```
ॐ Kash Chauhan ॐ
```

Framework from Scratch --> Playwright + TypeScript --> OpenCart

Install Playwright first

Steps:

```
    1. Install Dependencies (Packages)
    2. Project Folder Structure
    3. Update playwright.config.ts
    4. Create test.config.ts
    5. Create POM (Page Object Model) Classes, 9 files
    6. Create Test Data Files - create under test data folder -> login.json and
login.csv
    7. Utility files (utils) - reusable, dataProvider.ts and
randomDataGenerator.ts
    8. Create Test cases
```

Step: 1 -> Dependencies (Packages):

```
- npm install csv-parse
- npm install xlsx
- npm install -D allure-playwright
- npm install @faker-js/faker
```

Step: 2 -> Project Folder Structure

```
- tests/                  -> test files(cases)
- pages/                  -> Page Object Model (classes)
- utils/                  -> Utility Functions
- data/                   -> Test data files (JSON, CSV)
- reports/                -> Generated test reports
- test.config.ts          -> Test configuration values
- playwright.config.ts    -> Playwright main configuration
- package.json
```

Step: 3 -> playwright.config.ts

```
  - Update all config as required
```

Step: 4 -> test.config.ts -> reusable --> open url, credentials, product details

```
  - this is class file for global test data
  - for common links, username, passwords, product details
  - we do not need to add these details every time
  - username: kash@xyz.com
  - password: test@123
```

Step: 5 -> POM (Page Object Model) Classes

```
  - Create files in the pages/ folder for each page:
      1. HomePage.ts
      2. RegistrationPage.ts
      3. LoginPage.ts
      4. LogoutPage.ts
      5. MyAccountPage.ts
      6. ProductPage.ts
      7. SearchResultsPage.ts
      8. ShoppingCartPage.ts
      9. CheckoutPage.ts

  - Each class should contain methods for interacting with UI elements of that
page.


1. HomePage.ts --> Click on My Account and Registration (Login)
  - import {Page, expect, Locator} from "@playwright/test";
  - export class HomePage{ }
  - locators
  - constructor
  - action methods (try and catch method)
      - check if HomePage exists
      - click My Account Link
      - click Registration Link
      - click Login Link
      - enter Product name in the Searchbox
      - click Search Button

2. RegistrationPage.ts --> Fill the form with all details
  - import {Page, expect, Locator} from "@playwright/test";
  - export class HomePage{ }
  - locators
```

```
  - constructor
  - action methods for Registration form
      - Fill First Name
      - Fill Last Name
      - Fill Email
      - Fill Telephone
      - Fill Password
      - Fill Confirm Password
      - Check policy checkbox
      - Click continue button
      - Confirmation Message

3. LoginPage.ts
  - import {Page, Locator} from '@playwright/test';
  - export class LoginPage { }
  - locators
  - constructor
  - action methods to login
    - Email address on Login page
    - Password on Login page
    - Click Login Button
    - complete login action (all methods in one)

4. MyAccountPage.ts
    - import {Page, Locator, expect} from '@playwright/test';
    - import { LogoutPage } from './LogoutPage';
    - locators
    - constructor
    - action methods
      - Verify if My Account Page is displayed (try and catch)
      - Click Logout link (try and catch)
      - Alternative method to return page exists using Title

5. LogoutPage.ts
    - import {Page, Locator } from "@playwright/test";
    - import { HomePage } from "./HomePage";
    - locators
    - constructor
    - action methods
      - Click continue button after Logout
      - Verify Continue Button is visible


6. ProductPage.ts
   - import { Page, Locator, expect} from "@playwright/test";
   - import { ShoppingCartPage } from "./ShoppingCartPage";
   - locators
   - constructor
   - action methods
      - Sets the Product Quantity
      - Add Product to Cart
```

```
      - Check if confirmation message is visible
      - Click on Items button to navigate to cart
      - Click on View Cart link
      - Complete workflow to add product to cart

7. SearchResultsPage.ts
   - import { Page, Locator } from "@playwright/test";
   - import { ProductPage } from "./ProductPage";
   - locators
   - constructor
   - action methods
      - Verify if Search Results page exists by checking header text
      - Check if product exists in the search results by its name - parameter
productName
      - Select product from search results by its name - parameter productName
      - Get count of the product in search results

8. ShoppingCartPage.ts
  - import { Page, Locator } from "@playwright/test";
  - import { CheckoutPage } from "./CheckoutPage";
  - locators
  - constructor
  - action methods
      - Get the total price from the shopping cart
      - Click on the Checkout button
      - Verify if shopping cart page is loaded

9. CheckoutPage.ts
  - import {Page, expect, Locator} from "@playwright/test";
  - locators
  - constructor
  - action methods
      - Check if checkout page exists
      - Choose checkout option
      - Click on continue button
      - Form field methods
      - Continue button methods
      - Delivery method
      - Terms and conditions
      - Order confirmation
      - Handle alert if present
```

Step: 6 -> Add Test Data Files - create under test data folder

```
  - logindata.json
  - logindata.csv
  - logindata.xl (generally we do not use)

  - use for data-driven testing
```

```
  - test data need to create only for scenario with multiple sets of data, no for
every test case
  - every test case will not be the data driven
  - certain scenarios will need to test with different sets of data so only for
them need to add test data files.


* logindata.json:
  - Two scenarios: valid and invalid credentials
  - valid: login should be successful.


* logindata.csv:
  - testName,email,password,expected
  - Valid login,kash@xyz.com,test@123,success
  - Invalid login,xyaere@xyz.com,abcxye,failure
```

Step: 7 -> Utility files (utils) - reusable

```
  - to read data from json and csv files, need to create one utility file
  - dataProvider.ts  - Read JSON and CSV data
  - randomDataGenerator.ts  - Generate dummy data using faker

* dataProvider.ts:
  -  import fs and csv-parse/syn separately
  -  create two static functions in class
  -  one function will return data from JSON
  -  another function will return data from CSV
  -  that data wil use in test case

* randomDataGenerator.ts:
  - faker library data
  - random data generate; i.e., name, address, phone, alphanumeric, numeric..
```

Step: 8 -> Test Cases (under test cases folder)

```
  1. import
  2. Declare shared variables
  3. Setup beforeEach hook
  4. Initialize page objects within beforeEach hook section
  5. Setup afterEach hook
  6. Start test case steps (Scenario)

* import: Example:

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

```
  import { HomePage } from "../pages/HomePage";
  import { MyAccountPage } from "../pages/MyAccountPage";
  import { LoginPage } from "../pages/LoginPage";
  import { LogoutPage } from "../pages/LogoutPage";

  import { TestConfig } from "../test.config"; //to open URL


* Declare shared variables: Example:

  let config : TestConfig;
  let homePage : HomePage;
  let myAccountPage : MyAccountPage;
  let loginPage : LoginPage;
  let logoutPage : LogoutPage;


* Setup beforeEach hook: Example:

    test.beforeEach( async ({page}) => {

    config = new TestConfig(); //Load test config
    await page.goto(config.appUrl); //Step: 1  Open application URL

*  Initialize page objects inside beforeEach hook: Example:

    homePage = new HomePage(page);
    myAccountPage = new MyAccountPage(page);
    loginPage = new LoginPage(page);
    logoutPage = new LogoutPage(page);

  })

* Setup afterEach hook: Example:

    test.afterEach(async ({page}) => {
      await page.close(); //helps to keep tests clean

  })


* Start test case steps (Scenarios)


Tips:
* import class Names from file names given while creating
* hooks are required for more than one test block in same file to run before and
after every test
* for password and confirm password, need to set in a variable, so that it will
not generate different password for confirm password
* for check box no parameter needed, await registrationPage.policyCheckbox();
```

```
* for continue button, parameter needed, await registrationPage.continueButton();
* for confirmation message, expect(confirmationMsg).toContain('Your Account Has
Been Created!')
* after completing all the test, need to integrate everything in package.json file
and then start doing execution, so all the commands will need to configure in
package.json file
* add await where needed
```

Hooks:

```
  - beforeEach()
  - afterEach() //this is not mandatory, can just write await page.close()


  * beforeEach()
    - keep all common steps within this hook in each test file which has more than
one test blocks; i.e,

      let homePage = HomePage; //global variable
      let registrationPage = RegistrationPage; //global variable

      test.beforeEach(async({page}) => {

        const config = new TestConfig ();
        await page.goto(config.appUrl); //navigate to url
        homePage = HomePage;
        registrationPage = RegistrationPage;

      });


  * afterEach () - write it just after beforeEach hook
        i.e;

        test.afterEach (async ({page}) => {

          await page.close();

        })
```

Commands to generate and view allure reports //need to configure in playwright.config.ts first

```
  allure generate ./allure-results -o ./allure-report --clean

  allure open ./allure-report
```

```
  **playwright.config.ts**
  reporter: [
    ['html', { outputFolder: '../reports/html-report' }],
    ['allure-playwright', { outputFolder: '..reports/allure-results'}],
    ['dot'],
    ['list']
  ],
```

Run tests from Package.json:

```
"scripts": {
      "test:end-to-end": "playwright test --grep @end-to-end --headed",
      "test:master" : "playwright test --grep @master",
      "test:master:headed": "playwright test --grep @master --headed",
      "test:sanity" : "playwright test --grep @sanity",
      "test:regression" : "playwright test --grep @regression",
      "test:datadriven" : "playwright test --grep @datadriven",
      "test:sanity:debug" : "playwright test --grep @sanity --debug"

  },

  - Need to use keyword only to run particular test

  * Run Test:
    npm run test:master:headed  //this will run all the test in headed mode
```

Data-Driven Test

```
  Data-driven testing means we have to repeat the same login (for example) with
the multiple sets of data; we have to test the login with multiple sets of data

1. In test data folder create
  - logindata.json
    - In array two different blocks(Valid and Invalid scenarios) : test name,
email, password, expected
    - Make sure same thing should not repeat again.
    - Keep the test name different every time, as based on the test name the test
will repeat multiple times; otherwise, it will consider as duplicate
    - Can add multiple combinations of tests as required

  - logindata.csv
    - In three different lines (Valid and Invalid scenarios): test name, email,
password, expected.
    - Make sure same thing should not repeat again.
    - Can add multiple combinations of tests as required
```

```
2. To read login.json and login.csv data,
  In util folder create
  - dataProvider.ts  --> one class with two static functions ()


3. Crete data-driven test cases in tests
  * always import with class name

  - import pages; HomePage, MyAccountPage, LoginPage
  - import TestConfig to get appUrl
  - import DataProvider to read json and csv data

  - Load the data from file
      - Load one by one
      - Load Json data first -> logindata.json
      - Need to provide the path of the file
        const jsonPath = "testdata/logindata.json"; //this is called json path

      - To load this data into a variable, we need to call the function which
created in utils -> dataProvider.ts -> DataProvider (class name) ->
getTestDataFromJson (filePath:string) by passing the file path

          const jsonTestData = DataProvider.getTestDataFromJson(jsonPath)

      - then start writing looping statements because same login test need to
repeat multiple times and json data returns array

        for (const data of jsonTestData) {

          test(`  `) //start writing test here with the backtick

        }

      so from jsonTestData we are reading each data one by one and that data we
need to use in test
```

End-to-End Testing

```
  - It is complete User flow
  - No need to add hooks in the actual end to end test
  - Need to hardcode password and at the end need to write return email to log in
again after registration and logged out
  - Need to create separate functions for all steps of test case and call that
functions in one test block

  - can get emojis by pressing windows key + . or  from chatGPT paste code and ask
to generate emojis
```

```
©Kash Chauhan
```