Cucumber Testing Documentation

Step 1: Setting Up Cucumber

Cucumber is a powerful testing tool that supports Behavior-Driven Development (BDD) by allowing tests to be written in natural language. Integrating Cucumber into a Spring Boot project involves a series of steps to ensure seamless collaboration between development and testing teams. Below is a guide to setting up Cucumber in a Spring Boot project:

. Step 1: Add Cucumber Dependencies

. Maven (pom.xml)

xml

<dependencies>

<!-- Spring Boot Starter Web -->

<dependency>

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

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

</dependency>

<!-- Spring Boot Starter Test -->

<dependency>

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

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

<scope>test</scope>

</dependency>

<!-- Cucumber Dependencies -->

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-java</artifactId>

<version>7.15.0</version> <!-- Replace with the version in use -->

<scope>test</scope>

</dependency>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-junit</artifactId>

<version>7.15.0</version> <!-- Replace with the version in use -->

<scope>test</scope>

</dependency>

</dependencies>

. Gradle (build.gradle)

groovy

dependencies {

implementation 'org.springframework.boot:spring-boot-starter-web'

testImplementation 'org.springframework.boot:spring-boot-starter-test'

// Cucumber Dependencies

testImplementation 'io.cucumber:cucumber-java:7.15.0' // Replace with the version in use

testImplementation 'io.cucumber:cucumber-junit:7.15.0' // Replace with the version in usea

}

. Step 2: Create Cucumber Feature Files

Create feature files in the src/test/resources directory. These files will contain test scenarios written in Gherkin language.

Example feature file (greeting.feature):

gherkin

Feature: Greeting API

Scenario: User greets with a custom name

Given user has a greeting API

When user sends the data as "<name>"

Then the response should be "Hello <name>!"

Scenario: User greets with default name

Given user has a greeting API

When user sends the data as "world"

Then the response should be "Hello world!"

. Step 3: Implement Step Definitions

Create step definition classes in the src/test/java directory. These classes map Gherkin steps to Java code.

Example step definition class (GreetingStepDefinitions.java):

java

package com.example.restservice;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.Then;

import io.cucumber.java.en.When;

public class GreetingStepDefinitions {

@Given("user has a greeting API")

public void givenStep() {

// Implementation goes here

}

@When("user sends the data as {string}")

public void whenStep(String data) {

// Implementation goes here

}

@Then("the response should be {string}")

public void thenStep(String expectedResponse) {

// Implementation goes here

}

}

. Step 4: Configure Cucumber Options

Create a runner class or use a cucumber.properties file to specify features, glue, plugins, etc.

Example cucumber.properties:

properties

cucumber.features=src/test/resources

cucumber.glue=com.example.restservice

cucumber.plugin=pretty,html:target/cucumber-reports/report.html

Now, the Spring Boot project is set up with Cucumber for writing and executing tests. Run the test runner class or use Maven/Gradle commands to execute the Cucumber tests.

Review the generated reports in the specified directory for detailed test results.

Certainly, let's analyze the provided test cases for a greeting API using Cucumber scenarios:

. Test Case 1: User greets with a custom name

gherkin

Given user has a greeting API

When user sends the data as "<name>"

Then the response should be "Hello <name>!"

Explanation:

- \*Given user has a greeting API:\*

- This step sets up the initial conditions, indicating that the user has access to the greeting API.

- \*When user sends the data as "<name>":\*

- This step describes the action taken by the user, sending data to the API with a custom name represented by "<name>" as a placeholder.

- \*Then the response should be "Hello <name>!":\*

- This step specifies the expected outcome. It asserts that the response from the API should be a greeting message containing the custom name provided in the data.

. Test Case 2: User greets with default name

gherkin

Given user has a greeting API

When user sends the data as "world"

Then the response should be "Hello world!"

. Explanation:

- \*Given user has a greeting API:\*

- Similar to the first case, this step sets up the initial conditions, indicating that the user has access to the greeting API.

- \*When user sends the data as "world":\*

- This step describes the action of the user, sending data to the API with the default name "world."

- \*Then the response should be "Hello world!":\*

- This step specifies the expected outcome. It asserts that the response from the API should be a greeting message containing the default name "world."

. Purpose and Significance:

- \*Variety of Scenarios:\*

- These test cases cover scenarios where users interact with the greeting API with both custom and default names. This ensures that the API responds correctly to different input variations.

- \*Behavior Verification:\*

- By using Cucumber's Given-When-Then structure, the test cases clearly express the setup, action, and expected outcomes. This enhances collaboration between developers and non-technical stakeholders.

- \*Readability and Clarity:\*

- Gherkin language promotes readability. Stakeholders can understand and validate the expected behavior without delving into the implementation details.

- \*Easy Maintenance:\*

- Separating scenarios in Gherkin files and step definitions allows for easy maintenance and modification of test cases without impacting the underlying code.

In summary, these test cases demonstrate the flexibility and readability advantages of Cucumber, making it easier to collaborate, verify, and maintain the behavior of the greeting API across different input scenarios.