

## Prompt 5: Using cy.contains() without selectors

Smell: Using cy.contains() without selectors

### End to End test code without smell

```
// registerTest.cy.js

```js
//File : registerTest.cy.js

describe('Mantis test', () => {

    const expectedText = {
        register: "No Change Log information available. Issues are
included once projects have versions and issues are resolved with \"fixed
in version\" set."
    };

    it('Click in Registration',{ 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(':nth-child(4) > a > .menu-text').click();

        cy.get('.lead').invoke('text').should('include',
expectedText.register);

    })

})
```
```

### End to End test code with smell

```
// usingCyContainsWithoutSelectors.cy.js

```js
```

```
//File : registerTest.cy.js

describe('Mantis test', () => {

  const expectedText = {
    register: "No Change Log information available. Issues are
included once projects have versions and issues are resolved with \"fixed
in version\" set."
  };

  it('Click in Registration',{ 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.contains("Change Log").click();

    cy.get('.lead').invoke('text').should('include',
expectedText.register);

  })

})
\`\`\`
```

## 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: registerTest.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 : registerTest.cy.js

describe('Mantis test', () => {

    const expectedText = {
        register: "register: "No Change Log information available.  Issues
are included once projects have versions and issues are resolved with
\"fixed in version\" set."
    };

    it('Click in Registration', { 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.contains("Change Log").click();

        cy.get('.lead').invoke('text').should('include',
expectedText.register);

    })
})
```
```

**ChatGPT full answer**

### (1 attempt)

One code smell that occurs in the given test is "Assigning return values of async calls":

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

In the test, the variables `username` and `password` are assigned the return values of async calls `Cypress.config('username')` and `Cypress.config('password')`. While this may work in some cases, Cypress commands are asynchronous and typically return promises. Assigning their return values directly to variables may not give the desired results, especially when used in some complex scenarios. It's better to use these values inside Cypress commands as part of the test flow.

To properly handle asynchronous operations, you should either use Cypress commands directly or wrap these assignments inside a `.then()` block or use `async/await` syntax. Here's an example of how you can handle this properly:

```
```js
//File : registerTest.cy.js

it('Click in Registration',{ tags: '@smoke' }, () => {

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

    cy.get('[id="username"]').type(Cypress.config('username'));
    cy.get('input[type="submit"]').click();
    cy.get('[id="password"]').type(Cypress.config('password'));
    cy.get('input[type="submit"]').click();

    cy.contains("Change Log").click();

    cy.get('.lead').invoke('text').should('include',
expectedText.register);
})
```
```

By directly using `Cypress.config`

### (2 attempt)

One code smell that occurs in the provided test is "Cleaning up state with after() or afterEach()".

In the test, there is no cleanup mechanism or hooks (`after` or `afterEach`) to release resources acquired during the test execution. After running this test, there might be some lingering state that could potentially affect the subsequent tests. For example, if this test creates or modifies data during its execution, it might not clean up or reset the state, leading to interference with other tests in the suite.

To address this smell, it's good practice to add appropriate cleanup steps using `after` or `afterEach` hooks to reset the state or release resources after the test finishes its execution. This ensures that subsequent tests start with a clean slate and are not affected by the state of previous tests.

### (3 attempt)

One code smell that occurs in the provided test is:

**\*\*Unnecessary waiting:\*\*** In the test, there is a redundant wait using `cy.wait()` after clicking the submit button for entering the password. This wait does not serve any purpose as there is no reason to wait for an arbitrary time in this scenario.

```
```\njs\ncy.get('[id="password"]').type(password);\ncy.get('input[type="submit"]').click();\n\n// Unnecessary waiting\ncy.wait(2); // This wait is unnecessary and can be removed.\n```\n
```

This `cy.wait(2)` is an arbitrary wait of 2 seconds and doesn't add any value to the test. It's generally not recommended to use arbitrary waits like this in end-to-end tests, as they can lead to test flakiness and reduced test performance. Instead, it's better to use appropriate Cypress commands like `cy.contains()` or `cy.get()` with appropriate wait options to ensure synchronization with the application.

### (4 attempt)

One code smell that occurs in the given test (`registerTest.cy.js`) is:

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

In the test, the line `cy.contains("Change Log").click();` uses `cy.contains()` to select an element based on the text it contains. While this may work, it can lead to issues if there are multiple elements with the same text on the page. Using `cy.contains()` without providing a more specific selector can make the test brittle and prone to false positives or negatives if the page structure changes.

To improve this, it's better to use a more specific selector along with `cy.contains()` to target the desired element uniquely, ensuring the test is more reliable and less likely to break due to changes in the page layout or content.

For example, you could add a CSS class or unique attribute to the element and use that as a selector:

```
```\njs\ncy.get('[data-testid="change-log-button"]').click();\n```\n
```

This would be more robust than relying solely on the text content of the element.