BDD with Cucumber and Selenium - A Beginner to Advanced Guide

This guide is designed for new learners to understand the complete BDD (Behavior Driven Development) process using Cucumber and Selenium WebDriver. It includes live Example Mapping, framework structure, optimization techniques, and integration with other tools like Jenkins and Rest Assured, with hands-on code examples and explanations.

1. Understanding BDD and Live Example Mapping

What is BDD?

Behavior Driven Development is a software development approach that encourages collaboration between developers, testers, and business stakeholders. It focuses on writing test scenarios in plain English using Gherkin syntax.

2. Framework Overview

Goals:

- **Readable**: Business-friendly feature files
- **Maintainable**: Reusable code using Page Object Model
- **Scalable**: Supports parallel execution and CI/CD integration

Core Technologies:

- Cucumber (for BDD and Gherkin syntax)
- Selenium WebDriver (for browser automation)
- Java (language of implementation)
- Maven (build tool)
- TestNG / JUnit / CLI (for execution)

- POM (Page Object Model for structure)

3. Framework Structure

This is a modular structure that separates features, step definitions, and page objects:

```
src/test/java
├── features                # Gherkin feature files
│   └── login.feature
├── pages                   # Page classes with web elements and
actions
│   └── LoginPage.java
├── stepDefinitions         # Step definitions connecting Gherkin to
code
│   └── LoginSteps.java
├── utils                   # Utility classes like ConfigReader
│   └── ConfigReader.java
├── runner                  # Test runner class
│   └── TestRunner.java
```

config.properties - Stores environment settings:

```
browser=chrome
environment=qa
```

Constants.java - Holds constant values used across the framework:

```
public class Constants {
    public static final String QA_URL = "https://qa.example.com";
    public static final String BROWSER = "chrome";
}
```

ConfigReader.java - Reads values from the config file:

```
public class ConfigReader {
    Properties properties = new Properties();
```

```
    public ConfigReader() {
        try {
            FileInputStream fis = new
FileInputStream("config.properties");
            properties.load(fis);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public String getProperty(String key) {
        return properties.getProperty(key);
    }
}
```

4. Gherkin and Cucumber Anti-Patterns

What is Gherkin?

Gherkin is a language that uses keywords like Given, When, Then to write test cases in plain English. It bridges the gap between non-technical and technical stakeholders.

Correct Gherkin:

```
Scenario: Successful login
  Given user is on login page
  When user enters valid credentials
  Then user is redirected to dashboard
```

Anti-pattern Example:

```
Given user logs in using "username" and "password" and clicks submit
and waits for the page and verifies text
```

Why this is bad: It's hard to read, not reusable, and violates the single responsibility principle. Avoid combining too many actions into one step.

5. Mastering Cucumber Fundamentals

Tags

Used to filter scenarios during execution:

```
@smoke
Scenario: Login test
```

Hooks

Run before and after each scenario:

```
@Before
public void setup() {
    // Init driver
}

@After
public void tearDown() {
    driver.quit();
}
```

Cucumber Options

Configure feature paths, glue code, and plugins:

```
@RunWith(Cucumber.class)
@CucumberOptions(
    features = "src/test/java/features",
    glue = "stepDefinitions",
    plugin = {"pretty", "html:target/cucumber-reports"}
)
public class TestRunner {}
```

6. Runners: JUnit, TestNG, CLI, Maven

JUnit Runner

```
@RunWith(Cucumber.class)
@CucumberOptions(features = "features", glue = "stepDefinitions")
public class JUnitRunner {}
```

TestNG Runner

```
public class TestNGRunner extends AbstractTestNGCucumberTests {}
```

CLI / Maven Execution

```
mvn test -Dcucumber.options="--tags @smoke"
```

7. Parallel Execution

Run tests faster by executing in parallel:

With JUnit:

Use the cucumber-parallel plugin to split feature files.

With TestNG:

```
<suite name="Parallel" parallel="tests" thread-count="2">
  <test name="Test1">
    <classes>
      <class name="runner.TestRunner" />
    </classes>
  </test>
</suite>
```

8. Dependency Injection with PicoContainer

In Cucumber, if we want to share the state between multiple-step definition files, we will need to use dependency injection (DI).

What is Dependency Injection (DI)?

Dependency Injection is a pattern that allows objects to **receive their dependencies (e.g., WebDriver, utility classes)** from an external source, rather than creating them internally. This promotes **loose coupling, reusability, and easier testing**.

Why PicoContainer?

- **Lightweight DI framework**
- **Built into Cucumber by default**
- Allows sharing of **WebDriver**, **PageObjects**, and **utility classes** across step definitions
- Eliminates the need for `static` or `singleton` objects

Example:

Hooks.java:

```
public class Hooks {
    private WebDriver driver;

    @Inject // Optional if only one constructor
    public Hooks(DriverManager manager) {
        this.driver = manager.getDriver();
    }
}
```

DriverManager.java:

```
public class DriverManager {
    private WebDriver driver;

    public WebDriver getDriver() {
        if (driver == null) {
```

```
            driver = new ChromeDriver();
        }
        return driver;
    }
}
```

This ensures that all step definitions using this class will share the same WebDriver instance.

✅When and Where to Use @Inject

Scenario	Should You Use @Inject?	Notes
One constructor only	❌Not required	PicoContainer automatically injects
Multiple constructors	✅Required	Specifies which constructor to use
Using fields for DI (not recommended)	✅Required	But constructor injection is preferred
For clarity or consistency	✅Optional	Makes it explicit that DI is in use
Using external DI frameworks (e.g., Guice)	✅Required	Needed for integration

Summary

- **Reduced Boilerplate**: Automatically manages the instantiation of classes.
- **Simplified Codebase**: Removes the need for static access or singletons for shared components.

9. Gherkin Syntax Deep Dive

Data Tables

Used to pass complex input data:

```
Given user provides credentials:
  | username | password  |
  | user1    | pass12345 |
```

Step Definition:

```
@Given("user provides credentials:")
public void user_provides_credentials(DataTable table) {
    List<Map<String, String>> data = table.asMaps();
    loginPage.login(data.get(0).get("username"),
data.get(0).get("password"));
}
```

10. Custom Parameter and DataTable Types

What is it?

Custom types in Cucumber allow automatic transformation of step input or table rows into Java objects, improving code readability and reducing boilerplate code.

Custom Parameter Type Example:

```
@ParameterType("[a-zA-Z0-9]+@[a-z]+\\.com")
public String email(String email) {
    return email.toLowerCase();
}
```

In step:

```
Given the user email is john@example.com
```

Custom DataTable Type Example:

User.java:

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

    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }

    // Getters and setters
}
```

TypeRegistryConfig.java:

```
public class TypeRegistryConfig implements TypeRegistryConfigurer {
    @Override
    public Locale locale() {
        return Locale.ENGLISH;
    }

    @Override
    public void configureTypeRegistry(TypeRegistry registry) {
        registry.defineDataTableType(new DataTableType(User.class,
            (TableRowTransformer<User>) entry -> new
User(entry.get(0), entry.get(1))));
    }
}
```

Step Definition:

```
@Given("the following user credentials")
public void the_following_user_credentials(List<User> users) {
    for (User user : users) {
        loginPage.login(user.getUsername(), user.getPassword());
    }
}
```

Feature File:

```
Given the following user credentials
  | username | password  |
  | user1    | pass123   |
  | user2    | secret456 |
```

11. REST API Integration Using Rest Assured

What is Rest Assured?

A Java library for testing REST APIs. You can validate responses, status codes, headers, and more.

Why Use It?

- Integrates with Cucumber and TestNG
- Makes API testing simple and readable

Example:

```
Response res = given()
    .contentType("application/json")
    .body(payload)
    .post("/api/create");

String sessionCookie = res.getCookie("JSESSIONID");
driver.manage().addCookie(new Cookie("JSESSIONID", sessionCookie));
```

12. Step Definition Management

What is it?

It's the process of organizing and mapping Gherkin steps to Java methods in a clean and maintainable way.

LoginPage.java:

```
public class LoginPage {
    WebDriver driver;

    public LoginPage(WebDriver driver) {
        this.driver = driver;
    }

    public void enterUsername(String username) {
        driver.findElement(By.id("username")).sendKeys(username);
    }

    public void enterPassword(String password) {
        driver.findElement(By.id("password")).sendKeys(password);
    }

    public void clickLogin() {
        driver.findElement(By.id("loginBtn")).click();
    }
}
```

13. Jenkins Integration

What is Jenkins?

Jenkins is a CI/CD tool that automates code integration, testing, and deployment.

Jenkinsfile Example:

```
pipeline {
  agent any
  stages {
    stage('Build and Test') {
      steps {
        sh 'mvn clean test'
      }
```

```
    }
  }
}
```

GitHub Webhook Setup:

- Go to GitHub repo → Settings → Webhooks
- Add webhook URL: `http://<jenkins_url>/github-webhook/`

Auto-trigger Automation from Jenkins using GitHub Web Hooks, SCM Polling, and Build Frequency

GitHub WebHooks

Webhooks can be used to automatically trigger Jenkins builds whenever there is a change in the GitHub repository.

1. Go to **GitHub** → **Settings** → **Webhooks**.
2. Add the **Jenkins webhook URL** (e.g., `http://your-jenkins-server/github-webhook/`).

SCM Polling in Jenkins

You can configure Jenkins to poll the source code repository at regular intervals to check for changes.

1. In Jenkins, go to the **Job Configuration** page.
2. Under the **Build Triggers** section, check **Poll SCM** and set the schedule using cron syntax (e.g., `H/15 * * * *` to poll every 15 minutes).

Build Frequency

Set up build frequency within Jenkins for automated builds on a schedule, or set up **GitHub webhooks** to trigger builds on push events.

```
groovy
CopyEdit
pipeline {
  agent any
```

```
  triggers {
    cron('H 0 * * *')  // Trigger every day at midnight
  }
  stages {
    stage('Build and Test') {
      steps {
        sh 'mvn clean test'
      }
    }
  }
}
```

14. Report Management

Cucumber Plugins

```
plugin = {"html:target/report.html", "json:target/report.json"}
```

Why Reports Matter

- Show test summary (pass/fail)
- Provide step-by-step results
- Useful for stakeholders and debugging

Archive Reports in Jenkins

```
post {
  always {
    archiveArtifacts artifacts: 'target/*.html', fingerprint: true
  }
}
```

15. Key Principles for Writing Maintainable Code

Single Responsibility Principle (SRP)

The **SRP** dictates that each class should have one responsibility and should be responsible for only one part of the functionality. This keeps code clean, modular, and easier to maintain.

For example, in a Cucumber project:

- **LoginPage** class should only manage login page interactions.
- **LoginSteps** should define step definitions and interaction with page objects but not manage the browser driver or configurations.

```
java
CopyEdit
public class LoginPage {
    WebDriver driver;

    public LoginPage(WebDriver driver) {
        this.driver = driver;
    }

    public void enterUsername(String username) {
        driver.findElement(By.id("username")).sendKeys(username);
    }

    public void enterPassword(String password) {
        driver.findElement(By.id("password")).sendKeys(password);
    }

    public void clickLogin() {
        driver.findElement(By.id("loginBtn")).click();
    }
}
```

Don't Repeat Yourself (DRY)

The **DRY** principle emphasizes avoiding redundancy in code. Repeated code increases maintenance overhead and can lead to errors.

In Cucumber frameworks:

- **Reusable Methods**: If you're repeating the same code across steps or tests, create utility methods.
- **Page Object Model (POM)**: Abstract the interaction with the web page into a class, reducing repeated actions.

Example: Instead of writing code to interact with elements in each step definition, centralize it in the **LoginPage** class.

Applying Object-Oriented Programming (OOP) Concepts

OOP allows us to manage complex projects efficiently by using principles such as inheritance, polymorphism, encapsulation, and abstraction.

- **Encapsulation**: Hide the internal state of objects. In Selenium, classes like `WebDriver` encapsulate browser-specific functionality.
- **Abstraction**: Create abstract classes or interfaces for defining behavior that is common across multiple page objects or tests.
- **Inheritance**: Create base classes for common logic, allowing for shared behavior across tests.
- **Polymorphism**: Use method overriding or interfaces to allow objects to take many forms.

Example of **Inheritance**:

```
java
CopyEdit
public class BasePage {
    protected WebDriver driver;

    public BasePage(WebDriver driver) {
        this.driver = driver;
    }
```

```
    public void waitForElement(By locator) {
        new WebDriverWait(driver,
10).until(ExpectedConditions.visibilityOfElementLocated(locator));
    }
}

java
CopyEdit
public class LoginPage extends BasePage {
    public LoginPage(WebDriver driver) {
        super(driver);
    }

    public void enterUsername(String username) {
        driver.findElement(By.id("username")).sendKeys(username);
    }

    public void clickLogin() {
        driver.findElement(By.id("loginBtn")).click();
    }
}
```