**SPRING DATA JPA**

**Dependencies**

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
</dependency>

<dependency>  
 <groupId>org.postgresql</groupId>  
 <artifactId>postgresql</artifactId>  
 <scope>runtime</scope>  
</dependency>

**Application.Properties**

# URL for our Database  
spring.datasource.url=jdbc:postgresql://localhost:5432/amigoscode  
  
#Username and Password for our database  
spring.datasource.username= postgres  
spring.datasource.password= root  
  
#Spring Jpa configuration  
# CREATE=DROP : Create the schema and before the application shuts down, drop everything  
# We use the create-drop configuration just because we are testing a lot of things  
spring.jpa.hibernate.ddl-auto=create-drop  
  
#We want to see the SQL statements that hibernate will generate  
spring.jpa.show-sql=true  
  
# Here we specify the dialect  
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect  
  
#And we also want the sql to be formatted  
spring.jpa.properties.hibernate.format\_sql=true

1. **@Entity (name = \*nameOfEntity\*) : Used to create a new table from our class. The default name is the name of our class, but it’s a good practice to specify it.**
2. **@Id : Used for the primary key of our table**

**Text

Description automatically generated**

@Id  
@GeneratedValue(generator = "uuid2")  
@GenericGenerator(name = "uuid2", strategy = "uuid2")  
@Type(type = "uuid-binary")

**@SequenceGenerator : Used for generating a sequence where**

1. **allocationSize is the size we want to increase the id by**
2. **@GeneratedValue : generator has to have the same name as @SequenceGenerator : sequenceName**
3. **Text

   Description automatically generated**

**Other attributes for the @Column annotation might be**

* **Updateable : not to let this column be updated**
* **Insertable : not to let data be inserted in this column**

**Because we made the email be unique, we can see in the database that we have a new constraint but with an ugly name.**

**Text

Description automatically generated**

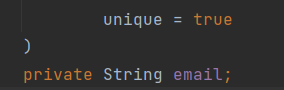
**We can make our constraints look good by using the**

**@Table annotation**

**Text

Description automatically generated**

**For the @Table annotation, we have the uniqueConstraints attribute which takes a list of @UniqueConstraint. We can see how we renamed the constraint for the email column here.**

**Now we no longer need to keep  the unique attribute for the email @Column**

**Now we need a repository to acces this data. Our repository will extend a repository provided by string data jpa.**

**Diagram

Description automatically generated**

**To create a repository, we create an interface which extends JpaRepository<\*NameOfClass\*, \*TypeOfPrimaryKey\*>**

**Text

Description automatically generated**

**DERIVED QUERIES**

[**https://www.youtube.com/watch?v=GVeY08vUiPE&list=PL50BZOuKafAYqbYZ18lqVet\_zA2MsZZcW&index=2&ab\_channel=ThorbenJanssen**](https://www.youtube.com/watch?v=GVeY08vUiPE&list=PL50BZOuKafAYqbYZ18lqVet_zA2MsZZcW&index=2&ab_channel=ThorbenJanssen)

**Queries that are generated by JPA, we just need to fill in the gaps.**

**Diagram

Description automatically generated**

**Graphical user interface, application

Description automatically generated**

While “find” might lead to no result at all, “get” will always return something – otherwise the JPA repository throws an exception.

**Example**

**Graphical user interface, text, application

Description automatically generated**

**SpringDataJPA also allows us to derive a query based on 2 properties that are linked by foreignKeys.**

**Here we reference the Books Attribute on the Author Entity and we associate it with the Title Attribute of the Book Entity.**

**Text

Description automatically generated with low confidence**

**SpringDataJpa also provides comparison Operators**

**Graphical user interface, text, application

Description automatically generated**

****

**We can also order the result we get:**

1. **Asc for ascending order**
2. **Desc for descending order**

****

**We can also choose the sorting dinamically with a parameter of type Sort**

****

****

**We can also limit the number of results by adding the words TOP or FIRST, followed by a number, between the find and by keywords**

****

**We can also paginate the results of a derived query**

**We can add a parameter of type Pageable and change the return type to Page**

**A picture containing graphical user interface

Description automatically generated**

**We only have to specify 2 parameters:**

1. **How many pages we want to retrieve**
2. **How many records should be on a page**

**A picture containing text

Description automatically generated**

**Example:**

**Text

Description automatically generated**

**Text

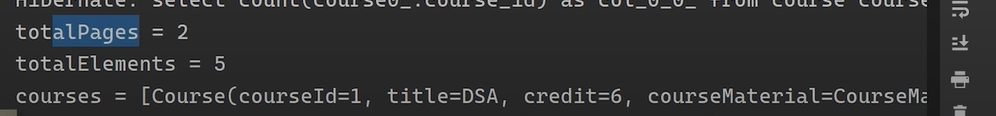
Description automatically generated**

**The database currently has 5 records:**

**Table

Description automatically generated**

**Results:**

****

**Why are there 2 pages? Because we defined that one page has 3 elements.**

**For the courses, it can’t be seen but we printed the first 3.**

**We can also sort and page at the same time**

**Text

Description automatically generated**

**Text

Description automatically generated**

**Another example of declaring our query with paging:**

**Graphical user interface, text

Description automatically generated**

**Text

Description automatically generated**

**RELATIONSHIPS IN SPRING DATA JPA**

**We will have the following database schema : A student, a Course and a Teacher.**

**We will have the following relationships:**

* **A subject will have many students**
* **A subject will have one teacher**
* **A student will have many subjects**
* **A teacher will have many subjects**

**@ManyToMany**

**So first, we know that a subject will have many students, and a student will have many subjects.**

**So on the Student’s side, we create a new variable representing the courses he is inrolled in.**

****

**And we go to the Subject side and we do the same**

****

**Now we specify a @ManyToMany relationship and to do this we specify that we need to join the tables using the @JoinTables annotation.**

**Graphical user interface, website

Description automatically generated**

**This means that we create a new table, called student\_enrolled, which will have one column of subject\_id and another one of student\_id.**

**On the Student side, it’s easier.**

**We just have to specify the @ManyToMany relationship and say by who it is mapped by. In our case, it’s mapped by the enrolledStudents Set variable from the Subject Object.**

**Graphical user interface, application

Description automatically generated**

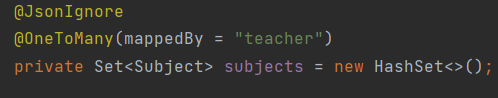
**Now for this @ManyToMany problem, we might have the recursive problem.**

**What we can do is use the @JsonIgnore annotation to do that.**

**So basically we use this annotation to tell the program that we don’t want to**

**@ManyToOne**

**Now we code the fact that a teacher can have many courses, but a course can only have one teacher.**

****

**mappedBy = \*nameOfPropertyDefinedInTheSubjectClass\***

**Text

Description automatically generated**

**@JoinColumn:**

* **Name: The name of the column that will be created**
* **referencedColumnName = the name of the referencedColumn**

**In this way, the Subject table will have a new column named teacher\_id referencing the id of the teacher table.**

**Graphical user interface, application

Description automatically generated**

**The teacher table will not have a new column**

**Graphical user interface, application

Description automatically generated**

**@OneToOne**

**The one to one relationship is much simpler**

**Graphical user interface, text

Description automatically generated**

**Graphical user interface, text, application

Description automatically generated**

**This is for our example where a smartDevice had only one sensor.**

**The student will have a new column called pet\_dog\_id**

**Graphical user interface, application

Description automatically generated**

**The dog will not have any new column**

**Graphical user interface, application

Description automatically generated**

**Another example of @OneToOne**

**In this example we have A course and A course Material**

**In the CourseMaterial Class:**

**Text

Description automatically generated**

**We will join a new column called course\_id which is referencing the column called courseId from the Course Object.**

**Graphical user interface, application

Description automatically generated**

**THE CLASS WITH „MAPPED BY” IS THE CLASS WHICH DOESN’T CONTAIN THE NEW COLUMN**

**@Embedded**

**We can use this to create separate Java Objects but store data in the same table by embedding one java object in the other**

**Let’s say for example our student has some fields called**

**Graphical user interface, text, application, chat or text message

Description automatically generated**

**So we can create a new Object called Guardian which has a separate table in the dabase, and then create a relation between the Student and the Guardian.**

**OR**

**We can create a new Object called Guardian but save its info in the student’s table. And that we do with embedded.**

**What we have to do is create the new Guardian Class and annotate it as @Embeddable**

**Text

Description automatically generated**

**And now go to the Student class and define a new object attribute for the Student class with the annotation @Embedded**

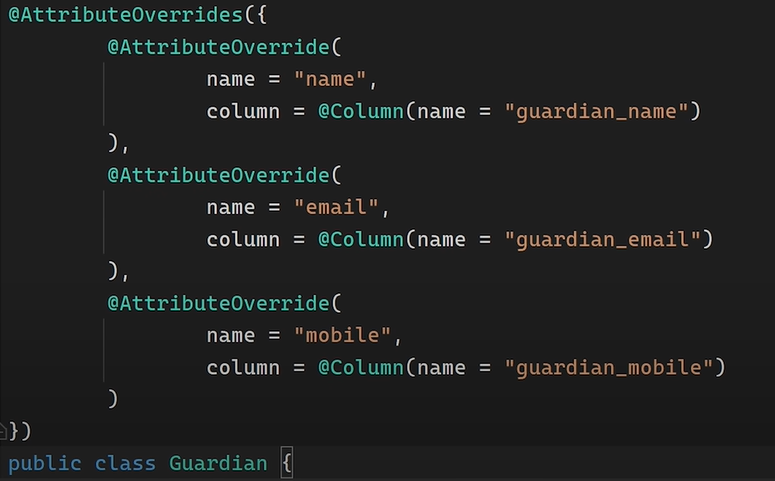
**Text

Description automatically generated**

**And everything works.**

**In case we already had the 3 guardian fields in the Student table ( so before creating a new Class called Guardian). And now we create a new class called Guardian but with different names for the attributes, we can map the new names to the old names that already existed in the database.**

**Just above the Guardian class**

****

**In this case, the student used to have the 3 attributes called guardian\_name, guardian\_email and guardian\_mobile, before we created the guardian class.**

**We create the guardian class and we create it with 3 attributes called name, email and mobile. We can now map them using the @AttributeOverrides annotation**

**(Native Queries, Select all)**

**We can create native queries like this**

**Graphical user interface, text

Description automatically generated**

**Which is equivalent to this**

**Text

Description automatically generated**

**In case we create queries for updating the database, not just fetching data, we need to annotate that query with @Modifying and @Transactional**

**Graphical user interface, text, application

Description automatically generated**

**The @Modifying annotation is used to enhance the @Query annotation so that we can execute not only SELECT queries, but also INSERT, UPDATE, DELETE, and even DDL queries.**

**FETCHING**

**When we have more classes with relationships between them, we can choose how the fetching is done**

**Fetch.EAGER: means that when u try to fetch data for a class, it will also fetch the data for the classes the initial class has relations with**

**Fetch.LAZY means the opposite. Only fetch data for your class, not data for the classes with which your class has relations.**