

Prompt 6: Test without tags

Smell: **Test without tags**

End to End test code without smell

```
// loginTest.cy.js

```js
//File : loginTest.cy.js

describe('TEST', () => {

 it('Login Test', { tags: '@smoke' } , () => {

 const username = Cypress.config('username');
 const password = Cypress.config('password');

 cy.visit(Cypress.config('baseUrl') + Cypress.env('loginUrl'), {
 failOnStatusCode: false });

 cy.get('[id="username"]').type(username);

 cy.get('input[type="submit"]').click();

 cy.get('[id="password"]').type(password);

 cy.get('input[type="submit"]').click();

 cy.get('.user-info').invoke('text').should('include',
 expectedText.textMenu);
 });

});

```
```

End to End test code with smell

```
// testWithoutTags.cy.js

```js
//File : loginTest.cy.js

describe('Test', () => {

 const expectedText = {
 textMenu: 'administrator'
 };

});

```
```

```

it('Login Test', () => {

  const username = Cypress.config('username');
  const password = Cypress.config('password');

  cy.visit(Cypress.config('baseUrl') + Cypress.env('loginUrl'), {
    failOnStatusCode: false });

  cy.get('[id="username"]').type(username);

  cy.get('input[type="submit"]').click();

  cy.get('[id="password"]').type(password);

  cy.get('input[type="submit"]').click();

  cy.get('.user-info').invoke('text').should('include',
    expectedText.textMenu);
  });
})
```

```

## Full prompt submitted to ChatGPT

Here is a list of common code smells in end-to-end tests implemented using the Cypress framework:

- \* Incorrect use of asserts: using asserts (e.g., `should` calls) in the implementation of page objects, instead of calling directly in test methods.

- \* Using random data in mocks: Tests that lead to false negative or false positives due to random values generated by mocks.

- \* Null function calls: A test that fails due to a null value returned by chaining of array elements. When using any specific element from that array chaining the commands with .eq(), .first(), .last(), etc. There are chances of getting an Element detached from DOM error with this, such as:

```

```js
cy.get('selector').eq(1).click()
```

```

- \* Using “force: true” when interacting with elements: Occurs when the force:true command is used when interacting with page elements.

- \* Using cy.contains() without selectors: A test that uses cy.contains() to select an element based on the text it contains, e.g., cy.contains("please click here").

- \* Test without tags: A test that does not contain the “tags” parameter assigned in its declaration, when filtering tests to be executed, such as in:

```

    ````js
      it('Description') , () => {
        ....
      }
    ````

```

\* Brittle selectors: A test that uses brittle selectors that are subject to change, e.g., `cy.get(["customized id"])` instead of `cy.get(["random id"])`.

\* Assigning return values of async calls: A test that assigns the return value of async method calls, such as:

```

    ````js
      let x = cy.get('selector');
      x.click();
    ````

```

\* Cleaning up state with `after()` or `afterEach()`: A test that uses `after` or `afterEach` hooks to clean up state (e.g., to release resources acquired by the test).

\* Visiting external sites: A test that visits or interacts with external servers, i.e., sites that we do not control.

\* Starting Web servers: A test that starts a web server using `cy.exec()` or `cy.task()`.

\* Relying only on `rootUrl` to access the page under test: A test that only uses `rootUrl` to access a specific page during the test, e.g., assuming that `xyz.com` redirects to `xyz.com/login`, which is the page that we need to test.

\* Missing global `BaseUrl`: A test suite that calls `cy.visit()` without setting a `baseUrl`. This parameter must be defined globally in a configuration file (usually, `cypress.config.js`).

\* Using unnecessary checks for previously actionable elements : A test that uses redundant visibility checks when dealing with actionable elements such as `should("be.visible")` or `should("exist")` before actions (click or type), such as in:

```

    ````js
      cy.get('selector').should('be.visible').click();
    ````

```

\* Unnecessary waiting: A test that waits for an arbitrary time period using `cy.wait()`.

\* Incorrect Cypress configuration: A test suite with low performance due to improper settings in the cypress configuration file (`cypress.json`), such as `port`, `timeout` or `video`.

\* Duplicated code: When two or more tests share identical code snippets.

\* Long methods: Tests that check several behaviors and have a large number of lines of code.

\* Large classes: Classes with a very high number of end-to-end tests.

\* Conditional overuse: Occurs when there is a condition in the test, i.e., due to a specific event the test may return different values.

Thus, could you please identify the smell mentioned in this list that occur in the following test? The test has 1 source code file: loginTest.cy.js . Each file is delimited using markdown code blocks. A comment in the first line of each file describes its name. Please, in your answer mention at most one smell.

```
```js
//File : loginTest.cy.js

describe('Test', () => {

  const expectedText = {
    textMenu: 'administrator'
  };

  it('Login Test', () => {

    const username = Cypress.config('username');
    const password = Cypress.config('password');

    cy.visit(Cypress.config('baseUrl') + Cypress.env('loginUrl'), {
      failOnStatusCode: false });

    cy.get('[id="username"]').type(username);

    cy.get('input[type="submit"]').click();

    cy.get('[id="password"]').type(password);

    cy.get('input[type="submit"]').click();

    cy.get('.user-info').invoke('text').should('include',
      expectedText.textMenu);
  });
})
```
```

## ChatGPT full answer

### (1 attempt)

One of the code smells in the given test is the "Unnecessary waiting" smell:

```
```js
cy.wait(time in seconds);
```
```

In the provided test code, there is no explicit use of `cy.wait()` command, but the test contains an unnecessary wait. The following lines of code perform back-to-back clicks on the same element without any wait or delay between them:

```
```js
cy.get('input[type="submit"]').click();

cy.get('input[type="submit"]').click();
```
```

This can be problematic as the test may not accurately reflect real user behavior. In a real scenario, users might not click on the same element twice rapidly without any time delay. The test should be adjusted to have a more realistic user interaction pattern, and unnecessary waits should be avoided.

It's important to ensure that test actions are performed at a pace that reasonably simulates user behavior and to avoid unnecessary waits that can lead to false positives or negatives in test results.

## (2 attempt)

One code smell that occurs in the provided test (loginTest.cy.js) is:

**\*\*Using cy.contains() without selectors\*\***

This code smell is present in the following line of the test:

```
```js
cy.get('.user-info').invoke('text').should('include',
expectedText.textMenu);
```
```

Here, `cy.get('.user-info')` selects an element with class "user-info," and then `.invoke('text')` is used to retrieve the text of that element. The subsequent `.should('include', expectedText.textMenu)` checks whether the retrieved text includes the expected text "administrator."

However, using `cy.contains()` without providing a specific selector can lead to issues in the test. If there are multiple elements with the class "user-info" on the page, `cy.contains()` will return the first matching element, which might not be the one intended for testing. This can result in false positives or incorrect test results.

To improve this, it is better to use a more specific selector to target the exact element that should contain the expected text, for example, by adding a unique ID or more specific class to the element. This helps ensure that the test targets the correct element and avoids ambiguity.

## (3 attempt)

One code smell in the provided test is:

**\*\*Brittle selectors\*\***: The test uses hardcoded, brittle selectors in the form of `[id="username"]` and `[id="password"]` to locate the username and password input fields, respectively. Using such specific, implementation-dependent selectors makes the

test vulnerable to breaking if the HTML structure or attribute values change even slightly.

To improve this, it's better to use more reliable selectors like data attributes, classes, or other non-fragile identifiers. For example, if the input fields have unique classes or data attributes, you can use those instead of relying on the specific `id` attributes.

Addressing the Brittle Selector smell, the test can be updated as follows:

```
```js
//File : loginTest.cy.js
describe('Test', () => {

  const expectedText = {
    textMenu: 'administrator'
  };

  it('Login Test', () => {

    const username = Cypress.config('username');
    const password = Cypress.config('password');

    cy.visit(Cypress.config('baseUrl') + Cypress.env('loginUrl'), {
      failOnStatusCode: false });

    cy.get('[data-cy="username"]').type(username);
    cy.get('[data-cy="login-button"]').click();
    cy.get('[data-cy="password"]').type(password);
    cy.get('[data-cy="login-button"]').click();

    cy.get('.user-info').invoke('text').should('include',
      expectedText.textMenu);
  });
});
```
```

In this updated version, we used `data-cy` attributes for selecting the username and password input fields. Data attributes are more robust and less likely to change during UI updates, reducing the chances of test failures due to selector changes. Additionally, we used `[data-cy="login-button"]` to identify the login button element, ensuring a clear and reliable way to locate the button.

#### (4 attempt)

One code smell that occurs in the provided test is:

**\* \*\*Test without tags\*\*:** The test function "Login Test" does not contain any `tags` parameter assigned in its declaration. The absence of tags may make it difficult to filter or categorize tests for specific purposes or scenarios.

To address this smell, you can add a `tags` parameter to the test declaration, like this:

```
```js
```

```
it('Login Test', { tags: ['smoke', 'login'] }, () => {  
  // Test code...  
});  
`;
```

In this example, I've added two tags: "smoke" and "login," which can be useful for categorizing the test and filtering tests during execution. You can choose relevant tags based on your project's requirements and organization's testing strategies.