

## Prompt 2: Using random data in mocks

Smell: **Using random data in mocks**

End to End test code without smell

```
// projectTest.cy.js

```js
// File: ProjectTest.cy.js

describe('Project', () => {

  it('Create Project',{ 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(7) > a > .menu-text').click();

    cy.get('a[href="/manage_proj_page.php"]').click();

    const name = "Project Name";

    const description = "Project Description";

    const nameP = name;

    cy.xpath('//button[@type="submit" and text()="Create New Project
"]').click();

    cy.get('[id="project-name"]').type(name);

    cy.get('[id="project-description"]').type(description);

    cy.xpath('//input[@type="submit"]').click();

    cy.get('a').invoke('text').should('include', nameP);

  })
})
```
```

## End to End test code with smell

```
// usingRandomDataInMocks.cy.js

```js
// File: ProjectTest.cy.js

import {
  faker
} from '@faker-js/faker';

describe('Project', () => {

  it('Create Project',{ 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(7) > a > .menu-text').click();

    cy.get('a[href="/manage_proj_page.php"]').click();

    const name = faker.name.jobTitle();

    const description = faker.lorem.lines(2);

    const nameP = name;

    cy.xpath('//button[@type="submit" and text()="Create New Project"]')
    .click();

    cy.get('[id="project-name"]').type(name);

    cy.get('[id="project-description"]').type(description);

    cy.xpath('//input[@type="submit"]').click();

    cy.get('a').invoke('text').should('include', nameP);

  })
})
```
```

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: projectTest.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: ProjectTest.cy.js

import {
  faker
} from '@faker-js/faker';

describe('Project', () => {

  it('Create Project',{ 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(7) > a > .menu-text').click();

    cy.get('a[href="/manage_proj_page.php"]').click();

    const name = faker.name.jobTitle();

    const description = faker.lorem.lines(2);

    const nameP = name;

    cy.xpath('//button[@type="submit" and text()="Create New Project
"]').click();

    cy.get('[id="project-name"]').type(name);

    cy.get('[id="project-description"]').type(description);

    cy.xpath('//input[@type="submit"]').click();

    cy.get('a').invoke('text').should('include', nameP);

  })
})
...

```

## ChatGPT full answer

### (1 attempt)

The code smell that occurs in the given test (projectTest.cy.js) is:

**\* \*\*Using cy.contains() without selectors\*\*:** The test uses `cy.contains()` to select an element based on the text it contains, without providing any additional selector to narrow down the search. This can lead to unintended matches and make the test less robust. In the test, `cy.get('a')` is used to select an anchor element, and then `.invoke('text').should('include', nameP)` is used to check if the text of any anchor element includes the value of `nameP`. This can potentially lead to false positives if there are multiple anchor elements with the same text content.

A better approach would be to use a more specific selector to identify the exact anchor element you want to verify. For example, if the anchor element has a specific class or a unique identifier, you should use that in combination with `cy.contains()` to make the selection more precise.

Overall, this code smell can be addressed by using a more specific selector with

``cy.contains()`` to target the exact element you want to check for the presence of ``nameP``.

## (2 attempt)

One smell in the given test is "Using random data in mocks."

The test is generating random data for the project name and description using the ``faker`` library. While using random data in tests can be useful to some extent, it can lead to false negatives or false positives if the test relies heavily on these randomly generated values. Randomly generated data might not always match the expected values, causing the test to fail even if the application behaves correctly.

```
```js
const name = faker.name.jobTitle();
const description = faker.lorem.lines(2);
```
```

To improve this, it's better to use predefined test data that represents specific scenarios or use test data from the test environment, which ensures more predictable and stable test results.