International Journal on Science and Technology (IJSAT)
E-ISSN: 2229-7677 ● Website: www.ijsat.org ● Email: editor@ijsat.org

Building a BDD Framework from Scratch for Automation Testing

Asha Rani Rajendran Nair Chandrika

Abstract

Behavior-Driven Development (BDD) is a widely adopted software development methodology that emphasizes collaboration among developers, testers, and business stakeholders by using a common language to define application behavior. This article explores the step-by-step process of building a robust BDD automation framework from scratch, tailored for long-term projects. It highlights the importance of choosing the right tools, such as Cucumber, Selenium, and Gherkin, to streamline automation efforts while bridging the gap between business requirements and technical implementation. A structured approach to framework design is presented, covering aspects like folder organization, dependency management, and integration of key components such as Page Object Models and test runner configurations. Readers will gain insights into writing feature files in Gherkin, implementing reusable step definitions, and executing tests to ensure scalability and maintainability. The article also discusses best practices for sustaining the framework, including modularization, version control, and handling future project expansions. With this guide, teams can enhance test coverage, improve communication, and deliver software that aligns seamlessly with user expectations and business goals. By automating behavior-driven scenarios effectively, organizations can achieve higher efficiency, early defect detection, and improved software quality.

Keywords: Behavior-Driven Development, BDD Framework, Test Automation, Cucumber, Selenium, Gherkin Syntax, Page Object Model, Automation Testing Tools, Software Quality Assurance, Scalable Framework Design.

I. INTRODUCTION

Behavior-Driven Development (BDD) is a powerful software development methodology that has gained immense popularity for its focus on collaboration, understanding, and communication across stakeholders such as business analysts, developers, and testers. BDD aims to bridge the gap between business requirements and the development team, ensuring the application behaves as expected from a user's perspective.

While BDD's role in the development lifecycle is clear, the challenge for many automation testers lies in building a robust framework that can sustain the project's long-term needs. The implementation of a BDD automation framework from scratch requires careful planning, design, and organization to ensure that the framework is scalable, maintainable, and adaptable to future changes.

In this article, we will guide you through how to design and implement a BDD framework for a long-term project, providing key insights into framework components, their interconnections, and best practices for sustainability.

II. Why Implement BDD in Automation?

Behavior-Driven Development (BDD) encourages collaboration between developers, testers, and business stakeholders by defining software behavior in a language that is accessible to all. This methodology brings several key benefits to the table:

- **Enhanced Communication**: The Gherkin syntax used in BDD allows writing feature files in plain English, making it easier for non-technical stakeholders to understand the behavior of the system.
- **Improved Test Coverage**: BDD drives comprehensive test coverage by ensuring the tests are written based on business requirements and user stories.
- **Promotes Automation**: BDD fosters test automation by automating the scenarios described in feature files. This ensures consistency and reusability of tests.
- **Early Defect Detection**: BDD tests are often written before the development work starts (test-first), which helps identify defects early in the software development lifecycle.

In a long-term project, BDD will help teams deliver software with minimal defects, increased confidence, and alignment with business goals. Therefore, building a solid BDD framework is essential for long-term success.

III. Step 1: Selecting the Right Tools for BDD Automation

Before diving into framework design, it's essential to choose the right tools that will help you achieve your goal. The most popular tools for BDD automation include:

- **Cucumber**: Cucumber is the most used tool for BDD testing in Java-based projects. It reads and executes tests written in Gherkin syntax and provides integration with various other tools.
- **Gherkin**: Gherkin is a language used to write the feature files. It uses plain English, which allows everyone to understand the behavior of the system.
- **Selenium**: Selenium is used for automating web browsers. It's highly compatible with Java and works well with Cucumber to automate the web-based UI tests.
- **JUnit/TestNG**: JUnit and TestNG are testing frameworks in Java that are commonly used to run Cucumber tests.
- **Maven/Gradle**: These build tools help you manage dependencies, automate builds, and run tests.

A. Setting Up Dependencies

To get started, you need to configure your project with the necessary dependencies. Below is an example of the Maven configuration:

```
<dependencies>
    <dependency>
        <groupId>io.cucumber</groupId>
        <artifactId>cucumber-java</artifactId>
        <version>latest_version</version>
    </dependency>
    <dependency>
        <groupId>io.cucumber</groupId>
        <artifactId>cucumber-junit</artifactId>
        <version>latest_version</version>
    </dependency>
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-java</artifactId>
        <version>latest_version</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.x</version>
    </dependency>
</dependencies>
```

Figure 1: Maven dependency Configuration

IV. Step 2: Designing the Folder Structure

When building a BDD framework, the organization of files and directories plays a crucial role in long-term maintainability. A clear and well-structured folder layout helps in better collaboration between developers, testers, and other stakeholders. Here's a typical folder structure for a BDD automation framework:

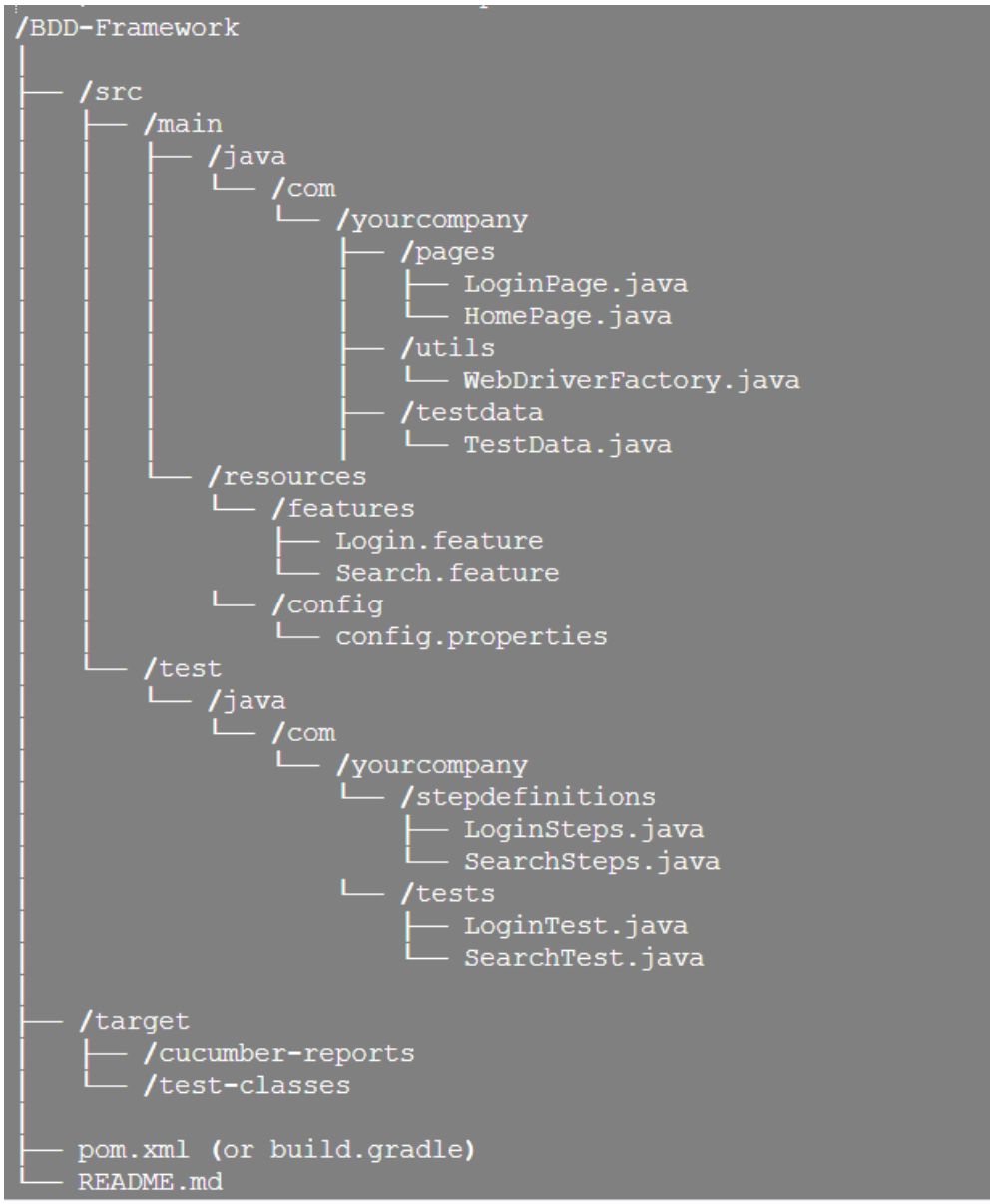

Figure 2: Sample folder structure

A. Explanation of Folder Structure

1. /src/main/java/com/yourcompany/pages:

This directory holds all **Page Object Model (POM)** classes. Each class corresponds to a page of the application. The page objects encapsulate the logic related to interacting with elements on that page (e.g., buttons, text fields).

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

    // Constructor to initialize the WebDriver
    public LoginPage(WebDriver driver) {
        this.driver = driver;
    }

    // Method to enter the username
    public void enterUsername(String username) {
        driver.findElement(By.id("username")).sendKeys(username);
    }

    // Method to enter the password
    public void enterPassword(String password) {
        driver.findElement(By.id("password")).sendKeys(password);
    }

    // Method to click the login button
    public void clickLoginButton() {
        driver.findElement(By.id("login")).click();
    }
}
```

Figure 3: Page Object Model

2. /src/main/resources/features:

This folder contains your **Gherkin feature files** written in plain English, outlining the behavior of the application in scenarios.

```
Feature: User Login

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

Figure 4: feature file (UserLogin.feature):

3. /src/test/java/com/yourcompany/stepdefinitions:

Step definitions are Java classes where each method corresponds to a step in the Gherkin feature file. These methods are annotated with @Given, @When, @Then.

```java
package com.yourcompany.stepdefinitions;

import com.yourcompany.pages.LoginPage;
import com.yourcompany.pages.HomePage;
import io.cucumber.java.en.Given;
import io.cucumber.java.en.When;
import io.cucumber.java.en.Then;
import org.openqa.selenium.WebDriver;

import static org.junit.Assert.assertTrue;

public class LoginSteps {
    WebDriver driver = new WebDriverFactory().getDriver();
    LoginPage loginPage;
    HomePage homePage;

    @Given("the user is on the login page")
    public void userIsOnLoginPage() {
        loginPage = new LoginPage(driver);
        loginPage.navigateToLoginPage();
    }

    @When("the user enters valid credentials")
    public void userEntersValidCredentials() {
        loginPage.enterUsername("validUser");
        loginPage.enterPassword("validPassword");
        loginPage.clickLoginButton();
    }

    @Then("the user is redirected to the home page")
    public void userIsRedirectedToHomePage() {
        homePage = new HomePage(driver);
        assertTrue(homePage.isHomePageDisplayed());
    }
}
```

Figure 5: Step Definitions (LoginSteps.java)

4. /src/test/java/com/yourcompany/tests:

This folder contains test runner classes that trigger the execution of your BDD scenarios. A typical class uses the Cucumber @RunWith annotation to link the feature files and step definitions.

```
package com.yourcompany.tests;

import io.cucumber.junit.Cucumber;
import io.cucumber.junit.CucumberOptions;
import org.junit.runner.RunWith;

@RunWith(Cucumber.class)
@CucumberOptions(
    features = "src/main/resources/features/Login.feature",  // Path to the feature file
    glue = "com.yourcompany.stepdefinitions",  // Path to step definitions
    plugin = {"pretty", "html:target/cucumber-reports/LoginTestReport.html"},
    monochrome = true
)
public class LoginTest {
}
```

Figure 6: Test Runner (LoginTest.java)

5. /target/cucumber-reports:

After executing tests, Cucumber generates **detailed test reports** here, which give you insights into the execution and results of your tests.

V. Step 3: Running the Tests:

- Run with Maven: Use the following Maven command to run the tests:
  mvn test
  This will compile the project, execute the Cucumber tests, and generate the HTML reports in the target/cucumber-reports directory.
- Run with IDE: You can also run the LoginTest.java directly from your IDE (e.g., IntelliJ IDEA or Eclipse), which will trigger the Cucumber execution.

VI. Step 4: Best Practices for Maintaining the Framework

- **Modularization**: Keep your framework modular by breaking down code into smaller, reusable components. For instance, separate page objects, utilities, and test data into distinct classes.
- **Scalability**: As your project grows, you will need to add more features and scenarios. Ensure that your framework can scale by adding new feature files and organizing step definitions efficiently.
- **Maintainability**: Ensure your framework is easy to maintain by following coding standards, using descriptive names for classes and methods, and ensuring your code is well-documented.
- **CI/CD Integration**: Integrate your BDD framework with Continuous Integration (CI) tools like Jenkins. This will allow you to run tests automatically as part of the build process.
- **Cross-Browser and Parallel Testing**: For large-scale applications, consider adding support for cross-browser testing (e.g., Chrome, Firefox, Safari) and parallel execution to speed up test runs.

VII. Conclusion

- Enhanced Collaboration: BDD fosters collaboration between developers, testers, and business stakeholders, ensuring software behavior aligns with business goals.
- Comprehensive Test Coverage: By writing tests based on business requirements, BDD guarantees comprehensive test coverage that aligns with the application's expected behavior.
- Early Defect Detection: Test-first BDD practices enable early identification of defects, reducing the cost and effort of fixing them later in the development cycle.

- Choosing the Right Tools: Tools like Cucumber, Gherkin, Selenium, and JUnit/TestNG are essential for building an efficient and scalable BDD automation framework.
- Effective Folder Structure: A well-organized folder structure, including directories for page objects, feature files, step definitions, and test runners, helps in the long-term maintainability of the framework.
- Best Practices: Following best practices such as modularization, scalability, maintainability, and CI/CD integration ensures the framework's robustness as the project grows.
- Scalability and Cross-Browser Testing: Ensure the framework can handle new feature additions and support cross-browser testing for large applications.

REFERENCE

[1] https://cucumber.io/docs/cucumber/
[2] https://www.selenium.dev/documentation/webdriver/
[3] https://cucumber.io/docs/gherkin/
[4] https://junit.org/junit5/docs/current/user-guide/
[5] https://testng.org/
[6] https://maven.apache.org/
[7] https://www.selenium.dev/documentation/test_practices/encouraged/page_object_models/
[8] https://www.geeksforgeeks.org/how-to-make-a-ci-cd-pipeline-in-jenkins/