Database Interaction – Read and Insert

Contents

Introduction

Up to this point, all of our data has been retrieved and stored in a data structure. This is in-memory and will be lost when the Java virtual machine is closed. This isn't good for most system, so we need to persist the data. For that, we are going to introduce a database (specifically a relational database).

We will use MariaDB as the database. This tutorial will assume that you know basic SQL for defining how data is stored (Data Definition (DDL)) and manipulating that data (Data Maniulation (DML)) in MariaDB.

To add database interaction to our application, we need to do the following:

- Add database connection libraries to our application so that Spring Boot can send and receive data from a relational database.
- Add a database-specific driver to our application, so that generic database calls can be translated into those required by the database of choice.
- Configure our application with the specific database connection properties.
- Write SQL to create tables and test data.
- Configure Spring Boot to initialise the database when the application starts. This is optional.
- Amend the application code so that the data is retrieved/stored into the database.

Add Database Connection Libraries

To connect a Java application to a relational database, we use the standard Java library wrapped in a Spring Boot library. The standard library is called JDBC (Java DB Connectivity) and it provides DB agnostic classes. This basic library can be difficult to work with and requires the developer to manage the connections to the database (especially when it comes to closing the connection). Poor connection management can quickly bring an application to its knees.

Spring Boot uses the JDBC template library. This absolves the developer from connection management (and similar issues) allowing the developer to focus on the SQL to issue and the mapping of tabular data into objects.

To add this to our Spring Boot application, we need to add a single line to our `build.gradle` file.

```
implementation 'org.springframework.boot:spring-boot-starter-jdbc'
```

Add Database Driver

JDBC is database agnostic, but each database has its own low-level API. So whereas our application may send a standard SQL query to any relational database (within reason), how the SQL passes over the network to the database and how the data is returned is database-specific. A driver provides the adaptation between generic and specific. We need to add the driver again using Gradle.

```
//from https://mariadb.com/kb/en/java-connector-using-gradle/
implementation 'org.mariadb.jdbc:mariadb-java-client:3.4.1'
```

Add Database Properties to Application

In this section, we will provide configuration to our application so that it knows which database to connect to. We know it will be a MariaDB database, but there are lots of those out there, so we need to tell the application which one – and provide credentials.

Before we can do that, we need to create a database (and schema) to connect to.

Create a database

Using a database client of your choice (e.g., DataGrip), create a database schema on your localhost and call it 'takeaway'. This will be a schema in your local MariaDB database which should have the standard NSA credentials.

```
create schema takeaway;
```

Add connection properties

Now that we have a database schema, we can tell Spring Boot to connect to it. We do this by adding properties to the `application.properties` file.

```
spring.datasource.url=jdbc:mariadb://localhost:3306/takeaway
spring.datasource.username=root
spring.datasource.password=comsc
```

These three lines tell Spring Boot the network location and port, plus the schema name. The "`mariadb`" part also tells Spring Boot which driver to use. In same examples, you will see the driver class provided as an additional property – but this is not usually required.

Note: this configuration of username and password is configured to the university supplied machines and the pre-installed and configured MariaDB installation.

Initialise the database

We can't really test the application if there are no tables in the schema, and there is no data in the tables. We can do this manually outside the application, but it's quite nice to know that the database will be in a certain state whenever to run the application. The problem is that this can be a bit slow (we'll discuss another option later).

We will adopt a partial approach. We've created the schema manually and we've pointed our application at that schema. We will create the tables and populate them with data with the application start-up.

Create the tables

Our first task is to create some tables, or in this case, a table.

Create a file called `schema.sql` in the `main/resources` directory and write this code.

```
drop table if exists menu_item;
create table if not exists menu_item (
   id bigint auto_increment primary key,
   name varchar(128),
   price_in_pence bigint,
   allergens varchar(128),
   description varchar(128),
   is_vegetarian boolean,
   is_vegan boolean
) engine=InnoDB;
```

Notice how we're using almost the same names for columns as we did for fields in the class, expect we're using snake case (underscores) rather than camel case (upper case letters). This will become more relevant as you explore alternative ways to persist data. For now, it's just a naming convention that makes our code more idiomatic.

Populate the tables

Now that we have a table, let's populate the table with some data.

Create a file called `data.sql` in the `main/resource` directory and write this code.

```
delete from menu_item;
insert into menu_item (name, price_in_pence, allergens, description,
is_vegetarian, is_vegan) values ( 'Egg and Bacon', 100,'egg','' ,false, false);
insert into menu_item (name, price_in_pence, allergens, description,
is_vegetarian, is_vegan) values ( 'Egg Sausage and Bacon', 200,'egg','' ,false,
false);
insert into menu_item (name, price_in_pence, allergens, description,
is_vegetarian, is_vegan) values ( 'Egg and Spam', 200,'egg','' ,false, false);
insert into menu_item (name, price_in_pence, allergens, description,
is_vegetarian, is_vegan) values ( 'Egg, Bacon and Spam', 250,'egg','' ,false,
false);
insert into menu_item (name, price_in_pence, allergens, description,
is_vegetarian, is_vegan) values ( 'Egg, Bacon, Sausage and Spam', 300,'egg',''
,false, false);
insert into menu_item (name, price_in_pence, allergens, description,
is_vegetarian, is_vegan) values ( 'Spam, Bacon, Sausage and Spam', 350,'','',
,false, false);
insert into menu_item (name, price_in_pence, allergens, description,
is_vegetarian, is_vegan) values ( 'Spam Egg Spam Spam Bacon and Spam',
500,'egg','' ,false, false);
insert into menu_item (name, price_in_pence, allergens, description,
is_vegetarian, is_vegan) values ( 'Egg and Quorn Spam', 600,'egg','' ,true,
false);
insert into menu_item (name, price_in_pence, allergens, description,
is_vegetarian, is_vegan) values ( 'Quorn Spam and Quorn Bacon', 550,'','' ,true,
false);
```

If you were to start the application now with `bootrun`, these scripts would not be run and the database would be unpopulated. We need to tell Spring Boot to initialize the database. To do this, we add another property to `application.properties`.

```
spring.sql.init.mode=always
```

This is always a choice. It is quite possible that you will prefer to manage the data outside of the application. There are scenarios where initialization is a good thing (having the system in a known state

before testing or a demonstration would be examples), but others where the database would be better left as it is after each step (exploratory testing). Spring Boot doesn't make you choose one. You can set it up to do different things in different situations – but that will be explained later (for the impatient, lookup Spring Profiles).

Now run the system. Make an amendment to the data in the table and commit the change. Stop and restart the server and then check the state of the data. Has it been reinitialized?

Amend the application code

Now we can actually amend the application code to use our database. We'll do this by introducing a new type of component known as a repository. Our repository will be called by the service. The service will (at this stage) add little value and will pass the data from the repository through to the controller.

This will take a few steps.

Create an interface for the repository

Start by creating a Java interface for the repository. This will mirror the methods offered by the service.

Create an interface called "`MenuRepository.java`" in the `uk.ac.cf.spring.takeaway/menu` package.

```
public interface MenuRepository {
    List<MenuItem> getMenuItems();
    MenuItem getMenuItem(Long id);
    void addMenuItem(MenuItem menuItem);
}
```

Inject the interface into the service

The next step is to replace the service implementation with one that will delegate the calls to the repository. Let's also rename the service implementation (as we're not using a list anymore). You can see the change to the constructor. TODO – change the name of the class yourself.

We'll use dependency injection again.

```
private MenuRepository menuRepository;

public MenuServiceImpl(MenuRepository aMenuRepository) {
    this.menuRepository = aMenuRepository;
}
```

Now we have a `MenuRepository` component that we can call. Rewrite the service methods to call the repository.

```
public List<MenuItem> getMenuItems() {
    return menuRepository.getMenuItems();
}

public MenuItem getMenuItem(Long id) {
```

```
    return menuRepository.getMenuItem(id);
}

public void addMenuItem(MenuItem menuItem) {
    menuRepository.addMenuItem(menuItem);
}
```

Implement the repository

Finally, we need to implement the repository using JDBC and some SQL.

Firstly, ask Spring to provide an instance of the `JdbcTemplate` component. This is dependency injection again, but this time we're asking Spring to provide a component from the framework rather than one of our own.

Guess what? This is what Spring was doing when we asked for `Model` and `BindingResult` objects earlier.

The repository also needs a way to convert the data that will come from JDBC into our defined objects. For this, we can use a `RowMapper` class.

```
@Repository
public class MenuRepositoryImpl implements MenuRepository {
    private JdbcTemplate jdbc;
    private RowMapper<MenuItem> menuItemMapper;

    public MenuRepositoryImpl(JdbcTemplate aJdbc) {
        this.jdbc = aJdbc;
        setMenuItemMapper();
    }
```

The `@Repository` annotation tells Spring Boot that this is a Spring component of stereotype "repository". This is useful later on when we do testing. It also means that the class is now a candidate for injection.

The class implements the interface that we created earlier, so Spring can provide an instance of this class where a dependency on the stated interface is specified in a constructor.

In the constructor, we ask Spring to provide an instance of the `JdbcTemplate` class. This is provided by Spring Boot and gives us a high-level API to contact the database.

We also call a method called `setMenuItemMapper`. This will set up our mapper object that we can use later.

```
private void setMenuItemMapper() {

    menuItemMapper = (rs, i) -> new MenuItem(
            rs.getLong("id"),
            rs.getString("name"),
            rs.getLong("price_in_pence"),
```

```
            rs.getString("allergens"),
            rs.getString("description"),
            rs.getBoolean("is_vegetarian"),
            rs.getBoolean("is_vegan")
            );
}
```

The mapper uses a functional style. The `JdbcTemplate` will call the mapper for each row in the `ResultSet` that comes back from the database. It will then create a new `MenuItem` object by calling the constructor and passing in the values of the fields retrieved from the columns in that row. It accesses the columns by name. It is necessary to specify the type of data that is expected in the column.

We'll see the mapper being used shortly.

Get all the menu items

Let's first implement the method that gets all of the menu items from the database. The SQL for this is quite straightforward:

```
select * from menu_item
```

Let's pass this SQL into a method on `JdbcTemplate` and then map the returned data into a list of `MenuItem` objects.

```
public List<MenuItem> getMenuItems() {
    String sql = "select * from menu_item";
    return jdbc.query(sql, menuItemMapper);
}
```

The work for the developer is quite small. They can focus on two things; the query and the mapping. The `JdbcTemplate` can do all the work if it is given those two things. This is a good example of abstraction because the code is hiding unnecessary detail from the developer.

Get a single menu item

We can use a slightly different method to get a single menu item.

```
public MenuItem getMenuItem(Long id) {
    String sql = "select * from menu_item where id = ?";
    return jdbc.queryForObject(sql, menuItemMapper, id);
}
```

In this method, we again provide the query and the mapper, but we can also use Java's support for optional parameters and provide as many parameters as required. Each parameter is passed into the query wherever there is a "?". In this case, the `id` that is passed in becomes part of the `where` clause.

It is worth noting here that the `queryForObject` method is very precisely named. This method will return one and only one object. It will throw an exception if the are zero or more than one objects returned by the query.

This presents a number of options:

- Let the exception be thrown and catch it in the service or higher.
- Catch the exception in the method and return an `Optional` object if there are no rows, or throw another exception if there are more than one.
- Replace the method with `query`, return a list and check the size of the list.

All of these have pros and cons. We'll return to error handling later on.

Save a new menu item

We are not limited to reading data using SQL. We can use any SQL command. Here, we will use an `insert` statement to add a menu item.

```
public void addMenuItem(MenuItem aMenuItem){
    String menuInsertSql =
            "insert into menu_item " +
                    "(name, price_in_pence, allergens, description,
is_vegetarian, is_vegan)" +
                    " values (?,?,?,?,?,?)";
    jdbc.update(menuInsertSql,
            aMenuItem.getName(),
            aMenuItem.getPriceInPence(),
            aMenuItem.getAllergens(),
            aMenuItem.getDescription(),
            aMenuItem.getIsVegetarian(),
            aMenuItem.getIsVegan()
    );
}
```

The `insert` statement has a number of placeholder question marks in the `values` clause. When we call the `update` method, we pass the SQL and the values for those parameters from the new `MenuItem` object.

Test the application

With all these changes in place, you can test the application. When running the application, use your preferred DB tool to update values in the table and then refresh your page. Do you see the changes being reflected?

Summary

In this tutorial, we've introduced database interaction using Spring Boot's JdbcTemplate that wraps the JDBC library of Java. We've used dependency injection to handle the dependency between the repository and the service. The service isn't doing a lot at the moment, but consider what might happen

if a piece of functionality needed access to distinct datasets. Each dataset could be isolated and managed by a repository, and the service would coordinate and monitor the changes. There are other reasons for having a service as well such as data transformation.

We now have the basics of an end-to-end architecture for Spring Boot applications.

Templates, Controllers, Services, Repositories, Database.