**Creating Feature Files:**

Feature files are written in Gherkin syntax and contain scenarios that describe the behavior of the system. Create a **\*.feature** file and define your scenarios. Here's an example of a feature file for a simple login feature:

Feature: User Authentication

As a registered user

I want to log in to the system

So that I can access my account

Scenario: Valid User Login

Given I am on the login page

When I enter valid credentials

And I click the login button

Then I should be redirected to the dashboard

And I should see a welcome message

**Implementing Step Definitions:**

Step definitions map each step in the feature file to corresponding automation code. Create step definition classes and define methods to match each step. Here's an example of step definitions for the login feature:

import io.cucumber.java.en.\*;

public class LoginStepDefinitions {

@Given("I am on the login page")

public void iAmOnTheLoginPage() {

// Implementation to navigate to the login page

}

@When("I enter valid credentials")

public void iEnterValidCredentials() {

// Implementation to enter valid credentials

}

@And("I click the login button")

public void iClickTheLoginButton() {

// Implementation to click the login button

}

@Then("I should be redirected to the dashboard")

public void iShouldBeRedirectedToTheDashboard() {

// Implementation to verify redirection to the dashboard

}

@And("I should see a welcome message")

public void iShouldSeeAWelcomeMessage() {

// Implementation to verify the presence of a welcome message

}

}

**Running Tests:**

You can run Cucumber tests using JUnit. Create a JUnit test runner class and specify the path to your feature files. Here's an example of a test runner class:

java

import io.cucumber.junit.Cucumber;

import io.cucumber.junit.CucumberOptions;

import org.junit.runner.RunWith;

@RunWith(Cucumber.class)

@CucumberOptions(features = "src/test/resources/features")

public class TestRunner {

}

Cucumber options

**Features:** Specifies the path to the directory or file containing feature files

@CucumberOptions(features = "src/test/resources/features")

**Glue:** Defines the package where Cucumber should look for step definitions.

@CucumberOptions(glue = "com.example.steps")

**Tags:** Executes scenarios with the specified tags.

@CucumberOptions(tags = "@smokeTest")

**Plugin:** Defines the output format and destination for test results. You can specify multiple plugins to generate different types of reports.

@CucumberOptions(plugin = {"pretty", "html:target/cucumber-reports"})

**Monochrome:** Outputs the console output in a readable format, without colors.

@CucumberOptions(monochrome = true)

**Strict:** Treats undefined and pending steps as errors.

@CucumberOptions(strict = true)

**Dry Run:** Checks if all steps have corresponding step definitions without executing the actual tests.

@CucumberOptions(dryRun = true)

**Step Notifications:** Prints each step in the console as it is executed.

@CucumberOptions(stepNotifications = true)

**Name Patterns:** Filters scenarios by name using regular expressions.

@CucumberOptions(name = "^Login.\*$")

**Background:** Defines a common set of steps to be executed before each scenario in a feature file.

@CucumberOptions(features = "src/test/resources/features", glue = "com.example.steps", background = "@CommonSetup")

These are just a few examples of the available options in Cucumber. You can combine these options to customize your test execution according to your requirements. Remember to import **@CucumberOptions** from **io.cucumber.junit.CucumberOptions** if you're using JUnit or **io.cucumber.testng.CucumberOptions** if you're using TestNG.

**Hook into Cucumber**

Create a TestNG runner class where you'll define Cucumber options and hooks to integrate with ExtentReports.

import io.cucumber.testng.AbstractTestNGCucumberTests;

import io.cucumber.testng.CucumberOptions;

@CucumberOptions(

features = {"src/test/resources/features"},

glue = {"stepdefinitions"},

plugin = {"com.aventstack.extentreports.cucumber.adapter.ExtentCucumberAdapter:"}

)

public class TestRunner extends AbstractTestNGCucumberTests {

// No code needed here, TestNG will automatically pick up Cucumber options

}

**Explanation:**

* **Dependencies**: We added dependencies for Cucumber, TestNG, and ExtentReports in the **pom.xml** file.
* **Base Test Class**: This class sets up ExtentReports. **setUp()** initializes ExtentReports, and **tearDown()** flushes the report after test execution.
* **Test Runner Class**: This class defines Cucumber options, specifying where the feature files and step definitions are located. It also integrates ExtentReports using the **ExtentCucumberAdapter** plugin.
* **Execution**: Execute the tests using TestNG. ExtentReports will generate an HTML report (**extent.html**) containing detailed test results.

### Capture Screenshots on Test Failure

import org.openqa.selenium.OutputType;

import org.openqa.selenium.TakesScreenshot;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.remote.RemoteWebDriver;

public class ScreenshotHelper {

public static void captureScreenshot(WebDriver driver, String screenshotName) {

try {

TakesScreenshot ts = (TakesScreenshot) driver;

byte[] screenshot = ts.getScreenshotAs(OutputType.BYTES);

scenario.fail("Failure Screenshot: " + screenshotName, MediaEntityBuilder.createScreenCaptureFromBytes(screenshot).build());

} catch (Exception e) {

e.printStackTrace();

}

}

}

### Integrate Screenshot Capture with Test Steps

import io.cucumber.java.After;

import io.cucumber.java.Before;

import io.cucumber.java.Scenario;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

public class Hooks {

private WebDriver driver;

@Before

public void setUp(Scenario scenario) {

driver = new ChromeDriver();

BaseTest.scenario = BaseTest.extent.createTest(scenario.getName());

}

@After

public void tearDown(Scenario scenario) {

if (scenario.isFailed()) {

ScreenshotHelper.captureScreenshot((RemoteWebDriver) driver, scenario.getName());

}

driver.quit();

}

}

### Explanation:

* **Screenshot Helper Class**: This class contains a static method **captureScreenshot()** which captures a screenshot when called. It takes WebDriver instance and a name for the screenshot as parameters. It uses Selenium's **TakesScreenshot** interface to capture the screenshot as bytes and then attaches it to the ExtentReports report using **MediaEntityBuilder**.
* **Hooks Class**: This class contains **@Before** and **@After** hooks. The **@Before** hook initializes the WebDriver and ExtentReports scenario. The **@After** hook checks if the scenario failed, and if so, it captures a screenshot using the **ScreenshotHelper** class and attaches it to the report. Finally, it quits the WebDriver.

### Implement Rerun Logic in TestNG

import org.testng.IAnnotationTransformer;

import org.testng.annotations.ITestAnnotation;

import java.lang.reflect.Constructor;

import java.lang.reflect.Method;

public class RerunAnnotationTransformer implements IAnnotationTransformer {

@Override

public void transform(ITestAnnotation annotation, Class testClass, Constructor testConstructor, Method testMethod) {

annotation.setRetryAnalyzer(RetryAnalyzer.class);

}

}

### Create a Retry Analyzer

import org.testng.IRetryAnalyzer;

import org.testng.ITestResult;

public class RetryAnalyzer implements IRetryAnalyzer {

private int count = 0;

private static final int MAX\_RETRY\_COUNT = 3; // Maximum number of retries

@Override

public boolean retry(ITestResult result) {

if (!result.isSuccess()) { // Retry if test fails

if (count < MAX\_RETRY\_COUNT) { // Check if maximum retries reached

count++;

return true;

}

}

return false;

}

}

### Configure TestNG Suite File

In your TestNG suite XML file, specify the listener to use the **RerunAnnotationTransformer** class.

<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">

<suite name="Test Suite">

<listeners>

<listener class-name="your.package.name.RerunAnnotationTransformer"/>

</listeners>

<test name="Test">

<classes>

<!-- Add your test classes here -->

</classes>

</test>

</suite>

### Explanation:

* **Rerun Annotation Transformer**: This class implements the **IAnnotationTransformer** interface, allowing you to modify test annotations at runtime. In this example, it sets the retry analyzer for each test method to **RetryAnalyzer**.
* **Retry Analyzer**: This class implements the **IRetryAnalyzer** interface, which decides whether a test method should be retried or not. It retries a test method up to a certain maximum count if it fails.
* **TestNG Suite Configuration**: In the TestNG suite XML file, you specify the listener to use the **RerunAnnotationTransformer** class. This configuration ensures that the rerun logic is applied to all test methods within the suite.

### Use DataTable as Parameters

#### **Feature file:**

Scenario: Adding numbers

Given I have the following numbers

| 10 |

| 20 |

When I add them

Then the result should be 30

#### **Step Definitions:**

import io.cucumber.datatable.DataTable;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.Then;

import io.cucumber.java.en.When;

import java.util.List;

public class MyStepDefinitions {

@Given("I have the following numbers")

public void i\_have\_the\_following\_numbers(DataTable dataTable) {

List<Integer> numbers = dataTable.asList(Integer.class);

// Process the numbers

}

@When("I add them")

public void i\_add\_them() {

// Add the numbers

}

@Then("the result should be {int}")

public void the\_result\_should\_be(Integer expectedSum) {

// Check the result

}

}

**Explanation:**

* In the step definition method **i\_have\_the\_following\_numbers**, the parameter is of type **DataTable**, which allows you to access the table data directly.
* You can convert the DataTable into a List of specific types, such as **Integer**, **String**, etc., using the **asList** method.

### Use DataTable with Example Keyword

#### **Feature file:**

Scenario Outline: Adding numbers

Given I have the following numbers

| Numbers |

| <num1> |

| <num2> |

When I add them

Then the result should be <result>

Examples:

| num1 | num2 | result |

| 10 | 20 | 30 |

| 15 | 25 | 40 |

**Explanation:**

* In the **Examples** section, you define placeholders (**<num1>**, **<num2>**, **<result>**) for different sets of data.
* Cucumber replaces these placeholders with actual values from the table, and each row represents one scenario execution.

### Use DataTable - List & Map

#### **Feature file:**

Scenario: Passing complex data

Given I have the following data

| Name | Age | Country |

| Alice | 25 | USA |

| Bob | 30 | Canada |

When I process the data

Then it should be successful

#### **Step Definitions:**

import io.cucumber.datatable.DataTable;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.Then;

import io.cucumber.java.en.When;

import java.util.List;

import java.util.Map;

public class MyStepDefinitions {

@Given("I have the following data")

public void i\_have\_the\_following\_data(DataTable dataTable) {

List<Map<String, String>> data = dataTable.asMaps(String.class, String.class);

// Process the data

}

@When("I process the data")

public void i\_process\_the\_data() {

// Process the data

}

@Then("it should be successful")

public void it\_should\_be\_successful() {

// Verify the processing

}

}

**Explanation:**

* In the **i\_have\_the\_following\_data** method, the DataTable is converted into a List of Maps, where each Map represents a row of data with column headers as keys and cell values as values.
* This allows you to work with structured data in a more meaningful way.

### Configure TestNG for Parallel Execution

TestNG provides built-in support for parallel execution. You can specify parallel execution mode at the suite or test level in the testng.xml file.

<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">

<suite name="Test Suite" parallel="tests" thread-count="2">

<test name="Test">

<classes>

<class name="com.example.TestClass1"/>

<class name="com.example.TestClass2"/>

<!-- Add more test classes as needed -->

</classes>

</test>

</suite>

### Implement Cucumber Hooks for Parallel Execution

When running scenarios in parallel, each scenario must have its own isolated context to avoid conflicts. You can achieve this by using TestNG's **ThreadLocal** variables

import io.cucumber.java.Before;

import io.cucumber.java.Scenario;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

import java.util.concurrent.TimeUnit;

public class Hooks {

private static ThreadLocal<WebDriver> webDriver = new ThreadLocal<>();

public static WebDriver getDriver() {

return webDriver.get();

}

@Before

public void setUp(Scenario scenario) {

WebDriver driver = new ChromeDriver();

driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

webDriver.set(driver);

}

// Other hooks for teardown, etc.

}

### Write Cucumber Tests

#### **Example feature file:**

Feature: Shopping Cart

Scenario: Add item to cart

Given I am on the product page

When I add the item to the cart

Then the item should be added successfully

#### **Example step definitions:**

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import static org.junit.Assert.assertTrue;

public class ShoppingCartSteps {

private WebDriver driver = Hooks.getDriver();

@Given("I am on the product page")

public void i\_am\_on\_the\_product\_page() {

driver.get("http://example.com/product");

}

@When("I add the item to the cart")

public void i\_add\_the\_item\_to\_the\_cart() {

driver.findElement(By.id("add-to-cart")).click();

}

@Then("the item should be added successfully")

public void the\_item\_should\_be\_added\_successfully() {

assertTrue(driver.findElement(By.id("cart-count")).getText().equals("1"));

}

}

**Explanation:**

* **TestNG Configuration**: In the testng.xml file, you specify the parallel mode (**parallel="tests"**) and the number of threads (**thread-count**) for parallel execution.
* **ThreadLocal WebDriver**: To ensure thread safety and avoid conflicts when running scenarios in parallel, each thread has its own instance of the WebDriver.
* **Hooks Class**: The **setUp** method initializes the WebDriver instance for each scenario using ChromeDriver.
* **Cucumber Tests**: Write your feature files and step definitions as usual, ensuring they are thread-safe by avoiding shared state between scenarios.

**Tags**

Tags in Cucumber allow you to organize and filter scenarios based on certain criteria. Here's how you can use tags effectively:

### Step 1: Tag Your Scenarios in Feature Files

You can tag scenarios using the **@** symbol followed by a tag name. Tags can be applied at the feature, scenario, or scenario outline level.

#### **Example Feature File:**

Feature: Shopping Cart

@smoke @regression

Scenario: Add item to cart

Given I am on the product page

When I add the item to the cart

Then the item should be added successfully

@regression

Scenario: Remove item from cart

Given I have items in the cart

When I remove the item from the cart

Then the item should be removed successfully

@smoke

Scenario: View cart

Given I have items in the cart

When I view the cart

Then I should see the items in the cart

### Step 2: Run Specific Tagged Scenarios

You can specify which scenarios to run based on their tags using the **@** symbol followed by the tag name.

#### **Command to Run Specific Tags:**

mvn test -Dcucumber.filter.tags="@smoke"

### Step 3: Implement Cucumber Runner Class

In your Cucumber runner class, you can specify which tags to include or exclude when running your tests.

#### **Cucumber Runner Class:**

import io.cucumber.junit.Cucumber;

import io.cucumber.junit.CucumberOptions;

import org.junit.runner.RunWith;

@RunWith(Cucumber.class)

@CucumberOptions(

features = "src/test/resources/features",

glue = "stepdefinitions",

tags = "@regression", // Specify the tag to include or exclude

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

)

public class TestRunner {

// No code needed here

}

**Explanation:**

* **Tagging Scenarios**: Tags provide a way to categorize scenarios based on their characteristics or requirements. In the feature file, you can tag scenarios using the **@** symbol followed by a tag name. You can apply multiple tags to a scenario.
* **Running Specific Tags**: When executing your tests, you can specify which tagged scenarios to run using the **cucumber.filter.tags** system property.
* **Cucumber Runner Class**: In the Cucumber runner class, you can specify the tags to include or exclude using the **@CucumberOptions** annotation. This allows you to control which scenarios get executed based on their tags.

**Hooks**

Hooks in Cucumber are blocks of code that run before or after each scenario. They allow you to set up preconditions and clean up after each scenario, making your tests more modular and maintainable. Here's how you can use hooks effectively:

### Step 1: Implement Hooks in Your Cucumber Project

#### **Step Definitions:**

Create a class containing your hook methods annotated with Cucumber annotations (**@Before** and **@After**).

**import io.cucumber.java.After;**

**import io.cucumber.java.Before;**

**public class Hooks {**

**@Before**

**public void setUp() {**

**// Code to set up preconditions before each scenario**

**System.out.println("Setting up before scenario...");**

**}**

**@After**

**public void tearDown() {**

**// Code to clean up after each scenario**

**System.out.println("Tearing down after scenario...");**

**}**

**}**

**Explanation:**

* The **@Before** annotation indicates that the method should be executed before each scenario.
* The **@After** annotation indicates that the method should be executed after each scenario.
* You can have multiple **@Before** and **@After** hooks in your project.

**Step 2: Configure Step Definitions**

Ensure that your step definitions class is configured to use the hooks.

**import io.cucumber.java.en.Given;**

**import io.cucumber.java.en.Then;**

**import io.cucumber.java.en.When;**

**public class StepDefinitions {**

**@Given("I have a precondition")**

**public void i\_have\_a\_precondition() {**

**// Step definition logic**

**System.out.println("Executing Given step...");**

**}**

**@When("I perform some action")**

**public void i\_perform\_some\_action() {**

**// Step definition logic**

**System.out.println("Executing When step...");**

**}**

**@Then("I should see the expected result")**

**public void i\_should\_see\_the\_expected\_result() {**

**// Step definition logic**

**System.out.println("Executing Then step...");**

**}**

**}**

### Step 3: Write Feature Files

Write your feature files with scenarios and corresponding steps.

**Feature: Example Feature**

**Scenario: Example Scenario**

**Given I have a precondition**

**When I perform some action**

**Then I should see the expected result**

**Explanation:**

* The feature file contains scenarios with Given-When-Then steps.
* Each step is associated with a step definition method in the Step Definitions class.

**Step 4: Execute Your Tests**

Run your Cucumber tests. You should see the hooks executing before and after each scenario.

**Explanation:**

* Before each scenario execution, the **setUp** hook method will run, setting up preconditions.
* After each scenario execution, the **tearDown** hook method will run, cleaning up resources.

**TaggedHooks**

In Cucumber, you can use tagged hooks to apply setup and teardown logic selectively based on scenario tags. This allows you to customize actions before or after scenarios with specific tags. Here's how you can implement tagged hooks:

### Step 1: Implement Tagged Hooks in Your Cucumber Project

#### **Step Definitions:**

Create a class containing hook methods annotated with Cucumber annotations (**@Before** and **@After**) and specify the tags to apply them to.

**import io.cucumber.java.After;**

**import io.cucumber.java.Before;**

**public class TaggedHooks {**

**@Before("@setup")**

**public void setUp() {**

**// Code to set up preconditions before scenarios with @setup tag**

**System.out.println("Setting up before scenario with @setup tag...");**

**}**

**@After("@cleanup")**

**public void tearDown() {**

**// Code to clean up after scenarios with @cleanup tag**

**System.out.println("Tearing down after scenario with @cleanup tag...");**

**}**

**}**

**Explanation:**

* The **@Before("@setup")** annotation indicates that the **setUp** method should be executed before each scenario with the **@setup** tag.
* The **@After("@cleanup")** annotation indicates that the **tearDown** method should be executed after each scenario with the **@cleanup** tag.
* You can have multiple tagged hooks in your project.

**Step 2: Configure Step Definitions**

Ensure that your step definitions class is configured to use the tagged hooks.

**import io.cucumber.java.en.Given;**

**import io.cucumber.java.en.Then;**

**import io.cucumber.java.en.When;**

**public class StepDefinitions {**

**@Given("I have a precondition")**

**public void i\_have\_a\_precondition() {**

**// Step definition logic**

**System.out.println("Executing Given step...");**

**}**

**@When("I perform some action")**

**public void i\_perform\_some\_action() {**

**// Step definition logic**

**System.out.println("Executing When step...");**

**}**

**@Then("I should see the expected result")**

**public void i\_should\_see\_the\_expected\_result() {**

**// Step definition logic**

**System.out.println("Executing Then step...");**

**}**

**}**

### Step 3: Write Feature Files

Write your feature files with scenarios and corresponding steps, applying tags where necessary.

**Feature: Example Feature**

**@setup**

**Scenario: Example Scenario with Setup**

**Given I have a precondition**

**When I perform some action**

**Then I should see the expected result**

**@cleanup**

**Scenario: Example Scenario with Cleanup**

**Given I have a precondition**

**When I perform some action**

**Then I should see the expected result**

**Explanation:**

* Each scenario is tagged with either **@setup** or **@cleanup**.
* The tagged hooks **setUp** and **tearDown** will run before and after scenarios with the respective tags.

**Step 4: Execute Your Tests**

Run your Cucumber tests. You should see the tagged hooks executing before and after scenarios with the corresponding tags.

**Explanation:**

* Before scenarios tagged with **@setup**, the **setUp** hook method will run, setting up preconditions.
* After scenarios tagged with **@cleanup**, the **tearDown** hook method will run, cleaning up resources.

**Ordering Hooks**

Ordering hooks in Cucumber allows you to specify the order in which hooks should execute before or after scenarios. This is useful when you have multiple hooks and need to ensure they run in a specific sequence. Here's how you can implement ordering hooks:

### Step 1: Implement Ordering Hooks in Your Cucumber Project

#### **Step Definitions:**

Create a class containing hook methods annotated with Cucumber annotations (**@Before** and **@After**) and specify the order using the **order** attribute.

**import io.cucumber.java.After;**

**import io.cucumber.java.Before;**

**public class OrderingHooks {**

**@Before(order = 1)**

**public void setUpFirst() {**

**// Code to set up preconditions before scenarios**

**System.out.println("Setting up first...");**

**}**

**@Before(order = 2)**

**public void setUpSecond() {**

**// Additional setup before scenarios**

**System.out.println("Setting up second...");**

**}**

**@After(order = 1)**

**public void tearDownFirst() {**

**// Code to clean up after scenarios**

**System.out.println("Tearing down first...");**

**}**

**@After(order = 2)**

**public void tearDownSecond() {**

**// Additional cleanup after scenarios**

**System.out.println("Tearing down second...");**

**}**

**}**

**Explanation:**

* The **@Before(order = 1)** annotation specifies that the **setUpFirst** method should be executed before each scenario with the lowest priority.
* The **@Before(order = 2)** annotation specifies that the **setUpSecond** method should be executed before each scenario with the second-lowest priority.
* Similarly, **@After(order = 1)** and **@After(order = 2)** annotations specify the execution order for teardown methods after each scenario.

### Step 2: Configure Step Definitions

Ensure that your step definitions class is configured to use the ordering hooks.

**import io.cucumber.java.en.Given;**

**import io.cucumber.java.en.Then;**

**import io.cucumber.java.en.When;**

**public class StepDefinitions {**

**@Given("I have a precondition")**

**public void i\_have\_a\_precondition() {**

**// Step definition logic**

**System.out.println("Executing Given step...");**

**}**

**@When("I perform some action")**

**public void i\_perform\_some\_action() {**

**// Step definition logic**

**System.out.println("Executing When step...");**

**}**

**@Then("I should see the expected result")**

**public void i\_should\_see\_the\_expected\_result() {**

**// Step definition logic**

**System.out.println("Executing Then step...");**

**}**

**}**

### Step 3: Write Feature Files

Write your feature files with scenarios and corresponding steps.

**Feature: Example Feature**

**Scenario: Example Scenario**

**Given I have a precondition**

**When I perform some action**

**Then I should see the expected result**

**Explanation:**

* Each scenario runs with the configured ordering hooks before and after it.
* The hooks execute in the specified order, ensuring proper setup and teardown for each scenario.

**Step 4: Execute Your Tests**

Run your Cucumber tests. You should see the ordering hooks executing in the specified order before and after each scenario.

**Explanation:**

* Before each scenario execution, the setup hooks (**setUpFirst** and **setUpSecond**) will run in the specified order.
* After each scenario execution, the teardown hooks (**tearDownFirst** and **tearDownSecond**) will run in the specified order.

**Ordering Tagged Hooks**

Ordering tagged hooks in Cucumber allows you to specify the order in which hooks should execute before or after scenarios with specific tags. This is useful when you have multiple hooks and need to ensure they run in a specific sequence for scenarios with particular tags. Here's how you can implement ordering tagged hooks:

### Step 1: Implement Ordering Tagged Hooks in Your Cucumber Project

#### **Step Definitions:**

Create a class containing hook methods annotated with Cucumber annotations (**@Before** and **@After**) and specify the order using the **order** attribute, along with tags to apply them to.

**import io.cucumber.java.After;**

**import io.cucumber.java.Before;**

**public class OrderingTaggedHooks {**

**@Before(order = 1, value = "@setup")**

**public void setUpFirst() {**

**// Code to set up preconditions before scenarios with @setup tag**

**System.out.println("Setting up first for @setup...");**

**}**

**@Before(order = 2, value = "@setup")**

**public void setUpSecond() {**

**// Additional setup before scenarios with @setup tag**

**System.out.println("Setting up second for @setup...");**

**}**

**@After(order = 1, value = "@cleanup")**

**public void tearDownFirst() {**

**// Code to clean up after scenarios with @cleanup tag**

**System.out.println("Tearing down first for @cleanup...");**

**}**

**@After(order = 2, value = "@cleanup")**

**public void tearDownSecond() {**

**// Additional cleanup after scenarios with @cleanup tag**

**System.out.println("Tearing down second for @cleanup...");**

**}**

**}**

**Explanation:**

* The **@Before(order = 1, value = "@setup")** annotation specifies that the **setUpFirst** method should be executed before each scenario with the **@setup** tag, with the lowest priority.
* The **@Before(order = 2, value = "@setup")** annotation specifies that the **setUpSecond** method should be executed before each scenario with the **@setup** tag, with the second-lowest priority.
* Similarly, **@After(order = 1, value = "@cleanup")** and **@After(order = 2, value = "@cleanup")** annotations specify the execution order for teardown methods after each scenario with the **@cleanup** tag.

### Step 2: Configure Step Definitions

Ensure that your step definitions class is configured to use the ordering tagged hooks.

**import io.cucumber.java.en.Given;**

**import io.cucumber.java.en.Then;**

**import io.cucumber.java.en.When;**

**public class StepDefinitions {**

**@Given("I have a precondition")**

**public void i\_have\_a\_precondition() {**

**// Step definition logic**

**System.out.println("Executing Given step...");**

**}**

**@When("I perform some action")**

**public void i\_perform\_some\_action() {**

**// Step definition logic**

**System.out.println("Executing When step...");**

**}**

**@Then("I should see the expected result")**

**public void i\_should\_see\_the\_expected\_result() {**

**// Step definition logic**

**System.out.println("Executing Then step...");**

**}**

**}**

### Step 3: Write Feature Files

Write your feature files with scenarios and corresponding steps, applying tags where necessary.

**Feature: Example Feature**

**@setup**

**Scenario: Example Scenario with Setup**

**Given I have a precondition**

**When I perform some action**

**Then I should see the expected result**

**@cleanup**

**Scenario: Example Scenario with Cleanup**

**Given I have a precondition**

**When I perform some action**

**Then I should see the expected result**

Ordering tagged hooks in Cucumber allows you to specify the order in which hooks should execute before or after scenarios with specific tags. This is useful when you have multiple hooks and need to ensure they run in a specific sequence for scenarios with particular tags. Here's how you can implement ordering tagged hooks:

### Step 1: Implement Ordering Tagged Hooks in Your Cucumber Project

#### **Step Definitions:**

Create a class containing hook methods annotated with Cucumber annotations (**@Before** and **@After**) and specify the order using the **order** attribute, along with tags to apply them to.

javaCopy code

import io.cucumber.java.After; import io.cucumber.java.Before; public class OrderingTaggedHooks { @Before(order = 1, value = "@setup") public void setUpFirst() { // Code to set up preconditions before scenarios with @setup tag System.out.println("Setting up first for @setup..."); } @Before(order = 2, value = "@setup") public void setUpSecond() { // Additional setup before scenarios with @setup tag System.out.println("Setting up second for @setup..."); } @After(order = 1, value = "@cleanup") public void tearDownFirst() { // Code to clean up after scenarios with @cleanup tag System.out.println("Tearing down first for @cleanup..."); } @After(order = 2, value = "@cleanup") public void tearDownSecond() { // Additional cleanup after scenarios with @cleanup tag System.out.println("Tearing down second for @cleanup..."); } }

### Explanation:

* The **@Before(order = 1, value = "@setup")** annotation specifies that the **setUpFirst** method should be executed before each scenario with the **@setup** tag, with the lowest priority.
* The **@Before(order = 2, value = "@setup")** annotation specifies that the **setUpSecond** method should be executed before each scenario with the **@setup** tag, with the second-lowest priority.
* Similarly, **@After(order = 1, value = "@cleanup")** and **@After(order = 2, value = "@cleanup")** annotations specify the execution order for teardown methods after each scenario with the **@cleanup** tag.

### Step 2: Configure Step Definitions

Ensure that your step definitions class is configured to use the ordering tagged hooks.

javaCopy code

import io.cucumber.java.en.Given; import io.cucumber.java.en.Then; import io.cucumber.java.en.When; public class StepDefinitions { @Given("I have a precondition") public void i\_have\_a\_precondition() { // Step definition logic System.out.println("Executing Given step..."); } @When("I perform some action") public void i\_perform\_some\_action() { // Step definition logic System.out.println("Executing When step..."); } @Then("I should see the expected result") public void i\_should\_see\_the\_expected\_result() { // Step definition logic System.out.println("Executing Then step..."); } }

### Step 3: Write Feature Files

Write your feature files with scenarios and corresponding steps, applying tags where necessary.

gherkinCopy code

Feature: Example Feature @setup Scenario: Example Scenario with Setup Given I have a precondition When I perform some action Then I should see the expected result @cleanup Scenario: Example Scenario with Cleanup Given I have a precondition When I perform some action Then I should see the expected result

### Explanation:

* Each scenario runs with the configured ordering tagged hooks before and after it.
* The hooks execute in the specified order, ensuring proper setup and teardown for each scenario with the respective tags.

### Step 4: Execute Your Tests

Run your Cucumber tests. You should see the ordering tagged hooks executing in the specified order before and after each scenario with the corresponding tags.

### Explanation:

* Before each scenario execution with the **@setup** tag, the setup hooks (**setUpFirst** and **setUpSecond**) will run in the specified order.
* After each scenario execution with the **@cleanup** tag, the cleanup hooks (**tearDownFirst** and **tearDownSecond**) will run in the specified order.

**Background**

The Background keyword in Cucumber allows you to define steps that should be executed before each scenario in a feature file. This is useful when you have common setup steps that are shared across multiple scenarios within a feature. Here's how you can use the Background keyword:

### Step 1: Write a Feature File with a Background Section

Create a feature file with the Background keyword and the common setup steps.

#### **Example Feature File:**

**Feature: Login Functionality**

**Background:**

**Given I navigate to the login page**

**And I enter my username**

**And I enter my password**

**When I click on the login button**

**Scenario: Successful Login**

**Then I should be logged in**

**Scenario: Logout**

**Given I am logged in**

**When I click on the logout button**

**Then I should be logged out**

**Explanation:**

* The Background section contains common setup steps (**Given**, **And**, **When**) that should be executed before each scenario in the feature file.
* These steps will be executed once before every scenario in the feature, providing a consistent starting point for each scenario.

### Step 2: Write Step Definitions

Write step definitions for the steps defined in the Background section.

#### **Step Definitions:**

**import io.cucumber.java.en.Given;**

**import io.cucumber.java.en.When;**

**import io.cucumber.java.en.Then;**

**public class StepDefinitions {**

**@Given("I navigate to the login page")**

**public void navigateToLoginPage() {**

**// Navigate to the login page**

**}**

**@Given("I enter my username")**

**public void enterUsername() {**

**// Enter username**

**}**

**@Given("I enter my password")**

**public void enterPassword() {**

**// Enter password**

**}**

**@When("I click on the login button")**

**public void clickLoginButton() {**

**// Click on the login button**

**}**

**@Then("I should be logged in")**

**public void verifyLoggedIn() {**

**// Verify user is logged in**

**}**

**@Given("I am logged in")**

**public void userIsLoggedIn() {**

**// Steps to ensure user is logged in**

**}**

**@When("I click on the logout button")**

**public void clickLogoutButton() {**

**// Click on the logout button**

**}**

**@Then("I should be logged out")**

**public void verifyLoggedOut() {**

**// Verify user is logged out**

**}**

**}**

**Explanation:**

* Define step definitions for each step in the Background section and for the scenarios in the feature file.
* These step definitions will be executed when Cucumber matches the steps in the feature file with the corresponding step definitions.

**Step 3: Execute Your Tests**

Run your Cucumber tests. You should see the Background steps executed before each scenario in the feature file.

**Explanation:**

* Before each scenario execution, the steps defined in the Background section will be executed, followed by the steps specific to each scenario.
* This ensures that the common setup steps are performed before each scenario, providing a consistent test environment.