POM

1) Centralized locators

It’s better to use more centralized and easy to update approach.
Instead of

```
get signInLink() { return cy.get('.login') }
get emailAddressTxt() { return cy.get('#email') }
```

Use:

```
private readonly selectors = {
  signInLink: '.login',
  emailAddressTxt: '#email',
}
```

2) Export instance into tests from POM

Creating a new instance in POM file during export isn't the best practice. Cypress recommends **creating a new instance per test** to avoid unexpected behavior.

Instead of:

```
export const loginPage: LoginPage = new LoginPage()
```

Export like this:

```
export class LoginPage {}
```

TESTS

1) Singleton imports

Singleton imports create a **shared instance** across tests.

```
import { loginPage } from '../pages/loginPage'
loginPage.launchApplication()
```

Better instantiate the page class inside the test instead:

```
beforeEach(() => { loginPage = new LoginPage();}
```

2) Handling of Fixtures

To handling fixtures in beforeEach() is incorrect. Cypress runs test functions asynchronously, so this.data may be undefined in some tests.

```
beforeEach(() => {
  loginPage.launchApplication()
  cy.fixture('users.json').then(function (data) { this.data = data; })
```

Use cy.fixture() with before() or beforeEach() and Store in a let Variable

```
describe('Login Functionality', () => {
  let testData;
  before(() => {
      cy.fixture('users.json').then((data) => {
          testData = data;
      });
  });
```

This eliminates flakiness caused by Cypress's asynchronous behavior and testData is available before all tests run.

3) Hardcoded Data

Hardcoded credentials make tests less reusable and harder to maintain.

```
loginPage.login("testautomation@cypresstest.com", "Test@1234");
```

Use:

```
loginPage.login(testData.valid_credentials.emailId,
testData.valid_credentials.password);
```

YAML FILE

1) Matrix

The **strategy.matrix** block is used for **scalability and flexibility**, even though it only includes **one Node.js version (`14.17.0`). Consistency with Best Practices.** Many CI/CD workflows use a matrix even when testing **only one configuration** for consistency.

```
strategy:
            matrix:
                node-version: [14.17.0, 14.15.0]
                https://nodejs.org/en/about/releases/
        steps:
            - uses: actions/checkout@v2
            - name: Use Node.js ${{ matrix.node-version }}
              uses: actions/setup-node@v2
                  with:
                       node-version: ${{ matrix.node-version }}
                       cache: "npm"
```

2) GitHub Artifacts step

Accessibility for Debugging

Artifacts give us a possibility to preserve the Report as a Backup, for example.

The artifact ensures the allure-report is stored even if subsequent steps (e.g., deployment to GitHub Pages) fail. You can later download the artifact manually from the workflow run in GitHub Actions if needed.

If a developer needs to view the report without deploying it or if deployment fails, they can download the artifact from the "Artifacts" section in the workflow summary.

PACKAGE JSON FILE

1) Dependencies VS devDependencies

- If running tests in a CI/CD pipeline, Cypress should be in "dependencies".

- Some Docker containers fail to install Cypress properly if it's only in devDependencies.

```
"dependencies": {
   "@faker-js/faker": "^9.3.0",
   "cypress": "^13.17.0"
 }
```

2) Added keywords, repository, bugs:

Keywords

Purpose: The keywords field is an array of strings that describe the project. These keywords help others discover your package when they search for related topics on package registries like npm (Node Package Manager).

Repository

Purpose: The repository field specifies where the source code for the project is hosted. It typically includes the type of version control system (e.g., git) and the URL of the repository.

This is useful for collaboration, bug reporting, and contributing to the project.

```
"repository": {
  "type": "git",
  "url": "git+https://github.com/AZANIR/cypressAllure.git"
}
```

Bugs

Purpose: The bugs field provides a URL where users can report issues or bugs they encounter while using the project. This is typically a link to the project's issue tracker on a platform like GitHub.

PLUGINS FOLDER

The **plugins/index.js** file is mainly used to extend Cypress functionality using Node.js in of Cypress less than 10

You do NOT need plugins/index.js in Cypress 10+ because all plugin-related configurations have been moved to cypress.config.js.

SUPPORT FOLDER STRUCTURE

Where Should You Write Custom Commands in Cypress?

- In Cypress 10+ and not big projects write custom commands inside **`cypress/support/commands.js`**
- For better organization, you can also split them into different files like **`cypress/support/commands/e2e.js`**

TSCONFIG.json

When such file should be used?

- Enables TypeScript compilation for Cypress
- Improves code quality with type safety
- Supports custom Cypress commands (commands.ts)
- Ensures compatibility with modern JavaScript (ESNext)

File in project is obsolete for 2025 year, it needs modern changes as:

"target": "ESNext"

"lib": ["dom", "ESNext"]

"types": ["node", "cypress", "cypress-real-events"]

```
{
    "compilerOptions": {
      "target": "ESNext",
      "module": "ESNext",
      "lib": ["dom", "ESNext"],
      "types": ["node", "cypress"],
      "strict": true,
      "skipLibCheck": true,
      "baseUrl": "./"
    },
```

If you're only using JavaScript (**.js** files) for Cypress, this file is NOT required.