DAY 1 Html, CSS, Javascript

HTML (Describes the structure of the web page)

Key Concepts:

- Document structure with <!DOCTYPE>, <html>, <head>, and <body> tags
- Semantic elements like headings (<h1> to <h6>)
- Text formatting with paragraph tags (<p>)
- Interactive elements like buttons
- Form creation with various input types

Code Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>My First Page</title>
  <link rel="stylesheet" href="styles.css">
  <script src="script.js"></script>
</head>
<body>
  <div>
    <h1>Main Heading</h1>
    <p id="content">This is a paragraph.</p>
    <button id="myButton" onclick="handleClick()">Click Me</button>
  </div>
</body>
</html>
```

CSS (Styling of the web page)

Key Concepts:

- Selectors (element, class, ID)
- Box model (margin, border, padding)
- Color and typography properties

Code Example:

```
body {
  font-family: sans-serif;
  background-color: #eaeaea;
  margin: 0;
  display: flex;
  justify-content: center;
  align-items: center;
  height: 100vh;
}

.container {
  background-color: white;
  padding: 20px;
  border-radius: 6px;
  width: 280px;
  box-shadow: 0 0 5px rgba(0, 0, 0, 0.1);
}

#content {
  font-style: italic;
}
```

JavaScript (Behaviour of the web page)

Key Concepts:

- Variables and data types
- Control structures (conditionals, loops)
- Functions
- Array manipulation methods
- Local storage for data persistence
- DOM manipulation

Code Examples:

Variables and Basic Operations:

```
// Variable declarations
let username = "Abden";
let age = 25;
const isLoggedIn = true;

// Array operations
let names = ["Aaryan", "Abden", "Alex"];
names.forEach(name => console.log(name));
names.push("New Name");
```

Local Storage:

```
// Store user details
function storeUserDetails() {
  const userDetails = {
    fullName: document.getElementById('fullname').value,
    username: document.getElementById('username').value,
    email: document.getElementById('email').value
  };
```

```
    localStorage.setItem('userDetails', JSON.stringify(userDetails));
}

// Retrieve user details
function displayUserDetails() {
    const userDetails = localStorage.getItem('userDetails');
    if (userDetails) {
        const data = JSON.parse(userDetails);
        document.getElementById('userDetails').innerHTML = `
            <p><strong>Name:</strong> ${data.fullName}</p>
            <p><strong>Username:</strong> ${data.username}</p>
        `;
    }
}
```

Git Version Control

Git provides tools for tracking changes and collaborating on code.

Key Git Commands:

```
git init
git add index.html
git add .
git commit -m "Add login functionality"
git remote add origin https://github.com/abden/repo.git
git push -u origin main
git pull origin main
```

DAY 2 Event, Validation, Promises

EVENT (Something a browser or user does)

- Page has finished loading
- HTML input was changed
- An HTML button was clicked (submit button onclick())
- **setTimeout** delays a task to run once after a set time.
- **setInterval** keeps running a task repeatedly at regular time intervals.
- **clearInterval** stops the repeated task started by setInterval.

Page on load event

```
<body onload="alert('Page is loaded')">
```

Button Clicked event

```
<button
    onclick="this.innerHTML = Date()">Time is :
</button>
```

Mouse Entered/left event

```
<button
    onmouseenter="this.innerHTML = 'Mouse Entered'"
    onmouseleave="this.innerHTML = 'Mouse Left'">Mouse
</button>
```

Key pressed

```
<input type="text" onkeypress="pressed()"/>
```

Set Timeout, setInterval, clearInterval (There are other events other than these 3)

```
<button onclick="setTimeout(buttonclick,2000)">Timeout</button>\
<button onclick="meow= setInterval(btnclick,2000)">Interval</button>
<button onclick="clearInterval(meow)">Stop Interval</button>
```

Form Validation

- Form to add vendors, users, with ID, name, rating, and products and so on
- Validation for unique IDs, name length (>=5), and products
- Dynamic product fields (add/remove)
- Display vendors sorted by rating
- Search functionality
- View detailed information in a modal
- Clear all vendor data functionality
- Store vendors as objects in **localStorage**
- Sort vendors by rating
- Validate input before saving
- Learned how to delete data from local Storage

Promises

- Learned about Promises, which are objects representing the eventual completion or failure of asynchronous operations.
- Can be used for Error Handling
- States: **resolve(), reject()**
- Similar to try catch finally

Fetch an image from a UPI

- Get image from an API
- Store it into a variable
- fetch(), then()
- async function

  async function getCat()
  await fetch(url).then(res => res.json()).then(data => data.url)

- await to fetch a data from API

Day 3 TypeScript

- **npm -v**
- **npm init -y**
- **npm install typescript –-save-dev**
- **npm tsc init**
- **npx tsc filename.ts**
- **node filename.js**

TypeScript adds static typing to JavaScript, which helps **catch errors before runtime.**

- **Basic Types & Functions**:

Started with simple functions and type annotations

```
let name: string = "aaryan";
```

- **Interfaces**:

Learned how to define object structures (similar to Java interfaces)

```
interface User {
 name: string;
 id: number;
 role: roles;
}
```

- **Custom Types**:

Created union types to restrict values

```
type roles = "Trainer" | "Developer" | "Tester";
```

- **Classes**:

Implemented classes with constructors and properties

```
class UserAccount {
 name: string;
 id: number;
}
```

- **Interface Implementation**:

Made classes implement interfaces

```
class Vendor implements VendorInterface {...}
```

- **Optional Properties**:

Used the ? symbol for optional properties

```
type Vendor = {
  lname?: string; // optional
}
```

- **Arrays & Sorting**:

Worked with typed arrays and sorting functions

```
const vendors: Vendor[] = [...];
vendors.sort((a, b) => a.id - b.id);
```

- **Type 'any'**:

Used for values where the type isn't known

```
productName: any;
```

- **String Interpolation**:

Used template literals for string formatting

```
console.log(` ID: ${e.id}, Name: ${e.name}` );
```

- **Conditional Expressions**:

Implemented ternary operators

```
e.lname != undefined ? e.name + " " + e.lname : e.name
```

- **Promise to Get the response from API:**

Get the data from url in json format data and display the json data using Promise<Response>.

```
function getFacts(url: string): Promise<Response> {
  return fetch(url).then(res => res.json());
getFacts("https://cataas.com/cat?width=200;height=200;json=true").then(data => console.log(data));
```

Day 4 MCQ Assignment, Practicing Typescript

1. **Microsoft**

Microsoft created and maintains TypeScript as an open-source programming anguage. It can be used to develop JS applications in both client and server side.

2. **Node**

Commonly used with node.js which runs javascript on server side and other backend technologies aswell.

3. **JavaScipt**

Type Script is superset of JavaScript that adds static typing. After compilation TypeScript is converted/compiled to JavaScript.

4. **extends**

Inheritance is implemented using the extends keyword similar to JavaScript Inheritance

5. **var x = "string";**

The variable has no type defined after declaring the variable there should be a colon space type.

This a JavaScript syntax, not a TypeScript syntax

6. **var x: number = 999;**

This is TypeScript syntax,

The variable x of number type is stored the value 999.

7 .**ts**

TypeScript files are typically stored in .ts format, for JavaScript it is .js

8. **tsc filename.ts**

tsc is the command to run TypeScript compiler to compile it to .js file, we can also executa a .ts file using node tsc filename.ts

9. **tsc filename.ts -w**

Make the compiler continuously watch for changes in TypeScript files and recompile automatically when changes are detected.

10. **super()**

use the super() keyword within the child class's constructor.

```
class Parent {
  constructor(public name: string) {
    console.log("Parent constructor called");
  }
}

class Child extends Parent {
  constructor(name: string, public age: number) {
    super(name);
    console.log("Child constructor called");
  }
}
const child = new Child("John", 25);
```

1. **Sort**

Practiced on how to use the sort function efficiently.

```
vendors.sort((a, b) => a.id - b.id);
```

2. **Filter**

Go through each item in the products array.

For every item (p), check if its price is less than or equal to maxPrice.

Only the products that pass this test will be included in the result.

filter() is a built-in method that returns a new array of items that match a condition.

```
products.filter(p => p.price <= maxPrice);
```

3. **Finding**

It stops at the first match.

If found, it prints the details

If not found, it says “Product not found.”

```
products.find(p => p.name === productName);
```

4. **Every/Some**

every - checks if all items match the condition.

some - checks if at least one item matches.

```
const allUnder100 = products.every(p => p.price < 100);

const anyUnder30 = products.some(p => p.price < 30);
```

5. **Mapping**

map() goes through every product.

For each product, it returns just the name.

You get a new array of only names (original array stays the same).

```
// Map to get only names

const productNames = products.map(p => p.name);
```

6. **Reduce**

(sum, p) => sum + p.price \\Adds the current product's price to the total sum.

0 is the starting value of the sum.

It goes through all items and returns one final result:

```
const totalPrice = products.reduce((sum, p) => sum + p.price, 0);
```

1. **Decorators/Modules**

Decorators = add logic to classes/functions automatically.

```
// Enable "experimentalDecorators": true in tsconfig.json

function Logger(constructor: Function) {

 console.log("Class created:", constructor.name);

}

@Logger

class Product {

 constructor(public name: string, public price: number) {}

}

const p1 = new Product("Book", 100);
```

Modules = split code into files and share using export / import

mathUtils.ts (a separate file)

```
export function add(a: number, b: number): number {
  return a + b;
}
export const PI = 3.14;
```

app.ts

```
import { add, PI } from './mathUtils';
let result = add(10, 5);
console.log(` Result: ${result}` );
console.log(` Value of PI: ${PI}` );
```

Day 5 Operators

1. **Keyof operator**

The keyof operator in TypeScript is used to extract the keys of an object type as a union of string literal types. It provides a way to ensure type safety when working with object properties dynamically.

```
function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
    return obj[key];
}
```

2. **Rest operator**

```
function sum(...numbers: number[]): number {
  return numbers.reduce((total, num) => total + num, 0);
}
const res = sum(1,2,3);
```

Collects all remaining arguments into an array. When you don't know how many arguments will be passed to a function, use rest operator.

Rest Operator is used in function **parameters**. **Spread Operator** (also ...) is used to **expand** elements, like:

```
sumof(...products)
```

3. **Overloading**

Define **multiple function signatures** for a single method, each with different parameter types or counts.

In TypeScript, **you define overloads with multiple function signatures**, and then provide **one actual implementation** that handles all cases.

```
speak(s: string): string;
speak(n: number): string;
speak(b: boolean): string;
```

These are the **overload declarations**. They tell TypeScript what calls are allowed

```
speak(arg: any): any {
    if (typeof arg === 'number') {
        return `Meow number ${arg}`;
    }
    if (typeof arg === 'string') {
        return `Meow string ${arg}`;
    }
    if (typeof arg === 'boolean') {
```

```
        return `Meow boolean ${arg}`;
    }
}
```

This is the **actual implementation** that handles **both overloads**. TypeScript only allows **one implementation**, and it must be compatible with all the declared signatures.

4. **Modules (import/export)**

Consider two files, one for importing and exporting. Now we will split into files and share using **import** and **export.**

```
//mathUtils.ts(a separate file)
export function add(a: number, b: number): number {
    return a + b;
}
export const PI = 3.14;
export function area(radius: number): number {
    return PI * radius * radius;
}
//sub.ts(a separate file)
export default function subtract(a: number, b: number): number {
    return a - b;
}
```

When we use **export**, we are making **functions, variables, classes, or interfaces available to other files**. **default** is used when you're exporting **one main thing** from a module.

```
//app.ts(another file)
import {add, PI } from './mathUtils';
let result = add(10, 5);
console.log(`Result: ${result}`);
console.log(`Value of PI: ${PI}`);

//subfun.ts(a separate file)
import subtract from './sub.ts';
console.log(subtract(5,10));
```

When we use import to **bring in code that was exported from another module.**

If we want to **import everything** from a module use import * as anyname from ‘./’;

```
import * as MathUtils from './mathUtils';

console.log(MathUtils.add(5,10));
console.log(MathUtils.PI);
```

To Rename things when importing we can do it like this

```
import { multiply as mul } from "./1_export";
console.log(mul(4, 5));
```

DAY 6 – Playwright

1. Playwright Installation

npm init playwright@latest

2. Page Object Model (POM)

It is a design pattern commonly used in **automation** testing

- Used with tools like **Playwright**, Selenium, Cypress
- Makes Tests more **maintainable**, **reusable**, and **readable**
- **test logic is separated from UI details.**
  In POM, each page or component of your application has a corresponding class (or object) that encapsulates:
- The **locators (selectors)** for elements on the page
- The **actions** you can perform on the page
- Any **assertions** or checks related to that page

3. Page Navigation

```
await page.goto('https://www.easemytrip.com/');
```

4. Title Assertion:

```
await expect(page).toHaveTitle(...)

//ensures the correct page is loaded.
```

5. Click Actions with Visibility Check

Always check visibility before clicking for stability:

```
await expect(button).toBeVisible();
await button.click();
```

6. Waiting Strategies

Use **waitForTimeout()** for debugging or better yet, wait for specific elements or states.

7. **Element locators**

XPath is powerful but use **CSS selectors** when possible for better performance.

```
const getStarted = page.getByText('Try Typescript Now');
const username = page.locator("#user-name");
```

8. **Form Interactions / Filling Inputs**

```
await username.fill("standard_user");
await password.fill("secret_sauce");
```

Features:

- Auto wait
- Cross browser Compatibility – Chrome, Firefox, Edge, Webkit.
- Multi-platform – Mac OS, Windows, Linux
- Multilingual Flexibility – Supports Java, Python, Java Script, C#, .NET

Advance Feature:

- Tracing and Debugging – take screenshot, video recording
- Network Interception
- Browser Context Management
- Codegen Tool
- Java Script – Asynchronous this one does not execute the order wise.
- **await – used to wait each action**

Installation of Playwright:

- npm install playwright@latest

Run the test:

- **npx playwright test**

Run the test headed mode:

- **npx playwright test –headed**

Run the test UI mode:

- npx playwright test –ui

Get the report in html page:

- npx playwright show-report

Import the playwright and expect in .ts file

- import {test, expect} from "@playwright/test";

1. **Navigate to the website**

```
import { test, expect } from '@playwright/test';

test("Place the Order", async({page})=>{
    await page.goto("https://www.saucedemo.com/");
```

2. **Assertion in Playwright:**

```
const orderConfirmation = await page.locator(".complete-header");
await expect(orderConfirmation).toHaveText("Thank you for your order!");
```

3. **Different type of assertion:**

await expect(locator).toBeChecked();

await expect(locator).not.toBeChecked();

await expect(locator).toBeDisabled();

await expect(locator).toBeEnabled();

await expect(locator).toBeEditable();

await expect(locator).toBeEmpty();

Example Automation Script:

```
test('End to end test for cart functionality', async ({ page }) => {
    await page.goto('https://www.saucedemo.com/');
    await expect(page).toHaveTitle("Swag Labs");
    const username = page.locator("#user-name");
    await username.fill("standard_user");
    const password = page.locator("#password");
    await password.fill("secret_sauce");
    const login = page.locator("#login-button");
    await login.click();
    const firstItem = page.locator(".inventory_item_name").first();
    await firstItem.click();
    const addToCart = page.locator("#add-to-cart");
    await addToCart.click();
    //I will now verify cart quantity is equals to one
    const cartIcon = page.locator(".shopping_cart_badge");
    await expect(cartIcon).toBeVisible();
    const cartQuantity = await cartIcon.textContent();
    if (cartQuantity === "1") {
        console.log("Cart quantity is 1");
    }
    cartIcon.click();
    const checkoutBtn = page.locator("#checkout");
    await expect(checkoutBtn).toBeVisible();
    await checkoutBtn.click();
    await page.fill("#first-name", "John");
    await page.fill("#last-name", "Doe");
    await page.fill("#postal-code", "12345");
    await page.click("#continue");
    const finishBtn = page.locator("#finish");
    await expect(finishBtn).toBeVisible();
    await finishBtn.click();
    const orderConfirmation = page.locator(".complete-header");
    await expect(orderConfirmation).toBeVisible();
    const orderText = await orderConfirmation.textContent();
    await expect(orderText).toContain("Thank you for your order!");


})
```

End-to-End Cart Functionality Test

This automated test script, written using Playwright, verifies the full cart and checkout flow on the **Swag Labs** demo website. The test covers the following steps:

1. **Navigation & Login:**

- Navigates to https://www.saucedemo.com/
- Asserts that the page title is "Swag Labs"
- Logs in using standard credentials: standard_user / secret_sauce

2. **Item Selection & Cart Interaction:**
   - Clicks on the first listed item
   - Adds the item to the cart
   - Asserts the cart badge is visible
   - Confirms that the cart quantity is correctly updated to **1**
3. **Checkout Process:**
   - Proceeds to the cart and initiates checkout
   - Fills out checkout information with mock user data (name and postal code)
   - Continues to the order summary and completes the purchase
4. **Order Confirmation:**
   - Verifies that the final confirmation message appears
   - Asserts the confirmation contains the expected message: *"Thank you for your order!"*

This test ensures that the cart functionality—from product selection to successful order placement—works as intended.

Day 2

1. **To execut a particular test**

npx playwright test -g "has title"

2. **To Rerun a failed test**

npx playwright test --last-failed

3. **To run the project automation on chromium and enable trace on as well:**

npx playwright test --project chromium --trace on

4. **Playwright's defineConfig**

To set up how my tests should run. This config gives me flexibility across browsers, environment control, retries, and even geolocation emulation.

5. **General Settings:**

testDir: Test files are located in the **./tests** directory.

fullyParallel: Enables full parallelism across test files.

forbidOnly: Ensures **.only is not left** accidentally in source (enabled on CI).

retries: Retries failed tests up to **2 times on CI**, none locally.

workers: Runs with **a single worker on CI** to avoid parallel execution issues.

reporter: Uses the **'html' reporter** for test results.

6. **use block (Shared Settings for All Projects):**

```
use: {
    /* Base URL to use in actions like `await page.goto('/')`. */
    // baseURL: 'http://127.0.0.1:3000',
    baseURL: 'https://playwright.dev/',
    /* Collect trace when retrying the failed test. See
    https://playwright.dev/docs/trace-viewer */
    trace: 'on-first-retry',
    screenshot: 'only-on-failure',
    video: 'on-first-retry'


    // Network configs
    acceptDownloads: false,


    extraHTTPHeaders: {
      'X-My-Header': 'value',
    },

    httpCredentials: {
      username: 'user',
      password: 'pass'
    },

    ignoreHTTPSErrors: true,

    offline: true,

    proxy: {
```

```
    server: '',
    bypass: 'localhost'
  }


  // Emulation options

  colorScheme: 'dark',
  geolocation: { longitude: 12.343535, latitude: 45.56575 },
  locale: 'en-GB',
  permissions: ['geolocation'],
  timezoneId: '',
  viewport: { width: 1280, height: 720 }
},
```

7. **Browser Projects**

```
/* Configure projects for major browsers */
  projects: [
    {
      name: 'chromium',
      use: { ...devices['Desktop Chrome'] },
    },

    {
      name: 'firefox',
      use: { ...devices['Desktop Firefox'] },
    },

    {
      name: 'webkit',
      use: { ...devices['Desktop Safari'] },
    },

    /* Test against mobile viewports. */
    {
      name: 'Mobile Chrome',
      use: { ...devices['Pixel 5'] },
    },
    {
      name: 'Mobile Safari',
      use: { ...devices['iPhone 12'] },
    },

    /* Test against branded browsers. */
    {
      name: 'Microsoft Edge',
      use: { ...devices['Desktop Edge'], channel: 'msedge' },
    },
```

```
    {
      name: 'Google Chrome',
      use: { ...devices['Desktop Chrome'], channel: 'chrome' },
    },
```

8. **Dev Server (Commented-out):**

Option to run a dev server (npm run start) before running tests.

Can be reused locally but starts fresh on CI.

```
/* Run your local dev server before starting the tests */
  webServer: {
    command: 'npm run start',
    url: 'http://127.0.0.1:3000',
    reuseExistingServer: !process.env.CI,
  },
```

9. **Special Test States:**

- **test.only:** Focuses the test runner only on this test (focus on this test case).
- **test.skip**: Skips the skipped test case.
- **Conditional skip**:
    - Skips a test for chromium only using test.skip(browserName === 'chromium').

- **test.beforeAll**
    - **Pre-test setup logic before all tests run.**
- **Handles**:
    - Waiting for and interacting with a popup (waitForEvent('popup')).
    - Inside the popup, finds a close button and clicks it.
    - Listens for browser dialogs like alert(), confirm(), prompt() and accepts them automatically.

Day 3 – Playwright

1. **Fixture Setup**

   Two types of Fixtures:

- Custom
- Bulit In – page, context, browser

Custom fixtures are declared to inject instances of each page class (LoginPage, ProductsPage, etc.). Fixtures act as a **bridge** between the test logic and the application's UI by providing **reusable setup** and context (like **page objects**, user sessions, etc.) to tests.

```
type SaucedemoFixtures = {
    loginPage: LoginPage,
    productsPage: ProductsPage,
    productDetailsPage: ProductDetailsPage,
    cartPage: CartPage,
    checkoutPage: CheckoutPage,
    orderConfirmationPage: OrderConfirmationPage
};
export const test = base.extend<SaucedemoFixtures>({
    loginPage: async ({ page }, use) => {
        const loginPage = new LoginPage(page);
        await loginPage.goto();
        await use(loginPage);
    },
```

2. **Login in beforeEach Hook**

Automatically logs in a user before every test

```
test.beforeEach(async ({ loginPage }) => {
  await loginPage.goto();
  await loginPage.login("standard_user", "secret_sauce");
});
```

3. **Parallelized End-To-End Testing**

```
test.describe.configure({ mode: 'parallel' }); //serial
```

Using this to run tests faster by adding necessary workers. If you want to run in serial use serial instead of parallel.

Or In playwright.config.ts file set the parallelism, it will test the testcase fully parallel mode.

```
fullyParallel: true,
```

Or we can achieve parallelism by specifying the number of workers required to achieve it in the CLI like this:

npx playwright test --project chromium -workers 3

4. **Parameterized Testing / Data-Driven Testing**

Iterates over different sets of user details specified by us to verify the functionality and test flow.

The same test logic is executed multiple times with **different sets of input data**. This helps verify **how the application behaves with various user inputs or scenarios**, improving test coverage and **reducing code duplication.**

```
[
  { firstName: 'Aaryan', lastName: 'ust'},
  { firstName: 'Abden', lastName: 'ust'},
  { firstName: 'Deepak', lastName: 'ust'},
].forEach(({ firstName}) => {
  test.only(`End to end test for cart functionality of ${firstName}`, async ({
productsPage }) => {

  await checkoutPage.enterDetails(firstName, lastName, pin);
  });
})
```

5. **npx playwright test --max-failures=2**

Runs all Playwright tests, but stops the test run after 2 test failures. Useful for saving time during debugging or CI runs by not continuing after multiple failures.

Or we can specify it in the playwright.config.ts in **defineConfig:**

```
maxFailures: 2,
```

6. An End-To-End Scenario

```
test.describe.configure({ mode: 'parallel' });

test.beforeEach(async ({ loginPage }) => {
  await loginPage.goto();
  await loginPage.login("standard_user", "secret_sauce");
});

[
  { firstName: 'Aaryan', lastName: 'ust', pin: '12345', orderConfirmMsg:
'Thank you for your order!' },
  { firstName: 'Abden', lastName: 'raj', pin: '12345', orderConfirmMsg: 'Thank
you for your order!' },
  { firstName: 'Deepak', lastName: 'Antony', pin: '12345', orderConfirmMsg:
'Thank you for your order!' },

].forEach(({ firstName, lastName, pin, orderConfirmMsg }) => {
  test.only(`End to end test for cart functionality of ${firstName}`, async ({
productsPage, productDetailsPage, cartPage, checkoutPage,
orderConfirmationPage }) => {
    await productsPage.clickFirstItem();
    await productDetailsPage.addToCart();
    const cartQuantity = await productDetailsPage.getCartQuantity();
    expect(cartQuantity).toBe("1");
    console.log("Cart quantity is 1");
    await productDetailsPage.gotocheckout();
    await cartPage.gotocheckout();
    await checkoutPage.enterDetails(firstName, lastName, pin);
    await checkoutPage.finishOrder();
    await orderConfirmationPage.verifyOrderConfirmation(orderConfirmMsg);
    await orderConfirmationPage.backToHome();
    await productsPage.verifyProductPageIsDisplayed();
  });
})
```

Day 4 – Amazon Test Case Assignment

Best Sellers Page Tests

This tests validates various functionalities of the **Amazon Best Sellers** page using Playwright and a bestSellersPage page object with the help of fixtures. The tests are configured to run **in parallel** for efficiency.

```
test.describe.configure({ mode: 'parallel' });
```

Test Coverage Overview:

1. **Navigation:**
    - Open the Best Sellers page before each test.

```
test.describe('Best Sellers Page Tests', () => {
  test.beforeEach(async ({ bestSellersPage }) => {
    await bestSellersPage.navigate();
  });
```

2. **Search Functionality:**
    - Search for products like **Laptop**, **Mobile**, and **Camera**.
    - Verify search results are displayed.

```
[
    { searchKeyword: 'Laptop' },
    { searchKeyword: 'Mobile' },
    { searchKeyword: 'Camera' },
  ].forEach(({ searchKeyword }) => {
    test(`verify and search results for "${searchKeyword}"`, async ({
bestSellersPage }) => {
      await bestSellersPage.searchProduct(searchKeyword);
      await expect(bestSellersPage.products.first()).toBeVisible();
    });
  });
```

3. **Page Title Validation:**
    - Ensure the page title matches:
      "Amazon.in Bestsellers: The most popular items on Amazon"

```
test('Verify Page Title', async ({ bestSellersPage }) => {
```

```
    await bestSellersPage.verifyPageTitleHasTitle("Amazon.in Bestsellers:
The most popular items on Amazon");
  });
```

4. **Category Display:**
    - Check that categories are visible.
    - Specifically verify presence of **Electronics** and **Books** categories.

```
test('Verify Categories Are Displayed', async ({ bestSellersPage }) => {
    await bestSellersPage.verifyCategoriesDisplayed();
  });
```

5. **Header Check:**
    - Confirm the presence of the **"Bestsellers"** header.

```
async verifyHasHeader(headervalue: string) {
    await expect(this.header.first()).toBeVisible();
    const headerText = await this.header.first().textContent();
    expect(headerText).toContain(headervalue);
  }
```

6. **Product Listings:**
    - Verify that products are listed on the page.

```
async verifyProductsDisplayed() {
    await expect(this.bestproducts.first()).toBeVisible();
  }
```

7. **Product Navigation:**
    - Simulate clicking on a product.
    - Confirm that product details (like an "Add to Cart" button) are displayed.

```
test('Verify Clicking a Product Opens Product Page', async ({
bestSellersPage }) => {
    await bestSellersPage.userClicksAProduct();
    await expect(bestSellersPage.addToCartButton.first()).toBeVisible();
  });
```

8. **Category Navigation:**
    - Navigate to a different category (e.g., **Books**) and verify the header updates appropriately.

```
test('Verify Navigation to a Different Category', async ({ bestSellersPage
}) => {
    await bestSellersPage.verifyCategoriesDisplayed();
    await bestSellersPage.clickCategory('Books');
    await expect(bestSellersPage.verifyCategoriesDisplayed).toBeTruthy();
    await bestSellersPage.verifyHasBooksHeader('Bestsellers in Books');
  });
```

9. **Sorting/Filtering Options:**
    - Search for "camera" and apply a sorting filter (e.g., **Price: Low to High**).

```
async applySortOption(optionValue: string) {
    console.log(this.sortDropdown);
    if (await this.sortDropdown.isVisible()) {
      await this.sortDropdown.selectOption(optionValue);
      const priceValue1 = await this.price.first().textContent();
      const price1 = parseFloat(priceValue1 ?? "0");
      const priceValue2 = await this.price.nth(1).textContent();
      const price2 = parseFloat(priceValue2 ?? "0");
      console.log(`Price of first product: ${price1}`);
      if (price1 < price2) {
        console.log("Products sorted successfully in ascending order.");
      }
    }
  }
```

10. **Pagination:**
    - Navigate to the next page within a category.

```
async goToNextPage() {
    if (await this.paginationNext.isVisible()) {
      await this.paginationNext.click();
      await expect(this.page).toHaveURL(  /pg=2/);
    }
  }
```

11. Add to Cart Flow:

- Add a product to the cart directly from the Best Sellers page after navigating to the product details page.

```
async addFirstProductToCart() {
    const addToCartButton = this.page.locator('#add-to-cart-button');
    if (await addToCartButton.isVisible()) {
      await addToCartButton.first().click();
      await expect(this.cartCount).not.toHaveText('0');
    }
  }
```

12. Verify user can navigate each department

- To ensure the user can navigate through every department listed on the Best Sellers page.
- After navigation, verify that each click leads to the correct page.

```
async validateEachDepartmentNavigateCorrectPage() {
    const departmentCount = await this.categories.count();
    console.log('Number of departments:', departmentCount);
    for (let i = 0; i < departmentCount; i++) {
      await this.categories.nth(i).click();
      await this.page.waitForLoadState();
      await expect(await this.page.locator('#zg_banner_text')).toBeVisible();
      await this.page.goBack();
    }
  }
```

Day 5 – Snapdeal Assignment

1. **Created a snapdeal.spec.ts for all the test cases to be automated via playwright typescript**

```
test.only('Verify End to End flow', async ({homePage, productsPage}) => {
        await homePage.verifyUserIsOnSnapdealHomePage();
        await homePage.userSearchForAProduct("Shoes for Women");
        await productsPage.verifyUserIsOnSnapdealProductsPage();
        const productDetailsTab = await productsPage.userSelectProduct();
        await productDetailsTab.waitForLoadState();
        const productDetailsPage = new ProductDetailsPage(productDetailsTab);
        await productDetailsPage.verifyUserIsOnSnapdealProductDetailsPage();
        await productDetailsPage.userAddToCart();
        const cartQuantity = await productDetailsPage.verifyCartQuantity();
        await expect(cartQuantity).toBe(1);
        await productDetailsPage.userClickOnCartIcon();
    })
```

2. **Created a snapdeal.fixture.ts for the fixtures to be implemented**

```
type SnapdealFixtures = {
    homePage: HomePage;
    productsPage: ProductsPage;
    productDetailsPage: ProductDetailsPage;
}
```

3. **Created Page Objects for the interaction with the UI elements in a structured and well efficient manner**

```
import { Page, Locator } from '@playwright/test';
import { expect } from './Snapdeal.fixture';

export class HomePage {

    private readonly searchInput: Locator;

    constructor(public readonly page: Page) {
        this.searchInput = page.locator("#inputValEnter");
    }

    async navigate() {
        await this.page.goto("/");
    }

    async verifyUserIsOnSnapdealHomePage() {
```

```
        await expect(this.page).toHaveTitle("Shop Online for Men, Women & Kids
Clothing, Shoes, Home Decor Items");
    }

    async userSearchForAProduct(productName : string) {
        await this.searchInput.isVisible();
        await this.searchInput.fill(productName);
        await this.searchInput.press("Enter");
        await this.searchInput.press("Enter");
    }
}
```

4. **Implement switching to another tab in a browser context using Promises and switch backing to the original page as well**

```
export async function switchToNewTab(context: BrowserContext, action: () =>
Promise<void>): Promise<Page> {
    const [newPage] = await Promise.all([
        context.waitForEvent('page'),
        action()
    ]);
    await newPage.waitForLoadState();
    return newPage;
}

export async function switchBackToPage(page: Page) {
    await page.bringToFront();
}
```

Day 6 – Clocks, Alert, Web Server

1. **HTML Structure (index.html):**

- Sets up a basic webpage with a div to display the current time, and three buttons: "Alert", "Confirm", and "Prompt".

2. **JavaScript Logic (index.js):**

- **Clock Functionality:**
    - **renderTime**() function: Updates the content of the div with the id current-time to display the current date and time, formatted using **toLocaleString**().
    - **setInterval**(): Calls **renderTime**() every 1000 milliseconds (1 second) to update the time dynamically.
- **Button Event Listeners:**
    - **Alert Button:** When clicked, displays a simple alert box with the message "This is an alert!".
    - **Confirm Button:** When clicked, displays a confirmation dialog with the message "Do you confirm this action?". The result (true/false) is logged to the console.
    - **Prompt Button:** When clicked, displays a prompt dialog asking the user to enter their name, with "Default Name" as the default value. The entered value (or null if canceled) is logged to the console.

3. **Playwright Tests:**

- **Test Suite:** A test suite named "Clock and Alert Tests" is defined.
- **Test 1: "Test with predefined time"**
    - Navigates to the specified HTML page.
    - Uses page.evaluate() to override the global Date.now() function, effectively freezing time at a specific date and time (2024-02-02T10:00:00).
    - Asserts that the text content of the element with the data-testid attribute current-time matches the expected time ("2/2/2024, 10:00:00 AM").
    - Changes the mocked Date.now() to a new time (2024-02-02T11:30:00).

- Asserts that the time displayed is updated to the new mocked time ("2/2/2024, 11:30:00 AM").

```
test.describe('Clock and Alert Tests', () => {
  test('Test with predefined time', async ({ page }) => {
    await
page.goto('http://127.0.0.1:5500/Playwright_UST/Day_6_Playwright_Commands/src/
index.html');
    await page.evaluate(() => {
      const fixedDate = new Date('2024-02-02T10:00:00');
      Date.now = () => fixedDate.getTime();
    });
    await expect(page.getByTestId('current-time')).toHaveText('2/2/2024,
10:00:00 AM');
    await page.evaluate(() => {
      const fixedDate = new Date('2024-02-02T11:30:00');
      Date.now = () => fixedDate.getTime();
    });
    await expect(page.getByTestId('current-time')).toHaveText('2/2/2024,
11:30:00 AM');
  });
```

- **Test 2: "Test alert, confirm, and prompt buttons"**
  - Navigates to the HTML page.
  - Sets up a dialog event listener to handle alert, confirm, and prompt dialogs.
  - **Dialog Handling:** Inside the dialog event listener:
    - Checks the dialog.type() to determine the type of dialog (alert, confirm, or prompt).
    - Asserts that the dialog.message() matches the expected message for each dialog type.
    - Calls dialog.accept() to close the alert and confirm dialogs, or dialog.accept('Playwright User') to close the prompt dialog and enter "Playwright User" as the input.
  - Clicks the "Alert", "Confirm", and "Prompt" buttons in sequence.

```
test('Test alert, confirm, and prompt buttons', async ({ page }) => {
    await
page.goto('http://127.0.0.1:5500/Playwright_UST/Day_6_Playwright_Commands/src/
index.html');
    page.on('dialog', async (dialog) => {
      if (dialog.type() === 'alert') {
```

```
        expect(dialog.message()).toBe('This is an alert!');
        await dialog.accept();
      } else if (dialog.type() === 'confirm') {
        expect(dialog.message()).toBe('Do you confirm this action?');
        await dialog.accept();
      } else if (dialog.type() === 'prompt') {
        expect(dialog.message()).toBe('Please enter your name:');
        await dialog.accept('Playwright User');
      }
    });
    await page.getByTestId('alert-button').click();
    await page.getByTestId('confirm-button').click();
    await page.getByTestId('prompt-button').click();
  });
});
```

4. **Host the web server in local system and run the test automation projects locally.**

```
webServer: {
    command: 'npm run start',
    url: 'http://127.0.0.1:5500',
    reuseExistingServer: !process.env.CI,
  },
```

5. **Sharding**

- Can further scale Playwright test execution by running tests on multiple machines simultaneously.
- We call this mode of operation "sharding". Sharding in Playwright means splitting your tests into smaller parts called "shards".
- Each shard is like a separate job that can run independently. The whole purpose is to divide your tests to **speed up test runtime.**
- **npx playwright test --shard=1/4**
- **npx playwright test --shard=2/4**
- **npx playwright test --shard=3/4**
- **npx playwright test --shard=4/4**