

Back-End Development

Workshop 1: Spring Boot & HTTP GET

N. Jacobs, J. Pieck, E. Steegmans, B. Van Impe, S. Van Peborgh, V. Witters

Contents

1 INTRODUCTION 3

2 GETTING STARTED 4

2.1 LOOK AT THE PROVIDED CODE 4

3 BOOTSTRAPPING OUR BACK-END 6

4 CODING THE FIRST USER STORY 12

4.1 REPOSITORY 13

4.2 SERVICE 15

4.3 CONTROLLER 17

4.4 TESTING THE API 19

5 CODING MORE ENDPOINTS 20

5.1 PATH VARIABLES 21

5.2 REQUEST PARAMETERS 21

6 ADDITIONAL PRACTICE 22

6.1 OPTIONAL DOMAIN EXTENSION 22

1 Introduction

In this workshop we will create our first back-end application and code our first HTTP endpoints. We will start from an existing collection of Java classes that contain the concepts and logic that our back-end will operate on (we call such a collection classes the *domain* of our back-end).



The concepts covered include:

- Java
  - Classes, Inheritance, Collections, Exceptions, Optionals, …
  - Unit Testing
- Maven
  - Dependencies
  - Lifecycles
- Spring Boot
  - @SpringBootApplication
  - @RestController
  - @GetMapping
- HTTP
- JSON



You should be able to complete this workshop with the theory slides from class.

2 Getting started

To begin this workshop, please accept the following GitHub Classroom assignment: https://classroom.github.com/a/vZrBe_G8.
This assignment contains a project with some code already provided to get you on your way.

On Toledo you will also find multiple User Stories: in software development and product management, a user story is an informal, natural language description of features of a software system. A user story is written from the perspective of a user of the system. A user story is thus basically a way of describing additional functionality that we need to code. Only open and read these files when instructed by this document. We recommend committing and pushing to your GitHub repository after every story.

Open the **start code** in IntelliJ to get started.

2.1 Look at the provided code



The following section is accompanied by movie “1. Startcode” on Toledo.

It is a good idea to get the know the start code a bit before transforming it in a back-end. As mentioned in the introduction, it is now a plain Java application consisting of classes and a main method.

The classes model the (very basic) functionality of a library. Try to answer the following questions by going through the start code:

- A person visiting the library is modelled in the class User. Why does the constructor use the “setter” methods instead of assigning fields directly? What is the advantage?
- How is inheritance used to avoid duplication between Book and Magazine?

- Lending out a Publication is done by creating a Loan object. Follow the code from the constructor of Loan onwards: what is the intention behind line 45-54 of the “setPublications” method?
- Would it be better to make the class Publication abstract?
- Is the parameter endDate in the Loan constructor useful?

Finish by checking out the provided unit tests and running the main method. Try to finish the main method by adding some Loans and returning the publications in them.

3 Bootstrapping our Back-End



The following section is accompanied by movie “2. Plain Java To Spring Backend” on Toledo.

While our domain is useful, the biggest downside of our current application is that is “closed”. To let people use our program we would need to make the following changes:

- Code a (terminal based) interface to our classes.
- Provide everyone that wants to use the app with the bytecode. If they install a JVM they can run the code and use it.

This can work, but we are going to take a different route. By exposing the functionality of our domain through a back-end we get the following advantages:

- The functionality will be accessible over HTTP. If we place the app somewhere where it’s URL is reachable, everyone can use the app without the need for distribution.
- There is only one version of the code. If we need to change something, we can easily make a new version available. Users do not need to update their own local version.

So, let’s take our project one step further by building a back-end with a REST API. This transformation requires adjustments primarily in the pom.xml for adding Spring dependencies and in LibraryApplication.java to bootstrap our Spring application.



Before proceeding with the project, make sure you understand the theory about Spring Boot – Intro and HTTP API GET.

Step 1: Add maven dependencies

Open the pom.xml and modify it to include the Spring Boot starter parent. Add the <parent> snippet directly below <modelVersion> and before <groupId>:

```
<modelVersion>4.0.0</modelVersion>
<parent>
<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-parent</artifactId>
	<version>3.4.1</version>
	<relativePath/>
</parent>
<groupId>be.ucll</groupId>
<artifactId>workshop</artifactId>
<version>1.0-SNAPSHOT</version>
```

We are going to use these Spring Boot dependencies:

spring-boot-starter-actuator	Offers production-ready features to help monitor and manage the application.
spring-boot-starter-web	Facilitates the development of web applications, including RESTful applications using Spring MVC.
spring-boot-devtools	Provides automatic restart and live reload capabilities to speed up development.
spring-boot-starter-test	Includes testing utilities such as JUnit and Mockito.

To add these dependencies, open pom.xml and add the following lines between <dependencies> and </dependencies>:

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-web</artifactId>
```

```
</dependency>

<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-devtools</artifactId>
     <scope>runtime</scope>
     <optional>true</optional>
</dependency>

<dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-tomcat</artifactId>
      <scope>provided</scope>
</dependency>

<dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-test</artifactId>
       <scope>test</scope>
</dependency>
```

Remove the existing JUnit dependency since spring-boot-starter-test already includes it. Click on the “m-icon” in the top-right corner of the pom to process the changes.

Re-run all the tests to see if everything still works.

Step 2: Modifying LibraryApplication.java

In the LibraryApplication.java file, add the @SpringBootApplication annotation, which auto-configures your application based on the dependencies you have added and enables component scanning to register beans. In the main method, call the static run method on the SpringApplication class to give Spring control over your application (you can put the old contents of your main method in comment or remove them):

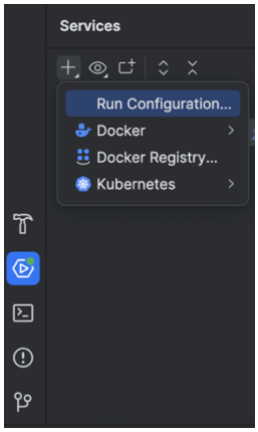
```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

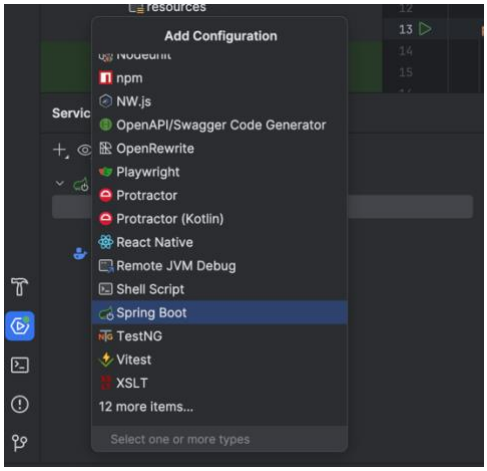
@SpringBootApplication
public class LibraryApplication {
    public static void main(String[] args) {
        SpringApplication.run(LibraryApplication.class, args);
    }
}
```

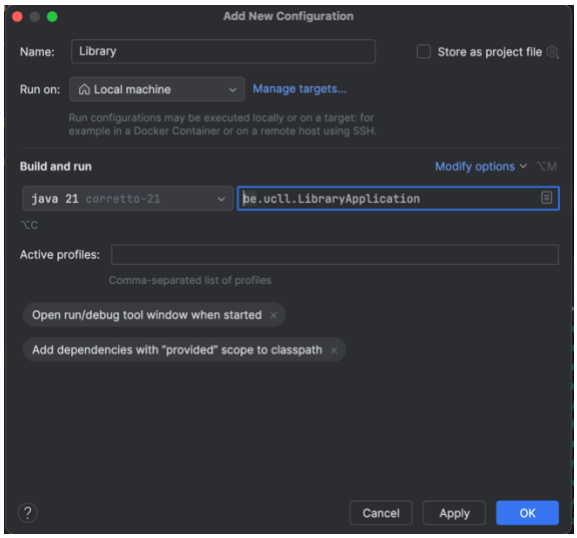
Step 3: Run the application

Now start the application by running the main class.

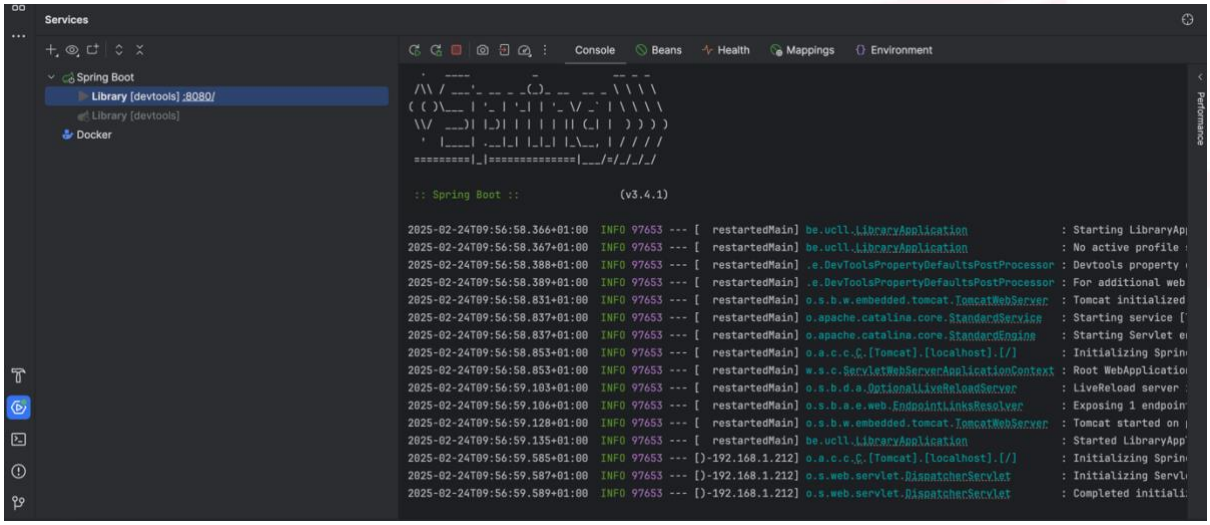
If you don’t see a play button at the left side of your LibraryApplication class, go to Services and add a Spring Boot configuration as follows:







The output of your running application should look like this:



The only thing our main method does is giving control to the Spring Framework. Spring will keep our application/back-end running until it is manually stopped.

Time to build some endpoints so that our back-end can react to some requests while it is running.

4 Coding the first user story



The following section is accompanied by these movies on Toledo:

- 3. Story 1-1
- 4. Story 1-2
- 5. Story 1-3
- 6. Story 1-4

The functionality of our first story is provided in user-story 1. Open and read through that story now.

Story 01	Retrieve all Users

To implement this story successfully, we will need to pay close attention to the API details provided in the table. The table outlines everything from endpoint names to required parameters and expected JSON for the output. The precision of following these API specifications cannot be overemphasized. Clients will only be able to use the endpoint if the specifications are followed to the letter!

We will guide you through the implementation of this first endpoint in the following sections.

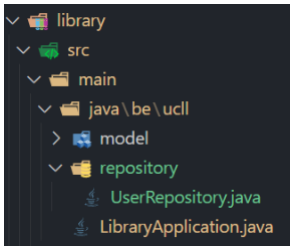
4.1 Repository

In our layered architecture, the bottom layer is comprised of repositories, which are responsible for data storage (most often using a database). In this early stage of development, communication with a database is not yet implemented. We will use a simple List in the class to simulate a database for now.



Before you proceed, make sure that you understand the theory about Spring and annotations (@Repository, @Service, @Controller, …)

Step 1: Creating the repository package and UserRepository.java
First, we need to organize our repositories in a structured manner. Create a package named "repository" within your project structure. Inside this package, create a file named UserRepository.java.



Step 2: Annotate the class with @Repository

Once UserRepository.java is created, the class should be annotated with @Repository. The @Repository annotation in Spring is a specialization of the @Component annotation. Just like @Component, classes annotated with @Repository are automatically detected by Spring's component scanning mechanism.

```
@Repository
public class UserRepository {
…
}
```

Step 3: Adding an instance field for users and initialize dummy data
Within UserRepository, add an instance field of type List<User> with construction logic and a getter. This list will serve as our in-memory database for now.

```
@Repository
public class UserRepository {
    private List<User> users;

    public UserRepository() {
        users = new ArrayList<>(List.of(
        new User("John Doe", 25, "john.doe@ucll.be", "john1234"),
        new User("Jane Toe", 30, "jane.toe@ucll.be", "jane1234"),
        new User("Jack Doe", 5, "jack.doe@ucll.be", "jack1234"),
        new User("Sarah Doe", 4, "sarah.doe@ucll.be", "sarah1234"),
        new User("Birgit Doe", 18, "birgit.doe@ucll.be", "birgit1234")
        ));
    }

    public List<User> getUsers() {
        return users;
    }
}
```

Note that the service classes that will use this repository do not see the difference between this in-memory implementation and an actual database. They just depend on the repository to store data. The fact that the repository is not doing this in an actual database does not impact them. This is encapsulation in action.

4.2 Service

The service layer acts as a bridge between the controllers and the repository layer. Together with the domain model, the service layer is responsible for the business logic of the application. The business logic for retrieving all users is very simple in this scenario: call the repository to get all the users.

Step 1: Create the service package and UserService.java

First create a "service" package that will contain all our service classes. Next, create a "UserService.java" in that package. This class will contain methods that define business operations related to users. For instance, fetching all users from the repository, adding a new user, updating user information, etc.



Step 2: Annotate the class with @Service

Annotate the service class with @Service so that this class is eligible for component scanning. When you import this class, make sure you import the one from Spring framework.

```
@Service
public class UserService {
    …
}
```

Step 3: Injecting the UserRepository dependency

Thanks to Spring, we don't need to instantiate the UserRepository ourselves. This is achieved by autowiring the UserRepository in the UserService class.

```
@Service
public class UserService {

private UserRepository userRepository;
```

```
  @Autowired
  public UserService(UserRepository userRepository) {
    this.userRepository = userRepository;
  }
}
```

Note: since we are using constructor injection, Spring already knows which dependencies the UserService class needs. Therefore, the @Autowired annotation can be omitted.

Step 4: Implement service methods

The next step is to implement the business logic methods within UserService. In the case of retrieving all users, this is very straightforward: delegate this to the UserRepository.

```
public List<User> getAllUsers() {
    return userRepository.getUsers();
}
```

Step 5: Write a unit test for UserService

Create a new UserServiceTest.java in *src/test/java/be/ucll/unit/service* and write a unit test. This is a simple step, as we are basically testing that our service does nothing to the output of the getUsers method of the repository. Make sure that you assert not only the size of the returned users, but also the users themselves.

```
public class UserServiceTest {


    @Test
    public void
givenRepository_whenCallingAllUsers_allUsersFromRepositoryArePresent() {
        UserRepository repository = new UserRepository();
        UserService service = new UserService(repository);


        List<User> result = service.getAllUsers();


        Assertions.assertEquals(repository.getUsers().size(),
result.size());
       Assertions.assertTrue(repository.getUsers().containsAll(result));
    }
}
```

4.3 Controller

The last step in our layered architecture is creating a controller class. By defining endpoints and handling HTTP requests (GET, POST, PUT, DELETE, etc.), the controller class acts as the central point where user requests are intercepted, and responses are generated. It delegates tasks to the service layer.

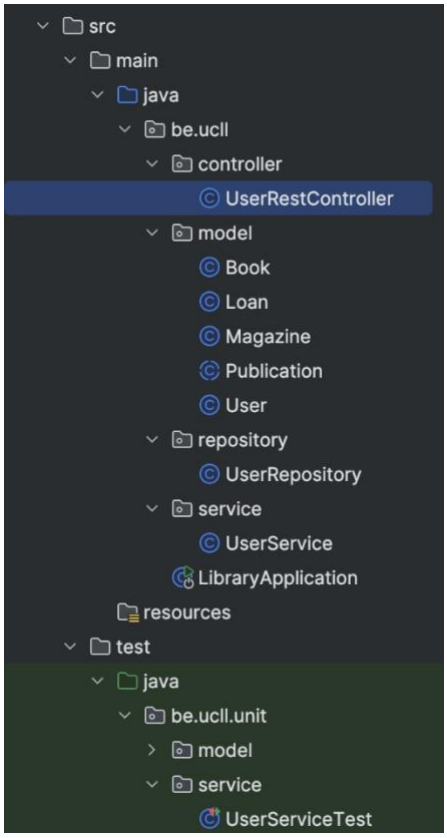


Before you proceed, make sure that you understand the theory about Rest controllers.

Step 1: Create the controller package and UserRestController.java

Create a new file “UserRestController.java” in the package *src/main/java/be/ucll/controller.*

Your project structure should look like this by now:



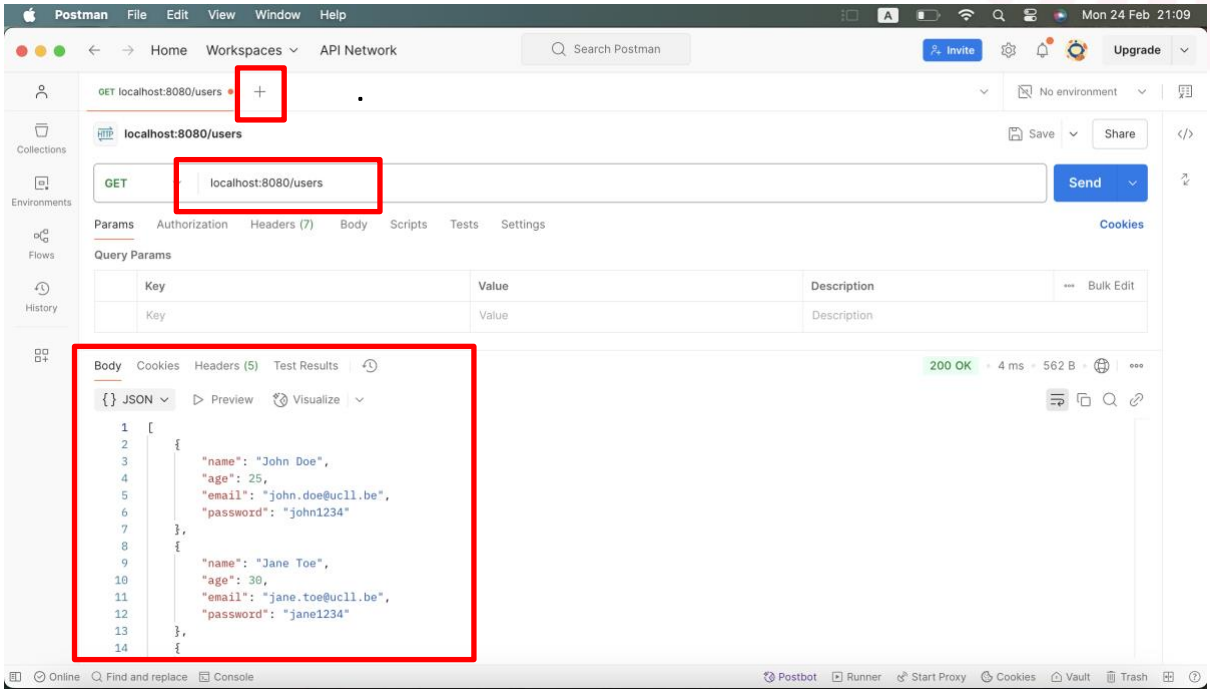
Step 2: Implementation steps
To implement the controller, follow these steps:

- Annotate this class with *@RestContoller*
- Add a request mapping for the path “/users”. Have a close look at the API as specified in story 01.
- Use *constructor injection* to inject an instance of UserService in this controller.
- Implement a method *getAllUsers* that serves a GET request on the endpoint “/users”:

```
@GetMapping
public List<User> getAllUsers() {
  return userService.getAllUsers();
}
```

4.4 Testing the API

Since we do not have a front-end application, we need a third-party tool to test our API. In this course, we will be using Postman. First start the application in IntelliJ and then open Postman:



Click the "+" icon and enter the full path to the users endpoint in the request URL field ("localhost:8080/users" by default). Click "Send".

At the bottom, you can see the JSON that is returned by the UserRestController.

You can save this request for later use in a new "Library" collection. You can consider story 1 finished!

5 Coding more endpoints

Story 02	Retrieve all adults

Read story 02 and try to implement this endpoint in a similar way. You will need an extra GetMapping in UserRestController that serves the endpoint “/users/adults”. Since our UserRestController already operates in the context of “/users” (RequestMapping), you can achieve this by adding a path with the following GetMapping:

```
@GetMapping(“/adults”)
```

The logic to get all adult users mimics a database query with a "where age >= 18" clause, so implement this in the repository (not in the service). It is the repository that handles storage and retrieval of data.

Don’t forget to create a new unit test for the service and to test your API via a Postman request.

5.1 Path variables

Path variables provide a way to include dynamic data directly in the URL of a request. They are used to specify resources or actions that are dynamic in nature, allowing to parameterize a request. For example, in a URL pattern like:

```
/users/{userId}
```

{userId} is a path variable that can be replaced with actual user IDs when making requests.

```
/users/12
```

Look at the examples and theory about path variables and implement story 03.

Story 03	Retrieve all users within age range

5.2 Request parameters

Request parameters and path variables are both methods used to pass data through HTTP requests. While path variables are part of the URL's path and used to identify a specific resource, request parameters (“/resource?key=value&key=value”) are appended to the URL after a question mark “?”. This enables controlling the behavior of the request in a more granular way.

Practice this by implementing:

Story 04	Filter users by name

6 Additional Practice

Using everything you have learned so far, complete the API by implementing the following stories:

Story 05	Filter publications by title and type
Story 06	Retrieve all publications with available copies
Story 07	Retrieve all loans by user

After each story, and preferably multiple times during a story, re-run **all** tests to make sure your changes did not break any other functionality.

6.1 Optional Domain Extension

The functionality of a library is rather crudely modelled in the domain.

Currently a Loan object has no indication of which publications are already returned. Meaning that you can return a Loan multiple times, each time increasing the number of available copies.

Try to improve the domain so that the following constraint is enforced:

- Every publication in a loan can only be returned once. An error is thrown if a loan is returned a second time.