Accelebrate®

ACCELERATED LEARNING, CELEBRATED RESULTS®

RESTful API Testing

Lab Guide

https://www.accelebrate.com
(877) 849-1850 ❖ info@accelebrate.com

Customized Technical Training

On-Site, Customized Private Training

Don't settle for a one-size-fits-all class! Let Accelerate tailor a private class to your group's goals and experience. Classes can be delivered at your site or online (or a combination of both) worldwide. Visit us at **https://www.accelerate.com** and contact us at **sales@accelerate.com** for details.

Public Online Training

Need to train just 1-3 people? Attend one of our regularly scheduled, live, instructor-led public online classes. For course dates, times, outlines, pricing, and registration, visit **https://www.accelerate.com/public-training-schedule**.

Newsletter

Want to find out about our latest class offerings? Subscribe to our newsletter **https://www.accelerate.com/newsletter**.

Blog

Get insights from our instructors and staff! Visit our blog, **https://www.accelerate.com/blog** and join the discussion threads to get feedback from our instructors!

Learning Resources

Get access to **tutorials**, **how-to's**, and both past and upcoming **webinars** in the **Insights** section of our website.

Call us for a training quote!

877 849 1850

Accelerate was founded in 2002 with the goal of delivering **private training** that rapidly achieves participants' goals. Each year, our experienced instructors deliver hundreds of classes online and at client sites all over the US, Canada, and abroad. We pride ourselves on our **instructors' real-world experience** and ability to **adapt the training** to your team and their objectives. We offer a wide range of topics, including:

- AWS, Azure, and Cloud Computing
- Angular, React, and Vue
- JavaScript
- Data Science using R, Python, & Julia
- Excel Power Query
- Power BI & Tableau
- .NET & VBA programming
- SharePoint & Microsoft 365
- DevOps & CI/CD
- iOS & Android Development
- PostgreSQL, Oracle, and SQL Server
- Java, Spring, and Groovy
- Agile, DEI, & IT Leadership
- Web/Application Server Admin
- HTML5 & Mobile Web Development
- Adobe & Articulate
- Docker, Kubernetes, Ansible, & Git
- Software Design and Testing
- AND MORE *(see back)*

"Clear communication, easy instructions, hands-on interactions...we couldn't ask for a better learning experience or instructor. Accelerate exceeded all my expectations."

— Nicole, John Hopkins Healthcare

Like us on Facebook + Follow us on Twitter + Watch us on YouTube

Visit our website for a complete list of courses!

Adobe & Articulate
Adobe Captivate
Adobe Presenter
Articulate Storyline / Studio
Camtasia
RoboHelp

AWS, Azure, & Cloud
Amazon Web Services (AWS)
Azure
Cloudflare
Google Cloud Platform (GCP)
OpenStack
Terraform
VMware

Big Data
Alteryx
Apache Spark
Teradata
Snowflake SQL

Data Science and RPA
Alteryx
Apache Airflow
Automation Anywhere
Azure Data & AI
Blue Prism
Data Literacy
Data Science
DataOps
Django
Flask
Julia
Jupyter
Machine Learning
MATLAB
Microsoft Power Automate
Minitab
Python / R Programming
RapidMiner
Splunk
SPSS
UiPath

Data Visualization
BusinessObjects
Crystal Reports
Excel Power Query
Microsoft Power Platform
Power BI
PivotTable and PowerPivot
Qlik
SSRS
Tableau

Database
Amazon RedShift
MongoDB
NoSQL
Oracle / Oracle APEX
PostgreSQL
SQL Server
SQL Using MySQL

DevOps, CI/CD & Agile
Agile / DEI / Six Sigma
IT Leadership / ITIL
Ansible
Apache Maven
Business Analysis
DevOps / DevSecOps
Docker & Kubernetes
Git
Gradle
Jenkins / Jira & Confluence
Kafka / Linux
Microservices
OpenShift
Software Design

Java
Bazel / BDD / Drools
Groovy and Grails
Java & Web App Security
JavaFX / JBoss / RxJava
Scala
Selenium & Cucumber
Spring Boot / Framework

JS, HTML5, & Mobile
Angular
CSS
Cypress
D3.js
Flutter
HTML5
iOS/Swift Development
JavaScript
Kotlin
Node.js & Express
React & Redux
Svelte / Swift
Symfony
Vue
Web Assembly

Microsoft & .NET
.NET Core / ASP.NET
Azure DevOps / Blazor / C#
Entity Framework Core
IIS
Microsoft Dynamics
Microsoft 365
Microsoft Power Platform
Microsoft Project
Microsoft SQL Server
Microsoft Windows Server
PowerPivot / PowerShell
VBA / Visual Studio
Web API

Security
.NET Web App Security
C and C++ Secure Coding
C# & Web App Security
CISSP Test Prep
DevSecOps
GCP Security
Microsoft Security
Linux Security Admin
Python Security
Secure Coding for Web Dev
Spring Security

SharePoint
Power Automate & Flow
SharePoint Administrator
SharePoint Developer
SharePoint End User
SharePoint Online
SharePoint Site Owner

SQL Server
Azure SQL Data Warehouse
Business Intelligence
Performance Tuning
SQL Server Administration
SQL Server Development
SSAS / SSIS / SSRS
Transact-SQL

Teleconferencing Tools
Adobe Connect
GoToMeeting
Microsoft Teams
WebEx and Zoom

Web/Application Server
Apache httpd and Tomcat
IIS / JBoss
Oracle WebLogic

Other
ArgoCD
C++
Erlang / Figma
Go Programming
Google Analytics and GA4
Lua
Mulesoft
RESTful API
Rust
Salesforce
Sitefinity
TestComplete
UX
XML
Writing and Communication

Visit ***www.accelebrate.com/newsletter*** *to sign up and receive our* **newsletters** *with information about* **new courses**, **free webinars**, **tutorials**, *and* **blog articles**.

Call us for a training quote! 877 849 1850 (US/Canada) or +1 678 648 3133

WA2678 Designing REST Services for Architects

Student Labs

Web Age Solutions

Table of Contents

Lab 1 - A Simple RESTful API in Spring Boot....4
Lab 2 - JSON....14
Lab 3 - Implementing OAuth2 using Spring Boot....23
Lab 4 - OpenAPI (Swagger)....33
Lab 5 - Manually Test an API....41

ENVIRONMENT

For this course you will have the following 2 environments running for you:

- **WA2678-REL_2_1**
- **VM_WA2785**

The **WA2678-REL_2_1** is on Windows and could be installed directly to the computer provided to you or in a VM (virtual Machine) and will be used in **Labs 1, 2, 3 and 5**

VM_WA2785 will be used in **Lab 4 only.**

Make sure to use the right setup for every lab.

Lab 1 - A Simple RESTful API in Spring Boot

In this lab we're going to build a simple "Hello World" API using Spring Framework and Spring Boot. The API will implement a single resource, "/hello-message" that returns a JSON object that contains a greeting.

Part 1 - Connect to WA2678-REL_2_1

In this Lab you will be working in the machine called: **WA2678-REL_2_1** **Start or connect to this machine if you don't have it opened yet**. Note. This is running on Windows.

__1. Open a command prompt window.

__2. Verify the java version installed:

```
java -version
```

Make sure you see the response shown below:

```
C:\Users\wasadmin>java -version
java version "11.0.8" 2020-07-14 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.8+10-LTS)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.8+10-LTS, mixed mode)
```

Part 2 - Create a Maven Project

We're going to start from scratch on this project, with an empty Apache Maven project, and add in the dependencies that will make a Spring Boot project with a core set of capabilities that we can use to implement our "Hello World" API.

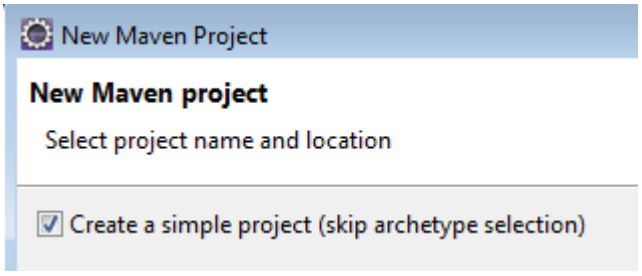
__1. Open Eclipse. There should be a shortcut in the desktop or you can find it from the start menu.

__2. In the **Workspace Launcher** dialog, enter **C:\Workspace** in the **Workspace** field, and then click **Launch**.

__3. Close the **Welcome** panel if it appears.

__4. From the main menu, select **File** → **New** → **Maven Project**.

__5. In the **New Maven Project** dialog, click on the checkbox to select "Create a simple project (skip archetype selection)", and then click **Next**.



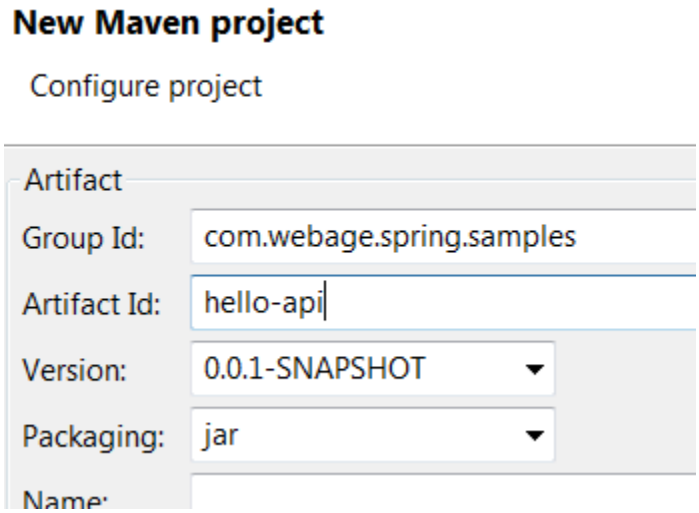
__6. Enter the following fields:

Group Id: com.webage.spring.samples

ArtifactId: hello-api

Leave all the other fields at their default values.

__7. When the dialog looks like below, click **Finish**.

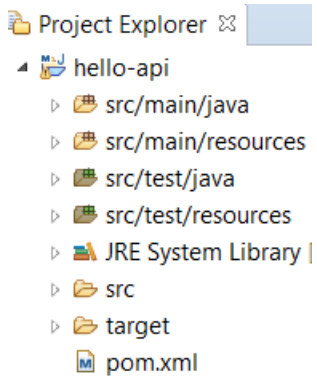


__8. If necessary, wait for Eclipse to finish any background process.

Part 3 - Configure the Project as a Spring Boot Project

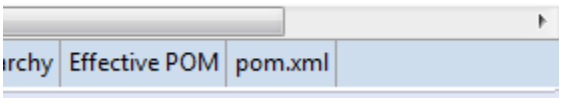
The steps so far have created a basic Maven project. Now we'll add the dependencies to make a Spring Boot project.

__1. Expand the **hello-api** project in the **Project Explorer**.



__2. Double-click on **pom.xml** to open it.

__3. At the bottom of the editor panel, click the **pom.xml** tab to view the XML source for **pom.xml**.



__4. Insert the following text after the "<version>...</version>" element, and before the closing "</project>" tag:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.2.6.RELEASE</version>
</parent>
```

__5. Save changes.

Note that you may see an error on **pom.xml** (pom.xml). Ignore it for now.

__6. Insert the following text after the parent tag:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

The entries above call out the Spring Boot Starter Parent project as the parent to this project, then call out the Spring Boot Starter Web dependencies. Finally the <build> element configures the Spring Boot Maven Plugin, which will build an executable jar file for the project.

__7. Save and close the file.

Note that you may see an error on the project. Ignore it for now.

__8. Right-click on the **hello-api** project and then select **Maven → Update Project**, and then click **OK** in the resulting dialog.

Wait until finish the downloading, installing and building the project.

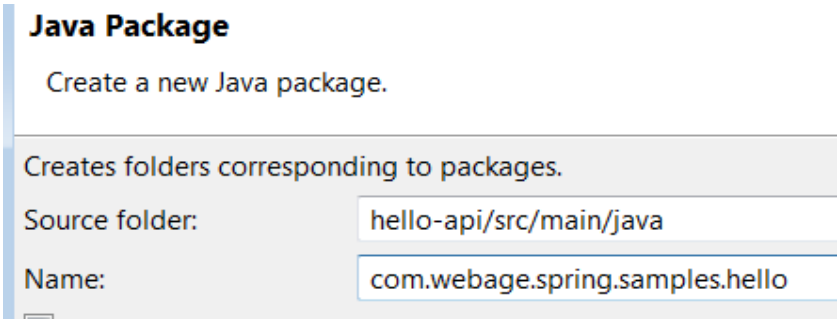
There should be no errors in the project.

Part 4 - Create an Application Class

Spring Boot uses a 'Main' class to startup the application and hold the configuration for the application. In this section, we'll create the main class.

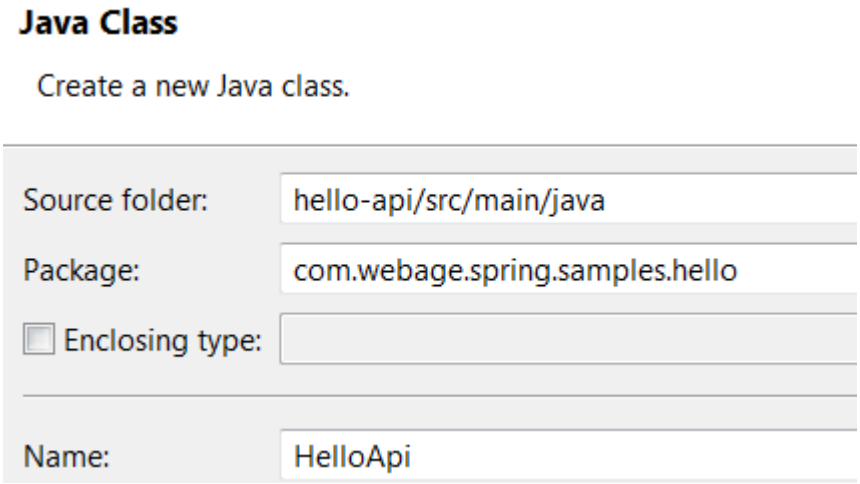
__1. In the **Project Explorer**, right-click on **src/main/java** and then select **New → Package**.

__2. Enter **com.webage.spring.samples.hello** in the **Name** field, and then click **Finish**.



__3. In the **Project Explorer**, right-click on the newly-created package and then select **New → Class**.

__4. In the **New Java Class** dialog, enter **HelloApi** as the **Name**, and then click **Finish**.



__5. Add the **@SpringBootApplication** annotation to the class, so it appears like:

```
@SpringBootApplication
public class HelloApi {
```

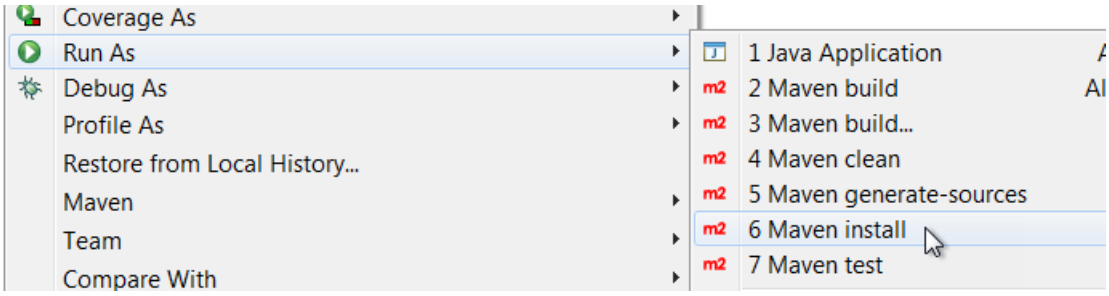
__6. Add the following 'main' method inside the class:

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

__7. The editor is probably showing errors due to missing 'import' statements. Press **Ctrl-Shift-O** to organize the imports.

__8. Save the file.

__9. In the **Project Explorer**, right-click on either the **hello-api** project node or the 'pom.xml' file and then select **Run As** → **Maven Install**.



Wait until finish the downloading, installing and building the project.

Note. If fails building try again and the second time should works.

The console should show a successful build. This ensures that we don't have any typos in the pom.xml entries we just did.

```
[INFO] Installing C:\Workspace\hello-api\pom.xml t
[INFO] ------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------
[INFO] Total time:  25.044 s
```

Now all we need to do is add a resource class and a response class.

Part 5 - Implement the RESTful Service

In this part of the lab, we will create a response class and a RESTful resource class.

__1. In the **Project Explorer**, right-click on **src/main/java** and then select **New** → **Package**.

__2. Enter **com.webage.spring.samples.hello.api** in the **Name** field, and then click **Finish**.

__3. In the **Project Explorer**, right-click on the newly-created package and then select **New** → **Class**.

__4. In the **New Java Class** dialog, enter **HelloResponse** as the **Name**, and then click **Finish**.

__5. Edit the body of the class so it reads as follows:

```
package com.webage.spring.samples.hello.api;

public class HelloResponse {
          String message;

          public HelloResponse(String message) {
                    super();
                    this.message = message;
          }

          public String getMessage() {
                    return message;
          }

          public void setMessage(String message) {
                    this.message = message;
          }
}
```

__6. Save the file.

__7. In the **Project Explorer**, right-click on the **com.webage.spring.samples.hello.api** package and then select **New → Class**.

__8. In the **New Java Class** dialog, enter **HelloResource** as the **Name**, and then click **Finish**.

__9. Add the following 'getMessage' method inside the new class:

```
public HelloResponse getMessage() {
          return new HelloResponse("Hello!");
}
```

Spring Boot recognizes and configures the RESTful resource components by the annotations that we're about to place on the resource class that we just created.

__10. Add the '@RestController' annotation to HelloResource, so it looks like:

```
@RestController
public class HelloResource {
```

__11. Add the '@GetMapping' annotation to the 'getMessage' method, so it looks like:

```
@GetMapping("/hello-message")
public HelloResponse getMessage() {
```

__12. Organize the imports by pressing **Ctrl-Shift-O**.

__13. Save all files by pressing **Ctrl-Shift-S**.

__14. In the **Project Explorer**, right-click on either the **hello-api** project node or the 'pom.xml' file and then select **Run As → Maven Install**.

Note. If fails building try again and the second time should works.

The console should show a successful build.

Part 6 - Run and Test

That's all the components required to create a simple RESTful API with Spring Boot. Now let's fire it up and test it!

__1. In the **Project Explorer**, right-click on the **HelloApi** class and then select **Run as → Java Application**.

__2. If the **Windows Security Alert** window pops up, click on **Allow Access**.

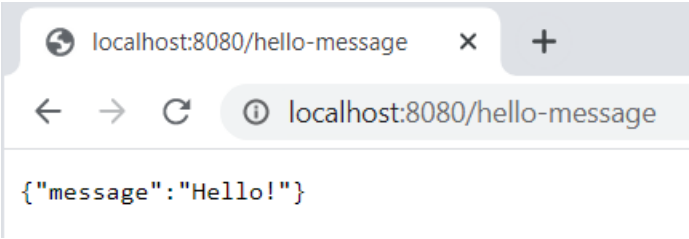
__3. Watch the **Console** panel. At the bottom of it, you should see a message indicating that the **HelloApi** program has started successfully:

```
o.a.c.c.C.[Tomcat].[localhost].[/]      : Initializing Spring embedded WebApplicationContext
o.s.web.context.ContextLoader           : Root WebApplicationContext: initialization completed in 2522 ms
o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
c.webage.spring.samples.hello.HelloApi  : Started HelloApi in 4.087 seconds (JVM running for 4.947)
```

__4. Open the **Chrome** browser and enter the following URL in the location bar:

```
http://localhost:8080/hello-message
```

__5. You should see the following response:



Notice that the response is in the form of a JSON object whose structure matches the 'HelloResponse' class contents.

__6. Close the browser.

__7. Click on the red 'Stop' button on the **Console** panel to stop the application.

__8. Click **Remove All Terminated Launches**.

__9. Close all open files.

Part 7 - Review

In this lab, we setup a rudimentary Spring Boot application. There are a few things you should notice:

- There was really very little code and configuration required to implement the very simple RESTful API.
- The resulting application runs in a standalone configuration without requiring a web or application server. It opens its own port on 8080.
- Although the Eclipse IDE is providing some nice features, like type-ahead support and automatic imports, the only tool we really need is a build tool that does dependency management (e.g. Apache Maven).

Lab 2 - JSON

In this lab, you will perform syntax and semantic data validation against a JSON schema using CLI tool and Node.js module.

In this Lab you will be continue working in the machine called WA2678-REL_2_1

Part 1 - Create Directory Structure

In this part you will create a directory structure to store JSON data, JSON Schema, and an application.

__1. Open **Command Prompt** from the **Start Menu**.

__2. Switch to the **Workspace** directory:

```
cd c:\workspace
```

Note: If the directory doesn't exist, create it by using md c:\workspace and then switch to it by executing the above command.

__3. Create a directory named "json" by using following command:

```
md json
```

__4. Switch to the newly created directory:

```
cd json
```

Part 2 - Create a JSON document

In this part you will create a simple JSON document. You will create a JSON Schema, later in the lab to validate it.

__1. Use Notepad / Notepad++ to create a blank file named **data.json** under **json** directory.

__2. In the **data.json** file, enter following data:

```
[
        {
                "productId":1,
                "productTitle":"iPhone",
                "price":699.95
        },
        {
                "productId":2,
                "productTitle":"iPad",
                "price":599.95
        }
]
```

Note: There is an array with two products. Each product has productId, productTitle, and price properties.

__3. Save the file and close Notepad.

__4. Syntactically validate the JSON file:

```
jsonlint data.json
```

Note: If you see the JSON data then the syntax is valid. If there's any syntax error, you would see the error description.

Part 3 - Create a JSON Schema

In this part you will create a JSON Schema to validate the JSON data you created previously.

__1. Using Notepad / Notepad++, create a file named **schema.json** under **json** directory.

__2. Add following code to the schema.json file:

```
{
  "$schema": "http://json-schema.org/draft-03/schema#",
}
```

Note: This is the minimal possible code you are required to enter in a JSON schema. Currently, it won't be able to semantically validate the JSON data since you are yet to define the rules.

__3. Below the $schema statement, add following: (Note: To make it easier to read, indent the code. You can easily indent the code by using online tools, such as https://codebeautify.org/jsonviewer)

```
"type": "array",
"minItems": 1,
"items": {
}
```

Note: These lines will validate the JSON data contains an array with at least 1 item in it.

__4. Within the **items** block, add following lines:

```
"type": "object",
"properties": {
}
```

Note: The **type** attribute specifies each item of the array is an object. You will specify the object properties in the next step.

__5. Within the **properties** block, add the following lines:

```
"productId": {
 "type": "integer",
 "minimum": 1
},
```

Note: These lines ensure each item of the array should have a productId property of type integer (whole number) and it's minimum value should be 1.

__6. Below the above lines, within the properties block, add following lines:

```
"productTitle": {
  "type": "string",
  "minLength": 3,
  "maxLength": 6,
  "enum": [ "iPhone", "iPad" ]
 },
```

Note: These lines ensure each item of the array should have a productTitle property of type string, it's minimum length should be 3, and it's possible values should be iPhone or iPad.

__7. Below the above lines, within the properties block, add following lines:

```
"price": {
  "type": "number",
  "minimum": 100
 }
```

Note: These lines ensure each item of the array should have a price property of type number (floating / whole) and it's minimum value should be 100.

__8. Save the file and close the file.

__9. Syntactically validate the schema.json file:

```
jsonlint schema.json
```

Note: If you see the JSON data then the syntax is valid. If there's any syntax error, you would see the error description.

Part 4 - Test the Schema

In this part you will test the data against the schema and experiment with various validation rules.

__1. In **Command Prompt** window, under **json** directory, run following command:

```
jsonlint data.json --validate schema.json
```

Notice if there is no semantic validation issue, the JSON data gets displayed as is.

__2. Open **schema.json** file in Notepad / Notepad++, change minItems from 1 to 3.

__3. Save the file.

__4. Semantically validate the data.json file again:

```
jsonlint data.json --validate schema.json
```

Notice it displays error message like this:

```
c:\Workspace\json>jsonlint data.json --validate schema.json
Validation Errors:

The number of items is less than the required minimum
uri: urn:uuid:da9e085d-5f8b-431d-aaef-a9c1b6c4d669#
schemaUri: urn:uuid:92d9ca11-fbad-4c92-b05a-e773ed2c9532#
attribute: minItems
details: 3
```

__5. Open data.json file in Notepad / Notepad++ and add another product like this:

```
,
{
  "productId":3,
  "productTitle":"Apple TV",
  "price":599.95
}
```

__6. Save the file.

__7. Semantically validate the data.json file again:

```
jsonlint data.json --validate schema.json
```

Notice there's a different error this time, complaining that enumeration value is invalid.

```
c:\Workspace\json>jsonlint data.jso→--validate schema.json
Validation Errors:

String is greater than the required maximum length
uri: urn:uuid:a4c6b7fa-b27c-40f3-b74c-3c6b5871daca#/2/productTitle
schemaUri: urn:uuid:7f58c193-da43-424d-9b5b-a2188642cc4c#/items/properties/productTitle
attribute: maxLength
details: 6

Instance is not one of the possible values
uri: urn:uuid:a4c6b7fa-b27c-40f3-b74c-3c6b5871daca#/2/productTitle
schemaUri: urn:uuid:7f58c193-da43-424d-9b5b-a2188642cc4c#/items/properties/productTitle
attribute: enum
details: ["iPhone","iPad"]
```

__8. In schema.json file, add Apple TV to the enum array so it looks like this:

```
"enum": [ "iPhone", "iPad", "Apple TV" ]
```

__9. Save the file.

__10. Semantically validate the data.json file again:

```
jsonlint data.json --validate schema.json
```

Notice there's a different error this time, complaining that productTitle max length is invalid.

```
C:\Workspace\json>jsonlint data.json --validate schema.json
Validation Errors:

String is greater than the required maximum length
uri: urn:uuid:24d269c7-da6c-4f31-b8f3-56e374d8fc66#/2/productTitle
schemaUri: urn:uuid:3f32c415-0109-4048-be58-dab640dcf7f3#/items/properties/produ
ctTitle
attribute: maxLength
details: 6
```

__11. In **schema.json** file, change maxLength to 10

__12. Save the file.

__13. Semantically validate the data.json file again:

```
jsonlint data.json --validate schema.json
```

Notice if there are no errors, the JSON data gets displayed as is.

Part 5 - Semantic Validation using Node.js

In this part you will validate data against a JSON Schema in a node.js application.

__1. Make sure you are in the json directory:

```
cd c:\workspace\json
```

__2. Initialize a new node application:

```
npm init
```

__3. Press the Enter key to use default value for each option.

__4. Install **jsonschema** node module.

```
npm install jsonschema --save-dev
```

__5. Using Notepad / Notepad++, create **index.js** file and save it under **c:\workspace\json** directory.

__6. Enter following code in the new file:

```
var validate = require('jsonschema').validate;
```

Note: this line imports the jsonschema module. It will be used for performing semantic validation against the JSON schema.

__7. Add following lines below the above statement:

```
var data = require('./data.json');
var schema = require('./schema.json');
```

Note: These lines import data.json and schema.json files you created in the previous parts of this lab. In real-world, you would want to read the data which is either stored in a file, or receive it from a REST service.

__8. Add following lines of code:

```
var result = validate(data, schema);

console.log(result.errors);
```

__9. These lines semantically validate the data and display error messages, if there any.

__10. Save the file.

__11. Run the application:

```
node index.js
```

If there are no errors, you would see an empty array displayed like this: []. Otherwise, if there are any errors you would see the error messages.

__12. Open **schema.json** file and change **minItems** to 10.

__13. Save the file.

__14. Run the node application again:

```
node index.js
```

Notice it displays error message like this:

```
[
  ValidationError {
    path: [],
    property: 'instance',
    message: 'does not meet minimum length of 10',
    schema: {
      '$schema': 'http://json-schema.org/draft-03/schema#',
      type: 'array',
      minItems: 10,
      items: [Object]
    },
    instance: [ [Object], [Object], [Object] ],
    name: 'minItems',
    argument: 10,
    stack: 'instance does not meet minimum length of 10'
  }
]
```

__15. Close all.

Part 6 - Review

In this lab, you performed syntax and semantic data validation against a JSON schema using a CLI tool and a Node.js module.

Lab 3 - Implementing OAuth2 using Spring Boot

In this lab, you will use Spring Boot's built-in OAuth2 functionality. Rather than implementing user credentials in a custom database, you will use GitHub for authentication/authorization.

The app you will work on, in OAuth2 terms, is a Client Application and it will use the authorization code grant to obtain an access token from GitHub (the authorization server). It will then use the access token to ask GitHub for some personal details (only what you permitted it to do), including your login ID and your name. GitHub will act as a Resource Server, decoding the token that you send and if the process is successful the app inserts the user details into the Spring Security context so that you are authenticated.

You can also use other OAuth2 servers, such as Azure, Facebook, Google, Twitter, and Apple.

In this Lab you will be continue working in the machine called WA2678-REL_2_1

Part 1 - Getting Started

Note: During this lab, you will modify several files. You can use any text editor of your choice, such as Notepad++ and Visual Studio Code.

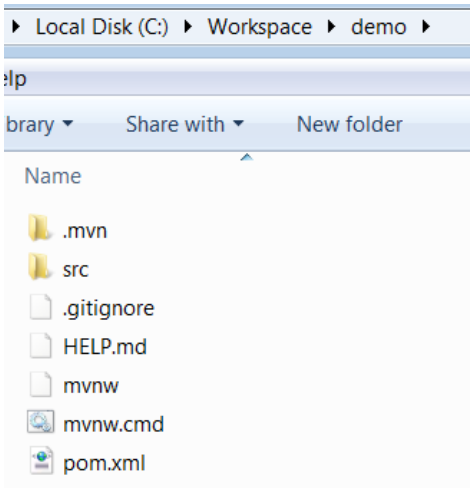
In this part, you will extract the starter project provided as part of this lab. You will modify the starter project and implement OAuth2 later in the lab.

__1. Make sure you are connected to the WA2678 VM.

__2. Using File Explorer, or any zip file archiving tool, extract **C:\LabFiles\oauth2-starter.zip** to **C:\Workspace\demo**.

Note: Make sure there is no nested folder named demo and it shows the following folder structure.

__3. Verify your folder looks like below:



Part 2 - Add a home page to the application and test it

In this part, you will add a home page to the application and test it.

__1. Open the file **c:\LabFiles\oauth2\index-1.txt** and copy the text to the clipboard.

__2. Open Visual Studio Code.

__3. From the menu, click **File → Open Folder**

__4. Expand **C:\Workspace\demo** and click **Select Folder**

__5. Click **Yes, I trust ..**

Note that we are using Visual Studio Code that a friendly interface to work on your project but you can use Notepad or other software to edit the files.

__6. Create a file named **index.html** under **c:\Workspace\demo\src\main\resources\static**

__7. Paste the text from the clipboard.

Note: The code displays the message Demo on the web page. It doesn't make any server-side calls. It adds references to bootstrap and jquery, but they aren't currently available under the specified path. You will download the resources later in this part of the lab.

__8. Save the file.

__9. Open a **Command Prompt** window.

__10. Switch to the **demo** directory:

```
cd c:\Workspace\demo
```

__11. Launch the application, by running the following command in the Command Prompt window:

```
mvn spring-boot:run
```

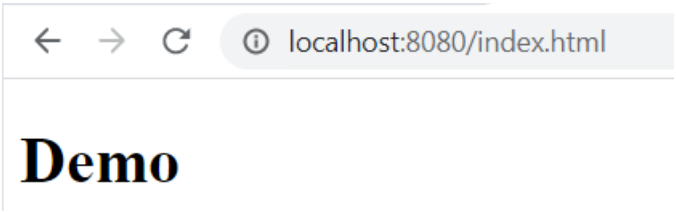
__12. Wait for the build to complete and the server and the application have been started:

```
2021-04-27 09:58:44.078  INFO 3664 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  :
Tomcat started on port(s): 8080 (http) with context path ''
2021-04-27 09:58:44.092  INFO 3664 --- [           main] com.example.demo.DemoApplication         :
Started DemoApplication in 2.069 seconds (JVM running for 2.459)
```

__13. Enter the following URL in the web browser:

```
http://localhost:8080/index.html
```

It will show 'Demo'.



__14. In the Command Prompt, press Ctrl+C, then press Y to stop the server.

__15. Open **c:\LabFiles\oauth2\pom-1.txt** and copy the text to the clipboard.

__16. Open **c:\Workspace\demo\pom.xml**

__17. To the dependencies section, paste the clipboard's contents, after the existing ones.

__18. Save the file.

__19. Launch the application, by running the following command in the Command Prompt window:

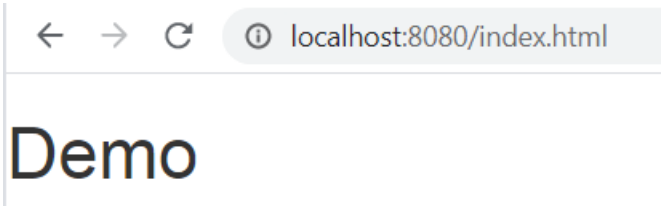
```
mvn spring-boot:run
```

Wait for the build to complete and the server and the application have been started.

__20. Enter the following URL in the web browser:

```
http://localhost:8080/index.html
```

Notice the bootstrap stylesheet has been applied and the message style is different than before.



__21. In the Command Prompt, press Ctrl+C, then press Y to stop the server.

Part 3 - Securing the Application

In this part, you will secure the application by adding Spring Security as a dependency. You will also add OAuth2 dependency and use GitHub as the authentication server.

__1. Open **c:\LabFiles\oauth2\pom-2.txt** and copy the text to the clipboard.

__2. Open **c:\Workspace\demo\pom.xml**

__3. To the dependencies section, paste the clipboard's contents, after the existing ones.

__4. Save the file.

__5. Open **c:\LabFiles\oauth2\application.txt** and copy the contents to the clipboard.

__6. In VS Code, open **c:\Workspace\demo\src\main\resources\application.properties** and then paste the clipboard's contents into it.

Note: Create the application.properties file if it doesn't exist in the specified folder. The configuration refers to a client app registered with GitHub in their developer's site, in which you have to supply a registered redirect (home page) for the app. This one is registered to "localhost:8080" so it only works in an app running on that address.

__7. Save the file.

Part 4 - Add a Welcome Page

In this part, you will modify the app and add an explicit link to log in with GitHub, instead of being redirected immediately. You will also utilize AngularJS on the client-side.

__1. Open **C:\Workspace\demo\src\main\resources\static\index.html** in an editor.

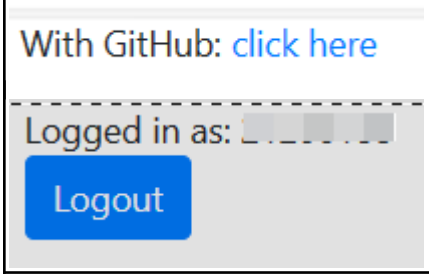
__2. Copy the contents of **c:\LabFiles\oauth2\index-2.txt** to the clipboard.

__3. Replace the contents of index.html's <body> tag with the contents in the clipboard.

__4. Copy the contents of **c:\LabFiles\oauth2\index-3.txt** to the clipboard.

__5. Paste the clipboard's contents immediately before the </body> (end of body tag).

The code snippet adds JavaScript code to show and hide a hyperlink and button on the page so the user can sign in or sign out.



__6. Save the file but do not run the project, yet.

__7. Open **c:\LabFiles\oauth2\imports-1.txt** and copy the text to the clipboard.

__8. Open **C:\Workspace\demo\src\main\java\com\example\demo\DemoApplication.java**

__9. Paste the clipboard's contents replacing the existing imports only.

Note: You are importing various classes/packages which will be utilized throughout the lab.

__10. Before the DemoApplication class, add the following annotation:

```
@RestController
```

Your DemoApplication class should look like this: (Note: Changes are highlighted below.)

@SpringBootApplication

```
@RestController
public class DemoApplication {
...
```

__11. Modify the DemoApplication class definition so it extends WebSecurityConfigurerAdapter. It should look like this:

```
public class DemoApplication extends WebSecurityConfigurerAdapter {
```

Note: If you see any errors or red wiggly lines in Visual Studio Code, ignore them.

__12. Before the main function, add the following code:

```
@GetMapping("/user")
public Principal user(Principal principal) {
  return principal;
}
```

__13. Add the following lines before the main function: (Note: The snippet is available in snippet-1.txt)

```
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http
    .authorizeRequests(a -> a
        .antMatchers("/", "/error", "/webjars/**").permitAll()
        .anyRequest().authenticated()
    )
    .exceptionHandling(e -> e
        .authenticationEntryPoint(new
HttpStatusEntryPoint(HttpStatus.UNAUTHORIZED))
    )
    .oauth2Login();
  }
```

Note: Spring Boot attaches special meaning to a class extending WebSecurityConfigurer. It uses it to configure the security filter chain that carries the OAuth2 authentication processor. So all you need to do to make your home page visible is to explicitly autheorizeRequests() to the home page and the static resources it contains.

__14. Save the file and make sure all files were saved.

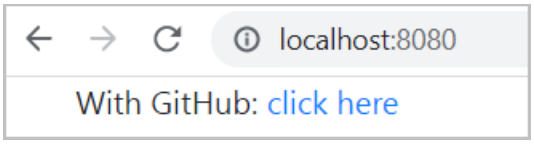
__15. Launch the application, by running the following command in the Command Prompt window:

```
mvn spring-boot:run
```

__16. Wait for the build to complete, then enter the following URL in the web browser:

```
http://localhost:8080
```

It will ask you to sign in with GitHub credentials then show the message.



If you are already signed in with GitHub credentials, it will show you a Login in as ### message on the page.

__17. In the Command Prompt window, press Ctrl+C to stop the server.

Part 5 - Add the Logout Button

In this part, you will modify the client-side and server-side code to add the Logout functionality.

__1. In the index.html file, modify the second <div class="container authenticated".> tag so the code looks like this:

```
<div class="container authenticated">
  Logged in as: <span id="user"></span>
  <div>
    <button onClick="logout()" class="btn btn-primary">Logout</button>
  </div>
</div>
```

__2. In the index.html file, add the following code just before the end of the script tag </script>

```
var logout = function() {
    $.post("/logout", function() {
        $("#user").html('');
        $(".unauthenticated").show();
        $(".authenticated").hide();
    })
    return true;
}
```

__3. Save the file.

__4. In the DemoApplication.java, modify the **configure** method so the code looks like this: (Note: changes are shown in bold and make sure to remove the ; from the previous line. To make it easier, you can copy/paste text from snippet-2.txt)

```
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http
    .csrf(c -> c
         .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFal
se())
    )
    .logout(log -> log.logoutSuccessUrl("/").permitAll())
    .authorizeRequests(a -> a
        .antMatchers("/", "/error", "/webjars/**").permitAll()
        .anyRequest().authenticated()
    )...
```

The /logout endpoint requires us to POST to it, and to protect the user from Cross Site Request Forgery (CSRF), it requires a token as part of the request that will be linked to the current session

Many JavaScript frameworks, such as Angular, have built in support for CSRF aka. XSRF. For instance, in Angular, the front end would like the server to send it a cookie called "XSRF-TOKEN" and if it sees that, it will send the value back as a header named "X-XSRF-TOKEN". We can implement the same behaviour in our Query client.

__5. Save the file.

__6. Open index.html for editing.

__7. In the body tag, before the existing script tag, add the following script reference.

```
<script type="text/javascript"
src="/webjars/js-cookie/js.cookie.js"></script>
```

__8. In the body tag, locate the custom script where the **$.get(…)** method is called and add the following code before it: (Note: The code snippet is also available in snippet3-.txt)

```
$.ajaxSetup({
  beforeSend : function(xhr, settings) {
    if (settings.type == 'POST' || settings.type == 'PUT'
        || settings.type == 'DELETE') {
      if (!(/^http:.*/.test(settings.url) || /^https:.*/
        .test(settings.url))) {
        // Only send the token to relative URLs i.e. locally.
        xhr.setRequestHeader("X-XSRF-TOKEN",
          Cookies.get('XSRF-TOKEN'));
      }
    }
  }
});
```

__9. Save the file.

__10. In the Command Prompt window run the following command to start the server:

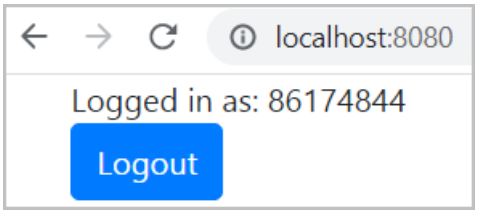
```
mvn spring-boot:run
```

__11. In the web browser enter the following URL:

```
http://localhost:8080
```

__12. If you are not signed then click the GitHub link to sign in.

It should show a message like below.



__13. Click the Logout button.

It should bring you back to the login page.

__14. Stop the server.

__15. Close the browser and command prompt windows.

Part 6 - Review

In this lab, you used Spring Boot's built-in OAuth2 functionality.

Lab 4 - OpenAPI (Swagger)

In this lab we will use a pre-built version of the car inventory service that has the SwaggerUI enabled you will explore the UI and try some of the testing features.

NOTE: for this lab you need to log into and use the VM_WA2785.

Part 1 - Use SwaggerUI

In this part we will start the car inventory service with the embedded SwaggerUI. We will then explore the UI and try some of the testing features.

All the file for this lab are in the **C:\LabFiles\swagger-rest** directory.

__1. Open a DOS shell.

__2. Navigate to:

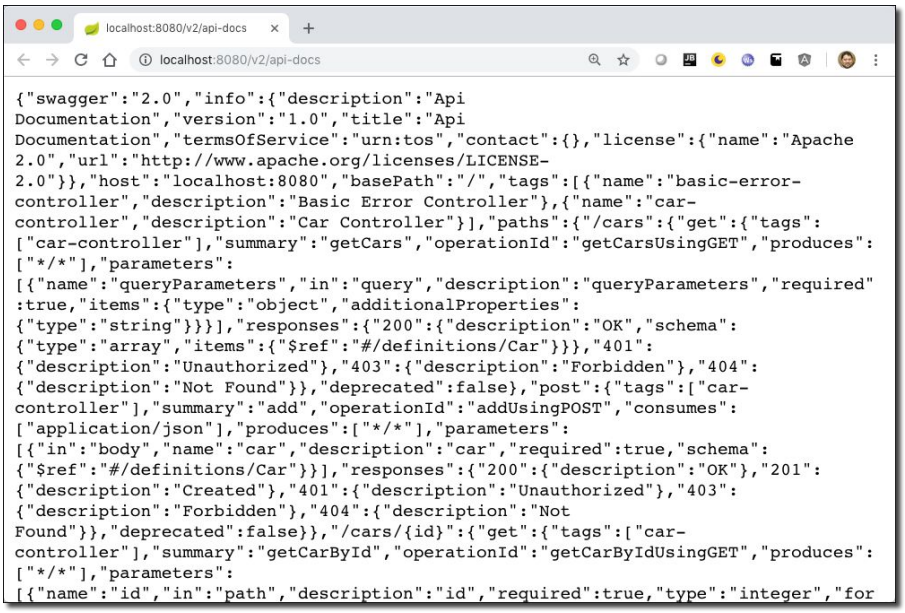
```
cd C:\LabFiles\swagger-rest
```

__3. Execute the command:

```
mvn spring-boot:run
```

__4. Open a browser and navigate to **http://localhost:8080/v2/api-docs**

You should see the OpenAPI definition of the car service.



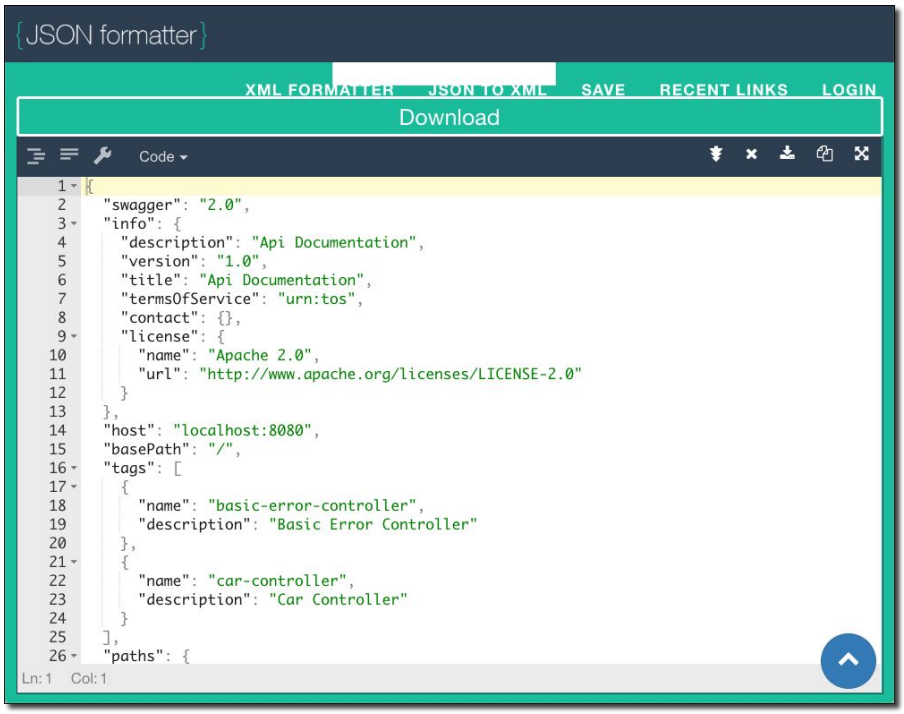
{"swagger":"2.0","info":{"description":"Api Documentation","version":"1.0","title":"Api Documentation","termsOfService":"urn:tos","contact":{},"license":{"name":"Apache 2.0","url":"http://www.apache.org/licenses/LICENSE-2.0"}},"host":"localhost:8080","basePath":"/","tags":[{"name":"basic-error-controller","description":"Basic Error Controller"},{"name":"car-controller","description":"Car Controller"}],"paths":{"/cars":{"get":{"tags":["car-controller"],"summary":"getCars","operationId":"getCarsUsingGET","produces":["*/*"],"parameters":[{"name":"queryParameters","in":"query","description":"queryParameters","required":true,"items":{"type":"object","additionalProperties":{"type":"string"}}}],"responses":{"200":{"description":"OK","schema":{"type":"array","items":{"$ref":"#/definitions/Car"}}},"401":{"description":"Unauthorized"},"403":{"description":"Forbidden"},"404":{"description":"Not Found"}},"deprecated":false},"post":{"tags":["car-controller"],"summary":"add","operationId":"addUsingPOST","consumes":["application/json"],"produces":["*/*"],"parameters":[{"in":"body","name":"car","description":"car","required":true,"schema":{"$ref":"#/definitions/Car"}}],"responses":{"200":{"description":"OK"},"201":{"description":"Created"},"401":{"description":"Unauthorized"},"403":{"description":"Forbidden"},"404":{"description":"Not Found"}},"deprecated":false}},"/cars/{id}":{"get":{"tags":["car-controller"],"summary":"getCarById","operationId":"getCarByIdUsingGET","produces":["*/*"],"parameters":[{"name":"id","in":"path","description":"id","required":true,"type":"integer","for

__5. Copy the JSON to the clipboard.

__6. In the browser navigate to **https://jsonformatter.org/json-pretty-print**

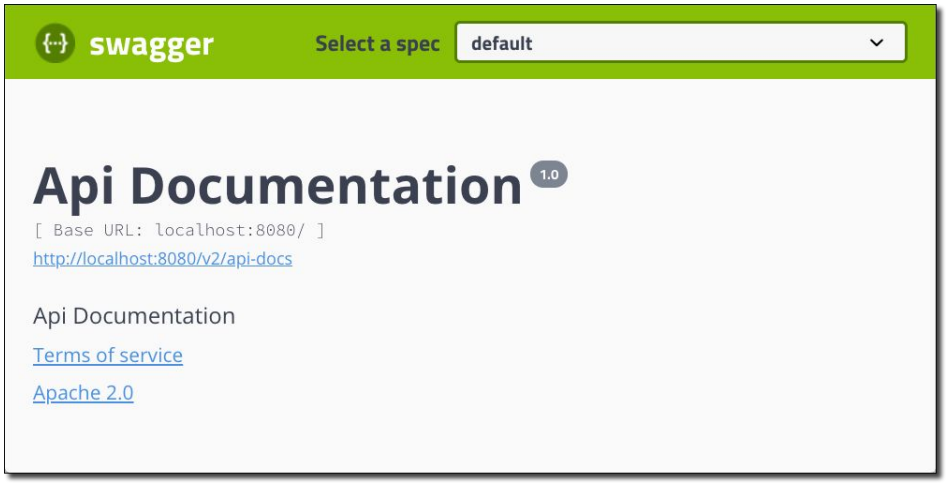
__7. Paste the JSON into the window and press the **Make Pretty** button.

You should see a formatted version of the OpenAPI.

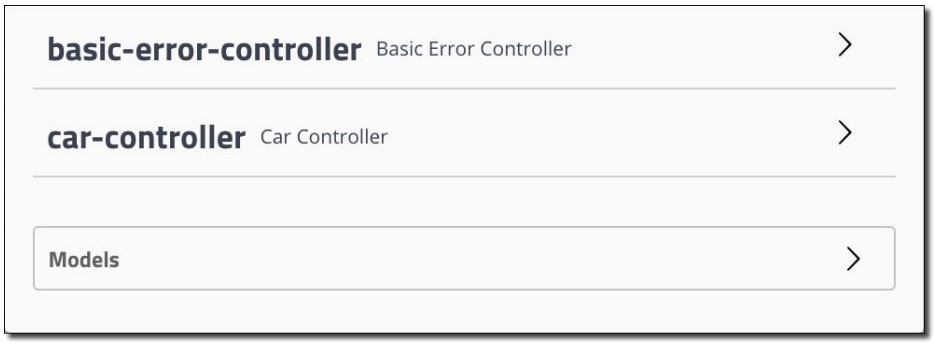


__8. Navigate to **http://localhost:8080/swagger-ui.html**

You should see the swagger UI.

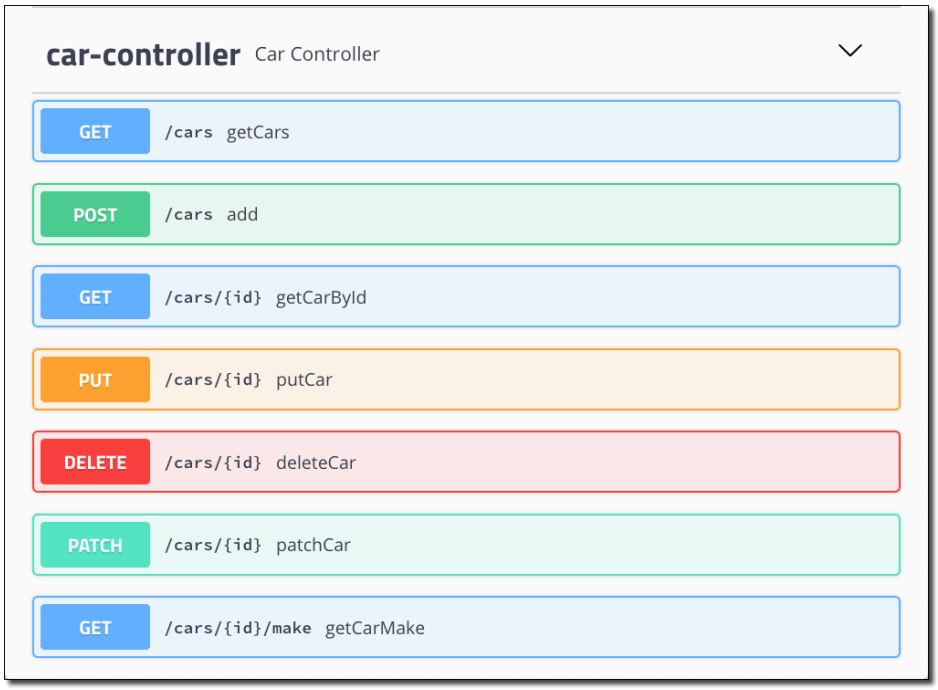


__9. Scroll down to see the controllers and model.



__10. Expand **models** and then expand **Car** to see the data types used in the service.

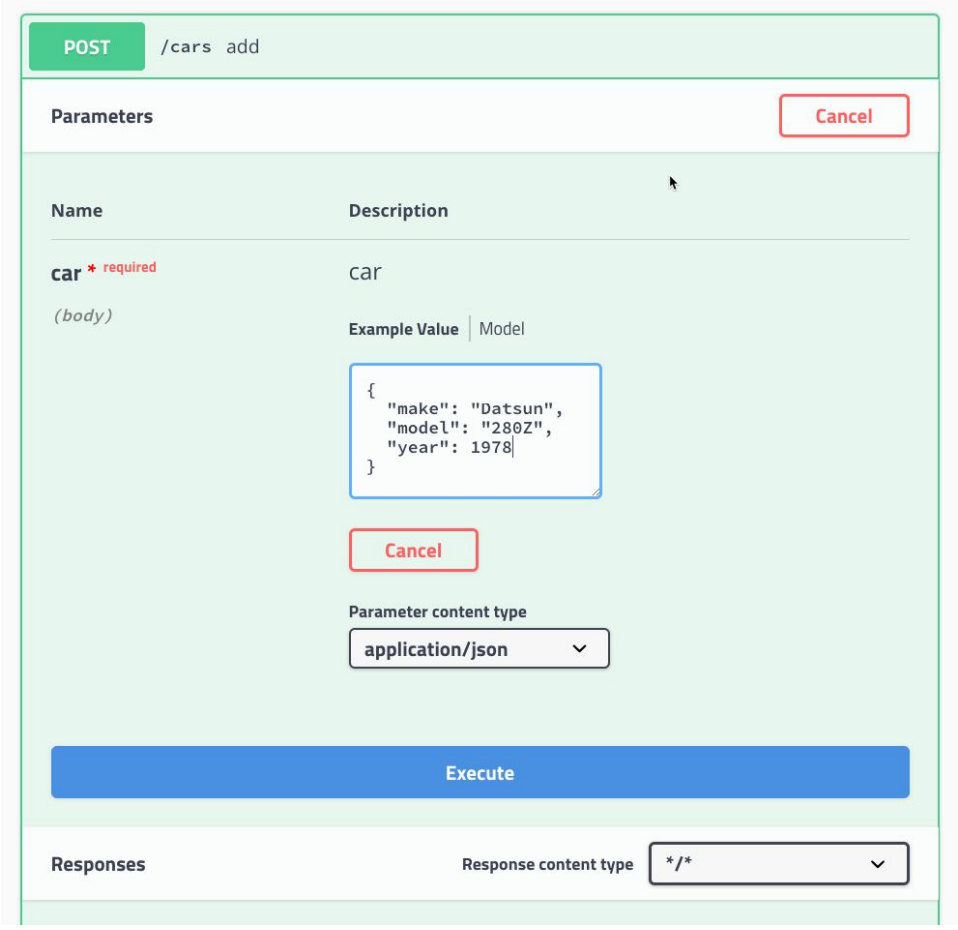
__11. Expand the CarController to see the supported operations.



__12. Click **POST** method.

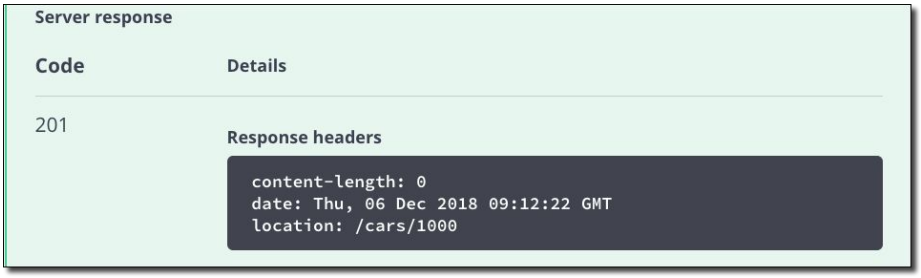
__13. Click the **Try it out** button.

__14. Enter a car in the body. (you do not need the id field since the POST request will set it)



__15. Click **Execute**.

Look at the location header. In this example the URI of the newly inserted car is **/cars/1000**

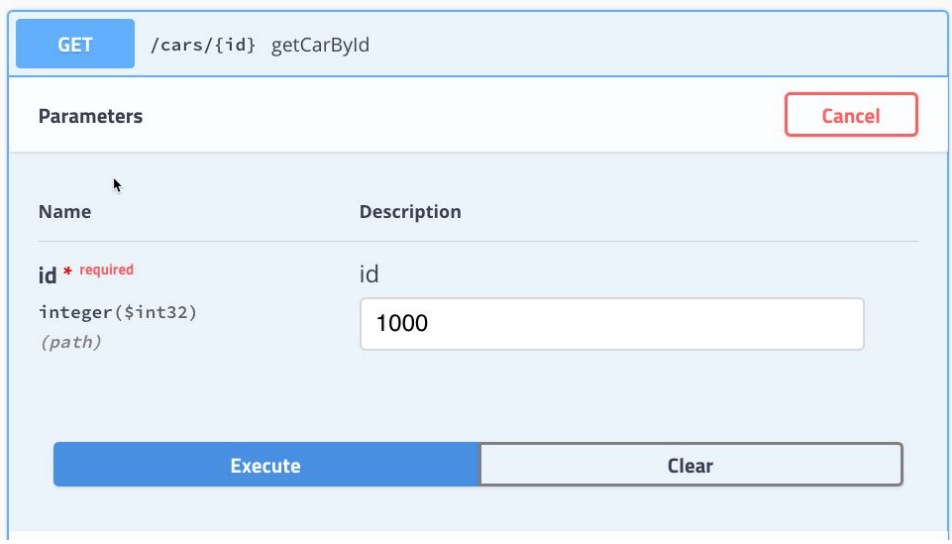


__16. Select the **GET /cars/{id}getCarById**

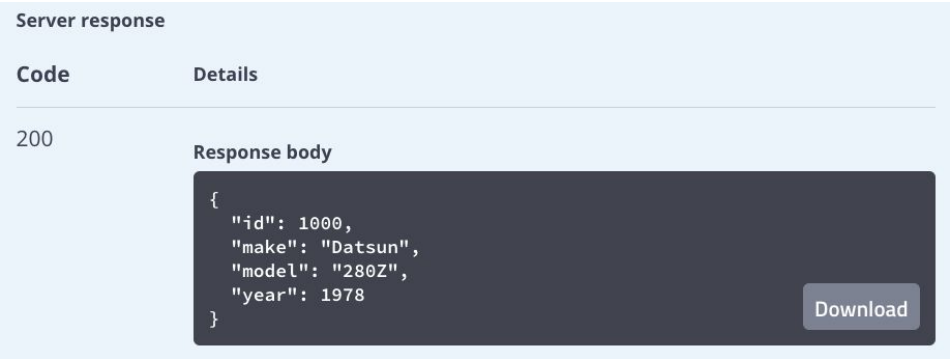
If it is not viosible, then click GET /cars/{id} and it will expand and show: /cars/{id}getCarById

__17. Press the **Try it out** button.

__18. Enter **1000** in the id field and click **Execute**.



You should see the following response.



__19. Try the other methods. (Hint: use PUT or POST to create some more cars)

__20. In the shell press CTRL+C to stop the server.

__21. Close all.

Part 2 - Review

In this lab we looked at the OpenAPI service definition for the Car inventory service. Then we used the SwaggerUI to test service operations.

Lab 5 - Manually Test an API

In this lab you will test REST API using Postman and cURL.

The sample API supports following operations:

Route	HTTP Verb	Description
/api/bears	GET	Get all the bears.
/api/bears	POST	Create a bear.
/api/bears/:bear_id	GET	Get a single bear.
/api/bears/:bear_id	PUT	Update a bear with new info.
/api/bears/:bear_id	DELETE	Delete a bear.

In this Lab you will be continue working in the machine called WA2678-REL_2_1

Part 1 - Start a sample REST API

In this part you will run a sample Node.js application which uses Express module to create a REST API.

__1. Open **Command Prompt** from the **Start Menu**.

__2. Start mongodb:

```
mongod
```

__3. Click Allow access to continue if prompt.

Mongodb will start.

```
2017-09-11T10:37:28.926-0400 I INDEX    [initandlisten] allocating new ns file C:\data\db\local.ns,
2017-09-11T10:37:28.984-0400 I STORAGE  [FileAllocator] allocating new datafile C:\data\db\local.0,
2017-09-11T10:37:28.984-0400 I STORAGE  [FileAllocator] creating directory C:\data\db\_tmp
2017-09-11T10:37:29.184-0400 I STORAGE  [FileAllocator] done allocating datafile C:\data\db\local.0,
2017-09-11T10:37:29.187-0400 I NETWORK  [initandlisten] waiting for connections on port 27017
```

__4. Open another **Command Prompt** from the **Start Menu**.

__5. Switch to the **sample_api** directory:

```
cd c:\LabFiles\sample_api
```

Note: If you don't have the sample_api directory, then extract sample-api.zip file located under c:\LabFiles.

__6. Start the application:

```
node server.js
```

__7. Open Chrome web browser and enter following URL to verify the REST API is running:

```
http://localhost:8080/api
```

Notice it displays a message "hooray! welcome to our api!"

__8. In the web browser, enter following URL to get list of existing records:

```
http://localhost:8080/api/bears
```

Notice it returns an empty array.

__9. Close the web browser.

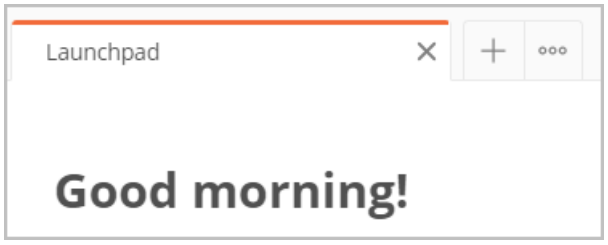
Part 2 - Manually Test API using Postman

In this part you will use Postman to test the sample API.

__1. From the **Start Menu**, launch Postman.

__2. You may see an update available dialog, click **Dismiss**.

__3. Close the Launchpad tab, if present.

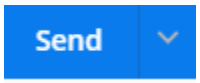


__4. Click the + to open a new tab.

__5. Ensure the method is set to **GET**.

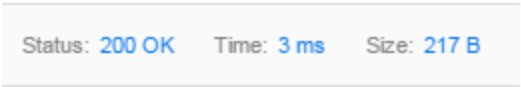
__6. Enter the URL **http://localhost:8080/api/**

__7. Click **Send** button.



The result should look like this:

It should also show additional information like this:



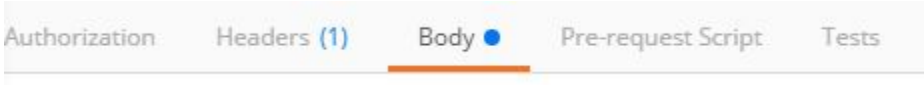
Part 3 - Manually Test POST and GET Operation

In this part you will manually test POST and GET operations using Postman.

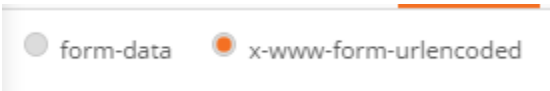
__1. Open another tab and change method to **POST**.

__2. Enter **http://localhost:8080/api/bears** in the URL:

__3. Click **Body** tab.



__4. Click **x-www-form-urlencoded** radio button.



Note: x-form-urlencoded is useful for relatively simple text values. For binary, or for huge values, form-data is used.

__5. Enter following values in the fields:

```
Key: name
Value: A Bear
```

__6. Click **Send** button.

Notice it displays a message "Bear created!"

__7. Replace the values as follows:

```
Key: name
Value: Another Bear
```

__8. Click **Send** button.

Notice it displays a message "Bear created!"

__9. In a new tab, change method to **GET**.

__10. Enter **http://localhost:8080/api/bears** in the URL.

__11. Click **Send** button.

Notice it displays result like this:

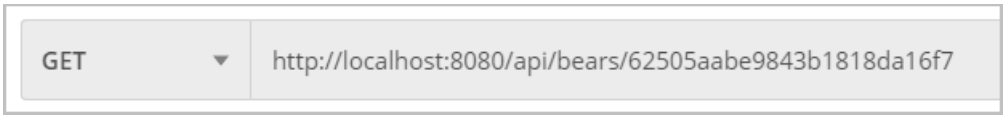
Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON


```
1  [
2      {
3          "_id": "62505aabe9843b1818da16f7",
4          "name": "A Bear",
5          "__v": 0
6      },
7      {
8          "_id": "62505ad2e9843b1818da16f8",
9          "name": "Another Bear",
10         "__v": 0
11     }
12 ]
```


__12. Select the id value of the first record and copy it to the clipboard.

__13. Open a new tab and ensure method is set to GET and enter **http://localhost:8080/api/bears/**<id_value>



__14. Click **Send** button.

Notice it shows result like this:

Part 4 - Manually Test PUT Operation

In this part you will manually test PUT operation using Postman.

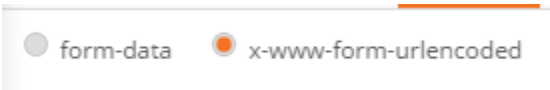
__1. In a new tab, change the method to PUT.

__2. Click **Body** tab.

__3. Using the previous value, enter **http://localhost:8080/api/bears/**<id_value> in the URL.

__4. Click **Body** tab.

__5. Click **x-www-form-urlencoded** radio button.



Note: x-form-urlencoded is useful for relatively simple text values. For binary, or for huge values, form-data is used.

__6. Enter following values in the fields:

```
Key: name
Value:  Care Bear
```

__7. Click **Send** button.

Notice it displays a message "Bear updated!"

__8. Add a new tab and change method to **GET** and enter **http://localhost:8080/api/bears** in the URL.

__9. Click **Send** button.

Notice the value of a record shows up as "Care Bear"

Part 5 - Manually Test DELETE Operation

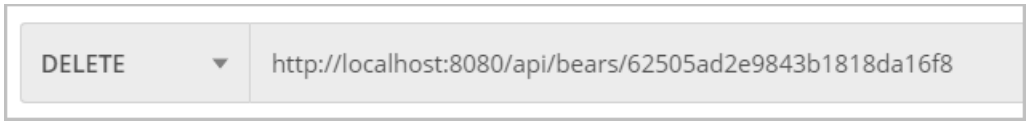
__1. Add a new tab and change method to **DELETE**.

In this part you will manually test DELETE operation using Postman.

__2. Enter **http://localhost:8080/api/bears** in the URL.

__3. Go back to the tab showing all records, select the id of "Another Bear" and copy it to the clipboard and paste it at the end of the URL:

```
Example: http://localhost:8080/api/bears/62505ad2e9843b1818da16f8
```



__4. Click **Send** button.

Notice it displays a message "Successfully deleted"

__5. In a new tab, change method to **GET** and enter **http://localhost:8080/api/bears** in the URL.

__6. Click **Send** button.

Notice "Another Bear" record is no longer listed.

__7. Close the Postman window.

__8. Keep other 2 Command Prompt windows open since they will be used in the next portion of this lab.

Part 6 - Getting Started with cURL

In this part you will open another Command Prompt window and connect to the REST API.

__1. Open another Command Prompt window from the Start Menu.

__2. Switch to the Workspace directory:

```
cd C:\Workspace
```

__3. Run following command:

```
curl http://localhost:8080/api
```

Notice it shows the message "hooray! welcome to our api!". It uses GET as the default method.

__4. Specify method / HTTP verb explicitly:

```
curl -X GET http://localhost:8080/api
```

Notice it displays the same message as seen previously. This time around you have specified the method explicitly.

__5. View header details:

```
curl -i -X GET http://localhost:8080/api
```

Notice it displays result like this:

```
C:\Workspace>curl -i -X GET http://localhost:8080/api
HTTP/1.1 200 OK
X-Powered-By: Express
content-type: application/json
content-length: 41
etag: "-1764523989"
Date: Fri, 08 Apr 2022 17:20:35 GMT
Connection: keep-alive
Keep-Alive: timeout=5

{"message":"hooray! welcome to our api!"}
```

Part 7 - Manually Test POST and GET Operation

In this part you will manually test POST and GET operations using cURL.

__1. Under the **C:\Workspace** directory, create a new file named **data.json** with Notepad.

__2. Add following text to the data.json file:

```
{"name": "Another Bear"}
```

__3. Save the file.

__4. Run following command to insert a new record by making POST API call (in 1 line):

```
curl -i -X POST -H "Content-Type:application/json"
http://localhost:8080/api/bears -d @data.json
```

Notice it shows the message "Bear created".

__5. Run following command to get list of all records:

```
curl -i -X GET http://localhost:8080/api/bears
```

The output should look like this:

```
C:\Workspace>curl -i -X GET http://localhost:8080/api/bears
HTTP/1.1 200 OK
X-Powered-By: Express
content-type: application/json
content-length: 128
etag: "-1934102237"
Date: Fri, 08 Apr 2022 17:23:01 GMT
Connection: keep-alive
Keep-Alive: timeout=5

[{"_id":"62505aabe9843b1818da16f7","name":"Care Bear","__v":0},{"_id":"62506f6ce
9843b1818da16f9","name":"Another Bear","__v":0}]
```

__6. Select the id value of first record and copy it to the clipboard.

__7. Run following command to return the first record:

```
curl -i -X GET http://localhost:8080/api/bears/<id value>
```

The output should look like this:

```
C:\Workspace>curl -i -X GET http://localhost:8080/api/bears/62505aabe9843b1818da
16f7
HTTP/1.1 200 OK
X-Powered-By: Express
content-type: application/json
content-length: 61
etag: "-543070538"
Date: Fri, 08 Apr 2022 17:24:49 GMT
Connection: keep-alive
Keep-Alive: timeout=5

{"_id":"62505aabe9843b1818da16f7","name":"Care Bear","__v":0}
```

Part 8 - Manually Test PUT Operation

In this part you will manually test PUT operation using cURL.

__1. Open **data.json** file and modify its contents like this:

```
{"name": "Grizzly Bear"}
```

__2. Save the file.

__3. Enter the following command and make sure to change the ID for the first item:

```
curl -i -X PUT -H "Content-Type:application/json"
http://localhost:8080/api/bears/<id value> -d @data.json
```

__4. Execute the command.

Notice it shows a message "Bear updated!"

__5. Run following command to get a list of all records.

```
curl -i -X GET http://localhost:8080/api/bears
```

Notice it shows result like this (Grizzly Bear is one of the records)

```
[{"_id":"62505aabe9843b1818da16f7","name":"Grizzly Bear","__v":0},{"_id":"62506f
6ce9843b1818da16f9","name":"Another Bear","__v":0}]
```

Part 9 - Manually Test DELETE Operation

In this part you will manually test DELETE operation using cURL.

__1. Copy id value of "Grizzly Bear" to the clipboard.

__2. Run following command:

```
curl -i -X DELETE http://localhost:8080/api/bears/<id value>
```

Notice it shows a message "Bear updated!"

__3. Get a list of all records and verify "Grizzly Bear" record has been deleted:

```
curl -i -X GET http://localhost:8080/api/bears
```

__4. Close the cURL Command Prompt window.

__5. Press CTRL+C in Command Prompt windows where node.exe and mongodb are running.

__6. Close all Command Prompt windows.

Part 10 - Review

In this lab you tested REST API using Postman and cURL.