What is Cucumber?

Cucumber is a testing approach which supports Behaviour Driven Development (BDD). It explains the behaviour of the application in a simple English text using Gherkin language.

Cucumber tool is based on the Behaviour Driven Development framework that **acts as the bridge** between the following people:

1. Software Engineer and Business Analyst.
2. Manual Tester and Automation Tester.
3. Manual Tester and Developers.

Cucumber also **benefits the client to understand the application code** as it uses Gherkin language which is in Plain Text. Anyone in the organization can understand the behaviour of the software. The syntax's of Gherkin is in simple text which is readable and understandable.

Cucumber is a software tool used by computer programmers for testing other software. It runs automated acceptance tests written in a behavior-driven development (BDD) style. Central to the Cucumber BDD approach is its plain language parser called Gherkin. It allows expected software behaviours to be specified in a logical language that customers can understand. As such, Cucumber allows the execution of feature documentation written in business-facing text.

Cucumber was originally written in the Ruby programming language. and was originally used exclusively for Ruby testing as a complement to the RSpec BDD framework. Cucumber now supports a variety of different programming languages through various implementations, including Java For example, Cuke4php and Cuke4Lua are software bridges that enable testing of PHP and Lua projects, respectively. Other implementations may simply leverage the Gherkin parser while implementing the rest of the testing framework in the target language.

Gherkin language

Gherkin is the language that Cucumber uses to define test cases. It is designed to be non-technical and human readable, and collectively describes use cases relating to a software system. The purpose behind Gherkin's syntax is to promote Behavior Driven Development practices across an entire development team, including business analysts and managers.It seeks to enforce firm, unambiguous requirements starting in the initial phases of requirements definition by business management and in other stages of the development lifecycle.

In addition to providing a script for automated testing, Gherkin's natural language syntax is designed to provide simple documentation of the code under test.Gherkin currently supports keywords in dozens of languages.

Syntax

All Gherkin files have the `.feature` file extension.

Features, Scenarios, and Steps

Cucumber tests are divided into individual Features. These Features are subdivided into Scenarios, which are sequences of Steps.

Features

A feature is a Use Case that describes a specific function of the software being tested. There are three parts to a Feature.

- The `Feature:` keyword
- The Feature name (on the same line as the keyword)
- An optional description on the following lines

Example Feature definition

```
Feature: Withdraw Money from ATM

  A user with an account at a bank would like to withdraw money from an ATM.

  Provided he has a valid account and debit or credit card, he should be allowed to make the
transaction. The ATM will tend the requested amount of money, return his card, and
subtract amount of the withdrawal from the user's account.

    Scenario: Scenario 1
      Given preconditions
      When actions
      Then results

    Scenario: Scenario 2
      ...
```

Scenarios

Each Feature is made of a collection of scenarios. A single scenario is a flow of events through the Feature being described and maps 1:1 with an executable test case for the system. Keeping with the example ATM withdrawal feature, a scenario might describe how a user requests money and what happens to their account.

```
Scenario: Eric wants to withdraw money from his bank account at an ATM
  Given Eric has a valid Credit or Debit card
  And his account balance is $100
  When he inserts his card
  And withdraws $45
  Then the ATM should return $45
  And his account balance is $55
```

In some cases, one might want to test multiple scenarios at once to perform Equivalence partitioning and Boundary-value analysis. A `Scenario Outline` provides a technique to specify multiple examples to test against a template scenario by using placeholders. For example,

```
Scenario Outline: A user withdraws money from an ATM
  Given <Name> has a valid Credit or Debit card
  And their account balance is <OriginalBalance>
  When they insert their card
  And withdraw <WithdrawalAmount>
  Then the ATM should return <WithdrawalAmount>
  And their account balance is <NewBalance>

  Examples:
   | Name   | OriginalBalance | WithdrawalAmount | NewBalance |
   | Eric   | 100             | 45               | 55         |
   | Gaurav | 100             | 40               | 60         |
   | Ed     | 1000            | 200              | 800        |
```

At runtime the scenario is run against each row in the table. Column values are substituted for each of the named placeholders in the scenario.

Steps

The crux of a Scenario is defined by a sequence of Steps outlining the preconditions and flow of events that will take place. The first word of a step is a keyword, typically one of

- `Given` - Describes the preconditions and initial state before the start of a test and allows for any pre-test setup that may occur
- `When` - Describes actions taken by a user during a test
- `Then` - Describes the outcome resulting from actions taken in the When clause

Occasionally, the combination of Given-When-Then uses other keywords to define conjunctions

- `And` - Logical and
- `But` - Logically the same as `And`, but used in the negative form

```
Scenario: A user attempts to withdraw more money than they have in their account
  Given John has a valid Credit or Debit card
  And his account balance is $20
  When he inserts his card
  And withdraws $40
  Then the ATM displays an error
  And returns his card
  But his balance remains $20
```

Hooks

Hooks are Cucumber's way of allowing for setup to be performed prior to tests being run and teardown to be run afterwards. They are defined as executable Ruby blocks, similar to JUnit methods marked with @Before, @After annotations. Conventionally they are placed under support/ and are applied globally. Three basic types of hooks exist

Before - Runs before a scenario
After - Runs after a scenario
Around - Assumes control and runs around a scenario

Additional hooks include

BeforeStep
AfterStep
AfterConfiguration - Runs after Cucumber configuration and is passed an instance of the configuration

Features of BDD

1. *Shifting from thinking in "tests" to thinking in "behaviour"*
2. *Collaboration between Business stakeholders, Business Analysts, QA Team and developers*
3. *Ubiquitous language, it is easy to describe*
4. *Driven by Business Value*
5. *Extends Test Driven Development (TDD) by utilizing natural language that non-technical stakeholders can understand*
6. *BDD frameworks such as Cucumber or JBehave are an enabler, acting a "bridge" between Business & Technical Language*

Example of a Cucumber or BDD Test:

The main feature of the Cucumber is that it focuses on Acceptance testing. It made it easy for anyone in the team to read and write test and with this feature it brings business users in to the test process, helping teams to explore and understand requirements.

***Feature**: Sign up*
Sign up should be quick and friendly.
***Scenario**: Successful sign up*
New users should get a confirmation email and be greeted personally by the site once signed in.
***Given** I have chosen to sign up*
***When** I sign up with valid details*
***Then** I should receive a confirmation email*
***And** I should see a personalized greeting message*
***Scenario**: Duplicate email*
Where someone tries to create an account for an email address that already exists.
***Given** I have chosen to sign up*
***But** I enter an email address that has already registered*
***Then** I should be told that the email is already registered*
***And** I should be offered the option to recover my password*

Test Driven Development (TDD)

Test-driven development (**TDD**) is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards. Kent Beck, who is credited with having developed or 'rediscovered' the technique, stated in 2003 that TDD encourages simple designs and inspires confidence