



Java Advanced – lesson 1 – Getting started

Wim Wiltenburg

(wim.wiltenburg@inholland.nl)

Objectives for today



- Understand the base concept of library management
- Set up our development environment
- Understand the base concept of Rest API and Spring Boot
- Create our first REST API

Table of content



- Libraries
- Maven
- Starting up
- Rest API
- Spring Boot
- Live Demo Guitar Shop

Libraries



- A collection of functionalities your application can depend on, usually bundled in a jar file.
 - Java native libraries, e.g. rt.jar (Runtime Environment)
 - External libraries, e.g. Apache Commons
- When creating a java project native libraries are automatically added to the classpath by the IDE.
- External libraries have to be added to the classpath manually or with a build tool, e.g. Maven, Ant, Gradle

Example: apache-commons



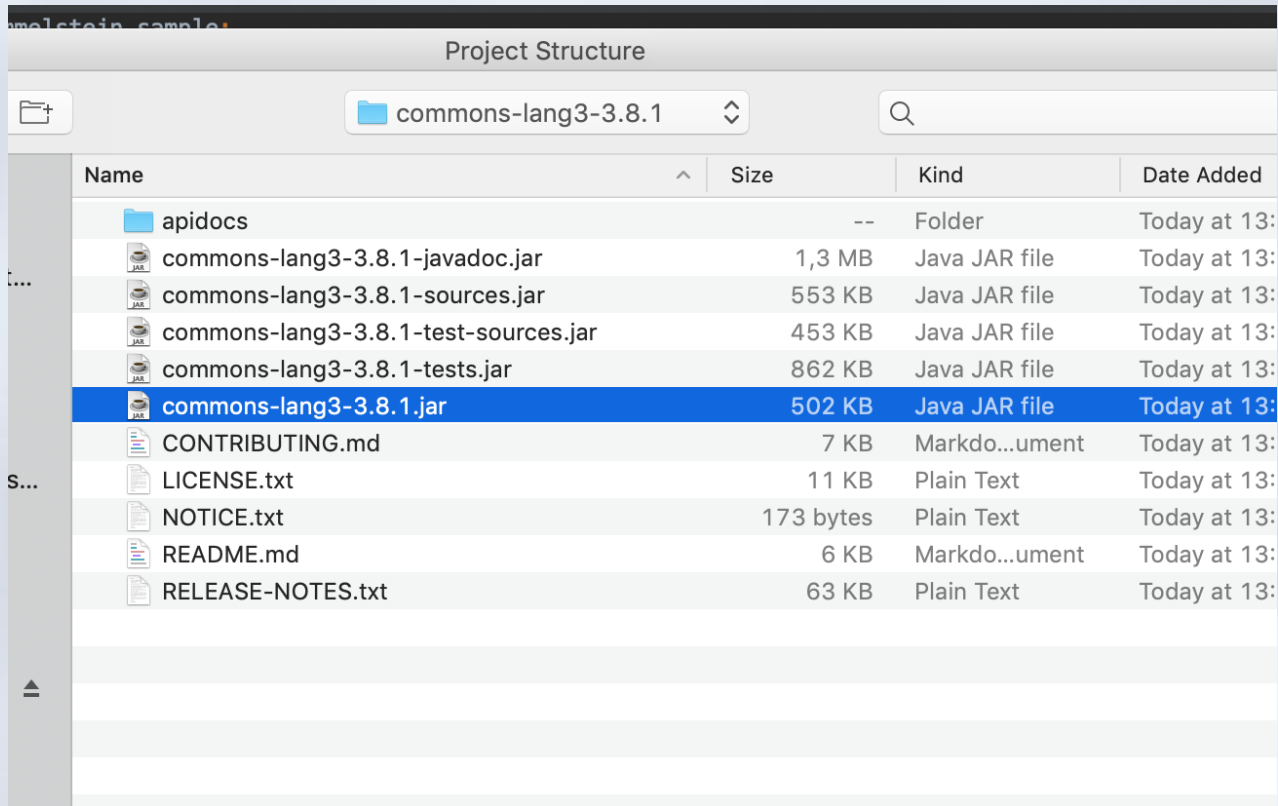
Apache Commons are libraries that make common java tasks easier. They are external libraries, so we need to download them, otherwise:

```
4  import javax.swing.*;
5  public class Application {
6
7      public static void main(String[] args) {
8
9          String num = "1";
10         int numFromString = NumberUtils.~
11     }
12
13 }
14
```

Example: apache-commons (2)



You download the library and add it to your project:



Project Structure

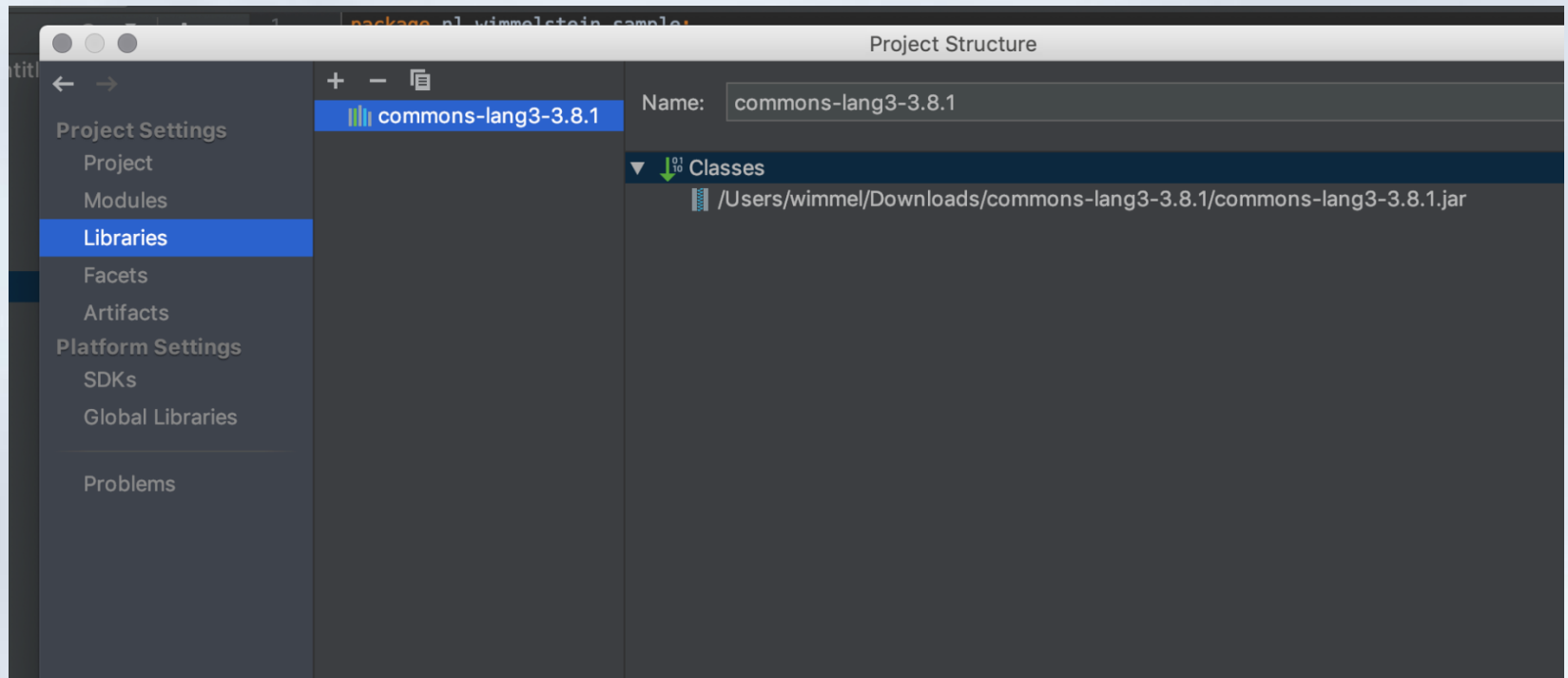
commons-lang3-3.8.1

Name	Size	Kind	Date Added
apidocs	--	Folder	Today at 13:
commons-lang3-3.8.1-javadoc.jar	1,3 MB	Java JAR file	Today at 13:
commons-lang3-3.8.1-sources.jar	553 KB	Java JAR file	Today at 13:
commons-lang3-3.8.1-test-sources.jar	453 KB	Java JAR file	Today at 13:
commons-lang3-3.8.1-tests.jar	862 KB	Java JAR file	Today at 13:
commons-lang3-3.8.1.jar	502 KB	Java JAR file	Today at 13:
CONTRIBUTING.md	7 KB	Markdo...ument	Today at 13:
LICENSE.txt	11 KB	Plain Text	Today at 13:
NOTICE.txt	173 bytes	Plain Text	Today at 13:
README.md	6 KB	Markdo...ument	Today at 13:
RELEASE-NOTES.txt	63 KB	Plain Text	Today at 13:

Example: apache-commons (3)



Now it's in the project you can use it:



Example: apache-commons (4)



```
package nl.wimmelstein.sample;

import org.apache.commons.lang3.math.NumberUtils;

public class Application {

    public static void main(String[] args) {

        String num = "1";
        int numFromString = NumberUtils.createInteger(num);

        // Rest of the application code

    }

}
```


On the command line



Add the library as option to the compiler and at runtime:

```
java — -bash — 80x24
[>>>
[>>>cat Application.java
import org.apache.commons.lang3.math.NumberUtils;

public class Application {

    public static void main(String[] args) {

        String num = "1";
        int numFromString = NumberUtils.createInteger(num);
        System.out.println(numFromString);
        // Rest of the application code
    }

}

[>>>javac -cp ../commons-lang3-3.8.1.jar Application.java
[>>>java -cp ../commons-lang3-3.8.1.jar Application
1
[>>>
[>>>
```

A ton of libraries



Java has the most extensive collection of libraries of almost any programming language. Commonly used libraries include:

- Apache Commons or Google Guave (general)
- BouncyCastle (cryptography)
- JUnit (testing)
- Jackson (working with JSON and XML)

Sometimes these libraries evolve into a whole framework, e.g. Spring

How to manage libraries



When your project grows, so does the number of libraries (usually). Common problems:

- Versioning
- Dependencies on other libraries (transitive)
- Security (CVC)
- Does it do the job you intend
- Always use libraries you trust, and test them.
- Use a dependency management tool, like Maven or Gradle.

Apache Maven



- Describes the project in POM.xml
 - Name, version, artifact, development team, modules, etc.
- Handles dependencies
- Has different objectives (goals):
 - Compile
 - Build
 - Test
 - Package
 - Install
 - Deploy

Maven structure



For maven to work correctly, a uniform structure needs to be implemented:

- `/src/main/java` – source code
- `/src/main/resources` – properties, static resources (html/images/js)
- `/src/test/java` - unit tests
- `/src/test/resources` – properties, static resources and functional test scenarios
- The `POM.xml` resides in the root of the project and each module has its own `POM.xml`

Finding your library



If you are dealing with a problem, chances are somewhere in the world someone already figured it out for you, and has shared it for anybody to use.

That sharing is done in Maven Central, a service that allows you to easily find a solution to your problem and incorporate your library in your project.

That site is Maven Central Repository:

<https://mvnrepository.com>

Starting up...



We need:

- Adopt OpenJDK 11
- Git
- IDEA IntelliJ version 2019.3.4 Community Edition (contains Apache Maven v3, auto discovers git – install git first)
- Postman



First: check what we have



```
PS D:\workspace> java -version
```

```
openjdk version "11.0.6" 2020-01-14
```

```
OpenJDK Runtime Environment AdoptOpenJDK (build 11.0.6+10)
```

```
OpenJDK 64-Bit Server VM AdoptOpenJDK (build 11.0.6+10,  
mixed mode)
```

```
PS D:\workspace> git --version
```

```
git version 2.7.4.windows.1
```

If these commands don't give you results,
you must install what's missing

Building our API using Spring Boot



- Open Source Framework for the JVM
- Create standalone Spring Applications
- Embed Tomcat, or other application server
- Starters contain all configuration and dependencies that you need to get started
- Convention over configuration
- Implements Servlet Specification
- Spring Boot Web Starter supports REST API
- Inversion of Control (IoC)

Inversion of Control



At the heart of Spring (Boot) there's a Spring Container. This container will scan the classpath and:

- Create objects
- Wire objects together
- Configure object
- Manage object life cycle from creation to destruction

It removes the tight coupling between objects (e.g. by instantiating via `new Object()`) and allows for easy management and testing

Rest API



- Software architecture for manipulating resources through Webservices using HTTP
- Implements HTTP methods for requests and response using JSON or XML
- Most common methods are:
 - GET – requests using URI parameters, no body
 - POST – handles large requests, using body
 - PUT – updates resources, using body
 - DELETE – deletes resources, ignored body
 - OPTIONS – queries server for accepted methods
 - HEAD – like GET, but requests only the headers



Josh Long (龙之春, जोश, Джош Лонг, جوش لونق) ✓ @ · 12 dec. 2017

If your children are restless and can't sleep: start dot Spring dot io.


If u suffer from Indigestion from a long night of alcohol abuse and PHP: start dot spring dot io.

If u want for inspiration in the morning before your first cup of tea or coffee: start dot spring dot io!

Starting a Spring Boot Project is as easy as going to
<https://start.spring.io>

Start.spring.io choices



 **spring** initializr



Project
☒ Maven Project ☐ Gradle Project

Spring Boot
☐ 2.3.0 M4 ☐ 2.3.0 (SNAPSHOT) ☐ 2.2.7 (SNAPSHOT) ☒ 2.2.6
☐ 2.1.14 (SNAPSHOT) ☐ 2.1.13

Project Metadata
Group
Artifact
Name
Description
Package name
Packaging ☒ Jar ☐ War
Java ☐ 14 ☒ 11 ☐ 8

Language
☒ Java ☐ Kotlin ☐ Groovy

Dependencies ADD DEPENDENCIES... CTRL + B
Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

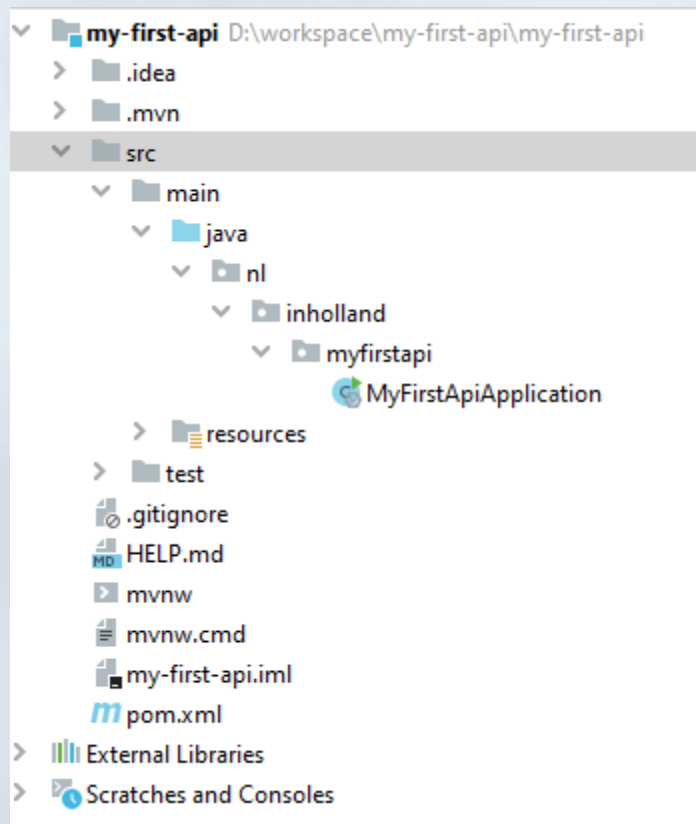


GENERATE CTRL + G EXPLORE CTRL + SPACE SHARE...

Opening the project



- When we first open directory as Maven project we see the maven structure we would expect:



The Spring Initializer has created:

- pom.xml
- Maven directory structure
- MyFirstApiApplication.java
- A test class
- HELP.md
- application.properties
- .gitignore, that will cover most of the files and folders we don't want to send to git

Maven will parse the pom.xml and take care of all the dependencies

Running the project



The MyFirstApiApplication class holds the main method:

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

@SpringBootApplication
public class MyFirstApiApplication {

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

The Annotation `@SpringBootApplication` turns on the magic, because this tells the Spring Container that it's now in control of the application, and the lifecycle of all objects.

The output ...



No active profile set, falling back to default profiles: default // we'll come back to that
Tomcat initialized with port(s): 8080 (http)
Starting service [Tomcat]
Starting Servlet engine: [Apache Tomcat/9.0.33]
Initializing Spring embedded WebApplicationContext
Root WebApplicationContext: initialization completed in 777 ms
Initializing ExecutorService 'applicationTaskExecutor' // MultiThreaded
Tomcat started on port(s): 8080 (http) with context path '' // This the the root
Started MyFirstApiApplication in 1.391 seconds (JVM running for 2.298)

We can now call our application on <http://localhost:8080> but we will see an error, because we haven't defined anything yet

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sun Apr 05 15:14:19 CEST 2020

There was an unexpected error (type=Not Found, status=404).

No message available

Model/View/Controller



Spring (Boot) is designed around the principle of MVC, Model/View/Controller:

- **Model:** this centers around the data and the mechanisms of manipulating it. This can be a service that knows how to access a database, and retrieve all objects that fit certain criteria.
- **Controller:** deals only with http requests and responses, and knows nothing of either the model, or the view that displays the data.
- **View:** This is whatever is used to display the data. It can be a web page, a desktop application, etc.

Stereotype annotations



Stereotypes are objects that are scanned at startup by the Spring Container and initialized, and if necessary linked together. The most important are:

- `@Component` – This is a generic type of class that can be auto-detected.
- `@Controller` – This is a class that is a web controller, dealing with http
- `@Service` – A class that serves as an interface to the model

Components are initialized as Singletons. Only one object of that type are present at any given moment.

By annotating a class with a stereotype, we allow the Spring container to scan it and initialize it

Service



A service is a class that is responsible for handling data. That can be data from a database, or from a file, or whatever is in memory.

By annotating the class as `@Service` we tell the Spring Container that it should be detected and initialized at startup.

Controllers



As said, a controller is an object that deals with HTTP requests.

The controller does that by listening on an endpoint. When we annotate the class with `@RequestMapping`, we define the top-level endpoint, e.g `/guitars`:

```
@Controller
@RequestMapping("/guitars")
public class GuitarController
```

When the annotation is on a method we will define the lower-level endpoint, what type of requests (GET/POST, etc.), and in which format the data should be represented:

```
@RequestMapping(value = "", method = RequestMethod.GET, produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity getAllGuitars() {
```

`ResponseEntity` is a datatype that will wrap the data, e.g. a list of guitars, but can also set e.g. the HTTP status

```
return ResponseEntity.status(200).body(guitars);
```

Wiring



When the controller is asked to serve a list of guitars, it will not know anything about that, but it does know where to find it: the service.

That means that the controller needs to have the service wired to itself, so it can access the service over and over again, without having to initialize it all the time. One way to do that is by using an annotation:

```
@Autowired  
private GuitarService service;
```

Now we can access the service by method:

```
List<Guitar> guitars = service.getAllGuitars();
```

DEMO: Guitar Shop



We will build a REST API that deals with a guitar shop, and the first class we create is a Guitar class. This has three properties: id (long), brand (String) and model (String)

Then we will build a service that will have a method for serving a list of all guitars that the shop has

Finally we will build a controller that will serve that list of guitars whenever a GET HTTP request is done on the /guitars endpoint.

Homework



Build a REST API in Spring Boot for whatever holds your interest, be it keeping track of your games collection, or a beer shop.

Upload your application to github/gitlab, and send me the link for review.

This demo is available on:

<https://github.com/wimmelstein/guitarshop-api>

Branch: lesson1/my-first-api

Remember:

Every time a to-do-list app is made a puppy dies.

Be kind to puppies