Hybrid Automation Framework Project Selenium with Java

Certainly! A **Hybrid Automation Framework** is a combination of multiple types of testing frameworks, generally combining the features of **data-driven** and **keyword-driven** frameworks. This approach allows the best features of each individual framework to be leveraged, making it more powerful and flexible for automation testing.

In the context of **Selenium with Java**, this project would involve using Selenium WebDriver (which is a popular tool for automating web browsers) in combination with Java to build an automation framework that supports various testing strategies. Here's a breakdown of the key concepts for this topic:

1. Selenium with Java Overview

- **Selenium WebDriver** is a tool for automating browsers. It provides a programming interface to interact with web elements on a webpage, simulating real user interactions.
- **Java** is the language used to write the automation scripts. Selenium WebDriver supports multiple languages like Java, Python, C#, Ruby, etc., and Java is one of the most commonly used languages.
- **Selenium Components**:
  - **Selenium WebDriver**: To interact with the browser.
  - **Selenium Grid**: To run tests on multiple machines and browsers in parallel.
  - **Selenium IDE**: A Chrome and Firefox extension for recording and replaying tests (mostly for simple tests).
  - **Selenium RC** (Remote Control): An older version that is now mostly replaced by WebDriver.

2. Hybrid Framework

- The **Hybrid Framework** combines multiple frameworks such as **Data-Driven Framework** and **Keyword-Driven Framework**.

- It allows tests to be created in a more modular and reusable way. A key benefit is the flexibility to execute tests using data from external sources like Excel, CSV, or databases.

Types of Frameworks You Can Combine:

- **Data-Driven Framework**: Focuses on executing the same test with multiple sets of data. The test scripts are separated from the test data.
- **Keyword-Driven Framework**: Focuses on executing a set of predefined actions or "keywords" that map to actual automation code.
- **Hybrid**: Combines both approaches, allowing the use of data from external sources along with reusable keywords for actions.

3. Components of a Hybrid Framework in Selenium

- **Test Scripts**: The automation scripts written in Java using Selenium WebDriver to perform actions like opening a browser, clicking buttons, verifying results, etc.
- **Object Repository**: A central place to store all the locators for web elements. This can be stored in a file (e.g., Excel, XML, or properties files) so that locators are not hardcoded in the test scripts.
- **Data Source**: External files like **Excel**, **CSV**, **JSON**, or databases that provide data inputs for tests. In the data-driven part of the framework, the same test can be executed with different sets of data.
- **Test Data**: Information used in the tests, which can be driven from external files. The test data could consist of inputs for forms, expected results, etc.
- **Reporting**: Automated frameworks often integrate with reporting tools like **TestNG Reports**, **Extent Reports**, or **Allure** to generate HTML-based or graphical reports for test execution.

4. How It Works

In a hybrid framework:

- **Test Case Execution**: A test script will execute actions based on keywords defined in a keyword-driven approach, and it will use test data from external sources like Excel files.
- **Modularity and Reusability**: Test steps are designed as reusable functions. For example, common actions like login or logout can be coded as

keywords or methods that are reused across different test cases.

- **Separation of Concerns**: Test scripts are separated from the test data and test actions, improving maintainability and scalability.

5. Framework Design Considerations

- **Page Object Model (POM)**: A design pattern often used in hybrid frameworks where each page of the application is represented by a separate class. This helps in reducing redundancy and promotes better organization of code.
- **TestNG or JUnit**: These are popular testing frameworks in Java for running the tests, providing features like test annotations, parallel execution, and report generation.
- **Maven**: A build automation tool used to manage project dependencies and run tests in a standardized manner.
- **Continuous Integration (CI)**: Integrating the automation framework with tools like Jenkins for automated test execution on every commit to a code repository.

6. Advantages of a Hybrid Framework

- **Flexibility**: You can combine different types of testing (e.g., functional, regression) and run tests with different data sets easily.
- **Maintainability**: The use of reusable test scripts, object repositories, and data files makes it easier to maintain the framework over time.
- **Scalability**: It is easier to scale the framework to handle large applications or multiple test scenarios with minimal changes.
- **Extensibility**: New test cases, keywords, or data sources can be added without affecting the existing functionality.
- **Test Reusability**: Since test steps and keywords are modular, they can be reused across different test cases, saving time and effort.

7. Example of a Hybrid Framework Workflow

- **Test Data**: Data is stored in an external file like Excel (e.g., login data for testing multiple user credentials).

- **Test Script**: A script reads this data and calls a method (like `login(String username, String password)`).
- **Keyword Definition**: Actions like clicking a button, entering text, or verifying text can be defined as reusable functions.
- **Execution**: The script is run for different sets of test data (e.g., different usernames/passwords).
- **Reporting**: After the tests are executed, detailed reports are generated showing pass/fail statuses, errors, screenshots, etc.

8. Tools and Technologies to Integrate

- **Apache POI** or **Selenium Excel Library** for reading/writing Excel files.
- **TestNG** for managing tests, providing annotations, and generating reports.
- **Maven** for dependency management and automating the build process.
- **Jenkins** for continuous integration and automating the execution of test suites.
- **Extent Reports** for enhanced reporting with visuals.
- **Log4j** or **SLF4J** for logging and tracking test execution details.

9. Challenges and Considerations

- **Test Data Management**: Managing large amounts of test data in external files can become complex.
- **Framework Complexity**: The hybrid approach can be more complex to design and maintain compared to simpler frameworks.
- **Performance**: The framework's performance can degrade if too many unnecessary actions are executed due to poorly structured code.

In summary, a **Hybrid Automation Framework using Selenium with Java** is an advanced, modular framework that leverages the benefits of both **Data-Driven** and **Keyword-Driven** testing approaches. It is highly flexible, scalable, and can be extended to handle large-scale automation projects. The combination of **Page Object Model**, **TestNG**, and **external data management** makes it an ideal choice for complex applications.

Sure! Here's a simple example of a **Hybrid Automation Framework** using **Selenium with Java**, incorporating **TestNG** for test management, **Excel** for data-driven testing, and the **Page Object Model (POM)** design pattern for better maintainability.

1. Setting up the Project

- **Maven Dependencies** (pom.xml):

```
<dependencies>
    <!-- Selenium WebDriver dependency -->
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-java</artifactId>
        <version>4.7.0</version>
    </dependency>

    <!-- TestNG dependency for test management -->
    <dependency>
        <groupId>org.testng</groupId>
        <artifactId>testng</artifactId>
        <version>7.4.0</version>
        <scope>test</scope>
    </dependency>

    <!-- Apache POI dependency for reading Excel files -->
    <dependency>
        <groupId>org.apache.poi</groupId>
        <artifactId>poi-ooxml</artifactId>
        <version>5.2.3</version>
    </dependency>

    <!-- Extent Reports for enhanced reporting -->
    <dependency>
        <groupId>com.aventstack</groupId>
        <artifactId>extentreports</artifactId>
        <version>5.0.9</version>
```

```
    </dependency>
</dependencies>
```

2. Page Object Model (POM) Design

Create a **Page Object** class for the **Login** page.

LoginPage.java

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;

public class LoginPage {
    WebDriver driver;

    // Locators for Login Page
    By usernameField = By.id("username");
    By passwordField = By.id("password");
    By loginButton = By.id("loginButton");

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

    // Method to enter username
    public void enterUsername(String username) {
        driver.findElement(usernameField).sendKeys(usernam
e);
    }

    // Method to enter password
    public void enterPassword(String password) {
        driver.findElement(passwordField).sendKeys(passwor
d);
    }

    // Method to click login button
    public void clickLoginButton() {
```

```
        driver.findElement(loginButton).click();
    }
}
```

3. Test Data Management (Excel Reading)

Create a utility class to read test data from an **Excel** file.

`ExcelReader.java`

```
import org.apache.poi.ss.usermodel.*;
import java.io.FileInputStream;
import java.io.IOException;

public class ExcelReader {
    private Workbook workbook;
    private Sheet sheet;

    public ExcelReader(String filePath, String sheetName) t
hrows IOException {
        FileInputStream fis = new FileInputStream(filePat
h);
        workbook = WorkbookFactory.create(fis);
        sheet = workbook.getSheet(sheetName);
    }

    public String getCellData(int rowNum, int colNum) {
        Row row = sheet.getRow(rowNum);
        Cell cell = row.getCell(colNum);
        return cell.getStringCellValue();
    }

    public int getRowCount() {
        return sheet.getPhysicalNumberOfRows();
    }
}
```


4. Test Script (TestNG)

Now, let's create the test case to perform the login functionality using the Hybrid framework approach. The data for login will be fetched from the Excel file.

LoginTest.java

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

public class LoginTest {
    WebDriver driver;
    LoginPage loginPage;

    @BeforeClass
    public void setUp() {
        // Set up WebDriver (make sure you have the correct
path for chromedriver)
        System.setProperty("webdriver.chrome.driver", "pat
h/to/chromedriver");
        driver = new ChromeDriver();
        loginPage = new LoginPage(driver);
        driver.get("https://example.com/login"); // Replace
with actual URL
    }

    @DataProvider(name = "loginData")
    public Object[][] getLoginData() throws Exception {
        ExcelReader reader = new ExcelReader("path/to/login
Data.xlsx", "LoginData");
        int rows = reader.getRowCount();
        Object[][] data = new Object[rows][2];

        for (int i = 1; i < rows; i++) {
            data[i-1][0] = reader.getCellData(i, 0); // Use
rname
```

```
            data[i-1][1] = reader.getCellData(i, 1); // Pas
sword
        }

        return data;
    }

    @Test(dataProvider = "loginData")
    public void testLogin(String username, String password)
{
        // Perform login actions using the Page Object
        loginPage.enterUsername(username);
        loginPage.enterPassword(password);
        loginPage.clickLoginButton();

        // Add assertions based on the login response (exam
ple)
        Assert.assertTrue(driver.getCurrentUrl().contains
("dashboard"));
    }

    // Clean up
    @AfterClass
    public void tearDown() {
        driver.quit();
    }
}
```


5. Excel File Structure (loginData.xlsx)

Here's what the **Excel** file (`loginData.xlsx`) might look like:

Username	Password
testuser1	password1
testuser2	password2
testuser3	password3

6. Test Execution

When you run the above code:

1. The **ExcelReader** class will read the data from the `loginData.xlsx` file.
2. The **LoginTest** class will run the test for each set of data provided by the **@DataProvider** annotation, simulating a login for each user and password combination.
3. The **LoginPage** class provides an abstraction for the actions on the login page.

7. Test Reporting (Optional)

You can integrate **Extent Reports** to generate detailed, visual reports for test execution:

ExtentReports Example:

```
import com.aventstack.extentreports.ExtentReports;
import com.aventstack.extentreports.ExtentTest;
import com.aventstack.extentreports.reporter.ExtentSparkRep
orter;

public class ExtentReportManager {
    static ExtentReports extent;

    public static ExtentReports createInstance() {
        if (extent == null) {
            String reportPath = "extent-report.html";
            ExtentSparkReporter spark = new ExtentSparkRepo
rter(reportPath);
            extent = new ExtentReports();
            extent.attachReporter(spark);
        }
        return extent;
    }
}

public class LoginTest {
    static ExtentReports extent = ExtentReportManager.creat
eInstance();
```

```
    ExtentTest test;

    @BeforeClass
    public void setUp() {
        // Setup WebDriver...
        test = extent.createTest("Login Test");
    }

    @Test(dataProvider = "loginData")
    public void testLogin(String username, String password)
{
        // Perform login...
        test.pass("Login successful for " + username);
    }

    @AfterClass
    public void tearDown() {
        driver.quit();
        extent.flush(); // Flush the report
    }
}
```


8. Conclusion

In this example, we combined **Data-Driven** (using Excel), **Keyword-Driven** (through Page Object Model for actions like `login()`, `enterUsername()`), and **TestNG** for managing the test flow, making the framework **Hybrid** in nature.

You can extend this framework by adding more tests, integrating other external data sources, and improving the modularity by adding reusable functions for common actions like clicking, typing, or verifying page elements.

This framework is a good starting point for larger automation projects, and can be scaled with additional features such as parallel execution, logging, and CI/CD integration with Jenkins.