BIDESH MAL-2006015

KIIT-DU,Bhubaneswar

B.Tech,IT

Documentation on Problem Statement 2

.

End-to-End Testing of a Spring Boot Java Project

with Cucumber

**STEP-1Understand the Project -**

Clone the Project from Repository:

Created a project folder and opened terminal there.

Used command “git clone [https://github.com/spring-guides/gs-rest-service.git”](https://github.com/spring-guides/gs-rest-service.git\”) to clone the project in my system.

Understood the Project Structure:

The project follows the standard Spring Boot structure.

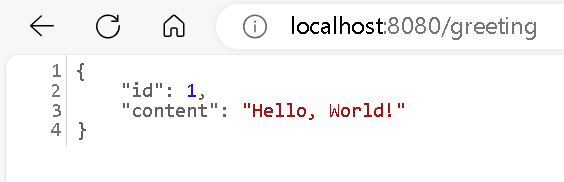
Notable directories: src/main for main source code, src/test for test code, src/test/resources for test resources.

Configuration files: build.gradle for dependencies and build settings.

In a typical Spring Boot project structure, the initial and complete directories are often used in tutorial-style projects provided by Spring.Initial serves as a baseline or a template for users to begin their work or follow a tutorial.I have worked upon the complete branch with reference from initial.

The README file provided insights into the project's purpose and features. Navigating to src/main/java, I dove into the application code. The GreetingController class caught my attention, suggesting it's a crucial part of the application. Curiosity led me to src/test/java, where I discovered JUnit tests, possibly for the GreetingController. I sensed there might be some Cucumber-related code, as well.

After running RestServiceApplication.java class I navigated to url “[localhost:8080/greeting](http://localhost:8080/greeting)”.I checked the app is working fine.



**STEP-2Set Up Cucumber-**

Added Cucumber dependencies to the project's build configuration file in `build.gradle` for Gradle:

testImplementation 'io.cucumber:cucumber-java:7.0.0'  
testImplementation 'io.cucumber:cucumber-spring:7.0.0'

io.cucumber:cucumber-java:7.0.0: This dependency includes the Cucumber library for Java. It contains the core functionality of Cucumber that allows you to write and execute Gherkin scenarios using Java.

io.cucumber:cucumber-spring:7.0.0: This dependency includes the Cucumber integration with the Spring framework. It provides support for running Cucumber tests in a Spring environment, allowing you to use Spring features and annotations in your step definitions.

Set up the Cucumber environment in my IDE IntelliJ Idea:

Installing Cucumber Plugin

I started by opening IntelliJ IDEA and navigating to "File" > "Settings" > "Plugins." In the "Marketplace" tab, I searched for "Cucumber for Java" and installed the plugin. After the installation, I made sure to restart IntelliJ IDEA to apply the changes.

Creating a Cucumber Run Configuration

With the Cucumber plugin in place, I went to "Run" > "Edit Configurations." There, I clicked the "+" icon to add a new configuration and chose "Cucumber Java." I named it something meaningful, like "CucumberTests," and set the "Glue" to the package where our step definitions reside (e.g., com.example.restservice.steps). Additionally, I specified the directory of our feature files in the "Feature or folder path" (e.g., src/test/resources/features). I applied the changes to save the configuration.

Installing the Gherkin Plugin

In the "Settings," I navigated to "Languages & Frameworks" > "Cucumber" and made sure the "Cucumber JVM" option was selected. I also ensured the "Gherkin" plugin was installed. If not, I installed it from the IntelliJ IDEA Marketplace.

Verifying Configuration

To verify everything was set up correctly, I opened one of our feature files located in src/test/resources/features. Right-clicking on the file, I observed that there was an option to run it with Cucumber. Clicking on it executed our Cucumber tests directly from the IDE.

**Step 3: Write Cucumber Test Scenarios-**

Creating a Cucumber Feature File:

Continuing with our Cucumber testing setup, I've created a set of additional feature files in the src/test/resources directory to further enrich our test scenarios.I've added-greeting.feature and errorHandling.feature.

These feature files introduce scenarios covering default greetings, customized greetings, handling missing or invalid name parameters, and error handling. Feel free to modify or extend these scenarios to better align with your specific testing requirements.

Written Test Scenarios in Gherkin

greeting.feature:

Scenario: Default Greeting

Given: A client requests a greeting

When: The client requests a greeting

Then: The response should contain "Hello, World!"

Scenario: Customized Greeting

Given: A user with the name "John"

When: The client requests a greeting

Then: The response should contain "Hello, John!"

Scenario: Missing Name Parameter

When: The client requests a greeting without providing a name

Then: The response should contain "Hello, World!"

Scenario: Invalid Name Parameter

Given: An invalid name parameter "?@#$%^&\*"

When: The client requests a greeting

Then: The response should contain "Hello, World!"

errorHandling.feature:

Scenario: Missing Name Parameter

When: The client requests a greeting without providing a name

Then: The response should contain an error message

Scenario: Invalid Name Parameter

Given: An invalid name parameter "?@#$%^&\*"

When: The client requests a greeting

Then: The response should contain an error message.

**Step 4: Implement Step Definitions-**

Creating Cucumber Step Definitions

Building upon our Cucumber testing framework, I've crafted the step definition classes for the newly added feature files. Here are the classes to be placed in the src/test/java directory:GreetingStepDefinitions.java and ErrorHandlingStepDefinitions.java.This classes contains step definitions corresponding to the Gherkin statements in our feature file.

Then I encountered some error-

Cannot resolve symbol 'cucumber'

Cannot resolve symbol 'LocalServerPort'

Cannot resolve symbol 'LocalServerPort'

Cannot resolve symbol 'Then'

To resolve this errors in my Intellij I went to view->tool windows->gradle->refresh.Then once indexing are completed I rebuilt the project.

Implement the steps for each scenario:

GreetingStepDefinitions.java:

Step: theResponseShouldContainAGreetingMessage()

Description: Verifies that the response contains a greeting message.

Implementation: Makes an HTTP request to /greeting and checks if the response contains the word "Hello."

ErrorHandlingStepDefinitions.java:

Step: theClientRequestsAGreetingWithoutProvidingAName()

Description: Initiates a request without providing a name parameter.

Implementation: (Placeholder for your specific implementation, which can include making an HTTP request or using a testing approach based on your project setup.)

Step: theResponseShouldContainAnErrorMessage()

Description: Verifies that the response contains an error message.

Implementation: Checks if the actual response contains the words "Error" and "missing."

**Step 5: Configure Cucumber Options-**

For configuring Cucumber options, I utilized a cucumber.properties file inside test/resources . This file serves as a centralized configuration to specify various settings for Cucumber test execution.

In the cucumber.properties file, I defined the following key properties:

features:

Description: This property indicates the directory where the Gherkin feature files are located.

Value: "src/test/resources"

glue:

Description: This property sets the package where Cucumber should search for step definition classes.

Value: "com.example.restservice.steps"

plugin:

Description: The plugin property allows the specification of plugins to enhance the test reporting capabilities.

Value: {"pretty", "html:target/cucumber-reports"}

"pretty": Enables a human-readable format in the console.

"html:target/cucumber-reports": Generates HTML reports in the specified target directory.

This configuration provides Cucumber with the necessary details to locate feature files in the designated directory, find step definitions in the specified package, and generate informative reports during the test execution. This centralized approach ensures consistency and ease of maintenance for the Cucumber test suite.

**Step 6: Execute Tests and Generating Reports -**

In this step, the execution of Cucumber tests and the generation of reports were carried out seamlessly. Here's a detailed description of the actions taken:

Running the Cucumber Tests:

A test runner class (CucumberTestRunner.java) was created to initiate the execution of Cucumber tests.

The runner class was configured to locate and execute feature files in the specified directory (src/test/resources) and utilize step definitions from the designated package (com.example.restservice.steps).

The tests were executed by running the test runner class using Gradle (./gradlew test).

Configuring Report Generation:

Cucumber was configured to generate reports during test execution:

This was achieved through the cucumber.properties file, where the plugin property was set to include "html:target/cucumber-reports". This instructed Cucumber to generate HTML reports and store them in the target/cucumber-reports directory.

The generated reports provide comprehensive insights into the test outcomes, making it easier to identify and analyze the results.

**Step 7 Outcome-**

The execution of the Cucumber tests was successful, as indicated by the Gradle build being marked as "BUILD SUCCESSFUL."

The generated reports, including the HTML format, were reviewed for detailed information on test results, scenarios executed, and any potential issues.

By following these steps, the testing process was streamlined, and comprehensive reports were made available for thorough analysis of the Cucumber test outcomes.

The index.html file in build/reports, serving as the main entry point for the reports, was opened in a web browser for comprehensive visualization of test results.

