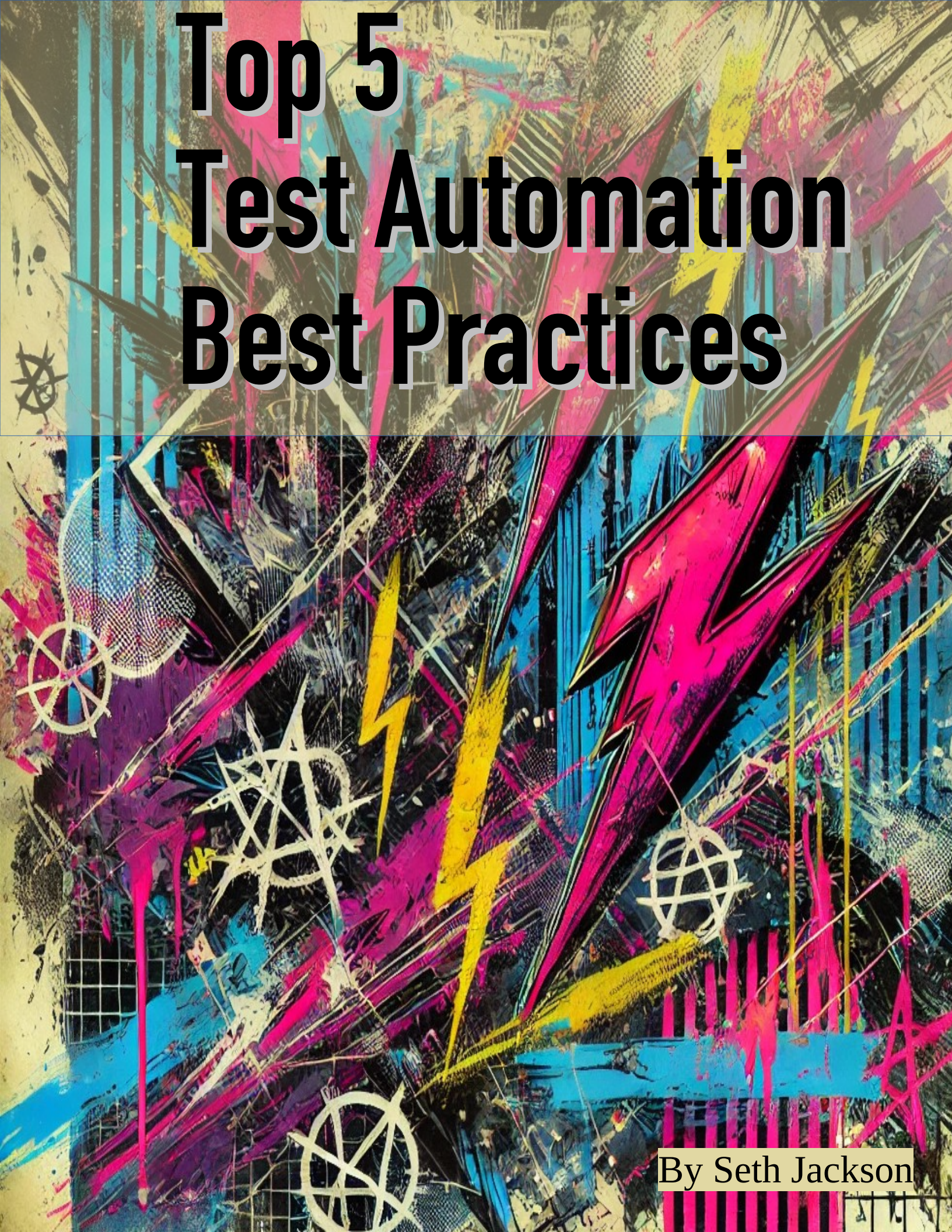
Top 5 Test Automation Best Practices



By Seth Jackson

Top 5 Test Automation Best Practices:

A Quick Guide for Software Testers

Introduction

Hello! My name is Seth Jackson, and I have 24 years of experience in software test automation. Over the years, I have worked with a wide range of testing tools, frameworks, and methodologies, helping teams improve their software quality through reliable automation.

In this guide, we will explore the **top 5 best practices** for test automation. These practices are designed to help testers build scalable, maintainable, and effective automation frameworks. Whether you are just starting with automation or looking to refine your existing strategy, these guidelines will provide practical insights to enhance your testing efforts.

1. Automate The Happy Path

Make sure each feature has the happy path tested. The happy path is the most common and expected user journey through an application, where everything works as intended without errors or unexpected deviations.

- Does the login form log you in?
- Does the “buy” button actually let you make the purchase?
- Does the “Save” button actually save your edits?

As testers, we think our job is to break things and find bugs. But that’s only part of it. The main part of our job is to make sure the customers can use the software. The app can be buggy as all hell, but if the customer can still get what they want done, the high-priority, critical features still work.

For instance, consider a login page. This page can have some misspellings (bug). Maybe it doesn’t look great on mobile (bug). That 1024 character email address might not work, yet (bug). However, if 99% of your customers can log in to the app, they can forgive the errors. The happy-path works.

On the other hand, everything could look perfect, but logging in doesn’t work. The happy-path is broken. Your customer support team is going to be swamped with angry customers. Your company will lose money.

Where does the happy path come from?

The main source should be your project manager. They are the ones who define the use-case scenarios the developers will implement. Those scenarios should be turned into tests.

The UX designers are another source. They design the workflows and UIs the customers will use. Check their designs (usually in an app like Figma) and make sure to automate their described workflows.

On a sprint-to-sprint basis, the developers’ tickets should have their happy-path defined. This is often called the “acceptance criteria”: what needs to be implemented to say the ticket is “done”. The acceptance criteria should be turned into tests.

Unfortunately, sometimes it’s up to you to define the happy-path scenarios for a feature. This is a sign of dysfunction in a company, though. You should work with management to make sure the happy paths are defined, use cases are understood, and the acceptance criteria are clearly listed in each ticket BEFORE you start testing.

A critical aspect of quality is CLARITY. A lack of clarity always leads to a lack of quality.

2. Design Tests for Reusability

Don't Repeat Yourself (D.R.Y.)

If you find yourself copy-and-pasting your code from test to test, STOP! Instead, ask yourself, "Can I turn this into a reusable method?" Copy-paste code is brittle code. If something in the app changes, you will have to hunt down all the places where you pasted your code and fix it. Most likely, you will miss some spots causing bad test reports and more time you have to stop and fix code.

If you make a reusable method, there's only one place you need to update the code if it changes. That's it. What's more, you are building a library of reusable methods. A library all your team members can use. This makes test development faster for everyone.

If you copy and paste code, just smack yourself. Hard. And then think of how you can make that code reusable.

Use Page Objects

Page Objects are excellent for reusability. You create a class that represents a page of your UI. The members and methods of that class represent elements and actions you can perform on that page.

For instance, a login page at YourSite.com/login could look like the following Python code:

```
class LoginPage:
    def __init__(self, driver):
        self.driver = driver
        self.username_input = driver.find_element(By.ID, "username")
        self.password_input = driver.find_element(By.ID, "password")
        self.login_button = driver.find_element(By.ID, "login")

    def login(self, username, password):
        self.username_input.send_keys(username)
        self.password_input.send_keys(password)
        self.login_button.click()
```

I also consider "components" to be page objects. For instance, the header and footer are the same on every page. The header is a component and can be its own page object. The same can be said for any component that is used on more than one page. Use the D.R.Y. principle and create separate page object classes for any reusable components.

Abstract Your Test Data

Often, you will need to test multiple types of data, hardware, operating systems, and so on. Your test doesn't change, but the target of your test does.

For instance, you need different screen dimensions for different mobile responsive views. Instead of just writing this in your test code directly, move all of this data into its own file (commonly a JSON file).

```
{
  "iphone16": {
    "screenWidth": 1179,
    "screenHeight": 2556,
    },
  "s24Ultra": {
    "screenWidth": 1440,
    "screenHeight": 3120,
  },
}
```

With the data abstracted, your tests can add or remove screen resolutions as needed. You can even use Data-Driven Testing to iterate through all the screen resolutions.

Now this data can be used in multiple tests, and it is in a format that's easy to modify.

3. Keep Tests Simple and Readable

Keep Code Readable

Do you know what this Cypress code is doing?

```
describe('Hard-to-Read Cypress Test', () => {
  it('Should select and interact with multiple elements', () => {
    cy.visit('/home');
    cy.get('div.container > ul.list-items > li:nth-child(3) >
a[role="button"][data-active="true"]')
      .click();
    cy.get('form#userForm > div.input-group >
input[type="text"][placeholder="Enter your name"]:not([disabled]) +
span.icon')
      .click();
    cy.get('div.form-wrapper > button.btn-submit.primary:hover:focus')
      .should('be.visible')
      .click();
  });
});
```

The above code is just an example, but you get the idea. You're just seeing a bunch of "cy.get()" calls with long, unintelligible CSS selectors. You have to put thought and effort into figuring out what this code is doing. Imagine the CSS selectors changing a few months down the road, and now you have to figure this code out and update it. Not fun. And this is just a simple example.

Instead, consider this:

```
describe('Hard-to-Read Cypress Test', () => {
  it('Should select and interact with multiple elements', () => {
    cy.visit('/home');

    clickLinkToEditName();
    editName( "Francis" );
    submitEditedName();
  });
});
```

Now you don't need to become a detective to understand what the code is doing. The methods read like sentences. The test itself almost reads like a paragraph. Comments aren't even needed because the methods are named to clearly indicate what they are doing.

A Test Should Only Test One Thing

A test should only test one thing. If you need to test more than one thing, you should have multiple tests.

When you try to test too much in one test case, it becomes confusing. What is the test actually doing? When it fails, what exactly failed?

Even worse, the first thing in this overcrowded test case could fail. Then, the other tests inside that one test case won't get executed. Do they still work? Is it just the first thing that broke? Or was the second (maybe more important) thing that also broke?

That could be the difference between a minor bug that may not get fixed to a show stopper, everyone's head is on fire, type of bug.

Instead, only test one thing at a time in a test case. It's easier to understand. Easier to maintain. And very clear what is wrong if the test fails.

Of course, there's always exceptions to this rule. For instance, if the set up for the test is extremely complicated and time consuming, testing multiple things in one go is the most economical way to do it. But that should not be the normal case.

Avoid Complexity. Don't Show Off

Trying to show off your knowledge of advanced computer science algorithms in your test is fun for you, but makes for an impossible to maintain test.

Just because you can implement a shortest path algorithm to navigate all the links in your applications doesn't mean you should. You might be impressed with your code, but the next person to have to fix that code when it breaks is going to hate you so much.

Most likely, the person will just turn off the test because they can't make sense of it.

This also goes for code where you have for loops nested inside if statements nested inside more loops. I like to call this type of code "spaghetti code". To debug it, you have to unfurl all the strands of noodles to figure out what is going on.

If you can do something simple, do it simple. Your code is more stable and maintainable the simpler it is.

If you must have complexity, break the complexity up into well named methods. Each method should do a small part of the complex algorithm. Heavily comment what you are doing. Make sure it is very clear exactly what is going on. Keep your code readable.

4. Have A Plan For Flaky Tests

Why Flaky Tests Are a Serious Problem

Your tests need to be stable. You want them to reliably catch **real** issues, **real** bugs, and **real** regressions. Stable, reliable tests are essential for CI/CD (Continuous Integration/Continuous Delivery) pipelines because they **prevent buggy code from reaching customers**.

But if your tests fail for **no good reason**, and failures are inconsistent, they will lose **trust**. Developers and QA engineers will start ignoring test results because they assume failures aren't real.

Even worse, **flaky tests can hide real bugs.** You can become desensitized to a flaky test and stop investigating it, potentially missing a **critical defect**.

So, how do you **combat flakiness** and regain control of your test suite?

Step 1: Separate Flaky Tests from Stable Tests

One of the biggest problems with flaky tests is that they **contaminate** otherwise reliable test runs. If a developer's commit **fails a test run**, but it's due to a flaky test and not their actual changes, they'll lose confidence in the test results.

A good solution is to **separate** flaky tests into a different test suite.

How to Do This:

1. **Create Two Test Runs:**
    - **Stable Test Run:** These tests must always pass. If a test becomes flaky, **move it to the flaky test run** until it is fixed.
    - **Flaky Test Run:** This test suite will still run, but failures here **don't block** development.
2. **Monitor and Fix Flaky Tests Gradually:**
    - **As you stabilize flaky tests**, move them back to the stable test suite.
    - Over time, your flaky suite should **shrink** until it disappears entirely.
3. **Set Clear Policies for Flaky Tests:**
    - If a test **fails more than 2 times out of 10** with no real issue, classify it as flaky.
    - Do **not** allow flaky tests in CI/CD gating.
    - Engineers should **actively track** flaky tests and investigate root causes.

Step 2: Use Timeouts and Retries

One of the most common causes of test flakiness, especially in UI and web testing, is **elements taking too long to load**.

Timeouts can help by ensuring tests wait long enough for elements to appear before failing.

Setting Custom Timeouts

Most web test frameworks have a **default timeout of 4-5 seconds** before they throw an error when an element isn't found. But sometimes, elements take a little longer to load.

Here's how you increase the timeout in **Cypress**:

```
cy.get('.dynamic-button', { timeout: 10000 }) // Waits up to 10 seconds
  .should('be.visible')
  .click();
```

Other frameworks like **Selenium WebDriver** allow similar timeout customizations.

Global Timeout Configuration

If your test suite **regularly runs against slow environments**, you might need to **increase timeouts globally**.

Example in Cypress:

```
Cypress.config('defaultCommandTimeout', 10000);
```

Example in Selenium (Python):

```
driver.implicitly_wait(10)  # Waits up to 10 seconds for elements
```

⚠ **Be Careful with Timeouts**

- **Too long of a timeout** (e.g., 60+ seconds) can hide performance issues.
- Discuss expected load times with your **developers** and **UX team**.
- If a page is taking **too long** to load, **open a performance bug!**

Step 3: Implement Test Retries

Sometimes, a test fails **once** but passes on a retry. Instead of assuming it's flaky, try **rerunning** it automatically.

Many frameworks allow **automatic retries** for unstable tests.

Enable Retries in Cypress

```
describe('Retry Example', () => {
  it('Should retry failed assertions', { retries: 2 }, () => {
    cy.get('.randomly-loading-element')
      .should('be.visible');
  });
});
```

This will retry the test **2 times** before marking it as failed.

Enable Retries in Jest (for API tests)

```
test.retry(3)("fetches user data", async () => {
  const response = await fetchUserData();
  expect(response.status).toBe(200);
});
```

Best Practices for Retries

- Don't rely on retries **instead of fixing** the root problem.
- Only retry tests that **fail due to known environmental flakiness**.
- Set a **retry limit** (e.g., 2-3 retries max).

Step 4: Investigate and Fix Flakiness

Not all flaky tests are caused by slow loading times. Other root causes include:

1. Bad Selectors (Locators)

- Are your tests breaking because elements **change frequently**?
- **Fix:** Use stable identifiers like `data-testid` instead of fragile CSS selectors.
  ```
  cy.get('[data-testid="login-button"]').click();
  ```
    - `data-testid` requires you editing the source code yourself or requesting these identifiers to be added by the developers.

2. Environment Instability

- Do tests fail more in **certain environments** (e.g., staging vs. production)?
- **Fix:** Ensure your test environment has **consistent data, stable network connections**, and sufficient **resources**.

3. Concurrency Issues (Shared Test Data)

- Are multiple tests **modifying the same test user account** at the same time?
- **Fix**: Generate new test accounts or data for each test.
    - Ask the developers if there is an api to generate new accounts.

4. Network Failures

- Are API calls failing intermittently?
- **Fix:** Check the **network tab** in DevTools for `400` or `500` errors. These are most likely bugs.

Step 5: Collaborate With Developers

You **don’t have to fix flaky tests alone!** Work with **developers and DevOps engineers** to:

- Improve **test environment stability**
- Fix API response delays or inconsistencies.
- Optimize **database queries** that slow down tests.

Developers have **insights into the system’s internals** that can help pinpoint flakiness. Schedule **regular sync meetings** to discuss test stability.

Final Thoughts: A Roadmap for Stable Tests

✅ **Step 1**: Separate flaky tests from stable tests.
✅ **Step 2**: Use timeouts and dynamic waits to avoid premature failures.
✅ **Step 3**: Enable retries to re-run transient failures.
✅ **Step 4**: Investigate root causes—bad locators, slow environments, concurrency issues.
✅ **Step 5**: Work with developers to **eliminate** instability at the source.

By taking these steps, you **restore confidence in your automated tests** and make them a valuable asset in your CI/CD pipeline.

Next Steps

Now that you've tackled flaky tests, let's move on to the **final best practice**: **Keeping tests independent!** (See next section.)

5. Keep Tests Independent

Why Test Independence Matters

Your test suite should be **fast, reliable, and scalable**. One of the biggest obstacles to achieving this is **test dependencies**—where one test relies on another to run successfully. Every test should be able to run by itself without relying on side effects of the previous test.

Why is this a problem?

- If **one test fails**, it **cascades failures** into other dependent tests.
- **Parallel execution becomes impossible**—tests have to run in a specific order.
- **Debugging becomes harder**—failures may not be caused by the test itself but by previous tests.
- **Longer execution times**—since tests have to run sequentially instead of in parallel.

By keeping tests **isolated and independent**, you ensure that:

✅ Tests can run in **any order** (or in parallel).
✅ Failures are **easier to diagnose**.
✅ The test suite **scales efficiently** as more tests are added.

Best Practices for Keeping Tests Independent

Here's how to **design truly independent tests**:

- Avoid test dependencies
- Use unique test data for each test
- Run tests in parallel
- Avoid hardcoded test order

1. Avoid Test Dependencies

A test should **never rely on another test** to set up its data, state, or environment.

🚨 Bad Example: Dependent Tests

```
it('Creates a new user', () => {
    cy.visit('/signup');
    cy.get('#username').type('testuser');
    cy.get('#password').type('password123');
    cy.get('#submit').click();
});

it('Logs in with created user', () => {
    cy.visit('/login');
    cy.get('#username').type('testuser');
    cy.get('#password').type('password123');
    cy.get('#login-btn').click();
});
```

Problem:

- If the **first test fails**, the **second test automatically fails**.
- The second test **assumes** the user was successfully created, which might not be true.

✅ Better Approach: Self-Sufficient Tests

```
it('Logs in with a dynamically created user', () => {
    const uniqueUsername = `user_${Date.now()}`;

    // Sign up in the same test (self-contained)
    cy.visit('/signup');
    cy.get('#username').type(uniqueUsername);
    cy.get('#password').type('password123');
    cy.get('#submit').click();

    // Now login with the same credentials
    cy.visit('/login');
    cy.get('#username').type(uniqueUsername);
    cy.get('#password').type('password123');
    cy.get('#login-btn').click();
});
```

✅ **Each test sets up its own data.**
✅ **No risk of failures cascading between tests.**

2. Use Unique Test Data for Each Test

Many tests **fail inconsistently** because they **share the same test data**, leading to:

- **Concurrency issues** (e.g., one test modifies the shared data, breaking another test).
- **Data pollution** (e.g., tests leave data behind, affecting future runs).

🚨 **Bad Example: Reusing the Same Test Account**

```
cy.get('#username').type('existing_test_user');
cy.get('#password').type('password123');
cy.get('#login-btn').click();
```

Problem:

- If another test **changes the user's password**, this test **fails unexpectedly**.
- If the test **deletes the user**, future test runs **break**.

✅ **Better Approach: Generate Unique Data**

Use **dynamic data** to prevent conflicts:

```
const uniqueEmail = `user_${Date.now()}@test.com`;

cy.get('#email').type(uniqueEmail);
cy.get('#password').type('password123');
cy.get('#signup-btn').click();
```

✅ **Each test has its own fresh data, avoiding conflicts.**

3. Run Tests in Parallel

Test execution time **increases as more tests are added**. To scale efficiently, tests should be **able to run in parallel**.

Parallel Execution Benefits:

- **Faster execution** by running multiple tests simultaneously.
- **Avoid bottlenecks** from long-running sequential tests.

🚨 **Bad Example: Tests That Can't Run in Parallel**

```
it('Modifies a shared test account', () => {
    cy.visit('/profile');
    cy.get('#username').clear().type('new_username');
    cy.get('#save-btn').click();
});
```

Problem:

- If multiple tests run in parallel, **they might overwrite each other's changes**.
- Test execution order **must be controlled**, slowing down the suite.

✅ **Better Approach: Parallel-Safe Tests**

- Use **unique test data per test**.
- Use **temporary test accounts** that are created and deleted within the test.
- Isolate tests **from shared state**.

4. Avoid Hardcoded Test Order

Test runners **execute tests in parallel** or **randomize execution order**. If your tests **must run in a specific order**, they are **not independent**.

🚨 **Bad Example: Enforcing Test Order**

```
describe('Sequential Tests', () => {
    it('Step 1: Create User', () => { ... });
    it('Step 2: Update User Profile', () => { ... });
    it('Step 3: Delete User', () => { ... });
});
```

Problem:

- If **Step 1 fails**, every subsequent test **also fails**.
- Parallel execution **is impossible**.

✅ **Better Approach: Independent Tests**

```
it('Can create a user', () => { ... });
it('Can update user profile with a new test user', () => { ... });
it('Can delete a unique test user', () => { ... });
```

✅ **Each test creates and cleans up its own data.**

Final Thoughts: The Roadmap to Independent Tests

✅ **Step 1:** Avoid test dependencies—each test should stand alone.
✅ **Step 2:** Use **unique, dynamically generated test data** to prevent conflicts.
✅ **Step 3:** Ensure tests **can run in parallel** by avoiding shared state.
✅ **Step 4:** Remove **hardcoded test order**—tests should work in any sequence.

By following these best practices, you **increase test reliability, scalability, and execution speed**. 🚀

Conclusion: Putting It All Together

The journey to effective test automation is a continuous one. Take the time to assess your current testing strategies, identify areas for improvement, and implement changes incrementally. Encourage your team to embrace these best practices and continuously refine them to enhance test efficiency and reliability. These best practices provide a strong foundation, but remember that improvement is always ongoing.

- Start **small** and **iterate**—begin by implementing one or two of these practices and refine them as your team gains experience.
- Collaborate with **developers, product managers, and DevOps teams** to seamlessly integrate automation into the development lifecycle.
- **Trust your tests**—your automation suite should act as a safety net, catching regressions and ensuring product quality with every release.
- **Measure success**—track key metrics such as test execution time, failure rates, and flakiness to assess the effectiveness of your automation strategy.
- **Keep learning**—stay updated on the latest tools, frameworks, and industry trends to continuously enhance your testing process.

By following these principles, you will not only improve the reliability of your test automation but also contribute to a more efficient and confident software development process.

Recommended Reading

If you purchase any of these books with the links provided, I will receive a small part of the proceeds at no cost to you. Thank you for helping!

Clean Code

This book fundamentally changed the way I write code. To be honest, some of the chapters are very Java heavy and specific. But the first few chapters are revolutionary in writing understandable and maintainable code.

Refactoring

This is the companion to Clean Code. Even though it's about "refactoring", this book is basically like Clean Code showing you how to write maintainable, readable code. This book is much more approachable than Clean Code. The examples are easily applied to any of your coding projects.

The Pragmatic Programmer

A classic book on writing code. Though not test specific, the book expounds on the importance of testing and how testing is used as the basis for all code development.

Modern Software Engineering

I listened to the audiobook version, and fell in love with the in-depth, but easily graspable, concepts of modern software engineering. There is quite a bit in there about testing which surprised me. When you see your company doing something wrong with the software development process, this book gives you great ammunition and principles to bring to your meetings.

BDD In Action

BDD stands for Behavior Driven Development. Even though most companies don't use this type of development, I believe it is the way of achieving highest quality software. BDD is a test-first approach to software development that is highly coupled with the business goals for the software. Trying to incorporate BDD into my testing has forever changed how I approach testing…even if the development team isn't ready for it.