Assignment: Data persistence

In this exercise you will practice with using JPA in combination with an H2 SQL database and also MySql.

Difficulty: ☆☆☆☆☆

Estimated time: 3hs

Learning objectives:

- I can map relational database tables into Java entities:
  - One-to-one relationship
  - One-to-many relationship
  - Many-to-many relationship
- I can use Spring Data's auto generated repositories for basic CRUD operations
- I can test my repositories with an in-memory database (h2)
- I can use Flyway for database versioning
- I can use Transactional annotation in order to delimit a database transaction
- I can use MySql as the real application database

Formatted: English (United States)

Formatted: English (United States)

Step 1: Download the startup project

Download the startup Java project and open it in IntelliJ (or another IDE). It contains example code on how to handle persistence using Spring boot and JPA.

Field Code Changed

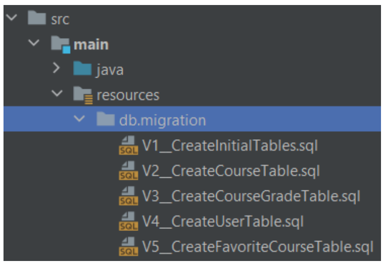
The project is configured to use H2 as the database. The great thing about H2 is that you don't need much to run it, just a simple dependency in *build.gradle* (recall that Gradle will automatically download and add the jar needed to use H2 to your application), and some setup options in the *application.properties*. It's great for prototyping and testing.

Step 2: Database versioning

When working with SQL databases it's important to keep track if all the expected schema changes are applied to it. The startup projects uses Flyway in order to provide database versioning.

Field Code Changed

Check the contents of folder *src/main/resources/db/migration*:



```
src
  main
    java
    resources
      db.migration
        V1__CreateInitialTables.sql
        V2__CreateCourseTable.sql
        V3__CreateCourseGradeTable.sql
        V4__CreateUserTable.sql
        V5__CreateFavoriteCourseTable.sql
```

Each file should be created with the pattern *V<SCRIPT NUMBER>__<SCRIPT NAME>.sql* and can contain any valid sql statements (just as in the example). Please note: there are TWO underscore chars in the pattern between the script's number and name.

When the application starts Flyway makes sure all scripts are applied to the database. If there's any issue then the application will not start at all for safety. Then the developer must manually undo the failed migration script changes (if any) and delete the failed migration row in table "flyway_schema_history" before retrying a fixed version of the script.

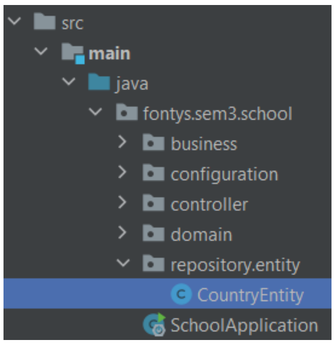
Take some time to read [Flyway documentation] to understand properly how it works. One important feature of migrations is that you cannot change a migration version (a .sql file) which has been executed successfully (i.e., it has been migrated), so it means you always need to create a new migration when changing your database (e.g., adding or changing tables).

In the following steps we will map each table in these scripts into Java classes.

PS: IntelliJ displays the path as "db.migration", because *db* folder is empty. Make sure you create a folder *db* and then a folder *migration* inside it.

Step 3: First JPA Entity

The first table in V1__CreateInitialTables.sql is "country". There's already a *CountryEntity* class in package *fontys.sem3.school.repository.entity*, but it's not a JPA entity yet.



```
src
  main
    java
      fontys.sem3.school
        business
        configuration
        controller
        domain
        repository.entity
          CountryEntity
        SchoolApplication
```

Please add the @Entity, @Table, @Id, @GeneratedValue and @Column annotations below in order to make it a JPA entity. The other existing annotations are providing constructors, getters/setters, validations and more.

```java
package fontys.sem3.school.repository.entity;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.hibernate.validator.constraints.Length;

import jakarta.persistence.*;
import jakarta.validation.constraints.NotBlank;

@Entity
@Table(name = "country")
```

```
@Builder
@Data
@AllArgsConstructor
@NoArgsConstructor
public class CountryEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @NotBlank
    @Length(min = 2 ,max = 50)
    @Column(name = "name")
    private String name;

    @Column(name = "code")
    @Length(min = 2, max = 2)
    private String code;
}
```

Now our first entity is mapped!

Step 4: Spring Data Auto-generated repository

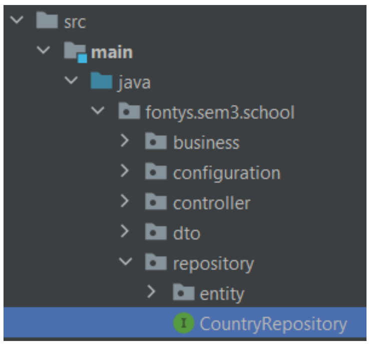
There are different ways of interacting with JPA entities, like using EntityManager or Spring Data auto-generated repositories.

Field Code Changed

Field Code Changed

In this assignment we'll take the second approach because it's simpler and we don't really have complex queries to implement.

Let's create a *CountryRepository* interface in the *repository* package:



```
src
  main
    java
      fontys.sem3.school
        business
        configuration
        controller
        dto
        repository
          entity
          CountryRepository
```

```
package fontys.sem3.school.repository;

import fontys.sem3.school.repository.entity.CountryEntity;
import org.springframework.data.jpa.repository.JpaRepository;

public interface CountryRepository extends JpaRepository<CountryEntity,
Long> {
    boolean existsByCode(String code);
}
```

The *magical* thing about using Spring Data here is that we don't need to actually provide an implementation for this interface. This will be done automatically by Spring when the application starts! Actually, the class which implements those methods is the SimpleJpaRepository from JPA.

Formatted: Font: Italic, Complex Script Font: Italic

CountryRepository extends interface JpaRepository which has many useful methods out of the box for saving, deleting and finding entities. And if you use the supported derived query methods syntax new methods can be defined in the interface (like existsByCode) and they will also be automatically implemented by Spring.

Field Code Changed

Step 5: Using the repository in the country related use cases

We have two use-case implementations partially complete:

1. GetCountriesUseCaseImpl and
2. CreateCountryUseCaseImpl.

They are only missing the *CountryRepository* injection and right method calls.

Please finish implementing the use-cases above.

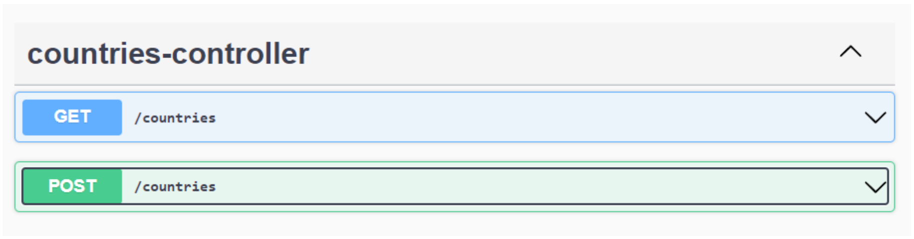
When you're ready to test your changes then follow to the next step.

Step 6: Running the App and testing the use-cases

Start the Spring Boot App and open the Swagger UI url. Here you can see all services provided by the application and also test them.

Field Code Changed

Look for *CountriesController* and then try out the POST end GET endpoints. The POST endpoint should add a new country and the GET should list all the saved ones.


countries-controller

GET /countries

POST /countries

Note: you can also use Postman for testing.

Step 7: Many-to-one relationship

The second table created in our V1** script file is “student”. Student has id, pcn, name and country_id columns.

When mapping student into a Java class we need to have a reference to the country entity instead of a simple country id field (this is the idea of ORM: the relationships are made using the entities instead of ids). And because many students can belong to the same country, then from student’s perspective this relationship is a many-to-one relationship.

So let’s create the student class inside our entity package:

```
package fontys.sem3.school.repository.entity;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.hibernate.validator.constraints.Length;

import jakarta.persistence.*;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;
import java.util.List;

@Entity
@Table(name = "student")
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class StudentEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @NotNull
    @Column(name = "pcn")
    private Long pcn;

    @NotBlank
    @Length(min = 2, max = 50)
    @Column(name = "name")
    private String name;

    @NotNull
    @ManyToOne
    @JoinColumn(name = "country_id")
    private CountryEntity country;

}
```

Notice the @ManyToOne and @JoinColumn annotations on field country. The name attribute of @JoinColumn refers to “student” table column that should be used in order to fetch the student’s country, which is “country_id”.

Now let’s create a Spring Data JPA repository for this entity in our repository package:

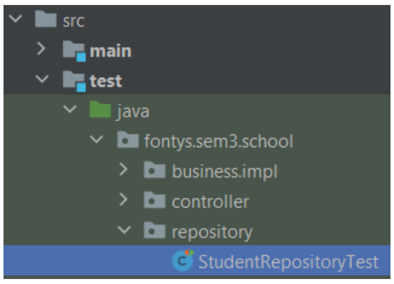
```
package fontys.sem3.school.repository;
```

```
import fontys.sem3.school.repository.entity.StudentEntity;
import org.springframework.data.jpa.repository.JpaRepository;

public interface StudentRepository extends JpaRepository<StudentEntity,
Long> {

}
```

That’s it, we’re done! When the App starts Spring will auto-generate an implementation of it for us.Is there a way of testing this repository and entity right away? Or do we need controller and business layers to be in-place?? Well, it is possible write a test and check right away if your entity works as expected.

Step 8: Testing a repository with Spring

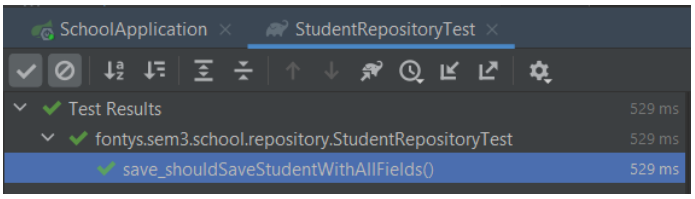
Let’s create the following *StudentRepositoryTest* class in package *repository* from our **test** folder.



```
package fontys.sem3.school.repository;

import fontys.sem3.school.repository.entity.CountryEntity;
import fontys.sem3.school.repository.entity.StudentEntity;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.test.context.junit.jupiter.SpringExtension;

import jakarta.persistence.EntityManager;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;

@ExtendWith(SpringExtension.class)
@DataJpaTest
class StudentRepositoryTest {

    @Autowired
    private EntityManager entityManager;

    @Autowired
    private StudentRepository studentRepository;

    @Test
    void save_shouldSaveStudentWithAllFields() {
        CountryEntity brazil = saveCountry("Brazil", "BR");
        StudentEntity student = StudentEntity.builder()
                .pcn(111L)
                .name("Ronaldo Nazario")
                .country(brazil)
                .build();

        StudentEntity savedStudent = studentRepository.save(student);
        assertNotNull(savedStudent.getId());

        savedStudent = entityManager.find(StudentEntity.class,
savedStudent.getId());
        StudentEntity expectedStudent = StudentEntity.builder()
                .id(savedStudent.getId())
                .pcn(111L)
                .name("Ronaldo Nazario")
                .country(brazil)
                .build();
        assertEquals(expectedStudent, savedStudent);
    }

    private CountryEntity saveCountry(String name, String code) {
        CountryEntity country =
CountryEntity.builder().name(name).code(code).build();
        entityManager.persist(country);
        return country;
    }

}
```

@ExtendWith(SpringExtension.class) and @DataJpaTest are special annotations that tell our test will be extended by Spring and also that it's a Spring Data JPA Test. When running the test Spring will enable dependency injection for repository related classes like the repository we want to test, StudentRepository, and also the previously referred EntityManager. EntityManager can be used to save or find any JPA entity, which is very handy for testing.

After running this test we should feel confident enough the mapping from student to country is working well and move on to our next entity.



```
Test Results                                              529 ms
  fontys.sem3.school.repository.StudentRepositoryTest     529 ms
    save_shouldSaveStudentWithAllFields()                 529 ms
```

Note 1: this test is using a fresh H2 database for every test class.

Note 2: As you can see this is not just another unit test, it's more. It requires part of the application components to be spun-up and interact with a database. It's considered a "partial integration test" and is very useful for testing the persistence layer for real.

Step 9: Entity “CourseEntity”. Your turn!

The next entity we need to map is Course (please check file V2__CreateCourseTable.sql). Please create a *CourseEntity* class, *CourseRepository* and *CourseRepositoryTest* just as we did with the previous entity.

Step 10: Entity “CourseGrade”. Your turn!

The next entity we need to map is CourseGradeEntity (please check file V3__CreateCourseGradeTable.sql). Please create a *CourseGradeEntity*, *CourseGradeRepository* and *CourseGradeRepositoryTest* just as we did with the previous entity.

Notice that CourseGrade has two many-to-one relationships: student and course. With other words: many grades can belong to the same student or to the same course. Refresh your mind how to map these relationships in Step 7.

This is how CourseGradeEntity should look like:

```java
package fontys.sem3.school.repository.entity;

import lombok.*;

import jakarta.persistence.*;
import jakarta.validation.constraints.Max;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotNull;

@Entity
@Table(name = "course_grade")
@Builder
@Data
@AllArgsConstructor
@NoArgsConstructor
public class CourseGradeEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @NotNull
    @ManyToOne
    @JoinColumn(name = "student_id")
    @EqualsAndHashCode.Exclude
    @ToString.Exclude
    private StudentEntity student;

    @NotNull
    @ManyToOne
    @JoinColumn(name = "course_id")
    @EqualsAndHashCode.Exclude
    @ToString.Exclude
    private CourseEntity course;
    @Column(name = "grade")
    @NotNull
    @Min(0)
    @Max(10)
    @EqualsAndHashCode.Exclude
    private Double grade;
}
```

Step 11: One-to-many relationship

Now *CourseGradeEntity* will be mapped into the *StudentEntity*. Each student can have many grades, what makes this relationship a one-to-many from student's perspective.

In the StudentEntity class add the following field:

```
@OneToMany(mappedBy = "student")
private List<CourseGradeEntity> courseGrades;
```

Now create a test method in *StudentRepositoryTest* where a student has grades to be retrieved. Assert that the expected grades are retrieved for the student.

The test method should look like the code below:

```
@Test
void find_shouldReturnStudentGrades_WhenTheyExist() {
    CountryEntity brazil =
CountryEntity.builder().name("Brazil").code("BR").build();
    entityManager.persist(brazil);

    CourseEntity softwareSemesterThree = CourseEntity.builder()
            .name("S3-SOFT")
            .description("Software semester 3")
            .build();
    entityManager.persist(softwareSemesterThree);

    StudentEntity romario = StudentEntity.builder()
            .pcn(11L)
            .name("Romario")
            .country(brazil)
            .build();
    entityManager.persist(romario);

    CourseGradeEntity grade = CourseGradeEntity.builder()
            .grade(10d)
            .course(softwareSemesterThree)
            .student(romario)
            .build();
    entityManager.persist(grade);

    entityManager.flush(); // Makes sure queries are sent to the database
    entityManager.clear(); // Makes sure internal cache is cleared

    StudentEntity savedStudent = entityManager.find(StudentEntity.class,
romario.getId());
```

```
    assertEquals(List.of(grade), savedStudent.getCourseGrades());
}
```

Step 12: One-to-one relationship

Time to map the “users” table into a class. You can find it in file *V4__CreateUserTable.sql*. This table has a *student_id* column which is a foreign key to table student.

As you can maybe guess a user can only be related with a single student (and vice-versa). So this is a new kind of relationship: one-to-one.

This is how *UserEntity* should look like.

```java
package fontys.sem3.school.repository.entity;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.hibernate.validator.constraints.Length;

import jakarta.persistence.*;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;

@Entity
@Table(name = "users")
@Builder
@Data
@AllArgsConstructor
@NoArgsConstructor
public class UserEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @NotBlank
    @Length(min = 2, max = 20)
    @Column(name = "username")
    private String username;

    @Column(name = "password")
    @Length(max = 50)
    private String password;

    @NotNull
    @OneToOne
    @JoinColumn(name = "student_id")
    private StudentEntity student;
}
```

Notice the @OneToOne and @JoinColumn annotations. Do you remember what the name attribute of @JoinColumn means?

Now you can create a *UserRepository* and test for it just as we did previously.

Does your mapping work as expected? Then let's move forward.

Step 13: Many-to-many relationship

A student will be able to mark his most beloved courses as favorite ones. Many students can "favorite" a single course and many courses can be "favorited" by a same student. This is the so called many-to-many relationship.

In order to implement this relationship we need a new table (also known as join table), *favorite_course* , which is provided in file *V5__CreateFavoriteCourseTable.sql*. Each row in this table represents a course "favorited" by a student.

Now we would like to map this relationship in StudentEntity class. This is how it looks like:

```
@ManyToMany(cascade = CascadeType.ALL)
@JoinTable(
        name = "favorite_course",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id"))
private List<CourseEntity> favoriteCourses;
```

Notice the use of annotations @ManyToMany and @JoinTable. Can you figure out how @JoinTable works?

Also notice the use cascade attribute in the @ManyToMany. "CascadeType.ALL" means whatever persistence operation we do with student should also happen with the mapped entity. This attribute is available in the other annotations but use it with wisdom. It's recommended to avoid it, when possible, because of performance.

Now let's add another test method in *StudentRepositoryTest* in order to test this relationship. It should look something like:

```
@Test
void find_shouldReturnStudentFavoriteCourses_WhenTheyExist() {
    CountryEntity brazil =
CountryEntity.builder().name("Brazil").code("BR").build();
    entityManager.persist(brazil);

    CourseEntity softwareSemesterThree = CourseEntity.builder()
            .name("S3-SOFT")
            .description("Software semester 3")
            .build();

    StudentEntity romario = StudentEntity.builder()
            .pcn(11L)
            .name("Romario")
            .country(brazil)
            .favoriteCourses(List.of(softwareSemesterThree))
            .build();
    entityManager.persist(romario);

    StudentEntity savedStudent = entityManager.find(StudentEntity.class,
romario.getId());

    assertEquals(List.of(softwareSemesterThree),
savedStudent.getFavoriteCourses());
}
```

Step 14: Use of @Transactional

Use-cases usually should behave atomically: either all database writes succeed, or they all fail. This can be provided by a database transaction.

Field Code Changed

Field Code Changed

We can delimit transactions in Spring applications by using @Transactional annotation. Use this annotation on the business layer. More specifically at the method that triggers the use-case or service logic.

In this example app, applying @Transactional to use-case *CreateCountryUseCaseImpl* will look like:

```
…
import jakarta.transaction.Transactional;

@Service
@RequiredArgsConstructor
public class CreateCountryUseCaseImpl implements CreateCountryUseCase {
    private final CountryRepository countryRepository;

    @Transactional
    @Override
    public CreateCountryResponse createCountry(CreateCountryRequest
request) {
…
}
```

Notice the import of “jakarta.transaction.Transactional” and also annotation @Transactional on method *createCountry*. Everything else keeps the same.

That’s all it takes to have all queries triggered inside the method inside a same database transaction. Always remember of using transactions in your use-cases! You don’t want to have data inconsistencies in your database!

Note: it’s also important to check if the database connections are not configured as auto-commit. This is already done in the *application.properties* of this project:

Field Code Changed

```
spring.jpa.properties.hibernate.connection.autocommit=false
```

Step 15: Switch to a MySql database

Time to use a real database. MySql is very popular database and maybe you already have it installed. If you don’t you will need it for this step.

Field Code Changed

After installing MySql in your computer, let’s configure the app to use it.

1. Change dependencies in build.gradle (highlighted):

```
dependencies {
   implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
   implementation 'org.springframework.boot:spring-boot-starter-web'
   implementation 'org.springframework.boot:spring-boot-starter-validation'
   implementation 'org.flywaydb:flyway-core'
   implementation 'org.flywaydb:flyway-mysql'
   implementation 'mysql:mysql-connector-java:8.0.33'
   implementation 'org.springdoc:springdoc-openapi-starter-webmvc-ui:2.1.0'
   compileOnly 'org.projectlombok:lombok'
   annotationProcessor 'org.projectlombok:lombok'
   testImplementation 'org.springframework.boot:spring-boot-starter-test'
   testRuntimeOnly 'com.h2database:h2'
}
```

Now h2 is a test dependency and will only be used in the tests.
MySql driver is dependency 'mysql:mysql-connector-java' and you also need an extra dependency for flyway ('org.flywaydb:flyway-mysql'). They are already in the code above.

2. Change *src/main/resources/application.properties*:

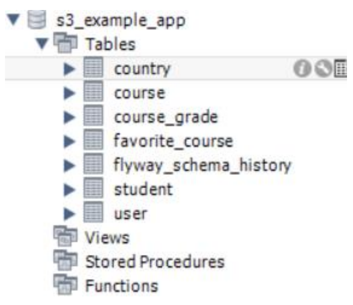
```
spring.jpa.hibernate.ddl-auto=none
spring.datasource.url=jdbc:mysql://localhost:3306/DATABASE_NAME
spring.datasource.username=USER
spring.datasource.password=PASSWORD
spring.jpa.open-in-view=false
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.connection.autocommit=false
```

Replace the upper-cased variables above to match your MySql installation.

Note: JPA can create, i.e. initialize, the database for you, however, this is usually not a good practice for larger and in production projects, thus make sure you understand it properly and when to use it.

Now it's just about running the App. Make sure your application started successfully.

If you connect to your database you should see all the tables created by Flyway there:



```
▼ s3_example_app
  ▼ Tables
    ▶ country
    ▶ course
    ▶ course_grade
    ▶ favorite_course
    ▶ flyway_schema_history
    ▶ student
    ▶ user
    Views
    Stored Procedures
    Functions
```

And everything should still be working via Swagger with the data being saved in MySql.

Formatted: Font: 10 pt, Complex Script Font: 10 pt

Formatted: Font: 10 pt, Font color: Accent 4, Complex Script Font: 10 pt

Formatted: Font: 10 pt, Font color: Accent 4, Complex Script Font: 10 pt

Formatted: Font: 10 pt, Complex Script Font: 10 pt

Formatted: Font: 10 pt, Complex Script Font: 10 pt, English (United States)

Formatted: Font: 10 pt, Font color: Accent 4, Complex Script Font: 10 pt

Formatted: Font color: Accent 4

Field Code Changed

And your persistence tests should still be running against h2 in-memory database (settings in file *src/**test**/resources/application.properties*).

Final remarks

Wow, this was a big one! If you made it to the end, you've earned a good base to tackle most issues regarding JPA and persistence that will appear during the group or individual assignment. Congrats for that!

A next step very common in professional projects is to split the execution of unit (very fast execution) and integration tests (slow execution). Feel free to do it if you have the time.

Field Code Changed