Lesson Monday

Intermediate JavaScript (/intermediate-javascript) / Test-Driven Development and Environments with JavaScript (/intermediate-javascript/test-driven-development-and-environments-with-javascript) / Testing Best Practices

Text

In this lesson, we'll cover some testing best practices — as well as go into a deeper dive regarding the differences between good and bad fails. Writing good fails instead of bad ones can really trip students up at first. And the implications of bad fails can be significant — extra time trying to find bugs, frustration, and less understanding about what's going wrong in the code. The implications get even bigger once we are out in the real world — for instance, if we don't test our code correctly, we might introduce breaking changes to production code, making our customers, coworkers, and employers upset. If possible, we don't want to do that.

Unfortunately, in the real world, there are many companies that don't test their code, though it's not due to anyone arguing that untested code is better. Instead, tight deadlines, small staff, tight budgets, poor management practices, legacy code, and many other complications can lead to code that isn't fully tested, or tested at all.

At Epicodus, we're focused on teaching best practices for testing. Ideally, you'll eventually work at a company that has good testing practices. Even if the company doesn't have good practices, you can bring your newfound knowledge to the company and make a difference immediately. In fact, it's very common for junior developers to start learning a codebase by writing tests first. Adding testing to a codebase is something you can often start doing right away — even if you don't fully understand the code you're working with. And doing so can add tremendous value to your company.

Good Testing Practices — and Failing Right

Here are some testing best practices — as well as steps to make sure you get good fails — and in turn, good passes.

- **Write tests for distinct behaviors.** Often a function just needs one test, because it does one thing. However, if our function does multiple things, we should consider having a test for each behavior. This improves how easy to understand our code and tests are. It's OK to have multiple expectations for a single test. However, if a test has many expectations it can be a red flag that the test may cover multiple distinct behaviors and may need to be broken into multiple tests to improve comprehension.
- **Use `Describe` blocks to improve comprehension of your tests.** You can use `Describe` blocks for entire object types or for individual methods and functions, depending on how complicated each is. Similar to writing tests for distinct behaviors to clearly communicate the functionality of your code, `Describe` blocks are meant to improve organization and comprehension of your test. If a class is more complex, you may want to use a `Describe` block for each method instead of just the class.

- **Write the test first, not the code.** We've already covered many of the reasons why we should write our tests before we write any code. This is the cornerstone of TDD because it's *test*-driven, not *code*-driven. We won't reiterate all the reasons we should write tests first as we've covered that elsewhere.
- **Write just enough of a function or method to get the code to fail.** We can't just write a test, run it, and move on. That would be a bad fail. If we are testing a function, for instance, we need to at least add the `function` keyword and the name of the function. Our test should at least return how our expectation wasn't met — such as by stating the expected result and returning the actual result (such as `undefined` because we haven't written a function body yet).
- **Keep the code in your test to a minimum.** In your tests, you should only write code that's required to run the piece of business logic that you are testing for. This usually involves invoking the business logic function you are testing and not much more. Adding more code than that can create problems by introducing bugs that are unrelated to our business logic. We want to isolate problems in our code, not create more problems by adding unnecessary code in our tests.
- **Read the Jest output for failing tests.** It's easy to just run a test, see that it's red, and assume that it's a good fail. It's tempting to be in a rush to write the code, especially if we're excited about it or have an idea about how to implement it. But just because it's red doesn't mean it's a good fail.
- **Always fix bad fails before moving onto the code.** TDD means having a good fail before writing the code to pass the test. It doesn't mean writing a test, having a bad fail, and then writing some code. That is actually a recipe for disaster. We'll often see students looking for bugs in the wrong places when this happens — or just being utterly confused because their tests aren't pointing them in the right direction.
- **Always commit your code after each passing test.** This is part of having a strong commit history. Also, if you break your code and can't get it working again, you can always return to a

commit where all tests are passing.

Next, let's look at some examples of bad fails.

Bad Fails

Let's cover some examples of bad fails based on the following tests:

```
import Example from '../src/example.js';

describe('Example', () => {

  test('should correctly demonstrate bad fails', () => {
    let example = new Example();
    expect(example.data).toEqual("Bad fail!");
  });

  test('should correctly demonstrate bad fails', () => {
    let example = new Example();
    expect(example.exampleFunction()).toEqual("This functio
n returns a bad fail!");
  });
});
```

The test won't run because it can't find a file. Let's say that when we run the test suite above, we get the following error: `Cannot find module '../src/example.js' from 'example.test.js'.`

This means that the file doesn't exist *or* there's an error in the path *or* the file exists but there is an error in the file name.

This is a bad fail. Our test should always be able to correctly find the file it is looking for. We're not even correctly testing any code yet. All we've confirmed is that our tests can't find a file.

The test won't run because it can't find a function or constructor. So we realize that we don't have a `src/example.js` file yet — or that it's named `src/exmple.js` — or even that for some reason it's not in the `src` directory. We fix that issue and run the test, only to get the following error:

```
TypeError: _example.default is not a constructor
```

This means that one of three things could be happening:

- We haven't added a constructor for `Example` yet.
- There's a typo either in our test (such as `new Exmple()`).
- There's a typo in our constructor.

In the case of a missing function, we'll get an error like this:

```
TypeError: example.exampleFunction is not a function
```

These are both bad fails. Any functions or constructors that a test uses at the very least need to exist so that the test can properly run or test the constructor.

The test won't run because there's an error that breaks all the tests in a suite. An error in a test file can blow up the whole suite — or it can cause an individual test to fail.

Here's an example that will blow up the entire suite:

```
Test suite failed to run

    SyntaxError: /Users/staff/Desktop/test_env/__tests__/ex
ample.test.js: Unexpected token, expected ";" (1:6)

    > 1 | impor Example from '../src/example.js';
        |       ^
```

Can you see the error? There's a typo in the import statement — it should be `import`, not `impor`. Any time you see `Test suite failed to run` for any reason, it's a bad fail. It means the entire test file isn't able to run. Don't you dare start writing your code yet if you're in this situation! Get the tests working first.

The test won't run because there's an error in an individual test. An error can also break just one test as well. Here's an example:

```
should correctly demonstrate bad fails

    ReferenceError: example is not defined

       9 |
      10 |   test('should correctly demonstrate bad fails',
() => {
    > 11 |     expect(example.exampleFunction()).toEqual("T
his function returns a bad fail!");
         |            ^
      12 |   });
```

Here we get the error `ReferenceError: example is not defined`. In this case, it can be tempting to think this error is happening because we haven't defined something in our source code. Take a closer look at the error, though. In reality, it's because the line `let example = new Example();` has been removed from the test — and there's no variable named `example` in our test's scope.

There are a million and one different errors that can occur in a test — but all of them are bad fails. The test needs to be running correctly and be error-free or it will be a bad fail. Once again, make sure you get this working correctly before writing any code.

The test fails because we wrote bogus data in our test. Let's say we've got all the code we need in our constructor to get the first test passing:

```
export default function Example() {
  this.data = "Bad fail!";
}
```

However, for some reason we want to make sure that we get a fail before our test passes... perhaps because we did things out of order and wrote our code before we wrote the test. So we update our test to do the following:

```
  test('should correctly demonstrate bad fails', () => {
    let example = new Example();
    expect(example.data).toEqual("I'm sneaky! I'm gonna pre
tend this is a good fail!");
  });
```

```
expect(received).toEqual(expected) // deep equality

Expected: "I'm sneaky! I'm gonna pretend this is a good fai
l!"
Received: "Bad fail!"
```

This is what is known as a test-based fail, not a code-based fail. The problem here is that we aren't actually testing to see if our code is working correctly — if we were, our expect statement would state

```
expect(example.data).toEqual("Bad fail!");
```

With `"Bad fail!"` to match the code in the constructor.

However, by changing the expect statement to match with `"I'm sneaky! I'm gonna pretend this is a good fail!"`, we're trying to manufacture a fail, instead of letting one happen naturally.

The tests we write always need to be a source of truth. We always need to write a test so that it passes. If we change the code in our test to fail it, we could start getting confused, and we might run into false positives (or negatives). In other words, if we've inserted errors into the test, it's no longer a source of truth, and it's no longer reliable for testing our code.

The test fails (or even passes) because we put too much code in our test. Let's say we want to test that our constructor automatically adds a `data` property of `"Bad fail!"` to an instance of `Example`. This would be a truly horrible test:

```
// This is hideous.

  test('should instantiate an Example with a data property
of Bad fail!', () => {
    let example = new Example();
    const data1 = "Bad "
    const data2 = "fail"
    const data3 = "!"
    const newData = data1.concat(data2).concat(data3);
    example.data = newData;
    expect(example.data).toEqual("Bad fail!");
  });
```

This test is actually going to pass if we at least correctly instantiate an `Example` object. But it's such a bad pass that it's truly a bad fail. What happens here is that we've written code in the test to get the test to artificially pass. But we're supposed to be testing our source code, and we're not actually testing that when an `Example` is instantiated, a `data` property with the right value is being created. Instead, we're writing the code manually in the test.

If we were quality control elves in Santa's lab and our job was to inspect little toy cars being churned out of a machine to make sure they are painted red, we can't just take a car that's painted blue, paint it red, and say the machine is working correctly. In this analogy, our code is the machine. Always test to make sure the machine works. Don't alter the results (the test) to fit what you're hoping for.

And here's the worst fail of all:

```
Test suite failed to run

Your test suite must contain at least one test.
```

This means your suite doesn't even have any tests in it yet. Time to start adding some tests!

Summary

Before we move on, let's reiterate an example of a good fail. We can do the following to ensure a good fail:

- Make sure the file is being accessed (no test errors).
- Make sure the function is being called.
- It's fine if the function returns `undefined` — don't write bogus code in either the function or your test to turn a good fail into a bad one.

Here's the error message on a good fail:

```
expect(received).toEqual(expected) // deep equality

Expected: "This function returns a good fail!"
Received: undefined
```

Here, we can see that the function was correctly called, which implies that our test was able to find the file as well. The function returns `undefined` because we haven't added any code to the function body yet.

At this point, you should have a clear sense of the difference between good and bad fails. Ensuring your test has a good fail first is an important part of the process, and doing so can help you isolate bugs in your code. In general, these testing practices apply no matter what language you are writing in — with a few small modifications depending on the language.

From here on out, you should be applying these best practices as best as you can whether you are writing tests in JavaScript, Ruby, C#, or another language.

Previous (/intermediate-javascript/test-driven-development-and-environments-with-javascript/tdd-with-jest-testing-the-triangle-prototype-checktype-method)
Next (/intermediate-javascript/test-driven-development-and-environments-with-javascript/expanding-our-testing-tools-adding-setup-and-teardown)

disable dark mode

(http://www.epicodus.com)

© 2023 Epicodus (http://www.epicodus.com/), Inc.