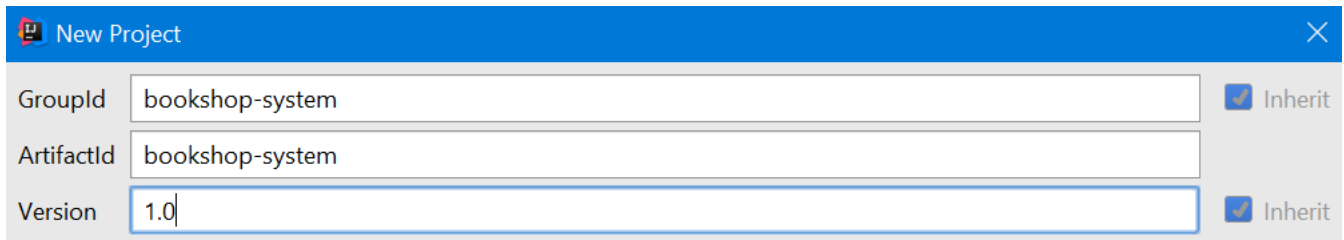


Exercise: Hibernate Code First - Bookshop

This document defines the **exercise assignments** for the ["Databases Advanced – Hibernate" course @ Software University](#).

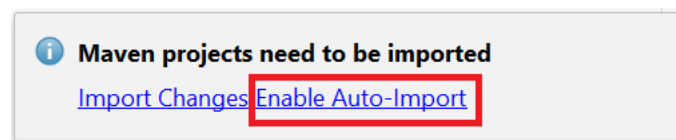
1. Create a Database for Student System using Code First

First of all, create a **new Maven project** and name it accordingly - in our case, **bookshop-system** sounds like a nice name.



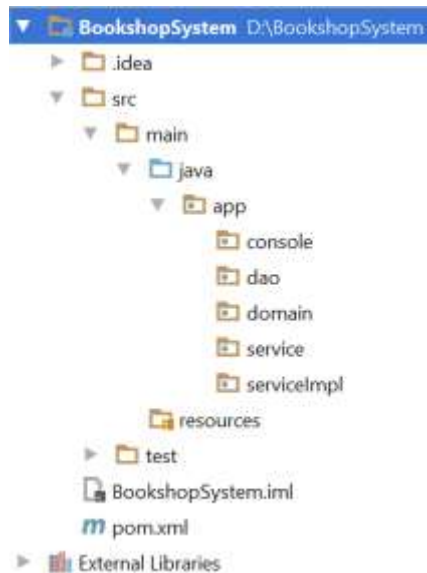
Step 1 – Prepare Environment

When you create new Maven project the IntelliJ idea ask you **enable auto import dependencies**. Make sure you turned that on.



First of all, we should make the structure of our project. In the java package create new package app, where we will keep all of our classes. Inside it add the following packages:

- **domain** – in that package we will keep all of our entities (classes that would represent our models for the tables in the database)
- **dao** – will keep classes from the data access layer (our repositories) which will perform CRUD operations with the database. (DAO stands for Data Access Objects)
- **service** – will keep the interfaces four our services
- **serviceImpl** – will keep the actual implementations of the services
- **console** – here will be the core functionality of the console interface of the project



In the **pom.xml** file set **spring-boot-starter** as parent and include the following dependencies:

- Spring Boot Starter Data JPA
- MySQL Connector

pom.xml
<pre> <parent> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-parent</artifactId> <version>1.4.1.RELEASE</version> </parent> <dependencies> <dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-data-jpa</artifactId> </dependency> <dependency> <groupId>mysql</groupId> <artifactId>mysql-connector-java</artifactId> <version>6.0.4</version> </dependency> </dependencies> </pre>

Also, we need to change the build configuration to target Java 8 instead of Java 5. Add the following code snippet to the **pom.xml** file.

pom.xml
<pre> <build> <plugins> <plugin> <groupId>org.apache.maven.plugins</groupId> <artifactId>maven-compiler-plugin</artifactId> <version>3.5.1</version> <configuration> <source>1.8</source> <target>1.8</target> </configuration> </plugin> </plugins> </build> </pre>

In `src\resources\` package create new **application.properties** file and put the following template in it and provide appropriate data for connection to the database. Set the database name to **bookshop_system** and use your username and password to access the database.

application.properties

```
#Data Source Properties
spring.datasource.driverClassName = com.mysql.jdbc.Driver
spring.datasource.url = jdbc:mysql://localhost:3306/<DATABASE_NAME>?useSSL=false
spring.datasource.username = <DATABASE_USERNAME>
spring.datasource.password = <DATABASE_PASSWORD>

#JPA Properties
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect
spring.jpa.properties.hibernate.format_sql = TRUE
spring.jpa.hibernate.ddl-auto = create

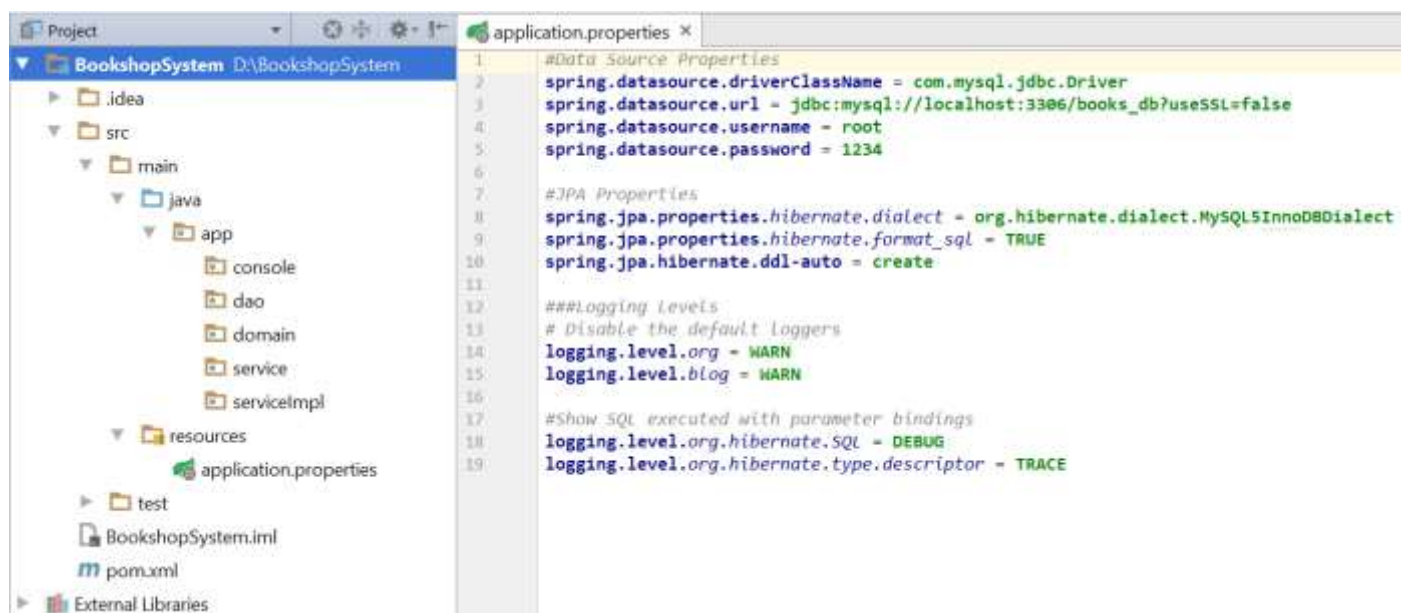
###Logging Levels
# Disable the default loggers
logging.level.org = WARN
logging.level.blog = WARN

#Show SQL executed with parameter bindings
logging.level.org.hibernate.SQL = DEBUG
logging.level.org.hibernate.type.descriptor = TRACE
```

The property **spring.jpa.hibernate.ddl-auto** automatically validates or exports schema DDL to the database. Possible values are:

- **validate** - hibernate only validates whether the table and columns are existing or not. If the table doesn't exist, then hibernate throws an exception. Validate is the default value for hibernate.ddl-auto.
- **update** - hibernate checks for the table and columns. If table doesn't exist, then it creates a new table and if a column doesn't exist it creates new column for it.
- **create** – hibernate first drops the existing table, then creates new table and then executes operations on the newly created table
- **create-drop** - hibernate first checks for a table and do the necessary operations and finally drops the table after all the operations are completed.

For using the **code first approach** to model database schema we should set the value to **create**.

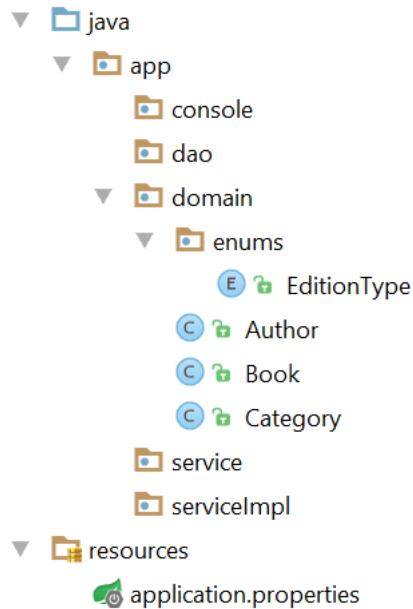


Step 2 - Model the Database

A bookshop keeps **books**. A book can have one **author** (for the sake of simplicity) and many **categories**. And each category can be placed on many books. Let's create a class for each of the main tables.

- **Book** - id, title (between 1..50 symbols), description (optional, up to 1000 symbols), edition type (**NORMAL**, **PROMO** or **GOLD**), price, copies, release date (optional)
- **Author** - id, first name (optional) and last name
- **Category** - id, name

Assume everything **not market optional** is mandatory.



The **classes** should describe with **properties** each of the **table columns**. Do **NOT** forget to create empty constructor for each entity class.

```

@Entity
@Table(name = "authors")
public class Author {
    private Long id;
    private String firstName;
    private String lastName;
    private Set<Book> booksByAuthor;

    public Author() {
    }

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "author_id")
    public Long getId() { return id; }

    public void setId(Long id) { this.id = id; }

    @Column(name = "fisrt_name")
    @Email(minLength = 5)
    public String getFirstName() { return firstName; }

    public void setFirstName(String firstName) { this.firstName = firstName; }

    @Column(name = "last_name", nullable = false)
    public String getLastName() { return lastName; }

    public void setLastName(String lastName) { this.lastName = lastName; }

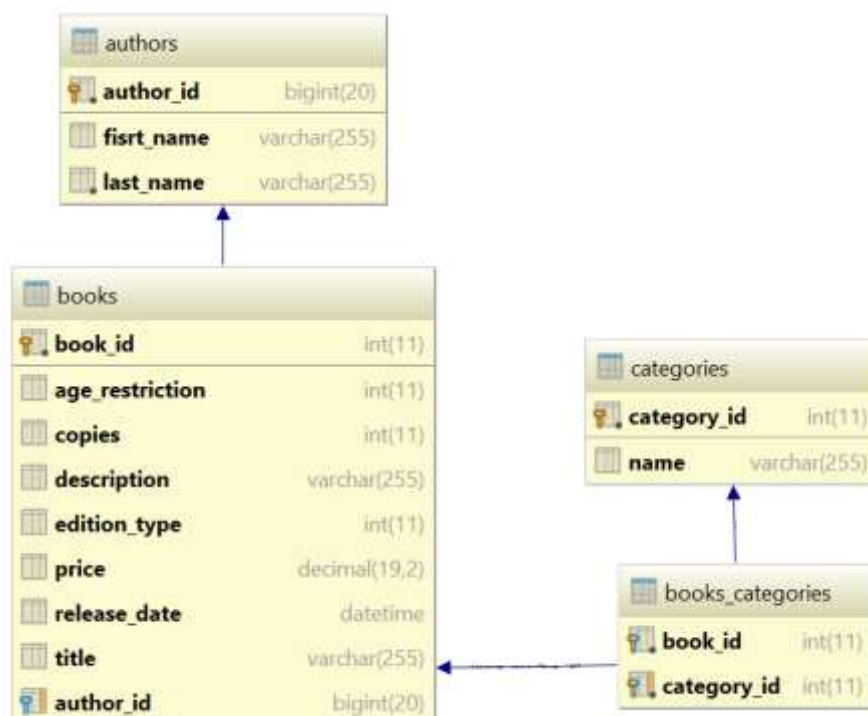
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "author")
    public Set<Book> getBooksByAuthor() { return booksByAuthor; }

    public void setBooksByAuthor(Set<Book> booksByAuthor) { this.booksByAuthor = booksByAuthor; }
}

```

Add constraints and validations for fields as described above. Do the same for the **Book** and **Category** models.

The final schema of the database should look like that:



Step 3 - Create the Data Layer

Once the entities are done, our next step is to write the so-called **Data Layer**. We should create **one interface repository** for each of our entities. Each repository should extend **CrudRepository<Object, ID>**. Remember to put **@Repository** to annotate the interface. For Example:

```
package app.dao;

import app.domain.Author;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface AuthorRepository extends CrudRepository<Author, Long>{
    Author findById(Long id);

    Iterable<Author> findAll();
}
```

With that repository, you can implement methods that could perform CRUD operations with the database. The names of the methods define the type of query. For example, **findById** automatically means that method would return object with given Id, **findAll** method would automatically return all objects of the given type. You can build more complex queries by building more complex method names for example: **Iterable<Author> findByFirstnameOrLastname** would return all authors by given first or last name. For more information check the official documentation of [Spring JPA Repositories](#).

For now, in the **dao package** make the other repositories for Books and Categories with only methods to get object by Id and getting all objects (similar to the **AuthorsRepository**).

Step 4 - Create the Service Layer

Service layer is the **link between** the **data access layer** (repositories) and the **presentation layer** (UI). Here we should make service interface for each of our entities. With that interface, we should be able to interact with the database and perform CRUD operations.

```
package app.service;

import app.domain.Author;

public interface AuthorService {

    void save(Author author);

    void delete(Author author);

    void delete(Long id);

    Author findAuthor(Long id);

    Iterable<Author> findAuthors();
}
```

Once we have interfaces for each of our entities we should implement them. The implementation of the service needs instance of the repository we would use to perform CRUD operations. And we should annotate it with **@Autowired**.

The implementation class of the service implementation should be annotated with **@Service** and **@Primary**.

- **@Service** – that annotation is used by the spring framework to know where the business logic of the application is contained
- **@Primary** – that annotation is used by the spring framework to give preference to that class in case multiple candidates are qualified to autowire a single-valued dependency.

```

@Service
@Primary
public class AuthorServiceImpl implements AuthorService{

    @Autowired
    private AuthorRepository authorRepository;

    @Override
    public void save(Author author) {
        authorRepository.save(author);
    }

    @Override
    public void delete(Author author) {
        authorRepository.delete(author);
    }

    @Override
    public void delete(Long id) {
        authorRepository.delete(id);
    }

    @Override
    public Author findAuthor(Long id) {
        return authorRepository.findById(id);
    }

    @Override
    public Iterable<Author> findAuthors() {
        return authorRepository.findAll();
    }
}

```

Make **interfaces and their implementations** for all of our remaining entities (**Books** and **Categories**). Put the interfaces in **service package** and their implementations in the **serviceImpl package**.

Step 5 - Console Client

Finally, it's time to make our presentation layer (UI). For sake of simplicity we would use command line interface or simply said our beloved console.

In the console package create new class called **ConsoleRunner** that would implement **CommandLineRunner** interface from **org.springframework.boot** package. That method would need instances of every of our services class (and make sure they are annotated with **@Autowired** so they can be injected by the spring framework). We should override the **run()** method and put our core logic of the program inside that method. Remember to put **@Component** before the class declaration so the spring framework should know about that class and would be able to use it.


```

@Component
public class ConsoleRunner implements CommandLineRunner {

    @Autowired
    private AuthorService authorService;

    @Autowired
    private BookService bookService;

    @Autowired
    private CategoryService categoryService;

    @Override
    public void run(String... strings) throws Exception {
        //TODO put core logic of the program here
    }
}

```

Last but not least create **Application** class in the app folder. That class would be **the entry point of our program**. Inside of it just make **main()** method and one single line of code is required here:

SpringApplication.run(Application.class, args)

The final touch is to annotate the Application class with **@SpringBootApplication** so the spring framework should know from where to start executing the code.

```

1 package app;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class Application {
8     public static void main(String[] args) {
9         SpringApplication.run(Application.class, args);
10    }
11 }

```

Step 6 - Make Changes to the Local Model

Let's add a new property to the **Book** class - **AgeRestriction** (MINOR, TEEN or ADULT). To update the schema we should change the value of **spring.jpa.hibernate.ddl-auto** property to **update**, so no data will be lost.

Step 7 - Seed Data into the Database

We have our database up and running. However, there is no data to work with. Let's seed some!

Create **seedDatabase()** method in the **ConsoleRunner** class. That method will fill records in the database.

Use the provided **files (categories.txt, authors.txt, books.txt)** and import the data from them.

Importing Books from File

```

//TODO seed Authors from file authors.txt

//TODO seed categories from file categories.txt

BufferedReader booksReader = new BufferedReader(new FileReader("books.txt"));
String line = booksReader.readLine();
while((line = booksReader.readLine()) != null) {
    String[] data = line.split("\\s+");

    int authorIndex = random.nextInt(authors.size());

```



```

Author author = authors.get(authorIndex);
EditionType editionType = EditionType.values()[Integer.parseInt(data[0])];
SimpleDateFormat formatter = new SimpleDateFormat("d/M/yyyy");
Date releaseDate = formatter.parse(data[1]);
int copies = Integer.parseInt(data[2]);
BigDecimal price = new BigDecimal(data[3]);
AgeRestriction ageRestriction = AgeRestriction.values()[Integer.parseInt(data[4])];
StringBuilder titleBuilder = new StringBuilder();
for (int i = 5; i < data.length; i++) {
    titleBuilder.append(data[i]).append(" ");
}
titleBuilder.delete(titleBuilder.lastIndexOf(" "), titleBuilder.lastIndexOf(" "));
String title = titleBuilder.toString();

Book book = new Book();
book.setAuthor(author);
book.setEditionType(editionType);
book.setReleaseDate(releaseDate);
book.setCopies(copies);
book.setPrice(price);
book.setAgeRestriction(ageRestriction);
book.setTitle(title);
//TODO add random categories for current book

bookService.save(book);
}

```

Step 8 - Write Queries

Now, let's leave something for you. Write the following programs that:

1. Get all **books** after the **year 2000**. Print only their **titles**.
2. Get all **authors** with at least **one book with release date before 1990**. Print their **first name** and **last name**.
3. Get all **authors**, ordered by the **number of their books** (descending). Print their **first name**, **last name** and **book count**.
4. Get all **books** from author **George Powell**, ordered by their **release date** (descending), then by **book title** (ascending). Print the book's **title**, **release date** and **copies**.

Step 9 - Related Books

Let's say at one point we decide that **books** should have **related books** - i.e. a book has many related books and each related book has related books as well.

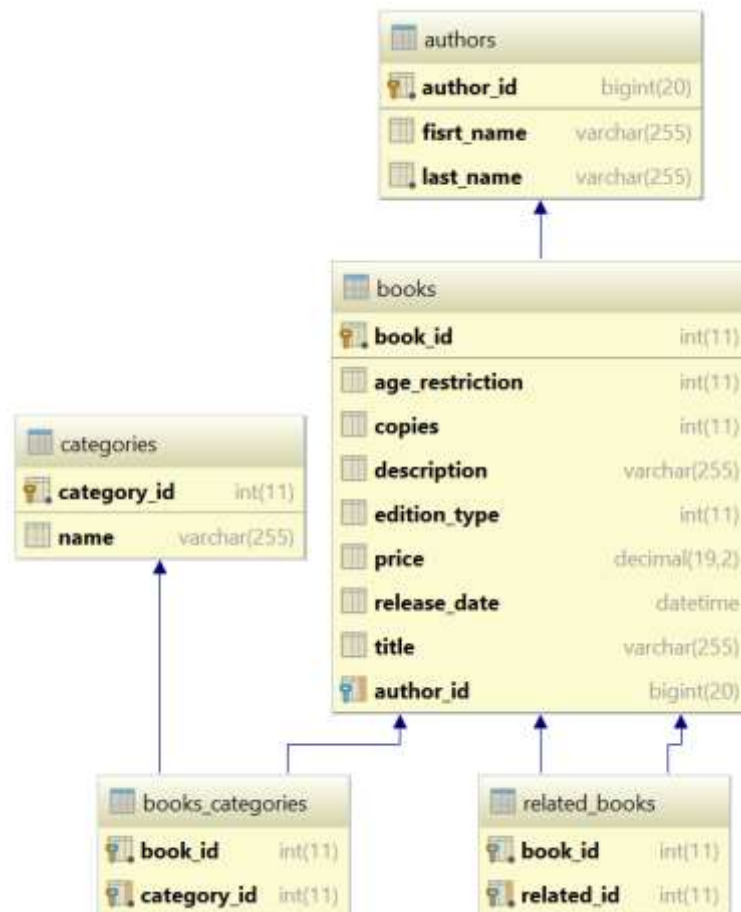
Go to the **Book** class and add field with getter and setter **Set<Book> relatedBooks**.

Use the **@ManyToMany** annotation with **CascadeType.ALL** and **FetchType.EAGER**. Also, use **@JoinTable** annotation and set its properties to:

- the **table name** to **related_books**
- **join columns** to **book_id**
- **inverse join columns** to **related_id**

```
@ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
@JoinTable(
    name = "related_books",
    joinColumns = @JoinColumn(name = "book_id"),
    inverseJoinColumns = @JoinColumn(name = "related_id")
)
public Set<Book> getRelatedBooks() {
    return relatedBooks;
}
```

Restart the program and if no exception is thrown, the new database schema should now look as follows:



And finally, let's test the **RelatedBooks** functionality it. **Get 3 books** from the database and set them as **related**.

Sample Code	Sample Output
<pre> List<Book> books = (List<Book>) bookService.findBooks(); List<Book> threeBooks = books.subList(0, 3); threeBooks.get(0).getRelatedBooks().add(threeBooks.get(1)); threeBooks.get(1).getRelatedBooks().add(threeBooks.get(0)); threeBooks.get(0).getRelatedBooks().add(threeBooks.get(2)); threeBooks.get(2).getRelatedBooks().add(threeBooks.get(0)); //save related books to the database for (Book book : threeBooks) { bookService.save(book); } for (Book book : threeBooks) { System.out.printf("--%s\n", book.getTitle()); for (Book relatedBook : book.getRelatedBooks()) { System.out.println(relatedBook.getTitle()); } } </pre>	<pre> --Absalom A che punto A" la notte After Many a Summer Dies the Swan --A che punto A" la notte Absalom --After Many a Summer Dies the Swan Absalom </pre>