1. If two features are done in parallel how would you make your branching strategy?

Answer: When two features are developed in parallel, it is crucial to have a robust branching strategy in test automation to ensure that both features can be tested independently without interference.

Here is a detailed approach to an effective branching strategy in such scenarios:

Branching Strategy

Main Branch (e.g., main or master):

This is the stable branch where the final, tested code is merged. It should always be in a deployable state.

Develop Branch (optional, e.g., develop):

This branch serves as an integration branch for features. Features are merged here before being moved to the main branch. It is especially useful for continuous integration.

Feature Branches:

Each new feature is developed in its own branch created from the main or develop branch. These branches are named based on the feature being developed (e.g., feature/feature1, feature/feature2).

Test Branches:

Create corresponding test branches for automation testing of each feature (e.g., test/feature1, test/feature2). These branches allow you to develop and run automated tests specific to each feature without impacting the main test suite.

2. What are Self-healing tests?

Answer: Self-healing tests are an advanced approach in test automation that automatically adapt to changes in the application under test (AUT) without human intervention. These tests use intelligent algorithms and machine learning techniques to identify and adjust to changes in the application, thereby reducing test maintenance efforts and increasing the robustness of the test suite.

Key Concepts and Benefits of Self-Healing Tests

Dynamic Locator Strategies:

Self-healing tests can dynamically update locators (such as XPath, CSS selectors) when they change due to modifications in the application's UI. Instead of tests failing due to a missing or changed element, the self-healing mechanism attempts to find the element using alternative locators or patterns.

Machine Learning Algorithms:

These tests often leverage machine learning models to predict and identify elements based on historical data and patterns. This helps in accurately locating elements even if their attributes change.

Reduced Maintenance:

Traditional automated tests require frequent updates to handle changes in the application. Self-healing tests minimize the need for manual intervention, reducing the maintenance burden on QA teams.

Improved Test Stability:

By automatically adjusting to changes, self-healing tests increase the stability and reliability of the test suite, ensuring continuous integration and delivery processes are less prone to disruptions caused by minor changes in the application.

3. Behavioral Interview Question - Different issues I faced in the team and what I did to solve them

Answer:

1. Lack of Test Coverage

Issue: Insufficient test coverage leading to missed defects and lower quality releases.

Solution:

- Conduct a Test Coverage Analysis: Evaluate current test coverage using tools like code coverage analyzers.
- Identify Gaps: Determine which areas of the application are under-tested.
- Prioritize Test Cases: Focus on critical functionalities and high-risk areas first.
- Increase Automated Testing: Implement or enhance automated testing to cover more scenarios and improve consistency.

- Regular Reviews: Schedule regular reviews to ensure new features and bug fixes are covered by tests.

2. Flaky Tests

Issue: Intermittent test failures leading to false positives/negatives and unreliable test results.

Solution:

- Analyze Failure Patterns: Identify and log flaky tests and analyze the patterns or common causes of failures.
- Improve Test Stability: Fix underlying issues such as timing problems, dependencies, or environmental inconsistencies.
- Mock External Services: Use mocking frameworks to simulate external dependencies and reduce test flakiness.
- Parallel Execution: Ensure tests can run in parallel without interference, isolating state and data.
- Retries and Logging: Implement retry mechanisms for known flaky tests and improve logging to diagnose failures better.

3. Resource Constraints

Issue: Limited resources (e.g., environments, tools, personnel) hindering testing activities.

Solution:

- Optimize Resource Usage: Schedule tests to run during off-peak hours or use cloud-based testing environments to scale resources as needed.
- Cross-Train Team Members: Ensure team members are skilled in multiple areas to fill gaps when resources are limited.
- Prioritize Critical Tests: Focus on high-priority tests and critical functionalities when resources are constrained.
- Automation: Increase the automation of repetitive and time-consuming tasks to free up resources for more complex testing.

4. How would you set automatic build trigger between Jenkins and your code base?

Setting up an automatic build trigger between Jenkins and your code base involves configuring Jenkins to automatically start a build whenever there is a change in your code repository. Here's a step-by-step guide to achieve this:

Prerequisites

- **Jenkins Installed:** Ensure you have Jenkins installed and running.
- **Source Code Repository:** A repository hosted on a service like GitHub, GitLab, Bitbucket, or any other version control system.

Steps to Set Up Automatic Build Trigger

1. Install Necessary Plugins

- For GitHub: Install the "GitHub Integration Plugin" or "Git Plugin".
- For GitLab: Install the "GitLab Plugin".
- For Bitbucket: Install the "Bitbucket Plugin".

2. Create a New Jenkins Job

- Open your Jenkins dashboard.
- Click on "New Item".
- Enter a name for your job and select "Freestyle project" or "Pipeline", then click "OK".

3. Configure the Source Code Repository

- In the job configuration page, go to the "Source Code Management" section.
- Select the repository type (e.g., Git).
- Enter the repository URL and credentials if needed.

4. Set Up Webhook in Your Code Repository

- Go to your repository on GitHub, GitLab, or Bitbucket.
- Navigate to the "Settings" page of the repository.
- Find the "Webhooks" section and add a new webhook.

5. Payload URL

- Enter the URL of your Jenkins server followed by /github-webhook/ for GitHub, /gitlab-webhook/ for GitLab, or the equivalent endpoint for your version control system.

- Content type: Select application/json.
- Events: Choose to trigger the webhook for push events and possibly other events like pull requests.
- Save the webhook configuration.

6. Configure Jenkins Job to Trigger on Push Events

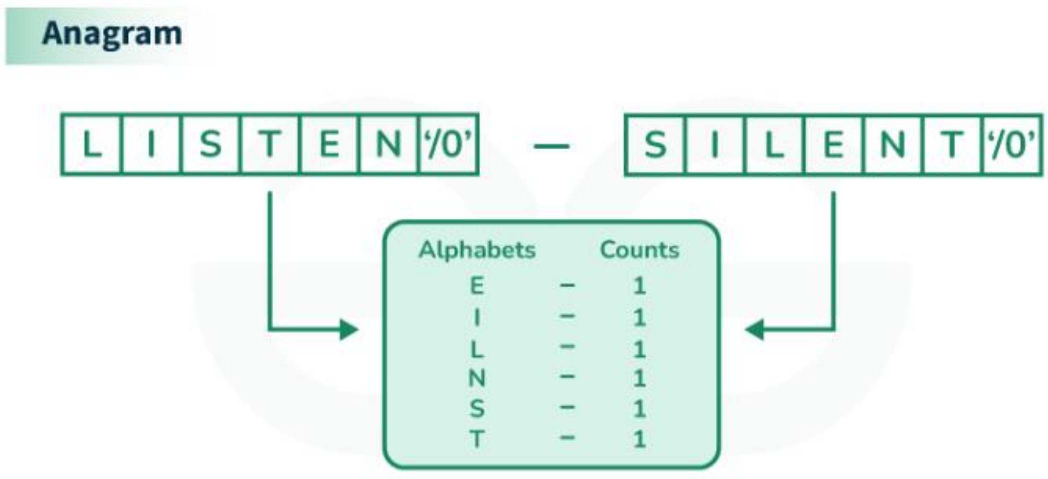
- In the job configuration page, scroll down to the "Build Triggers" section.
- Check the box for "GitHub hook trigger for GITScm polling" for GitHub or the equivalent option for your VCS.
- Save the configuration.

7. Configure Build Steps

- Add the necessary build steps to compile, test, and package your application as required.
- For example, you can add a build step to execute a shell command or call a specific build tool like Maven, Gradle, or npm.

7. Write a Java Program to check if two strings are anagram of each other or not.

Answer:



```java
class RD {
    // Function to check whether two strings
    // are anagram of each other
    static boolean areAnagram(char[] str1, char[] str2)
    {
        // Get lengths of both strings
        int n1 = str1.length;
        int n2 = str2.length;

        // If length of both strings is not
        // same, then they cannot be anagram
        if (n1 != n2)
            return false;

        // Sort both strings
        Arrays.sort(str1);
        Arrays.sort(str2);

        // Compare sorted strings
        for (int i = 0; i < n1; i++)
            if (str1[i] != str2[i])
                return false;

        return true;
    }

    // Driver Code
    public static void main(String args[])
    {
        char str1[] = { 't', 'e', 's', 't' };
        char str2[] = { 't', 't', 'e', 'w' };

        // Function Call
        if (areAnagram(str1, str2))
            System.out.println("The two strings are"
                                   + " anagram of each other");
        else
            System.out.println("The two strings are not"
                                   + " anagram of each other");
    }
}
```

8. How would you validate API response in API automation & which are the Best Practices?

Answer:

Define Test Cases

- Identify the endpoints to be tested.
- Define test cases for each endpoint, including the expected status codes, response body structure, and data.

Send API Requests

- Use the testing framework to send API requests to the endpoints.

Validate Status Codes

- Check that the API returns the expected status codes (e.g., 200 for success, 404 for not found, 400 for bad request).

Validate Response Body

- Validate the structure of the response body (e.g., JSON schema validation).
- Check the values of specific fields to ensure they are correct.
- Ensure that mandatory fields are present.

Validate Headers

- Check that the response headers contain the expected values (e.g., Content-Type, Authorization, etc.).

Validate Response Time

- Ensure that the API response time is within acceptable limits.

Error Handling and Edge Cases

- Validate the API's behavior for error scenarios and edge cases (e.g., invalid inputs, missing parameters).

Best Practices for API Validation

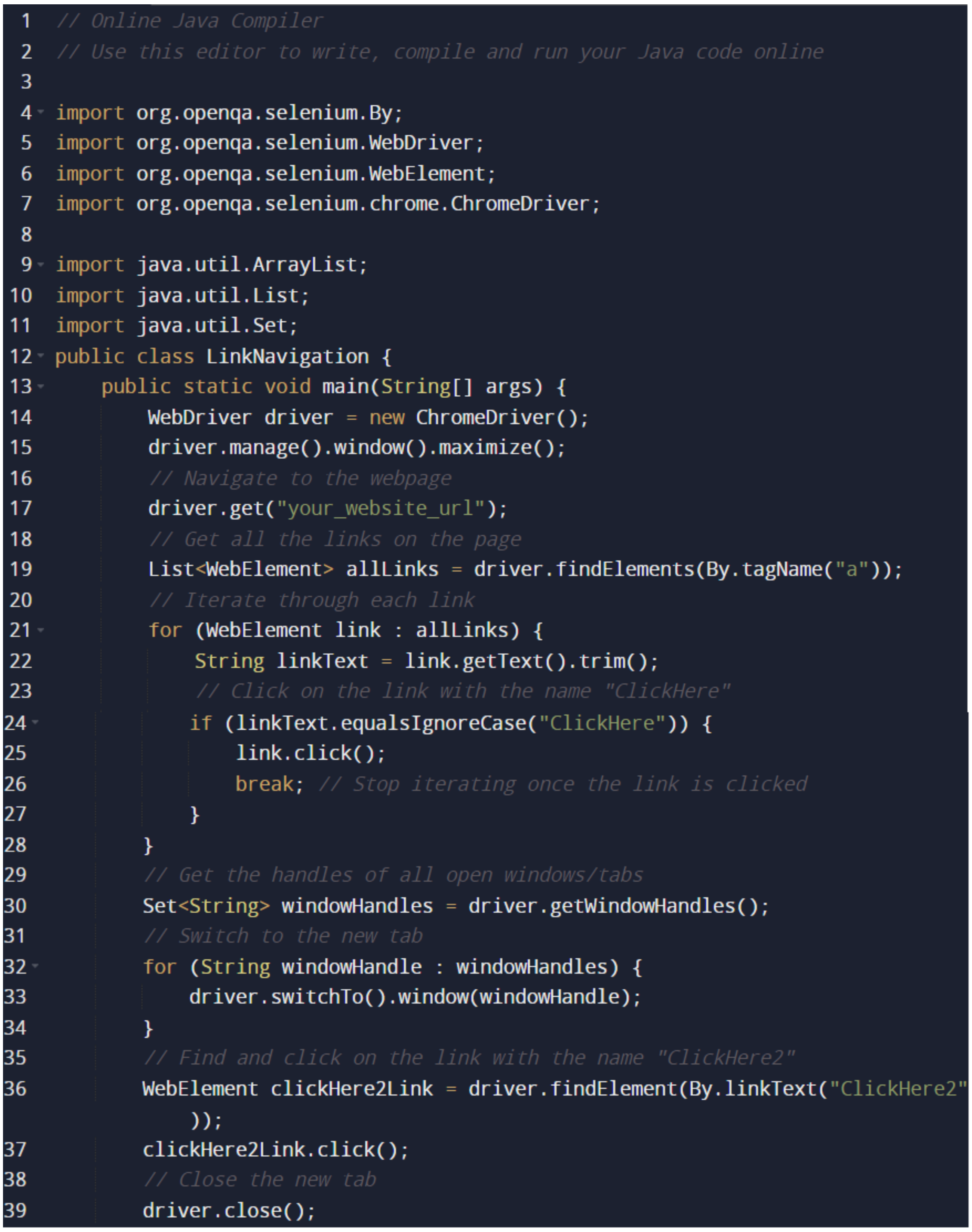
- **Use Assertions:** Ensure your tests include assertions to check the status codes, response body, headers, and response times.
- **Handle Dynamic Data:** When dealing with dynamic data, use appropriate strategies to validate the responses.
- **Error Scenarios:** Test for error conditions and edge cases to ensure the API handles them gracefully.
- **Schema Validation:** Use JSON schema validation to ensure the response structure is as expected.
- **Logging and Reporting:** Implement logging and detailed reporting to track test results and debug issues easily.
- **Continuous Integration:** Integrate your API tests into your CI/CD pipeline to run them automatically on code changes.

Q9. Key Differences between Hybrid & BDD Framework

- **Approach:**
  - **Cucumber (BDD):** Focuses on BDD, using plain language to define test cases, enhancing collaboration between technical and non-technical stakeholders.
  - **Hybrid Framework:** Focuses on integrating multiple testing approaches to create a versatile and robust testing solution.
- **Language:**
  - **Cucumber (BDD):** Uses Gherkin language for writing test scenarios.
  - **Hybrid Framework:** Can use various programming languages and frameworks depending on the components integrated.
- **Test Case Definition:**
  - **Cucumber (BDD):** Test cases are written in a structured, readable format (Given-When-Then) aimed at clarity and understanding.
  - **Hybrid Framework:** Test cases can be defined in multiple ways, leveraging the strengths of data-driven, keyword-driven, and modular approaches.
- **Stakeholder Involvement:**
  - **Cucumber (BDD):** Encourages active involvement of business stakeholders in the test creation process.
  - **Hybrid Framework:** Primarily focused on technical flexibility and might not involve business stakeholders directly in test creation.

In summary, Cucumber (BDD) is best suited for projects where collaboration between business and technical teams is crucial, and clear communication of requirements is needed. A hybrid framework is ideal for projects that require flexibility, reusability, and a combination of various testing approaches to achieve comprehensive test coverage.

Q10. Write the Code for fetching all links and click on the link with the name ClickHer e, then navigate to the new tab, and then click on the link inside this tab, the link na med ClickHere2. Then close this tab and switch to the present browser tab



```java
// Online Java Compiler
// Use this editor to write, compile and run your Java code online

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

import java.util.ArrayList;
import java.util.List;
import java.util.Set;
public class LinkNavigation {
    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver();
        driver.manage().window().maximize();
        // Navigate to the webpage
        driver.get("your_website_url");
        // Get all the links on the page
        List<WebElement> allLinks = driver.findElements(By.tagName("a"));
        // Iterate through each link
        for (WebElement link : allLinks) {
            String linkText = link.getText().trim();
            // Click on the link with the name "ClickHere"
            if (linkText.equalsIgnoreCase("ClickHere")) {
                link.click();
                break; // Stop iterating once the link is clicked
            }
        }
        // Get the handles of all open windows/tabs
        Set<String> windowHandles = driver.getWindowHandles();
        // Switch to the new tab
        for (String windowHandle : windowHandles) {
            driver.switchTo().window(windowHandle);
        }
        // Find and click on the link with the name "ClickHere2"
        WebElement clickHere2Link = driver.findElement(By.linkText("ClickHere2"
            ));
        clickHere2Link.click();
        // Close the new tab
        driver.close();
```