Cucumber Notes for Study

Introduction

Cucumber BDD (Behavior-Driven Development) is a software development approach that emphasizes collaboration between developers, testers, and business stakeholders to ensure that software products meet the needs of the business.

It uses a plain-text, human-readable language called Gherkin to define the behavior of the software in terms of scenarios, which are written in a structured format that includes Given, When, and Then statements.

How does Cucumber BDD work?

Define the Feature: The business stakeholder and the development team collaborate to define the feature and write a feature file in Gherkin format.

Write Scenarios: The team writes scenarios in Gherkin format that define the behavior of the feature.

Implement Step Definitions: The developers implement step definitions for each scenario. These are the code snippets that execute the actions defined in the scenarios.

Test the Feature: The testers execute the scenarios and verify that the feature works as expected.

Refactor: If necessary, the team refactors the code to improve its design, performance, or maintainability.

Benefits of Cucumber BDD

Ensures that software products meet the needs of the business.

Tests the software thoroughly before it is released to production.

Promotes collaboration between developers, testers, and business stakeholders.

Leads to a better understanding of the requirements and a more effective development process.

What the difference between TDD and BDD?

- BDD (Behavior-Driven Development) and TDD (Test-Driven Development) are two software development methodologies that share some similarities but also have some differences.

TDD:

TDD is a methodology that emphasizes writing automated tests before writing production code.

- The goal is to ensure that the code works as expected and that changes to the code do not break existing functionality.

BDD:

- BDD is a methodology that emphasizes collaboration between developers, testers, and business stakeholders to ensure that the software meets the needs of the business.
- BDD uses a plain-text, human-readable language called Gherkin to define the behavior of the software in terms of scenarios.

Differences between BDD and TDD

- BDD focuses on the behavior of the software from a business perspective, while TDD focuses on the technical implementation of the software.
- BDD scenarios are written in a language that is accessible to non-technical stakeholders, while TDD tests are written in a programming language.
- BDD involves collaboration between developers, testers, and business stakeholders, while TDD is primarily a developer-focused methodology.

- Both BDD and TDD are useful methodologies for ensuring the quality and reliability of software products.
- The choice between them depends on the specific needs and goals of the development team.

Note:

BDD can be seen as more suitable for larger, complex projects that involve collaboration between multiple teams, while TDD can be seen as more suitable for smaller projects where a single developer or a small team is responsible for the development and testing.

Ultimately, the decision on which methodology to use should be based on the needs of the project, the available resources, and the preferences and skills of the development team.

BDD Lifecycle Phases

- **Discovery**: Collaborate with business stakeholders to define the requirements for the software.
- **Formulation:** Create specific scenarios that describe the behavior of the software in Gherkin syntax.
- **Automation**: Write code to automate the scenarios using testing tools such as Cucumber or Selenium.
- **Execution**: Execute the automated tests to verify that the software works as expected using continuous integration and delivery tools.
- **Analysis**: Analyze the results of the tests to identify any defects or issues that need to be addressed using defect tracking software.
- **Maintenance**: Continuously maintain and improve the software using feedback from users and stakeholders and data analytics to monitor performance.

By following this BDD lifecycle, development teams can ensure that their software meets the needs of the business, is thoroughly tested, and is maintained and improved over time. The collaboration between developers, testers, and business stakeholders throughout the lifecycle also promotes a better understanding of the requirements and a more effective development process.

Collaborative Feature File Creation in BDD

In BDD, the feature file is ***a central artifact that defines the behavior of the software from a business perspective.***

The feature file is typically created collaboratively by the ***development team, business stakeholders, and testing team.***

Business stakeholders are responsible for defining the requirements and features of the software, and they may provide input on the language and structure of the feature file.

The development team is responsible for implementing the software and may provide input on the technical feasibility and complexity of the features.

The testing team is responsible for ensuring that the features are thoroughly tested and may provide ***input on the scenarios and test cases that should be included in the feature file.***

The creation of the feature file is ***a collaborative effort that involves all stakeholders to define the behavior of the software in a clear, concise, and unambiguous manner.***

The feature file serves as a shared understanding of the requirements and features of the software and is used as a reference throughout the development process to ensure that the software meets the business needs and is tested thoroughly.

Example

Simple Feature File:

Feature: Calculator

As a user

I want to be able to perform basic arithmetic operations

So that I can calculate the results of simple equations

Scenario: Addition

Given I have entered 2 into the calculator

And I have entered 3 into the calculator

When I press the add button

Then the result should be 5 on the screen

Scenario: Subtraction

Given I have entered 5 into the calculator

And I have entered 2 into the calculator

When I press the subtract button

Then the result should be 3 on the screen

Test Code:

```
import org.junit.Assert;

import org.junit.runner.RunWith;


import cucumber.api.CucumberOptions;

import cucumber.api.junit.Cucumber;


@RunWith(Cucumber.class)

@CucumberOptions(features = "src/test/resources/calculator.feature")

public class CalculatorTest {


   private int result;

   private Calculator calculator = new Calculator();


   @Given("^I have entered (\\d+) into the calculator$")

   public void i_have_entered_into_the_calculator(int arg1) throws Throwable {

      calculator.enter(arg1);

   }


   @When("^I press the add button$")

   public void i_press_the_add_button() throws Throwable {

      result = calculator.add();

   }


   @When("^I press the subtract button$")

   public void i_press_the_subtract_button() throws Throwable {

      result = calculator.subtract();

   }
```

```
    @Then("^the result should be (\\d+) on the screen$")
    public void the_result_should_be_on_the_screen(int arg1) throws Throwable {
        Assert.assertEquals(arg1, result);
    }
}
```

How to use Data Table to pass test Data to Test Code from Feature file

A data table in Cucumber is a way to pass a table of data as an argument to a step definition in a feature file. This can be useful for testing scenarios that involve multiple sets of input data or for testing different combinations of inputs.

Scenario: Verify addition of multiple numbers

Given I have the following numbers:

| 10 |

| 20 |

| 30 |

When I add them together

Then the result should be 60

In this scenario, we're using a data table to pass a list of numbers to the step definition that performs the addition. The data table is defined using the vertical bars (|) to separate the values, and each row represents a separate input value.

To use this data table in a step definition, we can define the step as follows:

```
@Given("^I have the following numbers:$")
public void i_have_the_following_numbers(List<Integer> numbers) throws Throwable {
   // do something with the list of numbers
}
```

How to pass POJO to a data table

In Cucumber, it is possible to pass Plain Old Java Objects (POJOs) to a data table as an argument. This can be useful when you need to test scenarios that involve complex input data, such as objects with multiple properties.

Scenario: Verify user registration

Given I have a new user:

| username | JohnDoe |

| email | johndoe@example.com |

| password | mysecretpassword |

When I register the user

Then the user should be created successfully

In this scenario, we're using a data table to pass a new user object to the step definition that performs the registration. The data table is defined using the vertical bars (|) to separate the property names and values, and each row represents a separate property-value pair.

To use this data table in a step definition, we can define the step as follows:

```
@Given("^I have a new user:$")
public void i_have_a_new_user(User user) throws Throwable {
   // do something with the User object
}
```

To make this work, you'll need to create a User class with properties that match the property names in the data table:

```
public class User {
   private String username;
   private String email;
```

```
    private String password;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

Best Practices for Writing Cucumber Feature Files

- Use descriptive and meaningful feature names
- Write clear and concise scenarios
- Use descriptive scenario names
- Use Gherkin keywords properly
- Use data tables and scenario outlines
- Use comments to improve readability
- Keep the feature file organized

Create high-quality and maintainable feature files

Ensure Cucumber tests are effective in testing software functionality

Provide valuable feedback to the development team

By following these best practices, you can create feature files that are easy to read, understand, and maintain, and that help to ensure the success of your Cucumber tests.

Interview Questions For Cucumber

What is Cucumber, and what is its purpose?

Cucumber is a testing tool that supports Behavior Driven Development (BDD). Its purpose is to facilitate collaboration between technical and non-technical team members by allowing them to write tests in plain English, using a syntax called Gherkin.

What are the different components of Cucumber?

The main components of Cucumber are the feature file, step definitions, and runners. The feature file contains the test scenarios written in Gherkin syntax, the step definitions contain the automation code that maps to the steps in the feature file, and the runners execute the tests and generate reports.

What is Gherkin, and how does it relate to Cucumber?

Gherkin is a plain-text syntax used to describe the behavior of a software system in a structured way. It is the language that Cucumber understands and uses to execute the tests. Gherkin uses keywords like Given, When, Then, And, and But to structure the scenarios in a way that is easy to read and understand by both technical and non-technical team members.

What is BDD, and how does it relate to Cucumber?

BDD is a software development methodology that emphasizes collaboration and communication between technical and non-technical team members. BDD uses examples to illustrate the desired behavior of a system and uses those examples to drive the development process. Cucumber is a tool that supports BDD by allowing teams to write tests in plain English using Gherkin syntax.

What are the advantages of using Cucumber for testing?

Cucumber has several advantages, including improved collaboration between technical and non-technical team members, better visibility into the testing process, more reliable tests due to the use of concrete examples, and the ability to generate human-readable reports.

How do you write a basic Cucumber feature file?

A basic Cucumber feature file includes a title, a brief description of the feature being tested, and one or more scenarios that describe the desired behavior of the system. Each scenario should include a series of steps written in Gherkin syntax.

How do you run a Cucumber test suite?

To run a Cucumber test suite, you need to create a runner class that specifies the location of the feature files and the step definitions. The runner class should include annotations that indicate which features and scenarios to run, as well as which plugins to use for reporting.

What is a step definition, and how do you write one in Cucumber?

A step definition is a piece of automation code that maps to a step in the feature file. Step definitions are written in Java or another programming language and should perform the actions described in the step. To write a step definition, you need to define a regular expression that matches the step text in the feature file and annotate the method with the corresponding keyword.

How do you pass parameters to a Cucumber step definition?

You can pass parameters to a Cucumber step definition by using placeholders in the step text and capturing them in the step definition using regular expressions. You can also use data tables or examples tables to pass multiple sets of parameters to a single step.

How do you use data tables in Cucumber, and what are some use cases for them?

Data tables allow you to pass tabular data to a step definition, which can be useful for testing scenarios that involve complex input data. Data tables can also be used to verify the output of a system by comparing the expected and actual results in a tabular format.

How do you integrate Cucumber with other testing frameworks and tools?

Cucumber can be integrated with other testing frameworks and toolso enhance its capabilities and enable more comprehensive testing. For example:

- JUnit or TestNG: Cucumber can be run as a JUnit or TestNG test, allowing it to be integrated with other testing frameworks or tools that are based on these frameworks.
- Selenium: Cucumber can be used to drive Selenium tests, allowing you to test the behavior of a web application in a more human-readable format.
- REST Assured: Cucumber can be used to test REST APIs using REST Assured, a popular testing library for RESTful web services.
- Maven or Gradle: Cucumber can be integrated with build tools like Maven or Gradle to automate the test execution and reporting process.

How do you handle asynchronous behavior in Cucumber tests?

Asynchronous behavior can be handled in Cucumber tests using techniques like waiting for a specific condition to be true, polling for a certain period of time, or using timeouts to limit the amount of time spent waiting. You can also use tools like explicit waits or async/await functions to handle asynchronous behavior in your step definitions.

How do you handle test data in Cucumber tests?

Test data can be handled in Cucumber tests using a variety of techniques, including using data tables or examples tables to pass data to step definitions, using external data sources like databases or files, or generating test data dynamically during the test execution.

How do you generate reports in Cucumber?

Cucumber generates reports in a variety of formats, including HTML, JSON, and PDF. You can use plugins to customize the format and content of the reports, and you can also use third-party tools like Jenkins or Bamboo to integrate the reports into a continuous integration or continuous delivery workflow.

What are some best practices for writing effective Cucumber tests?

Some best practices for writing effective Cucumber tests include:

1. Focusing on concrete examples that illustrate the desired behavior of the system.
2. Writing clear and concise step definitions that map to the steps in the feature file.
3. Using data tables and examples tables to parameterize the tests and make them more flexible.
4. Keeping the scenarios and step definitions independent and reusable.
5. Regularly reviewing and updating the tests to ensure they remain relevant and effective over time.

What is the difference between Cucumber and Gherkin?

Cucumber is a testing framework that uses the Gherkin language to write human-readable, executable specifications. Gherkin is a plain-text language that Cucumber understands, and it's used to write feature files that describe the behavior of the system in a way that's easy for both technical and non-technical stakeholders to understand.

How do you debug Cucumber tests?

You can use various debugging techniques to troubleshoot Cucumber tests, such as setting breakpoints in your code, using logging statements to trace the flow of the test execution, or using the Cucumber CLI to run the tests in debug mode.

How do you handle test dependencies in Cucumber?

Test dependencies can be handled in Cucumber using techniques like test doubles or mock objects to simulate the behavior of the dependencies, using dependency injection to inject the dependencies into the test code, or using integration testing to test the interactions between the system under test and its dependencies.

How do you ensure that Cucumber tests are maintainable?

You can ensure that Cucumber tests are maintainable by following best practices like keeping the tests focused and concise, using descriptive and meaningful names for the scenarios and step definitions, and designing the tests to be easy to modify and extend.

How do you determine what to test using Cucumber?

Determining what to test using Cucumber involves identifying the key features and user stories of the system under test, and then writing scenarios that illustrate the desired behavior of those features and stories. It's also important to consider edge cases and error scenarios, as well as any performance or security requirements that need to be tested. Additionally, it's a good practice to involve all stakeholders, including developers, testers, and business analysts, in the test planning process to ensure that all relevant scenarios are covered.

What are Hooks in Cucumber?

In Cucumber BDD, hooks are blocks of code that run before or after each scenario or step in a feature file. Hooks can be used to set up the test environment, clean up the test data, or perform other common tasks that need to be done before or after the tests. Here are the various hooks that can be used in Cucumber:

1. Before: This hook runs before each scenario and can be used to set up the test environment, initialize variables, or perform other tasks that need to be done before the scenario starts.
2. After: This hook runs after each scenario and can be used to clean up the test data, close database connections, or perform other tasks that need to be done after the scenario finishes.
3. BeforeStep: This hook runs before each step and can be used to set up the data or state needed for the step to run properly.
4. AfterStep: This hook runs after each step and can be used to perform clean up or validation tasks related to the step that just finished.
5. BeforeAll: This hook runs once before all scenarios in a feature file and can be used to set up any global test environment or dependencies that are needed for the entire test suite.
6. AfterAll: This hook runs once after all scenarios in a feature file have completed and can be used to clean up any global test environment or dependencies that were set up in the BeforeAll hook.

By using hooks in Cucumber, you can make your test suite more modular, maintainable, and easier to manage. Hooks can also be used to ensure that your tests are properly isolated and that the test environment is consistent across all scenarios and feature files.

How to run cucumber test via main class?

To run Cucumber tests via a main class in Java, you can use the JUnit test runner. Here's an example of how to set up a JUnit test runner in Java to run Cucumber tests:

Create a JUnit test class and add the @RunWith(Cucumber.class) annotation to specify that you want to use the Cucumber runner.

```
import org.junit.runner.RunWith;

import io.cucumber.junit.Cucumber;


@RunWith(Cucumber.class)
```

How to run cucumber test via main class using testng

To run Cucumber tests via a main class using TestNG, you can use the TestNG test runner. Here's an example of how to set up a TestNG test runner in Java to run Cucumber tests:

Create a TestNG test class and add the @Test annotation to specify that you want to run a test method.

```
import org.testng.annotations.Test;

@Test
public class RunCucumberTest {

}
```

In the @Test method, create an instance of the Cucumber io.cucumber.testng.CucumberRunner class and pass the location of your feature file(s) as a parameter.

```
import io.cucumber.testng.CucumberOptions;
import io.cucumber.testng.CucumberRunner;
import org.testng.annotations.Test;

@Test
@CucumberOptions(features = {"src/test/resources/features"})
public class RunCucumberTest {

  @Test
  public void runCucumber() {
    new CucumberRunner(getClass()).run();
  }
}
```

Add the @CucumberOptions annotation and specify the location of your step definitions and any other options you want to use.

```
import io.cucumber.testng.CucumberOptions;
import io.cucumber.testng.CucumberRunner;
import org.testng.annotations.Test;

@Test
@CucumberOptions(
  features = {"src/test/resources/features"},
  glue = {"com.example.steps"},
  tags = {"@smoke"},
  plugin = {"pretty"}
)
public class RunCucumberTest {

  @Test
  public void runCucumber() {
    new CucumberRunner(getClass()).run();
  }
}
```

Run the TestNG test class as you would any other TestNG test class. The Cucumber tests will be executed as part of the TestNG test run.

Integrate Cucumber with Jenkins

To integrate Cucumber with Jenkins, you can follow these steps:

1. Install the Cucumber plugin in Jenkins. To do this, go to the Jenkins dashboard and navigate to Manage Jenkins > Manage Plugins. In the "Available" tab, search for "Cucumber" and install the "Cucumber Reports" plugin.
2. Create a Jenkins job for running your Cucumber tests. To do this, go to the Jenkins dashboard and click on "New Item" to create a new job. Choose "Freestyle project" and give your job a name.
3. In the job configuration page, go to the "Source Code Management" section and configure your source code repository. Choose the appropriate SCM tool and provide the repository URL, credentials, and other required information.
4. In the "Build" section, add a build step to run your Cucumber tests. Choose "Execute shell" or "Execute Windows batch command" depending on your system, and enter the command to run your tests. For example, you might use a command like this:

mvn clean test

5. In the "Post-build Actions" section, add a post-build step to generate the Cucumber reports. Choose "Publish Cucumber Results" and configure the location of your Cucumber report files. By default, the reports are generated in the "target/cucumber-reports" directory.
6. Save your job configuration and run the job. The Cucumber tests will be executed, and the Cucumber reports will be generated and published in the job page. You can view the reports to see the test results, including the passed and failed scenarios, step definitions, and other details.