Cucumber Basic Initial Structure :

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

Video-3: Naveen

#### Summary

In this tutorial, Naveen introduces the basics of setting up a Cucumber framework with Java for behaviour-driven development (BDD) testing.

Concepts in Cucumber Explained and talked below:

#Author: your.email@your.domain.com

#Keywords Summary :

#Feature: List of scenarios.

#Scenario: Business rule through list of steps with arguments.

#Given: Some precondition step

#When: Some key actions

#Then: To observe outcomes or validation

#And,But: To enumerate more Given,When,Then steps

#Scenario Outline: List of steps for data-driven as an Examples and <placeholder>

#Examples: Container for s table

#Background: List of steps run before each of the scenarios

#""" (Doc Strings)

#| (Data Tables)

#@ (Tags/Labels):To group Scenarios

#<> (placeholder)

#""

## (Comments)

#Sample Feature Definition Template

#### Highlights

- 📑 \*\*BDD Introduction:\*\* Naveen explains the BDD concept, involving user stories, collaboration among PO, dev, and QA, and the creation of feature files with scenarios.

- 📋 \*\*Feature Files:\*\* Feature files are designed using Gherkin syntax, containing scenarios, steps, and keywords like Given, When, Then. Multiple features can be created for different functionalities.

- 🤖 \*\*Step Definitions:\*\* Step definitions are written in Java or other programming languages, mapping to the steps in feature files. Naveen emphasizes the importance of one-to-one mapping between feature files and step definitions.

- 🚀 \*\*Utility Methods:\*\* Step methods call utility methods for actions like interacting with Selenium, databases, or APIs. This allows reusability and separation of concerns.

- 🏗️ \*\*Framework Components:\*\* Naveen outlines the four main components - User Story & Feature Files, Step Definitions, Utilities, and Test Runner. These are crucial for creating a Cucumber-based framework.

- 🛠️ \*\*Maven Project Setup:\*\* Naveen guides through setting up a Maven project in Eclipse, adding Cucumber dependencies, and installing the Cucumber Eclipse plugin for enhanced development features.

- 🎯 \*\*Test Runner:\*\* The Test Runner class is introduced, acting as the entry point for test execution, defining paths to feature files, step definitions, and other configurations.

This tutorial provides a comprehensive overview for beginners to start implementing Cucumber with Java for BDD test automation.

#### Summary

This tutorial by Naveen introduces Cucumber, a tool for behavior-driven development (BDD) testing using Java. The process involves creating feature files with test scenarios, mapping them to Java code using step definitions, and executing tests through a Test Runner.

#### Highlights

- 📑 \*\*BDD Introduction:\*\* BDD is a collaborative approach involving Product Owners (PO), developers, and QA. They create user stories in the form of feature files, outlining various scenarios.

```gherkin

Feature: Login

Scenario: Successful Login

Given the user is on the login page

When the user enters valid credentials

Then the user should be logged in successfully

```

- 📋 \*\*Feature Files:\*\* Feature files use Gherkin syntax with Given, When, Then keywords. They describe features and scenarios. Multiple features can exist for different functionalities.

- 🤖 \*\*Step Definitions:\*\* Step definitions are Java methods that map to steps in feature files. They define the actions to be taken for each step.

```java

public class StepDefinitions {

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

public void userIsOnLoginPage() {

// Implementation for navigating to the login page

}

@When("the user enters valid credentials")

public void userEntersValidCredentials() {

// Implementation for entering valid credentials

}

@Then("the user should be logged in successfully")

public void userLoggedInSuccessfully() {

// Implementation for verifying successful login

}

}

```

- 🚀 \*\*Utility Methods:\*\* Step methods can call utility methods for common actions like interacting with a web page, databases, or APIs. This promotes code reusability.

```java

public class WebUtils {

public static void clickElement(By locator) {

// Implementation for clicking a web element

}

public static void waitForElementVisible(By locator) {

// Implementation for waiting until an element is visible

}

}

```

- 🏗️ \*\*Framework Components:\*\* The framework consists of User Stories & Feature Files, Step Definitions, and Utilities. These components work together to create a robust testing framework.

- 🛠️ \*\*Maven Project Setup:\*\* Naveen demonstrates setting up a Maven project in Eclipse, adding necessary dependencies, and installing the Cucumber Eclipse plugin.

- 🎯 \*\*Test Runner:\*\* The Test Runner class is crucial. It defines where feature files and step definitions are located, and it orchestrates the execution of tests.

```java

@RunWith(Cucumber.class)

@CucumberOptions(

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

glue = "com.example.stepdefinitions"

)

public class TestRunner {

}

```

This tutorial equips beginners to implement BDD testing using Cucumber with Java, breaking down complex concepts into understandable code snippets.

Cucumber-Java -Maven POM.xml Configuration

|  |  |
| --- | --- |
| Properties Configuration | <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <java.version>1.8</java.version>  </properties> |
| Dependency | <!-- https://mvnrepository.com/artifact/io.cucumber/cucumber-java -->  <dependency>  <groupId>io.cucumber</groupId>  <artifactId>cucumber-java</artifactId>  <version>7.14.1</version>  </dependency>  <!-- https://mvnrepository.com/artifact/junit/junit -->  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.13.2</version>  <scope>test</scope>  </dependency>  <!-- https://mvnrepository.com/artifact/io.cucumber/cucumber-junit -->  <dependency>  <groupId>io.cucumber</groupId>  <artifactId>cucumber-junit</artifactId>  <version>7.14.1</version>  <scope>test</scope>  </dependency> |
| Build | <build>  <plugins>  <plugin> <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <version>3.11.0</version>  <configuration>  <encoding>UTF-8</encoding> <source>${java.version}</source>  <target>${java.version}</target>  </configuration>  </plugin>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-surefire-plugin</artifactId>  <version>3.2.2</version>  <configuration>  <properties>  <!-- Work around. Surefire does not include enough  information to disambiguate between different  examples and scenarios. -->  <configurationParameters> cucumber.junit-platform.naming-strategy=long  </configurationParameters>  </properties>  </configuration>  </plugin>  </plugins>  </build> |

# #### Cucumber Feature File mapping with Step Definition - Important Tips:

Summary

#### Highlights

- [📂] Refactoring project structure: Move feature files to `src/test/resources` and step definitions to `src/test/java`.

- [🔧] Download and update Cucumber Eclipse Plugin for proper integration.

- [🚨] Flagging missing step definitions: Instant feedback on steps without corresponding code.

- [🔍] Easily navigate from feature file to step definition using the plugin.

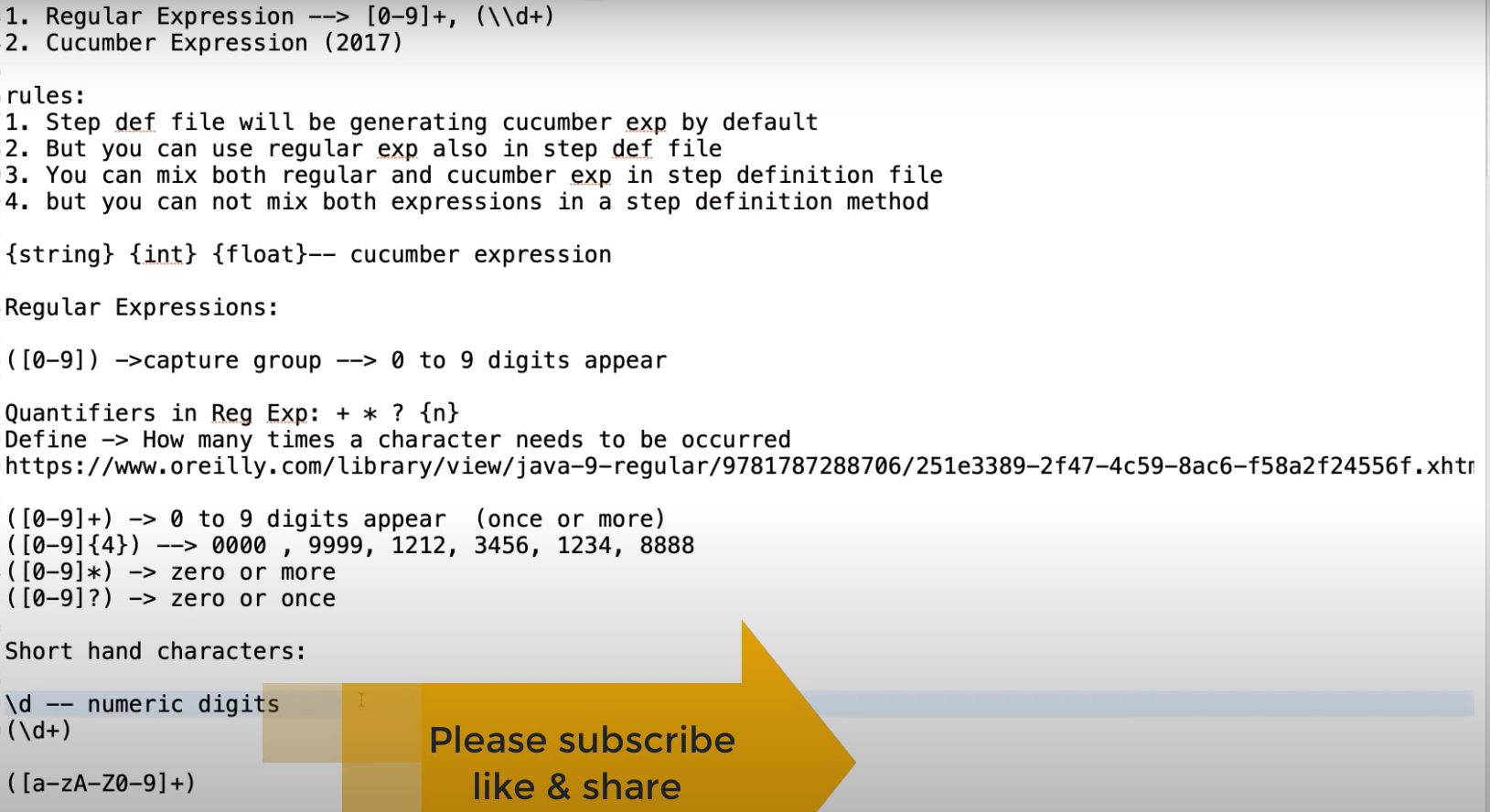
- [📝] Writing feature file with scenarios: Example of Uber booking scenario.

- [🚦] Automatic step suggestion: Get suggestions for step definitions when executing feature files.

- [✅] Successful mapping: Ensure all steps are mapped by converting the project to a Cucumber project and addressing warnings.

Regular Expression in Cucumber :

# Cucumber Expression vs Regular Expression:



#### Summary

In this tutorial, Navina discusses the distinctions between Cucumber Expressions and Regular Expressions in Cucumber 6.

#### Highlights

- 💡 Cucumber supports two types of expressions: Cucumber Expression and Regular Expression.

- 💻 By default, Cucumber generates step definitions using Cucumber Expression, not Regular Expression.

- 🔄 While you can use both expressions in the same step definition file, you cannot mix them within the same step definition method.

- 🚫 In a single step definition method, using both Cucumber Expression and Regular Expression together is not allowed.

- 🧠 Regular Expressions are powerful for extracting values from strings; examples include capturing links from email bodies or extracting transaction IDs.

- 🤖 Quantifiers in Regular Expressions, such as `+`, `\*`, and `?`, define how many times a character should occur.

- 📚 Navina recommends referring to official documentation for more Cucumber Expressions and using shorthand characters like `\d` for digits in Regular Expressions.

#### Summary

In this tutorial, Navina explains the difference between two ways of expressing things in Cucumber (a tool for testing software). She talks about Cucumber Expressions and Regular Expressions, and how they affect the way you write instructions for your tests.

#### Highlights

- 💡 \*\*Two Expression Types:\*\* There are two ways to tell Cucumber what to do - Cucumber Expression and Regular Expression.

- 💻 \*\*Default Behavior:\*\* When you tell Cucumber what to do, it usually uses Cucumber Expression automatically.

- 🔄 \*\*Mixing Expressions:\*\* You can use both Cucumber Expression and Regular Expression in the same set of instructions. However, there are some rules to follow.

- 🚫 \*\*No Mixing in a Single Instruction:\*\* You can't use both types of expressions within one specific instruction. For example, if you're saying something about a number, you can't mix two ways of saying it in the same sentence.

- 🧠 \*\*Power of Regular Expressions:\*\* Regular Expressions are like powerful codes that help you find specific things in a bunch of text. For instance, imagine you get an email with a lot of text, and you want to find a link in that text. Regular Expressions can help you do that.

- 🤖 \*\*Quantifiers:\*\* Think of quantifiers as instructions on how many times something should happen. For instance, if you want to find a series of numbers that are three digits long, you might use a quantifier to tell the computer that.

- 📚 \*\*Shortcut Codes:\*\* Navina suggests using codes like `\d` when you want to talk about numbers in your instructions. These are like shortcuts that make it easier for you to tell the computer what you're looking for.

In simpler terms, Navina is helping us understand how to tell a computer what we want it to do when we're testing our software, and she's giving us different 'languages' we can use to do that. She also explains some rules about how we can use these languages together.

# Tags in Cucumber:

#### Summary

In this video, Navinya discusses the importance of tags in the Cucumber BDD framework. Tags allow for the selective execution of test cases based on criteria such as smoke tests, regression tests, or production-related tests. The tutorial covers how to apply tags at different levels (feature, scenario) and demonstrates their use in executing specific scenarios.

#### Highlights

- 🏷️ Tags in Cucumber facilitate the selective execution of test cases based on criteria like smoke, regression, or production.

- 🔄 Tags can be applied at the feature level, scenario level, or even for scenario outlines with examples.

- 📝 Tags enable documentation, and their systematic use aids in categorizing and executing scenarios efficiently.

- 🚀 The tutorial illustrates the process of creating a runner file and applying tags for execution.

- 🤖 Tags can be combined using logical operators like 'and' and 'or' to execute scenarios meeting multiple criteria.

- 🛠️ Tags are not limited to the JUnit runner; they can also be applied through Maven commands, allowing runtime flexibility.

- 🌐 Tags can be utilized for documentation and reporting, linking to external tools like Jira or Zephyr Quality Center.

**Steps to Revise Tags:**

### Cucumber Tags: Step-by-Step Revision

#### 1. \*\*Introduction to Tags\*\*

- 🏷️ Tags in Cucumber help selectively execute test cases based on specific criteria.

- 🎯 Criteria examples: Smoke tests, regression tests, production-related tests.

#### 2. \*\*Applying Tags at Different Levels\*\*

- Feature level, Scenario level, or Scenario Outline with examples.

- Example: `@smoke`, `@regression`, `@prod`.

#### 3. \*\*Creating a Runner File\*\*

- Illustration using the example of an Uber test scenario.

- Copying and pasting scenarios in the runner file for execution.

#### 4. \*\*Executing Scenarios Based on Tags\*\*

- Using the `tags` option in the JUnit runner.

- Example: `tags = "@smoke"`, `tags = "@regression"`.

#### 5. \*\*Logical Operators for Tag Combination\*\*

- Using 'and' and 'or' to combine tags.

- Example: `tags = "@smoke and @regression"`, `tags = "@smoke or @regression"`.

#### 6. \*\*Applying Tags through Maven Commands\*\*

- Executing tests using Maven commands.

- Example: `mvn test -Dcucumber.filter.tags="@smoke and @regression"`.

#### 7. \*\*Runtime Overwriting of Tags\*\*

- Overwriting tags during runtime.

- Example: `mvn test -Dcucumber.filter.tags="@prod"`.

#### 8. \*\*Documentation and Reporting with Tags\*\*

- Utilizing tags for documentation purposes.

- Example: Adding tags like `@qa\_ready`, `@user\_story\_1001`, `@epic\_a100\_quality900`.

#### 9. \*\*Integration with External Tools\*\*

- Linking tags to external tools (e.g., Jira, Zephyr Quality Center) for better reporting.

- Example: Associating tags with user stories and epics.

#### 10. \*\*Practice and Git Repository\*\*

- Encouragement to practice using the provided Git repository.

- Checking in code for reference and addressing issues through comments.

#### Conclusion:

- Tags in Cucumber provide a powerful mechanism for organizing, documenting, and selectively executing scenarios.

- 🔄 \*\*Practice Tip:\*\* Use the Git repository for hands-on experience and refer to examples for better understanding.

- 🔗 \*\*Integration:\*\* Leverage tags for seamless integration with external tools, enhancing reporting capabilities.

Happy revising! 🚀

# Background in Cucumber:

Occasionally you’ll find yourself repeating the same Given steps in all of the scenarios in a Feature.

Since it is repeated in every scenario, this is an indication that those steps are not essential to describe the scenarios; they are incidental details. You can literally move such Given steps to the background, by grouping them under a Background section.

A Background allows you to add some context to the scenarios that follow it. It can contain one or more Given steps, which are run before each scenario, but after any [Before hooks](https://cucumber.io/docs/cucumber/api/#hooks).

A Background is placed before the first Scenario/Example, at the same level of indentation.

For example:

Feature: Multiple site support

Only blog owners can post to a blog, except administrators,

who can post to all blogs.

Background:

Given a global administrator named "Greg"

And a blog named "Greg's anti-tax rants"

And a customer named "Dr. Bill"

And a blog named "Expensive Therapy" owned by "Dr. Bill"

Scenario: Dr. Bill posts to his own blog

Given I am logged in as Dr. Bill

When I try to post to "Expensive Therapy"

Then I should see "Your article was published."

Scenario: Dr. Bill tries to post to somebody else's blog, and fails

Given I am logged in as Dr. Bill

When I try to post to "Greg's anti-tax rants"

Then I should see "Hey! That's not your blog!"

Scenario: Greg posts to a client's blog

Given I am logged in as Greg

When I try to post to "Expensive Therapy"

Then I should see "Your article was published."

Background is also supported at the Rule level, for example:

Feature: Overdue tasks

Let users know when tasks are overdue, even when using other

features of the app

Rule: Users are notified about overdue tasks on first use of the day

Background:

Given I have overdue tasks

Example: First use of the day

Given I last used the app yesterday

When I use the app

Then I am notified about overdue tasks

Example: Already used today

Given I last used the app earlier today

When I use the app

Then I am not notified about overdue tasks

...

You can only have one set of Background steps per Feature or Rule. If you need different Background steps for different scenarios, consider breaking up your set of scenarios into more Rules or more Features.

For a less explicit alternative to Background, check out [conditional hooks](https://cucumber.io/docs/cucumber/api/#conditional-hooks).

## Tips for using Background

* Don’t use Background to set up **complicated states**, unless that state is actually something the client needs to know.
  + For example, if the user and site names don’t matter to the client, use a higher-level step such as Given I am logged in as a site owner.
* Keep your Background section **short**.
  + The client needs to actually remember this stuff when reading the scenarios. If the Background is more than 4 lines long, consider moving some of the irrelevant details into higher-level steps.
* Make your Background section **vivid**.
  + Use colourful names, and try to tell a story. The human brain keeps track of stories much better than it keeps track of names like "User A", "User B", "Site 1", and so on.
* Keep your scenarios **short**, and don’t have too many.
  + If the Background section has scrolled off the screen, the reader no longer has a full overview of what’s happening. Think about using higher-level steps, or splitting the \*.feature file.

Alright, let's break down the concept of "Background" in easy layman terms:

Imagine you are writing a story, and there are certain details that are the same in every scene. For instance, if your story is about people interacting on a blog, you might always have to mention who the blog owner is, who some of the users are, and what blogs they have.

Now, instead of repeating these same details in every part of your story, you can create a "Background" section. This section is like the setup for the entire story. You list all the common details there, and then, when you get to specific scenes (called "Scenarios" in this context), you don't have to repeat those common details every time.

Here's a simple breakdown using your example:

\*\*Feature: Multiple site support\*\*

In this story, we're talking about multiple blogs.

\*\*Background:\*\*

- Greg is a super admin.

- There's a blog called "Greg's anti-tax rants."

- Dr. Bill is a customer.

- Another blog is "Expensive Therapy," and it's owned by Dr. Bill.

Now, when you get to specific scenes (Scenarios), you don't need to mention these details again. For example:

\*\*Scenario: Dr. Bill posts to his own blog\*\*

- Dr. Bill logs in.

- He tries to post to "Expensive Therapy."

- He should see "Your article was published."

By using the Background, you make your story more organized. The reader knows the setup, and then they focus on the specific actions and outcomes in each scene without repeating unnecessary information. It's like setting the stage once for the entire play.

Remember, keep the Background short and interesting, and only use it for stuff that really matters to the story. If the details are too much, consider splitting them into smaller parts or making them part of the individual scenes (Scenarios).

# Hooks in Cucumber:

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](https://cucumber.io/docs/cucumber/api/?lang=java#conditional-hooks).

You can declare hooks in any class.

## Scenario hooks

Scenario hooks run for every scenario.

### Before

Before hooks run before the first step of each scenario.

Annotated method style:

@Before

public void doSomethingBefore() {

}

Lambda style:

Before(() -> {

});

**Think twice before you use Before**

Whatever happens in a Before hook is invisible to people who only read the features. You should consider using a [background](https://cucumber.io/docs/gherkin/reference#background) as a more explicit alternative, especially if the setup should be readable by non-technical people. Only use a Before hook for low-level logic such as starting a browser or deleting data from a database.

You can specify an explicit order for hooks if you need to.

Annotated method style:

@Before(order = 10)

public void doSomething(){

// Do something before each scenario

}

Lambda style:

Before(10, () -> {

// Do something before each scenario

});

### After

After hooks run after the last step of each scenario, even when the step result is failed, undefined, pending, or skipped.

Annotated method style:

@After

public void doSomethingAfter(Scenario scenario){

// Do something after after scenario

}

Lambda style:

After((Scenario scenario) -> {

});

The scenario parameter is optional. If you use it, you can inspect the status of the scenario.

For example, you can take a screenshot with [WebDriver](https://www.seleniumhq.org/projects/webdriver/) for failed scenarios and embed them in Cucumber’s report.

See the [browser automation page](https://cucumber.io/docs/guides/browser-automation/#screenshot-on-failure) for an example on how to do so.

### Around

Cucumber-JVM does not support Around hooks.

## Step hooks

Step hooks are invoked before and after a step. The hooks have ‘invoke around’ semantics, meaning that if a BeforeStep hook is executed the AfterStep hooks will also be executed regardless of the result of the step. If a step did not pass, the following step and its hooks will be skipped.

### BeforeStep

Annotated method style:

@BeforeStep

public void doSomethingBeforeStep(Scenario scenario){

}

Lambda style:

BeforeStep((Scenario scenario) -> {

});

### AfterStep

Annotated method style:

@AfterStep

public void doSomethingAfterStep(Scenario scenario){

}

Lambda style:

AfterStep((Scenario scenario) -> {

});

## Conditional hooks

Hooks can be conditionally selected for execution based on the tags of the scenario. To run a particular hook only for certain scenarios, you can associate a Before or After hook with a [tag expression](https://cucumber.io/docs/cucumber/api/?lang=java#tag-expressions).

Annotated method style:

@After("@browser and not @headless")

public void doSomethingAfter(Scenario scenario){

}

Lambda style:

After("@browser and not @headless", (Scenario scenario) -> {

});

See more documentation on [tags](https://cucumber.io/docs/cucumber/api/?lang=java#tags).

## Global hooks

Global hooks will run once before any scenario is run or after all scenario have been run.

### BeforeAll

BeforeAll run before any scenario is run.

Annotated method style:

@BeforeAll

public static void beforeAll() {

// Runs before all scenarios

}

### AfterAll

AfterAll run after all scenarios have been executed.

Annotated method style:

@AfterAll

public static void afterAll() {

// Runs after all scenarios

}

# Reports in Cucumber:

#### Summary

This video discusses the different reporting options available in Cucumber BDD. The speaker demonstrates how to generate pretty output, JSON reports, and JUnit reports. They also explain how to publish reports to the Cucumber cloud and view them online.

#### Highlights

- The "pretty" plugin can be used to generate properly aligned console output.

- JSON and JUnit reports can be generated and saved in the target folder.

- Reports can be published to the Cucumber cloud by setting the "cucumber.publish.enabled" property to true.

- Reports published on the Cucumber cloud are accessible via a URL, but they self-destruct after 24 hours.

- The cucumber.properties file can be used to maintain the "cucumber.publish.enabled" property.

- [💡] The "pretty" plugin improves the readability of console output.

- [💡] JSON and JUnit reports can be generated and saved in the target folder.

- [💡] Reports can be published to the Cucumber cloud for easy access.

- [💡] Reports published on the Cucumber cloud self-destruct after 24 hours.

- [💡] The cucumber.properties file can be used to maintain the "cucumber.publish.enabled" property.

# Data Tables in Cucumber:

**As List**

**As Map**

# Scenario Outline in Cucumber: