## Test Automation Coding Challenge Part No. 3 Question 1

**GitHub Repo Link** : https://github.com/faizannmaqsood/AutomationTestQuestion3

# Evaluate the Current Framework

Review Existing Codebase: Understand the structure, libraries, and tools used in the current API automation framework.

Identify Reusable Components: Identify the components that can be reused for web and mobile testing, such as test data management, configuration files, reporting, and logging mechanisms.

# Choose Appropriate Tools

Web Automation Tools: Choose tools like Selenium, Playwright, or Cypress for web automation.

Mobile Automation Tools: Choose tools like Appium for mobile automation.

# Integrate Web and Mobile Testing with API Testing

Modularize the Framework: Refactor the existing framework to create a modular structure that supports API, web, and mobile testing.

Common Utilities: Create common utility classes and methods that can be used across all types of testing.

Unified Reporting: Integrate a reporting tool that consolidates test results from API, web, and mobile tests.

# Continuous Integration/Continuous Deployment (CI/CD)

CI/CD Pipeline: Set up a CI/CD pipeline (e.g., Jenkins, GitHub Actions, GitLab CI) to run API, web, and mobile tests.

Environment Setup: Ensure environments are properly configured for running tests on different platforms.

# Enhance Test Coverage

End-to-End Scenarios: Create end-to-end test scenarios that involve API calls followed by web or mobile actions.

Data-Driven Tests: Implement data-driven testing to enhance test coverage and maintainability.

# Documentation and Training

Documentation: Update the framework documentation to include details on how to run web and mobile tests.

Training: Conduct training sessions for the team to familiarize them with the new framework structure and tools.

# Continuous Monitoring and Maintenance

Monitoring: Continuously monitor the test results and maintain the framework to handle new changes in the application.

Feedback Loop: Establish a feedback loop to gather insights from test results and improve the framework iteratively.

### Example Implementation Steps

Here’s a high-level example of what the implementation steps might look like:

Refactor Framework Structure

/tests

/api

TestAPI1.java

TestAPI2.java

/web

TestWeb1.java

TestWeb2.java

/mobile

TestMobile1.java

TestMobile2.java

/common

BaseTest.java

Config.java

Utils.java

/reports

report.html

pom.xml

### Sample Code for Web Test Integration

import com.microsoft.playwright.\*;

public class TestWeb1 extends BaseTest {

@Test

public void testWebLogin() {

// Launch browser

try (Browser browser = playwright.chromium().launch(new BrowserType.LaunchOptions().setHeadless(false))) {

Page page = browser.newPage();

page.navigate("https://example.com/login");

page.fill("#username", "testuser");

page.fill("#password", "password");

page.click("button[type=submit]");

assertEquals("Dashboard", page.title());

}

}

}

### Sample Code for Mobile Test Integration

import io.appium.java\_client.MobileElement;

import io.appium.java\_client.android.AndroidDriver;

import org.openqa.selenium.remote.DesiredCapabilities;

public class TestMobile1 extends BaseTest {

@Test

public void testMobileLogin() throws MalformedURLException {

DesiredCapabilities caps = new DesiredCapabilities();

caps.setCapability("deviceName", "Android Emulator");

caps.setCapability("platformName", "Android");

caps.setCapability("appPackage", "com.example");

caps.setCapability("appActivity", "com.example.MainActivity");

AndroidDriver<MobileElement> driver = new AndroidDriver<>(new URL("http://localhost:4723/wd/hub"), caps);

driver.findElementById("com.example:id/username").sendKeys("testuser");

driver.findElementById("com.example:id/password").sendKeys("password");

driver.findElementById("com.example:id/login").click();

Assert.assertEquals(driver.findElementById("com.example:id/title").getText(), "Dashboard");

driver.quit();

}

## Test Automation Coding Challenge Part No. 3 Question 2

Contract testing is a type of testing used to ensure that two services (a consumer and a provider) can communicate and interact with each other as expected. It focuses on the interactions between services, verifying that they adhere to a predefined contract (agreement). This is particularly useful in microservices architectures where multiple services depend on each other.

### Tools Required

1. Pact: A tool for implementing consumer-driven contract tests.

2. Postman: For API testing and contract verification.

3. Spring Cloud Contract: A tool for contract testing in Spring applications.

4. WireMock: A simulator for HTTP-based APIs, often used for creating mock responses in contract tests.

### Example Scenario

An e-commerce application where the Frontend Service fetches user details from the User Service.

### Step-by-Step Process

#### 1. Consumer Side (Frontend Service)

- Define the Contract: The Frontend Service defines the expected interactions with the User Service.

- Example Contract: When a GET request is made to `/user/1`, the User Service should return a response containing user details with ID 1 and name "Faizan Maqsood".

import au.com.dius.pact.consumer.dsl.PactDslWithProvider;

import au.com.dius.pact.consumer.dsl.PactDslJsonBody;

import au.com.dius.pact.consumer.junit.Pact;

import au.com.dius.pact.consumer.junit.PactProviderRule;

import au.com.dius.pact.consumer.junit.PactVerification;

import org.junit.Rule;

import org.junit.Test;

import static io.restassured.RestAssured.given;

public class FrontendServiceTest {

@Rule

public PactProviderRule mockProvider = new PactProviderRule("UserService", this);

@Pact(consumer = "FrontendService")

public PactDslWithProvider createPact(PactDslWithProvider builder) {

return builder

.given("User 1 exists")

.uponReceiving("A request for user 1")

.path("/user/1")

.method("GET")

.willRespondWith()

.status(200)

.body(new PactDslJsonBody()

.integerType("id", 1)

.stringType("name", "John Doe"))

.toPact();

}

@Test

@PactVerification("UserService")

public void testGetUser() {

given()

.baseUri(mockProvider.getUrl())

.when()

.get("/user/1")

.then()

.statusCode(200)

.body("id", equalTo(1))

.body("name", equalTo("Faizan Maqsood"));

}

}

```

#### 2. Provider Side (User Service)

- Verify the Contract: The User Service verifies the contract to ensure it can fulfill the expectations of the Frontend Service.

import au.com.dius.pact.provider.junit.Provider;

import au.com.dius.pact.provider.junit.PactRunner;

import au.com.dius.pact.provider.junit.loader.PactFolder;

import au.com.dius.pact.provider.junit.target.HttpTarget;

import au.com.dius.pact.provider.junit.target.Target;

import au.com.dius.pact.provider.junit.target.TestTarget;

import org.junit.runner.RunWith;

@RunWith(PactRunner.class)

@Provider("UserService")

@PactFolder("pacts")

public class UserServiceTest {

@TestTarget

public final Target target = new HttpTarget("http", "localhost", 8080);

}

```

In this example, the User Service starts up and runs on `localhost:8080`, verifying that it can meet the contract defined by the Frontend Service.