Cucumber - Best Practices

To **keep feature files simple and readable in Cucumber**, follow these best practices:

✅ 1. Use Clear and Descriptive Feature Titles

- Keep the **Feature title** concise and focused on one functionality.

```gherkin
Feature: User login functionality
```

✅ 2. Write a Brief, Relevant Feature Description

- Add 1–2 lines explaining the purpose of the feature.

```gherkin
Feature: User login functionality
  As a registered user
  I want to log in to the application
  So that I can access my dashboard
```

✅ 3. Follow the Given-When-Then Structure Clearly

- Use each step (Given/When/Then) to describe user actions and expectations **clearly**.

```gherkin
Scenario: Successful login with valid credentials
  Given the user is on the login page
  When the user enters valid username and password
  And clicks the login button
  Then the user should be redirected to the dashboard
```

✅ 4. Avoid UI/Technical Details

- Don't mention **button IDs, HTML tags, or APIs**. Use business terms.

❌ Bad:

```gherkin
When the user clicks on the button with id "submit-btn"
```

✅ Good:

```
gherkin

When the user submits the login form
```

✅ 5. Reuse Steps with Background (for Repeated Steps)

Use **`Background`** to remove repetition across multiple scenarios.

```
gherkin

Background:
  Given the user is on the login page

Scenario: Login with valid credentials
  When the user enters valid credentials
  Then the user should be redirected to the dashboard

Scenario: Login with invalid credentials
  When the user enters invalid credentials
  Then an error message should be displayed
```

✅ 6. Use Scenario Outline for Data-Driven Testing

Keep variations in data outside of the steps using Examples.

```
gherkin

Scenario Outline: Login attempts with different credentials
  Given the user is on the login page
  When the user enters username "<username>" and password "<password>"
  Then the login result should be "<result>"

Examples:
  | username | password  | result          |
  | user1    | pass123   | success         |
  | user2    | wrongpass | invalid message |
```

✅ 7. Use Consistent Language

- Stick to a **shared vocabulary** across steps (e.g., "logs in" vs "signs in").
- Reuse step definitions wherever possible.

✅ 8. Keep Scenarios Focused

- Each scenario should test **one thing only**. Don't mix multiple actions or flows.

✅ 9. Avoid Logic in Feature Files

- Don’t use **`if`**, **`else`**, **`loop`**, etc. Feature files should describe **behavior**, not logic.

✅ 10. Use Tags for Organization (Optional)

Tags like **`@login`**, **`@smoke`**, **`@regression`** help in organizing tests.

```gherkin
@smoke @login
Scenario: Successful login
```

Would you like a sample feature file based on one of your real projects or topics like login, search, or cart?

Absolutely! In Cucumber, it's essential to **avoid too many steps in one scenario** to keep it **readable, maintainable, and focused**.

✅ Why You Should Avoid Too Many Steps:

Problem	Explanation
❌ Hard to read	Too many steps overwhelm the reader.
❌ Hard to maintain	More steps = more step definitions to update when things change.
❌ Confusing intent	It’s harder to tell what the scenario is really testing.
❌ Mixed responsibilities	One scenario might end up testing multiple behaviors.

✅ Best Practices to Keep Scenarios Short and Focused:

1. Keep to 5–7 Steps Max (Ideally)

Break long workflows into **multiple smaller scenarios**.

```
gherkin

# ✅ GOOD: Clear and focused
Scenario: User logs in successfully
  Given the user is on the login page
  When the user enters valid credentials
  Then the user should be redirected to the dashboard
```

---

2. Split Into Multiple Scenarios

If you’re covering a long flow, break it down.

```
gherkin

# ❌ BAD: Too many actions in one scenario
Scenario: User logs in, adds item to cart, checks out, and logs out

# ✅ BETTER: Break into smaller, focused scenarios
Scenario: User logs in successfully
Scenario: User adds item to cart
Scenario: User completes checkout
Scenario: User logs out
```

---

3. Use Background to Reduce Step Repetition

Instead of repeating steps in every scenario, extract them.

```
gherkin

Background:
  Given the user is logged in

Scenario: Add item to cart
```

```
    When the user adds a product to the cart
    Then the cart should contain the product
```

4. Keep Scenarios Business-Focused

Avoid writing technical details that add unnecessary steps.

❌ Too Detailed:

```
gherkin

When the user clicks on the button
And the modal appears
And the user types into the field
And the user clicks submit
```

✅ Simplified:

```
gherkin

When the user submits the feedback form
```

5. Use Reusable Composite Steps (Optional Advanced Tip)

If you're using tools like **Cucumber Expressions** or helper functions in code, you can **combine multiple UI actions into one business-readable step**.

✅ Summary

Do ✅	Avoid ❌
Limit to ~5–7 steps per scenario	Writing 10+ low-level steps
Split complex flows	Combining multiple flows
Use Background to reduce clutter	Repeating common setup steps
Write business-friendly steps	Describing every UI detail

Would you like an example of refactoring a long scenario into smaller ones?

Grouping scenarios logically using **tags** in Cucumber is a great way to organize your tests, control execution, and improve readability.

✅ What Are Tags?

Tags are annotations prefixed with `@` that you place **above a Scenario, Scenario Outline, or Feature** to categorize or group them.

```gherkin
@smoke
Scenario: Valid user login
  Given the user is on the login page
  When the user logs in with valid credentials
  Then the dashboard should be displayed
```

✅ Why Use Tags?

Benefit	Description
✅ Run specific scenarios	Filter scenarios by tag during execution.
✅ Organize by test type	e.g., **`@smoke`**, **`@regression`**, **`@sanity`**.
✅ Organize by module	e.g., **`@login`**, **`@cart`**, **`@checkout`**.
✅ Easy maintenance	Run or exclude related tests easily.

✅ Examples of Logical Grouping with Tags

By Test Type

```gherkin
@smoke
Scenario: Login with valid credentials

@regression
Scenario: Login fails with invalid credentials
```

By Feature Module

```gherkin
@login
Scenario: User logs in successfully

@cart
Scenario: User adds item to cart
```

By Priority or Environment

```gherkin
@high @staging
Scenario: Checkout with a saved address
```

✅ Running Scenarios Using Tags (Maven, Gradle, CLI)

Maven (with Cucumber-JVM)

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

Gradle

```bash
./gradlew test --tests "*RunnerClass*" -Dcucumber.filter.tags="@regression"
```

CLI (Cucumber-JS, Ruby, etc.)

```bash
cucumber --tags @login
```

✅ Combining Tags

Run scenarios with either tag:

```bash
@smoke or @regression
```

Run scenarios with both tags:

```bash
@smoke and @login
```

Run scenarios excluding a tag:

```bash
not @wip
```

✅ Example with Multiple Tags

```
gherkin

@smoke @login
Scenario: Successful login

@regression @login
Scenario: Login fails with wrong password

@wip
Scenario: Forgot password flow
```

✅ Best Practices

Do ✅	Avoid ❌
Use consistent tag naming	Random or unclear tag names
Tag related scenarios similarly	Overusing too many unique tags
Keep tags meaningful	Using tags that don’t reflect functionality
Use tags in test runners	Ignoring them in execution

Would you like me to help define a **tagging strategy** for your own project (e.g., API tests, UI login, checkout, etc.)?

Following **naming conventions** in Cucumber ensures that your **feature files**, **scenarios**, and **step definitions** are **clear, consistent, and easy to maintain**. Here's a complete guide:

✅ 1. Feature File Names

- Use **snake_case** or **kebab-case**
- Name should reflect the **feature or module** being tested.

✅ Examples:

```pgsql
user_login.feature
checkout_process.feature
add-product-to-cart.feature
```

❌ Avoid:

```pgsql
Login.Feature
test1.feature
featurefile.feature
```

✅ 2. Feature Titles

- Use **natural language**
- Start with **`Feature:`**
- Capitalize first letter only (optional: full sentence)

✅ Example:

```gherkin
Feature: User login functionality
```

❌ Avoid:

```gherkin
Feature: user can log in
Feature: Login123
```

✅ 3. Scenario Titles

- Use **descriptive sentences**
- Use **lowercase**, sentence-style writing

- Include **one purpose per scenario**

✅ Examples:

```
gherkin

Scenario: User logs in with valid credentials
Scenario: User fails to log in with incorrect password
```

❌ Avoid:

```
gherkin

Scenario: Login Test
Scenario: TC_001
```

✅ 4. Step Definitions (Java/JavaScript/Other)

- Follow standard code conventions (e.g., camelCase in Java)
- Use **verbs** to describe actions
- Group logically in step definition classes/files

✅ Java example:

```
java

@Given("the user is on the login page")
public void userIsOnLoginPage() { ... }

@When("the user enters valid credentials")
public void userEntersValidCredentials() { ... }

@Then("the user should see the dashboard")
public void userShouldSeeDashboard() { ... }
```

✅ 5. Tag Naming Conventions

- Use **lowercase**
- Separate words with _ or **-**
- Be meaningful: reflect purpose or scope

✅ Examples:

```
gherkin

@smoke
@regression
@login
@api
```

❌ Avoid:

```gherkin
@SmokeTest
@Test1
@mytag123
```

✅ 6. Reusable Step Language

- Use consistent action verbs:
  - **`Given`** → setup
  - **`When`** → action
  - **`Then`** → result
  - **`And`**, **`But`** → logical continuation

✅ Example:

```gherkin
Given the user is on the login page
When the user enters a valid username and password
Then the user should be redirected to the dashboard
```

✅ Summary Table

Element	Convention Type	Example
Feature file name	**`snake_case`** or **`kebab-case`**	**`user_login.feature`**
Feature title	Descriptive sentence	**`Feature: User login functionality`**
Scenario title	Lowercase, sentence-style	**`Scenario: User logs in successfully`**
Step definitions	camelCase (in code)	**`userIsOnLoginPage()`**
Tags	lowercase, clear	**`@smoke`**, **`@checkout`**

Would you like me to generate a **fully named and structured feature file** for a real use case like login, registration, or checkout using these conventions?

Absolutely! In Cucumber, it's a **best practice to always use `dryRun` before actual test execution** — especially when you're writing or editing feature files or step definitions.

✅ What is `dryRun` in Cucumber?

The **`dryRun`** option tells Cucumber to:

> ✅ **Check for missing step definitions**
> ❌ **Without running any actual test logic**

✅ Why Use `dryRun = true`?

Benefit	Description
✅ Fast feedback	Instantly see which steps need implementation.
✅ Avoid runtime failures	Catch missing glue code early.
✅ Saves time	Don't waste time running incomplete tests.
✅ Ensures all steps are matched	Keeps your suite in sync.

✅ Example in Java (JUnit)

```java
@RunWith(Cucumber.class)
@CucumberOptions(
    features = "src/test/resources/features",
    glue = "stepdefinitions",
    dryRun = true     // ✅ Only checks for unimplemented steps
)
public class TestRunner {
}
```

✅ When to Use `dryRun = true`

Situation	Use dryRun?
✅ Writing new feature files	Yes
✅ Adding new scenarios or steps	Yes
✅ Refactoring step definitions	Yes
❌ Ready for test execution	No – set **`dryRun = false`**

✅ Typical Workflow

1. **Write/Update feature files**
2. **Run with `dryRun = true`**

- ✅ Cucumber tells you which steps are undefined

3. **Implement missing step definitions**
4. **Set `dryRun = false`**
5. **Run actual tests**

🛠️ Example Output of a Dry Run

```shell
You can implement missing steps with the snippets below:

@Given("the user is on the login page")
public void the_user_is_on_the_login_page() {
    // Write code here
    throw new io.cucumber.java.PendingException();
}
```

This helps you quickly copy-paste the code templates into your step definition files.

✅ Summary

Practice	Status
Use **`dryRun = true`** before coding or running	✅ Required
Reduces wasted test runs	✅ Yes
Helps implement missing steps faster	✅ Yes

Would you like me to give you a **template runner file with dryRun toggle** for your current project setup (e.g., JUnit/TestNG/Gradle)?