Lab 3 Multi-layer web applications with Spring Boot

Updated: 2019-10-16.

Introduction to the lab

Leaning outcomes

— Start a web application with Spring Boot and Spring Initializr, combining the appropriate "starter" dependencies.

— Create and persist entities into a relational database using Spring Data.

— Expose a RESTful API, using Spring Annotations.

References and suggested readings

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

Submission

Each student should submit in Moodle evidence of the work in class. A (single) zip file with one folder per section (🗁ies2.1, 🗁ies2.2,...) should be prepared; in the root, there should be a file "readme.md" with the author ID.

You should use the **Readme.md as a notepad**; use this file both to take notes that will help you study later, and to provide a log of the work done.

The items that must to appear in the delivery are marked with 🗐.

3.1 Layered applications in Spring

The Spring Boot site offers an extensive list of guides. You will use the "Building REST services with Spring" for this exercise.

Note: to build Spring Boot applications, consider using a suitable IDE, with the specific plugins available; e.g.: IntelliJ IDEA (Ultimate edition); Eclipse or VS Code with Spring Tools; or Spring Tool Suite (STS, based on Eclipse).

a) This time, we will use the available code and study it (instead of developing). Be sure to get a general understanding about the tutorial example.

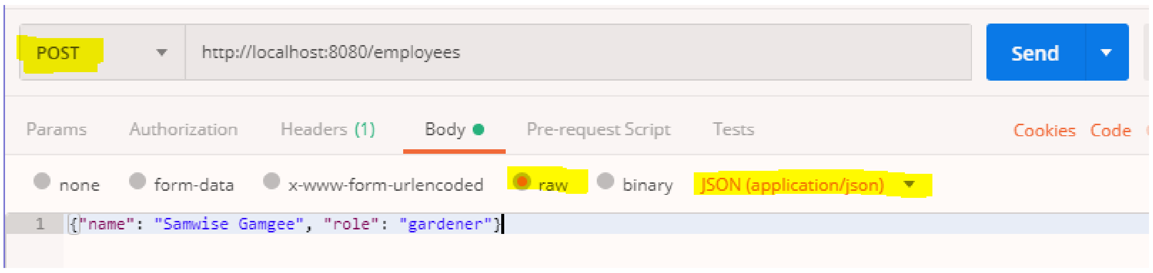
Download and run the code project. (Look for the "rest" sub-project inside the root project and run the *PayrollApplication*).

The application does have web pages; it exposes (REST) endpoints for client programs to connect.

b) Access the REST endpoints using Postman utility.

Be sure to list, filter and insert Employees, i.e., try the several web methods available.

Note: if you try to access from the browser, you may get an error "Could not marshal...". This is because the browser is trying to interpret JSON as XML. You can solve it by explicitly requesting JSON in the API responses (in **all** @GetMapping annotations).



http://localhost:8080/employees

```
{"name": "Samwise Gamgee", "role": "gardener"}
```

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

c) What happens to your data when the application is stopped and restarted? How could you change that behavior?

You don't need to implement a solution; just explain it (with details).

- Create a layered architecture view (diagram), displaying the key abstractions in the solution, in particular: entities, repositories and REST controllers.
- Describe the role of the elements modeled in the previous point.
- Be sure to provide evidence (e.g.: screenshots, JSON results view) that you have successfully used the API to insert new entries.
- Why is that the Employee entity does not have getters and setters defined? (tip: Lombok)

3.2 Accessing JPA Data with REST interface

The Java Persistence API (JPA) defines a standard interface to manage data over relational databases. Most of the work is provided by the adaption layer that implements the JPA and converts objects into relational data and vice-versa. A common implementation is the Hibernate framework. Spring Data will use and enhance the JPA.
Note that you Java code is independent from the specific database implementation.

d) In this exercise you will need an instance of MySQL server (stick with **version 5.x**). If you don't have one already, consider using a Docker container, e.g.:

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

e) The following guide will walk you through the details of preparing the entities, the database and expose RESTful endpoint for remote access, for a basic Employee management application.

This application is functionally like the previous one, but, this time, be sure to start the project from the scratch[1] (instead of downloading the solution).

In the Spring Initializr, add the "starters" dependencies for **Web**, **JPA**, **MySQL**, **DevTools**.

f) Create the Employee **entity**.

g) Create a "**Repository**" of Employees.

h) Create a "(REST) **controller**" to expose the Employee entity.

i) Be sure to define the database **connection properties** in the Application.properties resource file.

[1] You are expected to create the project, classes and methods declaration "by hand". Feel free to copy and paste the methods' body, as they tend to be verbose.

Test your application using *curl* or the Postman tool. Be sure to insert and list content.

Note: the previous tasks are detailed along the tutorial guide, if needed.

j) Enhance you REST API with a new method to retrieve 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=big@here.com`**

🗐 Explain the annotations @Table, @Colum, @Id found in the Employee entity.
🗐 Explain the use of the annotation @AutoWire
🗐 Submit the complete project (after a "clean").

3.3 From data to presentation (Thymeleaf)

In this guide, you will use the same strategy for data management, with Repository and Controller components, to develop an Issues Reporting solution.
This example shows how to connect with the web pages, using a templating system (Thymeleaf).
Note: the authors use the Gradle build system. While you may use it to, we suggest keeping Maven as the build tool.
Using the guide for support, be sure to:

k) Create a project using the dependencies: Spring Web, Spring Data JPA, H2 database, Thymeleaf, DevTools.
l) Create the controller for the web pages flow.
m) Create the Issue entity.
n) Create the Thymeleaf templates and set the required data-binding.
o) Configure the repositories and save date to the H2 database.

Note: the previous tasks are explained along the tutorial guide. You should stop at task #8 in the guide.

p) Setup a Docker container to host a persistent database (e.g.: Postgres, MySql)
Adapt the project to work with this database.
q) Deploy the Issues Reporting application to a Docker container, two.
Prepare a simple docker-compose to lunch and shutdown the system.

🗐 Project with the Issue Tracking application (after a "clean").
🗐 Docker compose configuration file.

[2] There is a lot of information in the hyperlinked page. Be selective and use just what you need in this context.