cypressAllure tests report

This review provides an examination of the cypressAllure tests, which are available by this link https://github.com/AZANIR/cypressAllure, highlighting areas of strength as well as potential improvements."

Unfortunately, the tested site is not available and is offline. So there is no possibility of making a proper Evaluation of the provided code because all tests fail to pass. So the review is made on the structure of the code, its organization, and cleanliness.

I want to start my report with a README file review.

README file review

1. **Title and Description:**
    - The title "cypress-e2e-allure# Cypress-e2e-Allure Automation-Framework with allure reports" is clear and informative, indicating that it's an automation framework using Cypress and Allure for reporting.
    - The description "This Framework contains sample code for # Cypress-e2e-Allure Automation-Framework" provides a brief overview of the framework.
2. **Features:**
    - It mentions key features of the framework, such as Page Object Model, reading test data from JSON files, login scenarios with valid and invalid credentials, and Allure report integration. This gives users an idea of what functionalities are included.
3. **Directory Structure:**
    - It mentions the structure of the project, which is helpful for users to understand where to find specific files or functionalities within the repository.
4. **Steps to Run:**
    - Provides clear steps for running the framework, including cloning the repository, changing username and password (though it would be helpful to specify where these changes need to be made), installing dependencies, and running the tests.
5. **Report View:**
    - Mentions that Allure reports are generated but doesn't provide detailed instructions on how to view them. It would be helpful to include steps on how users can access and interpret the reports.

Suggestions for Improvement:

- Detail each step of installing the project with precise commands and expected outcomes.
- Provide more detailed instructions for changing the username and password. Specify which file or files need to be modified for this purpose.

- Add a section or subsection for "How to View Allure Reports" with step-by-step instructions on accessing and interpreting the reports.
- Consider adding a section for troubleshooting common issues or FAQs, if applicable.

Overall, the README provides a good overview of the framework and how to use it, but could benefit from more detailed instructions in certain areas.

loginPage.ts Review

cypressAllure > cypress > e2e > pages > TS loginPage.ts > LoginPage > login

AZANIR, 21 months ago | 1 author (AZANIR)

```ts
/// <reference types="cypress" />

class LoginPage {
    get signinLink() { return cy.get('.login') }
    get emailAddressTxt() { return cy.get('#email') }
    get passwordTxt() { return cy.get('#passwd') }
    get signinBtn() { return cy.get('#SubmitLogin') }
    get alertBox() { return cy.get('p:contains("error")')}
    get alertMessage() { return cy.get('.alert-danger > ol > li') }

    public launchApplication() {
        cy.visit('/')
    }

    public login(emailId: string, password: string) {
        this.signinLink.click()
        this.emailAddressTxt.type(emailId)
        this.passwordTxt.type(password)
        this.signinBtn.click()
    }

    public validateLoginError(errorMessage: string) {
        this.alertBox.should('be.visible')
        this.alertMessage.should('have.text', errorMessage)

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

1. *Documentation and Comments*

- **Issue**: There are no comments or documentation in the code.
- **Recommendation**: Add comments to describe the purpose and functionality of the class and its methods. Use JSDoc to provide detailed information about the functions, parameters, and return types.

2. *Selector Usage*

- **Issue**: The method for selecting the alert box (`p:contains("error")`) may not be reliable or specific enough, which might lead to flaky tests if the DOM structure changes or if other elements also contain the text "error".
- **Recommendation**: Use a more specific selector that relies on an ID, class, or data attribute specific to the error message.

3. *Function Naming and Consistency*

- **Issue**: Function names like `launchApplication`, `login`, and `validateLoginError` are clear, but the getters have names that end with 'Txt' and 'Btn', which could be more consistent.
- **Recommendation**: Consider renaming `emailAddressTxt` to `emailAddressInput` and `passwordTxt` to `passwordInput` to more accurately describe the elements.

4. *Method Modifiers*

- **Issue**: The use of `public` keyword is redundant since all class methods in TypeScript are public by default unless specified otherwise.
- **Recommendation**: Remove the `public` keyword unless you are explicitly trying to differentiate from `private` or `protected` methods for clarity.

5. *Error Handling*

- **Issue**: There's no error handling if elements are not found or if the actions on the elements fail (e.g., if the login button is disabled).
- **Recommendation**: Add error handling or assertions to ensure elements are in a valid state before interaction (e.g., check if inputs are enabled and the button is clickable).

6. *TypeScript Usage*

- **Issue**: Proper use of TypeScript features such as typing the parameters and return types can enhance code robustness.
- **Recommendation**: Ensure that the method parameters (`emailId`, `password`, `errorMessage`) are typed correctly. Additionally, specify the return types of functions if not implicitly understood.

7. *Code Structure*

- **Issue**: Direct usage of the page object `loginPage` as an exported constant might limit flexibility in tests where you need a fresh page object or different instances with varying states.
- **Recommendation**: Consider exporting the class itself and creating instances in the test files. This approach supports more dynamic test scenarios and better isolation between tests.

8. *Test Readiness*

- **Issue**: There is an assumption that the root URL (`'/'`) directly leads to a page with the login elements.
- **Recommendation**: Confirm that visiting `'/'` is appropriate for all test cases where this page object is used. If different environments or preconditions are needed, the `launchApplication` method might require parameters or different URLs.

Example of Revised class

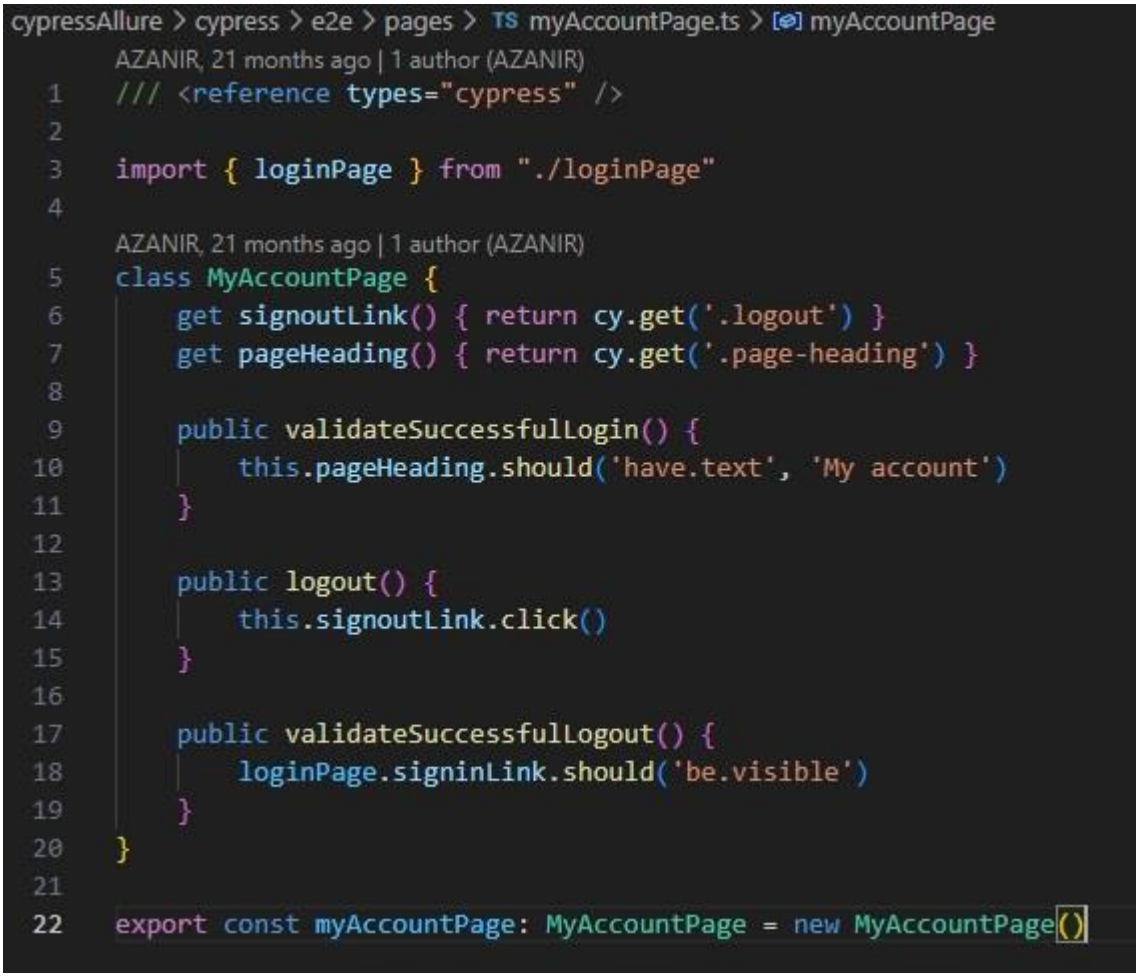
cypressAllure > cypress > e2e > pages > TS loginPage.ts > default

You, 1 second ago | 2 authors (You and others)

```typescript
class LoginPage {
    private get signinLink() { return cy.get('.login') }
    private get emailAddressInput() { return cy.get('#email') }
    private get passwordInput() { return cy.get('#passwd') }
    private get signinButton() { return cy.get('#SubmitLogin') }
    private get alertBox() { return cy.get('.alert-error') } // Adjusted for better specificity
    private get alertMessage() { return cy.get('.alert-danger > ol > li') }

    visitHomePage() {
        cy.visit('/')
    }

    login(email: string, password: string) {
        this.signinLink.click()
        this.emailAddressInput.type(email)
        this.passwordInput.type(password)
        this.signinButton.click()
    }

    verifyLoginError(expectedMessage: string) {
        this.alertBox.should('be.visible')
        this.alertMessage.should('have.text', expectedMessage)
    }
}

export default LoginPage;
```

You, 1 second ago • Uncommitted changes

myAccountPage.ts Review



1. *Modular Design*

- **Issue**: Direct import and usage of `loginPage` inside `MyAccountPage` can lead to tight coupling between page objects, which is not recommended in page object model design.
- **Recommendation**: Instead of directly referencing another page object within a method, consider passing necessary elements or statuses as parameters to the method, or handle the interaction at the test script level where different page objects interact.

2. *Selector Usage*

- **Issue**: Like the previous class, reliance on generic selectors (`.logout` and `.page-heading`) could become problematic if the page structure changes.
- **Recommendation**: Use more specific selectors, possibly including data attributes that explicitly denote them as test targets (e.g., `data-cy="logout"`).

3. *Method Access Modifiers*

- **Issue**: The use of `public` is again redundant, as it is the default access level for class methods in TypeScript.
- **Recommendation**: Remove the `public` keyword unless specifying it for clarity or contrasting with other access levels.

4. *Function Naming Consistency*

- **Issue**: The function names are descriptive but could be enhanced to match an action-oriented style consistently used across your test suite.
- **Recommendation**: Rename methods to include action verbs that clearly indicate what each method does, such as `verifySuccessfulLogin` and `performLogout`.

5. *Error Handling and Assertions*

- **Issue**: The methods assume successful interaction with elements without checking if elements are interactable (visible, not disabled).
- **Recommendation**: Add checks to ensure elements are ready for interaction, such as verifying visibility or enabled status before clicking or asserting.

6. *Code Structure and Export*

- **Issue**: Exporting an instance of the `MyAccountPage` as a constant (`myAccountPage`) may limit the flexibility needed in tests, particularly in scenarios requiring different instances or states.
- **Recommendation**: Export the class itself and allow test scripts to create instances as needed. This approach helps in managing different test contexts and ensures better test isolation.

Example of revised class

cypressAllure > cypress > e2e > pages > TS myAccountPage.ts > ...

```
You, 1 second ago | 2 authors (AZANIR and others)
/// <reference types="cypress" />

You, 1 second ago | 2 authors (You and others)
class MyAccountPage {
    private get signoutLink() { return cy.get('.logout') }
    private get pageHeading() { return cy.get('.page-heading') }

    visitAccountPage() {
        cy.visit('/my-account') // Adjust as necessary to ensure the right routing
    }

    verifySuccessfulLogin() {
        this.pageHeading.should('have.text', 'My account')
    }

    performLogout() {
        this.signoutLink.click()
    }

    verifySuccessfulLogout() {
        cy.get('.login').should('be.visible') // Consider passing a callback or using events to verify logout
    }
}

export default MyAccountPage;
```

Login.test.ts Review

```
import { loginPage } from '../pages/loginPage'      AZANIR, 21 months ago • update test and add allure
import { myAccountPage } from '../pages/myAccountPage'

describe('Login Functionality', () => {
    beforeEach(() => {
        loginPage.launchApplication()
        cy.fixture('users.json').then(function (data) {
            this.data = data;
        })
    })
    it('login with valid credentials', function () {
        loginPage.login("testautomation@cypresstest.com", "Test@1234")
        myAccountPage.validateSuccessfulLogin()
        myAccountPage.logout()
        myAccountPage.validateSuccessfulLogout()
    })
    it('login with valid credentials read data from fixture', function () {
        loginPage.login(this.data.valid_credentials.emailId, this.data.valid_credentials.password)
        myAccountPage.validateSuccessfulLogin()
        myAccountPage.logout()
        myAccountPage.validateSuccessfulLogout()
    })
    it('login with invalid email credentials read data from fixture', function () {
        loginPage.login(this.data.invalid_credentials.invalid_email.emailId,
            this.data.invalid_credentials.invalid_email.password)
        loginPage.validateLoginError('Authentication failed.')
    })
    it('login with invalid password credentials read data from fixture', function () {
        loginPage.login(this.data.invalid_credentials.invalid_password.emailId,
            this.data.invalid_credentials.invalid_password.password)
        loginPage.validateLoginError('Authentication failed.')
    })
    it('login with wrong email format credentials read data from fixture', function () {
        loginPage.login(this.data.invalid_credentials.wrong_email_format.emailId, this.data.invalid_credentials.wrong_email_format.password)
        loginPage.validateLoginError('Invalid email addresssssss.')
    })
})
```

1. *Asynchronous Handling of Fixtures*

- **Issue**: Fixtures are loaded asynchronously but used synchronously in tests, which can lead to undefined values if not handled correctly.
- **Recommendation**: Ensure that the fixture data is fully loaded before tests run. Using async/await with fixtures can sometimes help, though it's not directly supported by Cypress commands without additional plugins or workarounds.

2. *Error Message Validation*

- **Issue**: The error message validation in `validateLoginError` relies on exact text match, which can be problematic if additional messages or multilingual support are introduced.
- **Recommendation**: Use regular expressions or partial matches for error validation to make tests less brittle. For example, checking for substrings like "Authentication failed" or "Invalid email address".

3. *Misleading Test Names and Assertions*

- **Issue**: The error message in the last test case expects 'Invalid email addresssssss.' which seems to be a typo or incorrect expectation.
- **Recommendation**: Review and correct the expected messages to reflect actual application responses. Avoid typos and ensure that assertions match expected outputs precisely.

4. *Dependency Management*

- **Issue**: Direct import of page objects which could be instantiated multiple times across tests, potentially leading to state bleed-through.
- **Recommendation**: Consider creating new instances of page objects within each test or test suite to avoid unintended state persistence across tests.

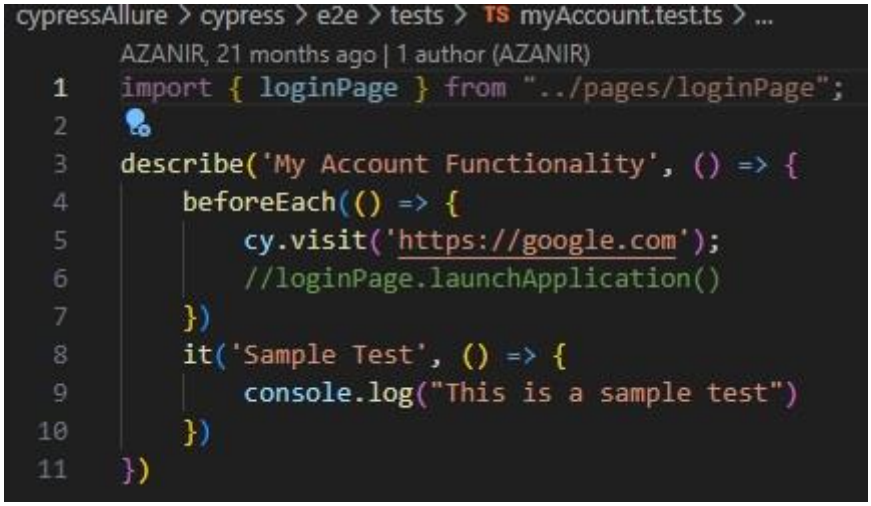
5. *Code Duplication*

- **Issue**: There is a repetitive pattern in calling login and validate functions.
- **Recommendation**: Consider abstracting common sequences into helper functions or using custom Cypress commands to reduce redundancy and improve test clarity.

Revised Example with Some Improvements

This revision uses a global setup to load fixtures once before all tests, assigning them to a local variable for easier and more reliable access. This avoids potential issues with the `this` context and ensures data is loaded and available when tests execute.

myAccount.test.ts Review



1. *Base URL Configuration*

- **Issue**: The test explicitly navigates to "https://google.com" within the `beforeEach` function. This might not align with testing your application if "https://google.com" is just a placeholder.
- **Recommendation**: Configure a base URL in the Cypress environment settings (`cypress.json`). This way, you can use `cy.visit('/')` to automatically append the base URL, which simplifies changing the URL across all tests when needed.

2. *Commented Code*

- **Issue**: The `loginPage.launchApplication()` is commented out, which suggests it was initially intended to be part of the setup but was replaced or deactivated.
- **Recommendation**: If the intention is to navigate to your application's login page, ensure the correct URL is set as the base URL and use the `loginPage.launchApplication()` as intended. Remove or update commented code to keep the test suite clean and understandable.

3. *Logging*

- **Issue**: There's a `console.log("This is a sample test")` statement within the test. While this might be useful for debugging during development, it does not contribute to the actual testing process.
- **Recommendation**: Remove or replace debug logging with actual assertions or test steps. If logging is necessary for debugging, consider more detailed messages that describe the test state or values being tested.

4. *Test Purpose and Assertions*

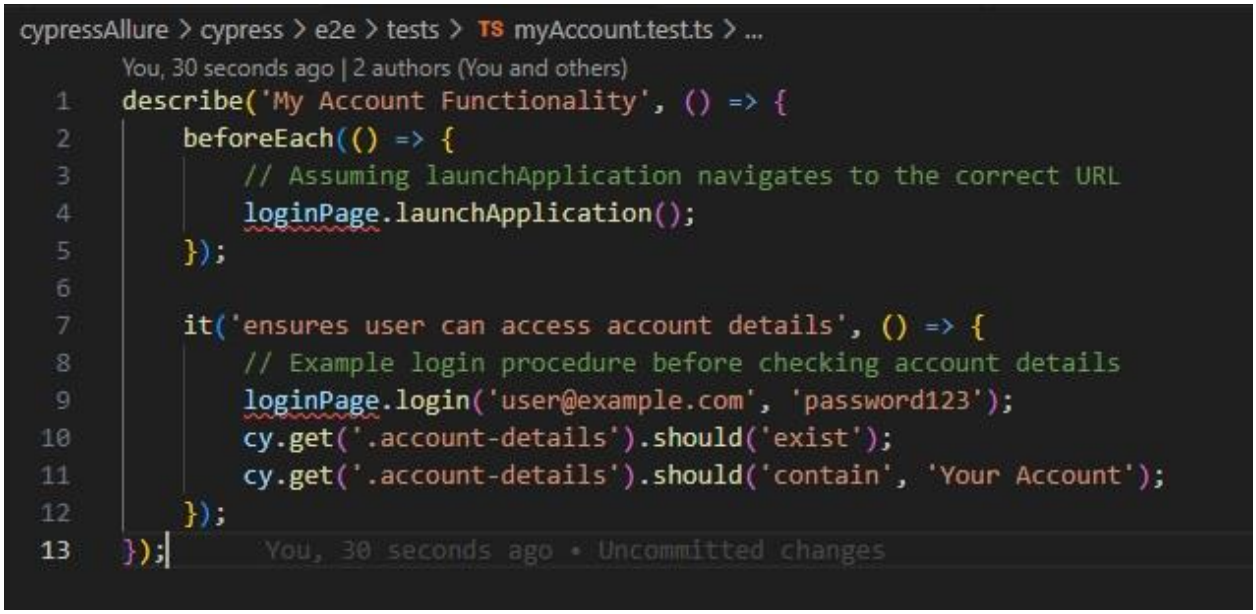
- **Issue**: The test named 'Sample Test' does not perform any assertions or interactions with the web page.
- **Recommendation**: Add meaningful assertions or interactions to ensure the test checks for specific conditions or behaviors in your application. Use Cypress commands like `cy.find()`, `cy.get()`, and `cy.should()` to interact with elements and verify states.

5. *Use of `describe` and `it`*

- **Good Practice**: The use of `describe` and `it` blocks is structured correctly, where `describe` defines the test suite, and `it` defines individual tests. This is a best practice in Cypress and other testing frameworks.

- **Suggestion**: Continue using this structure to organize tests logically. As your testing needs grow, consider grouping related tests under appropriate `describe` blocks and using meaningful descriptions for `it` blocks to clearly state what each test is supposed to verify.

Example of Revised class



config.config.ts Review

1. *Timeout Settings*

- You've set a consistent `defaultCommandTimeout`, `execTimeout`, `taskTimeout`, `requestTimeout`, and `responseTimeout`. These settings are crucial for managing how long Cypress will wait for commands and operations before timing out.
- **Consideration**: Ensure these values align with the expected behavior and responsiveness of your application. If your application has pages or actions known to take longer, you might need to adjust these timeouts specifically for those scenarios.

2. *Page Load and Response Timeouts*

- `pageLoadTimeout` and `responseTimeout` are set to higher values, which is suitable for ensuring that pages with heavy content and slower response times do not cause premature test failures.
- **Consideration**: While higher timeouts can prevent some flaky tests due to network and server delays, be cautious that it could also make the tests run longer than necessary if used indiscriminately.

3. *Video and Screenshots*

- Recording videos of test runs and capturing screenshots are enabled. These are invaluable for debugging and understanding the test flow, especially when tests fail.

- **Suggestion**: Make sure that your CI/CD environment has enough storage to handle the videos and screenshots, especially if you have numerous tests or long-running tests.

4. *Folders for Reports*

- Custom paths for `screenshotsFolder` and `videosFolder` are specified under `reports/screenshots` and `reports/videos`. This helps in organizing test artifacts.
- **Suggestion**: Periodically clean up these directories or implement a retention policy to manage disk space, especially in continuous integration environments.

5. *Environment Variables*

- `env` contains an `allureResultsPath`, which integrates with Allure reports, a popular test reporting tool.
- **Suggestion**: Ensure the relative path specified (`'../allure-results'`) correctly points to where you want the Allure results to be stored, relative to where Cypress runs its tests.

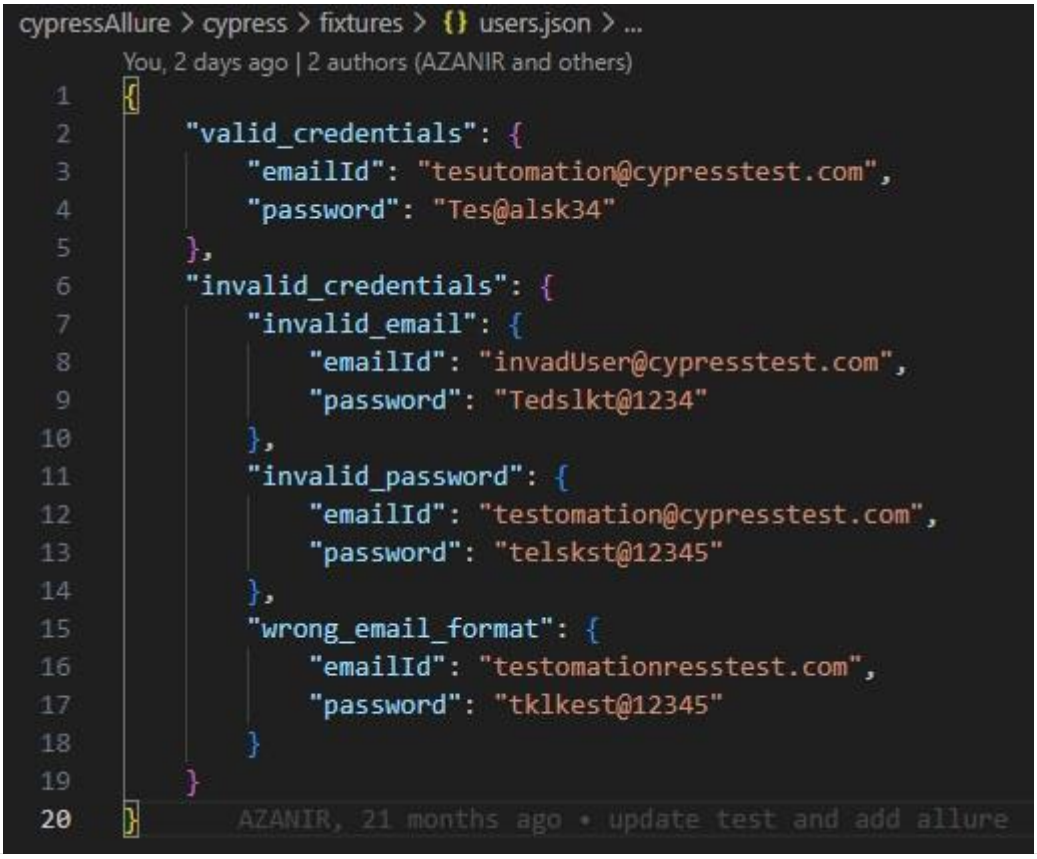
6. *End-to-End Test Specific Settings*

- `baseUrl` is set to `http://automationpractice.com/index.php`. This centralizes the base URL for all tests, reducing redundancy in test scripts.
- `specPattern` defines where to find the test files, helping Cypress to locate them quickly and accurately.
- **Consideration**: Regularly update the `baseUrl` if the testing environment changes (e.g., from staging to production, if applicable).

7. *Plugin Integration*

- `setupNodeEvents` includes a callback that imports an existing plugin configuration. This is essential for integrating additional Cypress plugins or handling complex setups.
- **Recommendation**: Review and update `cypress/plugins/index.js` as needed to ensure all plugins are correctly configured and up-to-date.

User.json Review



1. **Structure and Readability**:
    - Your JSON structure is well-organized into categories (`valid_credentials`, `invalid_credentials`), making it easy to understand and use in tests. Each category is further subdivided based on the type of credential error (invalid email, invalid password, wrong email format), which is excellent for targeted test scenarios.
2. **Content Accuracy and Realism**:
    - **Typos and Errors**: Ensure that all data, especially intended errors, are deliberate. For example, "tesutomation" in the email might be a typo. Decide if this is intended to be a valid or intentionally incorrect email.
    - **Email Format**: The `wrong_email_format` does well to demonstrate a common input error by omitting the '@' symbol. This is a good practice for testing form validation.
3. **Security Practices**:
    - Since this is test data, it's crucial to ensure that none of the credentials are real or could be mistaken for real user data. Always use clearly fictional data to prevent any security risks.
4. **Extensibility**:
    - Your file is structured in a way that makes it easy to add more test cases. For instance, you could add more scenarios under `invalid_credentials` for testing other common input errors like passwords that are too short, emails without a domain, etc.

Suggestions for Improvement

1. **Expand Error Scenarios**:
    - Consider adding more diverse examples of incorrect data to thoroughly test the application's resilience and error handling. For example:
        - Passwords that fail to meet complexity requirements (no special characters, all lowercase, too short, etc.).
        - Emails that are missing the domain part or have invalid characters.
2. **Improve Documentation**:
    - Add comments within the JSON file if your framework supports it (note that JSON itself does not support comments, but some preprocessors do), or document the fixture structure and intended use cases somewhere in your project's documentation. This helps new team members understand the purpose of each section quickly.
3. **Ensure Consistent Formatting**:
    - Review all entries for consistent formatting, especially around email addresses and passwords, to avoid accidental misclassification of valid and invalid credentials.
4. **Use Environment-Specific Data**:
    - If tests are run across different environments (development, staging, production-like), consider having different sets of data or add environment-specific modifications using environment variables.