

## Documento 1

### Organizing Tests, Logging In, Controlling State


Anti-Pattern: Sharing page objects, using your UI to log in, and not taking shortcuts.

Best Practice: Test specs in isolation, programmatically log into your application, and take control of your application's state.

### Selecting Elements

Anti-Pattern: Using highly brittle selectors that are subject to change.

Best Practice: Use `data-*` attributes to provide context to your selectors and isolate them from CSS or JS changes.

		selector
<code>cy.get('[data-cy=submit]').click()</code>	 Always	Best. Isolated from all changes.

### Assigning Return Values

Anti-Pattern: Trying to assign the return value of Commands with `const`, `let`, or `var`.

Best Practice: Use closures to access and store what Commands yield you.

```
// DONT DO THIS. IT DOES NOT WORK
// THE WAY YOU THINK IT DOES.
const a = cy.get('a')

cy.visit('https://example.cypress.io')

// nope, fails
a.first().click()
```

### Visiting external sites

Anti-Pattern: Trying to visit or interact with sites or servers you do not control.

Best Practice: Only test what you control. Try to avoid requiring a 3rd party server. When necessary, always use `cy.request()` to talk to 3rd party servers via their APIs.

## Having tests rely on the state of previous tests

Anti-Pattern: Coupling multiple tests together.

Best Practice: Tests should always be able to be run independently from one another and still pass .

## Creating "tiny" tests with a single assertion

Anti-Pattern: Acting like you're writing unit tests.

Best Practice: Add multiple assertions and don't worry about it

## Using after or afterEach hooks

Anti-Pattern: Using after or afterEach hooks to clean up state.

Best Practice: Clean up state before tests run.

## Unnecessary Waiting

Anti-Pattern: Waiting for arbitrary time periods using `cy.wait(Number)` .

Best Practice: Use route aliases or assertions to guard Cypress from proceeding until an explicit condition is met.

## Web Servers

Anti-Pattern: Trying to start a web server from within Cypress scripts with `cy.exec()` or `cy.task()` .

Best Practice: Start a web server prior to running Cypress.

## Setting a global baseUrl

Anti-Pattern: Using `cy.visit()` without setting a baseUrl .

Best Practice: Set a baseUrl in your configuration file (`cypress.json` by default) .

## Documento 2

### 1. Unprepared Element

An element is interacted with but the data the application expects to be in place is not yet in place, resulting in undesired behavior.

## **2. Flickering Element**

An element appears but is then quickly removed from the DOM before Cypress can complete its interaction.

## **3. Impatient Test**

A test kicks off an asynchronous operation, but then, instead of waiting for it to complete, the test moves on in a way that will cause a problem.

1. `cy.wait()` won't scale beyond a few tests. If you want to thoroughly test your app with Cypress, you'll have a lot of tests, and if they all have wait commands your test suite will be extremely slow.
2. Relying on `cy.wait()` makes your tests flaky. Exactly how long do you need to wait? What is fast enough now may not be fast enough in the future, or on CI.
3. `cy.wait()` masks real application bugs.

# **DOCUMENTO 3**

## **TODOS DO DOCUMENTO 3 MAIS**

11. Avoid { force: true }

12. Use `cy.contains()` along with a selector

When we use `cy.contains()` to select an element with some text or to assert.

13. Avoid chaining of array elements with `.eq()`, `.first()` etc. in the test case

14. Use `scrollIntoView()` to detect partially visible elements

15. Use `loginPath` instead of `rootUrl` for login test

16. No need of common assertions for actionable elements before performing actions

## **Documento 4**

## The Smell

You may be wondering why I am referring to the `cy.wait()` command as being “smelly”. Don’t get me wrong, it’s not the method but the way we implement it that makes the smell.

## Documento 5

Mesmos Smells

<https://docs.cypress.io/guides/references/best-practices>

## Documento 6

### 1. Independent tests

Automated tests must run in isolation, without the need for another test to run to create some state in the application under test.

*The failure of one test shouldn't impact the results of other tests.*

In addition, independent tests are easier to parallelize.

Tip: use the `beforeEach` hook when you need to perform repeated steps for all tests of a given describe or context.

### 2. Programmatic authentication

Logging in via the graphical user interface as a pre-condition for all tests is costly (in terms of execution time), and it makes the tests dependent on each other, which is a bad practice.

A disabled login button due to an HTML error, for example, shouldn't break an entire test suite.

By adding mechanisms to log in programmatically, you make tests faster and more independent.

### 3. State creation mechanisms

By creating such mechanisms, we ensure that tests are completely decoupled from each other, we don't need abstractions that add complexity to tests, such as Page Objects, and we guarantee fast and straight-to-the-point tests.

Examples of these mechanisms are API calls for creating resources, communication with the database via tasks, or running scripts at the operating system level.

### 4. data-\* attributes

Adding attributes to front-end elements, such as data-test, data-testid, or data-cy, adds to the application what I call "testability," as these attributes are specifically created for testing purposes, decreasing the chances of front-end changes break the tests.

5. Don't do this: `var el = cy.get('selector')`

Remember. Cypress is not Selenium! 😊

Cypress has its own architecture, where although in many cases it allows writing code that seems synchronous, even if it is asynchronous (since it puts each command in a queue for later execution), it is not possible to do something like:

```
const myBtn = cy.contains('button', 'My button')
myBtn.click()
```

However, you can do:

```
cy.contains('button', 'My button').as('myBtn')
cy.get('@myBtn').click()
```

*Most of Cypress's commands are chainable!*

6. Do not test external applications

Relying on Google's GUI login for your tests or some third-party API can make your tests unstable, as changes to these services (which you don't control) will break your tests, even if it's all right "on your side."

To ensure that such services are in line with your application, you can have a smoke-test suite, for example, or contract tests.

7. Tests too small or too large

End-to-end tests are not unit tests. Due to the cost of running them, it's worth adding more than one assertion per test to save time.

However, beware of extensive tests. Maybe these are testing a lot of unrelated things and could be broken down into smaller ones.

8. Do not use the after and afterEach hooks

If something goes wrong during the execution of the tests, such hooks run the risk of not being executed, leaving "junk" in the application.

As a good practice, do any cleanup before running the tests, using the beforeEach hook, for example.

9. Do not use `cy.wait(Number)`

Cypress already has automatic waiting with different default timeouts to wait for elements to be visible, for animations to end, for requests to be sent and responded and for pages to be loaded.

What you can do to make your tests even more robust is wait for elements to be visible:

E.g.: `cy.get('[data-cy="avatar"]').should('be.visible')`

Or, you can wait for a particular request that you gave an alias to finish.

E.g.:

`cy.intercept(...).as('myReq')`

#### 10. start-server-and-test

Use mechanisms such as the start-server-and-test library to initialize the application server before running the tests

Integration server.

#### 11. Set the baseUrl

By setting the baseUrl in the configuration file (cypress.json),

## DOCUMENTO 7

### Waiting...

Modern automation frameworks such as Cypress and Playwright use implicit and explicit waiting until a page has loaded, an element is visible, or an element can be interacted with. This ensures that tests are less flakey and prone to failure by giving the browser time to load into a testable state. Once the load has completed, testing may resume execution

### Taking UI Testing too Literally

If you or your engineers are setting up test state using the UI, you may be participating in a bad testing practice. Generating test state through the UI is cumbersome and can contribute to test flake. Instead, use existing application implementation to your advantage to create hermetically sealed user journeys.

When testing application login for a brand new user, do not build a new user through the UI, then attempt to login. Your test is no longer hermetically sealed as it is now testing two user journeys:

1. New user registration
2. Application login

Instead, create a request for a new user using your application's API, then attempt to login with the newly built credentials. This can be done effortlessly using Cypress' built-in request library and aliasing.

### **Testing Independent of CI**

An all-too-common anti-pattern in test automation is to run tests independently of Continuous Integration. The direct result being that failing tests have no immediate repercussions.

### **Behaviorally Driven Overhead**

I used to be a big proponent of Behaviorally Driven Development (BDD), so much so that I wrote a best practices standard for two of the companies I have worked for in the past. However, I have moved past using BDD after having come to the conclusion that the process provides little to no value while acting as a source of refactoring pain.

Behaviorally Driven Development tools such as Cucumber are wonderful to work with in the dreamy scenario where the entire company has bought into the art of BDD. All too often however, it is the testing team which writes in Gherkin while other aspects of the business either ignore the practice, or do not contribute. In this scenario, BDD loses its value as a collaboration tool between departments. Instead of bringing product management, development, and QA closer, the practice ends up alienating QA.

Another reason for removing BDD is the amount of overhead involved in writing and refactoring a test. Consider a user journey where a regular user navigates to a login page, submits valid information, and checks for success. We can write that in a feature.

## **Conditions**

Testing engineers should seek to remove as much conditional logic as possible from a test in order to reduce flake. Conditional testing generates non-deterministic tests, those being difficult to troubleshoot and run with confidence.<sup>2</sup>

## **Choosing the Wrong Selectors**

Often times using the correct selector criteria can be a difficult task. A common anti-pattern in test automation is to use highly brittle selectors when building page objects or writing tests. Brittle selectors are those that can change due to



implementation refactoring. Improper selector criteria would be the use of non-unique IDs and classes, or Xpath.<sup>3</sup>

## **Documento 8**

Cypress Intercept (Data Mocking) bad practice for functionality testing?

in my team we had the same discussion. It depends on what you want to test. If you just want to check that your Front-end code runs properly and does what it is expected to do with a given response, you should mock the responses. In this way you can also provide a wrong response and check that your Front-end shows the corresponding error, for instance. But we have decided not to mock responses because like this we can detect not only Front-end issues, but also the Back-end ones so we are able to warn their team, even if it is not our "fault".

## **DOCUMENTO 9**

Writing end2end tests with Cypress and cucumber, on a high level the test cases are as following  
the testing steps are now out of order. This may present a problem during future maintenance.  
I personally think there should be a more elegant solution. Any suggestions?

## **Documento 10**

Why cypress tests use force if it's not recommended

## **Documento 11**

Stop using Page Objects and Start using App Actions

## **Documento 12**

mesmos documento 2

## **Documento 13**

Cypress execution order, you need to wrap any calls to the interceptors with `cy.then()`.

Pitfall: Data that comes outside of Apollo

With all the ease that recording and code generation gives, it might be easy to forget that there might be other sources of data on the path to success.

If you've set-up all the mocks that were captured and your application still doesn't behave as expected, check the network tab to see if there ain't any good 'ol REST call on the way of our success!

If yes, and they look relevant to your action, you might need to make another mock for this purpose. Fortunately, Cypress provides us with the necessary built-in tools to move forward.

## DOCUMENTO 14

No Cypress, você pode usar uma instrução de espera para passar do tempo de debounce, mas adicionar instruções de espera baseadas em tempo no Cypress é um antipadrão

## Documento 15

Abandone Page Objects e comece a aplicar App Actions

## Documento 16

Cypress: Fix "use UI to login" anti-pattern

Keep a single test using the UI to login and add a Cypress command to login programatically because UI login is slow, cumbersome, and unnecessary.

## Documento 17

**Menos específicos**

Code Duplication

Poor Locators

Long Class or Methods

Bad Waits

Incorrect Use of Asserts

## Documento 18

Cypress – wait for button to be clickable

## Documento 19

In Cypress:

Anti-Pattern: Using highly brittle selectors that are subject to change.

Best Practice: Use data-\* attributes to provide context to your selectors and isolate them from CSS or JS changes.

## Documento 20

The Page Objects anti-pattern

## Documento 21

Anti-Pattern

Don't try to start a web server from `cy.exec()`.

## Documento 22

You rarely have to ever use `const`, `let`, or `var` in Cypress. If you're using them, it's usually a sign you're doing it wrong.

Best Practice: Tests should always be able to be run independently from one another and still pass.

## Documento 23

## Cypress Tip: Don't Overuse the Visibility Assertion

It makes sense why we might do this: to avoid interacting with an element before it's visible. However, this is usually unnecessary and could be considered bad practice.

First, it's unnecessary because of two implicit behaviors of Cypress: actionability assertions and command retry-ability. Cypress will not attempt to perform certain actions on an element unless it's visible. If it isn't visible, Cypress repeatedly retries this assertion until either the assertion passes and the next command is executed or the timeout is reached and it fails.

Now the test can be written this way:

```
describe('Sign In', () => {  
  before('navigate to Sign In', () => {  
    // ...  
  })  
  it('sign in', () => {  
    cy.get('#username')  
      .type('iheartjs')  
    cy.get('#password')  
      .type('password')  
    cy.get('button#sign-in')  
      .click()  
    // ...  
  })  
})
```

## Documento 24

Write one big test rather than several small ones

A real-world integration test typically involves signon, etc before testing the actual functionality. Do these as As Cypress's best practices document [explains](#), Cypress does some housekeeping between each test. This will itself slow you down if there are too many small tests.

Avoid waiting for arbitrary periods of time

Use before & beforeEach judiciously.

Tweak Cypress's configuration to remove unnecessary disk I/O

Tag tests and run only the ones you need to

Incorporate Cypress into your CI/CD

Stop using Page Objects and Start using App Actions