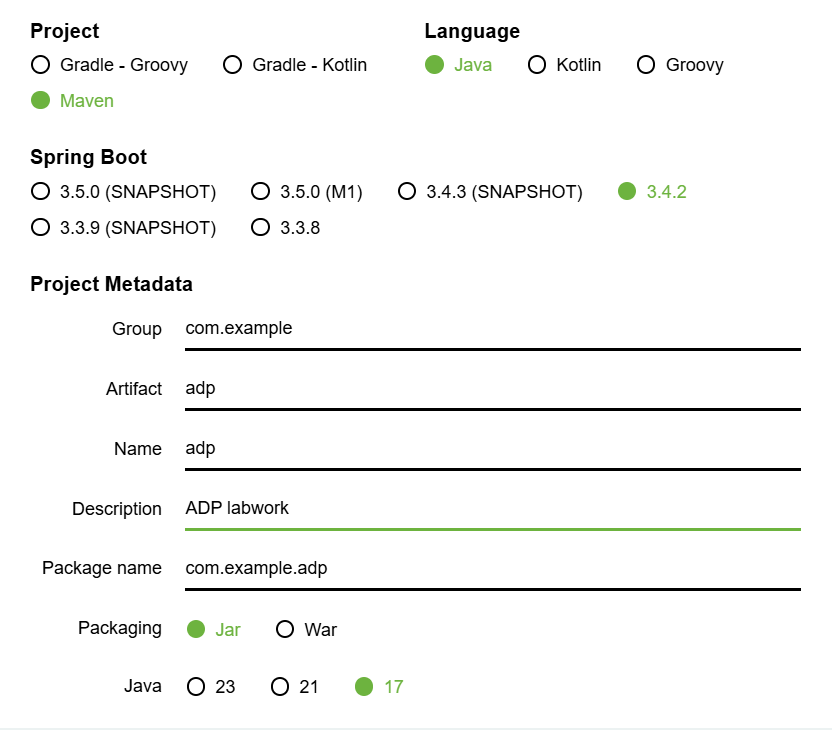
# **Part 1 : Using Spring Boot Initializr to create a Maven project for a web application demonstration**

|  |
| --- |
| **Important Note**  This document assumes you are using IntelliJ IDEA from AppsAnywhere for this exercise as I know it works and will use it for live demonstrations. Although NetBeans should also work, I have experienced problems with it (mainly it seemed very slow to process Maven projects).  You can of course install Eclipse to any machine as follows (but I still recommend IntelliJ for working in the labs):  Visit: <https://www.eclipse.org/downloads>  Download and run the current eclipse installer on the lab machine and it will install in C:\Users\*yourusename* |

## **Create a Spring Boot project with the web dependency:**

Visit <https://start.spring.io> and configure as follows:



* Select Maven Project
* Select Java Language
* Select the latest non-snapshot or candidate version of spring
* Set *Artifact* to "adp"
* Set *Description* to "ADP labwork"
* Set the Java version to highest not exceeding whatever you have in your IDE (e.g. 17)
* Add the ***Spring Web*** dependency.

Generate the project and download a zip archive that contains it. Select "Show in folder" from your download indicator and find the downloaded zip file in the explorer window. Right-click it and extract it. You are now advised to MOVE the extracted folder somewhere else, and potentially rename the folder. I will move it to IntelliJ's default project folder C:\users\username\IdeaProjects and rename it to ADPlab.

Examine the contents of the project folder - you will see there is not much there.

Burrow down to ***adp\src\main\java\com\example\adp*** to see the automatically provided application source file.

Take a look at the Maven configuration file ***pom.xml*** (at the top level) and see if you can identify the Spring Web dependency that we added.

## **Open in IntelliJ and run the basic application**

Now run IntelliJ.

On the **File** menu, choose the **Open…** item and then browse to the folder containing the project. Select the project folder, do not go inside it. Wait while it imports, updates and builds the project.

Now look at the project in the navigator in IntelliJ and inspect the External Libraries node. Here you will find all the jar files that Maven has automatically downloaded and added to your project (these are actually stored in a directory Maven manages so they do not need to be duplicated for multiple projects).

In **src/main/java** you will be able to find the demo application file. Run this just as you would any other Java program. You will see Spring Boot's output on the console and log lines indicating that an embedded webserver (Tomcat) has been started. This demonstrates everything is working as it should, but the application does nothing so there is nothing to see. Terminate the application by stopping it with the IDE's terminate button.

## **Demonstrate that there is a web server running**

In IntelliJ, navigate to the folder **src/main/resources/static**

This is basically the web server's web root folder.

Create a file in here called **index.html** (right-click the **static** folder, choose ***new -> file*** and give the name **index.html**). Now edit this file by right-clicking it and saying *open with text editor*). Put the following HTML code in this file:

|  |
| --- |
| <html>  <head>  <title>Index Page</title>  </head>  <body>  <h1>See, told you there was a webserver running!</h1>  </body>  </html> |

Now run the Spring Boot application again, and then open a web browser window and type **localhost:8080/index.html** into the address bar to see that this page gets served by the spring application. Terminate the Spring Boot application and then refresh the browser window to convince yourself it came from your program (it will no longer be there).

## **Add a Rest Controller to your application to deliver dynamically generated content**

In **src/main/java/com.example.adp1** create a new class called **HelloController**.

Edit it so it contains the following, adding any imports as indicated by the IDE:

|  |
| --- |
| @RestController  **class** HelloController {  @GetMapping("/")  **public** String hello() {  **return** "hello world!";  }  } |

Run the Spring Boot application again. Note that if you forgot to stop it last time it will fail to start saying that port 8080 is already in use. Always make sure to stop it before running with new changes.

Now visit **localhost:8080/** in your web browser. You should see it says "hello world!".

What is happening?

OK, the main application class is annotated with **@SpringBootApplication**, and that makes LOTS of things happen behind the scenes when it is run. One thing that happens is that it automatically searches for classes in the application with recognized annotations on them. It finds our class **HelloController** and sees that it is annotated with **@RestController**. It knows that this means this class will contain methods to be invoked in response to HTTP requests received by the web server. It automatically connects up our class with the web server to make this happen. To do this, it looks at the methods declared in this class and finds our hello() method is annotated with **@GetMapping**.

|  |
| --- |
| Methods in **@RestController** classes that are annotated with **@GetMapping** will be invoked in response to HTTP GET requests. In brackets after the **@GetMapping** we specify the specific URI we want the method to respond to. In this case it is "/" which means the root of the website.  Besides **@GetMapping**, there are similar annotations for other kinds of HTTP requests, for example **@PostMapping**, **@PutMapping** and **@DeleteMapping**. |

Because the **@GetMapping** annotation specifies the URI to respond to, we can have as many as we like in our controller class. Add the following method to the class after the ***hello()*** method:

|  |
| --- |
| @GetMapping("/helloagain")  **public** String[] helloAgain() {  **return** **new** String[] { "hello world!", "again", "hang on what's this?" };  } |

STOP the program and then run it again, and visit **localhost:8080/helloagain** in your browser.

Note that this method does not return a String, but a String *array*. What is displayed in your browser is actually a JSON representation of a String array:

["hello world!","again","hang on what's this?"]

|  |
| --- |
| The methods in a **@RestController** class convert their return type into a JSON representation automatically when sending to the web browser, and as we will soon see, automatically turn incoming JSON data into corresponding Java objects when reading the body of a request message. |

Let's demonstrate this a little better. Create this new class in your project:

|  |
| --- |
| **public** **class** MyPOJO {  **private** String firstName;  **private** String lastName;  **private** **long** idNumber;  **public** MyPOJO(**final** String firstName,  **final** String lastName,  **final** **long** idNumber) {  **this**.firstName = firstName;  **this**.lastName = lastName;  **this**.idNumber = idNumber;  }  **public** String getFirstName() {  **return** **this**.firstName;  }  **public** String getLastName() {  **return** **this**.lastName;  }  **public** **long** getIdNumber() {  **return** **this**.idNumber;  }  } |

This class is called MyPOJO for a reason - POJO is an acronym of *Plain Old Java Object* - there is nothing special about it and it does not have to extend any ancestor classes or implement any specific interfaces. It is also what is called a Java *Bean*. Java Beans are just classes that have a get method for every field they contain, in which the method has the same name as the field except with get in front of it. This is a recognized convention that helps code automation. The JSON serialiser can only serialise objects that are Java Beans like this.

Now add the following method to our HelloController class.

|  |
| --- |
| @GetMapping("/pojo")  **public** MyPOJO[] myPojo() {  **return** **new** MyPOJO[] {  **new** MyPOJO("John", "Smith", 1234567),  **new** MyPOJO("Sarah", "Brown", 7654321) };  } |

STOP the program and then run it again, and visit **localhost:8080/pojo** in your browser.

You will see a JSON array of MyPOJO objects in the browser.

Now try the following method in the HelloController class:

|  |
| --- |
| @GetMapping("/pojo/{first}/{last}/{id}")  **public** MyPOJO myPojo( @PathVariable("first") String firstName,  @PathVariable("last") String lastName,  @PathVariable("id") **int** idNumber) {  **return** **new** MyPOJO(firstName, lastName, idNumber);  } |

This one will be invoked by a GET request that starts "pojo" and then has another three fragments following it. Note how the @GetMapping annotation uses names enclosed in curly brackets {} for the three following fragments, and note how the parameters of the method have been annotated with @PathVariable using these names to identify which fragment to use. The fragments of the URI are converted into the type of these variables and provided to the method automatically.

With this method in place, stop and restart the application and then observe the results of calling the following URIs in your web browser:

* **localhost:8080/pojo/Mickey/Mouse/77** (returns a newly created POJO object using the values in the URI, encoded into JSON)
* **localhost:8080/pojo/Mickey/Mouse** (returns an error as we have no mapping for *pojo* followed by two fragments)
* **localhost:8080/pojo/Mickey/Mouse/Ooops** (returns an error as we have no mapping for *pojo* followed by three STRING fragments, only a mapping for String, String, Int).

This is to illustrate the way the URL can be used in requests.