**User Guide - UI Automation with Playwright using TypeScript**

**1. Introduction**

Welcome to the User Guide for UI Automation with Playwright using TypeScript. This guide will walk you through the process of setting up your development environment, writing and running tests, and leveraging various features of Playwright and TypeScript for efficient UI automation.

**2. Setting Up**

**Using VS Code Application:** To get started, download and install [Visual Studio Code's official website](https://code.visualstudio.com/).

**Downloading Extensions:** After installing Visual Studio Code, enhance your development experience by installing the following essential extensions:

* **Playwright Test**: Enables testing with Playwright directly in VS Code.
* **GitHub Pull Requests**: Facilitates integration with GitHub for managing your codebase.

**3. Installing Dependencies**

Before writing tests, install dependencies related to Playwright and your project by following these steps:

* **Create Folder**: Make a new directory for your automation project on your computer.
* **Initialize Git**: Open a terminal in the new folder and run `**git init**`.
* **Clone Repository**: Clone the Playwright template repository using git clone <https://github.com/testingmavens/Playwright-template.git>
* **Open in VS Code**: Launch Visual Studio Code and open the cloned folder.
* **Switch Branch**: If necessary, use `**git fetch**` to see available branches and switch or create a new branch.
* **Install Dependencies**: Run `**npm i**` to install dependencies, including Playwright, in your project directory.
* **Install Browsers**: Execute `**npx playwright install --with-deps**` to install required browsers.

**4. Creating a New Local Branch**

To create a new local branch for your changes, follow these steps:

* **Switch Branch**: Open the terminal in VS Code and run `**git fetch**` to see available branches.
* **Create Branch**: Use `**git checkout -b your\_branch\_name**` to create and switch to a new branch.
* **Commit Changes**: If needed, use ` **git add .** ` to stage changes, then `**git commit -m**` "Your message" to commit them.
* **Push Branch**: Optionally, push your branch to the remote repository with `**git push origin your\_branch\_name**`.

**5. Organizing Your Automation Project**

Maintain a structured organization for better readability and maintenance:

* **tests**: Contains test files with individual test cases.
* **pages**: Houses page files containing locators and functions for interacting with web elements.
* **test data**: Stores JSON files containing test data required for test cases.
* **common**: Includes an `**actions**` file which contains all the customized functions for actions like click, type, and other interactions.
* **env**: Contains environment files (`**.env**`) with details like base URL, domain, username, password, and other credentials.
* **helper**: Contains multiple utility files:
  + **apiFunctions**: Contains functions related to API calls, which hit the API and return the response.
  + **apiRequests**: Includes different HTTP requests like GET, POST, PUT, and DELETE. Requests from `**apiFunctions**` pass through this file and return responses.
  + **dateTime**: Provides functions that return various date and time formats.
  + **randoms**: Contains functions to generate random characters.
  + **TokenManager**: Includes functions to generate tokens for API authentication.
* **playwright/.auth**: Contains JSON files with cookie details for different users in the application, such as admin, user etc. These files get overwritten everytime a login session happens through the UI for the respective user profile.
* **reporters**: Houses files like `**jiraBot**` and `**mavensReporter**`, which are used to update the test results in Jira.
* **utils**: Contains the `**globalSetup**` file, which is used to set the environment path.

**6. Writing Reusable Functions**

When writing reusable functions, follow best practices and maintain clarity. Here are examples demonstrating common actions in UI automation:

**Example 1: Click Action**

This function performs a click action on the "Save" button. We use `**async**` to handle the asynchronous nature of the click action.

async clickSaveButton() {

await test.step("Click on save button", async () => {

await this.actions.clickButton(this.saveButton, "Save")

})

}

**Explanation:**

* **async**: Declares that the function will always return a promise. It allows us to use `**await**` inside the function to handle asynchronous operations.
* **await**: Pauses the execution of the function until the promise is resolved. This is essential for actions like clicking a button because we need to wait for the click action to complete before proceeding.

**Example 2: Getting a Confirmation Message**

This function retrieves a confirmation message after an action is performed. Since the function returns a value, we should not encode the return statement inside a test step.

async getSavedConfirmation() {

await test.step("Get saved confirmation message", async () => { })

const savedMessage = await this.savedToast.textContent();

return savedMessage;

}

**Explanation:**

* **async**: Declares that the function will always return a promise.
* **await**: Used to pause the function execution until the `**textContent**` method resolves. This is necessary to ensure that we get the correct message text.
* **Returning a Value**: When a function returns a value (like `**savedMessage**`), it should not return directly from within a `**test.step**` block. This ensures the value is correctly handled and prevents potential issues with unresolved promises or incorrect return values.

**Important Note:**

* If a function is returning any value, the return statement should not be encoded inside a `**test.step**`. This prevents issues related to asynchronous operations and ensures the function returns the expected value correctly.

**7. Understanding Await, Async, and Other Concepts**

* In TypeScript, `**async**` and `**await**` are used to handle asynchronous operations in a synchronous manner. Here's a brief explanation of these concepts:
* **async**: Declaring a function as `**async**` indicates that it will always return a promise.
* **await**: Pauses the execution of the function until the promise is resolved, allowing asynchronous code to be written in a synchronous style.
* **Promises**: Represent the eventual completion or failure of an asynchronous operation and its resulting value.

**Best Practice:**

All `**async**` functions should be used with `**await**` in the calling steps/functions to ensure proper handling of asynchronous operations. Here's an example:

async performAction() {

await this.clickSaveButton();

const message = await this.getSavedConfirmation();

console.log(message);

}

* `**performAction**`: This function calls two asynchronous functions (`**clickSaveButton**` and `**getSavedConfirmation**`) using `**await**` to ensure each step completes before moving to the next.

**8. Understanding the Playwright Config File**

The `**playwright.config.ts**` file is crucial for configuring how your Playwright tests are executed. It allows you to define global settings, specify test reporters, manage projects, and set test-specific options. Here’s a breakdown of the configuration file and how to use it effectively:

### **Key Components of the Configuration File**

**1. ReportPortal Configuration**

const RPconfig = {

token: '123abc',

endpoint: '<https://reportportal.testingmavens.tools/api/v1>',

project: 'projectName',

launch: 'projectName suite',

includeTestSteps: true,

attributes: [

{

key: 'key',

value: 'value',

},

{

value: 'value',

},

],

description: ' projectName Trial Launch',

};

* **RPconfig**: Configuration for ReportPortal, which includes the token, endpoint, project details, and other metadata.

**2. Global Setup**

globalSetup: 'utils/globalSetup.ts',

* Specifies a script that runs before all tests to set up global configurations, such as environment variables.

**3. Reporters**

reporter: process.env.CI ?

[['./reporters/mavensReporter.ts', { customOption: 'Github Execution' }], ['line'], ['html', { open: 'always' }]] :

[['line'], ['html', { open: 'always' }], ['allure-playwright', { detail: false, suiteTitle: true, outputFolder: 'allure-results' }]],

* Configures the reporters used to generate test reports. Different configurations can be set for CI environments and local development.

**4. Workers**

workers: 2,

* Specifies the number of concurrent worker processes for running tests.

**5. Retries**

retries: 1,

* Sets the number of times to retry a failed test.

**6. Test Directory and Matching**

testDir: './tests',

testMatch: ["/tests/demo.test.ts"],

* Specifies the directory where the test files are located and matches specific test files to run.

**7. Timeouts**

timeout: 3 \* 60 \* 1000,

expect: {

timeout: 30000

},

* Configures the timeout for each test and the timeout for assertions.

**8. Use Options**

use: {

headless: false,

screenshot: 'on',

video: 'retain-on-failure',

trace: 'off',

baseURL: process.env.ENV ? `${process.env.domain}` : '[https://abc.demo.io](https://guitests.fexa.io)',

navigationTimeout: 1 \* 60 \* 1000,

actionTimeout: 30 \* 1000

},

* Defines options for the browser context, such as running in headless mode, taking screenshots, recording videos, setting the base URL, and timeouts.

**9. Projects**

projects: [

{

name: 'adminSetup',

testMatch: /.\*\Admin.setup\.ts/,

retries: 1

},

{

name: 'projectName UI Test on Chrome',

use: {

...devices['Desktop Chrome'],

storageState: "playwright/.auth/admin.json",

},

dependencies: ['adminSetup'],

fullyParallel: true,

},

]

* Specifies different projects to run tests. Each project can have its own configuration, such as browser type, test match patterns, and dependencies on other projects.

### **Using the Configuration File**

**Global Setup**: Ensure that any necessary global setup (like setting environment variables) is done in `**utils/globalSetup.ts**`.

**Handling Projects**: You can define multiple projects within the configuration file, allowing you to run tests in different environments or with different settings. Each project can be executed separately or in parallel as configured.

By understanding and configuring `**playwright.config.ts**` effectively, you can control how tests are executed, manage different environments, and generate comprehensive reports, enhancing your overall testing framework.

**Running Tests**:

* To execute tests, use the command:

npm run execute:test1

* To execute only smoke test cases, use:

npx playwright test --grep @smoke

* (Ensure that the test case name is suffixed with `**@smoke**`.)

Note: The above sample commands interpret the shortkeys defined in script section of the package.json file and execute the complete command derrived from the script section.

**Viewing Test Results**:

* After running the tests using the above commands, the report will be generated and opened in the browser in HTML format.
* For example, an HTML report can be configured to open automatically by setting the reporter option in the `**playwright.config.ts**` file.

**Generating Allure Reports**:

npx allure serve allure-results – this command will display the execution results.

* To clean the existing Allure report, use the command:

npm run rimraf allure-report/ rimraf allure-results/ && rimraf test-results

* To generate a fresh report with a shorter command, add the below line to package.json in script section and use that key as the command:

npm run allure generate allure-results -o allure-report --clean && allure open ./allure-report

* To open the Allure report, navigate to the generated `**allure-report**` directory and open the `**index.html**` file in a web browser.