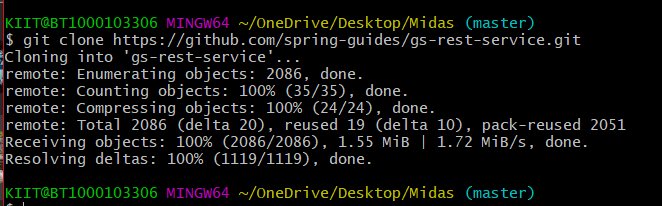
Midas Lab QA internship BE

# **Getting Started:**

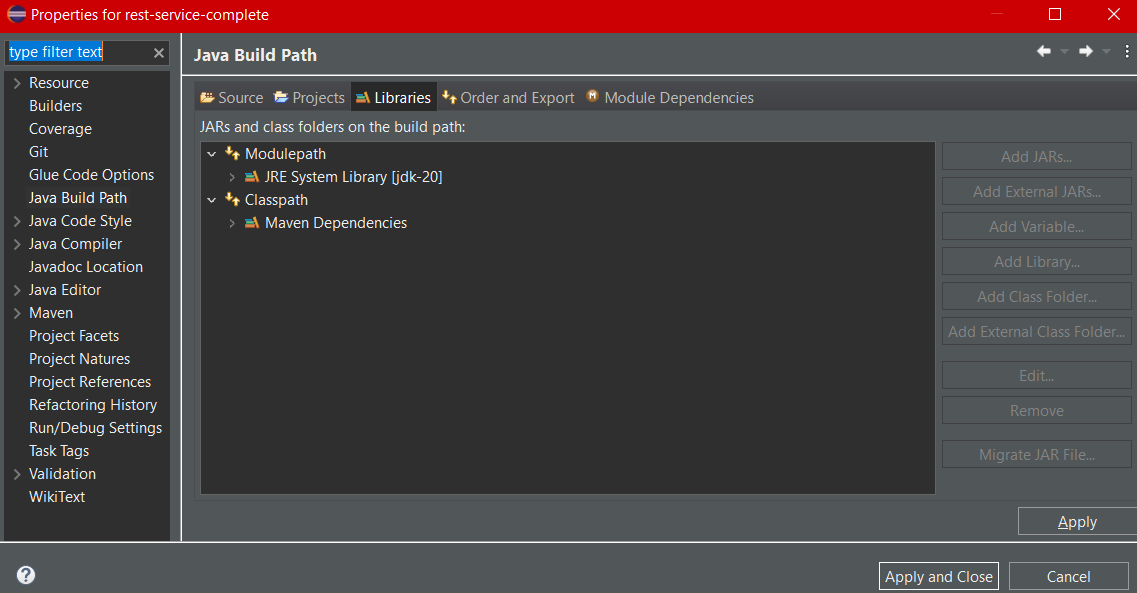
### Cloning the project from the git repository into the local workspace:

* **git clone https://github.com/spring-guides/gs-rest-service.git**



# **Setting up IDE:**

**- Checking the current Java version – v20**



# **Understanding the project:**

* The project is based on the spring-boot framework which is based on the MVC web-based model. **M (Model) V (View) C (Controller).**
* **Model –** A model can be an object or collection of objects that contains the data of the application.
* **View –** A view is used for displaying the information to the user in a specific format.
* **Controller –** It contains the logical part of the application.

**Greeting.java (Model):** It is a Java record that replaces the ideal POJO class (Plain old Java object). Java record initializes the parameters, toString, etc automatically reducing boilerplates.

**package com.example.restservice;**

**public record Greeting(long id, String content) { }**

**RestServiceApplication.java (View):**

**package com.example.restservice;**

**import org.springframework.boot.SpringApplication;**

**import org.springframework.boot.autoconfigure.SpringBootApplication;**

***@SpringBootApplication***

**public class RestServiceApplication {**

**public static void main(String[] args) {**

**System.*out*.println("reached");**

**SpringApplication.*run*(RestServiceApplication.class, args);**

**}**

**}**

**GreetingController.java (Controller):**

*@RestController*

public class GreetingController {

private static final String ***template*** = "Hello, %s!";

private final AtomicLong counter = new AtomicLong();

*@GetMapping*("/greeting")

public Greeting greeting(*@RequestParam*(value = "name", defaultValue = "World") String name) {

return new Greeting(counter.incrementAndGet(), String.*format*(***template***, name));

**GreetingControllerTests.java:**

These tests ensure that the GreetingController responds correctly to requests with and without a "name" query parameter. These tests are important for verifying the behavior of your controller and ensuring that it produces the expected responses.

***@SpringBootTest***

***@AutoConfigureMockMvc***

**public class GreetingControllerTests {**

***@Autowired***

**private MockMvc mockMvc;**

***@Test***

**public void noParamGreetingShouldReturnDefaultMessage() throws Exception {**

**//get request**

**this.mockMvc.perform(*get*("/greeting")).andDo(*print*()).andExpect(*status*().isOk())**

**.andExpect(*jsonPath*("$.content").value("Hello, World!"));**

**}**

***@Test***

**public void paramGreetingShouldReturnTailoredMessage() throws Exception {**

**//get request with query parameter "name"**

**this.mockMvc.perform(*get*("/greeting").param("name", "Spring Community"))**

**.andDo(*print*()).andExpect(*status*().isOk())**

**.andExpect(*jsonPath*("$.content").value("Hello, Spring Community!"));**

**}**

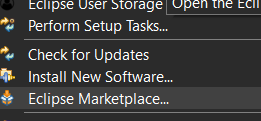
# 

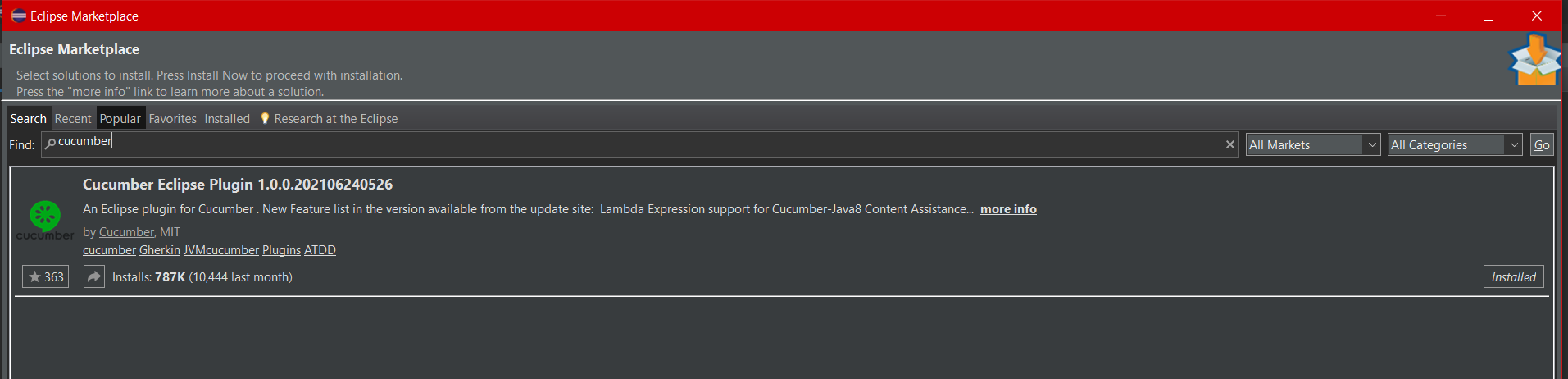
# 

# 

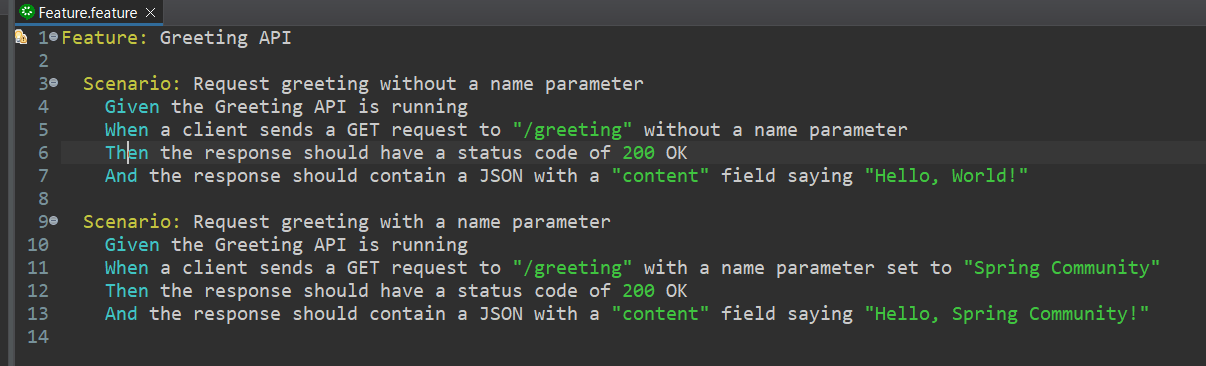
# **Setting up Cucumber environment in eclipse**

* **Eclipse Marketplace - > Cucumber**

****

****

Through this, we can access the **.feature file.** The feature file is the essential segment of the cucumber tool, which is used to write acceptance steps for automation testing. The Gherkin language test scenarios describe the application's expected behavior.



## **Feature.feature :-**

Feature: Greeting API

Scenario: Request greeting without a name parameter

Given the Greeting API is running

When a client sends a GET request to "/greeting" without a name parameter

Then the response should have a status code of 200 OK

And the response should contain a JSON with a "content" field saying "Hello, World!"

Scenario: Request greeting with a name parameter

Given the Greeting API is running

When a client sends a GET request to "/greeting" with a name parameter set to "Spring Community"

Then the response should have a status code of 200 OK

And the response should contain a JSON with a "content" field saying "Hello, Spring Community!"

# **Setting pom.xml**

**What is pom.xml ?**

The pom.xml file is a configuration file used in Maven-based Java projects. It contains metadata about the project, its dependencies, build settings, plugins, and other configuration details required by Maven to manage the project's lifecycle. The pom.xml file serves as a central configuration file that Maven uses to understand how to build, test, and package the project. It includes information such as project name, version, dependencies on external libraries, and other project-specific settings. The POM file helps automate the build process and ensures consistency in project management across different development environments.

## **Adding required cucumber dependencies**

**<!-- Cucumber dependencies -->**

<!-- https://mvnrepository.com/artifact/io.cucumber/cucumber-java -->

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-java</artifactId>

<version>7.15.0</version>

</dependency>

**<!-- JUnit dependency if not already present -->**

<!-- https://mvnrepository.com/artifact/io.cucumber/cucumber-junit -->

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-junit</artifactId>

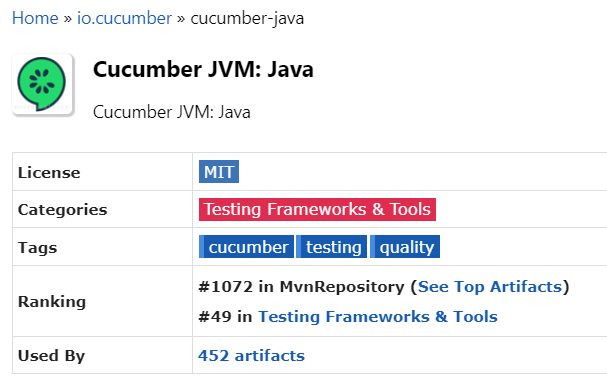
<version>7.15.0</version>

<scope>test</scope>

</dependency>

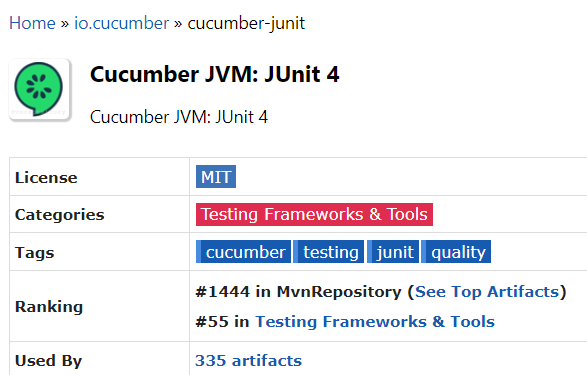
### **SOURCES :**

* **Cucumber -java**  
  <https://mvnrepository.com/artifact/io.cucumber/cucumber-java>



* **Cucumber -Junit**

<https://mvnrepository.com/artifact/io.cucumber/cucumber-junit>



### **Pom.xml**

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.2.0</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<groupId>com.example</groupId>

<artifactId>rest-service-complete</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>rest-service-complete</name>

<description>Demo project for Spring Boot</description>

<properties>

<!-- Remove the java.version property to use the default Java version -->

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<!-- Cucumber dependencies -->

<!-- https://mvnrepository.com/artifact/io.cucumber/cucumber-java -->

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-java</artifactId>

<version>7.15.0</version>

</dependency>

<!-- JUnit dependency if not already present -->

<!-- https://mvnrepository.com/artifact/io.cucumber/cucumber-junit -->

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-junit</artifactId>

<version>7.15.0</version>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<configuration>

<includes>

<include>\*\*/\*Test.java</include>

<include>\*\*/\*Tests.java</include>

<include>\*\*/\*IT.java</include>

<include>\*\*/\*Cucumber.java</include>

<!-- Cucumber runner class -->

</includes>

</configuration>

</plugin>

</plugins>

</build>

</project>

# 

# **Step Definitions:**

It implements the steps for each scenario mentioned in the feature file.

**GreetingApiStepDefinitions.java**

***@SpringBootTest***

***@AutoConfigureMockMvc***

**public class GreetingApiStepDefinitions {**

***@Autowired***

**private MockMvc mockMvc;**

***@Given*("the Greeting API is running")**

**public void theGreetingAPIIsRunning() {**

**// This step is usually used to set up the test environment.**

**// In this case, the API is already running as part of the Spring Boot application.**

**}**

***@When*("a client sends a GET request to {string} without a name parameter")**

**public void aClientSendsAGETRequestToWithoutANameParameter(String path) throws Exception {**

**mockMvc.perform(*get*(path)).andExpect(*status*().isOk());**

**}**

***@When*("a client sends a GET request to {string} with a name parameter set to {string}")**

**public void aClientSendsAGETRequestToWithANameParameterSetTo(String path, String name) throws Exception {**

**mockMvc.perform(*get*(path).param("name", name)).andExpect(*status*().isOk());**

**}**

***@Then*("the response should have a status code of {int} OK")**

**public void theResponseShouldHaveAStatusCodeOfOK(int statusCode) {**

**// The status code is already checked in the "When" steps.**

**}**

***@Then*("the response should contain a JSON with a {string} field saying {string}")**

**public void theResponseShouldContainAJSONWithAFieldSaying(String field, String message) throws Exception {**

**mockMvc.perform(*get*("/greeting")).andExpect(*jsonPath*("$." + field).value(message));**

**}**

**}**

* **@Given("the Greeting API is running"):** Describes a setup step indicating that the Greeting API is running. In this case, it's assumed that the API is part of the Spring Boot application under test.
* **@When("a client sends a GET request to {string} without a name parameter"):** Describes a step where a client sends a GET request to a specified path without providing a name parameter. The test performs this action using MockMvc and expects an HTTP status of OK (200).
* **@When("a client sends a GET request to {string} with a name parameter set to {string}"):** Describes a similar step, but this time the client sends a GET request with a specified name parameter. Again, the test uses MockMvc to perform the action and expects an HTTP status of OK.
* **@Then("the response should have a status code of {int} OK"):** Describes an assertion step where the HTTP status code of the response is verified. The expected status code is provided as a parameter.
* **@Then("the response should contain a JSON with a {string} field saying {string}"):** Describes an assertion step where the JSON response is checked to contain a specific field with a specified value. The field name and value are provided as parameters.

# 

# 

# **TestRunner.java**

This class is a test runner for Cucumber tests, and it's configured to look for feature files and step definitions in specific packages. The test results will be displayed in a readable format and also stored as HTML reports in the specified directory (target/cucumber-reports). To execute these tests, you can run this class using a JUnit test runner.

***@RunWith*(Cucumber.class)**

***@CucumberOptions*(**

**features = {"src/main/java/com/example/restservice"},**

**glue = {"com.example.restservice"},**

**plugin = {"pretty", "html:target/cucumber-reports"}**

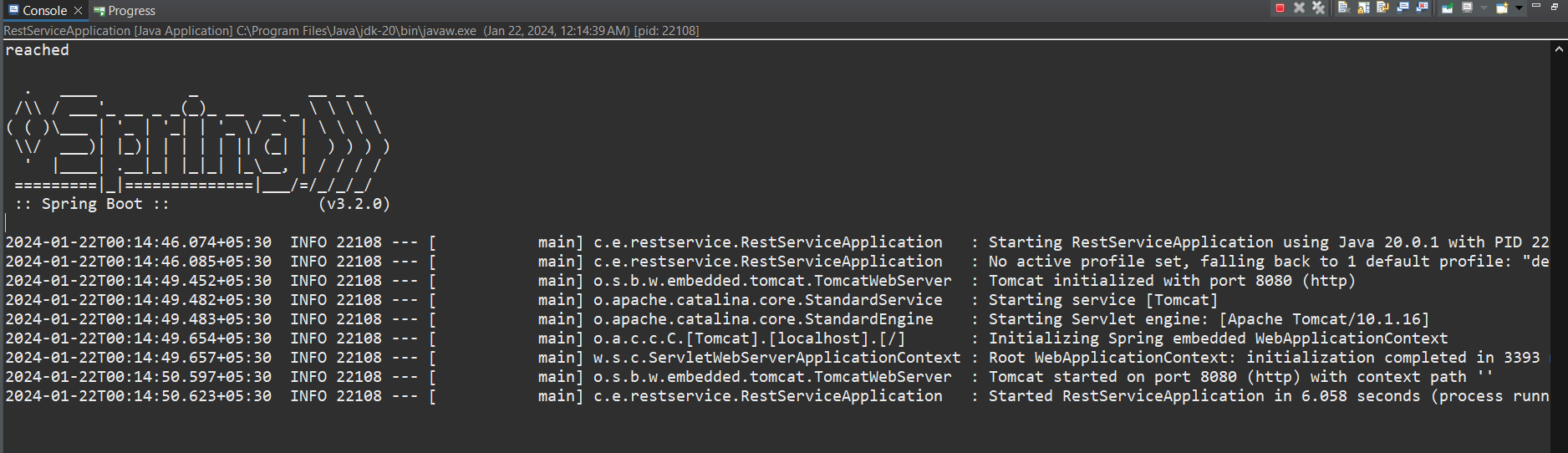
**)**

**public class TestRunner {**

**}**

* **features = {"src/main/java/com/example/restservice"}:** Specifies the location of the feature files. Feature files typically contain Gherkin syntax describing behavior-driven scenarios.\
* **glue = {"com.example.restservice"}:** Specifies the package(s) where Cucumber should look for step definitions. In this case, it points to the package com.example.restservice, where your step definitions are expected to be.
* **plugin = {"pretty", "html:target/cucumber-reports"}:** Configures the output formats for the test results. In this case, it specifies that the results should be displayed in a readable format (pretty) and also generate HTML reports in the target/cucumber-reports directory.

# **Initializing Springboot**



## **Junit :**

