Lab 3 Multi-layer web applications with Spring Boot

Updated: 2023-10-24.

Introduction to the lab

Learning outcomes

- Develop web projects with Spring Boot. Create and persist entities into a relational database using Spring Data.
- Deploy Spring Boot application in Docker.

References and suggested readings

- "7 Things to Know for Getting Started With Spring Boot"

Submission

This is a two-week lab. You may submit as you go but be sure to complete and submit all activities up to 48h after the second class.
Remember to create the "lab3" folder in your personal Git repository. Be sure to create and update the lab "notebook, e.g. in 🗁lab3/README.md . [The notebook is where you take notes of quick reference info, links and visuals, so you can study from this content later; it is an import part of your submission.]

3.1 Accessing databases in SpringBoot

The Jakarta Persistence API (JPA) defines a standard interface to manage data over relational databases; there are several frameworks that implement the JPA specification, such as the Hibernate framework. JPA offers a specification for ORM (object-relational mapping).
Spring Data uses and enhances the JPA. When you use Spring Data your Java code is independent from the specific database implementation[1].

a) Create a new project in the Spring Initializr, add all these "starter" dependencies to the project:
→ **Spring Web**, **Thymeleaf**, **Spring Data JPA**, **H2 Database**, and **Validation.**

b) Complete the guided exercise available from "Spring Boot CRUD Application with Thymeleaf".

Note 1: some databases may reserve the term "user" and the mapping of the User class into the "user" table will generate errors. You may specify a non-problematic table name for the mapping, e.g.: @Entity(name = "tbl_user")
Note 2: there is a reference implementation in the Git repository referred in the article, if needed. You should, however, create the required entities/files by hand and look only for the boilerplate code as needed.
Note 3: the solution code available may not use the namespaces appropriate for your configuration. For example, if using Spring Boot 3.x, expect "import jakarta.persistence.Entity;" rather than "import javax.persistence.Entity;".

[1] If you avoid native constructions; otherwise, it would be "almost" independent from the underlying database.

Run the application and be sure to insert some sample data.

c) Walkthrough the available solution and answers these questions:

- The “UserController” class gets an instance of “userRepository” through its constructor; how is this new repository instantiated?
- List the methods invoked in the “userRepository” object by the “UserController”. Where are these methods defined?
- Where is the data being saved?
- Where is the rule for the “not empty” email address defined?

d) Extend the solution by adding a new field to the User entity (e.g.: Phone) and refactor the code accordingly.

3.2 Multilayer applications: exposing data with REST interface

a) In this exercise you will need an instance of MySQL server (stick with **version 5.7**) to store Employee information.

Consider using a Docker container to host the database server. Is this example, we are mapping the container standard MySql port (3306) into a different one in our host.

```
$ docker run --name mysql5 -e MYSQL_ROOT_PASSWORD=secret1 -e MYSQL_DATABASE=demo -
e MYSQL_USER=demo -e MYSQL_PASSWORD=secret2 -p 33060:3306 -d mysql/mysql-
server:5.7
```

Be sure to complete the following tasks:

b) Crete the SB project for a basic Employee management application, with Spring Initializr; add the “starter” dependencies for:

→ **Spring Web**, Spring Data **JPA**, **MySQL** driver, **DevTools** and **Validation.**

The following tasks closely follow this guide. Fell free to follow the tutorial along.

c) Create the Employee **entity**. Note the rules in the definition of this entity: unique id, non-empty name, unique email).

d) Create a “**Repository**” of Employees.

e) Create a “(REST) **controller**” to expose the Employee entity.

f) Be sure to define the database **connection properties** in the **application.properties** resource file. E.g:

```
# MySQL
spring.datasource.url=jdbc:mysql://127.0.0.1:33060/demo
spring.datasource.username=demo
spring.datasource.password=secret2
spring.jpa.database-platform=org.hibernate.dialect.MySQL5InnoDBDialect

# Strategy to auto update the schemas  (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto = update
```

g) Test your application endpoints using Postman utility (or curl, from command line).

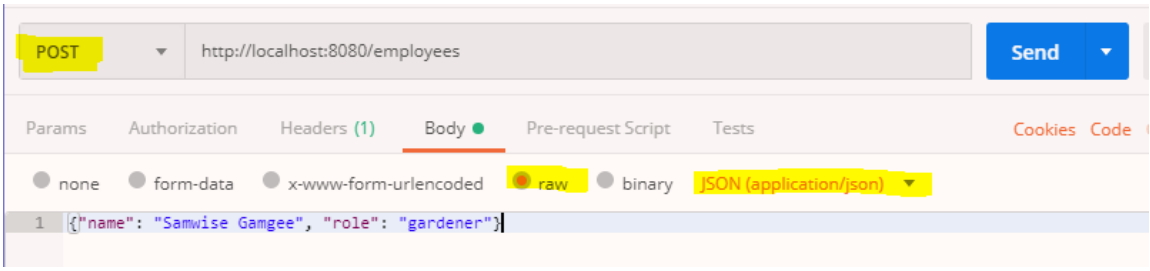


Figure 1- POST method, with JSON payload in the body of the request, to insert a new Employee.

Be sure to list all, filter one, insert and update Employees, i.e., try the **all web methods** available for the Employee resource. [You may wish to take snapshots of key elements in the Postman utility to report in your notebook…]

h) Enhance you REST API with a method to search an employee by email (search by email).

Be sure to extend the Repository with a corresponding query method[2].

Add a filter option to the Employees listing method by using an URL parameter, e.g.:

`http://localhost:8080/api/v1/employees`**`?email=green@mail.com`**

3.3 Wrapping-up and integrating concepts

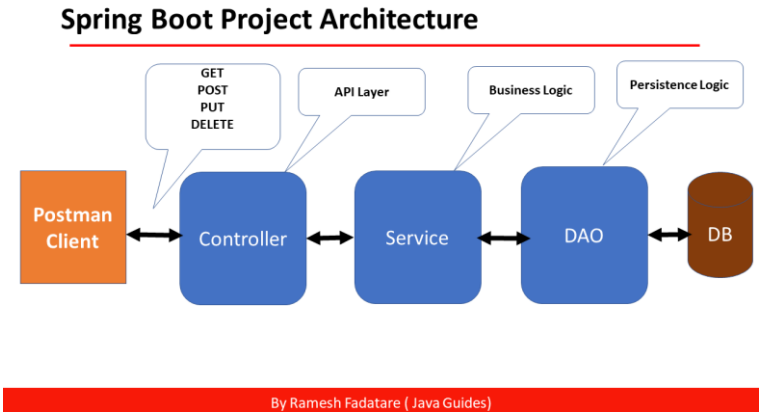
Consider the last exercise from the previous lab (random quotes from movies API).
Let us refactor the project and add the support to:

a) Work with a persistent database of quotes[3].

Consider a few attributes to describe a movie (e.g.: id, title, year)

Note that a "Quote" should refer to a "Movie" (two entities, forming a Many-to-One association).

b) Separate the "boundary" of the problem (@RestController) from the repository, by adding an intermediary @Service component, like in this example. The RestController provides the wiring for HTTP but requires the Service to answer all requests; the Service holds the business logic and interacts with the Repository (or other components) as needed. Note: the RestController should not have any reference to the Repository/data source.



[2] There is a lot of information in the hyperlinked page. Be selective and use just what you need in this context.
[3] Note that the word "show" is reserved word in MySQL; don't use it as an entity/field name (or explicitly map to a different SQL name)

c) Allow shows and quotes to be inserted as well. Respect the separation introduced in previous point.

d) Deploy the quotes application to a Docker container, i.e., *dockerize* your Spring Boot application. Given there are two services (database and the SB application), consider using docker compose (related example).