1. **What is Cucumber?**

Cucumber is an open-source tool that supports **Behavior-Driven Development (BDD)**, a software development methodology that bridges the gap between technical and non-technical team members. It allows teams to write tests in plain, human-readable language (often Gherkin syntax) that anyone can understand, making the development process more collaborative and transparent.

**Key Features of Cucumber:**

1. **Plain Language Test Scenarios**:
   * Test cases are written in Gherkin, a simple language using keywords like Given, When, Then, And & But.
   * Example:

Gherkin code

Feature: Login functionality

Scenario: Successful login

Given the user is on the login page

When the user enters valid credentials

Then the user should see the dashboard

| **Term** | **Meaning** | **Purpose** |
| --- | --- | --- |
| Feature | A high-level description of functionality | Describes **what** we are testing |
| Scenario | A concrete example/test case of a feature | Describes **how** it behaves |

1. **Bridges Technical and Non-Technical Teams**:
   * Encourages collaboration between developers, testers, and business stakeholders.
   * Focuses on the behavior of the application rather than technical implementation.
2. **Integrates with Testing Frameworks**:
   * Works seamlessly with programming languages like Java, Python, Ruby, etc.
   * Commonly used with **JUnit** or **TestNG** in Java.
3. **Supports Automation**:
   * Cucumber scenarios can be linked to underlying code (step definitions) that automate test execution.
   * For example, Java code implementing the above scenario:

java

@Given("the user is on the login page")

public void userOnLoginPage() {

// Code to navigate to the login page

}

@When("the user enters valid credentials")

public void userEntersCredentials() {

// Code to input credentials

}

@Then("the user should see the dashboard")

public void userSeesDashboard() {

// Code to verify the dashboard

}

1. **Promotes Reusability and Maintenance**:
   * Steps can be reused across different scenarios.
   * Improves the maintainability of test scripts as the business rules evolve.
2. **Multi-Platform Support**:
   * Works for web, mobile, and desktop applications.

## **Scenario: Online Shopping Cart (E-Commerce Website)**

Let’s say we’re building a feature for an online store: **“Adding items to the shopping cart.”**

### 1. ****Business Stakeholder’s View (Non-Technical)****

The product owner says:

"As a customer, I want to add products to my cart so I can buy them later."

This is a **business goal**, clear and understandable by everyone — but not technical.

### 2. ****Behavior-Driven View (Common Language for All)****

**Using BDD format (Given-When-Then), we write:**

|  |
| --- |
| **Scenario:** Add item to cart  **Given** the user is on the product page  **When** the user clicks "Add to Cart"  **Then** the product should be added to the shopping cart  **And** the cart count should increase by 1 |

**Now developers, testers, and business users all understand this behavior**.

* **Developer** knows what logic to write.
* **Tester** knows what to validate.
* **Business person** confirms the feature is useful.

### 3. ****Developer’s View (Technical Implementation)****

The developer may implement this by:

* Writing a function in JavaScript or Java to handle "Add to Cart"
* Updating the cart data in the backend
* Storing items in a session or database

But those technical details are **hidden from the business user**.  
They only care that: "The product gets added to the cart."

We’re **focusing on the behavior** — what the system should do — not **how** it does it technically.

This bridges the gap between:

* Developers (who write code),
* Testers (who verify behavior),
* Business users (who define goals).

**Benefits of Using Cucumber in BDD:**

* Enhances team communication by using a shared vocabulary.
* Makes tests readable and accessible to non-developers.
* Encourages a **"shift-left" approach** by defining behaviors early in the software development lifecycle.
* Reduces the gap between requirements and implementation.

**Typical Workflow in Cucumber:**

1. **Write Feature Files**:
   * Define behavior scenarios in .feature files using Gherkin syntax.
2. **Write Step Definitions**:
   * Implement code to automate the steps in feature files.
3. **Run the Tests**:
   * Execute the scenarios using Cucumber and generate detailed reports.
4. **Refine and Iterate**:
   * Adjust scenarios and code as the application evolves.
5. **List of Different BDD Tools available for (Cucumber BDD)**

Here’s a list of popular **BDD tools**, including **Cucumber** and its alternatives, categorized by programming languages and key features:

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Primary Language** | **Gherkin Syntax** | **Best For** |
| Cucumber | Multi-language | Yes | General-purpose BDD |
| SpecFlow | .NET | Yes | .NET projects |
| Behave | Python | Yes | Python projects |
| JBehave | Java | Partial | Java projects |
| Gauge | Multi-language | No (Markdown) | Modular testing |
| Serenity BDD | Java/Groovy | Yes | Detailed reporting |
| Behat | PHP | Yes | PHP web projects |
| Lettuce | Python | Yes | Python ecosystem integration |
| Pytest-BDD | Python | Yes | Python ecosystem integration |
| Robot Framework | Multi-language | Partial | Keyword-driven testing |
| Concordion | Java/.NET | No | Documentation-driven testing |
| Turnip | Ruby | Yes | Ruby-based BDD |
| Easyb | Groovy | Partial | Groovy testing |

### ****Recommendations Based on Use Cases****

#### **1. Web and Mobile Application Testing**

* **Tool**: Cucumber, Serenity BDD, Gauge.
* **Why?**:
  + Works well with Selenium/Appium for UI automation.
  + Supports integration with popular testing frameworks and CI tools.
  + Serenity BDD offers rich reporting for better test documentation.

#### **2. API Testing**

* **Tool**: Cucumber, Rest-Assured (with Serenity BDD).
* **Why?**:
  + Write API tests in BDD format for clear, behavior-focused validation.
  + Serenity BDD can integrate REST APIs with automated reporting.

#### **3. Enterprise Applications (.NET)**

* **Tool**: SpecFlow.
* **Why?**:
  + Best suited for .NET projects with native Visual Studio support.
  + Easily integrates with Azure DevOps and CI/CD pipelines.

#### **4. Small-Scale or Simple Projects**

* **Tool**: Behave, Lettuce, Gauge.
* **Why?**:
  + Lightweight tools with straightforward syntax.
  + Easy to set up and learn for smaller teams.

#### **5. Documentation-Driven Testing**

* **Tool**: Concordion, Robot Framework.
* **Why?**:
  + Allows creating executable documentation that combines testing with specifications.

### ****Final Suggestions****

* **If you're new to BDD**, start with **Cucumber** or **SpecFlow** due to their simplicity and wide adoption.
* **If you value reports and documentation**, consider **Serenity BDD** or **Robot Framework**.
* For **Python teams**, start with **Behave** or **Pytest-BDD**.
* If you're working in **.NET**, **SpecFlow** is the clear winner.

### What is Behavior-Driven Development (BDD)?

**Behavior-Driven Development (BDD)** is a **software development approach** that emphasizes collaboration between developers, testers, and business stakeholders. It focuses on defining the behavior of an application using **plain, human-readable language** that is easily understood by everyone involved in the project.

BDD aims to bridge the gap between technical teams and business stakeholders, ensuring that the application meets business requirements and delivers value.

### ****Key Principles of BDD****

1. **Focus on Behavior**:
   * Instead of focusing on how the system works internally, BDD focuses on what the system should do (its behavior).
   * Scenarios describe the desired outcomes from a user's perspective.
2. **Collaboration**:
   * Encourages active participation from developers, QA engineers, and business stakeholders during requirements gathering and test design.
3. **Executable Specifications**:
   * Requirements are written as **testable examples** that serve as both documentation and automated tests.
4. **Ubiquitous Language**:
   * Uses a shared vocabulary that everyone on the team can understand, ensuring clear communication.
5. **Test-First Approach**:
   * Similar to Test-Driven Development (TDD), but with a focus on describing behaviors rather than implementation details.

### ****Key Components of BDD****

1. **Feature Files**:
   * Written in **Gherkin syntax**, they define the behavior of the application.
   * Example of a feature file:

Gherkin code

Feature: User login

Scenario: Successful login

Given the user is on the login page

When the user enters valid credentials

Then the user should see the dashboard

1. **Step Definitions**:
   * Code that maps the steps in feature files to application logic.
   * Example (Java with Cucumber):

@Given("the user is on the login page")

public void userOnLoginPage() {

// Code to navigate to the login page

}

1. **BDD Tools**:
   * Popular tools like **Cucumber**, **SpecFlow**, and **Behave** help implement BDD by linking human-readable feature files with test automation code.

### ****Benefits of BDD****

1. **Improved Collaboration**:
   * Ensures developers, testers, and business stakeholders are on the same page.
   * Reduces misunderstandings and misaligned expectations.
2. **Increased Test Coverage**:
   * Encourages writing clear, detailed scenarios that result in comprehensive test cases.
3. **Living Documentation**:
   * Feature files double as documentation, ensuring requirements are always up-to-date and testable.
4. **Early Detection of Issues**:
   * Defining behaviors upfront minimizes late-stage defects.
5. **Business Value Focus**:
   * Ensures that development priorities align with business goals.

### ****BDD vs. TDD :****

### **Behavior-Driven Development (BDD)** and **Test-Driven Development (TDD)** are both development methodologies focused on ensuring software quality and correctness. However, they differ significantly in their **focus**, **process**, and **outcomes**.

| **Aspect** | **BDD** | **TDD** |
| --- | --- | --- |
| **Focus** | Behavior (what the system does). | Implementation (how the system works). |
| **Language** | Plain English, readable by all stakeholders. | Code-centric, aimed at developers. |
| **Collaboration** | Involves business stakeholders and testers. | Primarily involves developers. |
| **Output** | Testable examples and behavior specifications. | Unit tests for code functionality. |

### ****Which to Use When?****

* **TDD** is great for developers to design and test small units of code.
* **BDD** is ideal when you want **everyone on the team** — including non-coders — to understand and validate how the system behaves.

| * **BDD is about** | **"Are we building the right thing?" (Behavior validation)** |
| --- | --- |

|  |  |
| --- | --- |
| * **TDD** is about | **"Are we building the thing right?"** (Code correctness) |

**Summary Table:**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **BDD** | **TDD** |
| **Focus** | **Behavior of the system** | **Correctness of code** |
| **Language** | **Plain English (e.g., Gherkin)** | **Code-centric** |
| **Collaboration** | **Developers, testers, stakeholders** | **Primarily developers** |
| **Output** | **Executable specifications (features)** | **Unit tests** |
| **Scope** | **End-to-end, integration tests** | **Unit tests** |
| **Tools** | **Cucumber, SpecFlow, Behave, Serenity** | **JUnit, TestNG, NUnit, Mocha** |
| **Cost of Maintenance** | **Lower due to plain language features** | **Higher for changes in code logic** |
| **Who Benefits** | **, testers, devs Business stakeholders** | **Developers** |

### ****When to Use BDD****

* When **clear communication** between technical and non-technical team members is crucial.
* For projects with **complex requirements** that need constant validation against business goals.
* To align development efforts with **user needs** and **business priorities**.

The official website for Cucumber is: <https://cucumber.io>

**It provides resources for:**

* Learning Cucumber and BDD.
* Downloading tools and integrations.
* Accessing documentation, tutorials, and examples.
* Understanding Gherkin syntax and step definitions.

1. **What is feature file in cucumber framework?**

In the Cucumber framework, a **feature file** is a plain text file that contains the specifications of a feature or functionality to be tested. It is written in **Gherkin** syntax, which uses human-readable keywords like Feature, Scenario, Given, When, Then, and more.

**Key Characteristics of a Feature File**

1. **Plain Text Format**: The file has a .feature extension.
2. **Describes Features**: Each feature file corresponds to a feature or functionality of the application.
3. **Includes Scenarios**: Scenarios in the file represent specific test cases.
4. **Human-Readable**: Written in Gherkin, making it accessible to all stakeholders.
5. **Executable**: The steps in the feature file are linked to step definitions in the code, making them executable.

**Gherkin's syntax focuses on clarity and uses a set of predefined keywords to define tests in a human-readable format. Here's a list of Gherkin keywords:**

|  |  |  |
| --- | --- | --- |
| **Keyword** | **Description** | **Example** |
| Feature | Describes the functionality being tested. | Feature: Login functionality |
| Scenario | Represents a single test case. | Scenario: Successful login with valid credentials |
| Scenario Outline | Used for parameterized tests with multiple data sets. | Scenario Outline: Login with multiple users |
| Examples | Provides data for Scenario Outline. | Examples: | username | password | | user1 | pass123 | | user2 | pass456 | |
| Given | Describes initial context or preconditions. | Given the user is on the login page |
| When | Describes an action or event. | When the user enters valid credentials |
| Then | Describes the expected outcome or result. | Then the user should be redirected to the dashboard |
| And | Adds additional conditions or steps to Given, When, or Then. | And the user sees a logout button |
| But | Describes exceptions or additional conditions. | But the user should not see any admin options |
| Background | Provides a common setup for all scenarios in a feature file. | Background: Given the user has an active account And the user is logged into the app |
| Rule | Describes business rules related to the feature. | Rule: Password must be at least 8 characters |
| Doc Strings | Used for multiline text or additional details in steps. | Given the following user details: """ username: user1 password: pass123 """ |
| Tags | Metadata for scenarios, often used for filtering or grouping. | @smoke @login Scenario: Successful login |
| Comments | Ignored by Gherkin, starts with # | # This is a comment |

**5.What is Step Definition in cucumber framework?**

In the Cucumber framework, a **Step Definition** is a piece of code that specifies the implementation of steps written in the feature file. It acts as a bridge between the human-readable Gherkin steps in the feature file and the actual code logic that performs the corresponding actions.

Each step in the feature file (e.g., Given, When, Then, And, But) is mapped to a step definition written in a programming language like **Java**, **Python**, **Ruby**, etc.

**Purpose of Step Definitions :**

1. **Execution Logic**: It defines what happens when a specific step in the feature file is executed.
2. **Reusability**: A single step definition can be reused across multiple feature files and scenarios if the step text matches.
3. **Separation of Concerns**: Keeps the test specification (feature file) and implementation logic (step definitions) separate.

**Structure of Step Definitions :**

A step definition typically includes:

1. **Annotation**: The Cucumber annotation (@Given, @When, @Then, etc.) maps the step text in the feature file to the method.
2. **Regular Expression**: A pattern to match the step text in the feature file.
3. **Method**: A method that implements the logic for the step.

### ****Example of a Step Definition (in Java)****

#### **Feature File (Gherkin code)**

**Feature:** Login functionality

**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

#### **Step Definition File (Java code)**

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

public class LoginSteps {

**@Given**("the user is on the login page")

public void userIsOnLoginPage() {

// Code to navigate to the login page

System.out.println("User is on the login page.");

}

**@When**("the user enters valid credentials")

public void userEntersValidCredentials() {

// Code to enter username and password

System.out.println("User enters valid credentials.");

}

**@Then**("the user should be redirected to the dashboard")

public void userIsRedirectedToDashboard() {

// Code to verify dashboard redirection

System.out.println("User is redirected to the dashboard.");

}

}

**How Step Definitions Work**

1. **Matching Steps**: When Cucumber runs, it scans the feature file and searches for matching step definitions using annotations and regular expressions.
2. **Executing Steps**: When a match is found, the corresponding step definition method is executed.
3. **Error Handling**: If a step in the feature file doesn't have a corresponding step definition, Cucumber throws an error.

### ****Parameterized Steps****

You can use regular expressions in step definitions to capture dynamic values.

#### **Example Feature File (Gherkin code)**

**Scenario:** Login with a specific user

**Given** the user logs in with username **"user1"** and password **"password123"**

#### **Step Definition (Java code)**

**@Given**("the user logs in with username **{string}** and password **{string}**")

public void userLogsInWithCredentials(String username, String password) {

System.out.println("Username: " + username);

System.out.println("Password: " + password);

// Code to log in the user

}

6.Runner class

The **Runner class** in Cucumber is responsible for configuring and executing Cucumber tests. It integrates Cucumber with JUnit or TestNG and defines options such as the location of feature files, step definitions, and reporting plugins.

**Basic Syntax of a Runner Class:**

package runners;

import org.junit.runner.RunWith;

import io.cucumber.junit.Cucumber;

import io.cucumber.junit.CucumberOptions;

@RunWith(Cucumber.class) // Specify that this class will run with the Cucumber test runner

@CucumberOptions(

features = "src/test/resources/features", // Path to the feature files

glue = "stepDefinitions", // Path to the step definitions

plugin = { // Plugins for reporting

"pretty", // Provides a readable console output

"html:target/cucumber-reports.html", // Generates an HTML report

"json:target/cucumber-reports.json" // Generates a JSON report

},

monochrome = true, // Makes console output readable

tags = "@SmokeTest", // Executes scenarios with the specified tag

dryRun = true // Checks if all steps have definitions without executing them

)

public class TestRunner {

}

**Explanation of @CucumberOptions Parameters :**

|  |  |
| --- | --- |
| **Option** | **Description** |
| features | Specifies the path to the feature files. |
| glue | Specifies the package containing step definitions. |
| plugin | Used for generating test execution reports (e.g., HTML, JSON, XML). |
| tags | Runs scenarios or features tagged with specific tags (e.g., @SmokeTest, @Regression). |
| Monochrome, pretty | Makes the console output cleaner and more readable. |
| dryRun | If true, checks if all steps have corresponding step definitions without executing the tests. |

**7.Cucumber Scenario Outline**

**Cucumber Scenario Outline** using **Examples** for data-driven testing in a login feature scenario. This allows you to test the same scenario with different sets of data.

**Feature File:** login.feature

**Feature: Login Functionality**

**Scenario Outline: Successful login with valid credentials**

**Given** the user navigates to the login page

**When** the user enters username **"<username>"** and password **"<password>"**

**And** clicks on the login button

**Then** the user should see the homepage with the title **"<homeTitle>"**

**Examples:**

**| username | password | homeTitle |**

**|** user1 **|** pass123 **|** User Dashboard **|**

**|** admin **|** admin@123 **|** Admin Dashboard **|**

**Step Definitions in Java :**

package stepDefinitions;

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

import org.junit.Assert;

public class LoginSteps {

String currentHomeTitle;

@Given("the user navigates to the login page")

public void navigateToLoginPage() {

System.out.println("Navigated to the login page.");

}

@When("the user enters username {string} and password {string}")

public void enterCredentials(String username, String password) {

System.out.println("Entered Username: " + username);

System.out.println("Entered Password: " + password);

if (username.equals("user1") && password.equals("pass123")) {

currentHomeTitle = "User Dashboard";

} else if (username.equals("admin") && password.equals("admin@123")) {

currentHomeTitle = "Admin Dashboard";

} else {

currentHomeTitle = "Invalid Login";

}

}

@When("clicks on the login button")

public void clickLoginButton() {

System.out.println("Clicked on the login button.");

}

@Then("the user should see the homepage with the title {string}")

public void verifyHomePageTitle(String expectedTitle) {

Assert.assertEquals("Homepage title mismatch!", expectedTitle, currentHomeTitle);

System.out.println("Verified homepage title: " + currentHomeTitle);

}

}

**8.Tags**

Cucumber tags are used to filter and manage the execution of test cases based on specific conditions or categories. Tags can be applied at the **Feature**, **Scenario**, or **Examples** level in Gherkin files. Below is a comprehensive guide to Cucumber tags, their syntax, and examples.

### ****Tags in**** @CucumberOptions

The tags attribute in @CucumberOptions is used to filter scenarios for execution.

**Syntax:** @CucumberOptions( tags = "tagExpression" )

**1. Basic Tag Syntax**

* Tags are prefixed with @.
* Tags should not contain spaces. If needed, use underscores (\_) or hyphens (-).

|  |
| --- |
| @SmokeTest  **Feature:** Login functionality  @Positive  **Scenario:** Successful login with valid credentials  **Given** I navigate to the login page  **When** I enter valid credentials  **Then** I should see the dashboard |

### ****2. Combining Tags****

Cucumber allows the use of logical operators to combine multiple tags for more complex scenarios.

#### **a**. **OR (**,**)**

Run scenarios that have at least one of the specified tags.

**Example:** @SmokeTest,@Regression

#### **b.** **AND**

Run scenarios that have all specified tags. This is done using multiple --tags in CLI or within a tag expression.

**Example:** "@SmokeTest and @Critical"

#### **c.** **NOT (**~**)**

Exclude scenarios with a specific tag.

**Example:** "not @WIP"

### ****3. Tag Placement****

Tags can be placed at different levels in the feature file:

#### **a. Feature Level**

Applies to all scenarios in the feature.

**Example:**

@SmokeTest

Feature: User Registration

Scenario: Register with valid details

Given I navigate to the registration page

When I fill in valid user details

Then I should see a confirmation message

#### **b. Scenario Level**

Applies to a specific scenario.

**Example:**

Feature: User Login

@Regression

Scenario: Invalid login with wrong password

Given I navigate to the login page

When I enter an invalid password

Then I should see an error message

**9.Hooks in Cucumber**

1. **Hooks in Cucumber**

Hooks in Cucumber are special blocks of code that can be executed at specific points during the test execution lifecycle. They allow you to add logic that runs **before** or **after** scenarios, steps, or feature files.

#### **Common Types of Hooks**

1. @Before **Hook**
   * Runs before each scenario.
   * Used for setup tasks, such as initializing data, starting browsers, or setting configurations.

* **Example:**

**@Before**

**public void setUp()**

**{**

**System.out.println("This runs before each scenario.");**

**}**

1. **@After Hook**

* Runs after each scenario.
* Used for cleanup tasks, such as closing browsers or resetting data.
* **Example:**

**@After**

**public void tearDown()**

**{**

**System.out.println("This runs after each scenario.");**

**}**

1. **@BeforeStep and @AfterStep Hooks**

* @BeforeStep: Executes before each step in a scenario.
* @AfterStep: Executes after each step in a scenario.
* **Example:**

**@BeforeStep**

**public void beforeStep() {**

**System.out.println("This runs before each step.");**

**}**

**@AfterStep**

**public void afterStep() {**

**System.out.println("This runs after each step.");**

**}**

1. **Tagged Hooks in Cucumber**

Tagged Hooks are hooks that run only for scenarios or features marked with a specific tag. This is useful for applying setup or teardown logic selectively based on tags.

#### **How to Use Tagged Hooks**

* Add a tag to your scenario or feature in the .feature file.
* Use the same tag in the hook definition.

**Example:**

**Feature File** (example.feature):

**@smoke**

**Scenario:** Login test

**Given** I am on the login page

**When** I enter valid credentials

**Then** I should see the dashboard

**Java Hook:**

**@Before**("**@smoke**")

public void beforeSmokeTest() {

System.out.println("This runs before smoke test scenarios.");

}

**@After**("**@smoke**")

public void afterSmokeTest() {

System.out.println("This runs after smoke test scenarios.");

}

#### **Benefits of Tagged Hooks:**

1. **Selective Execution**: You can target specific scenarios or features based on tags.
2. **Better Organization**: Helps manage hooks for large projects.
3. **Flexibility**: Allows combining hooks for different purposes (e.g., smoke, regression, etc.).
4. **Value Attribute, Multiple Hooks and Their Order in Cucumber**

In Cucumber, the **value attribute** for hooks is used to define tags that determine when the hook should execute. This allows you to specify hooks for scenarios or features with specific tags. It is most commonly used with hooks like **@Before** and **@After**.

### ****Syntax for Using the**** value ****Attribute****

The **value attribute** is implied when you specify a tag for a hook. You can directly specify the tag name without explicitly writing **value**.

**Example Syntax:**

**@Before**(**value** = "**@smoke**")

public void beforeSmokeTests() {

System.out.println("This runs before Smoke tests.");

}

**@After**(**value** = "**@regression**")

public void afterRegressionTests() {

System.out.println("This runs after Regression tests.");

}

**Note :** Here, the **value** attribute is optional because tags are the default parameter.

### ****Handling Multiple Tags****

You can specify multiple tags for a hook by using a logical OR syntax (comma-separated tags).

**@Before**(**value** = **"@smoke, @regression**")

public void beforeTaggedTests()

{

System.out.println("This runs before Smoke or Regression tests.");

}

### ****Order with Tagged Hooks****

Tagged hooks can also be prioritized using the **order** attribute.

**@Before**(**value** = **"@smoke**", **orde**r = 1)

public void firstBeforeSmokeTests()

{

System.out.println("First hook for Smoke tests.");

}

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

public void secondBeforeSmokeTests()

{

System.out.println("Second hook for Smoke tests.");

}

1. **How to Retrieve Scenario Names in Hooks in Cucumber?**

You can use the **Scenario object** provided by Cucumber. It gives access to various details about the current scenario, including its **name, tags, and status**.

**Example Code :**

import io.cucumber.java.Before;

import io.cucumber.java.After;

import io.cucumber.java.Scenario;

public class Hooks {

**@Before**

public void beforeScenario(**Scenario** scenario) {

System.out.println("Starting scenario: " + scenario.getName());

}

**@After**

public void afterScenario(**Scenario** scenario) {

System.out.println("Finished scenario: " + scenario.getName());

System.out.println("Scenario status: " + scenario.getStatus());

}

}

### ****Explanation****

1. **Scenario Object**:
   * This is passed as an argument to the hook method.
   * It provides access to:
     + **getName():** Retrieves the name of the scenario.
     + **getStatus():** Retrieves the status (PASSED, FAILED, etc.).
     + **getSourceTagNames():** Retrieves a set of tags associated with the scenario.
2. **Using Hooks**:
   * The **@Before** hook runs before each scenario, printing the name of the scenario.
   * The **@After** hook runs after each scenario, printing the name and the status.

### ****Summary****

* **Group by Functionality**: Separate global, tagged, and step-level hooks.
* **Use Tags**: Execute hooks for specific scenarios or features using tags.
* **Separate Files**: Keep hooks in dedicated classes for clarity.
* **Control Order**: Use the order attribute to prioritize hook execution.
* **Scenario Logging**: Use the Scenario object for scenario details.

By following these practices, your Cucumber hooks will remain organized, reusable, and scalable.

#### **Use Separate Files for Hooks**

Place hooks in separate classes or packages to isolate their functionality.  
**Example folder structure:**

**src/test/java**

├── **hooks**

│ ├── GlobalHooks.java

│ ├── TaggedHooks.java

│ ├── StepHooks.java

├── **stepDefinitions**

├── **features**

**10.what is dryRun in Cucumber**

In Cucumber, dryRun is a feature in the @CucumberOptions annotation that allows you to verify the mapping between the steps in your .feature file and the corresponding step definitions in your code **without actually executing the test steps**.

**Purpose of dryRun**

The primary purpose of dryRun is to:

1. Ensure that all the steps in your feature files have corresponding step definitions.
2. Detect any unimplemented steps early, without running the actual logic.

### ****Syntax****

You use dryRun as a parameter in the @CucumberOptions annotation:

**@CucumberOptions**(

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

**glue** = "steps",

**dryRun** = true // Set to true for dry run

)

### ****How**** dryRun ****Works****

* **dryRun = true**: Cucumber checks for the presence of matching step definitions for every step in the feature files but **does not execute the tests**. If a step definition is missing, it will report the missing step and suggest its skeleton.
* **dryRun = false (default)**: Cucumber both checks for the matching step definitions and executes the tests.

**When to Use dryRun**

* During **initial development** to verify that all steps in feature files have corresponding step definitions.
* After adding new scenarios to ensure step definitions are implemented before actual execution.
* To catch unimplemented steps without executing existing ones (useful in CI/CD pipelines).

**Advantages**

1. Saves time by skipping execution while still validating the setup.
2. Identifies missing step definitions early.
3. Provides step definition skeletons for quick implementation.

**11.How to Organizing the Cucumber Project**

Organizing a Cucumber project properly ensures maintainability, scalability, and collaboration efficiency. A well-structured project helps in managing feature files, step definitions, hooks, test data, and configurations effectively. Below is a recommended structure for organizing a Cucumber project

**Recommended Folder Structure**

src

└── test

├── java

│ ├── steps # Step definition classes

│ ├── runners # Test runner classes

│ ├── hooks # Hooks (Before, After methods)

│ ├── utils # Utility classes (e.g., for data, WebDriver setup)

│ └── config # Configuration-related classes

└── resources

├── features # Feature files

├── data # Test data (e.g., CSV, JSON, Excel)

└── config # Configuration files (e.g., properties, YAML)

### ****1. Feature Files****

Feature files define test scenarios in Gherkin syntax and should be stored in src/test/resources/features.

### ****2. Step Definitions****

Step definitions are Java methods that implement the steps in the feature files. They are stored in the src/test/java/steps package.

### ****3. Test Runners****

Test runners execute the feature files. Store them in the src/test/java/runners package.

### ****4. Hooks****

Hooks allow you to run code before or after each scenario, providing setup or teardown functionality. Store them in the src/test/java/hooks package.

### ****5. Utilities****

Utility classes provide common functionality like reading test data, setting up WebDriver, or generating test reports. Store them in the src/test/java/utils package.

### ****6. Test Data****

Store test data in the src/test/resources/data folder. Data can be in formats like CSV, JSON, or Excel, depending on your requirements.

### ****7. Configuration Files****

Keep configuration details (e.g., URLs, credentials) in the src/test/resources/config folder.

**Best Practices**

1. **Modularize Step Definitions**: Separate step definitions by functionality (e.g., LoginSteps, RegistrationSteps).
2. **Reusable Hooks**: Use hooks for common setup/teardown logic (e.g., starting/stopping WebDriver).
3. **Data-Driven Testing**: Use external files for test data to minimize hardcoding in feature files or step definitions.
4. **Consistent Naming**: Use meaningful and consistent names for packages, files, and classes.
5. **Reporting**: Integrate reporting plugins like cucumber-html-reporter or extent-reports.

**12.what is difference between Extent HTML Reports and Extent Spark reports**

Both **Extent HTML Reports** and **Extent Spark Reports** are part of the Extent Reports library, a popular tool for generating detailed and visually appealing test reports in automation frameworks. The main difference between them lies in their rendering approach and features:

**Extent HTML Reports**

1. **Technology**:
   * Generates a standalone HTML report.
   * Uses older design templates.
2. **Appearance**:
   * More static and traditional.
   * Limited interactivity and visual appeal compared to Spark Reports.
3. **Features**:
   * Provides basic report customization.
   * Suited for legacy projects or simple reporting requirements.
4. **Usage**:
   * Typically used when simplicity or compatibility with older systems is a priority.
5. **Output File**:
   * Produces a .html file that doesn't require additional libraries or scripts for rendering.

**Extent Spark Reports**

1. **Technology**:
   * Built on modern web technologies like Spark framework.
   * Uses a responsive and dynamic design.
2. **Appearance**:
   * Sleek, modern, and highly interactive.
   * Includes advanced UI elements such as collapsible sections, graphs, and filters.
3. **Features**:
   * Supports more advanced customization.
   * Provides filtering options for passed, failed, and skipped tests.
   * Interactive charts and graphical summaries.
4. **Usage**:
   * Ideal for teams requiring modern, interactive reports with rich UI.
5. **Output File**:
   * Produces a .html file but uses JavaScript and CSS for interactivity. Browsers need to allow the execution of scripts.

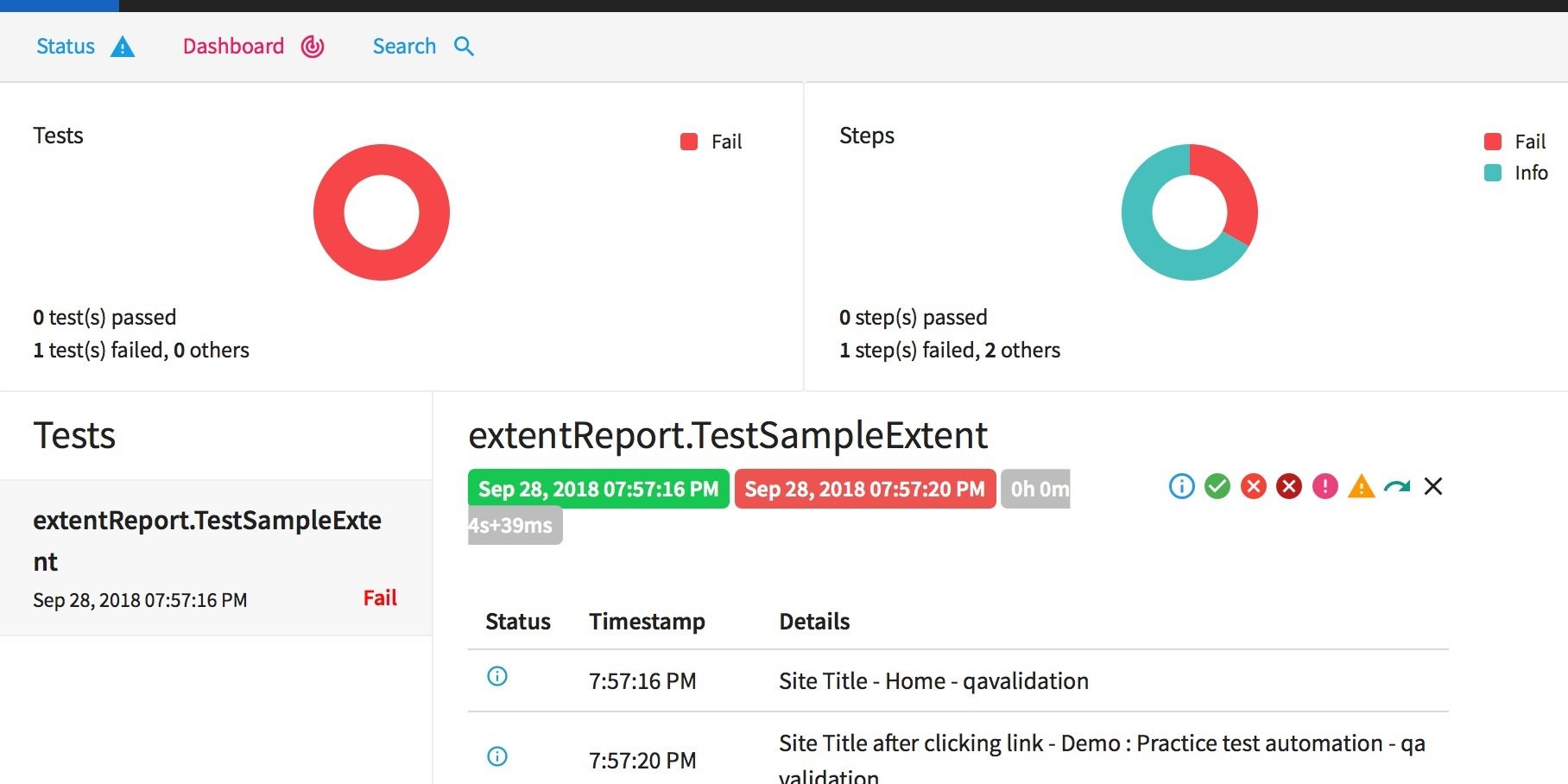
**Comparison Table:**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Extent HTML Reports** | **Extent Spark Reports** |
| Design | Traditional | Modern and interactive |
| Customization | Limited | Advanced |
| Graphs/Charts | Minimal support | Rich graphical visualizations |
| Interactivity | Static | Highly interactive |
| Performance | Lightweight | Slightly heavier due to JS/CSS |

### ****Which One Should You Use?****

* Use **Extent HTML Reports** for simpler projects where basic reporting is sufficient or where older browser/system compatibility is required.
* Use **Extent Spark Reports** for modern automation frameworks that benefit from advanced, interactive reporting and visualization.

**Extent HTML Reports:**



**Extent Spark Reports:**

