

## Web Services - Practical Worksheet 1

### Setup Your Environment

The aim of this worksheet is to install the software you'll need in the module and to implement a tiny web service to test that it's all working.

We will be using Java and the Spring framework, available at <https://spring.io>. Spring is a modern enterprise Java framework, and it comes with Spring Boot, which packages Spring applications with web server, database, etc so that they can be launched as standalone services directly from the build tool or IDE.

The worksheet loosely follows this Spring Boot guide: <https://spring.io/guides/gs/rest-service>. Occasionally, you may also want to take a look at the book *Building RESTful Web Services with Spring 5* by Raman and Dewailly; it's on the reading list.

#### Prerequisites

- A JDK for Java version 8, 11 or later.
  - Download JDKs here: <https://www.oracle.com/java/technologies/downloads/>
- A tool for testing a web service, such as curl.
  - Download and get documentation for curl here: <https://curl.se>
  - Any recent version of curl will do.
- curl is a command line tool. Postman is an alternative for those who prefer GUIs.
  - Download Postman here: <https://www.postman.com/downloads/>
- *Recommended:* Bash, the GNU Bourne-Again SHell.
  - Bash is the standard shell on Linux systems. (And Linux is the standard OS of most of the cloud servers that you might deploy web services on, so it's a good idea to get familiar with the shell environment that you'll encounter there.)
  - Bash is already installed on MacOS. (The version may be outdated but that doesn't concern us – the basic features of the shell haven't changed.)
  - On Windows, you can install git for windows, which includes a Bash. Download it here: <https://gitforwindows.org>

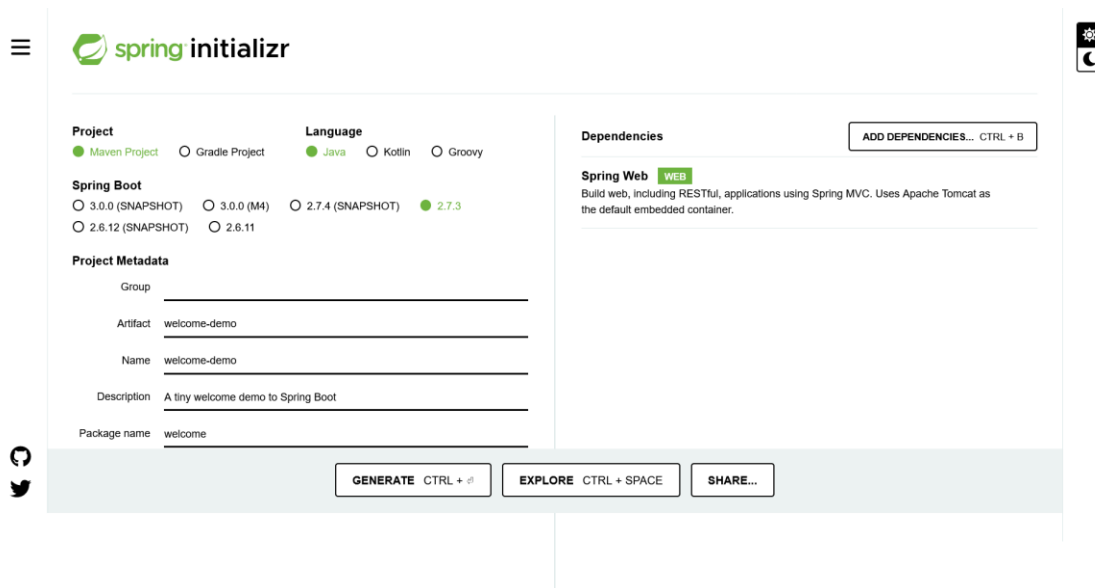
Personally, I prefer command line tools because I can store commands in scripts and replicate what I've done. I'll be using curl throughout the practicals, and I'll be showing the curl commands in Bash syntax. You can use other command lines but command lines often differ in their treatment of string quotes and escape characters. Those differences can mean that a command that will work in Bash won't work in Windows Powershell, and vice versa.

I'm also not using any Java IDE. I just use an editor with syntax highlighting, and compile and run on the command line. You are free to use any IDE you like; you should find tips on how to configure your IDE to work with Spring Boot in the Spring Boot documentation.

The tiny web service we are going to build should mimic a receptionist in a hotel. That is, the service should respond to a guest ringing the bell at the reception desk by welcoming them.

## Step 1: Set up your first Spring Boot project

Spring offers the Spring Initializr, a handy online tool to configure projects so that all build dependencies are taken care of. To use this service, follow the link to <https://start.spring.io> and fill in the form. You can leave most fields at their default value, but you must add **Spring Web** as a dependency, and the Java version selected must be no later than the JDK you have installed. I recommend also customizing the project metadata. Below is a screenshot of my Initializr form:



The screenshot shows the Spring Initializr web interface. It has a sidebar with a hamburger menu and a Twitter icon. The main area is divided into sections: Project, Language, Dependencies, and Project Metadata. The Project section has radio buttons for Maven Project (selected) and Gradle Project. The Language section has radio buttons for Java (selected), Kotlin, and Groovy. The Spring Boot section has radio buttons for 3.0.0 (SNAPSHOT), 3.0.0 (M4), 2.7.4 (SNAPSHOT), 2.7.3 (selected), 2.6.12 (SNAPSHOT), and 2.6.11. The Dependencies section has a button 'ADD DEPENDENCIES... CTRL + B' and a list with 'Spring Web WEB' (selected) and a description. The Project Metadata section has input fields for Group, Artifact (welcome-demo), Name (welcome-demo), Description (A tiny welcome demo to Spring Boot), and Package name (welcome). At the bottom are buttons 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'. There are also social media icons for GitHub and Twitter on the left.

Click **Generate** and download the ZIP file. Extract the ZIP file on your machine. The resulting directory is your project directory. If you've chosen Maven as your build tool (as I have) you will find a `pom.xml` file. This is an XML file describing the configuration of the project, including dependencies. Normally, you should not need to modify this file. But if you are interested in its structure, and how you could have generated it if the Initializr service did not exist, take a look at Chapter 2 of the book by Raman and Dewailly.

## Step 2: Creating a resource representation

The web service we are going to build is going to be very simple. It will only return a welcome message. The Initializr has already created the main class that will be used to launch the web service. Since I named the service `welcome` in the Initializr form, the name of the main class is `WelcomeDemoApplication`. Find this class in the `src` tree; in my case it lives in directory `src/main/java/welcome`. Create a directory for package `service` there. Add the following `Welcome` class to the new package.

```
package welcome.service;

public class Welcome {
```

```

    private final String lang;
    private final String msg;

    public Welcome() {
        lang = "en";
        msg = "Welcome";
    }

    public String getLang() { return lang; }
    public String getMsg() { return msg; }
}

```

This class only has two fields, a constructor and two getters. No setters because the fields can't be updated – instances of this class represent a boring welcome message (in English).

### Step 3: Create a resource controller

Add the following `WelcomeController` class to the same package.

```

package welcome.service;

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

@RestController
public class WelcomeController {

    @GetMapping("/ding")
    public Welcome welcome() {
        return new Welcome();
    }
}

```

This is as minimal as it gets. The `@RestController` annotation marks the class as a controller handling HTTP requests. The body of the class contains a single request handler, the method `welcome`, which handles GET requests for the route `/ding` thanks to the annotation `@GetMapping("/ding")`. In response to such a request (which represents the guest ringing the reception bell) the handler will return a `Welcome` object (which represents the receptionist's welcome).

### Step 4: Compile the project

Open a command line in the root directory of the project. The directory contains Maven wrapper scripts. On the command line, type `./mvnw clean package` to compile. This will download and run Maven, thereby downloading the project dependencies, compiling the classes, and linking it all into a single JAR file.

The project is small; building should only take a few seconds. You'll see lots of output on the command line. If everything went smoothly, Maven will report BUILD SUCCESS, and you will find the JAR file in the target directory.

## Step 5: Launch the service

On the command line, you can launch the service by starting the Java runtime with the JAR file. In my case, the command is `java -jar target/welcome-demo-0.0.1-SNAPSHOT.jar`

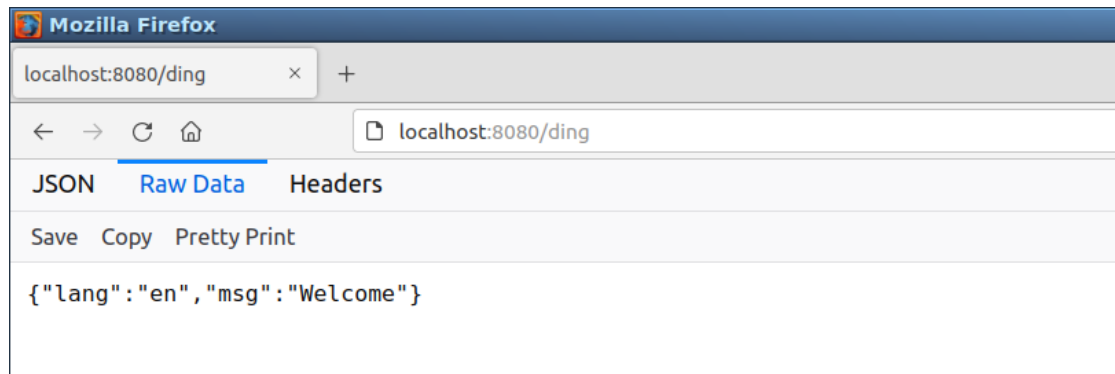
This starts a web server on localhost, listening for requests on port 8080. To stop the server, type Ctrl-C in the command line window (or kill the Java runtime in some other way).

You can also launch the service directly from Maven by typing `./mvnw spring-boot:run`

## Step 6: Test the service with your browser

Navigate your browser to the URL <http://localhost:8080/ding>

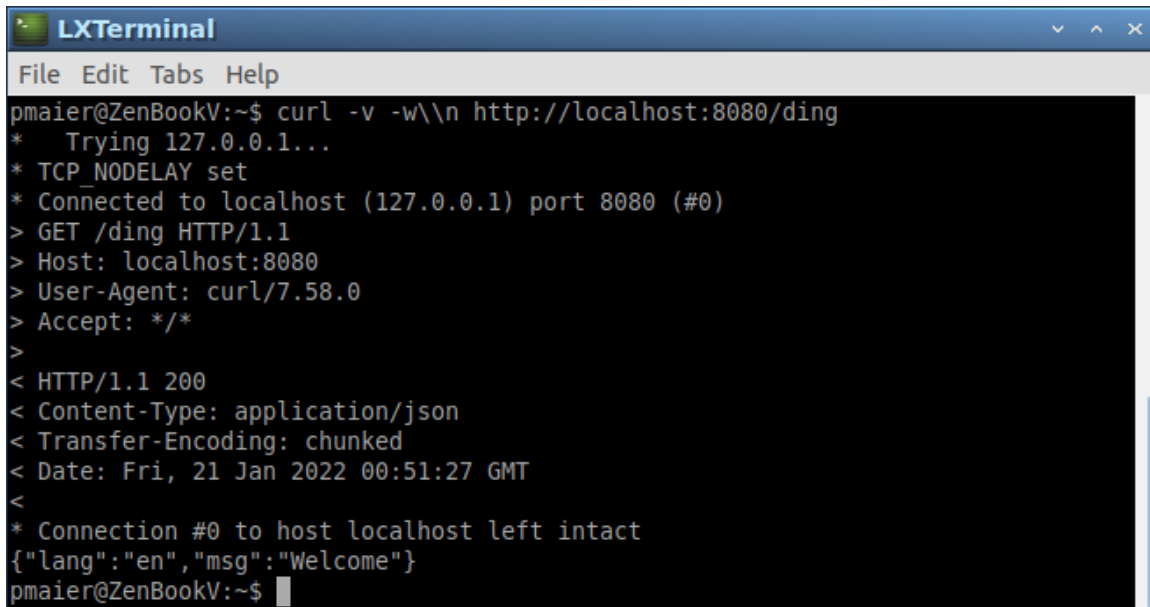
This will send a GET request for route `/ding` to the web server that's running on localhost and listening on port 8080. The browser should display the response, which is a representation of the Welcome object in JSON format. On my machine, it looks something like this.



## Step 7: Test the service with curl

curl can provide a lot of information about what your web service does. It is also indispensable for testing requests other than GET (but more on that later). In order to test your web service, type `curl -v -w\\n http://localhost:8080/ding`

This will again send a GET request for route `/ding` to the web server at localhost:8080. On my machine, the output looks like this.

A screenshot of an LXTerminal window. The title bar says "LXTerminal". The menu bar has "File", "Edit", "Tabs", and "Help". The terminal shows a user "pmaier@ZenBookV" at the prompt "~\$". They enter the command "curl -v -w\\n http://localhost:8080/ding". The output is verbose, showing the connection process to 127.0.0.1 on port 8080. It shows the GET request headers: "GET /ding HTTP/1.1", "Host: localhost:8080", "User-Agent: curl/7.58.0", and "Accept: /\*/\*". The response headers are: "HTTP/1.1 200", "Content-Type: application/json", "Transfer-Encoding: chunked", and "Date: Fri, 21 Jan 2022 00:51:27 GMT". The connection is marked as intact. The final output is a JSON object: {"lang": "en", "msg": "Welcome"}.

```
pmaier@ZenBookV:~$ curl -v -w\\n http://localhost:8080/ding
* Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET /ding HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.58.0
> Accept: /*/*
>
< HTTP/1.1 200
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Fri, 21 Jan 2022 00:51:27 GMT
<
* Connection #0 to host localhost left intact
{"lang": "en", "msg": "Welcome"}
pmaier@ZenBookV:~$
```

The option `-v` tells curl to be verbose and produce lots of output, such as header information, status codes, etc. Leave the option out and see how the output changes.

The option `-w\\n` tells curl to add a newline character `\\n` at the end of its output, to compensate for the fact that the server's response does not end with a newline character. (The double backslash is necessary to escape the backslash in `\\n` – your command line may use different conventions.) Leave the option out and see whether it makes a difference.

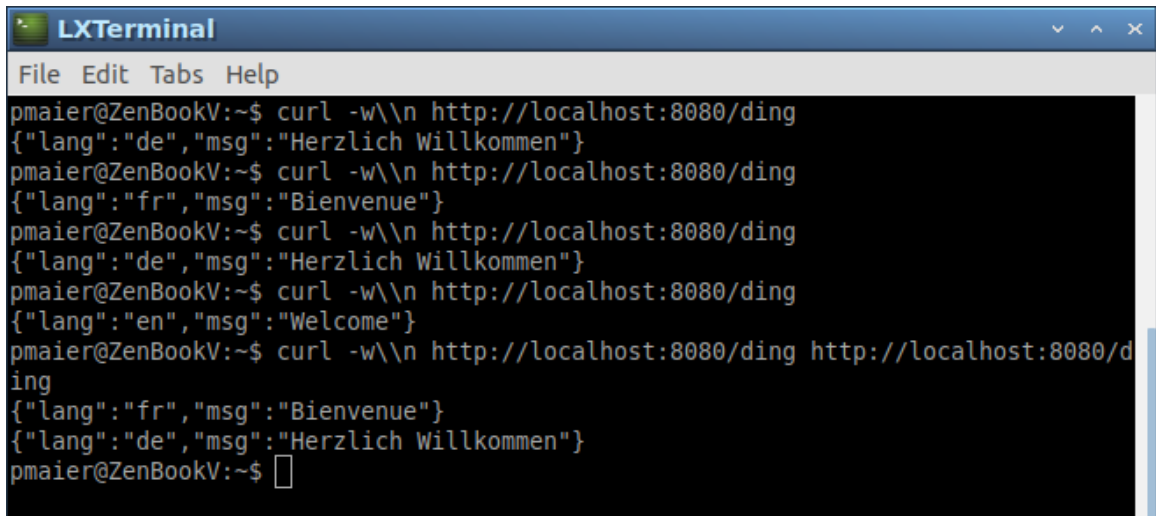
Try also changing the URL to see how the web server responds when non-existent URLs are requested, or how curl responds when requests are sent to non-existent servers.

## Step 8: Make the service a little more interesting

So far, our web service is very much like a static web page, except that the web server delivers the response in JSON format instead of HTML. To make it a little more interesting, let's make the response a little more dynamic.

**Task:** Modify the `Welcome` class such that the constructor randomly returns welcome messages in different languages.

Test the service using curl. As an example, here is some output from my implementation. Note that the last command shows how curl can send multiple requests to the server, one after the other, when multiple URL arguments are provided.



```
LXTerminal
File Edit Tabs Help
pmaier@ZenBookV:~$ curl -w\\n http://localhost:8080/ding
{"lang":"de","msg":"Herzlich Willkommen"}
pmaier@ZenBookV:~$ curl -w\\n http://localhost:8080/ding
{"lang":"fr","msg":"Bienvenue"}
pmaier@ZenBookV:~$ curl -w\\n http://localhost:8080/ding
{"lang":"de","msg":"Herzlich Willkommen"}
pmaier@ZenBookV:~$ curl -w\\n http://localhost:8080/ding
{"lang":"en","msg":"Welcome"}
pmaier@ZenBookV:~$ curl -w\\n http://localhost:8080/ding http://localhost:8080/d
ing
{"lang":"fr","msg":"Bienvenue"}
{"lang":"de","msg":"Herzlich Willkommen"}
pmaier@ZenBookV:~$
```

That's it for this worksheet.