**Test Plan**

**1. Types of Functional Tests Prioritized**

To ensure a comprehensive validation of the Saucedemo website’s critical functionalities, the following functional tests have been prioritised:

1. **Input Validation**: Tests will be conducted to ensure proper handling of user inputs. For instance, login attempts with invalid credentials verify that appropriate error messages are displayed and that the system doesn’t proceed with unauthorised access. This helps prevent security risks and data integrity issues.

2. **Boundary Testing**: Boundary testing ensures the system’s stability and reliability under edge conditions. For instance, checking the functionality of adding items to the cart or removing them validates the cart’s behaviour under various scenarios, including an empty cart, a single item, or multiple items.

3. **User Flow Testing**: This includes verifying that essential user workflows, such as logging in, navigating through product listings, adding items to the cart, and completing the checkout process, work seamlessly. By testing complete user flows, we can validate that each page transition works as expected and that users can accomplish tasks without disruptions.

4. **Sorting Functionality**: Sorting tests validate that users can effectively arrange products based on parameters like price and name. This ensures that filtering options provided to users are functional and offer the expected experience.

**2. Rationale for Chosen Testing Approach**

The testing approach is primarily based on **end-to-end automation**, where critical functionalities and user interactions are validated from start to finish. This approach has been chosen for the following reasons:

1. **Comprehensive Coverage**: By automating end-to-end tests, we cover essential workflows and critical paths that users are likely to follow. This allows us to catch bugs that might slip through isolated unit or integration tests.

2. **Consistency and Reusability**: Using reusable functions for actions like logging in, adding items to the cart, and handling page navigation ensures that our code is modular and easy to maintain. This makes it straightforward to modify tests if the UI changes while retaining the overall structure and logic.

3. **Rapid Feedback**: Automated testing enables faster execution and feedback cycles. Instead of manually validating the website after every change, we can execute tests quickly, ensuring immediate detection of any functionality that breaks.

4. **Data-Driven Elements**: With parameterised data like login credentials and sorting expectations, we can easily modify input values without altering the test code, making extending our testing to multiple scenarios simple.

**3. Tools and Frameworks Considered for Automation**

The following tools and frameworks were considered and selected based on the project requirements and the nature of the tests:

1. **Playwright**: Playwright has been chosen as the main testing framework for automating browser interactions. It provides robust support for cross-browser testing and allows us to easily simulate various user actions, such as clicking, typing, and navigating. Playwright’s built-in expects assertions provide a streamlined approach for verifying outcomes directly within the test code.

2. **Modular and Data-Driven Approaches**:

• **Modular**: Functions are structured to handle repetitive tasks such as logging in and adding/removing items from the cart. This modularity enhances code reuse and maintainability, allowing us to focus on specific application parts as separate units.

• **Data-Driven**: Test data is stored separately and referenced dynamically, allowing us to easily modify test inputs like product names, prices, and user credentials. This approach makes tests more adaptable and allows testing multiple scenarios without redundant code.

3. **GitHub**: Version control is handled through GitHub, where all automation scripts are stored and managed. This setup ensures version tracking, collaborative access, and the ability to roll back to previous versions if needed.

4. **Visual Studio Code (VS Code)**: This code editor is used for writing, managing, and debugging the Playwright test scripts. VS Code’s integrated terminal and debugging tools are ideal for developing and refining automated tests.

**4. File Structure and Naming Conventions**

For clarity and organisation, the following files and directories are utilised in the automation framework:

**Test Files**:

• standardCheckout.spec.ts: Contains test cases for the standard user checkout flow.

• removeItemAndContinueShopping.spec.ts: Validates adding and removing an item from the cart and continuing to shop.

• emptyCartCheckout.spec.ts: Tests the checkout process with an empty cart, identifying potential bugs.

• invalidLogin.spec.ts: Ensures the system handles invalid login attempts appropriately.

• sorting.spec.ts: Verifies the sorting functionality for product listings.

**Utility Files**:

• selectors.ts: Defines selectors used throughout the test scripts for various elements on the webpage.

• testData.ts: Contains parameterized data for use in the tests, such as user credentials, item names, and expected outcomes.