Databases and Object-Relational Mapping

License

This work by Thalita Vergilio at Leeds Beckett University is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Contents

License

Contents

Introduction

The Domain Model

JPA and Data Persistence

Implementation

Assignment

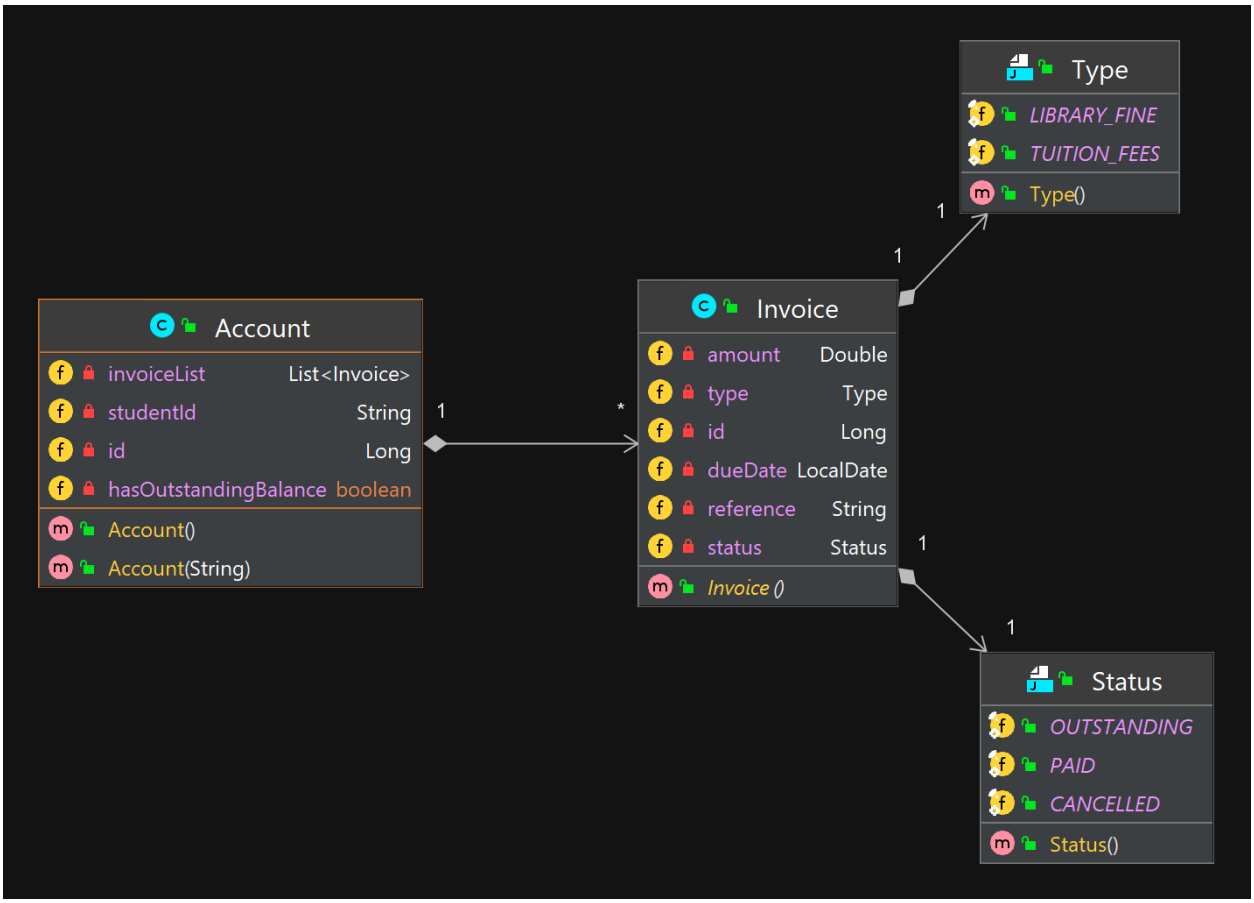
Conclusion

Introduction

The goal of this week's lab is to design and implement the domain model for our Student microservices. We are going to leverage Spring JPA's powerful abstractions to handle the saving of data programmatically, without writing a single line of SQL!

The Domain Model

When building software applications, it is inevitable that, at some stage, you will be faced with the issue of data persistence. In object-oriented design, a fundamental step following requirements gathering is to conceptualise the business in terms of abstractions called domain classes. Domain classes are data-driven conceptual units characterised by a coherent collection of properties and methods. They generally represent something meaningful within the business. If we take the Finance microservice as an example, its purpose is to issue invoices against student accounts and allow those invoices to be paid. Its domain model has therefore two main classes: Account and Invoice, as shown in the diagram below.



UML class diagram showing the domain model for the Finance microservice

Each class has a number of properties, shown in purple directly underneath the class name, and constructors, shown in yellow at the bottom. For instance, an account has a unique ID, a student ID, a boolean representing whether the account has any outstanding balance, and a list of invoices associated with it. It has two constructors: one which takes no arguments, and one which takes the student ID as a String.

Classes work like blueprints from which objects can be created. This is called instantiation. You can think of the Account class as representing accounts in general, or the blueprint of any account. If we wanted to represent actual accounts that exist in the system (such as Thalita's account or Duncan's account), we would need to create objects to represent them. This way, both accounts can have an invoice list, but Thalita's invoice list can be different from Duncan's invoice list (and Thalita's ID and student ID can also be different from Duncan's).

You may have noticed that our domain classes are very simple, they do not have any methods. In reality, they will have boilerplate getters and setters, equals and hashcode implementations provided by a very convenient library called Lombok. We do not want to add any other behaviour to our domain classes. Following best practices, our domain classes should do just this: provide us with a data-driven representation of our domain objects and specify how they should be persisted. Any desired behaviour involving these objects belong in a completely different category of class (i.e. services).

Now that we have touched upon data persistence, here is a question for you.

Answer in your log book: what gets persisted to a database, classes or objects? Why is this needed?

In your log book: explain the role of domain classes in an MVC architecture.

In your log book: based on the example provided, create a UML class diagram to represent the domain model for the Student microservice[1].

JPA and Data Persistence

[1] Hint: start with a Student and a Course. A student can enrol in many courses, and a course can have many students enrolled in it. There are different ways in which you can represent this many-to-many relationship. See, for example, https://www.baeldung.com/jpa-many-to-many.

Before we begin working on our domain model, let's have a look at how data persistence is handled in the Account domain class of the Finance microservice. Note how schema creation and data definition are abstracted away from the developer and handled by JPA annotations.

```java
package uk.ac.leedsbeckett.finance.model;

import com.fasterxml.jackson.annotation.JsonIgnore;
import lombok.Data;
import lombok.ToString;

import java.util.ArrayList;
import java.util.List;

import javax.persistence.*;

@Entity
@Data
public class Account {

    private @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    Long id;
    @Column(unique = true)
    private String studentId;
    @OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL, orphanRemoval = true)
    @ToString.Exclude
    @JsonIgnore
    private List<Invoice> invoiceList = new ArrayList<>();
    @Transient
    private boolean hasOutstandingBalance;

    public Account() {
    }

    public Account(String studentId) {
        this.studentId = studentId;
    }
}
```

Implementation of the Account domain class for the Finance microservice

The implementation is quite succinct, but there is a lot going on in terms of annotations. Let's take a closer look at them.

The @Entity annotation is used to tell JPA that this is a domain class or, in other words,

that this class can be mapped to a table on the database. As we mentioned earlier, we will not need to write any SQL, we will use JPA's annotations to handle all our interactions with the database, including schema creation.

The @Data annotation is there for convenience. It uses the Lombok library to generate getters and setters for each field, as well as equals() and hashcode() implementations[2]. If you prefer to write these by hand, you can do that instead and remove the Lombok dependency from your build.gradle. It is a matter of personal choice, we think it looks much cleaner with the annotation 🙂.

Note how the id field is annotated with @Id and @GeneratedValue(strategy = GenerationType.IDENTITY). This tells JPA that the id field should be the primary key of the table, and that this value should be auto-generated as the record's identity.

The next field represents the external student ID which is generated elsewhere. It is annotated with @Column(unique=true) to tell JPA to create a unique constraint on the database. This ensures that the values for each record are unique.

The invoiceList field contains a list of invoices. The @OneToMany annotation tells JPA to implement a one-to-many relationship between Account and Invoice. We also set the fetch type to LAZY and add @ToString.Exclude and @JSONIgnore to the field. This prevents circular referencing between Account and Invoice since we want to be able to fetch the account from an invoice object.

Finally, the hasOutstandingBalance field is annotated as @Transient. This means we do not want to persist this value on the database. Whether the account has an outstanding balance is something that is calculated at runtime. You can see the implementation of this logic in the screenshot below, taken from the AccountService class.

---

[2] For an explanation of getters and setters, have a look at this tutorial.
Equals and hashcode are explained in detail in Java's API documentation.

```
94      private Account populateOutstandingBalance(Account account) {
95          if (account != null) {
96              List<Invoice> invoices = invoiceRepository.findInvoiceByAccount_IdAndStatus(account.getId(), Status.OUTSTANDING);
97
98              if (invoices != null && !invoices.isEmpty()) {
99                  account.setHasOutstandingBalance(invoices
100                         .stream()
101                         .anyMatch(invoice -> invoice.getStatus().equals(Status.OUTSTANDING)));
102             }
103         }
104         return account;
105     }
106 }
```

Implementation of the populateOutstandingBalance method in the AccountService class

In our Finance example, Spring Data JPA is an abstraction over Hibernate, Java's most popular Object-Relational Mapping (ORM) tool. Although Hibernate comes as the default JPA implementation provider with Spring Data JPA, other compatible options such as EclipseLink can be used instead.

Answer in your log book: what are the advantages of using an ORM instead of embedding SQL statements in the code?

Implementation

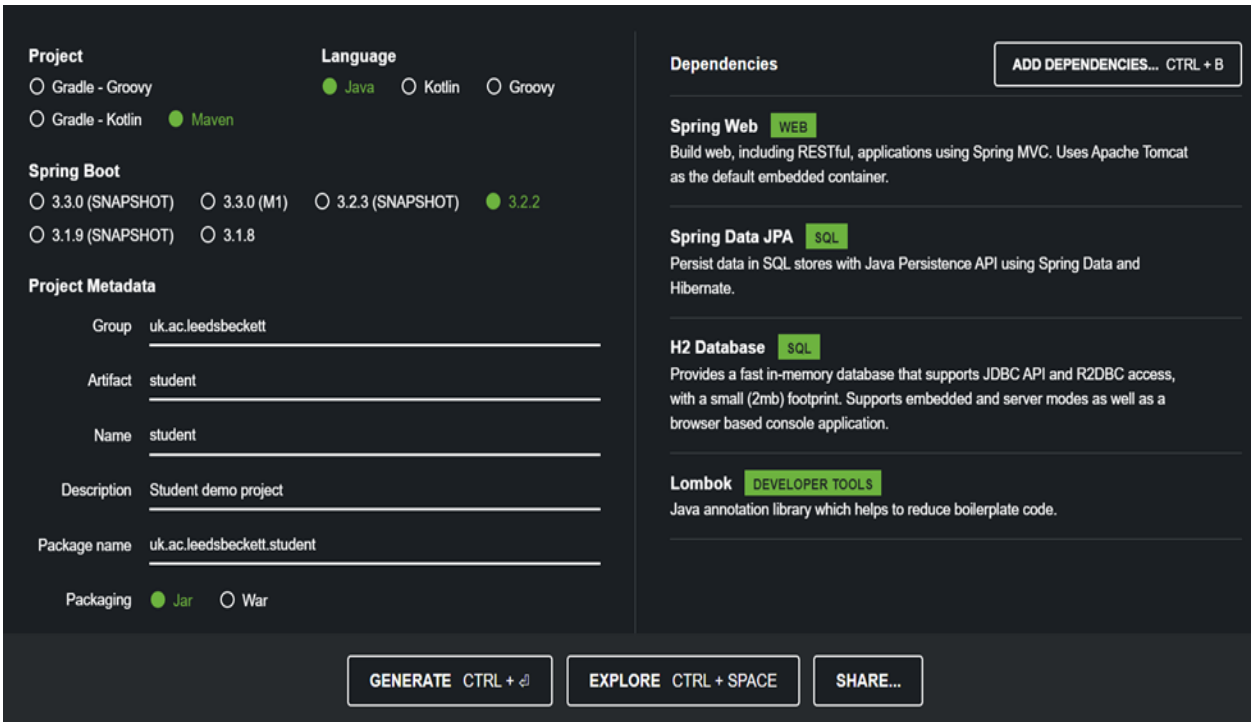
Now it is your turn to work on your Student implementation. We are going to use the H2 in-memory database in this week's lab as it is fast, simple to configure, and works out-of-the-box with Spring Boot. If you wish to experiment with a real database such as MySQL or PostgreSQL (or even a NoSQL alternative), feel free to do so after you have completed the lab.

Answer in your log book: would it be prudent to use the H2 database in a production setting? Why or why not?

Let's start with importing the dependencies we are going to use. If you don't already have a project, create one using the Spring Initializr and add the following dependencies:

- Lombok
- Spring Data JPA
- H2 Database
- Spring Boot Starter Web

Call your application and artifact **student**. Use **uk.ac.leedsbeckett** for the group name and package name. Follow the example below.



Creating a Spring Boot project using Spring Initializr and selecting the desired dependencies

Locate your application.properties file in src/main/resources and add the following configuration:

```
server.port=8090
spring.datasource.url=jdbc:h2:mem:student
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
```

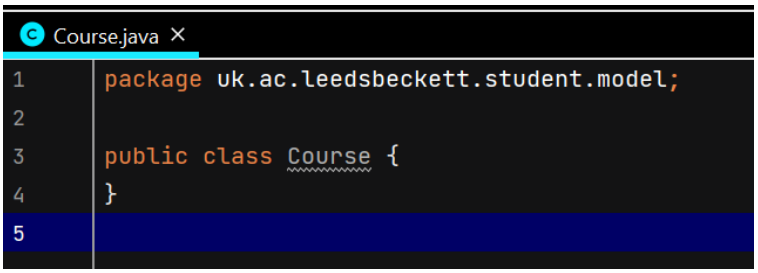
Now we are ready to create our first domain class. Let's start by creating a simplified version of Course.

In your src/main/java directory, create a package called:

uk.ac.leedsbeckett.student.model

This is where our domain model classes will live.

Create a class called Course in the package uk.ac.leedsbeckett.model.



Course domain model class in Student project

Let's go ahead and add the @Entity and @Data annotations, as well as the basic properties of our Student class. Aim for something like this to start with:

```
package uk.ac.leedsbeckett.student.model;


import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import lombok.Data;


@Entity
@Data
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    private String description;
    private Double fee;
}
```

Now that we have our first domain class, we can leverage the power of JPA to handle:

- creating a new course,
- updating an existing course,
- deleting a course, and
- finding a course.

All with a very simple interface implementation. Create an interface called CourseRepository in the uk.ac.leedsbeckett.student.model package.

```
1  package uk.ac.leedsbeckett.student.model;
2
3  public interface CourseRepository {
4  }
```

CourseRepository interface created

Ensure your interface extends JpaRepository. The generic arguments <Course, Long> represent the data type of the repository class (in this case, Course) and the data type of the id for that class (in this case, Long).

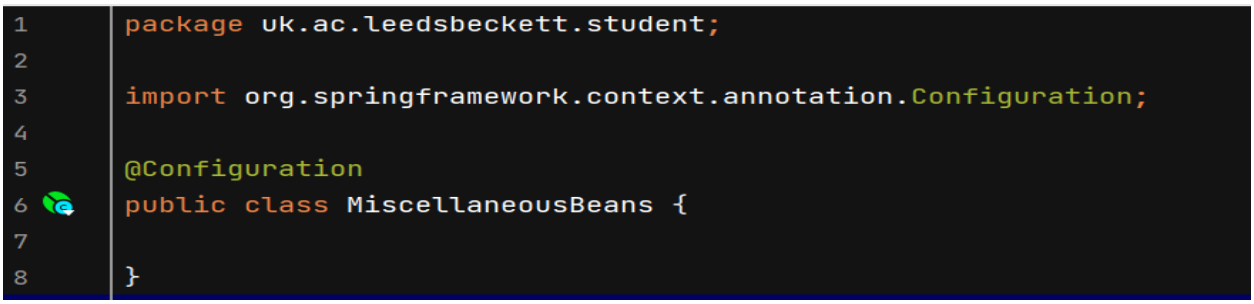
```
1  package uk.ac.leedsbeckett.student.model;
2
3  import org.springframework.data.jpa.repository.JpaRepository;
4
5  public interface CourseRepository extends JpaRepository<Course, Long> {
6  }
7
```

CourseRepository interface extending JpaRepository

This is all you need to do to implement basic CRUD functionality using Spring Data JPA!

Let's write some code to test our implementation.

Create a class called MiscellaneousBeans **in the uk.ac.leedsbeckett.student package**, and annotate it as a configuration class.



```
1  package uk.ac.leedsbeckett.student;
2
3  import org.springframework.context.annotation.Configuration;
4
5  @Configuration
6  public class MiscellaneousBeans {
7
8  }
```

Creation of a MiscellaneousBeans configuration class

Our configuration class will be automatically scanned by Spring during startup[3]. All we need to do now is create a few objects.

Create a method called initDatabase() that accepts an instance of CourseRepository. Follow the example below.

```
package uk.ac.leedsbeckett.student;


import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import uk.ac.leedsbeckett.student.model.Course;
import uk.ac.leedsbeckett.student.model.CourseRepository;

@Configuration
public class MiscellaneousBeans {

    @Bean
    CommandLineRunner initDatabase(CourseRepository courseRepository) {
        return args -> {
            Course myCourse = new Course();
            myCourse.setTitle("SESC");
            myCourse.setDescription("Software Engineering for Service Computing");
            myCourse.setFee(10.00);
            courseRepository.save(myCourse);
        };
    }
}
```

Note: If you get syntax errors on the setters above (setTitle, setDescription and setFee) it is

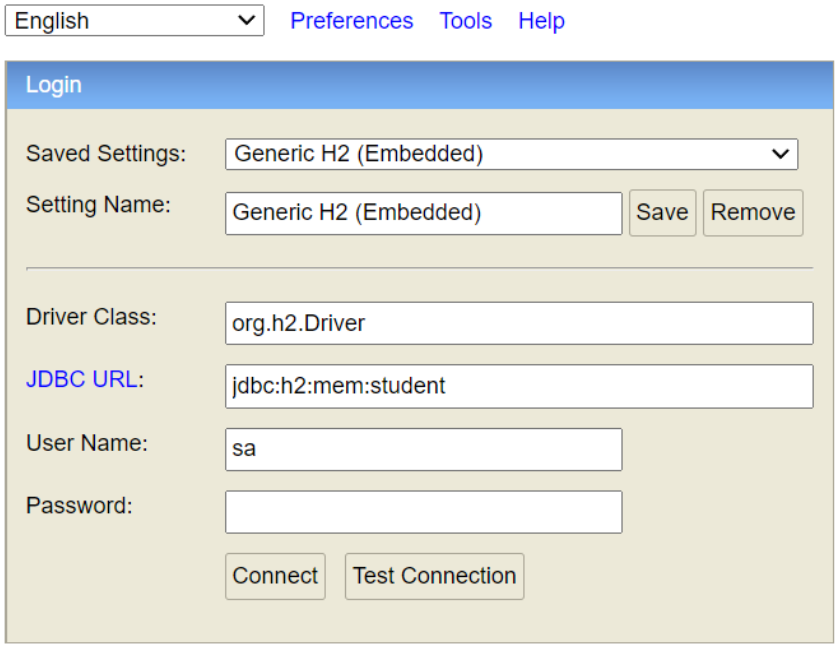
[3] For this to happen, the configuration class must be in the same package as the main application class, or in a subpackage. If you have followed the previous instructions and your main application class is in uk.ac.leedsbeckett.student, the example in the screenshot will work.

because Lombok, which does the tedious task of creating setters and getters for us, is not installed in your IDE. You need to download lombok.jar and run it from the command line, it will detect IDEs and install in them. See https://www.baeldung.com/lombok-ide . You should restart Eclipse and Maven->Update Project. (DM 2/24).

Let's see if the database schema was created and the record inserted for us.

In your browser, navigate to:

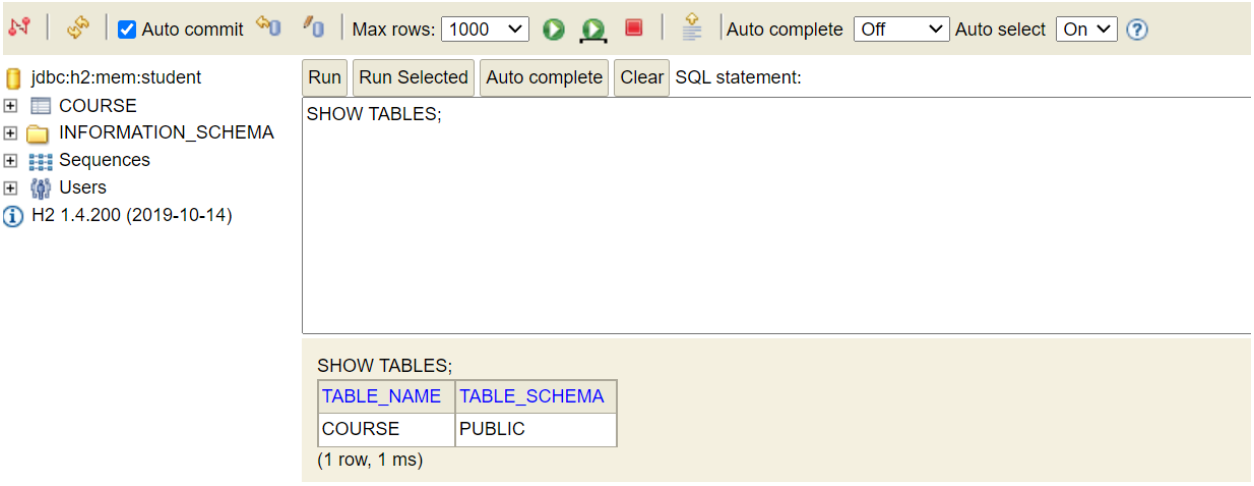
http://localhost:8090/h2-console/



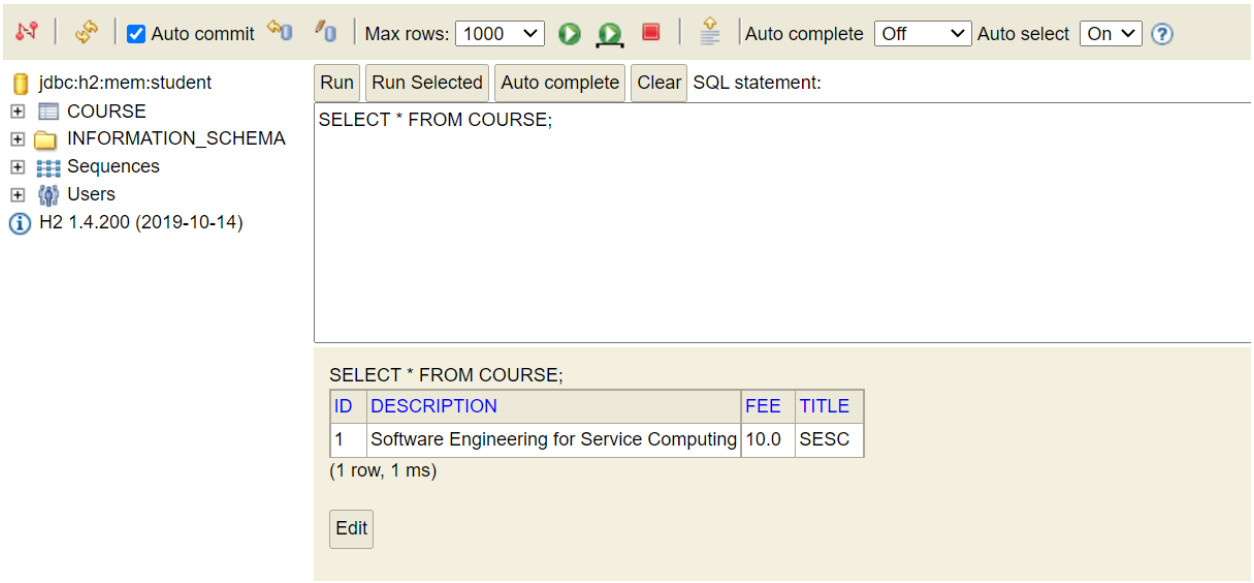
Connecting to the H2 database

Leave the password blank (default) and connect to the database.

You can run some SQL scripts to check that the schema has been created and the database populated programmatically.



Showing a list of tables on the H2 database



Selecting all records from the Course table

Your turn to work: create three more records programmatically.

In your log book: demonstrate that your records have been inserted in the correct database table.

Assignment

Now that you have set up your project to work with Spring Data JPA and have practiced creating domain classes, you can turn your attention to this module's assignment.

Implement the other domain classes identified in your UML class diagram. You may need to do some research on how to map many-to-many relationships using Spring Data JPA.

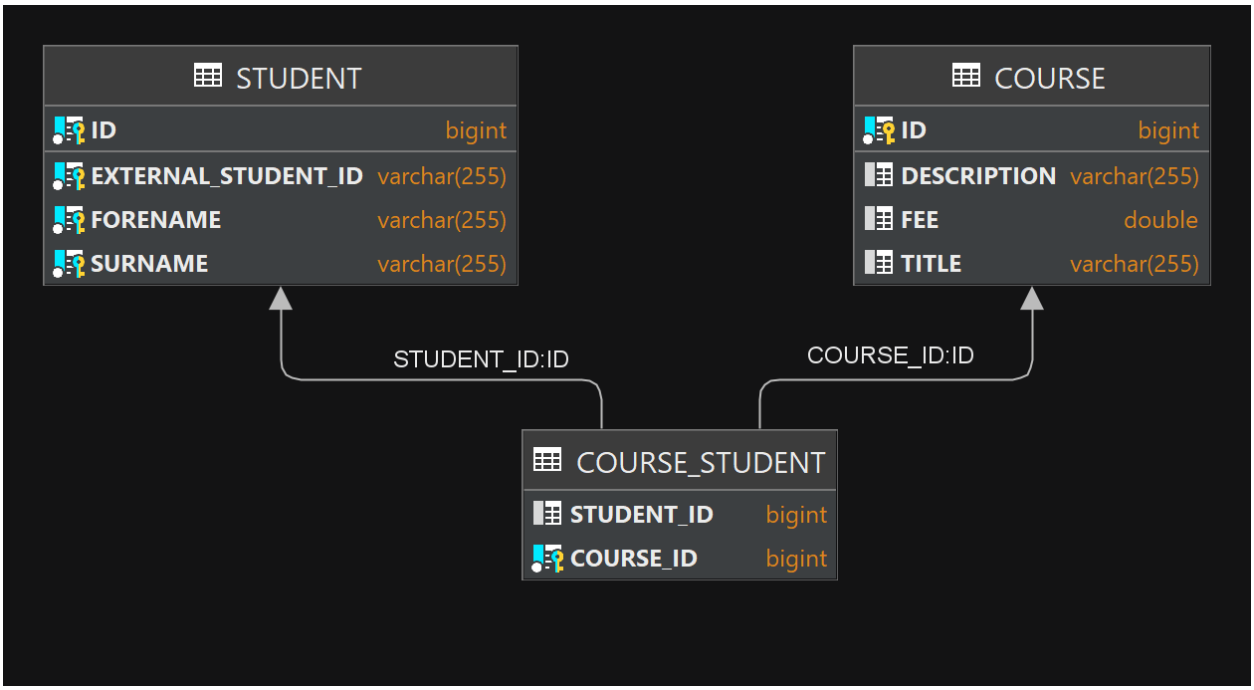
Add more sample data programmatically.

Optional: explore other functionality available through the JpaRepository.

In your log book: demonstrate that the database schema has been created correctly. Add screenshots showing that your sample data has been inserted in the correct tables.

Generate (or manually draw) an entity relationship diagram (ERD) for your application.

The diagram shown below is a starting point. However, you should remember to include **all the domain classes** identified earlier as relevant to your application.



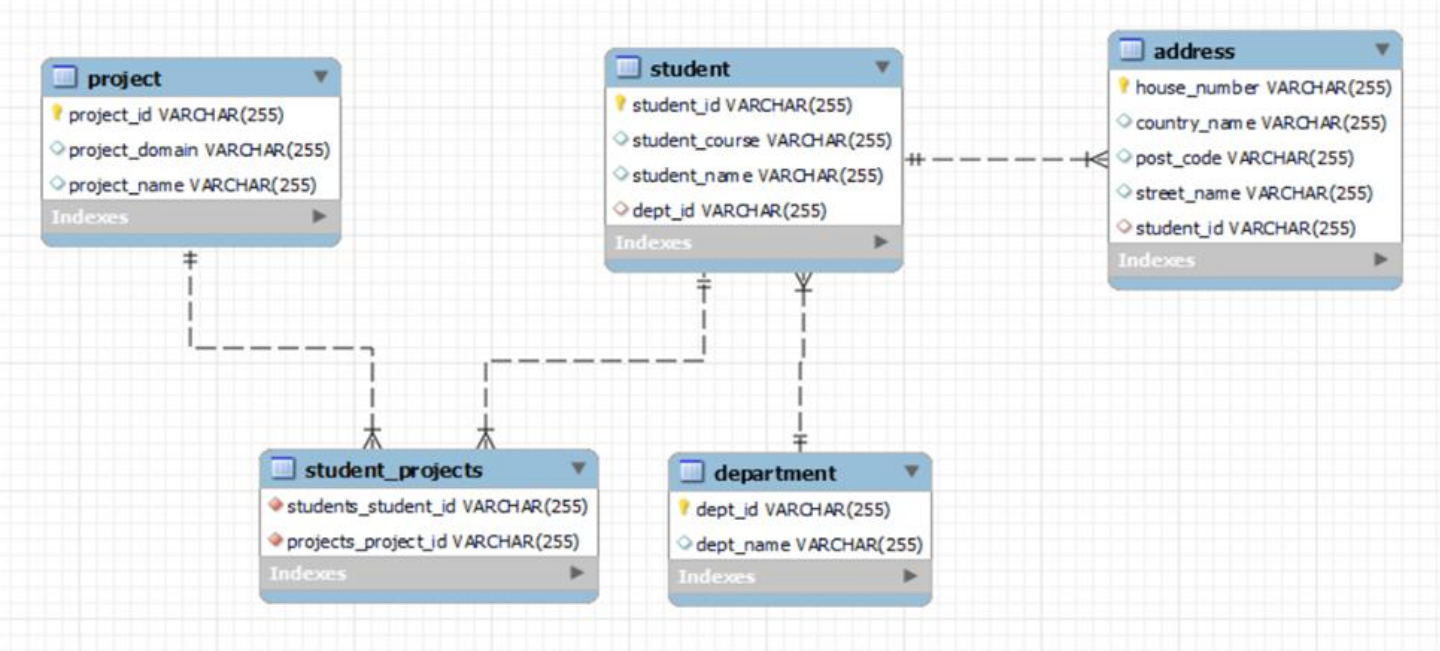
Simplified ERD generated by IntelliJ showing student-course relationship

Paste the full ERD **in your log book.**

Supplementary Practice

Let's practice implementing the following domain classes and their relational mapping as shown

in the diagram.



1. Student and Address Classes

One to One Mapping: student has an address.

One to One Mapping: address belongs to a student.

For your reference, we have provided this use case implementation at https://github.com/kumar-satish/mapping.git

Spring JPA annotation: *@OneToOne*

2. Department and Student Classes

One to Many Mapping: The department has many students,

Spring JPA annotation: *@OneToMany*

Many to One Mapping: students have the same department.

Spring JPA annotation: *@ManyToOne*

3. Project and Student Classes

Many to Many Mapping: students have many projects.

Many to Many Mapping: projects have many students.

Spring JPA annotation: *@ManyToMany*

text

Conclusion

At this point you have:

- Gained theoretical and practical knowledge of ORMs, and reflected on the advantages of using them in software development projects
- Understood the concept of domain model classes and practiced creating them using Spring Data JPA's annotations
- Used Lombok to generate boilerplate getters and setters, equals and hashcode
- Learned how to extend the JpaRepository interface to leverage basic CRUD operations
- Configured a database to persist your application data
- Interacted with the H2 Console to run queries against the database.
- Implemented the domain model for the assignment.

Well done!

There were many new concepts covered this week, particularly if you are new to Java development. You should take the time to work through the examples carefully, and don't be afraid to look things up to gain a deeper understanding of what is happening in the background. Any questions, please feel free to tag us on Discord.