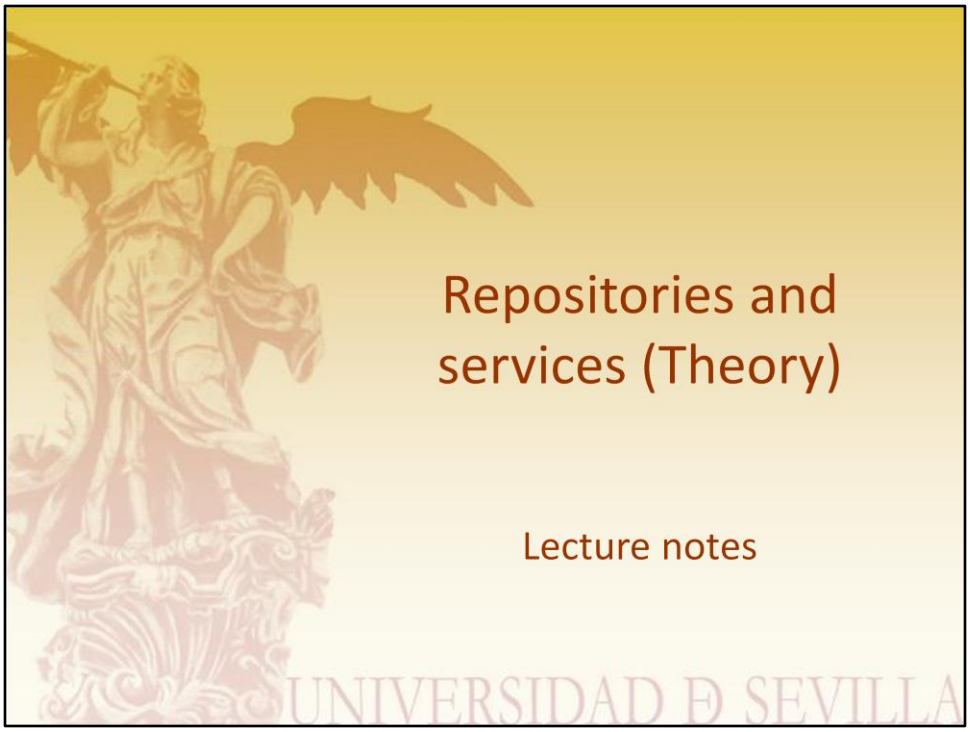


Welcome to this lesson! Today, our goal's to present the theory that you require to work on services and repositories.

--

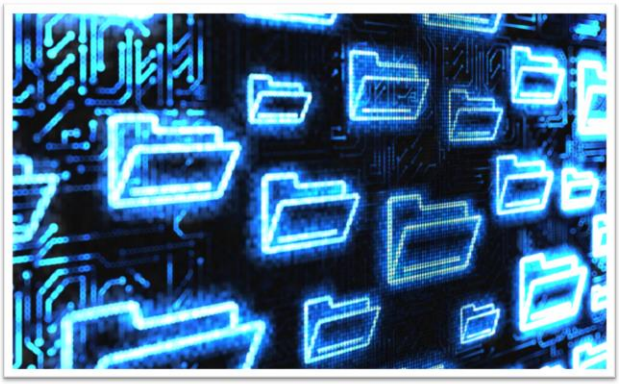
What are they?

UNIVERSIDAD Ð SEVILLA

2

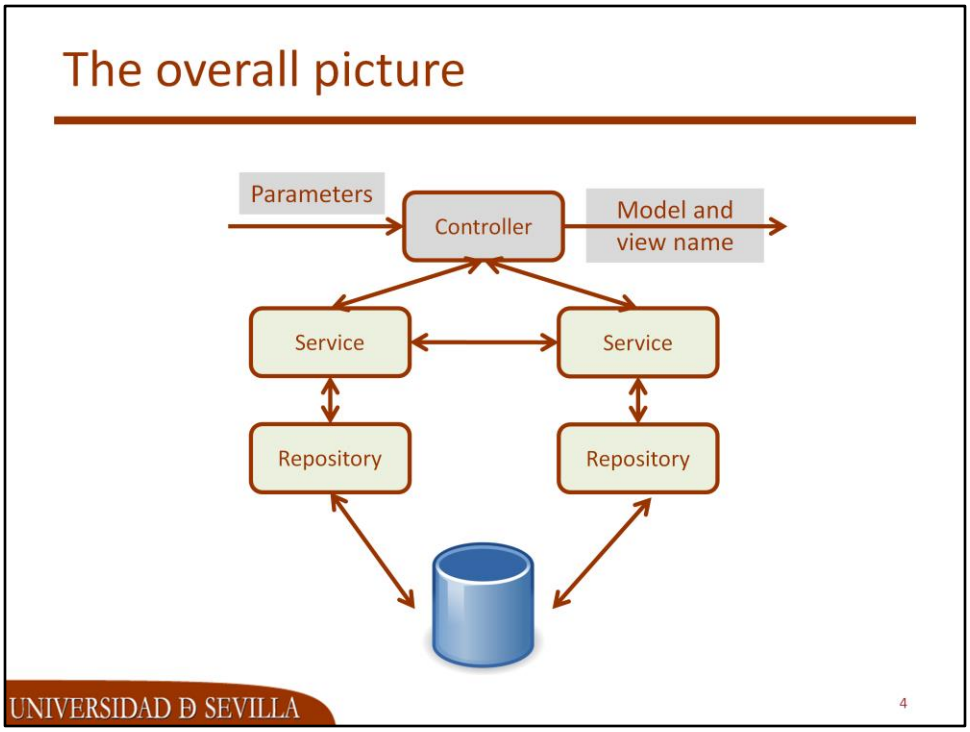
As usual it's a good idea to start asking what repositories and services are about. You should have an intuitive understanding because we introduced them in lesson "L01 – Introduction". Please, try to answer this question before you peek at the following slides.

This is a good definition



Repositories and services are classes that help implement use cases regarding persistence and business logic, respectively

This is a good definition: repositories and services are classes that help implement use cases regarding persistence and business logic, respectively. This is quite an abstract definition, so it makes sense to refine it a little now.



This slide provides an overall picture of a part of our architecture. Please, recall from lesson "L01 – Introduction" that sooner or later every request to a web information system is passed on to a controller, which gets the parameters of the request and is expected to return a model and a view name; the model is a map from variables onto values and the view name is the name of a template that describes how to render a user interface using the model. Controllers rely on services that implement the business logic, possibly with the help of other services; services, in turn, rely on repositories that allow to retrieve and/or persist objects to a database.

How are they devised?

UNIVERSIDAD Ð SEVILLA

5

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

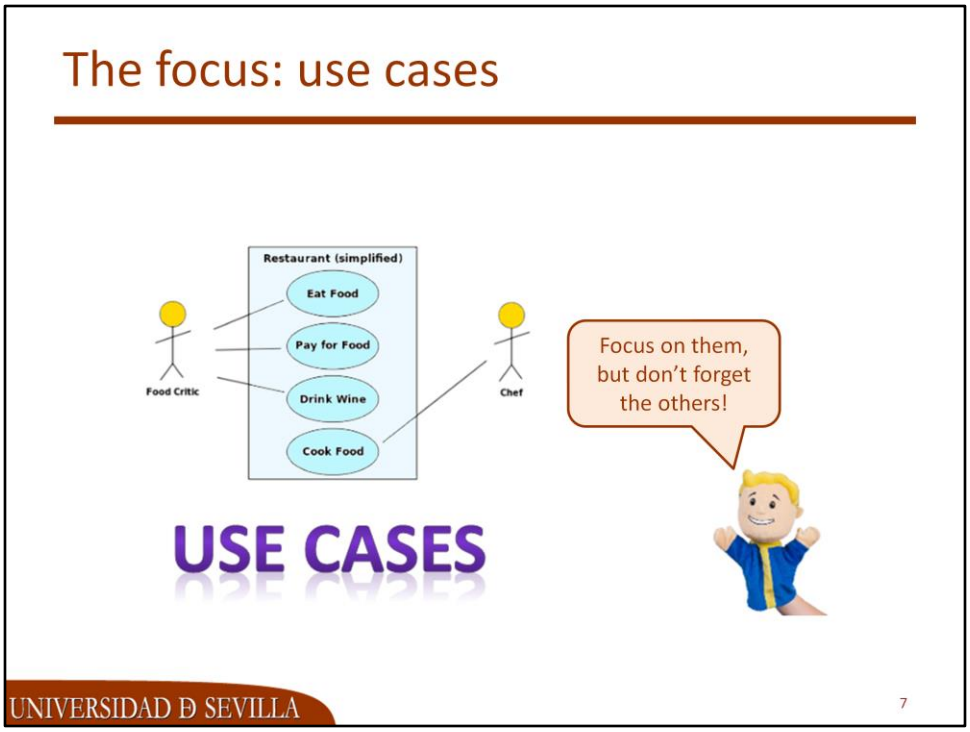
Starting point: your requirements

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

UNIVERSIDAD Ð SEVILLA

6

This shouldn't be surprising to you now: the starting point is your requirements elicitation document. By now, you should be familiar with this document and know that it's composed of three main sections: a description of the non-functional aspects of the system on which we have to work, a description of the data that it has to manipulate, and a description of the use cases.



Our focus to design repositories and services must be on the use cases. But, please, remember that having a focus doesn't mean forgetting about the other sections. Don't forget about the illities and the information requirements since you need a holistic perspective on your requirements if you wish to produce good repositories and services.

Step 1: implement your repositories

UNIVERSIDAD Ð SEVILLA

8

The steps are very simple: first, implement your repositories…

Step 2: implement your services

UNIVERSIDAD Ð SEVILLA

9

… and then implement your services.

Step 3: check them

UNIVERSIDAD Ð SEVILLA

10

And finally you should check that they work as expected, at least, informally.

In the following two sections, we'll provide an insight into both implementing repositories and services. We'll conclude this lecture with a few notes on how you can check that your repositories and services work as expected.

Let’s start with the theory that you need to command repositories.

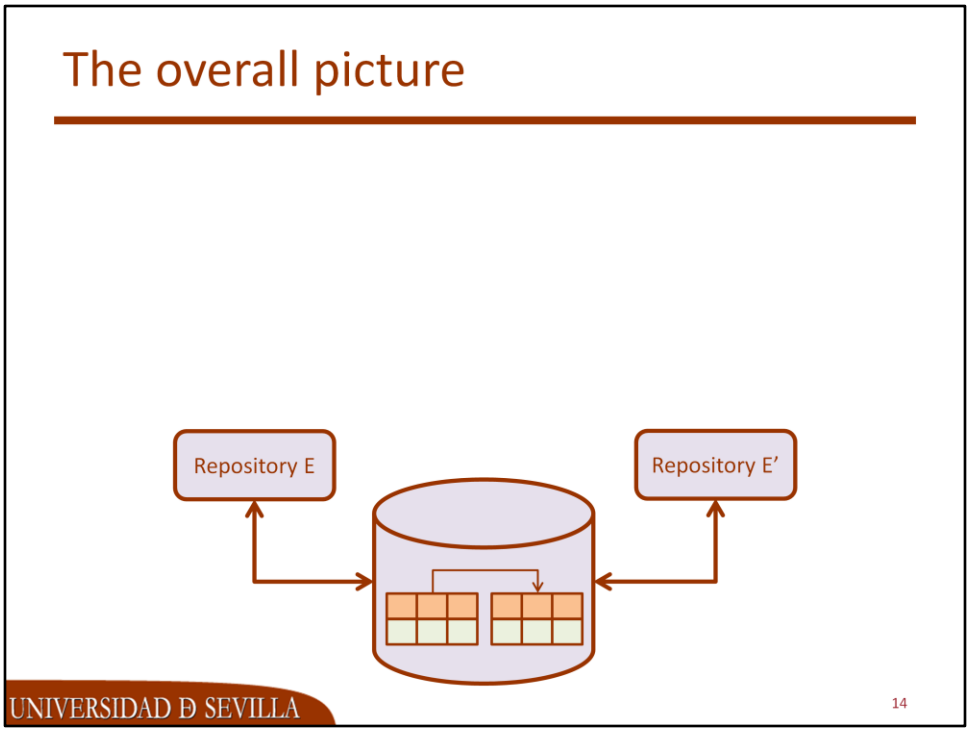
What's a repository?

A repository is a class by means of which we can persist a kind of domain entities to a relational database

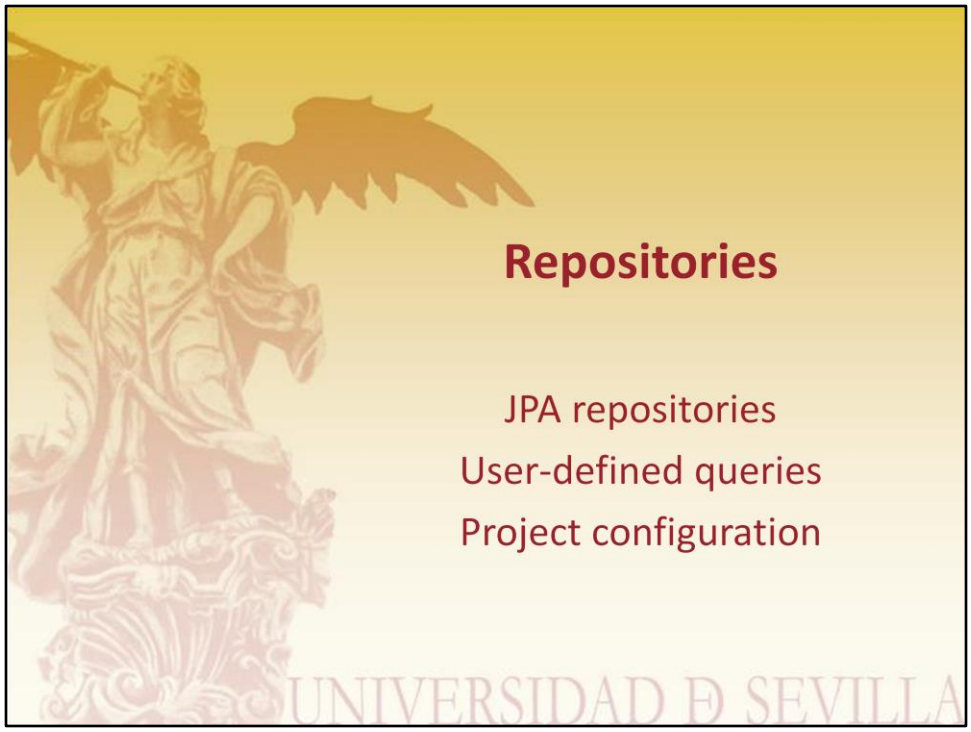
UNIVERSIDAD Đ SEVILLA

13

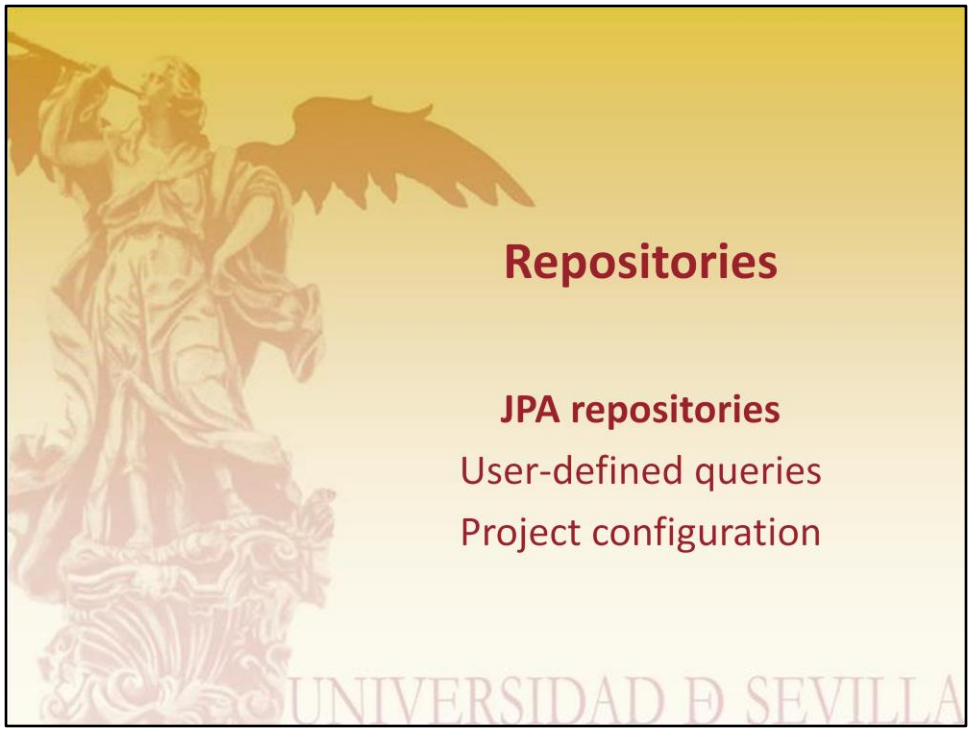
First question: what's a repository? So far you have an intuitive idea, but we'd like to provide a precise definition: a repository is a class by means of which we can persist a kind of domain entities to a relational database. In other words: it's a class that provides a number of methods by means of which we'll be able to retrieve, save, update, or delete entities without worrying about how the corresponding persistence model is implemented; it's the duty of a repository to analyse the persistence model that is attached to an entity and find a way to implement it automatically.



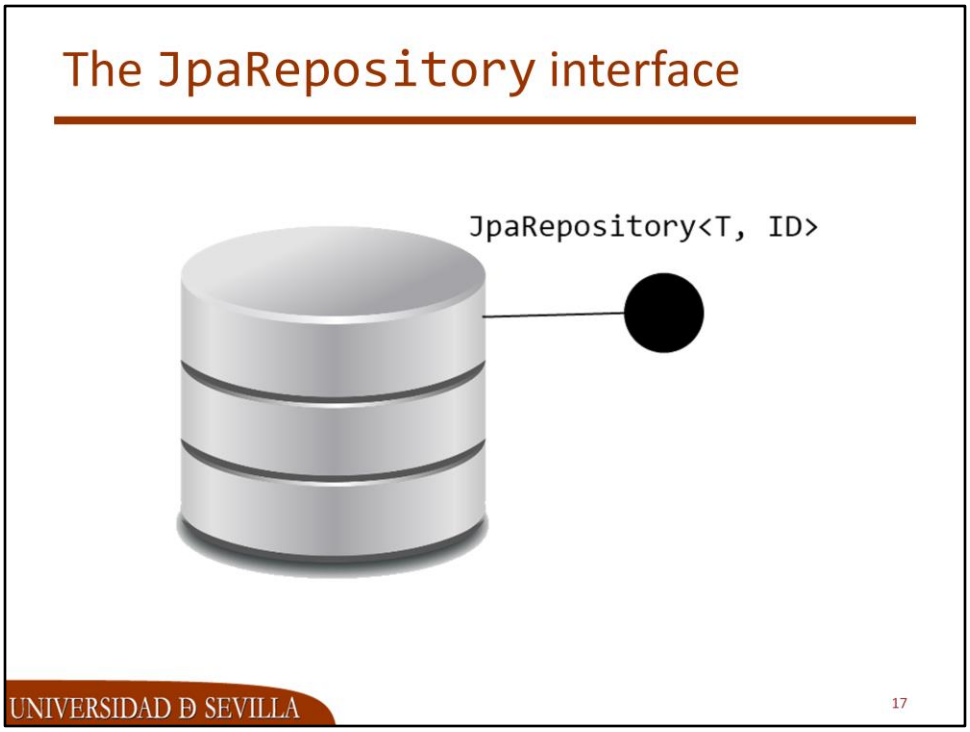
This slide illustrates the idea: there is a database with a schema that you've created and populated thanks to the "PopulateDatabase" utility. The idea is that each entity in our domain model must have an associated repository that provides methods such as find, save, or delete. These methods provide an abstraction layer that allows other layers in our system to work with Java objects all the time, independently from how they are stored in the relational database.



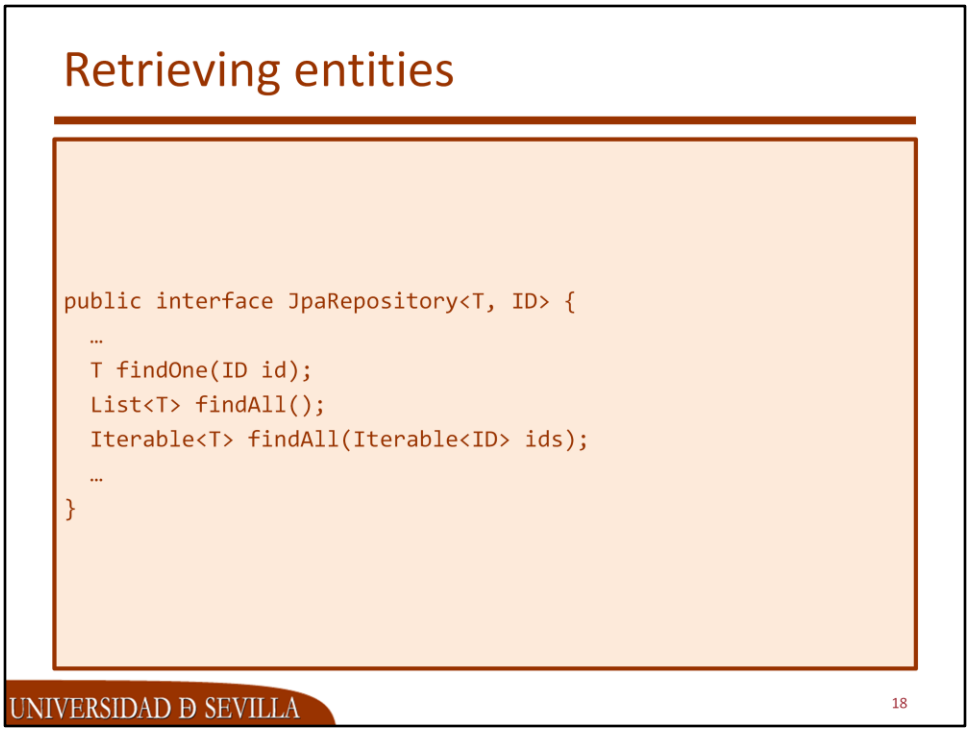
In this section, we'll first report on JPA repositories and then on user-defined queries. We'll also provide a few details on how to configure repositories in our project template.



Let’s start with JPA repositories.

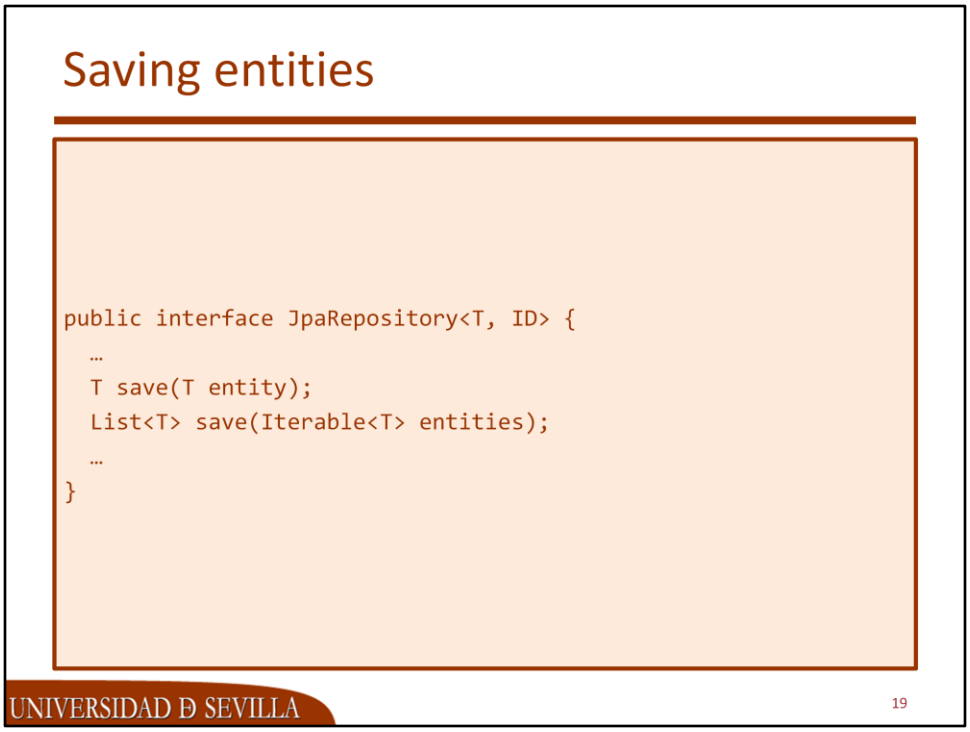


A JPA repository's an interface that extends the Spring "JpaRepository<T, ID>" parametric interface. It provides the methods required to retrieve, save, and delete entities. We'll explore it in the following slides.



Every Spring JPA repository will inherently have the following methods to retrieve entities from a database:

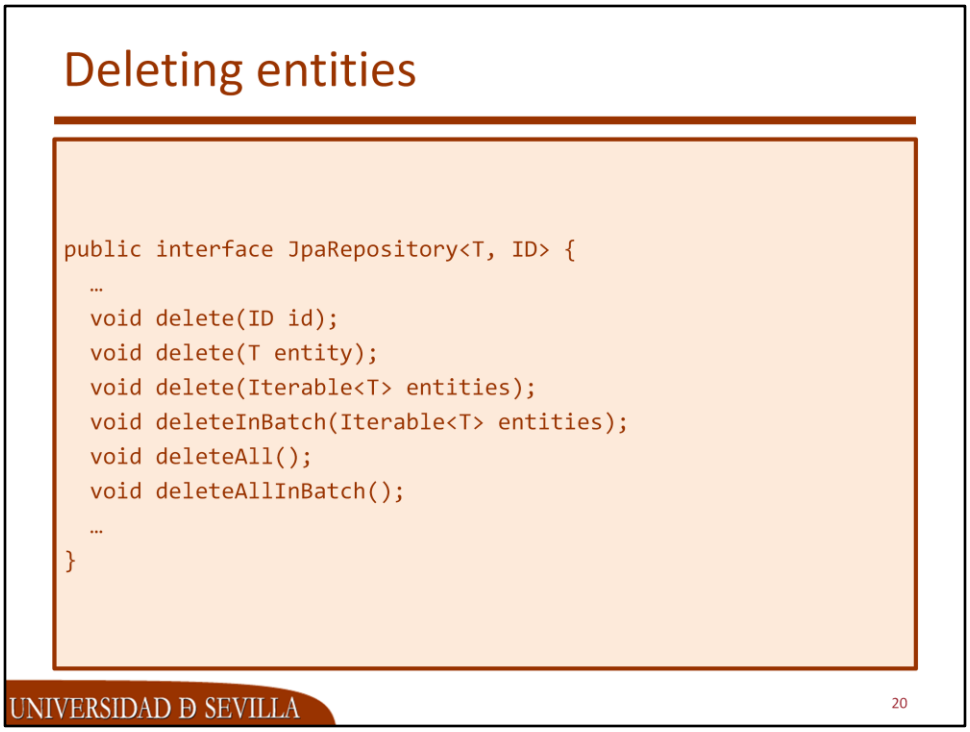
- “findOne”: this method retrieves the entity of type “T” whose identifier is passed as a parameter or “null” if no such entity exists.
- “findAll/1”: this method retrieves a list with all of the entities of type “T”.
- “findAll/2”: this method takes an iterable object with a collection of identifiers, e.g., a list or a set, and returns another iterable object with the corresponding entities of type “T” or “null” values if the corresponding identifier doesn’t correspond to an actual entity.



Saving entities isn't difficult at all with the help of the following methods:

- "save/1": this method takes an entity as a parameter and saves it to the database.
- "save/2": this method takes an iterable object with a list of entities, e.g., a set or a list, and saves them all to the database.

Both methods return the entities that are saved, after JPA instruments them. Instrumenting the entities means that JPA endows them with additional interfaces, e.g., to deal with transactions, changes the implementation of some attributes, e.g., so that collections can be fetched lazily, and a few other changes. You can safely ignore the entities that these methods return unless you wish to know which identifier the database assigns to them. Note that objects that are saved to a database are usually created a few lines before by means of the "new" operator, which creates a plain Java object; that means that their identifier is "0", which means that they've not been saved to the database, yet. It's only when the objects are instrumented and saved that they get a valid identifier. An additional feature of the saving process is that it increments the "version" attribute automatically. Can you remember this attribute? Sure, it counts the number of times an entity's been modified. Later in this lecture, we'll learn what this is intended for.



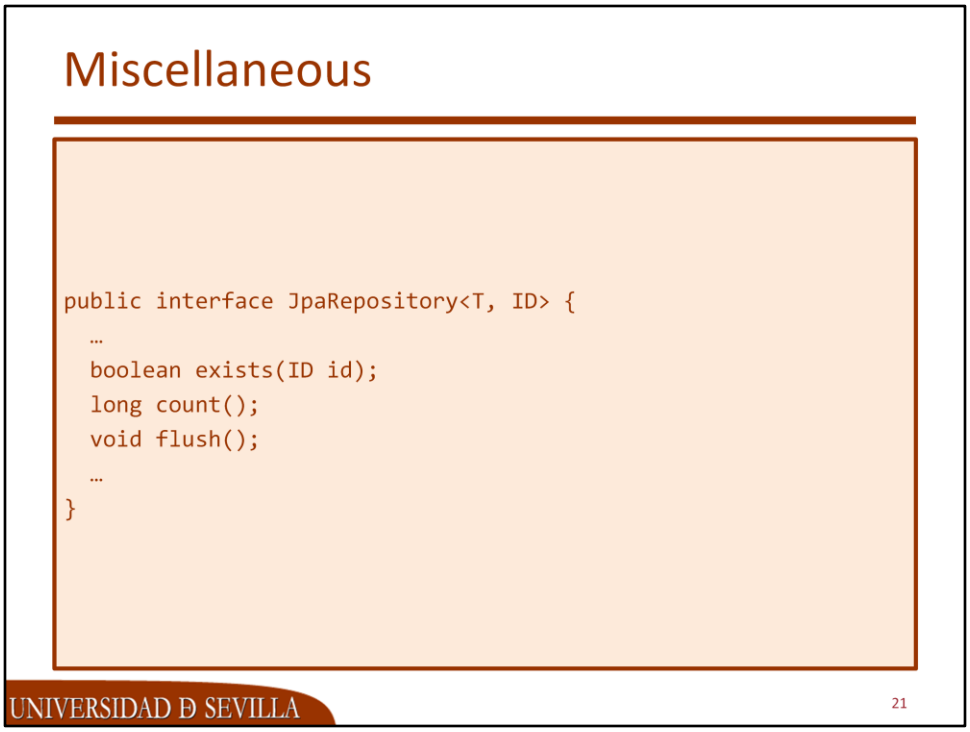
There are a variety of methods to delete entities. The semantics of the first three methods is as follows:

- "delete/1": deletes the entity whose identifier is indicated.
- "delete/2": deletes the entity that is indicated.
- "delete/3": deletes a collection of entities that are stored in an iterable object.

The semantics of the following methods is as follows:

- "deleteInBatch": deletes all of the entities that are passed as a parameter in a single SQL sentence.
- "deleteAll": deletes all of the entities managed by the repository, not necessarily in a single SQL sentence.
- "deleteAllInBatch": deletes all of the entities managed by the repository in a single SQL sentence.

Obviously, you're not very likely to use the "deleteAll" or the "deleteAllInBatch" methods in practice. These methods are very dangerous in a production system!



Before concluding, we'd like to report on three miscellaneous methods:

- "exists": returns "true" if there's an entity in the database with the indicated identifier.
- "count": returns the number of entities of type "T" in the database.
- "flush": flushes the repository.

The last method deserves a little explanation: note that modifications to the database should be always executed in the context of a transaction; otherwise a single problem might render the database incoherent and unusable. In the context of a transaction, changes to the database aren't usually saved immediately, but stored in a memory cache to improve performance; they are saved whenever the transaction is committed or whenever you invoke the "flush" method. You're very unlikely to use this method in practice, but sometimes it may help you debug a complex transaction that fails: invoke "flush" near the instruction that you think might be failing so that the exception is reported as soon as possible, not when the transaction is committed.

How to create your own repository?

UNIVERSIDAD Ð SEVILLA

22

Ok, it's simple, but this is just a Spring interface. We're pretty sure you're wondering how you can create your own repository building on this interface. It's not difficult at all.

Creating a user-defined JPA repository

```
@Repository
public interface CertificationRepository
       extends JpaRepository<Certification, Integer> {

       …

}
```

UNIVERSIDAD Ð SEVILLA

23

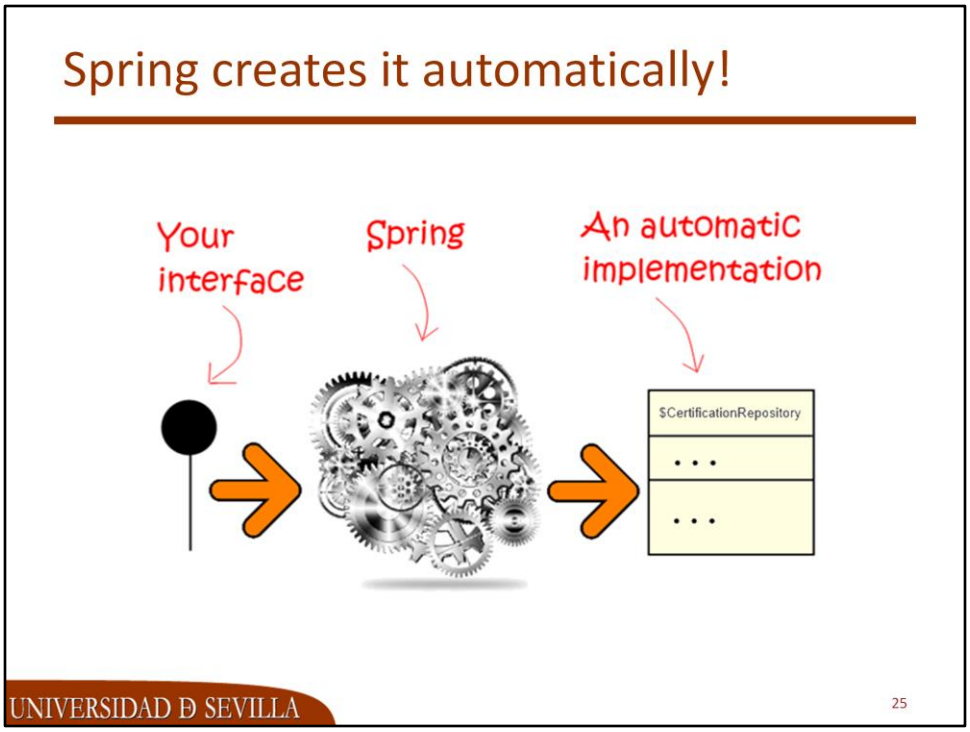
Creating a user-defined JPA repository is very simple: define an interface that extends the “JpaRepository<T, ID>” interface and put an “@Repository” annotation in front of it, where “T” denotes a domain entity class and “ID” the type of its identifier (or the corresponding wrapper type if it’s a primitive type). In the example in this slide, “T” is bound to class “Certification” and “ID” is bound to class “Integer”. That simple! This is enough to create a certification repository.

Where's the implementation?

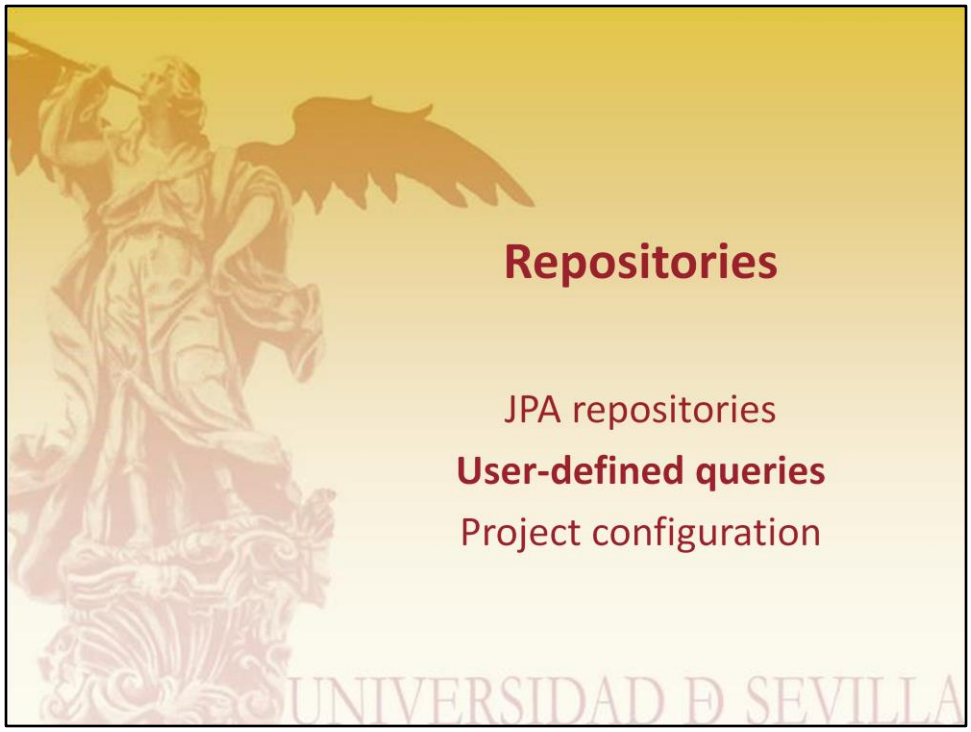
UNIVERSIDAD Ð SEVILLA

24

But you've been told that whenever you have an interface, you need an implementation, right? Where's the implementation of your repository. We have a good piece of news for you!



You don't have to create a class to implement your repositories; it's Spring that analyses your repository interfaces and creates the corresponding classes automatically. These classes are generated at runtime, that is, you won't have a file with their source code.



Let’s now report on how to endow JPA repositories with user-defined queries.

The @Query annotation

```
@Repository
public interface AnnouncementRepository
                 extends JpaRepository<Announcement, Integer> {

  @Query("select distinct r.announcement from Customer c ⇲
         join c.registrations r where c.id = ?1")
  Collection<Announcement> findByCustomerId(int customerId);

  @Query("select a from Announcement a where a.moment > ?1")
  Collection<Announcement> findAllActive(Date currentMoment);

}
```


In this slide, we show a Spring JPA repository for announcements. Realise that we've added the two user-defined queries on which we reported in the previous slide. They are introduced as two regular methods called "findByCustomerId" and "findAllActive"; the difference with respect to other methods is that they have an "@Query" annotation in which the user can specify the JPQL query that must be executed every time the corresponding method is called. Note, too, that parameters are introduced by means of positional variables like "?1", "?2", or "?3" in JPQL; for instance, "?1" refers to parameter "customerId" in the first query and parameter "currentMoment" in the second query.

Sample repositories and queries (I)

```
@Repository
public interface CertificationRepository
       extends JpaRepository<Certification, Integer> {
    @Query("select c from Certification c where c.extinctionDate > ?1")
    Collection<Certification> findAllActive(Date currentMoment);
}

@Repository
public interface CustomerRepository
       extends JpaRepository<Customer, Integer> {
    @Query("select c from Customer c where c.userAccount.id = ?1")
    Customer findByUserAccountId(int userAccountId);
}
```


It's time for a few more examples. Please, try to understand what the user-defined queries in this and the following slides mean:

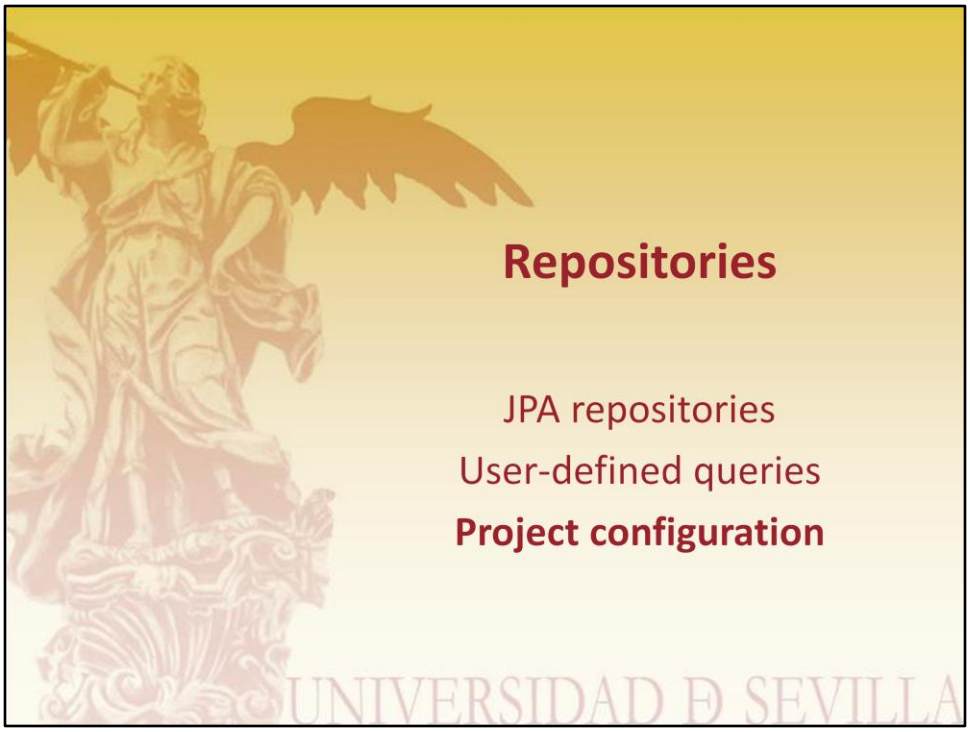
- Returns the collection of certifications whose extinction date is after the date that "currentMoment" specifies; if "currentMoment" is set to the current date and time, then the query returns the collection of active certifications.
- Returns the customer who is associated with the user account whose identifier is indicated.

Sample repositories and queries (II)

```
@Repository
public interface ExamRepository extends JpaRepository<Exam, Integer> {
    @Query("select e from Exam e where e.certification.id = ?1")
    Collection<Exam> findByCertificationId(int certificationId);
}

@Repository
public interface RegistrationRepository
       extends JpaRepository<Registration, Integer> {
    @Query("select r from Registration r where r.announcement.id = ?1")
    Collection<Registration> findByAnnouncementId(int announcementId);

    @Query("select r from Registration r where r.owner.id = ?1 and ↘
            r.announcement.id = ?2")
    Registration findByCustomerIdAndAnnouncementId(int customerId,
                                                   int announcementId);
}
```

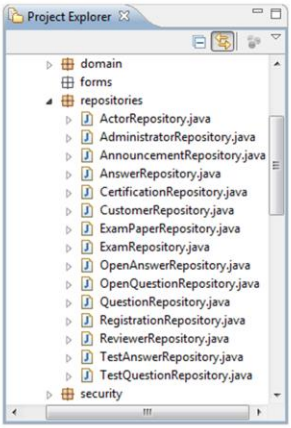

This slide shows a few more examples. Once again: please, try to find out what they mean before reading the explanations below:

- Returns the collection of exams that compose the certification whose identifier is indicated.
- Returns the collection of registrations that involve the announcement whose identifier is indicated.
- Returns the registration for the customer and announcement whose identifiers are indicated. (Note that this query requires two parameters that are specified as “?1” and “?2”.) If no such a registration exists, then “null” is returned.



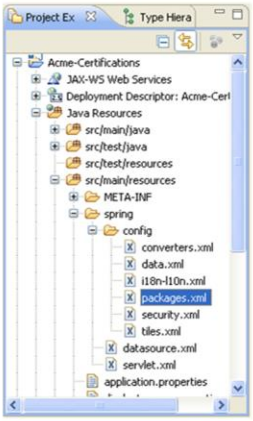
We think that the overall idea is simple and that you won't have any problems to create your own repositories. Before concluding, however, we must provide a few details on how the project template must be configured so that you can work with your own repositories.

The repositories package



The first thing you must know is that you must add your repositories to a package called “repositories”. That’s enough to use them, but you need to know additional configuration details. We’ll report on them in the following slides.

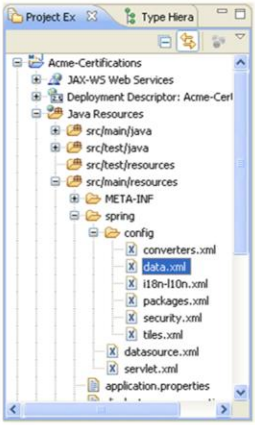
The `packages.xml` configuration file



The first file you must look at is "packages.xml". This file specifies which packages Spring must scan for repositories and other components: controllers, converters, domain entities, forms (which we haven't used thus far), security, and services. Just learn that we need to list the names of the packages where our components reside. If you use our project template verbatim, you won't have to change anything in this configuration file.

This is the first section of the file. Here you must use the “component-scan” element to inform Spring about the names of the packages that provide the repositories; in our template, these packages are “repositories” and “security”. You need to care of the repositories in the “repositories” package only, since the repositories in the “security” package are a part of our framework; you don’t need to modify them.

The `data.xml` configuration file



The “data.xml” configuration file’s another key file regarding repositories. This file’s a lot more complex than the previous one, but we need to command it. So, we’ll delve into its intricacies in the following slides.

Configuring `data.xml` (I)

```
<jpa:repositories base-package="repositories" />
<jpa:repositories base-package="security" />
```


Note that you have to configure a couple of packages here. Yes, we did configure them in “packages.xml”, too. Unfortunately, our framework has some inconsistencies that are difficult to grasp, but we’ve got to get accustomed to them.

Configuring `data.xml` (II)

```
<bean id="dataSource"
      class="com.mchange.v2.c3p0.ComboPooledDataSource"
      destroy-method="close">
    <property name="driverClass" value="com.mysql.jdbc.Driver" />
    <property name="jdbcUrl"
          value="jdbc:mysql://localhost:3306/Acme-Certifications"/>
    <property name="user" value="acme-user" />
    <property name="password" value="ACME-Us3r-P@ssw0rd" />
</bean>
```


Next, we have to configure a data source, which is an object that specialises in handling multiple concurrent connections. In this subject, we'll use C3P0, which is one of the best data source implementations available. The configuration's very simple: just provide a driver class, a connection URL, a user and a password, and it'll be able to manage thousands of simultaneous connections. That simple!

Configuring data.xml (III)

```
<bean id="persistenceUnit" class="java.lang.String">
    <constructor-arg value="Acme-Certifications" />
</bean>

<bean id="sqlDialect" class="java.lang.String">
    <constructor-arg value="org.hibernate.dialect.MySQLDialect" />
</bean>

<util:properties id="jpaProperties">
    <prop key="hibernate.show_sql">false</prop>
    <prop key="hibernate.format_sql">true</prop>
    <prop key="hibernate.hbm2ddl.auto">verify</prop>
    <prop key="hibernate.cglib.use_reflection_optimizer">true</prop>
</util:properties>
```


Finally, you need to configure some JPA properties, namely:

- “persistenceUnit”: this is the name of the persistence unit you defined in file “persistence.xml”.
- “sqlDialect”: this indicates the SQL dialect with which JPA must work.
- “hibernate.show_sql”: this instructs JPA to print out the SQL statements that it sends to the database.
- “hibernate.format_sql”: this instructs JPA to format the SQL statements that it prints out.
- “hibernate.hbm2ddl.auto”: this instructs JPA to verify that the database is OK everytime that an application that uses the data source is started.
- “hibernate.cglib.use_reflection_optimizer”: this instructs JPA to perform additional optimisations. There’s no reason to deactivate this option.

The section on the repositories was quite interesting, wasn't it? If you command JPQL, writing a repository is quite a simple task. Let's now delve into the services. Neither are they too difficult.

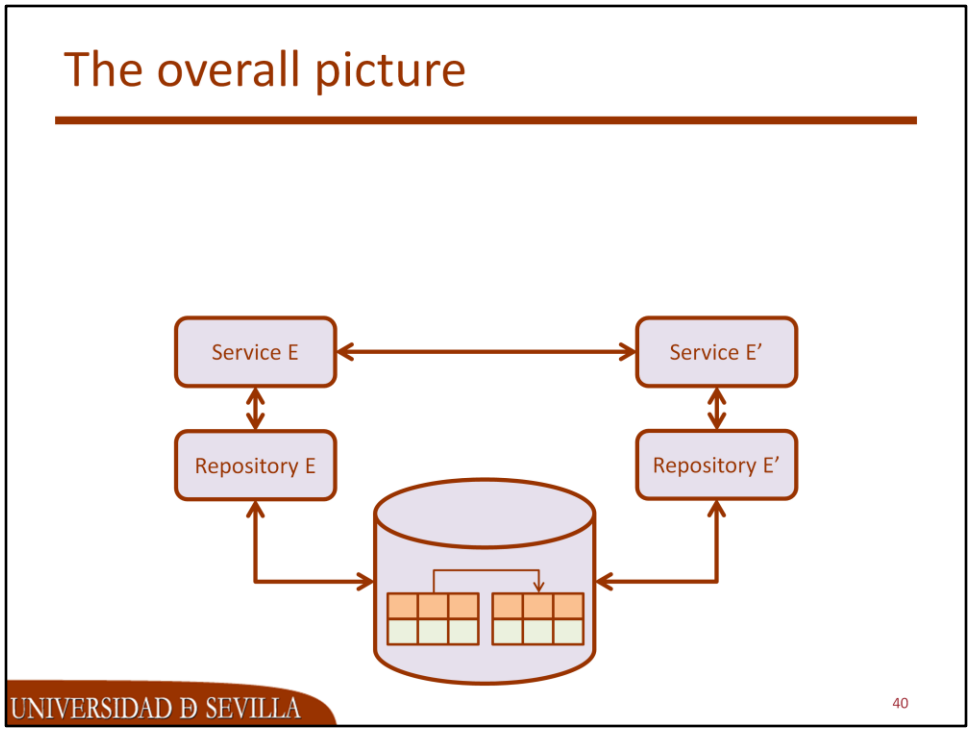
What's a service?

A service is a class that implements business logic and business rules regarding an entity

UNIVERSIDAD D SEVILLA

39

As usual, we'll start the section with a precise definition: simply put, a service is a class that implements business logic and business rules regarding an entity. Business logic refers to algorithms that are intended to manipulate data, e.g., listing customers, updating certifications, or computing exam marks and scores. A business rule is a high-level constraint, e.g., a customer cannot delete the registrations of another customer or an announcement may not be deleted if a customer has registered to it, or a high-level action, e.g., a customer who has registered to more than 10 announcements gets a discount of 10% or the price of a certification that is awarded more than 100 times is increased by 5% automatically.



This slide helps complete the overall picture that we presented before. The idea is that we should have a service for each entity in our domain model, so that each service manages its corresponding repository and uses other services. In general, it's not a good idea that a service manages several repositories since a repository is a thin layer of abstraction that simply cares of persisting entities; it's the service that is associated with an entity that enforces business rules. If a service uses the repository of another entity, it might easily break the business rules that are enforced by that entity's service. This idea should be clear, isn't it?

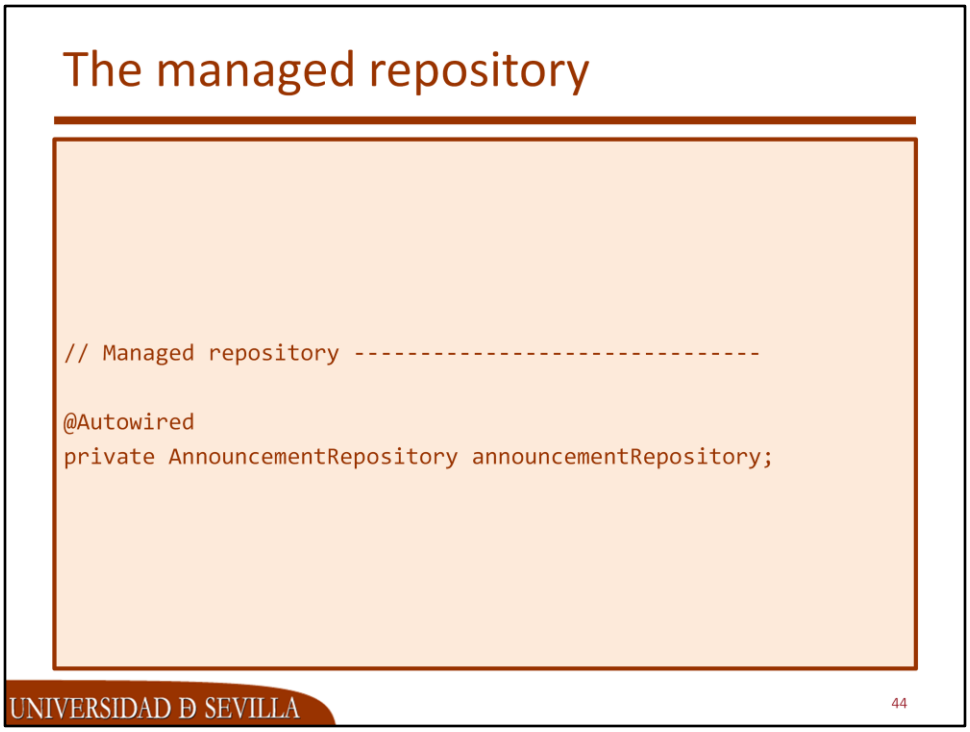
In this section, we'll talk about the structure of our services, about implementing business rules, and we'll also report on the configuration that services require in our project template.

Let’s start with the structure.

The class

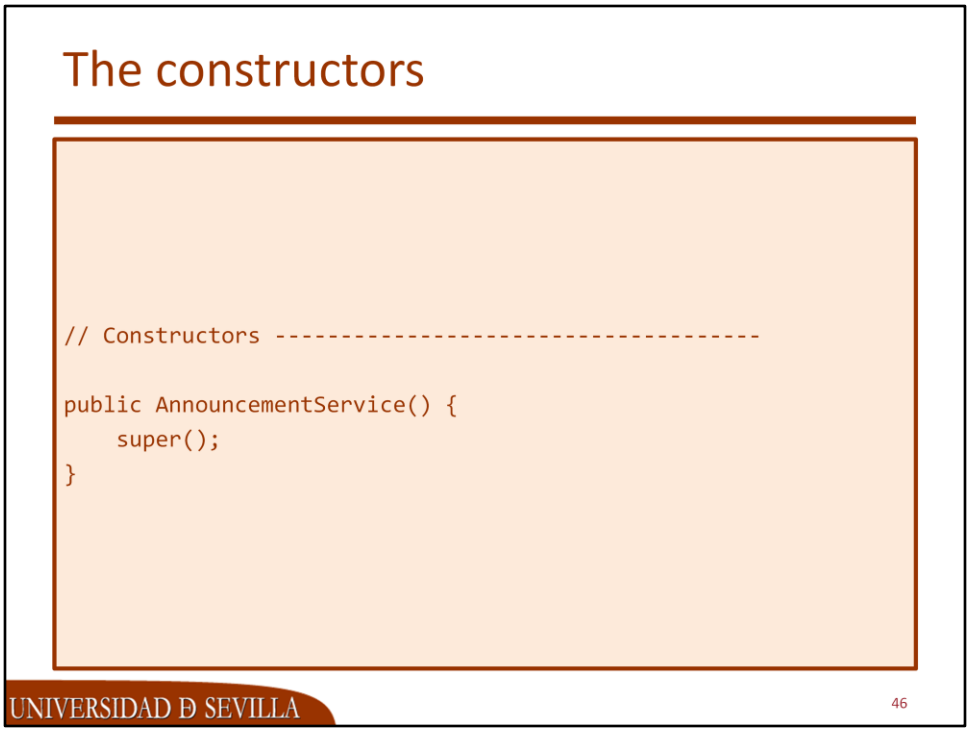
```
@Service
@Transactional
public class AnnouncementService {
        …
}
```


A service's implemented by means of a regular class to which we add annotation "@Service". (More on the "@Transactional" annotation later.)



The first section in a service class is about the managed repository. A few slides ago we mentioned that every service manages a repository. That repository is declared as a private attribute with an "@Autowired" annotation in front of it. This annotation instructs Spring to inject an instance of interface "AnnouncementRepository" into attribute "announcementRepository". Please, recall that you don't have to create an implementation for this interface; it's Spring that creates it and instantiates it whenever necessary.

After the managed repository comes the supporting services. You already know that a service manages a repository and relies on some services that manage other repositories. Such services are called supporting services, and they are introduced as depicted in this slide. Service "AnnouncementService" requires two additional services: "CustomerService" and "RegistrationService"; that shouldn't be surprising since, for instance, to display the announcements to which a customer has registered you need the customer service to return who is using the application; similarly, before removing an announcement, we need to check that it doesn't have any registrations, which is a piece of information that is returned by the registration service. Note that supporting services are introduced very similarly to the managed repository: as private attributes of the appropriate types that have an "@Autowired" annotation in front of their declarations. Again, whenever an instance of class "AnnoucementService" is created, Spring injects the corresponding instances of the supporting services.



Then come the constructors of the service. Although, in theory, you might have more than one constructor, in practice it's not generally necessary to write any constructors. If you wish, you may add a parameter-less constructor for debugging purposes, but you're not likely to leverage the effort.

WARNING: please, note that the constructor's not the following:

```
public AnnouncementService() {
   super();
   this.announcementRepository = new AnnouncementRepositoryImpl();
   this.customerService = new CustomerService();
   this.registrationService = new RegistrationService();
}
```

You don't have to create any instances of your repositories or supporting services; as we told you before, it's Spring that creates the instances and injects them into your services. That simple!

Simple CRUD methods

```
// Simple CRUD methods ------------------------------

public Announcement create() { … }
public Collection<Announcement> findAll() { … }
public Announcement findOne(int announcementId) { … }
public Announcement save(Announcement announcement) { … }
public void delete(Announcement announcement) { … }
```


In this section, we should introduce a number of methods that resemble the simple CRUD methods in the repository, the difference being that they must enforce business rules. For instance, method “delete” in class “AnnouncementService” should enforce that no announcement can be deleted if there’s a registration that involves that announcement.

Other business methods

```
// Other business methods --------------------------

public Collection<Announcement> findRegistered() { … }
public Collection<Announcement> findAllActive() { … }
```


The final section is entitled “other business methods”, which is a term that refers to methods that are not simple CRUD methods. For instance, regarding the announcement service, we’ve added a couple of methods to find the collection of announcements in which the user who is using the application has registered and to find all active announcements, that is, the announcements that haven’t expired.

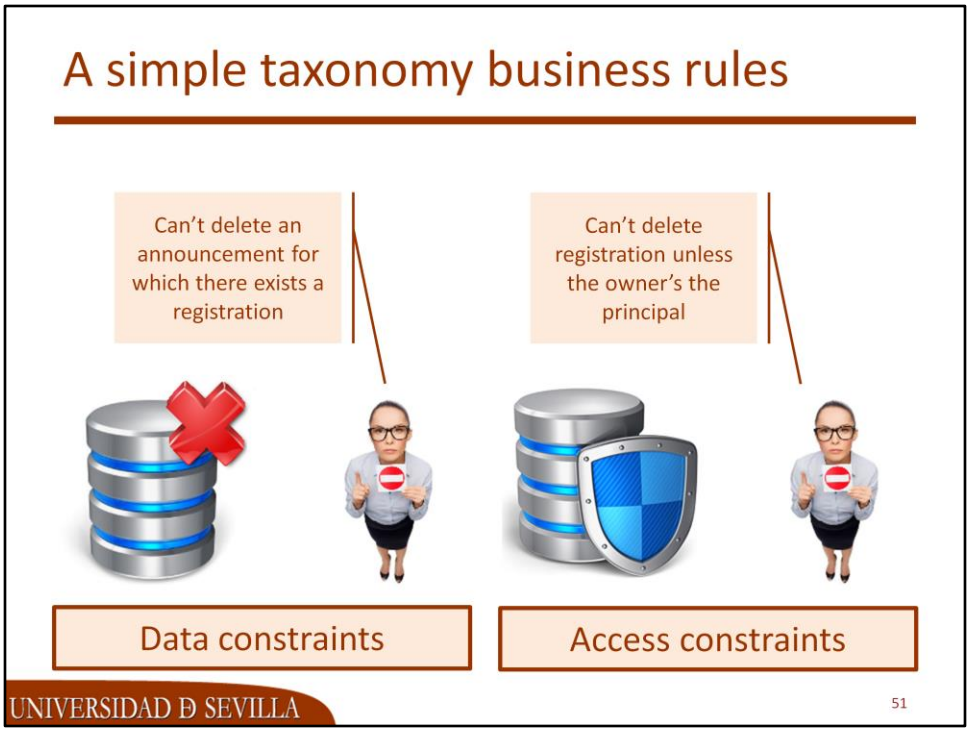
The structure wasn't difficult, was it? Let's now provide a few additional details on business rules.

A simple definition

A business rule is a constraint that must be met so that a service can be executed

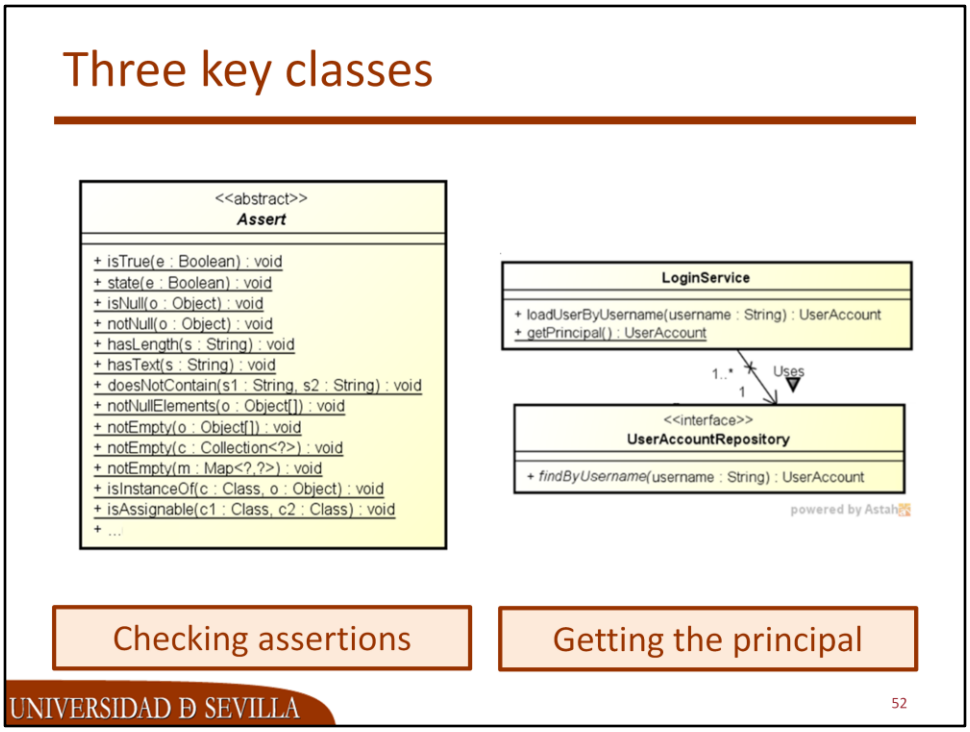
UNIVERSIDAD Ð SEVILLA

This slide provides a simple definition: a business rule is a constraint that must be met so that a service can be executed.



Business rules can be classified as data constraints or access constraints. The former prevent the data from becoming inconsistent and the second prevent unauthorised users to have access to them.

NOTE: realise that it's very common to use term "principal" to refer to the user who is currently logged in an information system and is interacting with it.



To implement business rules, the framework provides three key classes, namely:

- "Assert", which provides a collection of methods that allow to check assertions. The most commonly used is method "isTrue", which gets a Boolean expression as input and does nothing if the expression is true (i.e., the assertion is passed), but raises an exception if the expression is false (i.e., the assertion fails).
- "LoginService", which provides a couple of methods to fetch a user account building on a username, and retrieve the user account of the principal.
- "UserAccountRepository", which is a JPA repository that allows to persist user accounts in your database.

Let's see a couple of examples to learn how to use these classes to implement some typical business rules.

NOTE: there are several "Assert" classes in the framework. Hereinafter, we use the one provided by package "org.springframework.util".

Implementing data constraints

```
@Service
@Transactional
public class AnnouncementService {
  …
  public void delete(Announcement announcement) {
    Assert.isTrue(!registrationService.
              existsRegistrationForAnnouncement(announcement));

    announcementRepository.delete(announcement);
  }
  …
}
```


In this slide, we show how to implement a typical data constraint in method “delete” in service “AnnouncementService”. The business rule checks that the registration service doesn’t report on the existence of any registrations for the given announcement. If the business rule isn’t satisfied, then an exception is thrown and the execution of the method is aborted; otherwise, the “delete” method is invoked on the announcement repository and the announcement is deleted. Recall that services are transactional thanks to annotation “@Transactional”, so every previous operation on the database is roll-backed automatically if an exception happens.

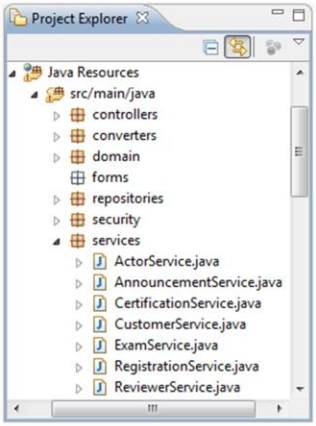
Implementing access constraints

```
@Service
@Transactional
public class RegistrationService {
  …
  public void delete(Registration registration) {
    UserAccount userAccount;

    userAccount = LoginService.getPrincipal();
    Assert.isTrue(
        registration.getOwner().getUserAccount().equals(userAccount));

    registrationRepository.delete(registration);
  }
  …
}
```


This slide shows how to implement a typical access constraint in method “delete” in service “RegistrationService”. The business rule first retrieves the user account of the principal and then checks that it is the same as the user account of the owner of the registration to be deleted. If the previous check is passed, then the call is forwarded to the registration repository; otherwise, an exception is thrown and the corresponding transaction is roll-backed.

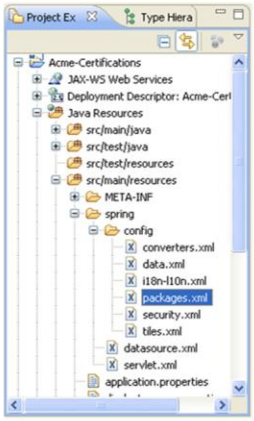
Finally, let's report on the configuration you have to perform in our project template.

The services package



The first thing you must know is that you must add your services to a package called "services" in your project template.

The `packages.xml` configuration file



The only configuration file involved in configuring services is “packages.xml”. This should be familiar to you, now. This file specifies which packages Spring must scan for services and other components: controllers, converters, domain entities, forms (which we haven’t used thus far), repositories, and security.

Configuring packages.xml

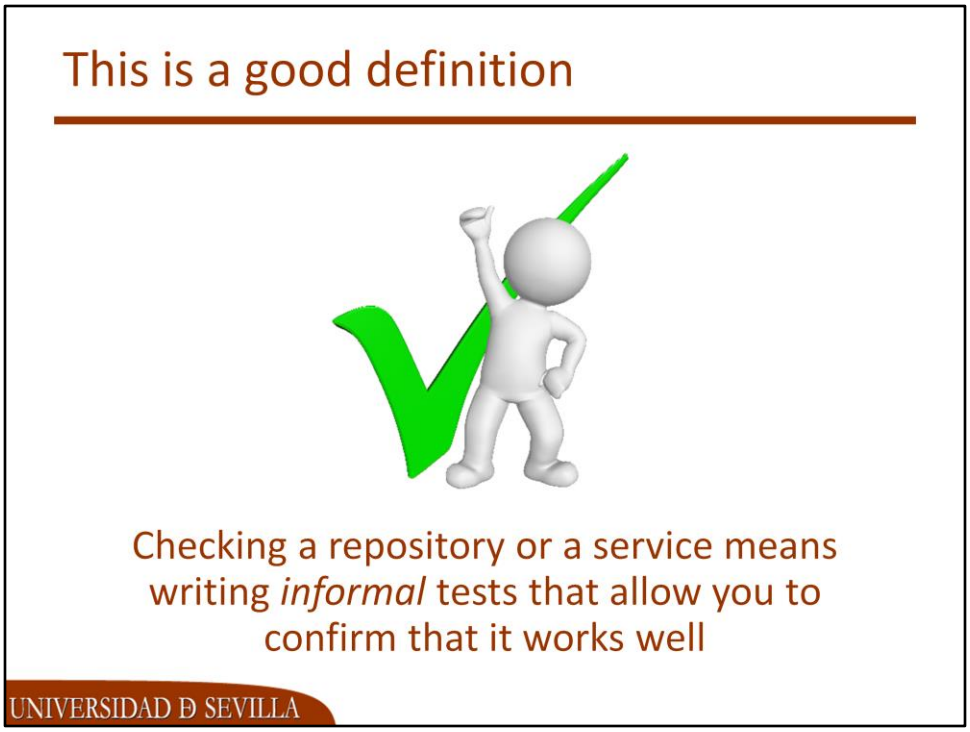
```
<context:component-scan base-package="services" />
<context:component-scan base-package="security" />¡
```

UNIVERSIDAD Ð SEVILLA

58

This is the first section of the file. Here you must use the “component-scan” element to inform Spring about the names of the packages that provide the services; in our template, these packages are “services” and “security”. You need to care of the repositories in the “repositories” package only, since the repositories in the “security” package are a part of our framework; you don’t need to modify them.

It’s now time to learn a little on how to check repositories and services.



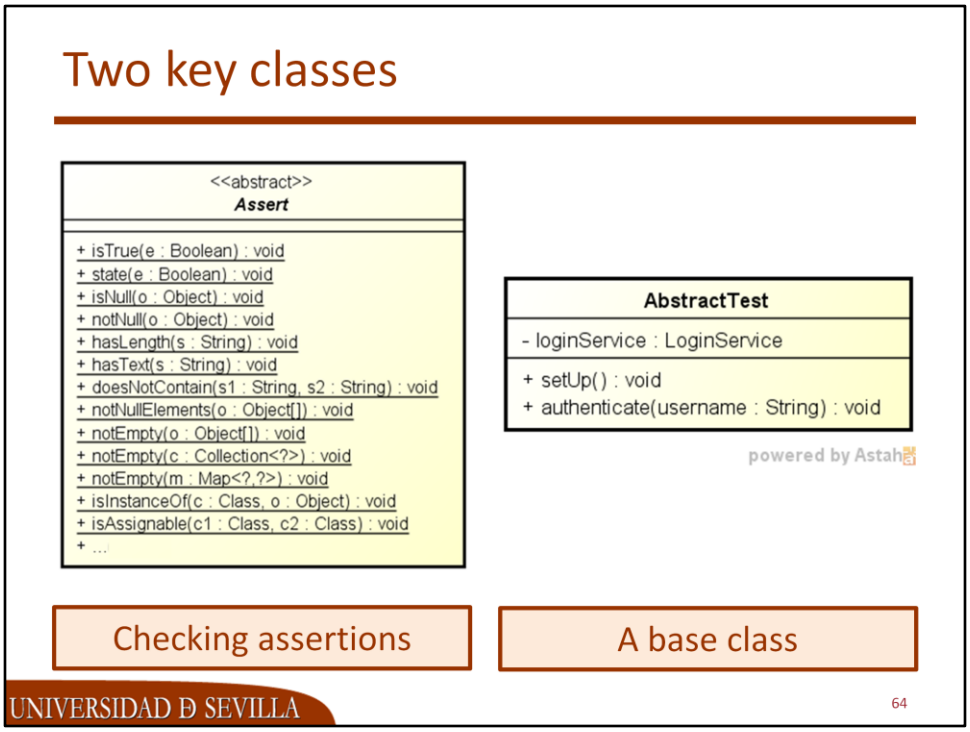
Checking a repository or a service means writing *informal* tests that allow you to confirm that it works well. Note that we emphasise informal because we're going to learn a few foundations that allow us to write simple tests without a clear guideline. Our only purpose is that you can execute the code in your repositories or your services to get an impression on how they work. We'll explore formal testing during the second semester.

The tool to implement them

To implement our checks, we're going to use a tool called JUnit.

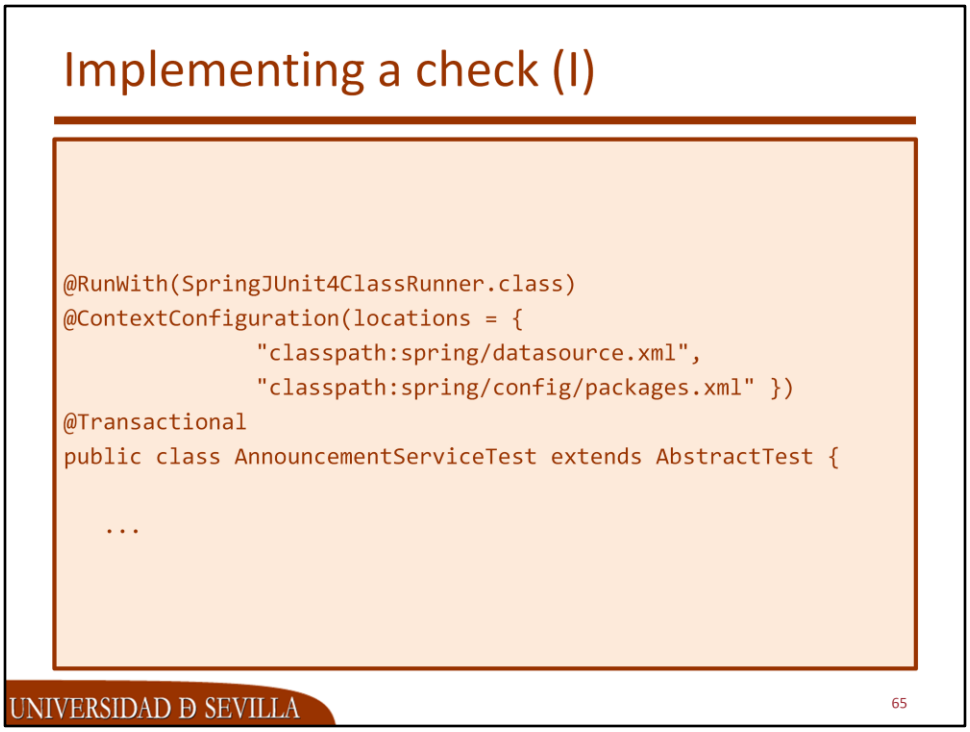
First, we'll learn on how to implement them and then on how to configure and run them.

Let’s start with the implementation details.



There are two key classes to implement checks, namely: “Assert”, which we presented in a previous slide, and “AbstractTest”, which we’re going to describe here. Class “AbstractTest” provides a foundation to write your own tests or checks, namely:

- “setUp”: this method is invoked automatically before each test or check is run; please, note that you don’t have to invoke it explicitly, it’s JUnit that invokes it.
- “authenticate”: this method takes a username as a parameter and simulates that the corresponding user has authenticated to the system. If the parameter is “null”, it then unauthenticates the user who is currently authenticated, if any.



You can write your own test classes very easily: extend class "AbstractTest" and add the following annotations:

1. "@RunWith(SpringJUnit4ClassRunner.class)": this tells JUnit that it must use the Spring runner so that Spring can instantiate the classes required by your code.
2. "@ContextConfiguration(locations = {…})": this specifies the Spring configuration that must be used to run your checks. In this case, this includes the configuration file that has the information about the connection to the database ("datasource.xml") and the file that knows the packages in which repositories and services are located ("packages.xml").
3. "@Transactional": this instructs Spring to activate the transaction support that it provides.

Implementing a check (II)

```
...
// Service under test ---------------------------------

@Autowired
private AnnouncementService announcementService;
...
```

UNIVERSIDAD D SEVILLA

66

Then comes an attribute that allows to reference the repository or service that we need to check. These attributes are annotated with “@Autowired”, which tells Spring to instantiate them automatically. Note that this works because we have told Junit to use the Spring runner with the “@RunWith” annotation; otherwise, the “@Autowired” annotation would be ignored and it’ll result in null pointer exceptions.

Implementing a check (III)

```
...
// Tests ------------------------------------------

@Test
public void testSaveAnnouncements() {
    Annnouncement announcement, saved;
    Collection<announcement> announcements;

    announcement = announcementService.create();
    ... Initialise the announcement ...
    saved = announcementService.save(announcement);
    announcements = announcementService.findAll();
    Assert.isTrue(announcements.contains(saved));
}
...
```


Now, you have to write your code to check a repository or a service in a method annotated with "@Test". In this slide, we show an example to test saving an announcement. Note that we use the announcement service to create a new announcement and then initialise its properties; next, we save the announcement and check that it is in the list of announcements that we also retrieve through the service. Just realise that the announcement that we create in memory is not instrumented; calling the save method creates a new object that is instrumented and returns it. We strongly recommend that you should perform the following experiment: use the "announcement" object instead of the "saved" object in the assertion; you'll soon realise that the check fails because the objects that are retrieved from the database are instrumented and have nothing to do with the object that you created in memory.

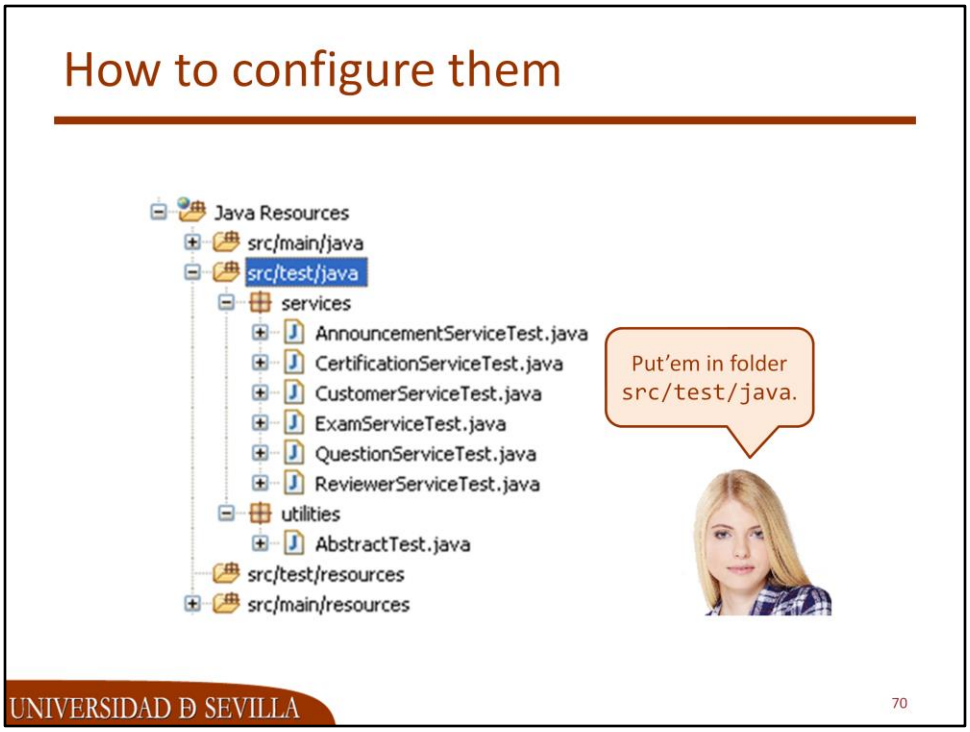
NOTE: by default, JUnit rollbacks the database when it finishes executing a test; this means that every test is executed on the same database; that is, you *should not* expect that the changes made in one test are available to the next test.

Implementing a check (IV)

```
...
@Test
public void testFindRegistered() {
     Collection<Announcement> all;

     super.authenticate("customer1");
     all = announcementService.findRegistered();
     Assert.isTrue(all.size() == 5);
     super.authenticate(null);
}
...
```

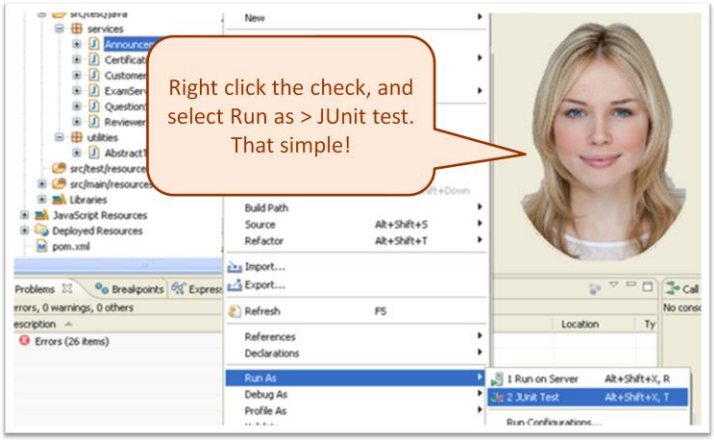

We finally explore how to implement checks that require a specific user to have logged in to the system. In this example, we are checking a method to list the announcements to which a user has registered. Such a method requires a user to authenticate, but we don't have a user interface, yet. In those cases, we can use the "authenticate" method that is inherited from the "AbstractTest" class; this method simulates that a user is authenticating; please, note that no passwords are involved, since we're simulating authentication internally. To log out, you must simply call "authenticate" with a null parameter. Note that this check's very weak, since we only check that "customer1" has registered to five announcements; a stronger check might involve checking that each of the announcements is actually the correct announcement.

Let’s now explore how to configure and run checks.



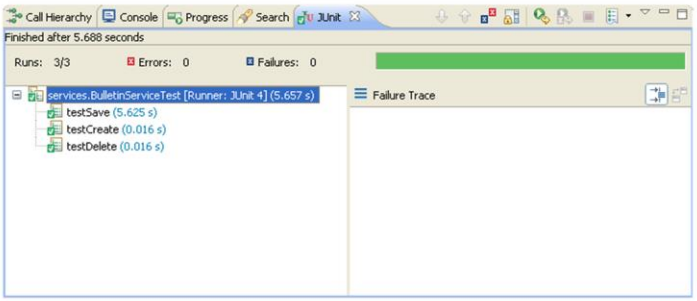
Configuring them is really simple: just put them at "src/tests/java". We strongly recommend that you should put the "AbstractTest" class in a package called "utilities", your service checks in a folder called "services", and your repository checks in a folder called "repositories". Note that it's not likely that you need to check a repository individually since they are typically checked transitively when you check your services; but it might be the case that you need to implement a quick check to make sure that a repository works well before the corresponding service is implemented.

How to run checks?

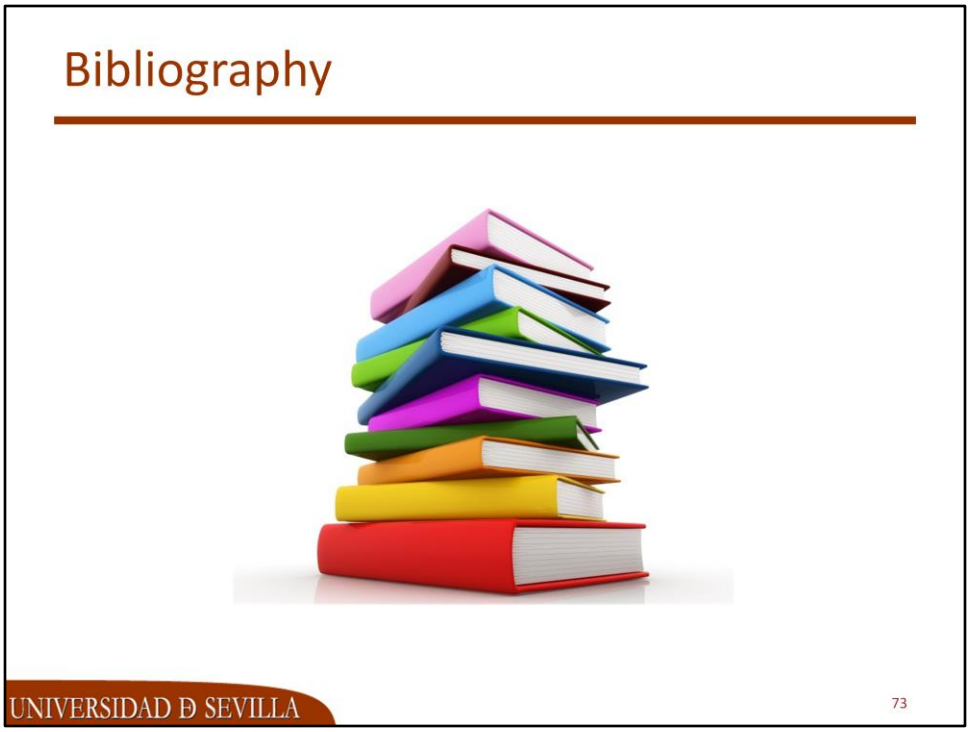


It's very easy from Eclipse. You just have to right click the class that you want to run, choose "Run As…", and then "JUnit Test". This will execute every method with a "@Test" annotation in the selected class.

What's the output?



The output looks like in this slide. Click on the JUnit view and you should see a green bar if everything's gone right or a red bar if something's failed. If you get the red bar, then you've found a problem and you must debug your code to find it.



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

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

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

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

Time for questions, please

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

The next lecture

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

UNIVERSIDAD Ð SEVILLA

75

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

Thanks for attending this lecture! See you next day!