Spring REST Workshop
Lab 1

1 Lab setup

Make sure you have the following items installed

- Latest version of JDK 8 / 11 (note: labs are tested with JDK 8 but should work on JDK 11 with no or minimal changes)
- Eclipse Enterprise Edition for Java and Spring Tool Suite (STS).
- Latest version of Maven
- Wireshark and Rawcap
- A free account at Postman and installed the Postman app
- A suitable text editor (Notepad ++)
- A utility to extract zip files

In each of the main lab folders, there are two subfolders: `changes` and `final`. The `changes` subfolder just holds the source code files for the lab, while the `final` subfolder holds the complete Eclipse project starting from its project root folder. We will use the code from the `changes` subfolder to build up our applications from scratch and you can always fall back on the complete Eclipse project if you encounter any errors while building up the application.

2 Creating a basic Spring REST project

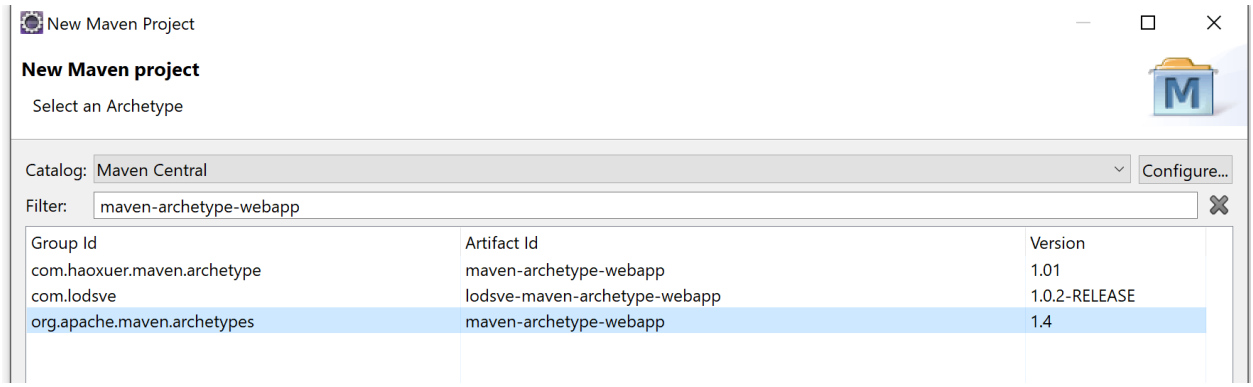
There are several ways to create a basic REST project in the Spring framework.

The main folder for this lab is `First-Spring-Rest`

2.1 Using a basic Maven project archetype

Switch to Java EE perspective.

Start with File -> New -> Maven Project. Select Next and type `maven-archetype-webapp` in the filter. Select the entry with group id: `org.apache.maven.archetypes` and click Next



Enter in the following details in the New Maven Project dialog box and click Finish

Group Id: `com.workshop.boot`
Artifact Id: `FirstRESTMaven`
Version: `0.0.1-SNAPSHOT`
Package: `com.workshop.boot`

You will initially have an error flagged in project entry as there is an `index.jsp` generated automatically and the necessary Servlet classes to compile it are not yet present on the build path. To correct this, right click on the project entry -> Properties -> Targeted runtimes. In the dialog box, select the Apache Tomcat Server (or any other targeted application server of choice). Click Apply and Close.

Replace the contents of the `pom.xml` in the project with `pom.xml` from `changes`. Right click on the project, select Maven -> Update Project, and click OK.

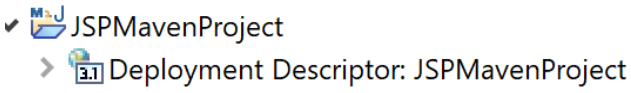
Currently, Web Module (or Servlet) version is set to 2.3. We need to update it to a slightly more recent version (3.1 or 4.0). We will use version 3.1.
Replace the contents of Deployed Resources -> `webapp/WEB-INF/web.xml` in the project with `web.xml` from `changes`. This is based on the schema for Servlet 3.1

Right click on the project entry Properties -> Resource. Click on the drop down box next to the Location entry in the Resource dialog box. This opens a File Explorer at the project folder in your current Eclipse workspace. Navigate into the `settings` subfolder and edit the file: `org.eclipse.wst.common.project.facet.core.xml`

Put in the new Web Module version (3.1) in this element below and save the file.

```
<installed facet="jst.web" version="3.1"/>
```

Right click on the project name -> Refresh. The deployment descriptor should now show version 3.1



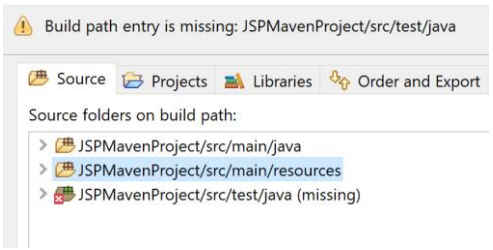
We now need to update the folder structure to match that expected of a typical Maven web app project:

https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

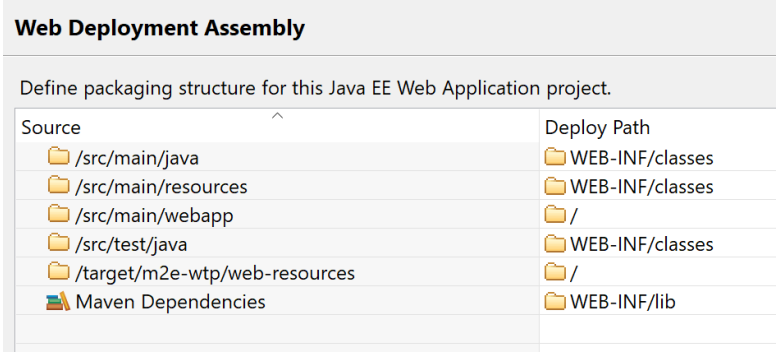
In `src/main`, create two new folders:

- `src/main/java`
- `src/main/resources`

Right click on the project, Properties -> Java Build Path. In the Source tab view, add the `src/main/resources` folder to the build path and click Apply and Close. Eclipse complains that we haven't created `src/test/java` yet, but we don't need to in this example as we are not doing any testing.



Right click on the project, Properties -> Deployment Assembly. This shows how the folders in your current Maven project maps to the document root of the project when it is deployed in the Tomcat server. It should look like something below:



If we check the new `pom.xml` that we added in, we see three new dependencies besides the standard ones that we expect to see in a Spring MVC app:

- logback-classic - default logging framework for a Spring REST app
- jackson-databind - To perform serialization/deserialization of incoming and outgoing JSON requests for the REST service
- tomcat7-maven-plugin - To run the app in an embedded Tomcat server (rather than the Tomcat server integrated in Eclipse)

```
		<!-- Dependency for Logging using Logback -->
		<!-- Default logging framework for Spring Boot -->
		<!-- Optional to be included, only if you want to do logging -->
		<dependency>
			<groupId>ch.qos.logback</groupId>
			<artifactId>logback-classic</artifactId>
			<version>1.2.3</version>
		</dependency>


		<!-- Dependency for JSON binding -->
		<!-- To serialize incoming JSON requests for REST API -->
		<dependency>
			<groupId>com.fasterxml.jackson.core</groupId>
			<artifactId>jackson-databind</artifactId>
			<version>2.9.8</version>
		</dependency>


		<!-- Dependency for Spring MVC -->
		<dependency>
			<groupId>org.springframework</groupId>
			<artifactId>spring-webmvc</artifactId>
			<version>${spring.framework.version}</version>
		</dependency>

		<!-- Dependency for Servlet 3.1 -->
		<!-- So that Maven can run a build properly -->
		<dependency>
			<groupId>javax.servlet</groupId>
			<artifactId>javax.servlet-api</artifactId>
			<version>3.1.0</version>
			<scope>provided</scope>
		</dependency>


				<!-- Optional plugin for embedded Tomcat server -->
				<plugin>
					<groupId>org.apache.tomcat.maven</groupId>
					<artifactId>tomcat7-maven-plugin</artifactId>
					<version>2.2</version>
				</plugin>
```

In `src/main/java`, create a new package: `com.workshop.config`

Place the following classes from `changes` into it:

```
WebConfig
CustomWebAppInitializer
```

In `src/main/java`, create another new package: `com.workshop.boot`

Place the following classes from `changes` into it:

```
Greeting
GreetingController
```

In `src/main/resources`, place `logback.xml` to provide configuration for the Logback framework

We are now going to run the embedded Tomcat server via the plugin in the project POM. This server will start on port 8080 by default, so first make sure there is no other server that is running on this port (for e.g. any existing Tomcat server instances that you have integrated into Eclipse).

Right click on the project entry, select Run As -> 4 Maven Build

In the Goals field in the dialog box, type: `tomcat7:run`

This will deploy the app in the embedded Tomcat server and start it.

In a browser tab, navigate to:

http://localhost:8080/FirstRESTMaven/greeting

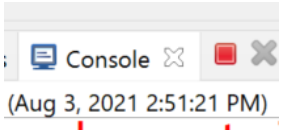
You should see JSON content returned in the form of:

```
{"id":0,"content":"Hello from Spiderman !"}
```

Keep refreshing the browser (pressing F5) to send repeated HTTP calls to the REST app, which results in a new JSON response with the id incremented by 1 each time.

Test sending out GET requests to that URL from Postman as well.

To stop the embedded Tomcat server, click on the red button next to the Console view.

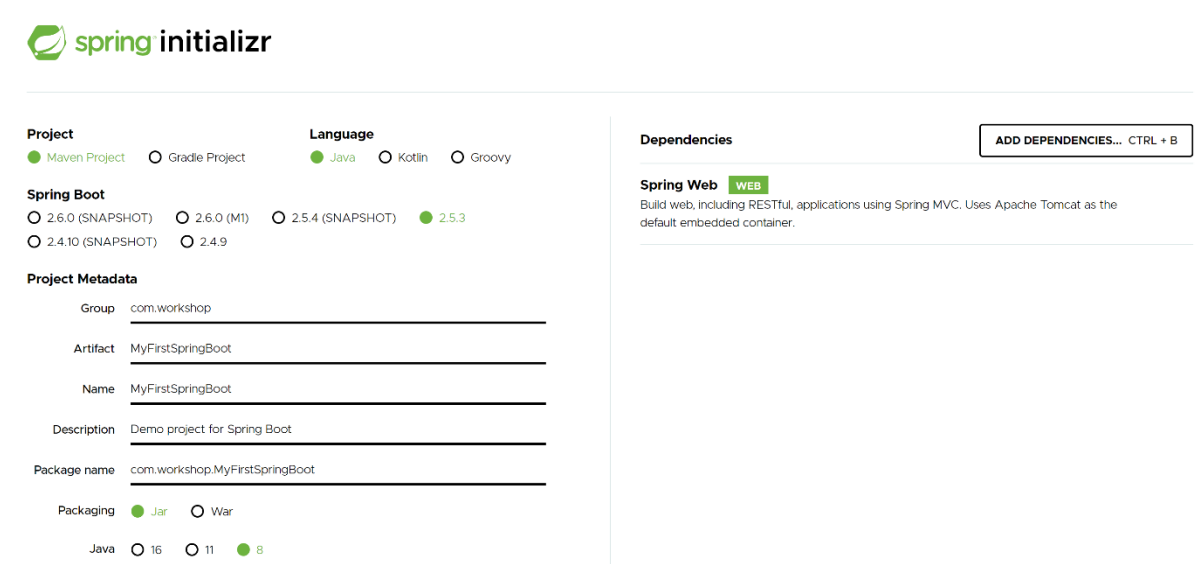


2.2 Using Spring Boot via Spring Initialzr

Navigate to the Spring Initialzr at:

https://start.spring.io/

Key in the values for the fields as shown below and click Generate



Download and unzip the file. Move the folder `MyFirstSpringBoot` to your Eclipse workspace directory (if you are not sure where this is, go to File-> Switch Workspace-> Other, and the dialog box will show the name of the last closed workspace - which should be the current one as well, if you have not switched workspace).

Switch to Java EE perspective.

Do File -> Import -> Maven -> Existing Maven Projects and select `MyFirstSpringBoot`

Examine `pom.xml`

Notice that there are only 2 dependencies here (`spring-boot-starter-web` and `spring-boot-starter-test`), both of which are Spring Boot starters. They aggregate all the dependencies that we used earlier to create a REST project from a Maven webapp archetype. This simplifies the configuration of the project POM.

Notice as well that there are two additional files here that are different from a typical Maven project: these are `mvnw` and `mvnw.cmd`. They are part of the Maven wrapper which includes a prebuilt Maven so that the Maven project here can be built independently of the Maven version installed in Eclipse or on your local machine.

In the package `com.workshop.MyFirstSpringBoot`, place these classes from `changes`:

```
Greeting
GreetingController
```

Make sure to change the package names if they are not changed automatically by Eclipse.

Spring Boot provides an embedded Tomcat server which will start on port 8080 by default, so first make sure there is no other server that is running on this port (for e.g. any existing Tomcat server instances that you have integrated into Eclipse).

Right click on the project entry, select Run As -> 3 Maven Build

In the Goals field in the dialog box, type: `spring-boot:run`

then click Run.

In the Goals field of the dialog box, type: `spring-boot:run`, then click Run

This starts up the REST app in the embedded Tomcat server. The last couple of lines in the Console should look something like this:

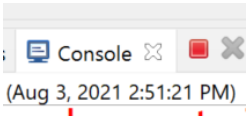
```
2021-08-03 07:29:44.009  INFO 20612 --- [         main] o.s.b.w.embedded.tomcat.TomcatWebServer
: Tomcat started on port(s): 8080 (http) with context path ''
2021-08-03 07:29:44.015  INFO 20612 --- [            main] c.w.M.MyFirstSpringBootApplication
: Started MyFirstSpringBootApplication in 1.116 seconds (JVM running for 1.347)
```

Once the application has started up successfully, open a browser tab at:

http://localhost:8080/greeting

and check for the same results as in the previous lab.

To stop the embedded Tomcat server, click on the red button next to the Console view.



We can also produce an executable JAR file for this app.
Right click on the project entry, select Run As -> 4 Maven Build
For the Goals field, type: `clean package`
Then click Run.
Right click and Refresh the project.

Open a command prompt and navigate to the `target` subfolder of this project.
A shortcut for this is to right click on the target subfolder in the Project Explorer, select Show in Local Terminal -> Terminal.

At the terminal type:

```
java -jar MyFirstSpringBoot-0.0.1-SNAPSHOT.jar
```

As before, test the app by opening a browser tab at:

http://localhost:8080/greeting

To terminate the app server, type Ctrl+C in the terminal.

2.3 Using Spring Boot via Spring Tool Suite

The Spring Tool Suite (STS) is a plugin for the Eclipse Enterprise IDE which provides additional features that supports development of Spring projects. This can be installed as a plugin:

https://www.codejava.net/ides/eclipse/install-spring-tool-suite-for-existing-eclipse-ide

or downloaded as a full Eclipse version by itself at:

https://spring.io/tools

Start up STS and select an appropriate folder for your workspace.

If you are starting with a new workspace, the latest version of STS comes with its own installed JRE for Java 16. It would be better to change the installed JRE and compiler compliance to the current version of JDK installed on your machine.

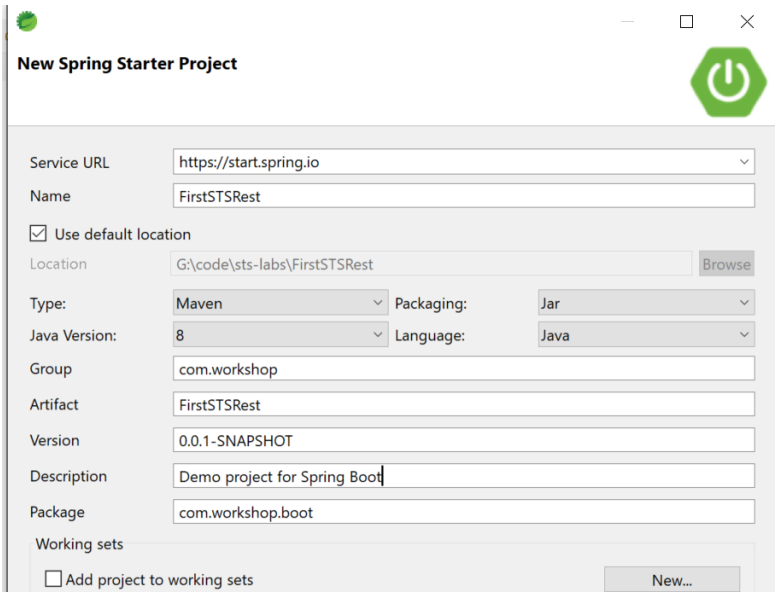
Go to Window -> Preferences -> Java -> Installed JREs.

Click on Add. In the JRE Type dialog box, select Standard VM. Click Next.
In the JRE Home entry, click the Directory button and navigate and select the installation directory of the JDK, for e.g: C:\Program Files\Java\jdk1.8.0_251
Click Finish, then select the JDK entry and click Apply and Close.

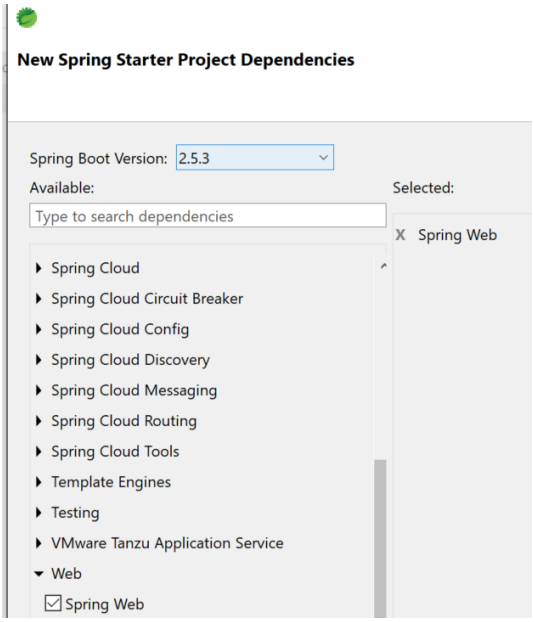
Go to Window -> Preferences -> Java -> Compiler.
In the Compiler Compliance Settings, set it to the version of the JDK installed on your machine (for e.g. 1.8). Click Apply and Close.

Switch to the Java perspective.

Go to File -> New -> Spring Starter Project. The dialog box that appears provides the same options that are presented at Spring Initialzr (https://start.spring.io/). Complete it with the following details:



Click Next and select the following Boot Starter dependencies.
`Web -> Spring Web`



Click Finish.

This generates a Maven project with a Maven wrapper that is identical to the one that we created via the Spring Initialzr in the earlier lab.

In the package `com.workshop.boot`, place these classes from `changes`:

```
Greeting
GreetingController
```

Make sure to change the package names if they are not changed automatically by Eclipse.

Right click on the project entry, select Run As -> Spring Boot App.

The same console output that you saw in the previous lab is produced, but this time with color formatting to facilitate viewing:

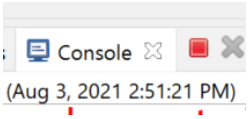
```
2021-08-03 17:05:55.965_[0;39m _[32m INFO_[0;39m _[35m11880_[0;39m _[2m---_[0;39m _[2m[
main]_[0;39m _[36mw.s.c.ServletWebServerApplicationContext_[0;39m _[2m:_[0;39m Root
WebApplicationContext: initialization completed in 493 ms
_[2m2021-08-03 17:05:56.178_[0;39m _[32m INFO_[0;39m _[35m11880_[0;39m _[2m---_[0;39m _[2m[
main]_[0;39m _[36mo.s.b.w.embedded.tomcat.TomcatWebServer _[0;39m _[2m:_[0;39m Tomcat started on
port(s): 8080 (http) with context path ''
_[2m2021-08-03 17:05:56.184_[0;39m _[32m INFO_[0;39m _[35m11880_[0;39m _[2m---_[0;39m _[2m[
main]_[0;39m _[36mc.workshop.rest.FirstStsRestApplication _[0;39m _[2m:_[0;39m Started
FirstStsRestApplication in 0.957 seconds (JVM running for 1.396)
```

Once the application has started up successfully, open a browser tab at:

http://localhost:8080/greeting

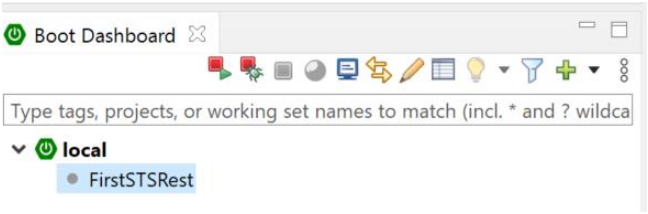
and check for the same results as in the previous lab.

To stop the embedded Tomcat server, click on the red button next to the Console view.



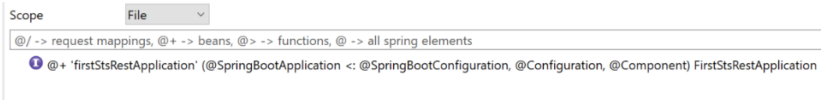
The Boot Dashboard on the left helps facilitate working with the Spring Boot apps. If you can't see this view, go to Window -> Show View -> Other -> look in the Other folder -> Boot Dashboard.

From the drop down list for local, you can see the list of Spring Boot apps available for interacting with. Select an entry and a list of action icons (such as running, stopping or debugging the app)



STS provides a Spring Symbols view which helps to identify Spring related annotations in your various classes. If you can't see this view, go to Window -> Show View -> Other -> look in the Other folder -> Spring Symbols. Select the FirstSTSRest project entry and select Project in the Spring Symbols view. You should be able to see all the Spring annotated annotations in the Project:

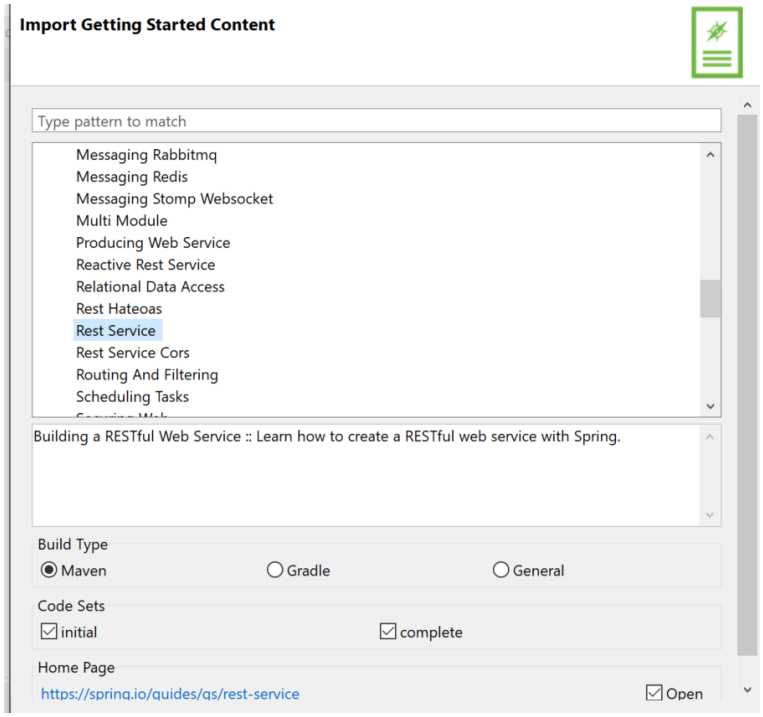
Similarly, you can go to the individual source code files (`GreetingController`, `FirstStsRestApplication`) and select File in the Spring Symbols view to see the annotations at the individual file level.



STS also provides the ability to directly import any of the projects from the Getting Started guides from the main Spring home page:
https://spring.io/guides#getting-started-guides

For e.g. this tutorial details how we can build a simple REST service in the Spring framework step-by-step:
https://spring.io/guides/gs/rest-service/

To create a new Spring Boot project that illustrates this tutorial, go to File -> New -> Import Getting Started Content:
In the Dialog Box, select Getting Starting Guide -> Rest Service and check the other options as follows:



This creates two projects in the Package Explorer:

- gs-rest-service-initial (the initial code base from which you can follow the tutorial)
- gs-rest-service-complete (the final code base at the end of the tutorial)

as well as serving up the webpage documenting that particular tutorial.

Select `gs-rest-service-complete` in the Boot Dashboard and run it.

Once the application has started up successfully, open a browser tab at:

http://localhost:8080/greeting

and check for the same results as in the previous lab.

You can use STS in this way to facilitate walking through the ever growing number of tutorials available on the Spring ecosystem.

3 Monitoring REST HTTP traffic

When developing our own custom REST services and clients and getting them to work together, it is often useful to monitor the HTTP traffic between them to facilitate the debugging process. Often it may not be clear whether an error is caused by the client-side or server-side logic. There are several approaches available to monitor traffic: we look at two common ones:

3.1 Using TCP/IP monitor in Eclipse

Go to Window -> Show View -> Other -> look in the Debug folder -> TCP/IP monitor.

The monitor can be set up to proxy HTTP traffic from a particular source to a given destination, thereby allowing it to monitor the contents of the HTTP packets flowing through it.

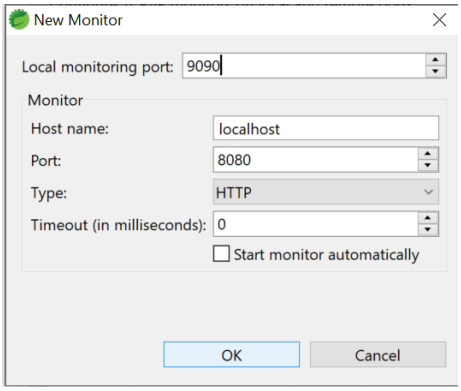
First check for a free port which the monitor to listen on: you can do this with

`netstat -aon | findstr port-number`

in Windows.

Select Properties from the upper right hand corner. In the dialog box, select Add.

Assuming that the port that the REST API server is running on is 8080 and the free port that your monitor will listen to is 9090, enter these details:



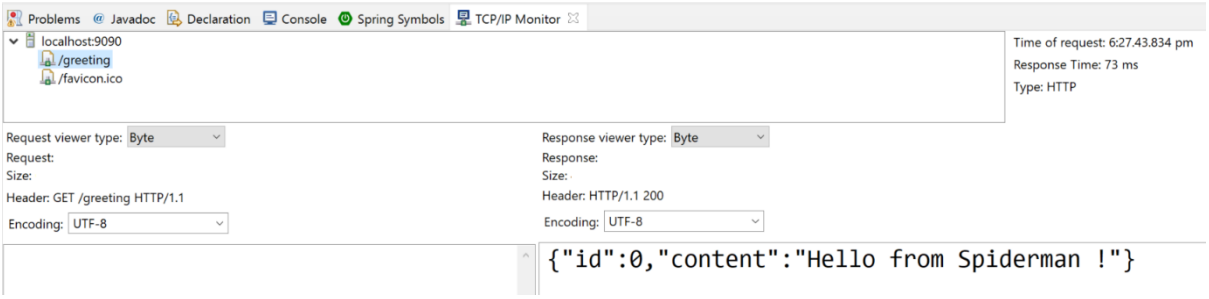
Then select Start and click Apply and Close.

Start up the FirstSTSRest app from the Boot Dashboard.

Using any REST client (Postman, browser, cURL, etc) navigate to:

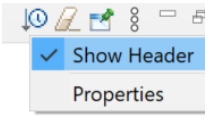
http://localhost:9090/greeting

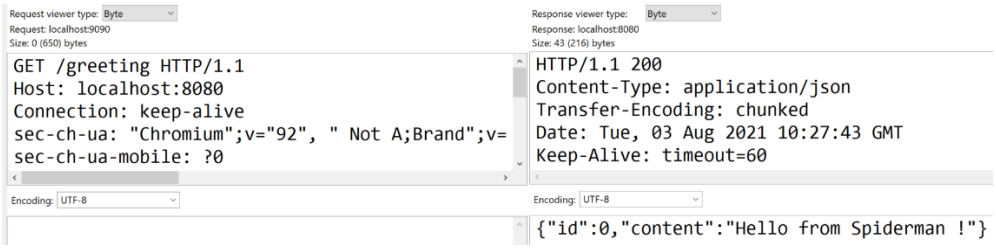
Remember to use the port number that the monitor is listening on (9090 for this example). The HTTP request is relayed through the monitor to the server and is returned via it as well. This allows it to capture and display the contents of the request and response. You should see something similar to the following in the TCP/IP monitor view.



The monitor keeps a historical record of all your previous request/response interactions in a list that you can scroll through to examine. Keep sending multiple GET requests to the URL to test this out.

You can also see header information for the requests / responses by selecting Show Header from the top right menu icon.





When you are done monitoring traffic, shut down the app and stop the monitor from the properties dialog box.

You can add additional monitors to inspect multiple HTTP interactions if you wish.

3.2 Using Rawcap and Wireshark

Wireshark is an open-source packet analyzer that is widely used for network troubleshooting and analysis. It is capable of monitoring traffic for a large amount of protocols, including HTTP. However, we cannot use Wireshark to monitor traffic on localhost as this is virtual rather than physical interface. For that purpose, we will need to use Rawcap.

Start up the `FirstSTSRest` app from the Boot Dashboard.

Place RawCap.exe in a suitable directory, open a command prompt with administrator privilege, and type this:

```
RawCap.exe -f 127.0.0.1 dumpfile.pcap
```

This starts the capturing of packets from the localhost interface.

Using any REST client (Postman, browser, cURL, etc) navigate to:

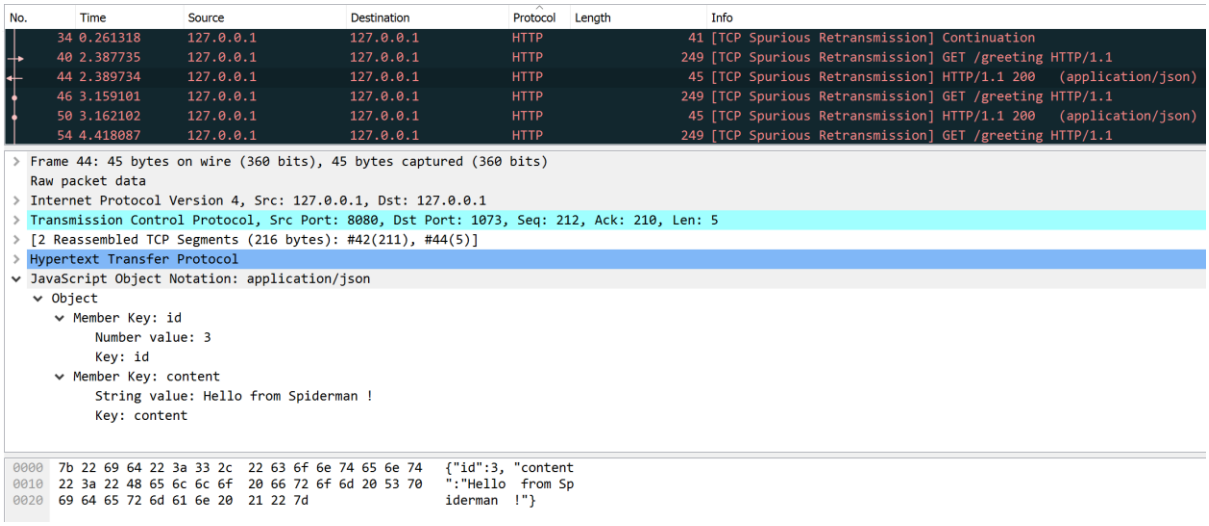
http://localhost:8080/greeting

Send several GET requests to this URL.

Switch back to the command prompt and press Ctrl-C to terminate the RawCap process.

Double click on `dumpfile.pcap` . This should open up Wireshark to list all the packets captured in that file.

Click on the Protocol tab to categorize the packets listed according to protocol type. If you scroll down/up, you should see some HTTP packets pertaining to the GET requests / responses that you sent out earlier. You can drill down into the specific portions of the HTTP packet to view its contents in the lower bottom view panes.



If you are not able to see HTTP traffic, trying shutting down the app as well as STS, and restarting it again. Then repeat the monitoring command involving RawCap.

4 Configuring a Spring Boot Project

The main folder for this lab is `Spring-Boot-Config`

Start up STS. Switch to the Java perspective.

Go to File -> New -> Spring Starter Project. Complete it with the following details:

Name: `BootConfigApp`
Group: `com.workshop.boot`
Artifact: `BootConfigApp`
Version: `0.0.1-SNAPSHOT`
Description: `Demo Configuration for a Spring Boot App`
Package: `com.workshop.boot`

Add the following dependencies:
`Web -> Spring Web`

4.1 Configuring logging

Spring Boot uses Commons Logging for all internal logging but provides options for configuring the underlying implementation. Default configurations are provided for Java Util Logging, Log4J2, and Logback. The default implementation is Logback.
http://logback.qos.ch/

In the package `com.workshop.boot`, place the file: `LoggingController`

In `src/main/resources`, place the file `application.properties`

Start the app and use a REST client (browser or Postman) to send multiple GET requests to:

http://localhost:8080/showlogs

Check the log output from the app in the Console view.

Notice that our logging code implementation is based on SLF4J, which allows to swap a different underlying implementation (for e.g. Log4J2) if we want to without the need to change our source code.

Examine the settings in `application.properties`

```
logging.level.root=WARN
```

sets the logging level for the Spring framework and embedded Tomcat server as it starts up, as well as all other classes for the application.

```
logging.level.com.workshop.boot=TRACE
```

sets the logging level for the classes contained in the package `com.workshop.boot`, which overrides the root logging level setting (if any).

Change the levels for these two settings and restart the app to see the effect on the log output in the Console view. In particular, see how many log messages appear in the framework startup when the logging level is set to `TRACE`

Important note for Eclipse:
Occasionally changing the content of `application.properties` may sometimes result in an error flagged for the POM file. This will usually be due to the `<parent>` element. Just make some trivial change to the POM file (like adding an extra space somewhere), save it and then do Maven -> Update Project, and this error should disappear.

Although simple logging settings can be accomplished in `application.properties` is adequate, more complex settings are better set up via a separate logging configuration file.

When a file in the classpath has one of the following names, Spring Boot will automatically load it over the configuration provided in `application.properties`:

- `logback-spring.xml`
- `logback.xml`
- `logback-spring.groovy`
- `logback.groovy`

Spring recommends using the `-spring` variant over the plain ones whenever possible

Delete the current contents of `application.properties`
In `src/main/resources`, place the file `logback-spring.xml`

Restart the app and use a REST client to send multiple GET requests to:

http://localhost:8080/showlogs

Check the log output from the app in the Console view.

Examine the settings in `logback-spring.xml`
Notice that the log output is also placed in a file whose location is given by the values of the properties `LogDirectory` and `Logfile` in `logback-spring.xml` The location is relative to the root folder of the project (BootConfigApp).

Change the logging levels for the entire framework and also for a specific package and restart the app. Note the logging output in the Console view as previously.
You can also limit the log output to console only or file only by commenting out the relevant elements in the root level logger and the package level logger.

```
<appender-ref ref="Console" />
<appender-ref ref="File" />
```

Try this out and verify the results for yourself.

4.2 Setting up properties

We can also place other properties in `application.properties` which we can use to configure the behaviour of the Spring Boot application as well as to access within the application itself.

The full list of configurable properties are available at:
https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html

In `src/main/resources`, make the following change:

`application.properties-v2`

In `src/main/java`, make the following change:

`LoggingController-v2`

Before restarting the app, check that 9090 is a free port. If it is not, locate a free port on your machine and change the value accordingly in `server.port` in `application.properties`

Restart the app and use a REST client to send multiple GET requests to:

http://localhost:9090/superman/showlogs

Firstly, notice that the Spring banner no longer appears when the Spring framework starts up, due to the property setting:

```
spring.main.banner-mode=off
```

Secondly, to change the default port of 8080 for the embedded Tomcat server, we use the property setting:

```
server.port=9090
```

Thirdly, notice that we now have a slightly modified URL to access the relevant @RequestMapping method (`superman/showlogs`), instead of the original /`showlogs`. Spring Boot by default serves content on the root context path (/). For situations where we would like to have a custom path, we can use the property:

```
server.servlet.context-path=/superman
```

Lastly, notice that we can retrieve key-value pairs (`schoolName=Jefferson High School`) declared in via the `@Value` annotation in the `@RestController` class.

There are occasions when we may want the Spring Boot app to run as a console-based app rather than starting up in the embedded Tomcat server. To do this, simply add the following to `application.properties`

```
spring.main.web-application-type=none
```

Restart the app. Notice now there are no start up messages for the app related to the Tomcat server. If you attempt to use a REST client to send GET requests to:

http://localhost:9090/superman/showlogs

you will obtain an error message as there is no server running on any port.

Remove the line that you added in just now so that the app continues to start up in the Tomcat server.

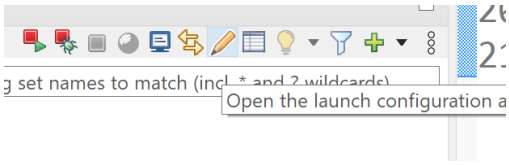
4.3 Implementing start up logic and passing command line arguments

Occasionally we may want specific start up logic to run as soon as the app is deployed in the Tomcat server. At the moment, code within the @RestController class is only executed in response to an incoming HTTP request delegated via the Spring MVC framework.

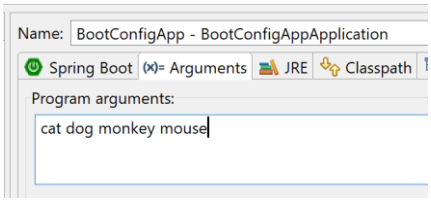
We can use a class that implements the CommandLineRunner interface and its `run` method to provide start up logic.

In `src/main/java`, place the class: `CommandLineAppStartupRunner`

Stop the app if it is still running.
Open the launch configuration for the app



Provide the following arguments:



Click Apply and Close. Start the app in the usual way.

Notice now that the log statements from the `run` method in `CommandLineAppStartupRunner` appear in the console view immediately after the framework starts up (without the need to send any HTTP GET requests to the app). The arguments we provided earlier are listed as well. Notice it also includes a predefined property `--spring.output.ansi.enabled=always` passed as default to configure the logging output.

Stop the app.

Lets produce an executable JAR file for this app.
Right click on the project entry, select Run As -> 5 Maven Build
For the Goals field, type: `clean package`
Then click Run.
Right click and Refresh the project.

Open a command prompt and navigate to the `target` subfolder of this project.
A shortcut for this is to right click on the target subfolder in the Project Explorer, select Show in Local Terminal -> Terminal.

At the terminal type:

```
java -jar BootConfigApp-0.0.1-SNAPSHOT.jar cat dog monkey
```

Verify that we get the same log output as before when running from inside STS. Close the terminal.

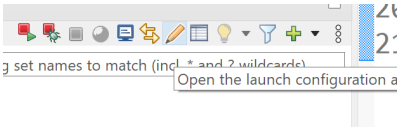
If you are passing arguments in the form of properties (i.e. *`key=value`* pairs), a better option might be to use a class that implements the `ApplicationRunner` interface and its `run` method to provide start up logic.

Property arguments are provided in the form of `--key=value`
They can be retrieved using the various methods of the ApplicationArguments class.
Property arguments can also be used to initialize values of fields marked with @Value.

In `src/main/java`, place the class:

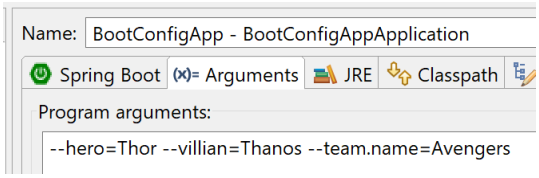
```
AppStartupRunner
```

Stop the app if it is still running.
Open the launch configuration for the app



Provide the following property arguments:

```
--hero=Thor --villian=Thanos --team.name=Avengers
```



Click Apply and Close. Start the app in the usual way.

Notice now that the log statements from the `run` methods in both `AppStartupRunner` and `CommandLineAppStartupRunner` appear in the console view immediately after the framework starts up (without the need to send any HTTP GET requests to the app). Notice that the `team.name` property argument value is used to initialize the field `myTeam` which is also logged to the console.

We can also pass configuration properties for the Spring Boot app in a similar way as well, for e.g.
`--server.port=xxxx`
`--spring.main.banner-mode=off`

Any properties that we pass as command line arguments will automatically override existing properties in `application.properties`

For e.g. add the following

```
--server.port=8080
```

to the existing property arguments in the app launch configuration as we did earlier.
Click Apply and Close.

In `application.properties`, comment out this property to allow the app to start up in the embedded Tomcat server in the usual manner:

```
#spring.main.web-application-type=none
```

Start the app in the usual way.

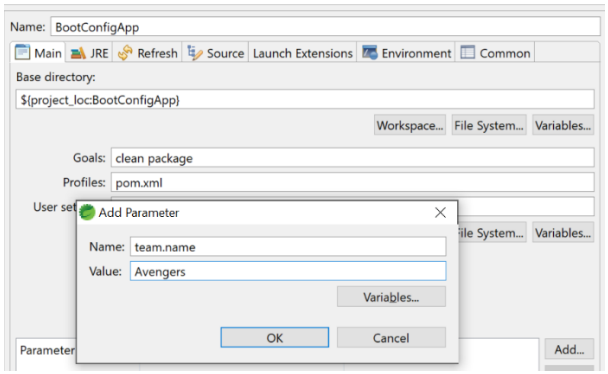
To access the app on the Tomcat server, we now need to use port 8080

http://localhost:8080/superman/showlogs

Stop the app.

Lets produce an executable JAR file for this app. Before doing this, make sure you have closed any terminal windows in the `target` folder as this will interfere with the Maven `clean` goal.

Right click on the project entry, select Run As -> 5 Maven Build
For the Goals field, type: `clean package`
and remember to also add a parameter value for `team.name` as this is required for DI for the field, without which the build process will fail.



Then click Run.
Right click and Refresh the project.

Open a command prompt and navigate to the `target` subfolder of this project.
A shortcut for this is to right click on the target subfolder in the Project Explorer, select Show in Local Terminal -> Terminal.

At the terminal type:

```
java -jar BootConfigApp-0.0.1-SNAPSHOT.jar --hero=Thor --villian=Thanos --team.name=Avengers --server.port=8080
```

Verify that we get the same log output as before when running from inside STS.

At this point, we have two classes that provide start up logic (`AppStartupRunner` and `CommandLineAppStartupRunner`). We might want to determine the order in which their `run` methods are invoked in. Since both these classes are beans (marked with @Component), we can fix their order of invocation using the @Order annotation.

Above the @Component in `CommandLineAppStartupRunner`, add this annotation:

```
@Order(value=1)
```

and save.

Above the @Component in `AppStartupRunner`, add this annotation:

```
@Order(value=2)
```

and save.

Restart the app. Notice the order in which the run methods are invoked (`value=1` comes before `value=2`). Swap the `@Order` values for both classes and run again to verify this.

4.4 Deploying to an external Tomcat server

While deploying the app in an embedded Tomcat server is useful for facilitating the development process, at some point of time it is likely you will need to package and deploy the app to an external Tomcat server or application server.

Shut down the app if it is still running.

Make the following changes to the `pom.xml`

- Just below the GAV elements, add in an element to specify WAR packaging:

```
<packaging>war</packaging>
```

- In the `<build>` section, modify the final WAR file name to a simpler form to add in the deployment (since the WAR filename will become the context root that is part of the URL path):

```
<build>
    <finalName>${artifactId}</finalName>
    ...
</build>
```

- Next, add in a Tomcat dependency:

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

Save your changes and do a Maven -> Update Project.

- Initialize the Servlet context required by Tomcat by implementing the SpringBootServletInitializer interface in the @SpringBootApplication class:

```
package com.workshop.boot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.boot.web.servlet.support.SpringBootServletInitializer;

@SpringBootApplication
public class BootConfigAppApplication extends SpringBootServletInitializer {
```

- Provide all the necessary properties in the `application.properties` file that is necessary to instantiate all dependencies in the app (e.g. `@Value("${schoolName}")`, `@Value("${team.name}")` and also enable access logging in the Tomcat server

Make the changes in `application.properties-v3`

We can now produce a deployable WAR for this app.
Right click on the project entry, select Run As -> 5 Maven Build
For the Goals field, type: `clean package`

Then click Run.
Right click and Refresh the project.

The WAR file should now be in the `target` subfolder of this project. Copy this WAR to a suitable location (i.e. Desktop)

Start up the Tomcat instance that is running as a Windows service from the Services dialog box. Navigate to the port that is deployed on, for e.g. : http://localhost:8181/

There are several ways to deploy a web app into a standalone Tomcat server instance:
https://tomcat.apache.org/tomcat-9.0-doc/appdev/deployment.html

Click on Manager App and complete the username / password prompt. You will be navigated to the Tomcat Web Application Manager main page.

In the WAR file to deploy section, select Choose File button. Then select `BootConfigApp.war` and click Deploy. You should be able to see `/BootConfigApp` among the list of deployed applications; click on this link to be redirected to the app.

http://localhost:8181/BootConfigApp/

Verify that the functionality is exactly the same as when it was running in the embedded Tomcat server in STS by navigating to:

http://localhost:8181/BootConfigApp/showlogs

To undeploy, simply click on the Undeploy button for the `/BootConfigApp` entry. The application disappears from the list and the project folder is also removed from `webapps`

Another way to deploy is to simply copy and paste `BootConfigApp.war` into the `webapps` folder. After a short while, Tomcat will automatically unzip this WAR into a project folder and the app is now deployed (you should be able to see it listed in the Tomcat Web Application Manager).