



Cucumber

Course Content

- What is Behaviour Driven Development
- Introduction to Cucumber
- Cucumber Installation and Setup
- Writing Feature Files in Cucumber
- Step Definitions in Cucumber
- Running Tests in Cucumber
- Cucumber Hooks
- Cucumber Reporting
- Cucumber Integration with Testing Tools
- Cucumber Best Practices & Tips

What is Behaviour-Driven Development (BDD)

- Behaviour-Driven Development (BDD) is a software development approach that encourages collaboration between developers, testers, and business stakeholders.
- It focuses on understanding the behavior of an application from the user's perspective before writing the code.
- BDD enhances communication and ensures that development meets business goals.

Core Principles of BDD

- **Collaboration:** Involves developers, testers, and non-technical stakeholders in the discussion.
- **Common Language:** Uses natural language (Gherkin) to write test scenarios that are easy to understand.
- **Shared Understanding:** Creates a shared understanding of system behavior and requirements.
- **Test-First Approach:** Tests are written in the form of scenarios before development begins.

Why Use BDD?

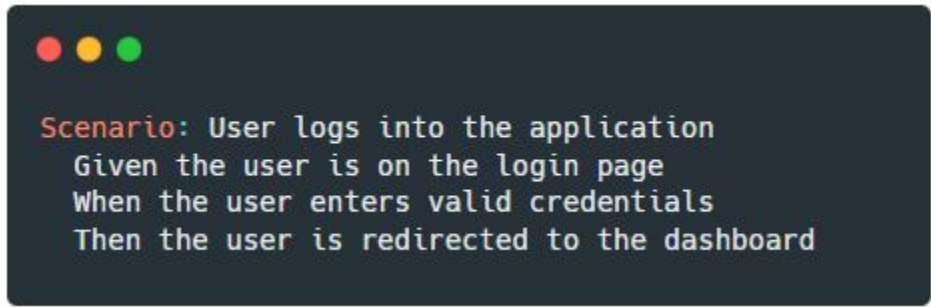
- **Bridges the Gap:** BDD bridges the gap between business requirements and technical implementation.
- **Improves Communication:** Ensures all team members (business analysts, developers, testers) have a common understanding of the expected behavior.
- **Reduces Ambiguity:** Written scenarios clarify requirements, reducing misunderstandings and misinterpretation.
- **Living Documentation:** BDD scenarios act as documentation that stays up to date with the system's behavior.

Gherkin Syntax

Gherkin is the language used to write BDD scenarios. It uses a structured format, making it easy to understand by all stakeholders.

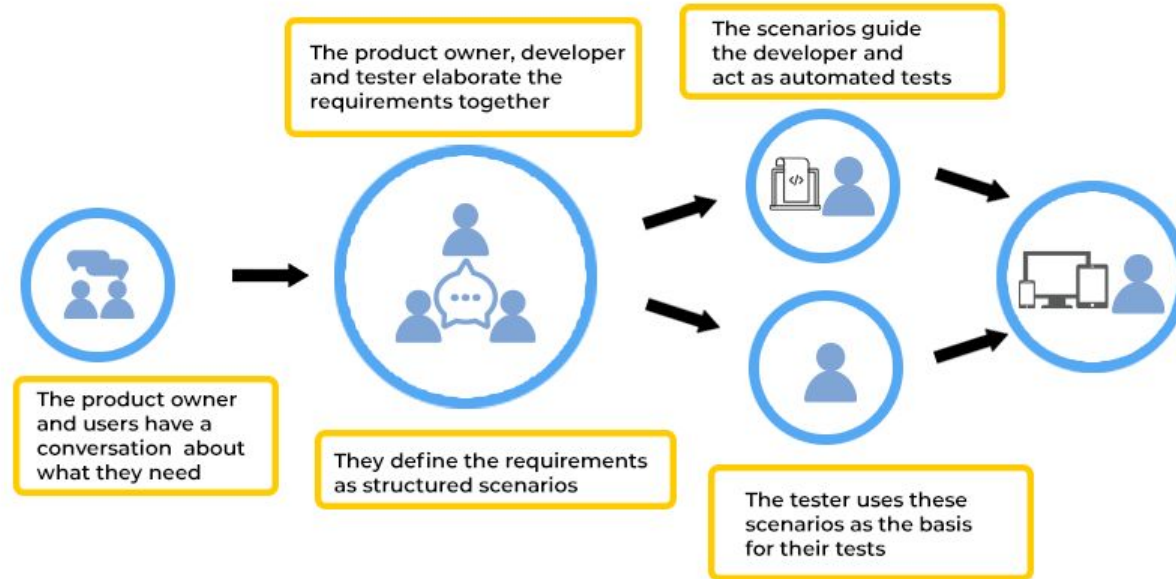
Example:

- **Given:** Precondition or setup.
- **When:** Action or event that triggers behavior.
- **Then:** Expected outcome or result.



```
Scenario: User logs into the application
  Given the user is on the login page
  When the user enters valid credentials
  Then the user is redirected to the dashboard
```

BDD DEVELOPMENT PROCESS



TDD vs. BDD: A Simple Comparison

Test-Driven Development (TDD):

- TDD is a development technique where you write tests **before** writing the actual code.
- The process involves writing a failing test, writing code to pass the test, and then refactoring.
- It ensures that the code meets the requirements and helps in identifying defects early.

Behavior-Driven Development (BDD):

- BDD is an extension of TDD where tests are written in natural, human-readable language.
- Focuses on the **behavior** of an application rather than just verifying functionality.
- BDD encourages collaboration between developers, testers, and non-technical stakeholders by using a shared language.

Key Differences Between TDD and BDD

Aspect	TDD	BDD
Focus	Testing code functionality	Testing application behavior
Audience	Developers	Developers, testers, business analysts
Language	Code-based tests (unit tests)	Natural language (Gherkin syntax)
Collaboration	Less focus on collaboration	Strong collaboration between teams

Advantages of BDD Over TDD

- **Enhanced Collaboration:** BDD promotes communication between technical and non-technical teams, ensuring that everyone understands the requirements.
- **Readable Tests:** BDD uses plain language to describe the behavior, making the tests easy to understand for all stakeholders.
- **Clearer Requirements:** With BDD, scenarios are written based on user stories, ensuring that the development aligns with business goals.
- **Living Documentation:** BDD scenarios serve as documentation for how the system is expected to behave, reducing the need for additional documentation.

What is Cucumber?

- Cucumber is a **testing framework** that supports Behavior-Driven Development (BDD).
- It allows writing tests in **plain, natural language** (usually English) which can be understood by non-technical stakeholders.
- Cucumber supports a wide range of programming languages like Java, Ruby, JavaScript, etc.

Key Features of Cucumber

- **Gherkin Language:** Cucumber uses Gherkin, a structured language for defining test scenarios in plain text.
- **Collaboration Tool:** Cucumber facilitates collaboration between developers, testers, and business stakeholders.
- **Cross-language Compatibility:** It can be used with multiple programming languages.
- **Executable Specifications:** The scenarios written in Gherkin are turned into executable tests.

Basic Structure of a Cucumber Test

1. **Feature File:**

- Contains the Gherkin language, defining the feature and scenarios.

2. **Step Definitions:**

- The actual code that maps the Gherkin steps to code functions.

3. **Runner Class:**

- Executes the Cucumber tests and connects the feature files to the step definitions.

How Cucumber Works?

- **Step 1:** Write the behavior of the feature in a plain text Gherkin file.
- **Step 2:** Create the step definitions that implement the behavior using code.
- **Step 3:** Execute the tests using the Cucumber runner.

The image displays a 2x3 grid of screenshots illustrating the Cucumber workflow:

- 1: Describe behaviour in plain text**
A Gherkin file snippet:

```
Feature: Addition
  In order to avoid silly mistakes
  As a math idiot
  I want to be told the sum of two numbers

  Scenario: Add two numbers
    Given I have entered 50 into the calculator
    And I have entered 70 into the calculator
    When I press add
    Then the result should be 120 on the screen
```
- 2: Write a step definition in Ruby**
A Ruby step definition snippet:

```
Given /I have entered (.*) into the calculator/ do |n|
  calculator = Calculator.new
  calculator.push(n.to_i)
end
```
- 3: Run and watch it fail**
Terminal output showing a failure:

```
$ cucumber features/addition.feature
Feature: Addition # Features/addition.feature
  In order to avoid silly mistakes
  As a math idiot
  I want to be told the sum of two numbers # Features/addition.feature/step_1
  Scenario: Add two numbers # Features/addition.feature/step_1
    Given I have entered 50 into the calculator # Features/addition.feature/step_1
    uninitialized constant Calculator (NameError)
    /features/step_definitions/calculator_steps.rb:2:in "Given I have entered 50 into the calculator"
    And I have entered 70 into the calculator # Features/addition.feature/step_1
    When I press add # Features/addition.feature/step_1
    Then the result should be 120 on the screen # Features/addition.feature/step_1
```
- 4. Write code to make the step pass**
A Ruby class snippet:

```
class Calculator
  def push(n)
    @args ||= []
    @args << n
  end
end
```
- 5. Run again and see the step pass**
Terminal output showing a passing test:

```
$ cucumber features/addition.feature
Feature: Addition # Features/addition.feature
  In order to avoid silly mistakes
  As a math idiot
  I want to be told the sum of two numbers # Features/addition.feature/step_1
  Scenario: Add two numbers # Features/addition.feature/step_1
    Given I have entered 50 into the calculator # Features/addition.feature/step_1
    And I have entered 70 into the calculator # Features/addition.feature/step_1
    When I press add # Features/addition.feature/step_1
    Then the result should be 120 on the screen # Features/addition.feature/step_1
```
- 6. Repeat 2-5 until green like a cucumber**
Terminal output showing a passing test (repeated):

```
$ cucumber features/addition.feature
Feature: Addition # Features/addition.feature
  In order to avoid silly mistakes
  As a math idiot
  I want to be told the sum of two numbers # Features/addition.feature/step_1
  Scenario: Add two numbers # Features/addition.feature/step_1
    Given I have entered 50 into the calculator # Features/addition.feature/step_1
    And I have entered 70 into the calculator # Features/addition.feature/step_1
    When I press add # Features/addition.feature/step_1
    Then the result should be 120 on the screen # Features/addition.feature/step_1
```

Source: <https://www.slideshare.net/slideshow/bdd-and-automation-testing-using-cucumber-framework-webinar/42554672>

Prerequisites:

1. Java should be installed in the system

```
java -version
```

```
openjdk version "11.0.19" 2023-04-18  
OpenJDK Runtime Environment Temurin-11.0.19+7 (build 11.0.19+7)  
OpenJDK 64-Bit Server VM Temurin-11.0.19+7 (build 11.0.19+7, mixed mode)
```

2. Maven should be installed in the system

```
mvn -version
```

```
Apache Maven 3.8.4 (9b656c72d54e5baced989b64718c159fe39b537)  
Maven home: ~/softwares/apache-maven-3.8.4
```

Installing/Using Cucumber:

1. Add below maven dependency to pom.xml file

```
<!-- Cucumber Java Dependency -->
<dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-java</artifactId>
  <version>7.11.0</version>
  <scope>test</scope>
</dependency>

<!-- Cucumber JUnit Dependency -->
<dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-junit</artifactId>
  <version>7.11.0</version>
  <scope>test</scope>
</dependency>
```


Installing/Using Cucumber:

2. Add Cucumber Plugins specific to your IDE

IntelliJ IDEA

- <https://plugins.jetbrains.com/plugin/7212-cucumber-for-java>
- <https://plugins.jetbrains.com/plugin/16289-cucumber>

Eclipse

- <https://marketplace.eclipse.org/content/cucumber-eclipse-plugin>

Writing Feature Files in Cucumber

- A **Feature File** is where BDD scenarios are written using the **Gherkin** language.
- It defines the **feature** of the system being tested and the various **scenarios** that describe specific behaviors of that feature.
- Feature files are plain text files with a **.feature** extension.

Basic Structure of a Feature File:

A feature file consists of the following parts:

1. **Feature:** Describes the feature under test.
2. **Scenario:** Defines a specific test case within the feature.
3. **Steps:** Written in **Given-When-Then** format, describing the flow of the scenario.

Gherkin Syntax and Keywords

In Gherkin, we use specific keywords to define steps:

- **Feature:** Describes a specific functionality or feature of the application.
- **Scenario:** Outlines a single test case that demonstrates a particular behavior or outcome.
- **Given:** Sets up the initial context or state of the system before the action occurs.
- **When:** Describes the action or event that triggers the behavior being tested.
- **Then:** Specifies the expected outcome or result after the action is performed.
- **And:** Used to add additional conditions or steps that follow logically from the previous line.
- **But:** Introduces a contrasting condition or exception to the previous statement.
- **Background:** Provides a set of steps that are common to all scenarios in a feature.
- **Scenario Outline:** Allows for parameterization of scenarios, enabling multiple examples to be run with different data.
- **Examples:** Lists the variable data that will be used with a Scenario Outline.

Writing Scenarios with Given-When-Then

Each scenario uses the **Given-When-Then** structure to ensure clarity and organization.

- **Given:** The initial state of the system.
 - *Example:* "Given the user is logged in"
- **When:** The action the user takes.
 - *Example:* "When the user clicks the logout button"
- **Then:** The expected result.
 - *Example:* "Then the user should see the login page"

Step Definitions

- Step definitions are the **glue** between the Gherkin feature files and the actual test code.
- Each Gherkin step in the feature file is mapped to a corresponding method in the step definition class.
- These methods contain the code that executes the action described in the step.

How Step-Definitions work?

- In Cucumber, each step in a feature file is defined with a **regular expression** in the step definition.
- When Cucumber executes the feature file, it looks for matching step definitions based on the text of each step.

Step Definitions



```
# Login.feature
```

```
Given the user is on the login page
```



```
// LoginSteps.java
```

```
@Given("the user is on the login page")  
public void userIsOnLoginPage() {  
    // Code to navigate to the login page  
}
```

Annotations for Step Definitions

Cucumber uses specific annotations to match the Gherkin steps:

- **@Given**: Maps to the "Given" step in the feature file.
- **@When**: Maps to the "When" step.
- **@Then**: Maps to the "Then" step.
- **@And** / **@But**: Can be used to extend steps.

Common Mistakes in Step Definitions

- **Mismatched Step Text:** Ensure that the step in the feature file exactly matches the step in the definition.
- **Undefined Steps:** If a step is not defined, Cucumber will report an error. Always define all steps.
- **Overcomplicating Definitions:** Break complex steps into smaller, reusable methods for clarity and simplicity.

Test Runner in Cucumber

- A **Test Runner** is a class in Cucumber that **executes the feature files** and connects them to step definitions.
- It allows you to configure options such as feature file location, step definition location, and reporting format.
- The test runner uses the **JUnit** or **TestNG** framework to execute the tests.

Basic Structure of a Test Runner

How a Test Runner Works?

- The Test Runner class is typically annotated with **@RunWith** and **@CucumberOptions**.
- **@RunWith(Cucumber.class)** tells **JUnit** to run the test using the Cucumber framework.
- **@CucumberOptions** allows you to configure the test execution.

Basic Structure of a Test Runner - Example

```
import org.junit.runner.RunWith;
import io.cucumber.junit.Cucumber;
import io.cucumber.junit.CucumberOptions;

@RunWith(Cucumber.class)
@CucumberOptions({
    features = "src/test/resources/features",    // Path to feature files
    glue = "stepDefinitions",                  // Path to step definitions
    plugin = {"pretty", "html:target/cucumber-reports.html"} // Reports configuration
})
public class TestRunner {
}
```

Explanation:

- **features:** Path to your feature files.
- **glue:** Path to your step definitions (glue code).
- **plugin:** Specifies how the output of the test should be generated (e.g., pretty console output, HTML reports).

Key Options in @CucumberOptions

Important Configuration Options

1. **features**: Location of the feature files.
2. **glue**: Location of the step definitions (glue code).
3. **tags**: Used to specify which scenarios to run based on tags in the feature files.
4. **plugin**: Specifies the type of report (e.g., **pretty**, **html**, **json**).
5. **dryRun**: When set to true, checks if all steps have step definitions without actually executing the tests.
6. **strict**: If set to true, it fails the execution if any undefined or pending steps are found.

Advanced Example of Test Runner


```
@RunWith(Cucumber.class)
@cucumberOptions(
    features = "src/test/resources/features",
    glue = {"stepDefinitions"},
    tags = "@smoke", // Run only scenarios tagged as @smoke
    plugin = {"pretty", "html:target/cucumber-reports", "json:target/cucumber.json"},
    monochrome = true, // Makes console output more readable
    dryRun = false, // Set to true to check for undefined steps without running tests
    strict = true // Fail the test if undefined steps exist
)
public class TestRunner {
}
```

Explanation:

- **tags:** Runs only scenarios tagged with **@smoke**.
- **monochrome:** Clean and readable console output.
- **dryRun:** Ensures all steps have definitions before running.
- **strict:** Fails if there are undefined or unimplemented steps.

Using Tags in Test Runner

- Tags help you group scenarios and execute only the scenarios with specific tags.
- In the test runner, you can specify tags using the `@CucumberOptions` annotation.



```
Feature: User Login

@smoke
Scenario: Successful login
  Given the user is on the login page
  When the user enters valid credentials
  Then the user should be redirected to the dashboard
```

In the Test Runner:



```
@CucumberOptions(tags = "@smoke")
```

Monochrome and DryRun Options

Improving Test Readability and Execution:

- **monochrome**: Makes console output more readable by removing unnecessary characters.
- **dryRun**: Useful for **checking if all steps have corresponding definitions** without actually executing the tests.

```
@CucumberOptions(  
    monochrome = true,    // Improves console output readability  
    dryRun = true        // Checks step definitions without running tests  
)
```

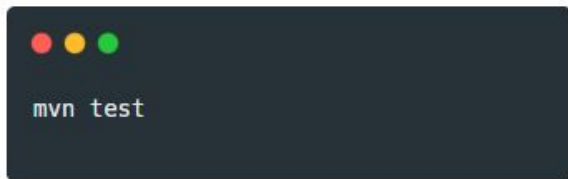
Running Cucumber Tests

1. Executing Tests from Your IDE

- After setting up the **Test Runner**, you can execute the tests directly from your IDE (IntelliJ IDEA, Eclipse, etc.).
- Right-click on the Test Runner class and choose **Run** to execute the Cucumber tests.

2. Using Maven for Command Line Execution

- You can run Cucumber tests from the command line using **Maven**.
- Navigate to your project folder and use the following command:

A dark-themed terminal window with three colored window control buttons (red, yellow, green) in the top-left corner. The text `mvn test` is displayed in the terminal.

```
mvn test
```


Best Practices for Test Runners

1. **Use Tags Efficiently:** Organize and run tests using tags like **@smoke**, **@regression**, etc.
2. **Generate Reports:** Always generate detailed reports (HTML, JSON) for test visibility and debugging.
3. **Dry Run Before Execution:** Use **dryRun** to ensure all steps are implemented before running the tests.
4. **Avoid Duplicates:** Ensure feature file paths and glue paths are correctly set to avoid running the same tests multiple times.

Parameterization in Cucumber

Introduction to Parameterization:

- Parameterizing tests in Cucumber allows you to **pass dynamic data** into your test steps.
- It helps in **reducing repetition** and making your tests more flexible and reusable.
- You can **run the same test scenario** with different sets of input data.

Approaches to Parameterization:

1. Using Step Definition Parameters.
2. Scenario Outline with Examples Table.

Using Step Definition Parameters

Parameterizing Directly in Steps:

- You can pass parameters directly into a Gherkin step by using quotes (") or curly braces ({}).
- The values from the step are passed into the corresponding step definition method.

Example Feature File:

```
Scenario: Login with different credentials
  Given the user enters "admin" as the username and "admin123" as the password
  When the user clicks the login button
  Then the user should be redirected to the dashboard
```

Step Definition:

```
@Given("the user enters {string} as the username and {string} as the password")
public void userEntersCredentials(String username, String password) {
    // Code to input the username and password
}
```

Data Types in Step Definitions

Parameterizing with Different Data Types:

Cucumber can handle different types of data like:

1. **String:** Use `{string}`.
2. **Integer:** Use `{int}`.
3. **Double:** Use `{float}` or `{double}`.

Data Types in Step Definitions - Example

Example:

```
Scenario: Applying discount on a product
  Given the product price is 100
  When a discount of 10% is applied
  Then the final price should be 90
```

Step Definition:

```
@Given("the product price is {int}")
public void setProductPrice(int price) {
    // Code to set product price
}

@When("a discount of {double} is applied")
public void applyDiscount(double discount) {
    // Code to apply discount
}

@Then("the final price should be {int}")
public void verifyFinalPrice(int finalPrice) {
    // Code to verify final price
}
```

Scenario Outline with Examples Table

Using Scenario Outline for Multiple Data Sets:

- **Scenario Outline** allows you to **define a scenario template** and run it with different sets of data.
- Data is provided through the **Examples** table.

Scenario Outline: Login with multiple credentials

Given the user enters "<username>" as the username and "<password>" as the password

When the user clicks the login button

Then the user should be redirected to the "<page>"

Examples:

username	password	page
admin	admin123	dashboard
guest	guest123	guest home
invalid	wrong	error page

Step Definition for Scenario Outline

Handling Scenario Outline Parameters:

- Placeholders in the **Scenario Outline** are replaced with actual values from the **Examples** table during execution.
- The same step definition can be reused for each set of data.

```
@Given("the user enters {string} as the username and {string} as the password")
public void userEntersCredentials(String username, String password) {
    // Code to enter username and password
}

@Then("the user should be redirected to the {string}")
public void verifyRedirectPage(String page) {
    // Code to verify the user is redirected to the correct page
}
```

Benefits of Using Scenario Outline

- **Reusable Tests:** Define a single scenario that works for multiple data sets.
- **Less Code Duplication:** Avoid writing multiple scenarios for similar tests.
- **Clear Data Representation:** The Examples table clearly outlines all test cases and data.

Example: Data-Driven Testing for Shopping Cart

Feature File:

```
Scenario Outline: Adding different items to the cart
  Given the user adds "<item>" to the cart
  When the user proceeds to checkout
  Then the cart should contain "<item>" and the total should be "<price>"
```

Examples:

item	price
Laptop	1000
Headphones	100
Smartphone	800

Step Definition:

```
@Given("the user adds {string} to the cart")
public void addItemToCart(String item) {
    // Code to add item to the cart
}

@Then("the cart should contain {string} and the total should be {int}")
public void verifyCartContent(String item, int price) {
    // Code to verify cart contents and price
}
```

Advanced Parameterization with Data Tables

Using Data Tables for More Complex Inputs:

- For more complex data sets or inputs, you can use **Data Tables**.
- A data table allows you to pass multiple rows and columns of data to a step.

Example:

```
Scenario: Multiple items added to the cart
  Given the user adds the following items to the cart:
    | item      | price |
    | Laptop    | 1000  |
    | Headphones | 100   |
    | Mouse     | 50    |
  Then the total price should be 1150
```

Step Definition:

```
@Given("the user adds the following items to the cart:")
public void addItemToCart(DataTable dataTable) {
    List<Map<String, String>> items = dataTable.asMaps(String.class, String.class);
    for (Map<String, String> item : items) {
        // Code to add each item to the cart
        String itemName = item.get("item");
        String price = item.get("price");
    }
}
```

Best Practices for Parameterizing Tests

1. **Keep Parameters Flexible:**

- a. Use parameterization to make your tests more versatile and less repetitive.

2. **Use Scenario Outline for Data-Driven Tests:**

- a. When you have multiple sets of inputs, prefer Scenario Outline.

3. **Avoid Overloading Steps:**

- a. Use parameterization to keep your steps simple and focused.

Common Mistakes in Parameterization

1. **Mismatch Between Steps and Definitions:**

- a. Ensure the parameters in the step match the parameters in the step definition.

2. **Over-Parameterization:**

- a. Don't overuse parameters for unrelated steps, as it can make tests difficult to read.

3. **Unnecessary Data in Examples Table:**

- a. Only provide relevant data in the Examples table to keep the test clean and easy to follow.

Cucumber Hooks

- **Cucumber Hooks** allow you to **run blocks of code before or after** each scenario or step.
- They are similar to **setup** and **teardown** methods in testing frameworks like JUnit or TestNG.
- Hooks are mainly used for **initializing test prerequisites** (e.g., starting a browser) or **cleaning up** after a test (e.g., closing a browser).

Types of Cucumber Hooks

Two Types of Hooks in Cucumber

1. **@Before**: Runs **before** each scenario or step.
2. **@After**: Runs **after** each scenario or step.

```
@Before
public void setup() {
    // Code to set up preconditions (e.g., open browser, initialize objects)
}

@After
public void tearDown() {
    // Code to clean up after test (e.g., close browser, reset state)
}
```

Use Cases for Hooks

When to Use Hooks?

- **@Before:**
 - Starting a web browser.
 - Loading initial test data.
 - Setting up database connections.
- **@After:**
 - Closing the browser.
 - Cleaning up temporary test data.
 - Logging out of the application.

Example: Using @Before and @After Hooks

```
@Before
public void setUp() {
    System.out.println("Opening browser and navigating to the homepage");
    // Code to launch the browser and navigate to the homepage
}

@After
public void tearDown() {
    System.out.println("Closing the browser");
    // Code to close the browser and clean up
}
```

Explanation:

- The **@Before** hook launches the browser before each scenario.
- The **@After** hook closes the browser after each scenario.

Order of Hook Execution

Controlling Hook Execution Order:

- By default, hooks are executed in the order they are defined.
- You can **control the execution order** of hooks using the **order** parameter.
- Hooks with **lower order values** run first, and hooks with **higher values** run later.

Explanation:

- Hooks with **@Before(order = 1)** will run before
@Before(order = 2).
- Similarly, **@After(order = 1)** will run before
@After(order = 2).

```
@Before(order = 1)
public void firstHook() {
    System.out.println("First @Before Hook");
}

@Before(order = 2)
public void secondHook() {
    System.out.println("Second @Before Hook");
}

@After(order = 1)
public void firstAfterHook() {
    System.out.println("First @After Hook");
}

@After(order = 2)
public void secondAfterHook() {
    System.out.println("Second @After Hook");
}
```

Tagged Hooks

Running Hooks for Specific Scenarios:

- You can use **tagged hooks** to apply hooks only to scenarios with specific tags.
- Tagged hooks allow more **fine-grained control** over which hooks should run for specific scenarios.

Explanation:

- **@Before("@smoke")** runs only for scenarios tagged with **@smoke**.
- **@After("@regression")** runs only for scenarios tagged with **@regression**.

```
@Before("@smoke")
public void setupSmokeTest() {
    System.out.println("Setup for Smoke Test");
    // Code to run before smoke tests
}

@After("@regression")
public void tearDownRegressionTest() {
    System.out.println("Teardown for Regression Test");
    // Code to run after regression tests
}
```

Scenario vs Step Hooks

- **Scenario Hooks** (`@Before` and `@After`) run **before or after the entire scenario**.
- **Step Hooks** (using `@BeforeStep` and `@AfterStep`) run **before or after each step** in the scenario.

Explanation:

- Use `@BeforeStep` and `@AfterStep` for code that needs to run around each step (e.g., taking screenshots for every step in UI automation).

```
@BeforeStep
public void beforeEachStep() {
    System.out.println("This runs before each step");
}

@AfterStep
public void afterEachStep() {
    System.out.println("This runs after each step");
}
```

Cucumber Background Keyword

- **Purpose:** To define common steps that need to be executed before each scenario in a feature file.
- **Usage:** Replaces repeated steps in scenarios, keeping the feature file clean and readable.
- **Execution:** The steps in the **Background** section are automatically run before every scenario.
- **When to Use:** Use **Background** when multiple scenarios share the same preconditions (e.g., logging in, navigating to a page).

```
Feature: Login functionality

  Background:
    Given the user is on the login page
    And the user has entered valid credentials

  Scenario: Successful login
    When the user clicks the login button
    Then the user should be redirected to the home page

  Scenario: Logout functionality
    Given the user is logged in
    When the user clicks the logout button
    Then the user should be redirected to the login page
```

Background Keyword vs Before Hook

Use **Background** When:

- You have **common steps** (e.g., navigation, filling forms) that you want to share across all scenarios in a **specific feature**.
- You want these steps to be **readable in Gherkin**, so non-technical team members can understand the test flow.
- You want to keep scenarios clean and avoid repetition in the feature file.

Use **@Before** Hook When:

- You need to perform **technical setup tasks** before each scenario that **don't need to be expressed in Gherkin**.
- You need to execute code that **doesn't represent user interactions**, like initializing drivers, setting up test data, or resetting the application state.
- You want to apply the same setup across **multiple feature files**, as hooks can be global or scoped to specific tags.

Real-Time Example: Hooks in Web Testing

Hooks for Web UI Automation with Selenium:

```

@Before
public void setup() {
    System.out.println("Launching Chrome browser");
    WebDriver driver = new ChromeDriver();
    driver.manage().window().maximize();
    driver.get("http://example.com");
}

@After
public void teardown() {
    System.out.println("Closing browser");
    driver.quit();
}

```

Explanation:

- The **@Before** hook opens a Chrome browser and navigates to the test site.
- The **@After** hook closes the browser after the scenario is completed.

Best Practices for Using Hooks

1. **Avoid Business Logic in Hooks:**

- a. Hooks should primarily be used for **setup** and **teardown**, not actual test logic.

2. **Use Tagged Hooks:**

- a. Apply hooks to specific scenarios with tags to **avoid unnecessary setup** for unrelated scenarios.

3. **Clean Up Resources in @After Hooks:**

- a. Always ensure that resources (like browser instances, database connections) are properly cleaned up.

4. **Use Order Sparingly:**

- a. Only control the order of hooks when absolutely necessary.

Common Mistakes with Hooks

1. **Overusing Hooks:**

- a. Hooks should not contain too much logic, or they can become difficult to maintain.

2. **Ignoring Cleanup:**

- a. Always implement **@After** hooks to clean up after each scenario to avoid side effects on other tests.

3. **Forgetting Tagged Hooks:**

- a. Failing to use tagged hooks when specific behavior is needed for certain scenarios can lead to **unnecessary setup** for all tests.

Cucumber Reporting

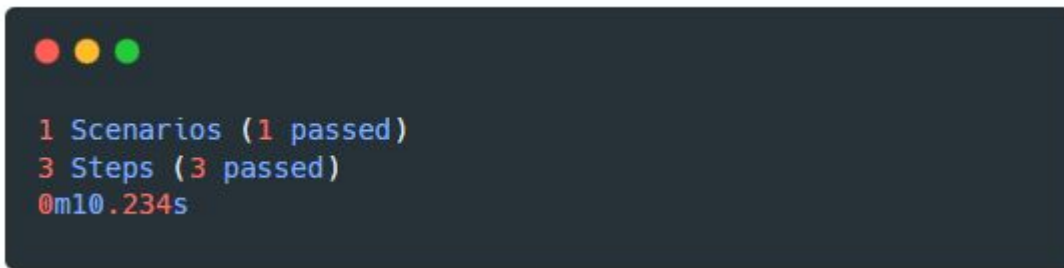
Importance of Test Reports:

- Test reports help in **visualizing the test results**, giving a clear overview of which tests passed or failed.
- They provide **detailed feedback** about the executed scenarios, helping teams to **identify issues** quickly.
- Reports are crucial for **continuous integration (CI)** pipelines, ensuring quality throughout development.

Cucumber's Built-in Reports

- Cucumber automatically generates simple reports after running tests in the console or terminal.
- Basic output includes:
 - Number of scenarios passed/failed.
 - Number of steps passed/failed/skipped.
 - Time taken for each scenario.

Example Console Output:

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The terminal displays the following output in a monospaced font:

```
1 Scenarios (1 passed)
3 Steps (3 passed)
@10.234s
```

Types of Cucumber Reports

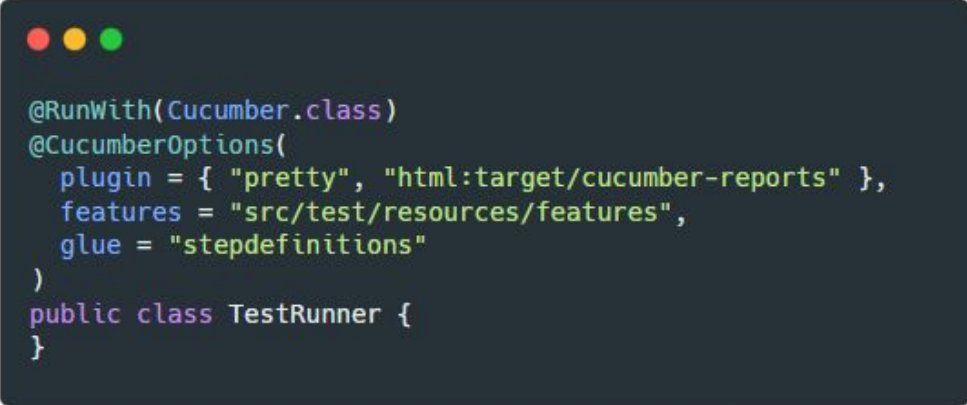
Different Reporting Formats:

Cucumber can generate reports in the following formats:

1. **HTML Reports:** A more readable, browser-friendly report.
2. **JSON Reports:** Machine-readable format, used in CI systems.
3. **JUnit XML Reports:** Compatible with CI tools like Jenkins, allowing integration with build pipelines.

Generating HTML Reports

To generate an HTML report, you need to configure the **Cucumber Runner** to output the report in the desired format.



```
@RunWith(Cucumber.class)
@cucumberOptions(
    plugin = { "pretty", "html:target/cucumber-reports" },
    features = "src/test/resources/features",
    glue = "stepdefinitions"
)
public class TestRunner {
}
```

"html:target/cucumber-reports": This specifies that the report should be generated in **HTML format** and stored in the **target/cucumber-reports** directory.

Generating JSON Reports

JSON reports are used for detailed reporting or integration with CI tools.

```
@RunWith(Cucumber.class)
@cucumberOptions(
    plugin = { "pretty", "json:target/cucumber.json" },
    features = "src/test/resources/features",
    glue = "stepdefinitions"
)
public class TestRunner {
}
```

"json:target/cucumber.json": Generates a **JSON report** that can be processed by various tools, such as Jenkins or Allure.

Generating JUnit XML Reports

JUnit reports are compatible with many CI systems like Jenkins.

```
@RunWith(Cucumber.class)
@cucumberOptions(
    plugin = { "pretty", "junit:target/cucumber.xml" },
    features = "src/test/resources/features",
    glue = "stepdefinitions"
)
public class TestRunner {
}
```

"junit:target/cucumber.xml": Generates a **JUnit XML report** in the target directory.

Third-Party Reporting Plugins

You can enhance Cucumber reporting by integrating third-party plugins like:

1. **Extent Reports:** Provides visually appealing reports with detailed charts and logs.
2. **Allure Reports:** Offers rich graphical reports, integrating with both Cucumber and Jenkins.

Best Practices for Reporting

1. Use Meaningful Scenario Names:

- a. Ensure scenario names in feature files are clear and descriptive for better report readability.

2. Organize Reports by Tags:

- a. Tagging scenarios (e.g., @smoke, @regression) can make the reports easier to filter and analyze.

3. Generate Multiple Formats:

- a. It's a good practice to generate **both HTML and JSON reports** for different use cases (manual review vs CI integration).

4. Monitor Failures with Reports:

- a. Always review test reports to catch recurring failures and patterns.

Common Mistakes in Cucumber Reporting

1. **Overlooking Report Files:**

- a. Ensure report paths are correctly specified in the runner; otherwise, reports won't be generated.

2. **Ignoring Failed Steps:**

- a. Failed steps should always be investigated using the detailed information provided in the reports.

3. **Not Using CI Integration:**

- a. Failing to integrate Cucumber reports into your CI pipeline makes it harder to monitor test results.

Cucumber Integration with Selenium and REST Assured

What are Selenium and REST Assured?:

Selenium WebDriver:

- An open-source tool for automating web browsers.
- Enables testing of web applications across different browsers and platforms.
- Supports multiple programming languages like Java, Python, and C#.

REST Assured:

- A Java library for testing and validating RESTful APIs.
- Simplifies the process of testing HTTP endpoints.
- Allows for easy integration with testing frameworks like JUnit and Cucumber.

Why Integrate Cucumber with Selenium and REST Assured?

1. Unified Testing Approach:

- Use Cucumber to write BDD-style tests for both UI and API layers.
- Promote collaboration between technical and non-technical team members.

2. Readable Tests:

- Feature files describe the behavior in plain English.
- Easier for stakeholders to understand the testing process.

3. Reusable Steps:

- Create common step definitions that can be reused across multiple scenarios.

4. Comprehensive Test Coverage:

- Validate both the frontend (UI) and backend (API) functionalities.

Setting Up Cucumber with Selenium

Dependencies: Include the following in your `pom.xml` for Maven projects:

1. **Cucumber**
2. **Selenium WebDriver**
3. **JUnit/TestNG** (for running tests)

```
<dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-java</artifactId>
  <version>7.x.x</version>
</dependency>

<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-java</artifactId>
  <version>4.x.x</version>
</dependency>

<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>5.x.x</version>
</dependency>
```

Example: Cucumber + Selenium Test

Writing a UI Test with Cucumber and Selenium:

Feature File Example:

```
Feature: User Login

  Scenario: Valid login
    Given the user is on the login page
    When the user enters valid credentials
    Then the user should be redirected to the dashboard
```

Step Definitions Example:

```
@Given("the user is on the login page")
public void navigateToLoginPage() {
    driver = new ChromeDriver();
    driver.get("http://example.com/login");
}


@When("the user enters valid credentials")
public void enterValidCredentials() {
    driver.findElement(By.id("username")).sendKeys("user");
    driver.findElement(By.id("password")).sendKeys("password");
    driver.findElement(By.id("submit")).click();
}

@Then("the user should be redirected to the dashboard")
public void verifyDashboard() {
    Assert.assertEquals("Dashboard", driver.getTitle());
}
```

Integrating Cucumber with REST Assured

Setting Up Cucumber and REST Assured:

- **REST Assured** allows for easy testing of REST APIs by writing BDD-style tests.
- Add the following dependencies to your `pom.xml`:



```
<dependency>
  <groupId>io.rest-assured</groupId>
  <artifactId>rest-assured</artifactId>
  <version>4.x.x</version>
</dependency>

<dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-java</artifactId>
  <version>7.x.x</version>
</dependency>
```

Example: Cucumber + REST Assured Test

Writing an API Test with Cucumber and REST Assured:

Feature File Example:

```
Feature: Fetch user data

Scenario: Get user details by ID
    Given the API endpoint is available
    When a GET request is sent to "/users/1"
    Then the response code should be 200
    And the response body should contain the username "john"
```

Step Definitions Example:

```
@Given("the API endpoint is available")
public void apiEndpointAvailable() {
    RestAssured.baseURI = "https://jsonplaceholder.typicode.com";
}

@When("a GET request is sent to \"/users/{int}\"")
public void sendGetRequest(int userId) {
    response = RestAssured.get("/users/" + userId);
}

@Then("the response code should be {int}")
public void verifyResponseCode(int expectedStatusCode) {
    response.then().statusCode(expectedStatusCode);
}

@And("the response body should contain the username {string}")
public void verifyUsername(String expectedUsername) {
    response.then().body("username", equalTo(expectedUsername));
}
```

Cucumber BDD Best Practices and Tips

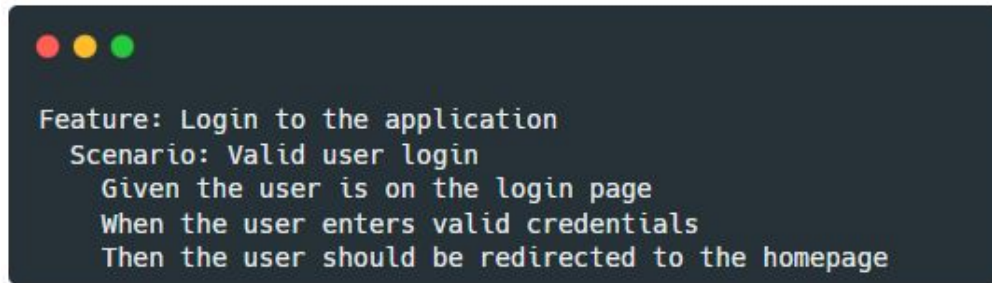
Importance of Best Practices:

- Helps maintain **readable and maintainable** tests.
- Ensures **collaboration** between developers, testers, and stakeholders.
- Increases the **reusability** and **scalability** of tests.
- Reduces the risk of **flaky tests** and **false negatives**.

1. Keep Feature Files Simple and Understandable

Write Human-Readable Feature Files:

- Feature files are primarily for **communication** between teams.
- Use **plain language** that all stakeholders can understand.
- Keep scenarios **short** and **focused** on the behavior, not implementation details.



```
Feature: Login to the application
  Scenario: Valid user login
    Given the user is on the login page
    When the user enters valid credentials
    Then the user should be redirected to the homepage
```

Tip: Avoid using **technical jargon** or complex language in feature files.

2. Avoid Long Scenarios

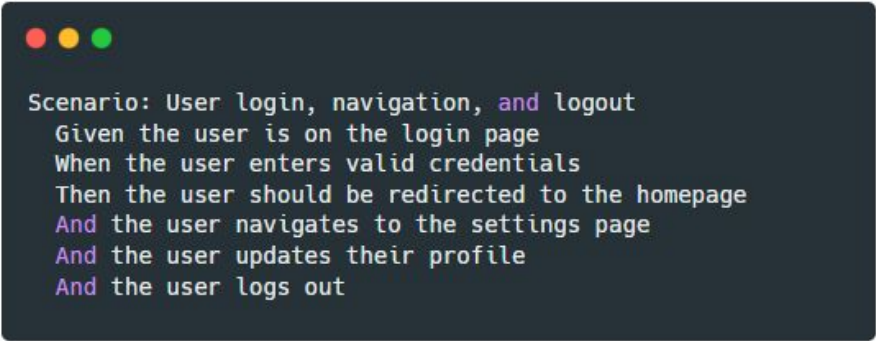
Keep Scenarios Short and Focused:

- A scenario should **focus on one behavior** and test it in isolation.
- Avoid over-complicating a scenario with multiple steps and conditions.

Best Practice:

- Split complex scenarios into **multiple simpler ones**.

Bad Example:



```
Scenario: User login, navigation, and logout
  Given the user is on the login page
  When the user enters valid credentials
  Then the user should be redirected to the homepage
  And the user navigates to the settings page
  And the user updates their profile
  And the user logs out
```

3. Use Meaningful and Consistent Naming

Write Clear Scenario Names:

- Scenario names should clearly describe the **behavior being tested**.
- Consistent naming makes it easy to understand what is being tested.

Tip: Scenario names should read like **user stories** or **acceptance criteria**.

Example:



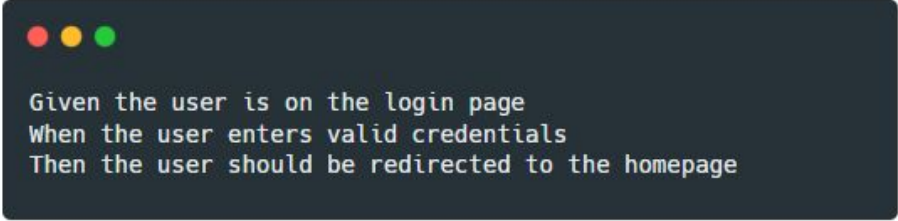
```
Scenario: User logs in with valid credentials  
Scenario: User attempts to log in with an invalid password
```

4. Reuse Step Definitions

Avoid Duplication in Step Definitions:

- **Reuse step definitions** across different scenarios whenever possible.
- Maintain a **library** of common step definitions to avoid redundancy.

Example:



```
Given the user is on the login page  
When the user enters valid credentials  
Then the user should be redirected to the homepage
```

Tip: Stick to a **consistent naming convention** for step definitions.

5. Avoid Technical Details in Feature Files

Focus on Business Behavior, Not Implementation:

- Feature files should describe **what the system should do**, not how it does it.
- Avoid mentioning **UI elements**, **API details**, or **database queries** in feature files.

Bad Example:



When the user clicks the submit button

Good Example:



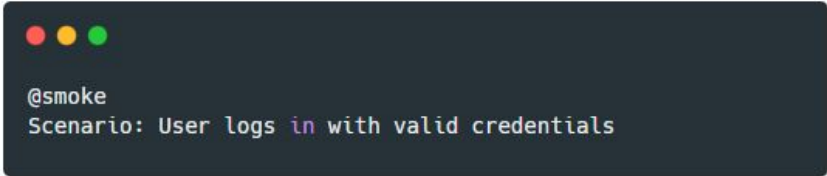
When the user submits the login form

6. Use Tags Wisely

Organizing Tests with Tags:

- Use **tags** to organize scenarios for different test suites (e.g., `@smoke`, `@regression`, `@critical`).
- Tags help in running specific subsets of tests, making the test suite more manageable.

Example:



```
@smoke
Scenario: User logs in with valid credentials
```

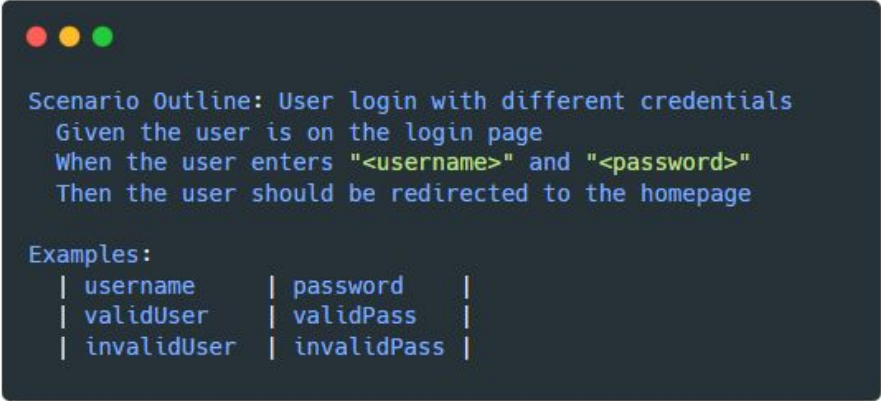
Tip: Avoid **over-tagging** or creating too many custom tags.

7. Parameterize Steps for Reusability

Parameterize Steps for Different Data:

- Parameterize steps to avoid **duplicating step definitions** for similar actions.
- Allows the same step to be used with different sets of data.

Example:



```
Scenario Outline: User login with different credentials
  Given the user is on the login page
  When the user enters "<username>" and "<password>"
  Then the user should be redirected to the homepage
```

Examples:

username	password	
validUser	validPass	
invalidUser	invalidPass	

8. Keep Step Definitions Clean and Concise

Write Simple and Maintainable Step Definitions:

- Step definitions should contain **minimal logic** and focus on calling **helper methods** or **services**.
- Avoid writing complex conditions and calculations directly in the step definitions.

Bad Example:

```
@When("^the user enters valid credentials$")
public void userLogsIn() {
    WebDriver driver = new ChromeDriver();
    driver.get("http://example.com/login");
    driver.findElement(By.id("username")).sendKeys("validUser");
    driver.findElement(By.id("password")).sendKeys("validPass");
    driver.findElement(By.id("submit")).click();
}
```

Good Example:

```
@When("^the user enters valid credentials$")
public void userLogsIn() {
    loginPage.enterCredentials("validUser", "validPass");
    loginPage.submitLogin();
}
```


9. Isolate Test Data

Separate Test Data from Feature Files:

- Keep the **test data** in external files (like CSV, JSON, or Excel) instead of hardcoding it into feature files.
- This makes tests more **scalable** and easier to **maintain**.

Tip: Use parameterized steps to read data from external sources.

10. Use Cucumber Hooks Efficiently

Use Hooks for Setup and Teardown:

- Use **@Before** and **@After** hooks to handle **test setup** and **cleanup**.
- Ensure that hooks are used for **initialization** (e.g., opening a browser, connecting to a database) and **not for test logic**.

Tip: Don't overload hooks with too many tasks. Use **tagged hooks** to control when certain setup tasks should run.

11. Prioritize Test Scenarios

Focus on Critical User Journeys:

- Prioritize scenarios that represent **key business flows** or **critical paths** in the application.
- Ensure that tests cover the most **impactful features** and potential **risk areas**.

12. Make Tests Independent

Ensure Test Independence:

- Test scenarios should be **independent** of each other.
- Avoid having scenarios that **depend on the outcome** of previous scenarios.
- Each test should **reset** the application state.

Tip: Use **Cucumber hooks** to reset the state between tests.

13. Review and Refactor Regularly

Maintain Your Test Suite:

- Regularly **review** and **refactor** feature files and step definitions to keep them clean.
- Remove **redundant steps** and consolidate duplicate scenarios.

Tip: Have a dedicated time for **test maintenance** in your sprint planning.