EducationHub is a project that connects schools, teachers, and students. It uses three linked tables to store data about these entities. The tables are linked to each other in different ways. For example, the Teachers and Students table are linked in a many-to-many relationship, which means that a teacher can have many students and a student can have many teachers. The Schools and Teachers table are linked in a one-to-many relationship, which means that a school can have many teachers but a teacher can only belong to one school. The Schools and Students table are also linked in a one-to-many relationship, which means that a school can have many students but a student can only belong to one school.

EducationHub supports the following CRUD (Create, Read, Update, Delete) operations:

- Schools:
    - Create a new school
    - Modify school details
    - Retrieve a list of schools without teachers and students data
    - Retrieve school information by its unique ID, including teachers and students
    - Delete a school by its ID, including teachers and students and implement cascade All students and teachers
- Teachers:
    - Create a new teacher profile and associate it with school by it’s ID
    - Update teacher details based on teacher ID and associated school ID
    - Retrieve a list of all teachers associated with a specific school ID
    - Retrieve teacher information, including linked students, via their ID
    - Delete a teacher using their unique ID, including students, and implement cascade persistence for students.
- Students:
    - Create/Enroll a student under a specific teacher's ID and school Id
    - Modify student information using student id
    - Delete a student using their ID
    - Associate a student to another teacher by teacher ID
    - Delete the relationship between a student and a teacher
    - Retrieve student information with associated teachers and school by student ID.

Action:

1. Create Maven : project build tool, using dependency which is code library. Maven comes with Eclips
2. Create Spring Boot Project=> start.spring.io , build POM file and using the latest version of spring boot, then past it into Maven project and create main application class file
3. What @SpringBootApplication annotation, what this annotation is doing will enable component Scan and auto-configuration
4. What happens with a component scan is Spring will load up our classes and load up the libraries and it will start at the package that Spring application annotation is found and looks at all the sub-packages of that as well.
5. Spring will load up our classes, that mean Spring should manage the class(lifecycle of class) => called a "managed Bean " or Bean
   =>Spring creates a single instance
   => store that instance in object repository we can request that object of that type using @Autowired
6. create controller class

7. Spring component scan will map HTTP requests to a methods that we write

8. we use annotation to tell Spring which HTTP verb to map to the method

9. auto-configuration, spring examines the class path to see what we have loaded up and based on the application functionality or configuration for example if we want a web application we would include a web application container like Tomcat

10. create schema and user, password and add privileges by using MySql Workbench

11. Create …. Entity Classes:

. Define the entity classes that represent the core data in my education hub. School, Student, Teacher.

.Annotate the entity classes with appropriate JPA annotations such as @Entity, @Table, @Id, @GeneratedValue, and relationships like @OneToMany, @ManyToOne, etc.

12. Configure the database connection in `application.yml`. Add JPA and Spring Boot configuration .Start the application and show that the tables are created by JPA and populated by Spring boot

13. Add 3 configuration sections

```
spring:
```

```
   datasource: //which has a connection instruction (username, password and
url
      username: educationhub
      password: educationhub
      url: jdbc:mysql://localhost:3306/educationhub

   jpa: // JPA setup
      hibernate:
         ddl-auto: update
      show-sql: true
      defer-datasource-initialization: true

   sql: //spring.sql: spring data instruction
      init:
         mode: never
```

14. use Dbeaver to verify the tables were created and populated

15. Create DTOs (Data Transfer Objects) when sending data between the client and server. They act like simplified messengers between them. This keeps the conversation clear and avoids issues. DTOs also help in avoiding loops or repetition in classes. This way, when turning data into JSON using JACKSON, there's no mix-up or trouble.
16. Create controller classes to handle HTTP requests and responses. Using @RestController and @RequestMapping annotations to define endpoints also Inject service classes into controllers and call methods to handle requests.also using @Slfj which is a Lombok annotation that automatically generates logging code in the class. It's commonly used in Java classes to simplify the process of adding logging statements without explicitly creating a logger instance.
17. Create service classes Implement service classes that provide business logic and act as an intermediary between the controller and repository/Dao. Using dependency injection to inject the repository interfaces into the service classes.

18. create DAOs (Data Access Objects) that communicate with the database and provide an interface for interaction. Utilize JPARepository to extend its functionality and provide the implementation.

19. Repository Layer:
    a. Create JPA repositories for each entity. These interfaces extend the JpaRepository interface provided by Spring Data JPA. Spring Data JPA provides basic CRUD operations and query methods.
    b. Define custom query methods using method naming conventions or JPQL queries if needed.
20. Service Layer:

a. Implement service classes that provide business logic and act as an intermediary between the controller and repository.
b. Use dependency injection to inject the repository interfaces into the service classes.

21. Controller Layer:
a. Create controller classes to handle HTTP requests and responses.
b. Use @RestController and @RequestMapping annotations to define endpoints.
c. Inject service classes into controllers and call methods to handle requests.

22. Exception Handling:
a. Implement global exception handling using @ControllerAdvice to handle different types of exceptions and return appropriate error responses.

23. Database Configuration:
a. Configure the database connection in application.yml. Provide the database URL, username, and password.

24. Testing:
a. Write unit tests.
b. Use an in-memory database like H2 for testing.
c. Create schema.sql file and data.sql file to use for test
d. Create another configuration file to use for test too(application-test.yaml