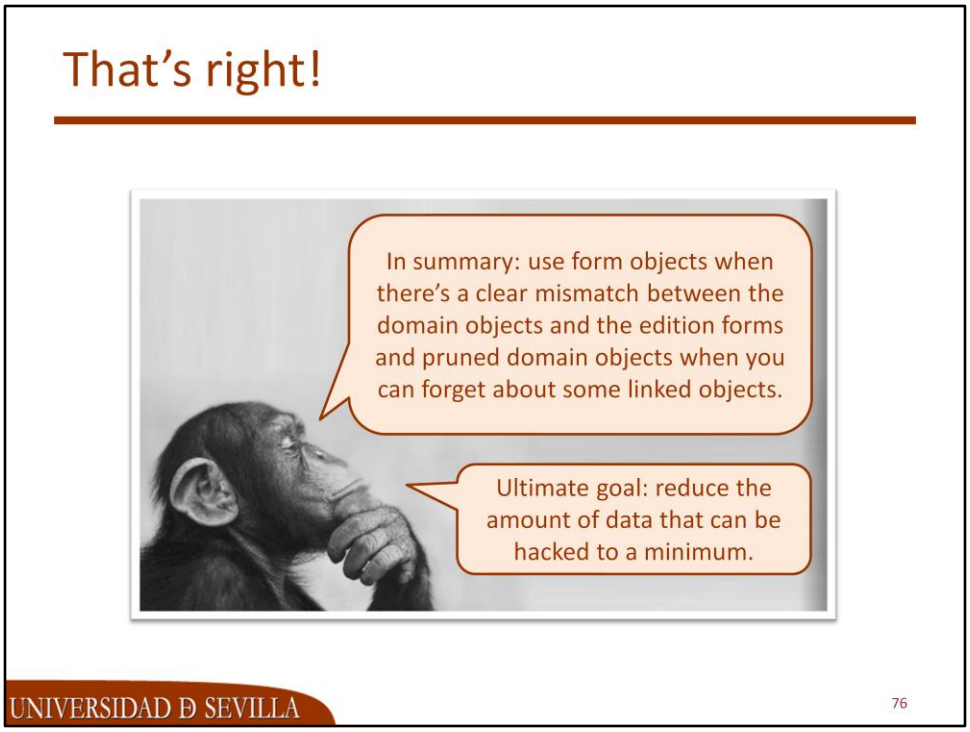


Typical form objects are used in the following situations:

- Double checks: think, for instance, of a typical user registration form in which the user's requested to enter his or her password twice. The corresponding domain object doesn't have two attributes to store the password and the confirmation, but a single attribute to store the hash of the password that the user's entered. In these situations, a form object's the best solution because it can easily store both passwords independently from the domain model.
- Volatile data: for instance, think of a typical check-out form in which the user has to enter some data and then check a box to confirm that he or she agrees with some terms and conditions. What the system's typically requested to store is the date when the user checks out or cancels his or her order. In this case, it's a good idea to use a form object because it may include the checkbox independently from the domain model.
- Edit linked objects: for instance, when a customer registers to an announcement, he or she must provide information about his or her payment; that involves editing two different domain objects in a single form: a "Registration" domain object and a "Payment" domain object. The simplest solution is to create a new form object that has the information to be edited regarding the registration itself and the linked payment.



There might be other cases in which form objects might well help, but not many. Usually, instead of using a form object, you can use a pruned domain object. For instance, think of a situation in which you have to edit an actor, e.g., a customer, a clerk, or an administrator. You know that an actor may be linked to a collection of social identities, folders, and messages; you also know that when you edit an actor, you don't actually need to edit those collections. In such cases, the best idea is to prune the domain object and forget about those collections when editing it. The less information you include in your forms, the less chances to be POST hacked.



In summary: use form objects when there's a clear mismatch between the domain objects and the edition forms and pruned domain objects when you can forget about some linked objects. The ultimate goal: to reduce the amount of data that can be hacked to a minimum.

This is a good question. So far we've used controllers that receive domain objects that are expected to be valid. Using form objects or pruned domain objects has a small impact on the controllers. Let's analyse it.

Impact on controllers (I)

```
@RequestMapping(value="/edit", method=RequestMethod.POST, params="save")
public ModelAndView save(RegistrationForm registrationForm,
                         BindingResult binding) {
    ModelAndView result;
    Registration registration;

    registration = registrationService.reconstruct(registrationForm, binding);
    if (binding.hasErrors()) {
        result = createEditModelAndView(registration);
    } else {
        try {
            registrationService.save(registration);
            result = new ModelAndView("redirect:list.do");
        } catch (Throwable oops) {
            result = createEditModelAndView(registrationForm,
                                            "registration.commit.error");
        }
    }

    return result;
}
```

It gets an object form as parameter, not a domain object.

It reconstructs a registration domain object from the registration form and then checks its validity and updates "binding".

In this slide, we show a snippet of a controller that uses form objects, just to get the idea across. Please, note that it's a simple controller that matches one of the patterns that we taught in lesson "L06 – Controllers". The only difference's that it doesn't get a domain object as a parameter, but a form object, and that it has to use a service method to transform it into the corresponding domain object. It simply fetches the appropriate domain objects using the repositories, then moves the data in the form objects to those domain objects, and, finally, checks the validity of the resulting domain object and accumulates the corresponding errors, if any, in object "binding". After that, the controller behaves normally, as if it had got a "Registration" domain object as parameter.

Impact on controllers (II)

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

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

    return result;
}
```

It gets a pruned domain object, so no validation's possible here.

It reconstructs a customer domain object from the pruned domain object and then checks its validity and updates "binding".

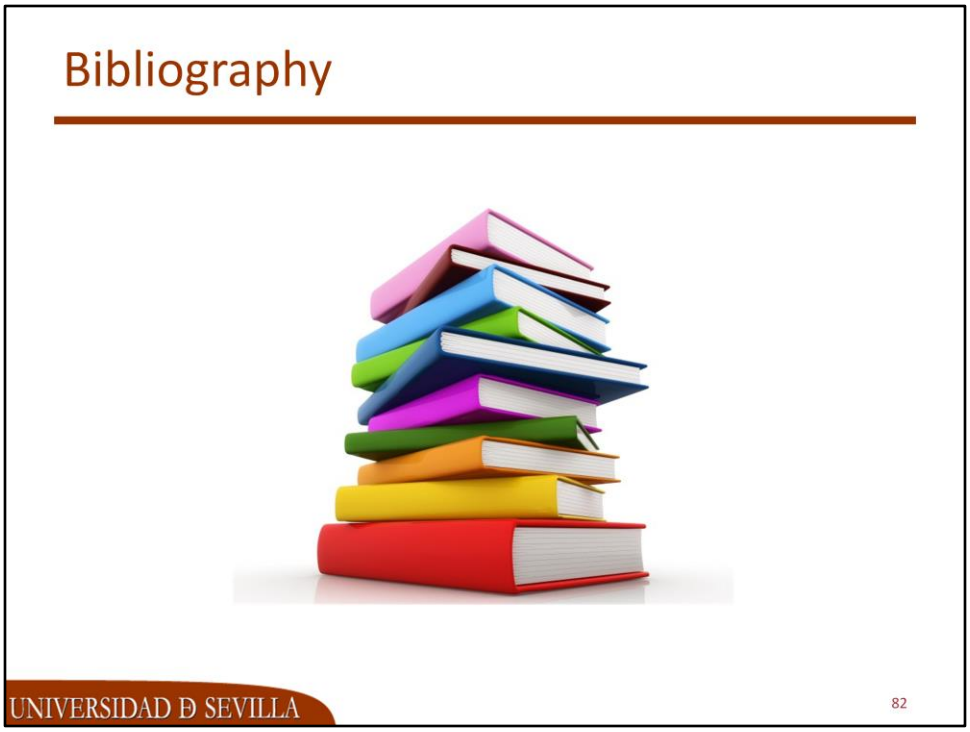
In this slide, we show a snippet of a controller that uses pruned domain objects, just to get the idea across. Assume that this is the method to edit a "Customer" object and that it's been pruned; this means that we can't require it to be valid because it lacks some attributes that have been pruned, e.g., the social identities or the folders and messages. Note that the first sentence in the controller invokes the customer service to reconstruct the object: it must fetch it from the database and update it according to the data that the controller's got; furthermore, it must check the validity of the resulting domain object and accumulate the corresponding errors, if any, in object "binding".

Reconstructing domain objects

```
@Autowired
private Validator validator;

public Customer reconstruct(Customer customer, BindingResult binding) {
    Customer result;

    if (customer.getId() == 0)
        result = customer;
    else {
        result = customerRepository.findOne(customer.getId());

        result.setName(customer.getName());
        result.setEmail(customer.getEmail());
        result.setAddress(customer.getAddress());

        validator.validate(result, binding);
    }

    return result;
}
```


Reconstructing a domain object is fairly simple. In this slide, we show a simple method to reconstruct a customer. It gets two parameters as input: the object to be reconstructed and a binding result to which it'll add validation errors, if any. The method first makes a difference between objects that cannot be reconstructed because they've never been persisted and objects that can be reconstructed. In the former case, the result is the input object; in the latter case, the method uses a repository to retrieve the corresponding domain object from the database; the following sentences update this object with the information that the user's entered in the corresponding edition form, which can be retrieved from the parameter; next, the resulting object's validated. Note that validation's carried out by means of a validator that is introduced as an @Autowired object of type "Validator" (There are several "Validator" classes and interfaces available; choose the one that's provided by Spring). The validator checks the constrains in a domain object and pushes the errors, if any, onto a binding result object. Pretty easy, isn't it?

Oh... one final question: where should you put your form objects? In folder "src/main/java/forms". That folder's been empty so far. It's time to put some classes in there to help your systems to be more robust regarding POST hacking!



We recommend that you should take these lecture notes as your primary source of knowledge. If you wish to go a step further, you can consult the following web references:

- JSP custom tags: http://docs.oracle.com/javaee/5/tutorial/doc/bnalj.html
- Hibernate: http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html_single

We also recommend that you should take a look at the following textbook:

Hacking web apps
Mike Shema
Syngress, 2012
ISBN: 9781597499569

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 lecture! See you next day!