1. Project Structure (descriptive)

The project follows a modular structure aligned with best practices for automated testing. Key components include:

- `playwright.config.ts` – main configuration file for the Playwright test runner
- `.env` – environment variables (base url, usernames, passwords)
- `global-setup.ts` – setup logic executed before tests start
- `src/pages/` – Page Object classes (e.g., LoginPage, ProductsPage) encapsulating selectors and actions
- `src/tests/` – end-to-end tests grouped by functionality (e.g., login.spec.ts, products.cart.spec.ts)
- `src/utils/` – utility functions, such as a data generator using `faker`
- `src/types/` – enums and type definitions used across the test suite
- `playwright-report/` – html reports generated after test execution
- `test-results/` – traces, screenshots, and videos (on failure)
- `src/fixtures/` – Playwright fixtures exposing page objects like loginPage, productsPage
- `.github/workflows/` – GitHub Actions CI pipeline to run Playwright tests on push and pull requests

2. Test Architecture

Tests are executed using the Playwright Test Runner, which provides:

- `test.beforeEach()` to handle setup like logging in
- `test.describe()` for grouping related tests
- `expect.soft()` to allow test steps to continue after soft assertion failures

Example test:

```
test("should clicking 'Logout' returns to login page - error
user", async ({ page, productsPage }) => {
  await productsPage.openMenu();
  await
expect.soft(productsPage.menuLogoutLink).toBeVisible();
  await productsPage.clickLogout();

  await
expect.soft(page).toHaveURL(process.env.SAUCE_DEMO_BASEURL ??
“");});});
```

3. Page Object Model (POM)

Each app screen is represented by a class containing methods and locators.

Example:

```
export class ProductsPage {
  public readonly cartBadge =
this.page.locator('.shopping_cart_badge');

  public async clickResetApp(): Promise<void> {
    await this.menuResetAppLink.waitFor({ state:
"visible" });
    await this.menuResetAppLink.scrollIntoViewIfNeeded();
    await this.menuResetAppLink.click();
  }
}
```

Constructors are not used in the Page Object classes. The page object is inherited from a shared BasePage, simplifying object creation and keeping classes clean.

4. Types and Enums (`src/types/`)

The framework uses TypeScript along with enums to represent data such as user types:

```
export enum UserType {
  Standard = "standard_user",
  Problem = "problem_user",
  Error = "error_user",
```

Benefits of using enums:

- Avoid hardcoded strings and typos
- Improved readability
- Better developer experience through autocompletion

5. Utilities and Faker (`src/utils/`)

The project includes utility helpers like `testData.ts`, which uses the faker library to create randomized test data.

Example usage:

```
import { faker } from '@faker-js/faker';

export const generateCheckoutData = (): CheckoutFormData => ({
  firstName: faker.person.firstName(),
  lastName: faker.person.lastName(),
  postalCode: faker.location.zipCode(),
});
```

- Provides realistic and varied input
- Avoids repetitive static values
- Ensures unique data for fields like email or username

6. Reporting

HTML reports are generated automatically after tests are run.

- Configuration is defined in `playwright.config.ts`
- Reports include test results, traces, screenshots, and videos (on failure)
- You can view the report using:

```
npx playwright show-report or npm run report
```

7. NPM Scripts

The `package.json` includes handy scripts for running and debugging tests:

```
"scripts": {
  "test": "npx playwright test",
  "test-debug": "npx playwright test --debug",
  "report": "npx playwright show-report"
}
```

8. Environment Configuration

Environment variables like base url and user credentials are stored in the `.env` file. This allows flexible switching between environments without modifying test code.

Example:

```
SAUCE_DEMO_BASEURL=https://www.saucedemo.com
SAUCE_DEMO_STANDARD_USER=standard_user
SAUCE_DEMO_PASSWORD=secret_sauce
```

9. Summary

The avenge playwright test project is a scalable and maintainable foundation for end-to-end web testing. It features:

- Playwright Test Runner with built-in reporting
- Modular design using the Page Object Model
- TypeScript with enums and type safety
- Dynamic test data via faker
- Full trace and video support on failures