

Enterprise Java with Spring

Spring REST API

Lab 1

1	LAB SETUP	1
2	USING A BROWSER AS A REST CLIENT	1
3	USING POSTMAN AS A REST CLIENT	4
4	CREATING A SPRING BOOT WEB MVC APPLICATION VIA SPRING INITIALIZR	13
4.1	TOMCAT SERVER PORT ISSUES	17
4.2	BASIC WEB APPLICATION CONFIGURATION	18
4.2.1	Changing default Tomcat port	18
4.2.2	Setting logging level for log messages from Spring framework and application	19
5	CREATING A SPRING BOOT REST PROJECT VIA SPRING INITIALIZR	22
5.1	CREATING AN EXECUTABLE JAR	25
6	CREATING SPRING BOOT REST PROJECTS VIA SPRING TOOL SUITE (STS)	25
7	MONITORING REST HTTP TRAFFIC WITH TCP/IP MONITOR IN ECLIPSE	32

1 Lab setup

Make sure you have the following items installed

- Latest LTS JDK version (at this point: either JDK 21)
- A suitable IDE (Eclipse Enterprise Edition for Java) or IntelliJ IDEA
- Latest version of Maven (at this point: Maven 3.9.9)
- A free account at Postman and installed the Postman app
- A suitable text editor (Notepad ++)
- A utility to extract zip files (7-zip)

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 Using a browser as a REST client

There are a few sites on the Web that provide fake REST APIs for purposes of testing and prototyping.

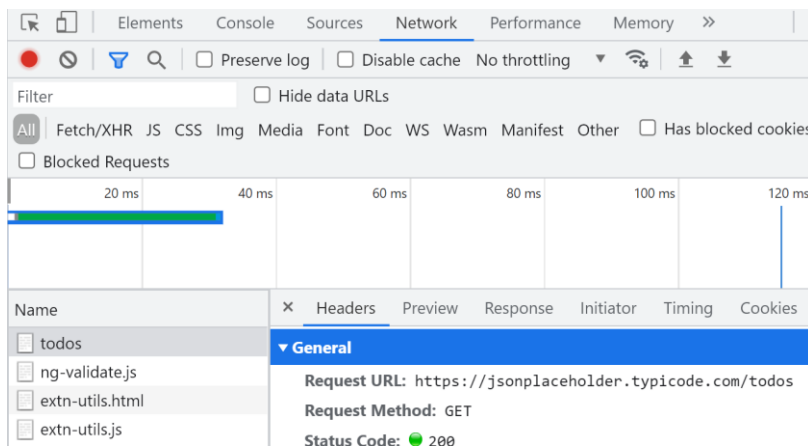
A well-known one is:

<https://jsonplaceholder.typicode.com/>

Open a browser tab at this URL and scroll down to the end where you see the subtopics Resources and Routes. Click on any of the links highlighted in green. This sends out a HTTP GET request from the browser to the specified API endpoint whose URL you will be able to see in the address bar. Notice that the complete URL for a given endpoint is the server name (<https://jsonplaceholder.typicode.com>) appended with the path portion (`/posts`) for that endpoint.

You can verify that the returned content is in the form of JSON.

If you wish to inspect the Headers in the request and response messages, and Response in detail, switch to the Network view tab in the Developer Tools when clicking on the URL (For Chrome: More Tools -> Developer Tools to access this, a similar tab view exists for other major browser Developer Tool environments).



The Preview view shows the returned JSON formatted in a more structured form.

Notice that the links for POST, PUT, PATCH and DELETE in the ROUTES section are not highlighted because you are not able to activate any of these HTTP methods: since clicking on a browser link simply sends a HTTP GET request.

For each of the API endpoints for the 6 common resources, you can append a number to the end preceded by a forward slash to retrieve a single item from the collection of resources, for e.g.

<https://jsonplaceholder.typicode.com/posts/5>

<https://jsonplaceholder.typicode.com/comments/2>

<https://jsonplaceholder.typicode.com/photos/6>

This number will typically, for the majority of REST APIs, represent a unique ID for the resource in question. Make sure that the number that you append at the end does not exceed the number listed for that particular resource (e.g. 100 posts, 500 comments, etc).

You can also use query parameters (the portion after the ? at the end of the URL path) in your request to drill down to a single item or subset of items from the collection of resources available based on an id for the item, for e.g:

<https://jsonplaceholder.typicode.com/comments?postId=5>

<https://jsonplaceholder.typicode.com/comments?postId=2&id=7>

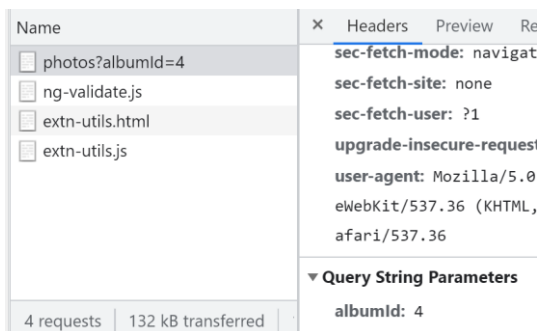
<https://jsonplaceholder.typicode.com/comments?id=10>

<https://jsonplaceholder.typicode.com/posts?id=5>

<https://jsonplaceholder.typicode.com/posts?userId=3&id=23>

Be careful with case as the path components of the URL after the domain name (jsonplaceholder.typicode.com) are case-sensitive: so `postId=5` is not the same as `postId=5`

If you check the Network view in the Developer Tools, you will see the query string parameters shown at the bottom, for e.g.



You can use any of these online JSON validators to verify that the returned response is proper JSON: just copy and paste the responses into the validator views.

<https://jsonlint.com/>

<https://jsonformatter.org/>

<https://codebeautify.org/jsonvalidator>

Experiment with introducing deliberate errors in the JSON formatting in these validators and see how they flag these errors. The second site (<https://jsonformatter.org/>) also provides options for converting the JSON to its equivalent representation in other related formats such as XML and YAML, as well

In order to test out the other REST HTTP methods such as POST, PUT, etc, you will need to run the JavaScript code to send out these methods. Go to this link:

<https://jsonplaceholder.typicode.com/guide/>

Switch to the Console view in the Developer Tools and copy one of the Javascript snippets and paste it into the Console and press enter. You should be able to see the returned response in the Console view. In the Network view, you will also be able to see the request details (headers and content). Select to filter based on Fetch/XHR to be able to locate the relevant request more quickly.

In a real functional REST API, requests such as PUT, POST, PATCH should modify or add to the content of the resources stored on the server side. Then, subsequent GET requests should be able to retrieve this modified or newly added content. However, for this fake API, no such modification is performed (due to the heavy traffic load from multiple users testing out this API), so subsequent GET requests that you issue will still show identical content.

Scroll down to the bottom of the page to see the nested routes that are available:

For e.g.

`/posts/1/comments`

would mean to retrieve all the comments related to the post with the id of 1

`/users/1/photos`

would mean to retrieve all the photos related to the user with the id of 1

This nesting relationship between resources would be reflective of the Entity Relationship Diagram in the relational table that is used to store the records.

If you return to the main page:

<https://jsonplaceholder.typicode.com/>

you will see that there is fixed number on the 6 common resources (for e.g. 100 posts, 500 comments and so on). If you attempt to access a resource ID beyond the range for the specified resource, you are effectively attempting to access a non-existent resource. Try this now by typing this URL into a browser tab:

<https://jsonplaceholder.typicode.com/posts/2000>

In the Network view of the Developer Tools, you should an entry with status 404 outlined in red. Responses outlined in red indicate either a client-side error (code 4xx) or server-side error (code 5xx). In this case, there is nothing contained in the response body.

You can repeat this with other resources, e.g.

<https://jsonplaceholder.typicode.com/comments/1000>

In fact, any kind of URL which is not matched by the server-side API code to a resource will result in a 404 error, for e.g.

<https://jsonplaceholder.typicode.com/asdfasdf/2323>

3 Using Postman as a REST client

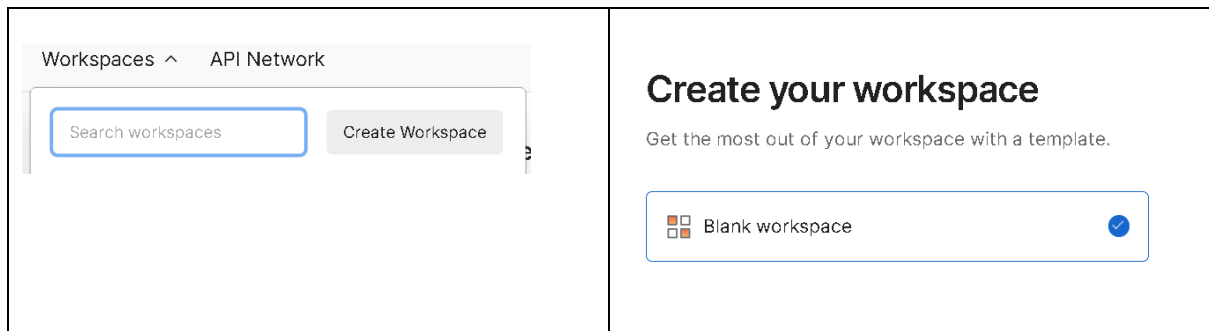
The disadvantages of using a browser as a REST client is that it is only able to issue HTTP GET requests. To issue other HTTP REST request methods, we would have to type (or copy and paste) relevant

JavaScript code snippets in the Console view of the Developer tools, which is less than ideal for rapid testing.

There are many GUI-based REST clients which provide access to the full range of REST HTTP methods, as well as ability to tailor the request headers and bodies as well as manipulate and store the results. The most popular and widely used of this at the moment is Postman

Start Postman (this process might take some time if there is an automatic update in the background during start up) and login using your Gmail account or existing account username or password.

We will create a new workspace for this lab. Select the Workspaces option from the main menu, and select Create New Workspace. Choose a Blank Workspace and click Next.



Providing the following details for the workspace for Only me (Personal)

Name: FirstLab

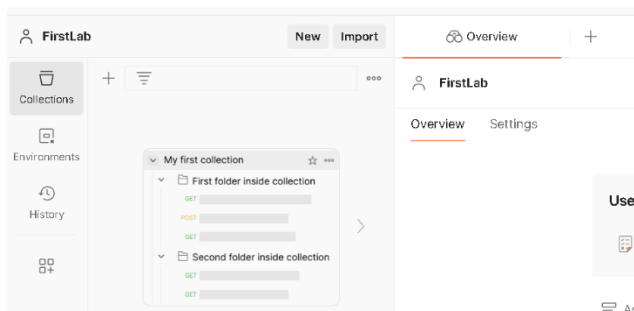
Create your workspace

Name

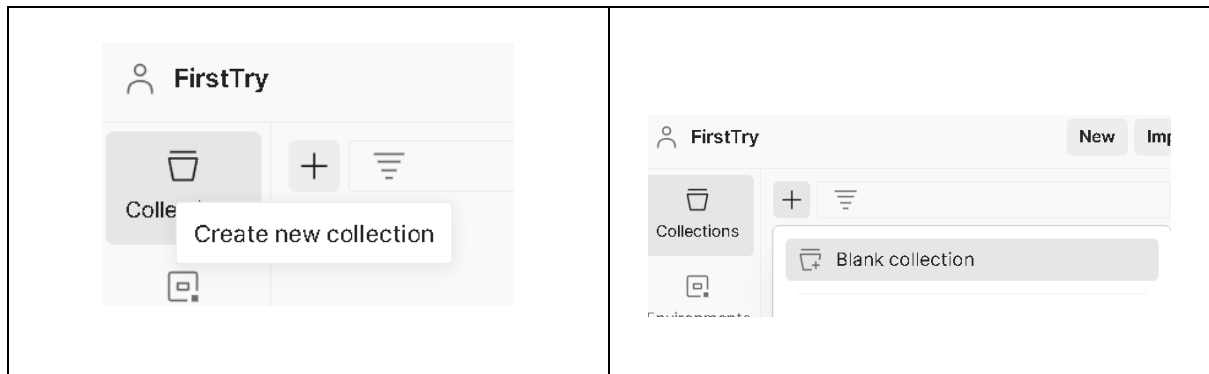
Who can access your workspace?

☒ Only me
Personal

Then select the Create button. Postman will spend a couple of minutes creating the workspace and navigate you into it.



A workspace consists of a series of collections, which in turn contain a group of one or more HTTP requests. We will first click the Create Collection button to create a collection, which we will name FirstCollection.



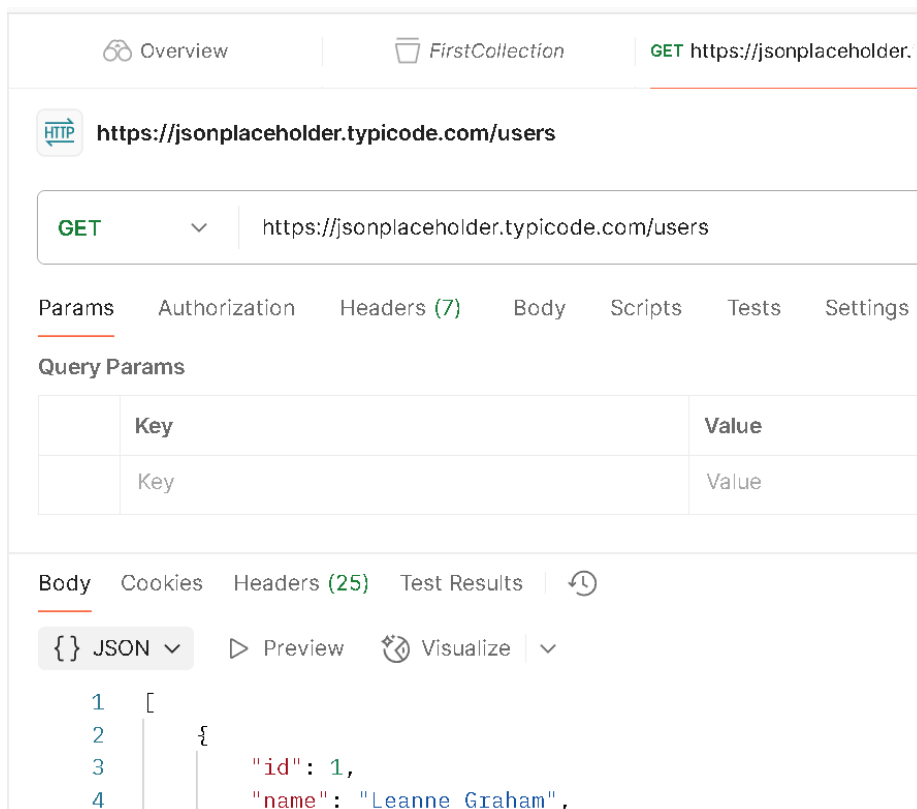
Then we will create a new HTTP request by any of these actions below:

- Clicking on New icon, selecting the HTTP icon
- Clicking on the + icon next to the most recent tab in the main view
- Click on the hamburger Main Menu icon in the upper left hand corner, select File -> New Tab

Type in the complete URL for the API endpoint that you want to send a request to and select the method you wish to use (by default this is GET). For e.g. for the API endpoint:

<https://jsonplaceholder.typicode.com/users>

then click the blue Send button



The top part of the main view shows details regarding the outgoing HTTP request issued by Postman, for which you can customize in relevant ways: for e.g. by selecting HTTP method to be used, setting query string parameters, putting content in the body and creating new headers or modifying values for existing ones.

The bottom part of the main view show details regarding the incoming HTTP response received by Postman. This includes relevant details such as status code, headers and the body of the content (which can be viewed in a variety of ways by selecting the Pretty, Raw, Preview tabs as well as changing between JSON, XML, HTML formatting, etc).

Create two more new requests to send a GET request to the following API endpoints:

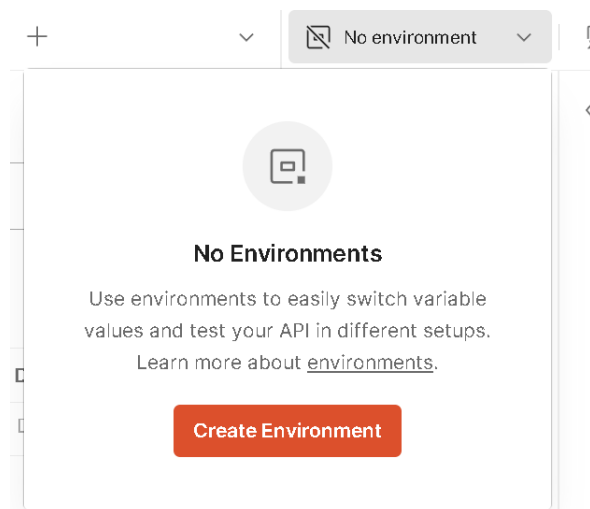
<https://jsonplaceholder.typicode.com/posts>

<https://jsonplaceholder.typicode.com/comments/2>

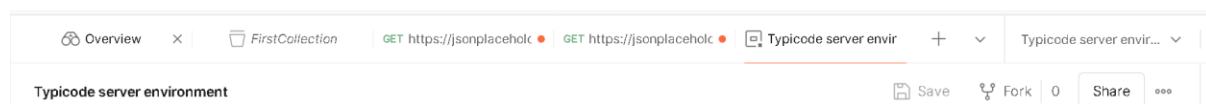
Examine the contents of the response.

So far we have been typing in the entire URL in the address bar. Since we are essentially using the same base URL () repeatedly for all our requests, we could store it in a temporary variable which we can then substitute in when typing the URL to reduce its length. Postman provides a facility for this.

Click on the Environment icon in the upper right hand corner and then click Create Environment.



In the Environment tab, give this new environment the name: Typicode server environment



Add this variable entry

Variable: dummyAPI

Initial value: `https://jsonplaceholder.typicode.com`

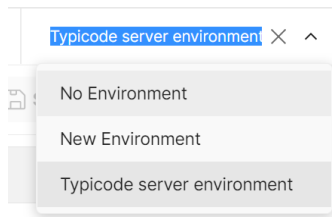
Typicode server environment Save Fc

Filter variables

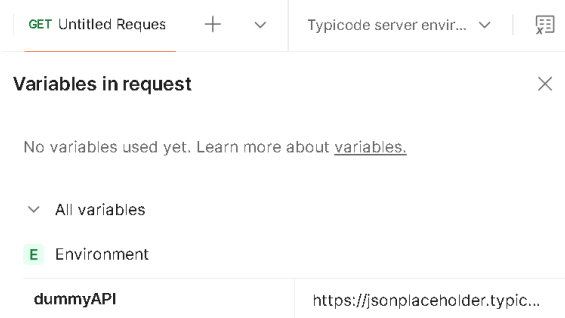
	Variable	Type	Initial value	Current value
<input checked="" type="checkbox"/>	dummyAPI	default	https://jsonplaceholder.typicode.com	https://jsonplaceholder.typicode.com

Then click Save and close the tab.

Create a new Request tab. Select the environment you just created from the drop down list at the upper right hand corner. This makes all the variables you have defined in that environment available to use in the current request you are forming.



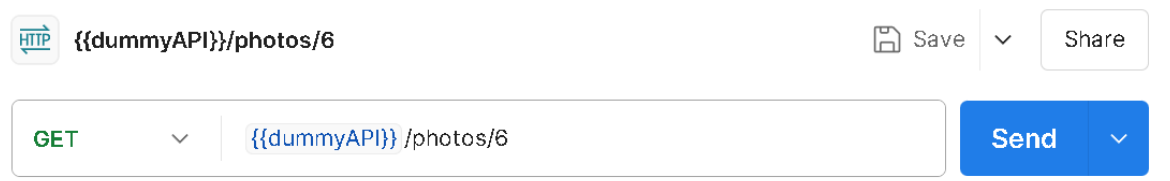
Click on the Environment quick look icon on the upper right hand corner to confirm that the variable is still available in the Typicode Server environment.



Now type in the URL to make the request to, but this time using the variable you defined earlier

```
{{dummyAPI}}/photos/6
```


The variable should be highlighted to indicate that it will be substituted with the actual value by Postman before the request is made.






Create a GET request with query parameters:

```
{{dummyAPI}}/posts?userId=3&id=23
```

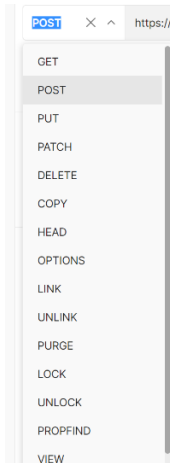

Notice that the parameters are shown in the Params view, and you can change them in this view which will result in a reflected change in the URL itself in the main address bar and vice versa. You can deselect specific key-value pairs or add in new key-value pairs in this view.

GET  https://jsonplaceholder.typicode.com/posts?userId=3&id=23

Params  Authorization Headers (6) Body Pre-request Script Tests Settings

	KEY	VALUE
	userId	3
	id	23
	Key	Value

Let's try sending other types of HTTP requests. The drop down list next to the method type gives you list of request methods available, which you can see is more than the standard ones associated with a typical REST API.



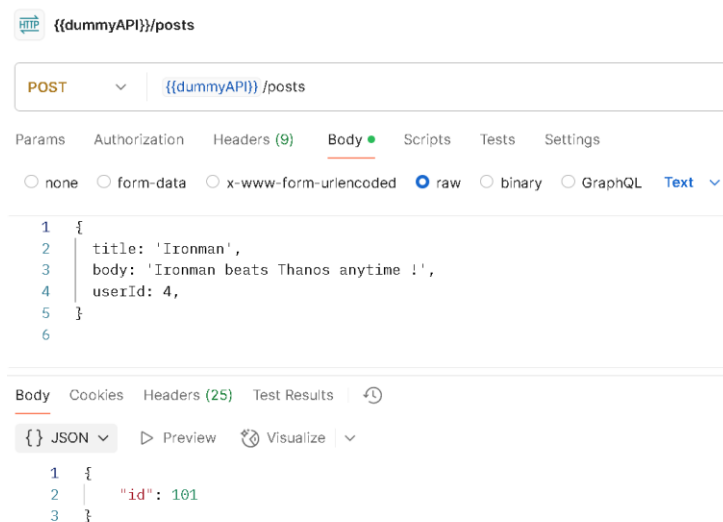
We can send a POST request to:

```
{{dummyAPI}}/posts
```

with the following body content:

```
{
  title: 'Ironman',
  body: 'Ironman beats Thanos anytime !',
  userId: 4,
}
```

This how we would place the content in Postman, using the `raw` and `Text` option, and the click `Send` to transmit it to the server.



Notice the response has a status 201 Created, which is a typical status for a successful creation of a new resource on the server end. The body of the response contains the id of the newly created resource (in this case 101) because there are currently 100 posts in the collection. However, as mentioned earlier, this is a fake API and there is no actual creation of a new resource on the server end. Thus, if you keep issuing multiple POST requests to the same API endpoint, you will constantly keep getting back the same response.

You can try sending a POST request to:

`{{dummyAPI}}/comments`

with the content:

```

{
  "id": 1,
  "name": "Peter Parker",
  "email": "spiderman@gmail.com",
  "body": "My webs are unbreakable !"
}

```

and you would get a response with the `id: 501` as there are currently 500 comments in the resource collection.

We can modify a resource with the PUT method. Send a PUT request to this endpoint:

`{{dummyAPI}}/posts/1`

with the following content:

```

{
  "userId": 1,
  "id": 1,
  "title": "Marvel comics",
  "body": "I love the MCU. Its awesome !"
}

```

Notice that the content contains all the fields expected in a single `posts` resource, because a PUT essentially overrides the entire content of that resource. The response is a 200 OK with the id of the resource being updated.

We can also modify a resource with the PATCH method. Send a PATCH request to the same endpoint:

```
{{dummyAPI}}/posts/1
```

with the following content:

```
{
  "title": "DC comics",
}
```

Notice that the content only contains the field to be changed and not all the fields expected in a single `posts` resource. So PUT does a complete update of the entire resource identified, while PATCH does a partial update. This is the main difference between the two.

The response is still a 200 OK but the body of the response contains the entire resource that was being modified (and not just the id as in the case of a PUT). This allows the client to also verify the contents of the other fields in the modified resource. In this example here, the `title` field remains unchanged because this is a fake API, but in a real API this would now have the new value of `DC comics`

Finally, we can send a DELETE request to the same API endpoint without any body content:

```
{{dummyAPI}}/posts/1
```

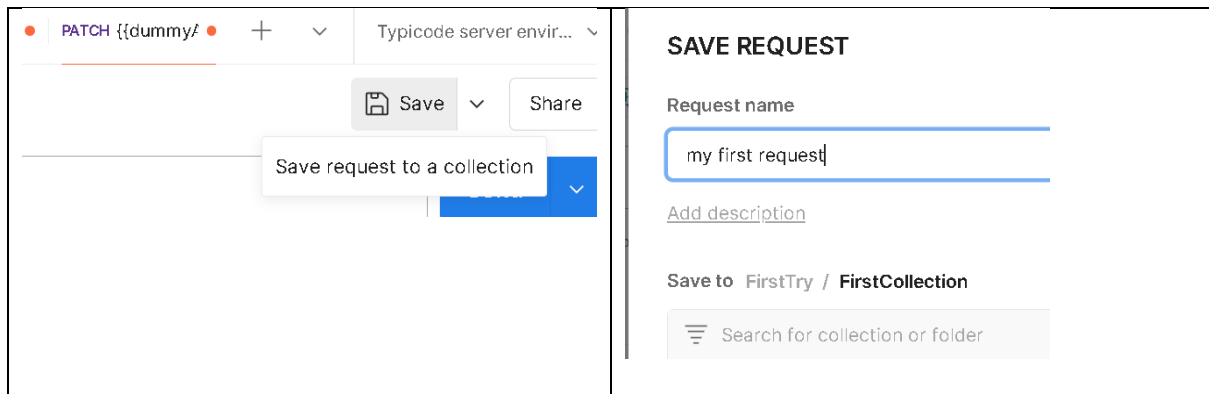
The response is a 200 OK with an empty body. There is no need to provide any content in the response, the 200 OK status indicates that the delete operation was successful. Again, because this is a fake API, the resource is not actually deleted so if you attempt a GET to the same API endpoint, it will succeed. In a functional REST API, attempting a GET to a resource that has already been deleted should return a 404 Not found error.

You can experiment around with sending different REST HTTP requests (GET, POST, PUT, PATCH, DELETE) to multiple API endpoints, keeping in mind that you will not get proper effect on the server-side end as this is a FAKE API.

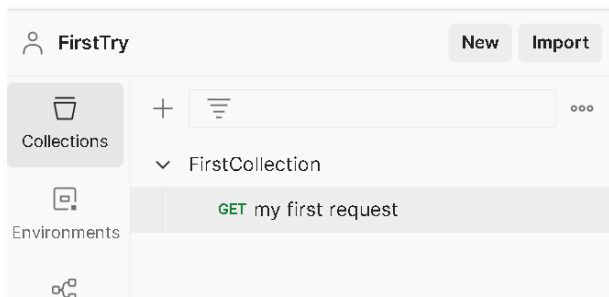
When you are done, you can save the numerous requests you have made in a Collection - which is a way to gather together related requests.

Select any request by clicking on its tab, and then select Save. You will be presented with a dialog box to provide a new name for the request and the collection it will be saved in. You can select the FirstCollection we created earlier, and give any appropriate name (for e.g. GET request to POST endpoint, etc) and then click Save.

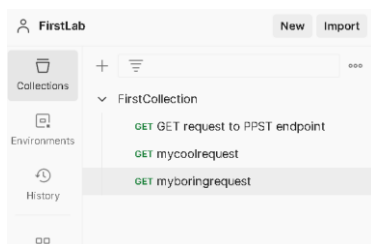
--	--



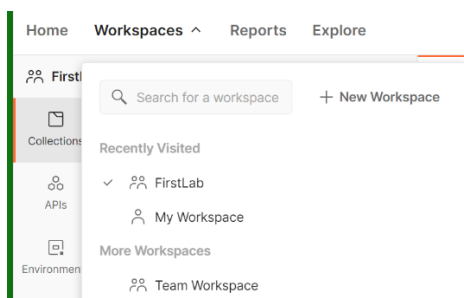
You should now be able to see the newly created collection and the single saved request in the left hand pane.



Close the tab for the request that you have saved. Repeat the save operation for a few (not necessarily all) of the other open tabs for unsaved requests, saving them to the same collection and then close them. When you are done, you should be left with the Overview pane and the list of saved requests in the collection on the left.

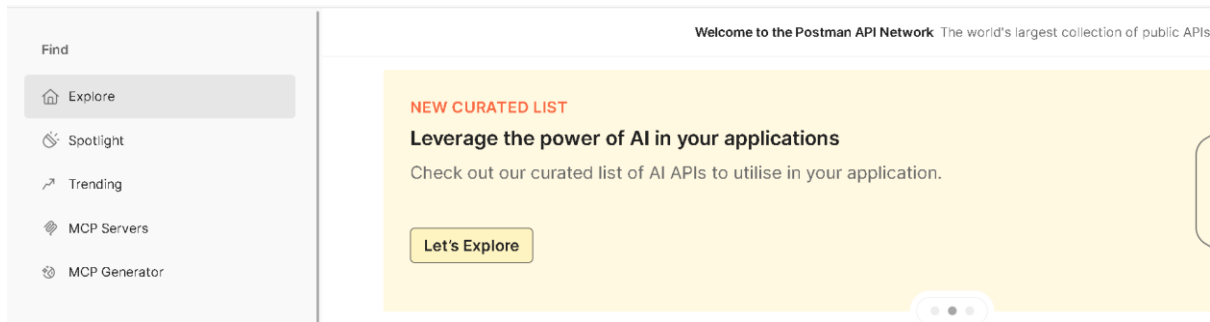


Close the Postman app and start it up again. Select Workspaces and open up the workspace that you created (FirstLab), and you should be able to see the collection you created and all the requests currently saved in it.



Click on any of the saved requests to retrieve it, then you can work on it in the usual manner and then send it.

Click on the API Network option on the main menu, which allow you to view a large collection of public API requests. This collection is analogous to GitHub: it is a public repository of collections created by Postman users globally to share with others. You can use the left hand pane to zoom in on specific collections of public API requests.



When you are done, click on the Workspaces -> FirstLab to return back to the current workspace.

A popular global marketplace of public REST APIs is [RapidAPI](#)

4 Creating a Spring Boot Web MVC application via Spring Initializr

The source code for this lab is found in `First-Spring-Rest/changes` folder.

Navigate to the Spring Initializr at:

<https://start.spring.io/>

Key in the values for the fields as shown below as well as the following 2 dependencies:

Group: `com.workshop`
Artifact: `FirstSpringBootMVC`
Name: `FirstSpringBootMVC`
Description: `Demo project for Spring Boot`
Package: `com.workshop.mvc`

- Spring Web - this is a Spring Boot Starter template which includes all the dependencies required to build a complete web application as well as REST API web services.
- Spring Boot DevTools - this provides live reloads of the embedded Tomcat web server that is included in the Spring Web template. This facilitates application development and debugging by automatically restarting the Tomcat server every time code changes are made to the web application.



Project
☐ Gradle - Groovy
 ☐ Gradle - Kotlin
 ☒ **Java**
☐ Kotlin
 ☐ Groovy
☒ **Maven**

Language
☒ **Java**
☐ Kotlin
 ☐ Groovy

Spring Boot
☐ 3.4.0 (SNAPSHOT)
 ☐ 3.4.0 (RC1)
 ☐ 3.3.6 (SNAPSHOT)
 ☒ **3.3.5**
☐ 3.2.12 (SNAPSHOT)
 ☐ 3.2.11

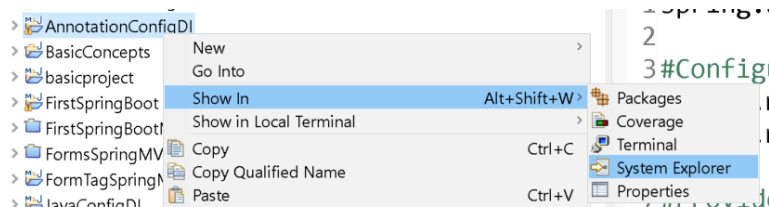
Project Metadata
 Group
 Artifact
 Name
 Description
 Package name
 Packaging ☒ **Jar** ☐ War
 Java ☐ 23 ☒ **21** ☐ 17

Dependencies
Spring Web WEB
 Build web, including RESTful, applications using Spring MVC embedded container.
Spring Boot DevTools DEVELOPER TOOLS
 Provides fast application restarts, LiveReload, and configurat experience.

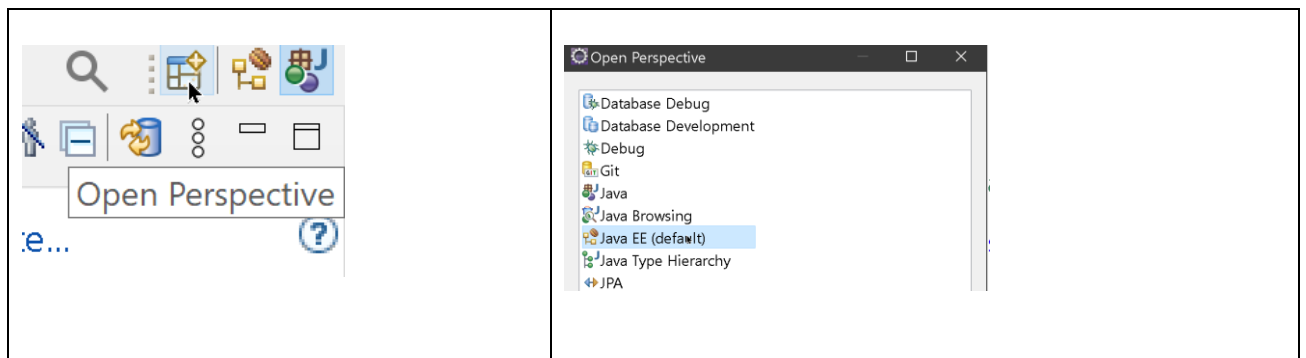
Click Generate.

Download the Zip file and extract its contents with 7-zip. Place the extracted folder in your Eclipse workspace directory.

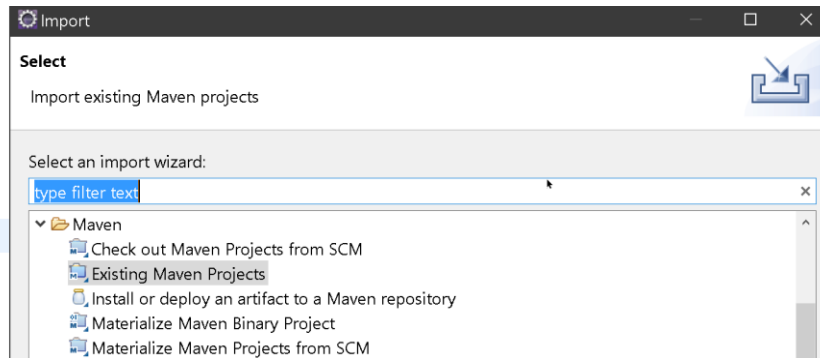
You can navigate to workspace directory from Windows Explorer. An alternative shortcut is to open the subfolder of any existing project in the workspace directory by right clicking on any existing project entry in the Project Explorer pane, then selecting Show in -> System Explorer



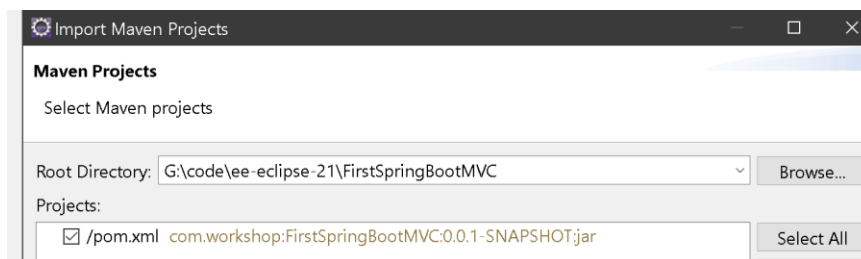
Switch to Java EE perspective if you are not already in there, either using the icons at the upper right hand corner or via the menu: Window -> Perspective -> Open Perspective



Go to File -> Import -> Maven -> Existing Maven Project.



Click Next, select Browse, and select the FirstSpringBootMVC folder, which should also automatically highlight the pom.xml in that folder. Click Finish.



Open the `pom.xml`.

In the dependency hierarchy tab, note that the `spring-boot-starter-web` includes a variety of other transitive dependencies, including the basic `spring-boot-starter` that comes with all Spring Boot projects. The other dependencies are:

- `spring-boot-starter-json` – provides support for JSON, a popular format for message interchange between web applications and web services. It includes libraries like Jackson, which is a popular JSON processing library for Java. Jackson automatically converts Java objects to JSON and vice versa, making it easy to work with JSON data.
- `spring-boot-starter-tomcat` – provides an embedded Tomcat server. Tomcat is an open source Java servlet container that allows Java application to run in a web-based environment. In a Spring MVC application, Tomcat serves as the web server and servlet container, handling HTTP requests and directing them to the appropriate servlets
- `spring-web` – provide foundational web-related functionality to build web applications such as HTTP client libraries to handle and process HTTP request and responses.
- `spring-webmvc` – implements the Model-View-Controller (MVC) framework in Spring, including key components such as `DispatcherServlet`, `Controller`, `ViewResolver`, and `ModelAndView`. It builds on the basic functionality of `spring-web`.

Notice that `spring-boot-devtools` has `runtime` scope, which means application code will not be able to access the libraries from this dependency as it is meant to only be used at runtime when performing a hot reload of the web app. In the same way, `spring-boot-starter-test` has `test` scope and is only accessible to test code available in `src/test/java`.

Every Spring Boot project, including a MVC application such as this, comes with an autogenerated class `xxxApplication` located in the specified package `src/main/java`. Here, the name of the file is `FirstSpringBootMvcApplication.java`

This class is annotated with `@SpringBootApplication`. As we have already seen, `@SpringBootApplication` annotation is actually a shorthand for the combination of 3 separate annotations:

- `@Configuration`
- `@ComponentScan`
- `@EnableAutoConfiguration`

We will start this simple web MVC application in the usual way.

Go to `FirstSpringBootMvcApplication`, right click and select `Run As -> Java Application`

This starts up the embedded Tomcat server, which by default will run at port 8080 on localhost (your Windows machine). You should see some log messages in the Console view similar to the ones below notifying you of this:

```
...
: LiveReload server is running on port 35729
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path
 '/'
: Started FirstSpringBootMvcApplication in 1.799 seconds (process running for 2.249)
o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet
'dispatcherServlet'
o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
o.s.web.servlet.DispatcherServlet        : Completed initialization in 1 ms
...
```

Open a new browser tab at `localhost:8080`. You should get back a `Whitelabel Error Page` similar to the one shown below.



Whitelabel Error Page

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

Sat Nov 02 09:52:22 MYT 2024

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

No static resource .

`org.springframework.web.servlet.resource.NoResourceFoundException: No static resource .`

This indicates the embedded Tomcat server is running and accessible at port 8080, but is currently returning this error page because there is no mapping provided yet for URL paths to servlets in our app.

To stop the running Tomcat server, click on the red stop button in the lower right corner of Eclipse pane.



If you attempt to access `localhost:8080` again (just refresh your browser tab with `F5`), you will get back an error indicating an inability to connect as the embedded Tomcat server is no longer actively listening on this port:



This site can't be reached

localhost refused to connect.

Try:

- Checking the connection
- [Checking the proxy and the firewall](#)

ERR_CONNECTION_REFUSED

You can restart it again in the usual way.

4.1 Tomcat server port issues

When attempting to run the application, the embedded Tomcat server may not be able to bind to its default port of 8080 due to another existing process listening on it, and an error message similar to the below will appear in the console.

```
*****
APPLICATION FAILED TO START
*****
```

Description:

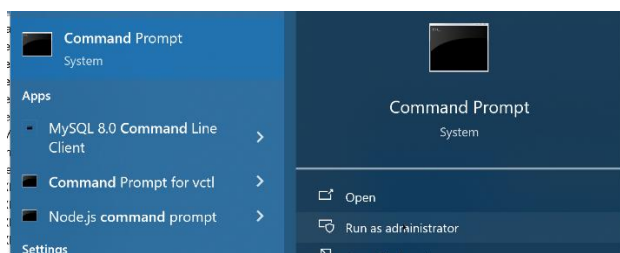
Web server failed to start. Port 8080 was already in use.

Action:

Identify and stop the process that's listening on port 8080 or configure this application to listen on another port.

A common reason for this occurring is accidentally restarting the app without first stopping a previously running instance. This previous instance of Tomcat will still be running on port 8080 which results in a port conflict for the next embedded Tomcat server instance to start up.

On Windows, to check for processes that are bound to the active ports, you will need to open a command prompt with admin privilege



Run the following command to identify process listening on a given *port-number*, for e.g. 8080

```
netstat -aon | findstr :port-number
```

This returns to you the PID of any process that is listening on that given port, for e.g.

TCP	0.0.0.0: <i>port-number</i>	0.0.0.0:0	LISTENING	<i>PID</i>
TCP	[::]: <i>port-number</i>	[::]:0	LISTENING	<i>PID</i>

Next you can find the actual name of the process using its *PID* with the command:

```
tasklist | findstr PID
```

This returns the actual name of the process executable

```
someprogram.exe      PID Console          11      219,532 K
```

Once you have identified the process, you can decide to shut it down if it's not an important process or application that is required. To do this, you can terminate the process with:

```
taskkill /F /PID PID
```

```
SUCCESS: The process with PID PID has been terminated.
```

Another alternative is to start the embedded Tomcat server on a port that is currently not used through configuring `application.properties` as we will see next.

4.2 Basic web application configuration

We have seen in a previous lab on Spring Boot that the `application.properties` file is the central place to define various settings and properties that control the application's behavior. It allows you to define application-wide properties in one place, making it easier to manage configurations without scattering settings across multiple files or hardcoding values.

The full [list of properties that can be set in the file](#)

4.2.1 Changing default Tomcat port

A common configuration option is to change the port number that the embedded Tomcat server starts on from the default value of 8080. This could be for a variety of reasons for e.g.

- a) There is already an existing crucial process on the local machine that needs to use port 8080, and we cannot terminate it
- b) We may wish to specify different ports to be used for different environment that a web application might be deployed in, for e.g. development environment, a QA environment, a staging environment, and a production environment. These environments typically require different configurations for the same application.

If we wish to do this, we will need to first decide on a random port number in the range of 3000 – 20000 to configure the Tomcat server on, and then verify that there is no other process listening on that port with:

```
netstat -aon | findstr :port-number
```

A common alternative port number would be 8081.

We can then set this as the alternative port for the Tomcat in `application.properties` with this entry:

```
server.port=8081
```

Notice that after saving your changes in `application.properties`, the Console log messages are reprinted which indicate that the Tomcat embedded server has restarted on its own and is now running on the new port of 8081. This is due to the hot reload facility introduced by the `spring-boot-devtools` dependency, which simplifies development work since you will no longer need to manually restart the server to reflect the latest changes you have made to the code base.

All future changes you make to `application.properties` and any other source code file in this project will automatically be detected and will restart the Tomcat server in the same way.

You can now remove the property or revert the value back to the default of 8080 (unless you wish to keep it at the new port value for whatever reason)

4.2.2 Setting logging level for log messages from Spring framework and application

Another common configuration option is to change the detail of logging messages displayed in the console when the application starts up.

Logging is an essential aspect of any Spring application, as it provides visibility into the application's behavior, helps in diagnosing issues, and offers insights into how the application is performing.

During development, logging helps developers understand the flow of the application more clearly. It helps in troubleshooting and debugging errors as developers can trace back errors to their source, analyze the sequence of events, and better understand what led to a specific problem. To control the amount of log messages produced (the verbosity), we can set different logging levels (TRACE, DEBUG, INFO, WARNING, ERROR). We can then toggle these levels appropriately corresponding to different environments (e.g., more detailed logs in development stage and minimal logs in production stage).

In a Spring Boot application, the conventional logging library used by default is SLF4J (Simple Logging Facade for Java) in combination with Logback.

Some of the more popular and widely used logging libraries or frameworks in Java are:

- a) Logback
- b) Log4j 2
- c) Java Util Logging
- d) Apache Commons Logging

SLF4J acts as a facade, or abstraction layer, for various logging frameworks. This means it provides a standard API with a list of method calls which then needs to be implemented by any underlying logging framework. This standard API facade allows developers to switch the underlying logging framework without modifying the application code, making it highly flexible and widely adopted.

Logback is the default logging framework used in Spring Boot and serves as the underlying implementation of SLF4J.

You can see both SLF4J and Logback as transitive dependencies of `spring-boot-starter` in the project POM.

Dependency Hierarchy

```
▼ spring-boot-starter-web : 3.3.5 [compile]
  ▼ spring-boot-starter : 3.3.5 [compile]
    spring-boot : 3.3.5 [compile]
    spring-boot-autoconfigure : 3.3.5 [compile]
  ▼ spring-boot-starter-logging : 3.3.5 [compile]
    ▼ logback-classic : 1.5.11 [compile]
      logback-core : 1.5.11 [compile]
      slf4j-api : 2.0.16 (managed from 2.0.15) [compile]
    ▼ log4j-to-slf4j : 2.23.1 [compile]
      log4j-api : 2.23.1 [compile]
      slf4j-api : 2.0.16 (managed from 2.0.9) [compile]
```

The 5 standard logging levels common with most logging frameworks in order of increasing severity:

- trace
- debug
- info
- warn
- error

Make changes to following file in `com.workshop.mvc` in `src/main/java` from changes:

`FirstSpringBootApplication.java`

To configure different logging levels for code in classes in specific packages, we need to specify the package name as a property in `application.properties`, for e.g.

```
logging.level.packagehierarchy=logginglevel
```

By default, logging level is set to `info` in a Spring Boot project: both for the existing components of the Spring framework as well as any other application code added in by a developer.

Add this entry to `application.properties` and save

```
logging.level.com.workshop.mvc=error
```

The application should restart. In the console, you should see the standard log messages from the Spring framework which is at INFO level, and a single output message corresponding to the `logger.error` statement in `FirstSpringBootMVCApplication`, since this class is placed in the package specified in the property above: `com.workshop.mvc`

The log messages for the Spring framework is at the default level of INFO as we did not explicitly change it, but we now explicitly specify that only log messages at the ERROR level are output from the application code through the newly introduced property.

Change the value of this property to all the possible different log level values and save to see the effect each time.

```
logging.level.com.workshop.mvc=xxxx
```

Set `xxxx` is set to warn, info, debug and trace

The application should reload each time after you make a change and save `application.properties` (due to the live reload facility introduced by `spring-boot-devtools`) and you will be able to see different log output messages from the application itself.

The basic principle for the log output is that all log messages at the configured severity and higher will be output. As an example, if we configure the property to be:

```
logging.level.com.workshop.mvc=info
```

Then all the log statements in application code at that specified severity level (`info`) and any level higher than it (`warn`, `error`) will be executed, which will be these 3 statements:

```
logger.info("FirstSpringBootMvcApplication: Message at INFO level");
logger.warn("FirstSpringBootMvcApplication: Message at WARN level");
logger.error("FirstSpringBootMvcApplication: Message at ERROR level");
```

To change the logging level for the Spring framework itself, we specify the top level package for all Spring framework classes (`org.springframework`) through a new property in `application.properties`. We can disable the previous property that we set for classes in `com.workshop.mvc` by commenting it out with a `#`

```
#logging.level.com.workshop.mvc=info
logging.level.org.springframework=debug
```

We will now see a large number of log output messages related to the Spring framework at both the `DEBUG` and `INFO` level in the Console view. We will still be able to view log messages from the application (in package `com.workshop.mvc`) at the `INFO`, `WARN` and `ERROR` levels: because `INFO` is the default level if no level is specified for any given package.

If you change the level for `logging.level.org.springframework` to either `warn` or `error` in `application.properties`, you will still only see `INFO` level messages being printed out since there was no problems in the execution of the Spring application.

Thus, you will typically only need to set this logging level for the Spring framework in the event you encounter an unexpected error while starting up or running your Spring application. The log messages at the `warn` or `error` level will then give you additional info that will help you in attempting to locate and debug the error.

Similarly, while you develop your application code, it is good practice to add various log statements at level of `DEBUG` or `TRACE` that output values of important variables in your code to help you trace the flow of program execution logic and its impact on the values of those variables. You can then stop

output from these statements when you run your application code in a production environment by setting the logging level to INFO (which is also already the default if you don't specify explicitly specify any logging level in application.properties).

At the same time, you can also add log statements at the level of WARN or ERROR at specific points in your code where something occurs that would critically interfere with normal operation of the app. Users running your application at the default log level of INFO would then be immediately alerted of these issues should they ever occur.

You can also set a logging level for every single class in every single package in the application: regardless of whether they are Spring framework related or packages developed by the user.

```
logging.level.root=debug
```

Using `root` as shown above will ensure that all log messages at the specified level severity (DEBUG) or higher (INFO, WARN, ERROR) within the application (in package `com.workshop.mvc`) or in the Spring framework (`org.springframework`) are output to the console.

Verify this for yourself

When done with this lab session, you can either comment out or remove the logging property settings in `application.properties`

Also remove all the logger statements demonstrating log output from `FirstSpringBootApplication`.

5 Creating a Spring Boot REST project via Spring Initializr

We will repeat creating a Spring Boot MVC application, but this time to expose some REST API endpoints for consumption.

The source code for this lab is found in `First-Spring-Rest/changes` folder.

Navigate to the Spring Initializr at:

<https://start.spring.io/>

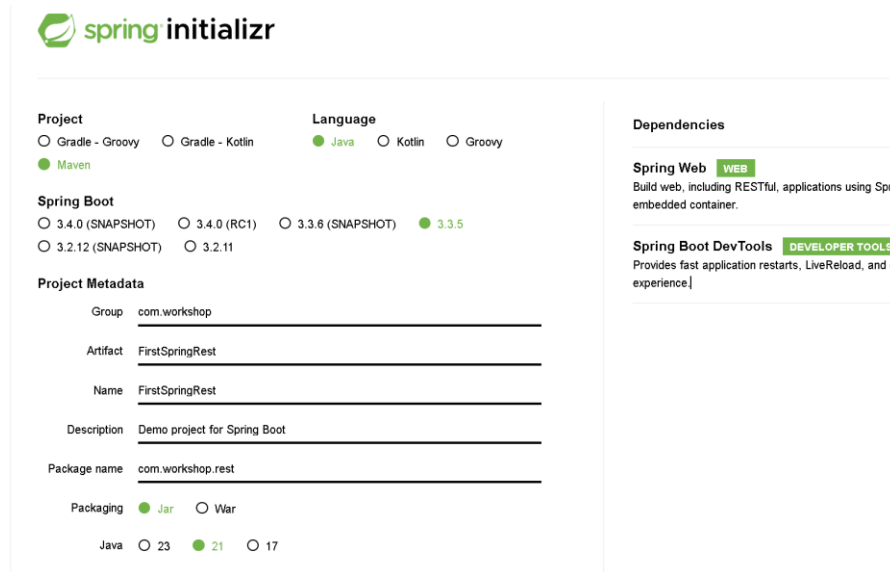
Key in the values for the fields as shown below.

Group: `com.workshop`
Artifact: `FirstSpringRest`
Name: `FirstSpringRest`
Description: `Demo project for Spring Boot`
Package: `com.workshop.rest`

Add in the following 2 dependencies as well

- Spring Web - this is a Spring Boot Starter template which includes all the dependencies required to build a complete web application as well as REST API web services.

- Spring Boot DevTools - this provides live reloads of the embedded Tomcat web server that is included in the Spring Web template. This facilitates application development and debugging by automatically restarting the Tomcat server every time source code changes are made to the web application and the compiled byte classes become available on the classpath



The image shows the Spring Initializr web form for creating a new project. It includes sections for Project, Language, Spring Boot, Project Metadata, and Dependencies.

Project

☐ Gradle - Groovy ☐ Gradle - Kotlin ☒ Maven

Language

☒ Java ☐ Kotlin ☐ Groovy

Spring Boot

☐ 3.4.0 (SNAPSHOT) ☐ 3.4.0 (RC1) ☐ 3.3.6 (SNAPSHOT) ☒ 3.3.5 ☐ 3.2.12 (SNAPSHOT) ☐ 3.2.11

Project Metadata

Group:

Artifact:

Name:

Description:

Package name:

Packaging: ☒ Jar ☐ War

Java: ☐ 23 ☒ 21 ☐ 17

Dependencies

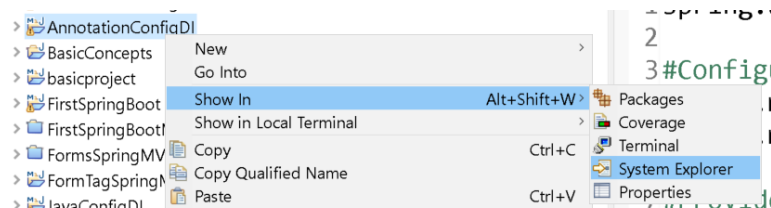
Spring Web **WEB**
Build web, including RESTful, applications using Spring embedded container.

Spring Boot DevTools **DEVELOPER TOOLS**
Provides fast application restarts, LiveReload, and cc experience.

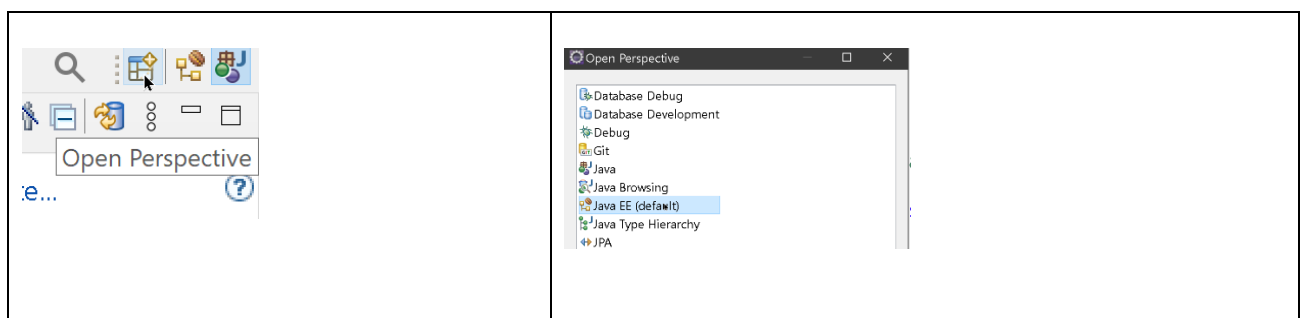
Click Generate.

Download the Zip file and extract its contents with 7-zip. Place the extracted folder in your Eclipse workspace directory.

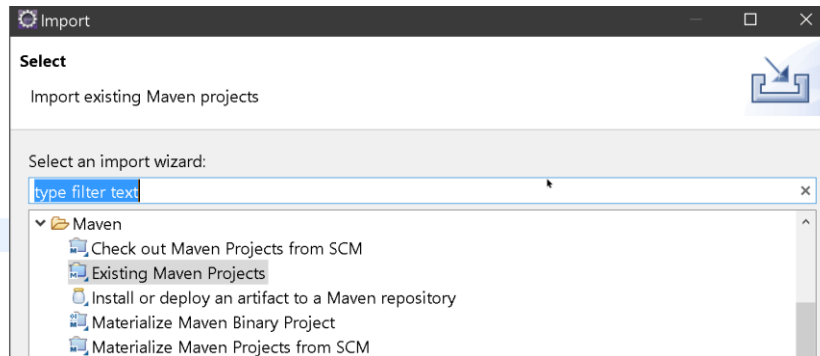
You can navigate to workspace directory from Windows Explorer. An alternative shortcut is to open the subfolder of any existing project in the workspace directory by right clicking on any existing project entry in the Project Explorer pane, then selecting Show in -> System Explorer



Switch to Java EE perspective if you are not already in there, either using the icons at the upper right hand corner or via the menu: Window -> Perspective -> Open Perspective



Go to File -> Import -> Maven -> Existing Maven Project.



Click Next, select Browse, and select the `FirstSpringRest` folder, which should also automatically highlight the `pom.xml` in that folder. Click Finish.

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

```
Greeting  
GreetingController
```

The `@RestController` annotation is a specialized version of the `@Controller` annotation. It is a combination of `@Controller` and `@ResponseBody`, which means that any method inside a class annotated with `@RestController` will return data (usually JSON, but can also be XML) instead of a view like a JSP file in Spring MVC. It is used specifically in RESTful web services to simplify the development of APIs that produce JSON or XML responses directly.

The `@GetMapping` handler methods in a `@RestController` class will return an object, whose fields will be serialized into JSON or XML content and sent back in the HTTP response body.

Start the web application in the usual way.

Go to `FirstSpringRestApplication`, right click and select Run As -> Java Application

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

<http://localhost:8080/greeting>

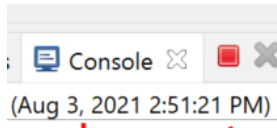
You should see JSON content returned in the form of:

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

Keep refreshing the browser (pressing F5) to issue repeated HTTP GET calls to this simple 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.



5.1 Creating an executable JAR

We can also produce an executable JAR file for this app.

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

For the Goals field, type: `clean package`

Then click Run.

Right click and Refresh the project.

Open a command prompt terminal in the `target` subfolder of this project (right click on the folder -> Show in Local Terminal -> Terminal), or use a separate command prompt of your own.

At the terminal type:

```
java -jar FirstSpringRest-0.0.1-SNAPSHOT.jar
```

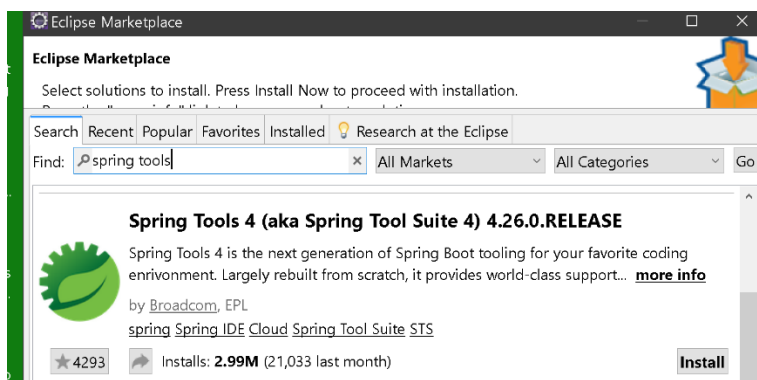
As before, test the app by opening a browser tab at:

<http://localhost:8080/greeting>

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

6 Creating Spring Boot REST projects via Spring Tool Suite (STS)

The Spring Tool Suite (STS) is a plugin for the Eclipse Enterprise IDE which provides additional features that supports development of Spring projects. You can install this plugin into your current Eclipse IDE by going to Help -> Eclipse Marketplace, searching for Spring Tools and identifying the STS entry and installing it.



Alternatively, you can download a totally new Eclipse IDE installation with STS fully installed from the [main STS website](#):

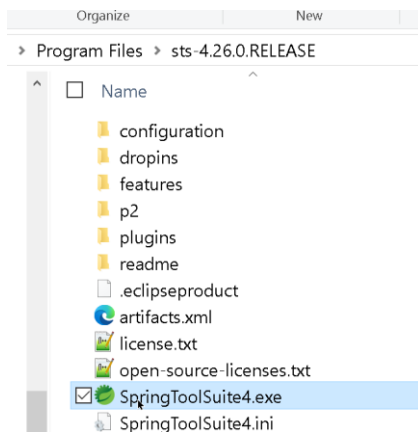
This is what we will do for this workshop. Scroll down to the STS4 for Eclipse and download the zip xxx-Windows X86-64.

Once you have downloaded the zip, extract it using the 7-zip tool that you installed earlier. Avoid using the normal built-in Windows zip facility as this may result in problems due to the large number of files in this directory.

Move the extracted folder to a suitable location on your drive (for e.g. C:\Program Files\sts-xxx-RELEASE).

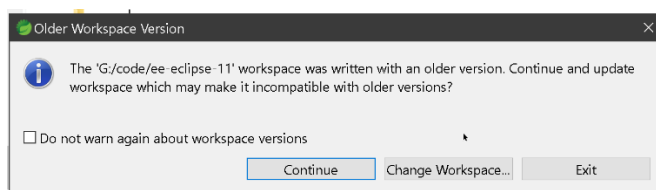
You can close your normal Eclipse Enterprise IDE that you have been using so far.

Then start the STS IDE by double clicking on the icon in the root installation folder.

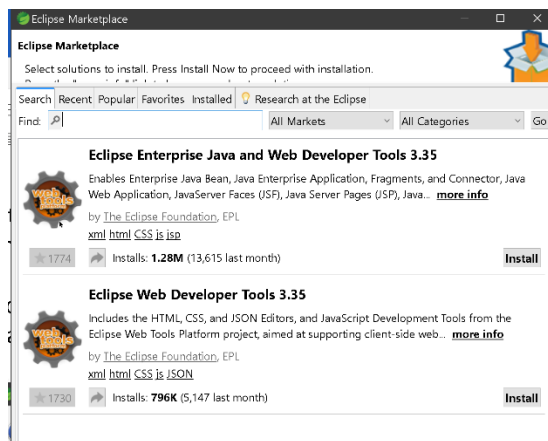


At the startup, STS IDE then prompts you to select a workspace, and you can specify the same folder that you had used with your normal Eclipse IDE, and click Launch.

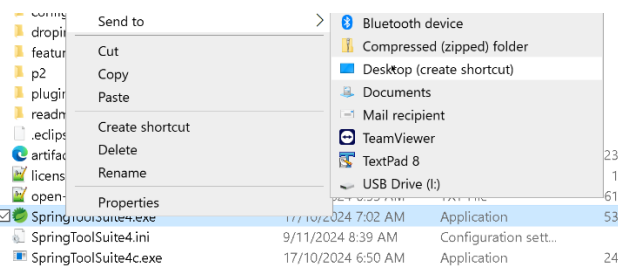
You may get a message about updating the workspace to use the latest version of Eclipse that STS is based on, and you can go ahead and click Continue.



After starting up, it may alert you to any missing extensions that you might need to install. In that case, click yes to transition to the Marketplace dialog place and then install whatever recommended plugins you see there.



Once you are done, you can close the IDE and send the executable to the desktop as a shortcut.

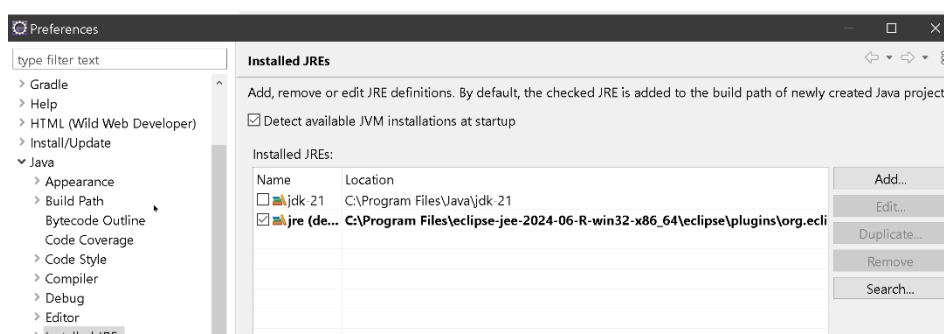


Clicking on the shortcut on the Desktop will then allow you to start STS straightaway.

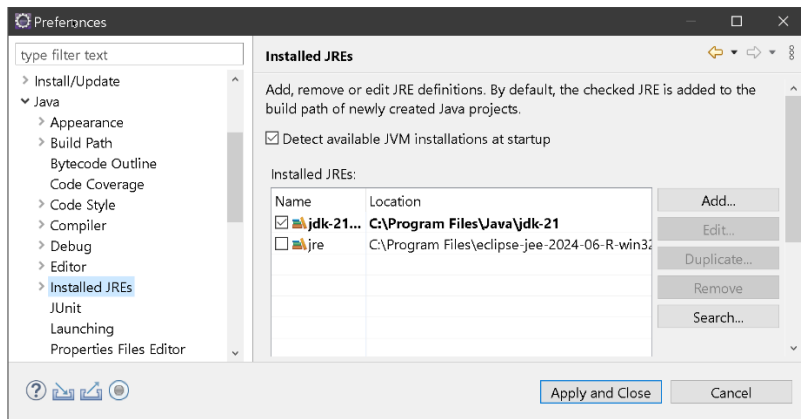
For a new STS installation and also when you change to a new workspace, you may need to double check to ensure the JDK is set properly to ensure that your programs compile properly and Maven (the build tool that we are using for this workshop) can execute correctly.

Newer versions of STS already include a built-in JRE which is typically 17 or higher. To avoid potential issues with Maven and other Java ecosystem tools, it is best to configure STS to use the JDK that you have installed locally rather than the built-in JRE.

To do this, go to Window -> Preferences, and in the Dialog Box, Java -> Installed JREs. Eclipse will detect available JVM installations by default at start up and your local JDK should be shown here.



If Eclipse already detects this local JDK, you can directly select it in the Installed JREs dialog box and click Apply and Close.

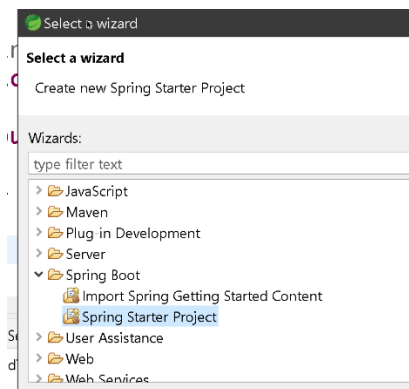


Go to Window -> Preferences -> Java -> Compiler.

In the Compiler Compliance Settings, set it to the version of the JDK installed on your machine (for e.g. 21). Click Apply and Close.

Switch to the Java EE perspective.

Go to File -> New -> Other. In the Wizard dialog box, scroll down to Spring Boot and select Spring Starter and click Next.



The dialog box that appears provides the same options that are presented at [Spring Initializr](https://spring.io/guides-topics/docs/spring-boot-spring-initializr/). Complete it with the following details:

Name: FirstSTSRest

Type: Maven

Java version: 21

Packaging: Jar

Group: com.workshop

Artifact: FirstSTSRest

Version: 0.0.1-SNAPSHOT

Description: Demo Project for Spring REST Boot app

Package: com.workshop.rest

Click Next and select the following Boot Starter dependencies.

Web -> Spring Web

Web -> Spring Boot DevTools

Click Finish.

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

In the package `com.workshop.rest`, 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:

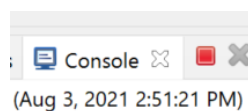
```
2021-08-03 17:05:55.965_[0;39m _[32m INFO_[0;39m _[35m11880_[0;39m _[2m--_[0;39m _[2m[
main]_[0;39m _[36mw.s.c.ServletWebServerApplicationContext_[0;39m _[2m:_[0;39m Root
WebApplicationContext: initialization completed in 493 ms
_2m2021-08-03 17:05:56.178_[0;39m _[32m INFO_[0;39m _[35m11880_[0;39m _[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.

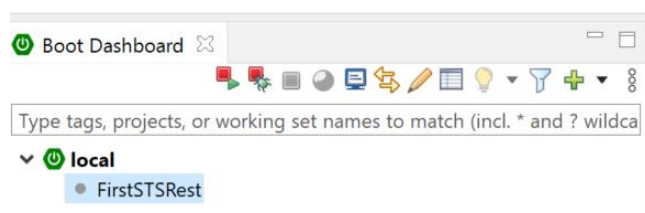
To stop the embedded Tomcat server, click on the red button next to the Console view.



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)



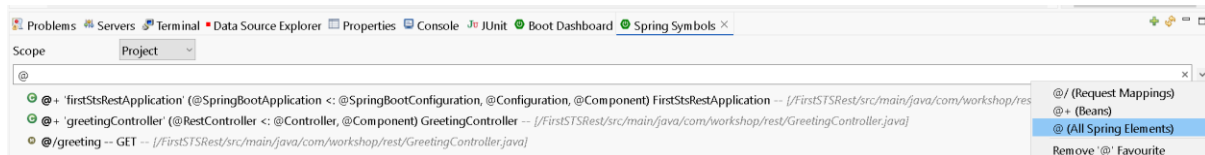
STS provides a Spring Symbols view which helps to identify Spring related annotations in your various classes.

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.

Select the All Spring Elements from the drop down menu.

You should be able to see all the Spring related annotations in the Project:



Double clicking on any of the entries in this view will navigate to that particular annotation in the specific source code file. This allows you to quickly determine which classes have specific notations, for e.g. @Component, @RestController, etc, etc

You can also click on any other existing Spring Boot projects that you have already created in the workspace folder, and you will also be able to see their Spring related annotations as well.

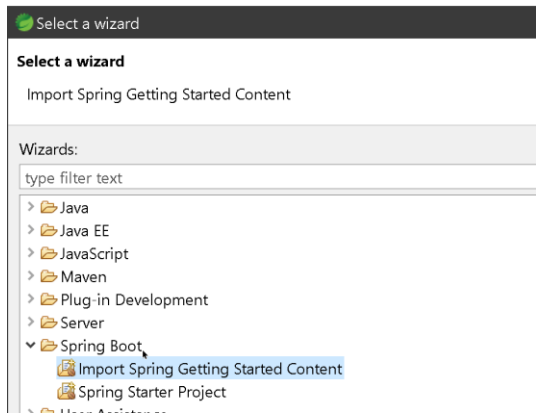
Similarly, you can go to the individual source code files (GreetingController, FirstStsRestApplication) and select File in the Spring Symbols view to see the annotations at the individual file level.



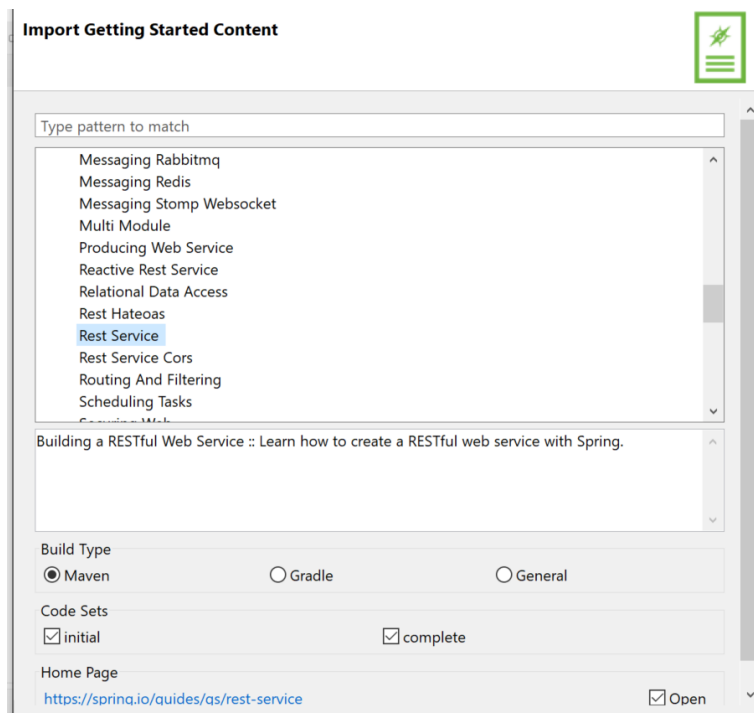
STS also provides the ability to directly import any of the projects from the [Getting Started guides](#) from the main Spring home page:

For e.g. this tutorial details how we can [build a simple REST service](#) in the Spring framework step-by-step:

To create a new Spring Boot project that illustrates this tutorial, go to File -> New Project -> Spring Boot -> Import Getting Started Content:



In the Dialog Box, select Getting Starting Guide -> Rest Service and check the other options as follows:



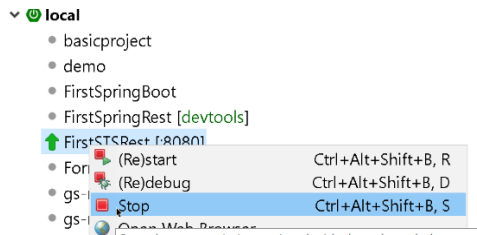
Click Finish.

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.

In the Boot dashboard select any running Spring Boot projects with an embedded Tomcat server (for e.g. FirstSTSRest), and right click to bring up the context menu to stop it.



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.

7 Monitoring REST HTTP traffic with TCP/IP monitor in Eclipse

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 a variety of TCP / IP monitoring software which can monitor this HTTP traffic flow in detail, we can use a simple tool that is already provided in Eclipse.

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

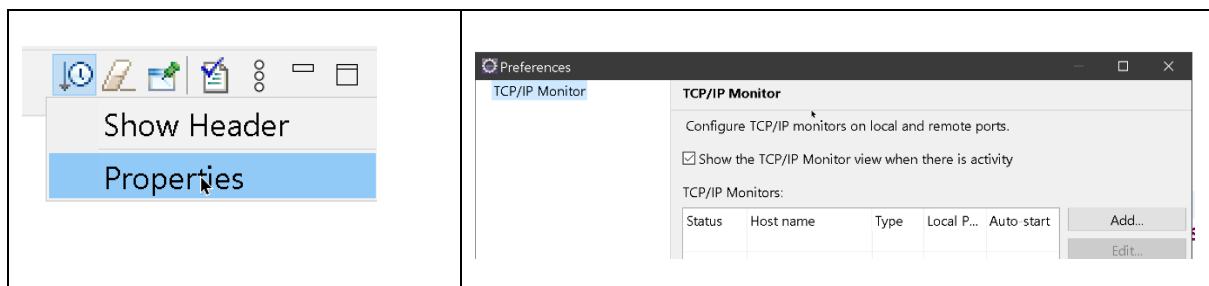
The monitor can be set up to proxy HTTP traffic from a particular source to a given destination, thereby allowing it to monitor the contents of the HTTP packets flowing through it.

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

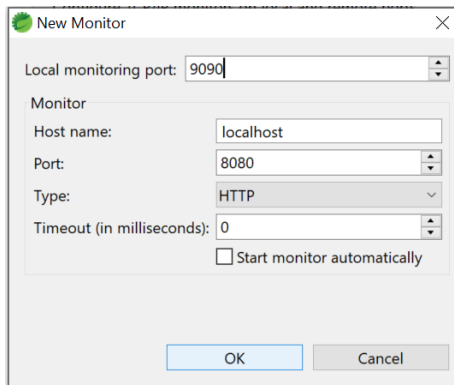
```
netstat -aon | findstr port-number
```

in Windows. A good example of a typical port that is free on most Windows machine is 9090.

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.

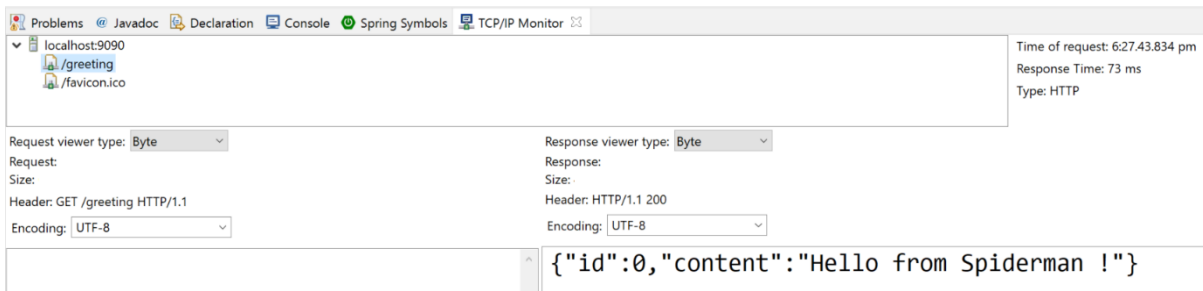
Right click on the project entry in the Project explorer pane and then select Run As -> Spring Boot App, or right click on the project entry in Spring Boot dashboard (Window -> Show View -> Other -> look in the Other folder -> Boot Dashboard) and select Start / Restart.

Using the browser (or any REST client such as Postman or Talend API tester):

<http://localhost:9090/greeting>

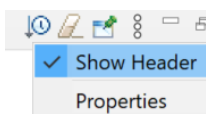
Remember to use the port number that the monitor is listening on (9090 for this example).

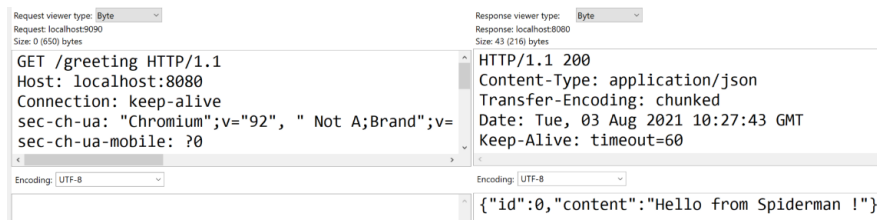
The HTTP request is relayed through the monitor to the embedded Tomcat server and the response is relayed back through the monitor as well. This allows the monitor to capture and display the contents of the request and response. You should see something similar to the following in the TCP/IP monitor 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. To do this, select Properties from the upper right hand corner, highlight the monitor and click Stop.



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