# Exercises

## 1. Creating a New Project

1. Go to [http://start.spring.io](http://start.spring.io/) to access the Spring Initializr
2. In the "Generate a" drop-down, select "Maven Project"
3. Specify the Group as com.unibuc and the Artifact as management
4. Add the Spring Web and Thymeleaf dependencies
5. Choose jar, Click the "Generate Project" button to download a zip file containing the project files
6. Unzip the downloaded " management.zip"
7. Import the project into your IDE
8. As part of the import process, the IDE will download all the required dependencies

<https://stackoverflow.com/questions/63468269/maven-plugin-not-found-in-intellij-ide>

1. Open the file src/main/java/com/unibuc/managementprep/ ManagementPrepApplication.java and note that it contains a standard Java "main" method (with signature: public static void main(String[] args))
2. Start the application by running this method. There won’t be any web components available yet, but you can see the start up of the application in the command window.
3. Add a controller by creating a file called CourseController.java in the src/main/java directory. The goal is to have the CourseController class in the com.unibuc.managementprep.controllers package starting at the root directory src/main/java
4. The code for the CourseController is:

package com.unibuc.managementprep.controllers;  
  
import org.springframework.stereotype.Controller;  
import org.springframework.ui.Model;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RequestParam;  
  
@Controller  
public class CourseController {  
 @GetMapping("/course")  
 public String displayCourse(  
 @RequestParam(value = "code",   
 defaultValue = "001") String code, Model model) {  
 model.addAttribute("code", code);  
 return "course";  
 }  
}

* + The @Controller annotation => check documentation (later we will use @RestController) -> the controller will be detected as it is in the root package (com.unibuc.managementprep)
  + I want this controller to present me the details of a course, so I will use codes for courses; add method displayCourse

public String displayCourse(String code) {  
 return "course";  
}

* + The code parameter is not yet used, but this is the method that I want to invoke based on an incoming URL
  + The job of a Controller is to receive an HTTP request then return to a view or return an HTTP response
  + In this case we are going to redirect to a view; return “course” will forward to src/main/resources/templates/course.html;This file doesn’t exist, so we will need to create it later
  + How the string “course” in the return connects to that file? => this is using viewResolver from Spring; Spring will convert that String into a destination
  + To connect displayCourse method to the outside world we need to declare the URL that will get to this method; for that we will use the annotation @GetMapping => check documentation; it takes an URL that will be appended to the localhost:8080
  + The url is currently /hello
  + If I want the URL to be /hello?code=001 I will add a @RequestParam String code
  + I want the url to work even if I don’t supply a code, so I will provide a default value to the RequestParam
  + We need to pass the code parameter (the value) to the view
  + We will use the Model supplied by Spring
  + DI will be used by Spring when translating an URL call to an invoke on the displayCode method; Model is a collection of keys and values, and we will add such a pair

model.addAttribute("code", code);

* + When Spring goes to the course page, it will send the model with the HTTP request to the view resolver
  + Create the course.html page

1. Create a file called course.html in the src/main/resources/templates folder
2. The code for the course.html file is:

<!DOCTYPE HTML>  
<html xmlns:th="http://www.thymeleaf.org" lang="en">  
<head>  
 <title>Universitatea din Bucuresti - Cursuri</title>  
</head>  
<body>  
<h2 th:text="'Pagina curs ' + ${code} + '!'"></h2>  
</body>  
</html>

The prefix xmlns:th indicates that the page will include tags defined by thymeleaf. Check <https://www.thymeleaf.org/> for an overview of tags

1. Start up the application and navigate to <http://localhost:8080/course> . You should see the string "Pagina curs 001” in the browser => All the Thymeleaf was compiled on the server side
2. Change the URL in the browser to <http://localhost:8080/course?code=TWJava> . You should now see the string "Pagina curs: TWJava!” in the browser
   * **!**We have a model (map of keys and values), a view (html page rendered with thymeleaf) and a controller (detected with component scan, URL registered with the GetMapping, extracts the parameter, adds it to the model and returns that String that the ViewResolver will forwards us to course.html in the templates folder)
3. localhost:8080 presents an error; Add a home page to the app by creating a file called index.html in the src/main/resources/static folder
   * The files under the static folder doesn’t change with each request, can be cached. The ones under templates have something dynamic and might change with each request
4. The code for the index.html file is:

<!DOCTYPE HTML>  
<html lang="en">  
<head>  
 <title>Welcome - UNIBUC</title>  
</head>  
<body>  
<h2>Say hello</h2>  
<form method="get" action="/course">  
 <label for="code">Course code:</label>  
 <input type="text" id="code" name="code"><br><br>  
 <input type="submit" value="Check course by code">  
</form>  
</body>  
</html>

1. Restart application, discuss the form and how the URL changes, how index.html can be cached
2. Go to application.properties => standard configuration file that comes with Spring boot

logging.level.web=debug

1. Restart application (application.properties is read at startup)

Go through the form to display course than check the console (for view negotiating)

1. Let’s generate the output artifact that would be deployed to a cloud provider

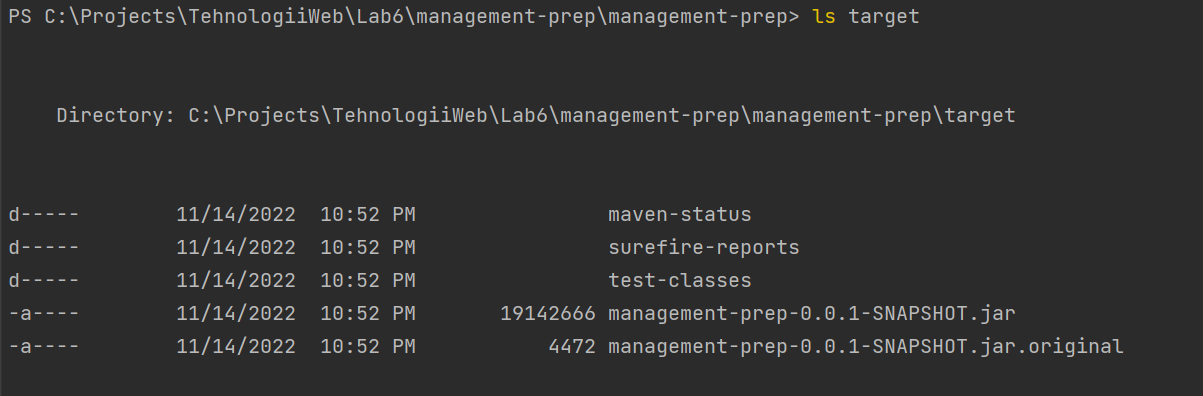
Run mvn clean package (or mvn clean install)

package will compile your code and also package it. For example, if your pom says the project is a jar, it will create a jar for you when you package it and put it somewhere in the target directory (by default).

install will compile and package, but it will also put the package in your local repository. This will make it so other projects can refer to it and grab it from your local repository.

Under target we can see two jars:

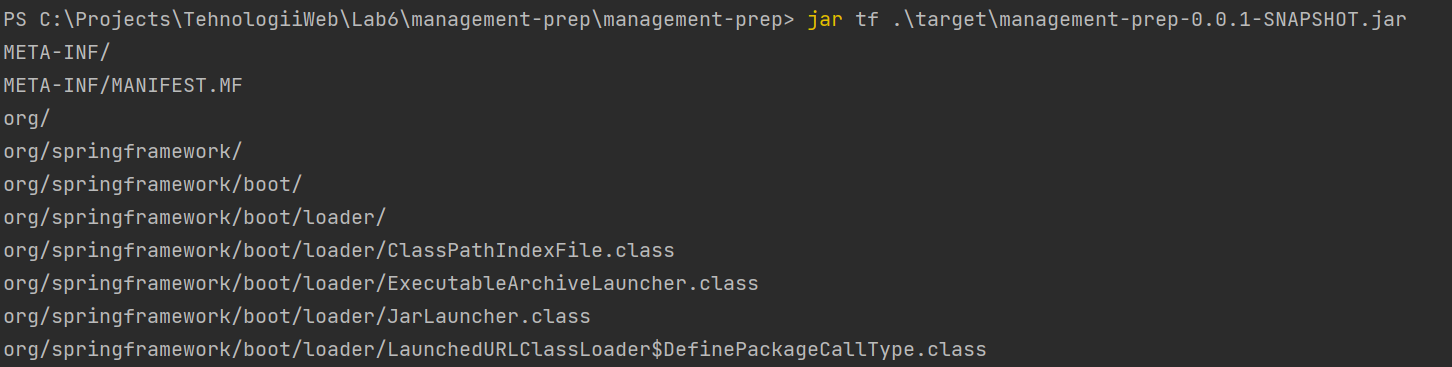
C:\Projects\TehnologiiWeb\Lab6\management-prep\management-prep> ls target



* + One contains the embedded tomcat (the big one)
  + The other one contains only our code

Jar command can help inspect the content of a jar

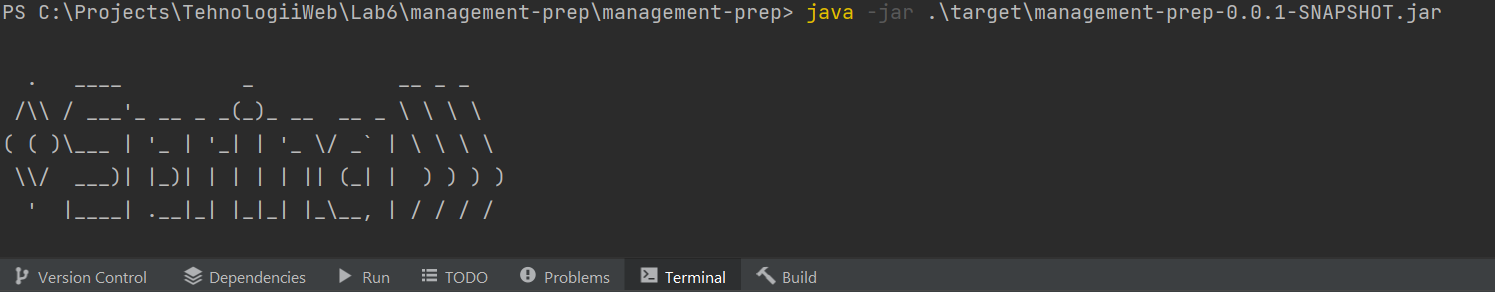
C:\Projects\TehnologiiWeb\Lab6\management-prep\management-prep> jar tf .\target\management-prep-0.0.1-SNAPSHOT.jar



Inspect the classes

A full Java application that can be started from the command line

Java -jar



This is what we will deploy into a cloud provider, or any machine with Java installed.

1. Let’s test with Unit Test

Go to CourseController class name, Alt+Enter (Create Test) – Or Generate ->Test

CourseControllerUnitTest

OK

Add @Test annotation

Public sayHello() {}

//for this unit test we will instantiate the Course controller, call the displayCode method and verify the String that comes out

//the complication is with Model; this is an interface and I need to supply an actual instance of a class that implements an interface; choose from the documentation or checking when running the application (~model.getClass.getName in displayCourse)

@Test  
void displayCourse() {  
 //given  
 CourseController courseController = new CourseController();  
 Model model = new BindingAwareModelMap(); //an implementation class for Model  
 //when  
 String result = courseController.displayCourse("003", model);  
 //then  
 *assertEquals*("course", result);  
 *assertEquals*( "003", model.getAttribute("code"));

//assertAll -> JUnit5 addition -> if the first assert fails, the second one is still executed  
*assertAll*(  
 () -> *assertEquals*("course", result),  
 () -> *assertEquals*( "003", model.getAttribute("code"))  
);

}

This test ignored all the annotations, it’s just a Java call

Run the test by executing this class as a JUnit test. It should pass. It’s not terribly useful, however, since it isn’t affected by the request mapping or the request parameter.

1. Let’s test with Spring MVC => To perform an integration test instead, use the MockMVC classes available in Spring. Create a new class called HelloControllerMockMVCTest in the com.oreilly.demo.controllers package in src/test/java
2. The code for the integration test is:

package com.oreilly.demo.controllers;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;

import org.springframework.http.MediaType;

import org.springframework.test.web.servlet.MockMvc;

import static org.hamcrest.Matchers.is;

import static org.junit.jupiter.api.Assertions.assertNotNull;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;

@WebMvcTest(HelloController.class)

public class HelloControllerIntegrationTest {

@Autowired // Ask Spring for an instance of this class from the Application Context

private MockMvc mvc;

(and(v

@Test

public void testHelloWithoutName() throws Exception {

mvc.perform(get("/hello").accept(MediaType.TEXT\_HTML))

.andExpect(status().isOk())

.andExpect(view().name("welcome"))

.andExpect(model().attribute("user", is("World")));

}

@Test

public void testHelloWithName() throws Exception {

mvc.perform(get("/hello").param("name", "Dolly").accept(MediaType.TEXT\_HTML))

.andExpect(status().isOk())

.andExpect(view().name("welcome"))

.andExpect(model().attribute("user", is("Dolly")));

}

}

1. The tests should pass successfully. One of the advantages of the @WebMvcTest annotation over the generic @SpringBootTest annotation is that it allows you to automatically inject an instance of MockMvc, as shown.

## 2. Add a Rest Controller

1. Add another class to the com.oreilly.demo.controllers package called HelloRestController. This controller will be used to model a RESTful web service, though at this stage it will be limited to HTTP GET requests (for reasons explained below).
2. Add the @RestController annotation to the class.
3. By default, REST controllers will serialize and deserialize Java classes into JSON data using the Jackson 2 JSON library, which is currently on the classpath by default. To have an object (other than a trivial String) to serialize, add a class called Greeting to the com.oreilly.demo.json package. In a larger application, this would represent a domain class that you can store in a database or other persistent storage mechanism.
4. In the Greeting class, add a private attribute of type String called message.
5. Add a getMessage method for the greeting attribute that returns the current message.
6. Add a constructor to Greeting that takes a String argument and saves it to the attribute.
7. Add a default constructor that does nothing. This constructor will be used by the JSON parser to convert a JSON response into an instance of Greeting.
8. Add an equals method, a hashCode method, and a toString method in the usual manner. A reasonable version would be:

**package** com.oreilly.demo.json;

**import** java.util.Objects;

**public** **class** **Greeting** {

**private** String message;

**public** Greeting() {}

**public** Greeting(String message) {

this.message = message;

}

**public** String getMessage() {

**return** message;

}

@Override

**public** **boolean** equals(Object o) {

**if** (this == o) **return** true;

**if** (!(o **instanceof** Greeting)) **return** false;

Greeting gr = (Greeting) o;

**return** Objects.equals(message, gr.message);

}

@Override

**public** **int** hashCode() {

**return** Objects.hash(message);

}

@Override

**public** String toString() {

**return** message;

}

}

1. Back in the HelloRestController, add a method called greet that takes a String called name as an argument and returns a Greeting.
2. Annotate the greet method with a @GetMapping whose argument is "/rest", which means that the URL to access the method will be <http://localhost:8080/rest> .
3. Add the @RequestParam annotation to the argument, with the properties required set to false and defaultValue set to World.
4. In the body of the method, return a new instance of Greeting whose constructor argument should be "Hello, " + name + "!".
5. The full class looks like (note that the string concatenation has been replaced with a String.format method)

**package** com.oreilly.hello.controllers;

**import** com.oreilly.hello.json.Greeting;

**import** org.springframework.web.bind.annotation.GetMapping;

**import** org.springframework.web.bind.annotation.RequestParam;

**import** org.springframework.web.bind.annotation.RestController;

@RestController

**public** **class** **HelloRestController** {

@GetMapping("/rest")

**public** Greeting greet(@RequestParam(required = false,

defaultValue = "World") String name) {

**return** **new** Greeting(String.format("Hello, %s!", name));

}

}

1. You can now run the application and check the behavior using either curl or a similar command-line tool, or simply accessing the URL in a browser, either with or without a name.
2. To create a test for the REST controller, we will use the TestRestTemplate class, because we included the web dependency rather than webflux which we’ll use in the next exercise. Add a class called HelloRestControllerIntegrationTest in the src/test/java tree in the same package as the REST controller class.
3. This time, when adding the @SpringBootTest annotation, add the argument webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT. This will autoconfigure several properties of the test, including making a TestRestTemplate available to inject.
4. Add two tests, one for greetings without a name and one for greetings with a name.
5. The tests should look like:

@Test

**public** **void** greetWithName(@Autowired TestRestTemplate template) {

Greeting response = template.getForObject("/rest?name=Dolly", Greeting.class);

assertEquals("Hello, Dolly!", response.getMessage());

}

@Test

**public** **void** greetWithoutName(@Autowired TestRestTemplate template) {

ResponseEntity<Greeting> entity = template.getForEntity("/rest", Greeting.class);

assertEquals(HttpStatus.OK, entity.getStatusCode());

assertEquals(MediaType.APPLICATION\_JSON, entity.getHeaders().getContentType());

Greeting response = entity.getBody();

**if** (response != null) {

assertEquals("Hello, World!", response.getMessage());

1. }

}

1. One test uses the getForEntity method of the template, which returns a ResponseEntity<Greeting>. The response entity gives access to the headers, so the two provided asserts check the status code and the media type of the response. The actual response is inside the body. By calling getBody, the response is returned as a de-serialized Greeting instance, which allows you to check its message.
2. The other test uses the getForObject method, which returns the de-serialized response directly. This is simpler, but does not allow access to the headers. You can use either approach in your code.
3. The tests should now pass. This application only checks HTTP GET requests, because the application doesn’t have any way to save Greeting instances. Once that is added, you could include analogous POST, PUT, and DELETE operations.