Implementing E2E tests with Cypress and Page Object Model: evolution of approaches

Inessa V. Krasnokutska, Oleksandr S. Krasnokutskyi

Yuriy Fedkovych Chernivtsi National University, 2 Kotsiubynskoho Str., Chernivtsi, 58002, Ukraine

Abstract

This article shows eight approaches how to construct Cypress tests using POM. The connections between them are stressed as their evolution while writing code developing E2E tests. The authors highlight advantages and disadvantages of the approaches and offer the solution of problems. This article can be used both as a combined overview of different approaches and as a manual for those who are struggling to write tests with Cypress in a better way.

Keywords

Cypress JS, Page Object Model, POM design pattern, automation testing, E2E tests

1. Introduction

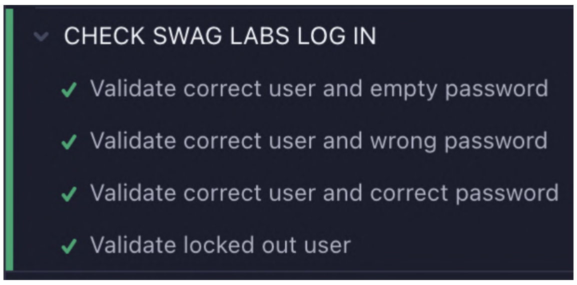
Suppose we have a problem to cover the functionality of certain website with automation tests. For instance we can consider the website https://www.saucedemo.com/v1/index.html without loss of generality. The website is devoted to illustrating of different test cases that occur during unsuccessful login process. It contains the form with username and password fields to get the access to the next page, and in case of unsuccessful log in, the error message with corresponding text is shown. This website is convenient to be used as a model example for all typical flows of users behaviour that could be covered by tests with a minimum amount of effort.

Successful login without any doubt as a positive test case means inputting correct username and password. Negative tests occur when a user is left on the same page and an appropriate error message is shown. It can happen when a user enters an empty or wrong username or password. Different tests can be considered to cover those cases. However, taking into account that most of developed tests are quite similar and optimising the cases authors demonstrate only valuable and significant of them in this article. More test cases can be found in the project repository at https://github.com/InaKrasnokutska/CypressPOM. To run the solution you just need to pull and run **npm i** to setup dependencies, and later **npm run cy:open** to see how it works.

Cypress is chosen to demonstrate how to create tests in different manners. All pros and cons of each approach are analysed and highlighted. There is a lack of good resources on the Internet

CS&SE@SW 2023: 6th Workshop for Young Scientists in Computer Science & Software Engineering, February 2, 2024, Kryvyi Rih, Ukraine

i.krasnokutska@chnu.edu.ua (I. V. Krasnokutska); okrasn.mail@gmail.com (O. S. Krasnokutskyi)
https://amit.chnu.edu.ua/pro-kafedru/personalii/krasnokutska-inessa-volodymyrivna (I. V. Krasnokutska)
0000-0002-7034-7291 (I. V. Krasnokutska)


CHECK SWAG LABS LOG IN

✓ Validate correct user and empty password

✓ Validate correct user and wrong password

✓ Validate correct user and correct password

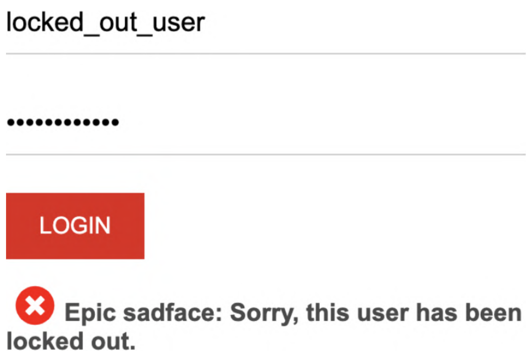
✓ Validate locked out user

Figure 1: Test cases you can find at https://github.com/InaKrasnokutska/CypressPOM and run in Cypress.

about usage of the Page Object design pattern in Cypress Cypress [1], and each resource shows only one point of view and it is hard to understand the correlation between them.

2. Main results

All possible variations of implementing Page Object Model (POM) are constructed in this article. It happens when a locked out user enters his/her username and password and receives an appropriate error message.


locked_out_user

••••••••••••

LOGIN

Epic sadface: Sorry, this user has been locked out.

Figure 2: Error message the locked out user tries to log in.

Other test cases can be found at https://github.com/InaKrasnokutska/CypressPOM. In the next subsections we will discuss 9 approaches to organise selected test case. For convenience and quick search of files with tests (specs) in the repository they have names according to subsection numbers in this article where they are discussed.

2.1. Tests without POM

Before diving into the Page Object design pattern, let us create a Cypress test without it.

According to this decision the code appears very strict forward because in the spec file we need to prescribe the interaction with each DOM element of the page. This interaction is done

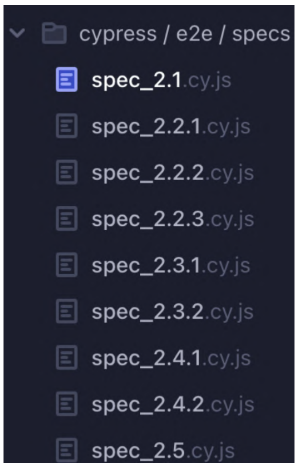

cypress / e2e / specs
spec_2.1.cy.js
spec_2.2.1.cy.js
spec_2.2.2.cy.js
spec_2.2.3.cy.js
spec_2.3.1.cy.js
spec_2.3.2.cy.js
spec_2.4.1.cy.js
spec_2.4.2.cy.js
spec_2.5.cy.js

Figure 3: Spec's names corresponding to subsections where they were discussed.

with the help of the function **cy.get()** that locates an element with the corresponding selector of the element passed into it as the parameter.

Next we emulate the user's action by typing username and password and clicking the button (**.type()** and **.click()** functions). At the end of the scenario we validate the error message using **.should()** function for assertion.

Listing 1: File **spec_2.1.cy.js**

```
describe('CHECK SWAG LABS LOG IN WITHOUT POM 2.1', () => {
  it('Validate locked out user', () => {
    cy.get('#user-name').type('locked_out_user');
    cy.get('#password').type('secret_sauce');
    cy.get('#login-button').click();
    cy.get('[data-test="error"]').should('be.visible')
      .and('contain.text',
        'Epic sadface: Sorry, this user has been locked out.');
  });
});
```

This approach lets us to obtain expected result of testing but in the same time it does not fit list of quality criteria [2] because of

- complexity of maintenance and updating,
- level of duplication increase dependent on covered scenarios growth,
- violation of principles: KISS, DRY, SOLID etc.

2.2. Tests with POM using selectors for elements

For improving the code above let us use the Page Object design pattern and update the code according to style used in Selenium/Java [3].

2.2.1. POM using selectors

First of all, we describe the class that contains selectors of elements in variables. Then, implementing functions and locators we use selectors predefined in variables. Here locators are implemented with **cy.get()** help and are used to perform actions on page.

Listing 2: File **login_2.2.1.page.js**

```
class MainPage {
  usernameInputSelector = '#user-name';
  passwordInputSelector = '#password';
  loginButtonSelector   = '#login-button';
  errorMessageSelector  = '[data-test="error"]';

  typeUsernameInput(username)
    { cy.get(this.usernameInputSelector).type(username); }

  typePasswordInput(password)
    { cy.get(this.passwordInputSelector).type(password); }

  clickLoginButton()
    { cy.get(this.loginButtonSelector).click(); }

  checkErrorMessage(message)
    { cy.get(this.errorMessageSelector).should('be.visible')
        .and('contain.text', message); }
}
module.exports = new LoginPage();
```

As a result spec file will take the following form.

Listing 3: File **spec_2.2.1.cy.js**

```
import loginPage  from "../pages/login_2.2.1.page";

describe('CHECK SWAG LABS LOG IN WITH POM 2.2.1', () => {
 it('Validate locked out user', () => {
   loginPage.typeUsernameInput('locked_out_user');
   loginPage.typePasswordInput('secret_sauce');
   loginPage.clickLoginButton();
   loginPage.checkErrorMessage('Epic sadface: " +
     'Sorry, this user has been locked out.');
 });
```

Considering pros and cons of this approach, we can say that it isolates the logics of selecting elements apart from processing scenarios. This isolation improves disadvantages illuminated in the previous approach. At the same time this approach has disadvantage because assertions are performed by class methods, and it is better to leave assertions in the spec file.

2.2.2. POM using getter for error message

Thus, the next step of optimization is extension of class functionality for providing a possibility to make assertions within the spec file.

With this aim, let us define a function in class that will give access to the element, a kind of getter for error messages.

Listing 4: File **login_2.2.2.page.js**

```
  getErrorMessage()
    { return cy.get(this.errorMessageSelector); }
```

Listing 5: File **spec_2.2.2.cy.js**

```
import loginPage  from "../pages/login_2.2.2.page";

describe('CHECK SWAG LABS LOG IN WITH POM 2.2.2', () => {
 it('Validate locked out user', () => {
   loginPage.typeUsernameInput('locked_out_user');
   loginPage.typePasswordInput('secret_sauce');
   loginPage.clickLoginButton();
   loginPage.getErrorMessage().should('be.visible')
     .and('contain.text',
       'Epic sadface: Sorry, this user has been locked out.');
 });
```

As a result of optimisation we obtained improvement of code. However, at the same time, the code is still not uniformed, as one element on the page has accessor function, and others do not have. Code can be understood better when we suppose that the necessity to call those elements directly in a spec file exists. For instance, when we need to verify that in case of unsuccessful username input the element changes its style (red color etc.).

2.2.3. POM using getters for all elements

Let us eliminate the shortcomings of the previous version and define functions for access to each page element.

Listing 6: File **login_2.2.3.page.js**

```
class MainPage {
  usernameInputSelector = '#user-name';
  passwordInputSelector = '#password';
  loginButtonSelector   = '#login-button';
  errorMessageSelector  = '[data-test="error"]';

  getUsernameInput(){ return cy.get(this.usernameInputSelector);}
  getPasswordInput(){ return cy.get(this.passwordInputSelector);}
  getLoginButton()  { return cy.get(this.loginButtonSelector);}
  getErrorMessage() { return cy.get(this.errorMessageSelector);}
  typeUsernameInput(username)
    { this.getUsernameInput().type(username); }
  typePasswordInput(password)
    { this.getPasswordInput().type(password); }
  clickLoginButton()
    { this.getLoginButton().click(); }
}
module.exports = new LoginPage();
```

Tests in **spec_2.2.3.cy.js** file are the same as in **spec_2.2.2.cy.js** file. At the same time, the new version contains more functionality for further extension and can cover bigger amount of user behavior scenarios with slight updates.

Evaluating the code, we observe that it currently seems overloaded with functions. That happens because we keep separately selectors of each element and create distinct functions to access each element that with help of selectors return us the locators of elements.

2.3. Tests with POM using locators for elements

One of the methods to simplify and optimise the code is the union of selectors and locators. It can be done because out of class only wrapped elements are used to perform actions on them. That is why there is no sense to keep distinct properties only for selectors.

2.3.1. POM using named functions as locators

Let us delete from the previous variant all the variables for selectors and strictly pass selectors as arguments in **cy.get()**. Each locator is obtained as a method that returns the element.

Listing 7: Part of file **login_2.3.1.page.js**

```
  getUsernameInput(){ return cy.get('#user-name');}
  getPasswordInput(){ return cy.get('#password');}
  getLoginButton   (){ return cy.get('#login-button');}
  getErrorMessage  (){ return cy.get('[data-test="error"]');}
```

We obtain 4 getter functions and focus on the necessary functionality. Then when we work with functions (**typeUsernameInput()**, **typePasswordInput()**, **clickLoginButton()**), we use the predefined locators of this class.

As a consequence the tests in **spec_2.3.1.cy.js** look the same as ones in **spec_2.2.2.cy.js** and work as well.

2.3.2. POM with anonymous functions as locators

Another way to define methods is the usage of anonymous functions. Function expressions are not hoisted, which means it is created only when the execution flow reaches it and can be used from that moment onwards.

Listing 8: Part of file **login_2.3.2.page.js**

```
  getUsernameInput=function(){ return cy.get('#user-name');}
  getPasswordInput=function(){ return cy.get('#password');}
  getLoginButton   =function(){ return cy.get('#login-button');}
  getErrorMessage  =function()
                         { return cy.get('[data-test="error"]');}
```

The same as in the previous case, tests do not have changes (tests in **spec_2.2.3.cy.js** are the same as in **spec_2.2.2.cy.js**).

2.4. Tests using POM using locators in elements object

The usage of previous approach does not make semantic separation between different parts of the Page Object file, that is why we can perform aggregation and place locators into the **elements** object thereby dividing the functions into two groups: functions devoted to access elements and functions devoted to actions.

2.4.1. POM using locators as functions in elements object

Let us use getters names as the keys and locators functions as the values in **elements**.

Listing 9: File **login_2.4.1.page.js**

```
class MainPage {
  elements = {
    getUsernameInput: function(){ return cy.get('#user-name');},
    getPasswordInput: function(){ return cy.get('#password');},
    getLoginButton   : function()
                         { return cy.get('#login-button');},
    getErrorMessage  : function()
                         { return cy.get('[data-test="error"]');}
  }
  typeUsernameInput(username)
    { this.elements.getUsernameInput().type(username); }
   typePasswordInput(password)
     { this.elements.getPasswordInput().type(password); }
   clickLoginButton()
     { this.elements.getLoginButton().click(); }
}
module.exports = new LoginPage();
```

According to this approach we refer firstly to the **elements** object and then to corresponding function every time we need to refer to the locators functions in code of action functions. Similar style we need to keep in the spec code also.

Listing 10: File **spec_2.4.1.cy.js**

```
import loginPage  from "../pages/login_2.4.1.page";
```

```
describe('CHECK SWAG LABS LOG IN WITH POM 2.4.1', () => {
 it('Validate locked out user', () => {
   loginPage.typeUsernameInput('locked_out_user');
   loginPage.typePasswordInput('secret_sauce');
   loginPage.clickLoginButton();
   loginPage.elements.getErrorMessage().should('be.visible')
     .and('contain.text',
       'Epic sadface: Sorry, this user has been locked out.');
 });
});
```

This approach can be found on some legacy projects or on projects not supporting modern stack.

2.4.2. POM using arrow functions as locators in elements object

Analyzing a block of elements we can note that it contains a long construction **function() { return ...}** and the idea to shorten or optimize it appears. It can be done if the project uses standard **JavaScript EcmaScript 6** or higher. Then, using arrow functions syntax we can obtain a more concise look of code.

Listing 11: Part of file **login_2.4.2.page.js**

```
  elements = {
    getUsernameInput: () => cy.get('#user-name'),
    getPasswordInput: () => cy.get('#password'),
    getLoginButton   : () => cy.get('#login-button'),
    getErrorMessage  : () => cy.get('[data-test="error"]')
 }
```

Due to point localization of the file, no updates needed for tests in **spec_2.4.2.cy.js** (they look the same as in **spec_2.4.1.cy.js**).

Investigating available articles on this topic we found out that most of the resources on the Internet recommend writing code with POM as it is shown in this subsection. However, in most cases they use incorrect names for functions in the **elements** object. The emphasis is on the subject and not on performing an action over the subject (verbs **get**, **take** etc. are omitted), for instance **usernameInput()**, **loginButton()**. When we need to refer to the part of elements in the tests definitely we want to name the elements as a noun, but round brackets for function call cannot be omitted. In that way code looks incomprehensible and weird violating name convention (**loginPage.elements.errorMessage().should('be.visible')**).

On the one hand this kind of names has benefits because they are used to denote locators of elements. But on the other hand it has the drawback: this part of code is performing some action and first of all it is a function and it should respect the name conventions for functions.

2.5. Tests with POM using assessor properties

The drawback described above can be avoided by using assessor properties. They are property getters, new types of properties (along with the regular data properties). They are essentially functions that execute on getting value, but look like regular properties to an external code. "getter" methods are accessor properties . In an object literal they are denoted by **get**. Descriptor for accessor properties has **get** – a function without arguments, that works when a property is read.

Listing 12: File **login_2.5.page.js**

```
class MainPage {
  get usernameInput()  { return cy.get('#user-name');}
  get passwordInput()  { return cy.get('#password');}
  get loginButton()    { return cy.get('#login-button');}
  get errorMessage()   { return cy.get('[data-test="error"]');}

  typeUsernameInput(username)
    { this.usernameInput.type(username); }
  typePasswordInput(password)
    { this.passwordInput.type(password); }
  clickLoginButton()
    { this.loginButton.click(); }
}
module.exports = new LoginPage();
```

Listing 13: File **spec_2.5.cy.js**

```
import loginPage  from "../pages/login_2.5.page";
describe('CHECK SWAG LABS LOG IN WITH POM 2.5', () => {
 it('Validate locked out user', () => {
   loginPage.typeUsernameInput('locked_out_user');
   loginPage.typePasswordInput('secret_sauce');
   loginPage.clickLoginButton();
   loginPage.errorMessage.should('be.visible')
     .and('contain.text',
       'Epic sadface: Sorry, this user has been locked out.');
 });
```

The getter works when **errorMessage** is read in spec. From the outside, an accessor property looks like a regular one. That's the idea of accessor properties. We don't call **errorMessage** as a function, we read it normally: the getter runs behind the scenes.

A minor drawback of this approach is that sometimes it is necessary to pass a parameter to the locator functions (for example referring to the **i**-th row of the table), then we need to combine the use of ordinary functions and assessor properties.

3. Conclusions

This article shows eight approaches how to construct Cypress tests using POM. The connections between them are stressed as their evolution while writing code developing E2E tests. Project using approaches highlighted in first subsections still can be found on the Internet while studying how to write tests with Cypress JS. The authors highlight advantages and disadvantages of the approaches and offer the solution of problems. This article can be used both as a combined overview of different approaches and as a manual for those who are struggling to write tests with Cypress in a better way. For further investigation other patterns and solutions like Cypress commands, aliases, application actions, could be discovered.

References

[1] J. T. Othayoth, S. Anuar, Modern web automation with cypress.IO, Open International Journal of Informatics 10 (2022) 182–196. URL: https://oiji.utm.my/index.php/oiji/article/view/229.

[2] M. Leotta, M. Biagiola, F. Ricca, M. Ceccato, P. Tonella, A Family of Experiments to Assess the Impact of Page Object Pattern in Web Test Suite Development, in: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), 2020, pp. 263–273. doi:10.1109/ICST46399.2020.00035.

[3] E. Vila, G. Novakova, D. Todorova, Automation Testing Framework for Web Applications with Selenium WebDriver: Opportunities and Threats, in: Proceedings of the International Conference on Advances in Image Processing, ICAIP '17, Association for Computing Machinery, New York, NY, USA, 2017, p. 144–150. doi:10.1145/3133264.3133300.