# Introduction

Cucumber is a tool that supports Behaviour-Driven Development (BDD).

# Behaviour-Driven Development (BDD)

**What is BDD?**

BDD is an **agile software development technique** that **encourages collaboration between developers, QA and non-technical or business participants** in a software project. In other words, BDD helps **close the gap between business people and technical people** by:

- Encouraging collaboration across roles to build shared understanding of the problem to be solved.

- Working in rapid, small iterations to increase feedback and the flow of value

- Producing system documentation that is automatically checked against the system’s behaviour

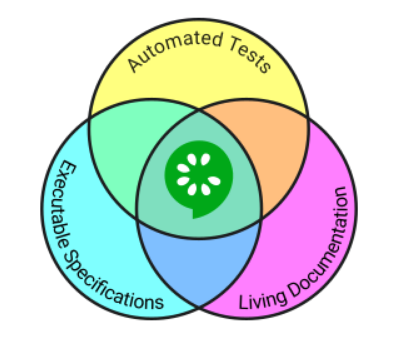
We do this by focusing collaborative work around concrete, real-world examples that illustrate how we want the system to behave. We use those examples to guide us from concept through to implementation, in a process of continuous collaboration.

Read what the creator of BDD introduced about it: <https://dannorth.net/introducing-bdd/>

# Gherkin

## What Is Gherkin?

Gherkin is a set of grammar rules that makes plain text structured enough for Cucumber to understand. The scenario above is written in Gherkin.



## How to Write Gherkin?

|  |  |  |
| --- | --- | --- |
| **Keyword** | **Description** | **Note** |
| Feature  (following by a :) | Provide a **high-level description of a software feature**, and to group related scenarios. | Think of it like a **main feature**. |
| Rule | Represents one business rule. Provides additional info for a feature. Used to **group several scenarios** that belong to a business rule. | Think of it like a **sub feature**. |
| Scenario  (or Example) | Consists of a list of steps.  Follow this same pattern:   * Describe an initial context (Given steps) * Describe an event (When steps) * Describe an expected outcome (Then steps) | Think of it like a **testcase** of the feature. |
| Scenario Outline  (or Scenario Template) | Used to run the same Scenario multiple times, with different combinations of values. | Think of it like a **testcase with different test data**. |
| Scenarios  (or Examples) | A Scenario Outline must contain one or more Scenarios (or Examples) section(s) | Think of it like several **testcases** of the feature. |
| Given | A step.  Used to **describe the initial context of the system**. The purpose is to put the system in a known state before the user (or external system) starts interacting with the system (in the When steps). | Think of it like a **precondition** of the testcase. |
| When | A step.  Used to **describe an event, or an action**. This can be a person interacting with the system, or it can be an event triggered by another system. | Think of it like a **step** in the testcase. |
| Then | A step.  Used to **describe an expected outcome, or result**.  Should use an assertion to compare the actual outcome to the expected outcome. | Think of it like an **expect result** of the testcase. |
| And | A step.  Used to add additional successive Given’s or Then’s. |  |
| But | A step.  Used to add contrast successive Given’s or Then’s. | And and But are different in terms of natual langugage, but Cucumber considers them the same in terms of syntax. |
| Background | If the same **Given steps are repeated in all scenarios** in a Feature, then they can be moved to Background.  Background is also supported at the Rule level. | Think of it like a **setup** of the testcase.  For a teardown, how? |
| \* | Used in place of any of the normal step keywords to effectively add a list of things, where otherwise the natural language of And might not read so elegantly. | Just a way to replace And. But not really helpful :D |
| | | Data Tables.  Useful for passing a list of values to a step definition.  Like Docs Strings, Data Tables is **passed to the step definition as the last argument**. | Just a way to create **columns in test data** table. |
| """ (or ```) | Docs Strings.  Useful for passing a larger piece of text to a step definition.  Like Data Tables, Docs Strings is **passed to the step definition as the last argument**. |  |
| @ | Tag.  Used to control **which groups of scenarios to run**. A great way to organise your features and scenarios.  A tag can only be placed above: Feature, Scenario, Scenario Outline, Examples. | E.g.  Status: @wip, @pending  Speed: @fast, @slow  Testcase type: @normal, @abnormal  Features: @ui, @database, @log |
| # | Comment. The Gherkin compiler will **ignore these texts**.  Only permitted at the start of a new line, anywhere in the feature file. |  |
|  | Free-form descriptions can also be placed underneath Feature, Example/Scenario, Background, Scenario Outline and Rule.  You can write anything you like, as long as no line starts with a keyword.  These description lines are **ignored by Cucumber at runtime, but are available for reporting** (they are included by reporting tools like the official HTML formatter).  Descriptions can be in the form of Markdown - formatters including the official HTML formatter support this. | E.g.  Feature: Guess the word  The word guess game is a turn-based game for two players.  The Maker makes a word for the Breaker to guess. The game is over when the Breaker guesses the Maker's word.  Example: Maker starts a game |

<https://cucumber.io/docs/gherkin/reference/>

## Example

This example covers all keywords mentioned above:

# Feature: User login process

# This feature covers the entire login process for users, including successful login, invalid login attempts, and password reset scenarios.

@LoginFeature

Feature: User login process

  # Background section sets up preconditions common to all scenarios in this feature

  Background:

    Given the user is on the login page

    And the user has a valid username and password

  # Rule for successful login scenarios

  Rule: Successful login

    Scenario: User logs in successfully with valid credentials

      When the user enters their username and password

      Then the user should be redirected to the dashboard

      And a welcome message should be displayed

    Scenario: User tries to log in with invalid credentials

      When the user enters an invalid username and password

      Then an error message should be displayed

"""

Invalid username or password. Please try again.

"""

      But the user should remain on the login page

  # Rule for password reset scenarios

  Rule: Password reset

    Scenario Outline: User requests a password reset

      Given the user has forgotten their password

      When the user clicks on the "Forgot Password" link

      And provides their registered email "<email>"

      Then a password reset email should be sent to "<email>"

    Examples:

      | email              |

      | user1@example.com  |

      | user2@example.com  |

      | user3@example.com  |

  Scenario: User cancels password reset

      Given the user has forgotten their password

      When the user clicks on the "Forgot Password" link

      And decides to cancel the process

      Then the user should return to the login page

      But no password reset email should be sent

Scenario: User receives help with password issues

Given the user is on the login page

When the user clicks on the "Need Help?" link

Then the following information should be displayed:

"""

If you forgot your password, click on 'Forgot Password'.

If you need further assistance, contact support at support@example.com.

"""

## How to Write Better Gherkin?

### Less Detailed

Your scenarios should **describe the intended behaviour of the system, not the implementation**. In other words, it should **describe WHAT, not HOW**.

|  |  |
| --- | --- |
| **DO** | **DON'T** |
| Feature: Subscribers see different articles based on their subscription level    Scenario: Free subscribers see only the free articles    Given Free Frieda has a free subscription    When Free Frieda logs in with her valid credentials    Then she sees a Free article  Scenario: Subscriber with a paid subscription can access both free and paid articles    Given Paid Patty has a basic-level paid subscription    When Paid Patty logs in with her valid credentials    Then she sees a Free article and a Paid article | Feature: Subscribers see different articles based on their subscription level  Scenario: Free subscribers see only the free articles    Given users with a free subscription can access "FreeArticle1" but not "PaidArticle1"    When I type "freeFrieda@example.com" in the email field    And I type "validPassword123" in the password field    And I press the "Submit" button    Then I see "FreeArticle1" on the home page    And I do not see "PaidArticle1" on the home page  Scenario: Subscriber with a paid subscription can access "FreeArticle1" and "PaidArticle1"    Given I am on the login page    When I type "paidPattya@example.com" in the email field    And I type "validPassword123" in the password field    And I press the "Submit" button    Then I see "FreeArticle1" and "PaidArticle1" on the home page |

Having less details brings some benefits:

* **Improving readbility**: You don’t want yourself and others having to read a lengthy test documents. You just want to focus on the key steps and necessary conditions.
* **Easier to maintance**: Less effort for modification and maintanance in case of personal mistakes or requirement changes.

## Gherkin Grammer

The following section provides a short overview of the hierarchical containment that is possible in the Gherkin grammer:

# -- SIMPLIFIED GHERKIN GRAMMAR (for Gherkin v6):

# CARDINALITY DECORATOR: '\*' means 0..N (many), '?' means 0..1 (optional)

# EXAMPLE: Feature

#   A Feature can have many Tags (as TaggableStatement: zero or more tags before its keyword).

#   A Feature can have an optional Background.

#   A Feature can have many Scenario(s), meaning zero or more Scenarios.

#   A Feature can have many ScenarioOutline(s).

#   A Feature can have many Rule(s).

Feature(TaggableStatement):

    Background?

    Scenario\*

    ScenarioOutline\*

    Rule\*

Background:

    Step\*           # Background steps are injected into any Scenario of its scope.

Scenario(TaggableStatement):

    Step\*

ScenarioOutline(ScenarioTemplateWithPlaceholders):

    Scenario\*       # Rendered Template by using ScenarioOutline.Examples.rows placeholder values.

Rule(TaggableStatement):

    Background?     # Behave-specific extension (after removal from final Gherkin v6).

    Scenario\*

    ScenarioOutline\*

# Cucumber

## Step

A step is analogous to a method call or function invocation. It can be anything **starting with a preposition or an adverb – Given, When, Then, And, But**.

A step in Cucumber is the same as a step in Gherkin.

For example:

Given I have 2 cukes in my belly

Steps are declared in your .feature files.

## Step Arguments

In above example, Cucumber extracts the text 2 from the step, converts it to an int and passes it as an argument to the step definition's method.

The number of parameters in the method has to match the number of capture groups in the expression. If there is a mismatch, Cucumber will throw an error.

## Step Definition

A Step Definition is a **method or function with an expression that links it to one or more Gherkin steps**. When Cucumber executes a Gherkin step in a scenario, it will look for a matching step definition to execute.

To illustrate how this works, look at the following Gherkin scenario:

Scenario: Some cukes

Given I have 2 cukes in my belly

In Java, the step definition for the step "I have 48 cukes in my belly" is:

package com.example;

import io.cucumber.java.en.Given;

public class StepDefinitions {

    @Given("I have {int} cukes in my belly")

    public void i\_have\_n\_cukes\_in\_my\_belly(int cukes) {

        System.out.format("Cukes: %n\n", cukes);

    }

}

The function i\_have\_n\_cukes\_in\_my\_belly is nothing special, except that it has a decorator @Given. Cucumber calls it a **step definition's expression**.

### Expression

There are two ways to express a step definition:

#### Using Cucumber Expression

In the above example, we use Cucumber expression. You can see that when the text is matched against that expression, the number 2 is extracted from the {int} output parameter and passed as an argument to the step definition.

The following text would NOT match the expression because the number doesn't fit into an int:

I have 2.5 cukes in my belly

Change to float instead and it will work:

I have {float} cucumbers in my belly

##### Parameter Types

Part in the curly bracket {} is called paramter type.

**Built-In Type**

|  |  |
| --- | --- |
| **Parameter Type** | **Description** |
| {int} | Matches **integers**, for example 71 or -19. Converts to a 32-bit signed integer if the platform supports it. |
| {float} | Matches **floats**, for example 3.6, .8 or -9.2. Converts to a 32 bit float if the platform supports it. |
| {word} | Matches **words** without whitespace, for example banana (but not banana split). |
| {string} | Matches **single-quoted or double-quoted strings**, for example "banana split" or 'banana split' (but not banana split). Only the text between the quotes will be extracted. The quotes themselves are discarded. Empty pairs of quotes are valid and will be matched and passed to step code as empty strings. |
| {} anonymous | Matches **anything** (/.\*/). |
| {bigdecimal} | Matches the same as {float}, but converts to a BigDecimal if the platform supports it. |
| {double} | Matches the same as {float}, but converts to a 64 bit float if the platform supports it. |
| {biginteger} | Matches the same as {int}, but converts to a BigInteger if the platform supports it. |
| {byte} | Matches the same as {int}, but converts to an 8 bit signed integer if the platform supports it. |
| {short} | Matches the same as {int}, but converts to a 16 bit signed integer if the platform supports it. |
| {long} | Matches the same as {int}, but converts to a 64 bit signed integer if the platform supports it. |

**Custom Types**

Cucumber Expressions can be extended so they automatically convert output parameters to your own types. Consider this expression:

I have a {color} ball

If we want the {color} output parameter to be converted to a Color object, we can define a custom parameter type in Cucumber's [configuration](https://cucumber.io/docs/cucumber/configuration).

##### Optional Text

It's grammatically incorrect to say 1 cucumbers, so we should make the plural s optional. That can be done by **surrounding the optional text with parentheses**:

I have {int} cucumber(s) in my belly

That expression would match either of those textx:

I have 1 cucumber in my belly

I have 42 cucumbers in my belly

In Regular Expressions, parentheses indicate a capture group, but in Cucumber Expressions they mean *optional text*.

##### Alternative Text

Sometimes you want to simply your language. For example:

I have {int} cucumber(s) in my belly/stomach

This would match either of those texts:

I have 1 cucumber in my belly

I have 1 cucumber in my stomach

I have 42 cucumbers in my stomach

I have 42 cucumbers in my belly

Alternative text **only works when there is no whitespace between the alternative parts**.

##### Escaping

If you ever need to match () or {} literally, you can escape the opening tag with a **backslash** \:

I have {int} \{what} cucumber(s) in my belly \(amazing!)

This expression would match:

I have 1 {what} cucumber in my belly (amazing!)

I have 42 {what} cucumbers in my belly (amazing!)

You may have to escape the \ character itself with another \. For example, in Java, you have to use escape character \ with another backslash.

I have {int} \\{what} cucumber(s) in my belly \\(amazing!)

I have 1 \{what} cucumber in my belly \(amazing!)

I have 42 \{what} cucumbers in my belly \(amazing!)

The / character will always be interpreted as an alternative, unless escaped, such as with \/.

#### Using Regular Expression

The above example can be re-written to:

package com.example;

import io.cucumber.java.en.Given;

public class StepDefinitions {

    @Given("^I have {\\d+} cukes in my belly$")

    public void i\_have\_n\_cukes\_in\_my\_belly(int cukes) {

        System.out.format("Cukes: %n\n", cukes);

    }

}

The rules here are:

* Each capture group from the match will be passed as arguments to the step definition’s method.
* If the capture group expression is identical to one of the registered parameter types’s regexp, the captured string will be transformed before it's passed to the step definition’s method. In the example above, the cukes argument will be an integer, because the built-in int parameter type’s regexp is \d+ .

### Scope

Step definitions aren’t linked to a particular feature file or scenario, but it **links to the expression**. In other words, Cucumber step definitions have a **global scope**.

The advantage of global scope is that this allows step definitions to be reused across different feature files without being restricted by their location in the project structure. However, the disadvantage is that it **can cause conflicts** if there are similar step definition's expressions across scenarios.

The best way to **prevent these conflicts is to name your expression very specificly**. Another effective technique to **separate your features into different folders**.

The following example separates the big feature into sub features. **When you run Cucumber, go to the specific feature’s folder (featureA or featureB)** and execute the cucumber command:

features

    ├───featureA

    │   │   featureA1.feature

    │   │   featureA2.feature

    │   └───steps

    │           featureA1.java

    │           featureA2.java     # featureA1.java and featureA2.java CANNOT have two step definitions with the same expression

    │

    └───featureB

        │   featureB.feature

        └───steps

                featureB.java      # but featureB.java and featureA2.java CAN have two step definitions with the same expression

Also, note that keywords are not taken into account when looking for a step definition. This means you **cannot have a Given, When, Then, And or But step with the same text as another step**.

## Execution Flow

Following is what Cucumber does:

* Matches a Gherkin step against a step definition’s Regexp
* Gathers any capture groups or variables in the step definition.
* Passes them to the step definition’s method and executes it.

All step definitions are loaded (and defined) before Cucumber starts to execute the plain text in the feature file.

## Step Results

Each step can have one of the following results:

### Success

When Cucumber finds a matching step definition, it will execute it. If the step definition's method **doesn’t raise an error**, the step is marked as successful (green). Anything you return from a step definition has no significance whatsoever.

### Undefined

When Cucumber **can’t find a matching step definition**, the step gets marked as undefined (yellow), and all subsequent steps in the scenario are skipped.

### Pending

When a step definition’s method invokes the pending method, the step is marked as pending (yellow), indicating that you have work to do.

### Failed Steps

When a step definition’s method is executed and **raises an error**, the step is marked as failed (red). What you return from a step definition has no significance whatsoever.

### Skipped

Steps that follow undefined, pending, or failed steps are never executed, even if there is a matching step definition. These steps are marked as skipped (cyan).

### Ambiguous

Step definitions have to be unique for Cucumber to know what to execute. If you use ambiguous step definitions, Cucumber will raise an AmbiguousStepDefinitionsException, telling you to fix the ambiguity.

## Hooks (or Environment Controls)

Hooks are blocks of **code that can run at various points** in the Cucumber execution cycle. They are typically used for **setup** and **teardown** of the environment before and after each scenario.

Where a hook is defined has no impact on what scenarios or steps it is run for. If you want more fine-grained control, you can use conditional hooks.

You **can declare hooks in any class**.

Cucumber supports various types of hooks:

|  |  |  |  |
| --- | --- | --- | --- |
| **Hook** | **Description** | **Syntax in Java** | **Note** |
| Before | Run before the first step of each scenario. | @Before  public void doSomethingBefore() {  } | Whatever happens in a Before hook is invisible to people who only read the features.  Consider using a Background in feature file if the setup should be readable by non-technical people. And only use a Before hook for low-level logic such as starting a browser or deleting data from a database. |
| After | Run after the last step of each scenario, even when the step result is failed, undefined, pending, or skipped. | @After  public void doSomethingAfter() {  } |  |
| BeforeStep | Run before a step. | @BeforeStep  public void doSomethingBefore() {  } |  |
| AfterStep | Run after a step. | @AfterStep  public void doSomethingAfter() {  } |  |
| BeforeAll | Run before any scenario is run. | @BeforeAll  public void beforeAll() {  } |  |
| Afterall | Run after all scenarios have been executed. | @AfterAll  public void afterAll() {  } |  |

## Tags

Tags are a great way to organise your features and scenarios. They can be used for different purposes:

* Running a subset of scenarios (in example 1)
* Restricting hooks to a subset of scenarios (in example 2)
* Link to other documents (in example 3)
* Etc.

**Example 1**: The following example defines data sets for mobile and desktop platform. Usually, depending on the platform you’re testing on, you only need to execute tests on either mobile or desktop, not both. With the use of tags, you can easily select which test data to use:

Scenario Outline: Steps will run conditionally if tagged

  Given user is logged in

  When user clicks <link>

  Then user will be logged out

  @mobile

  Examples:

    | link                  |

    | logout link on mobile |

  @desktop

  Examples:

    | link                   |

    | logout link on desktop |

Your Cucumber command will be:

$ cucumber --tags @mobile

$ cucumber --tags @desktop

$ cucumber --tags "not @desktop"

$ cucumber --tags "not @mobile"

**Example 2**:

**Example 3**:

Tags can refer to IDs in external systems such as requirements, issue trackers or test tools:

@BJ-x98.77 @BJ-z12.33

Feature: Convert transaction

You **can use a custom Cucumber reporting plugin that will turn tags into links** pointing to documents in your external tool.

Another creative way to use tags is to keep track of where in the development process a certain feature is:

@qa\_ready

Feature: Index projects

Note that you can add underscope (\_) to tags, but cannot spaces.

Also, Cucumber commands allow to **combine multiple tags in one command** using boolean expression:

$ cucumber --tags="@wip or @slow"

$ cucumber --tags="@wip and @slow"

$ cucumber --tags="@wip and not @slow"

## Test Execution

How Cucumber finds your features and step definitions? Regardless of the directory structure, Cucumber effectively flattens the features/ directory tree when running tests. Anything ending in .java, .js, rb, etc. (in short, programming languages that Cucumber supports) inside that directory will be treated as a step definition.

## Test Reporting

[Reporting - Cucumber Documentation](https://cucumber.io/docs/cucumber/reporting/?lang=java)

## Programming Language Supports

<https://cucumber.io/docs/installation/>

## VSCode Extension Supports

## [Cucumber](https://marketplace.visualstudio.com/items?itemName=CucumberOpen.cucumber-official)

# External Library

## JUnit Report To HTML

Convert JUnit reports geneated by Cucumber to HTML:

[GitHub - inorton/junit2html: Turn Junit XML reports into self contained HTML reports](https://github.com/inorton/junit2html)