Introduction to Testing

A Frontend Masters course.

Steve Kinney, @stevekinney

Writing tests isn't hard.

But it's easy write code that's hard to test.

We're going to look at how to write a test or two.

But, we're also going to look at how to write code that's easy—or at least easier—to test.

And, we'll look at how to test code that's hard-to-test.

What are the prerequisites?

Do you JavaScript? Then, we're good.

I'm not going to assume you've written a single test before in your life.

We'll test some components that happen to be written in a framework like React or Svelte. But, I don't expect that you'll have any familiarity with any of these.

As long as you know some JavaScript, have Node installed, and get around the command line—we should be good.

```ts
import { it, expect } from 'vitest';

it('is a super simple test', () => {
  expect(true).toBe(true);
});
```

```
it('is a super simple test', () => {
  expect(add(2,2)).toBe(4);
});
```

There are a few stages in everyone’s testing journey.

Testing? I've never heard of it.

I've heard of testing.

But, I don't know how to do it and this makes me feel *bad*.

I’m doing it? I think?

But, I still have bugs and my pager is still going off at night.

Stage Four: The Danger Zone

I’m going to test all the things. 100% of the way. And I’m going to hassle everyone that they’re not writing enough tests and I’m never going to approve a PR ever again and I heard that Jeff Bezos said that the build shouldn’t pass if the test coverage drops and I’d like to argue you about the difference between unit, integration, and end-to-end tests. Do you even know what it means to be idempotent? Who cares is all of my code is unreadable? Did I tell you I’ve writing my own functional programming language?

I have a healthy relationship with testing.

I write tests to give myself confidence that my code is working as expected. I don’t take it too far.

The end goal of testing.

Why even bother?

Spoiler: It’s not 100% test coverage.

It’s about getting rid of that feeling of existential dread when go to refactor your code.

Someone is always testing your code.

Hopefully, it's you.

It's either you or it's your users.

And if it's you, then you're either doing it manually every time you make a change—or, you have an automated system in place.

My job is to help you get that automated system in place.

A bias towards action.

We're going to learn about testing by writing tests—not talking about it.

A lot of the content around testing has a tendency to get really philosophical.

We're going to focus on learning how to write tests by writing tests.

And, we'll touch upon all of that other stuff as we go along.

What are we going to cover today?

Let’s take a look at how I’m thinking we should spend our time together.

We’re going to write some super simple tests.

Suspiciously simple, to be clear.

We’re going start at the very basics.

But. don’t worry—things will escalate from there.

We’re going to handle edge cases & errors.

Other people’s code breaks—not mine.

It would be great if we only hard to worry about when things go exactly as planned.

But, they don’t always do that and we need to be prepared for that.

We’re going to look at some little tricks.

Write better tests with this one weird trick.

Are you going to use these tricks every day? Probably not.

Do I expect you to memorize them? I don’t.

I just want you to be aware that they exist if you need them.

We're going to look at testing the DOM.

An unfortunate reality of being a front-end engineer.

I've been told that a lot of JavaScript developers work on UIs.

I've been told that browser's have this thing called the DOM.

We should probably test that.

How to deal with stuff you don’t control.

Which is more things than I’d like, frankly.

I’ve also heard that some of these web applications make network requests.

Did you know you can install packages from npm?

How do we test this kind of stuff?

We’re going to look at browser-based tests.

This is probably a topic in it’s own right.

When all us fails, we can just have our tests grab ahold of a browser.

Let’s look at how to truly test from the user’s perspective.

There’s course website.

https://stevekinney.net/courses/testing

There is also a repository of examples.

https://github.com/stevekinney/introduction-to-testing

Tools of the trade.

What are some of the tooling out in the world?

We’re going to use a test runner called Vitest.

But, it doesn’t matter. They’re all pretty much the same.

It’s what I use on a daily basis and you probably want me using the tools that I’m most comfortable with for the next few hours.

Example

examples/scratchpad

A basic test.

It’s good and you should do it when it’s appropriate.

I don’t do it 100% of the time and I don’t trust anyone who says they do.

It’s not always easy to start with tests.

Don’t let anyone make you feel bad about this.

Sometimes, you need to mess around and find out before you settle on your final approach.

And, this tends to lead you writing some tests after the fact

Example

examples/arithmetic

Basic addition.

Exercise

examples/arithmetic

Subtraction, multiplication, and division.

On the topic of test-driven development.

It's hard to write code that's hard to test if you start with the tests and make them pass.

And now…

Steve's Rules of Testing.

They're more like guidelines.

Writing tests isn't hard. But, some code is hard to test.

Your tests don't pass because your code works. They pass because they didn't fail.

Someone is always testing your code.

No one has ever broken their code into too many, small, well-named, easy-to-test functions.

Testing Terminology

The types of tests you'll meet.

Unit tests.

Take one function or object and test it in isolation.

This is where we're going to spend a lot of our time today.

Integration tests.

Take two of more things and test how they work when interacting with each other.

End-to-End tests.

Tests all of the things. Usually this involves driving a browser of some sort.

The Unhappy Path™

The Unhappy Path.

Or, the importance of pessimism.

Sure passing some math equations some numbers does what we expect.

But, what about all of the other weird stuff that can get in there?

What about undefined? How about a string?

Testing the unhappy path is about making sure that you've thought through all of the stuff that can get weird.

```
true + true === 2
1 + '1' === '11'
```

```
NaN !== NaN
```

```
add(1);
add(null, 1);
add('1', 2);
add(2, 'potato');
subtract('1', 1);
divide(5, 0);
```

What to do when things go wrong.

Three outcomes; only one wrong answer.

Fail gracefully.

Flip a table and throw an error.

Let things play out as they will. 🤷

Example

examples/utility-belt

Convert string to number.

Exercise

examples/arithmetic

Mathematical edge cases.

All Things Being Equal

Referential equality.

Just because they’re basically the same, it doesn’t mean they’re actually the same.

Sure, `1 === 1` and `‘string’ === ‘string’`.

But, `{ foo: 1 } !== { foo: 1 }` and `[1,2,3] !== [1,2,3]`.

This is where `toBe`, `toEqual`, and `toStrictEqual` all come in.

Example

examples/strictly-speaking

Strictly speaking.

toBe versus toEqual.

The same in memory versus effectively the same.

`toBe` is useful for comparing primitive values that would `===` each other.

`toEqual` looks at the contents of an object or array to see if the values are equal to each other.

toEqual versus toStrictEqual.

How equal is equal?

`toEqual` checks if two objects or arrays have the same values and structure, allowing for loosely defined properties (e.g., `undefined` properties are not strictly compared).

`toStrictEqual` ensures a more precise match, where even undefined properties, types, and object prototypes must exactly match.

```
it(‘has what we are re looking for', () ⇒ {
  expect(new Person('Alice')).toEqual({ name: 'Alice' });
});
```

```
test('strictly equal or no good’, () ⇒ {
  expect(new Person('Alice')).not.toStrictEqual({ name: 'Alice' });
});
```

Example

examples/characters

Asymmetric Matchers.

Exercise

examples/characters

Characters.

Before and After

Doing stuff before and after each test.

Not to be confused with those other kinds of hooks.

If you realize that you're doing the same thing before and after every test or you want to do something before all of the tests and then clean up after all of the tests are done, then you can use hooks.

But, be warned: They're convenient, but convenience sometimes comes at the cost of clarity.

Example

examples/arithmetic

Counter.

Example

example/characters

Refactoring the Person tests.

Testing Asynchronous Code

Testing asynchronous code.

It used to be tricky; now, it's not.

This use to be a bit more of a pain.

But, basically, if you remember to use `async` and `await`, you should be mostly good.

But, yea—you have to remember to use `async` and `await`.

Testing the DOM

Testing the DOM.

The potential problem.

Your tests run in Node.

Node isn’t a browser.

This means it doesn’t have any of the Browser APIs.

The DOM is one of those APIs.

This means that Node doesn’t have the DOM.

This means you can’t test the DOM from Node.

We’re going to do it anyway.

Using a DOM library.

Simulate the DOM to get around this problem.

Out of the box, Vitest supports two DOM libraries: **JSDOM** and **Happy DOM**.

HappyDOM is small and lightweight.

JSDOM is an industry standard, but it’s a heavier tool.

It probably doesn’t matter which one you pick.

Some caveats.

There is no such thing as a free lunch.

- **It’s still not a real browser**. You're not getting every subtlety of a specific Chrome, Safari, or Firefox version. It's designed to *act* like a browser.
- Running tests with `jsdom` can be **a bit slower**. It’s the cost of emulating browser stuff.
- You might still run into **browser-specific issues**. Just because something works in Browser Mode doesn’t mean it’ll work in *all* browsers. (I’m looking at you, Safari.)

Setting Up the Environment

Just tell Vite that you want to use a DOM library.

It’s as easy as tweaking one small configuration.

You can also do it on a per-file basis.

```ts
export default defineConfig({
  test: {
    globals: true,
    environment: 'happy-dom',
    setupFiles: ['@testing-library/jest-dom/vitest'],
  },
});
```

Example

examples/element-factory

Testing a button.

Exercise

examples/element-factory

Testing a login form. Part I.

Exercise

examples/element-factory

Testing local storage.

Querying.

It’s kind of like jQuery of `document.querySelector`, but you’re trying to do it from the perspective an assistant device.

This is basically a way to trick yourself into writing accessible components.

Example

Testing Library can extend the built-in matchers in order to make your life easier.
You can either import these on a per-file basis—or just do it globally with a setup file.

Switching tabs.

Exercise

Counting accidents.

Test Doubles

Faking it.

Mocking and spying.

If a unit test is supposed to where we test something in isolation?

Then how do we make sure it’s actually isolated?

Sometimes you can’t control everything.

Sometimes you want to test that a built-in function was called with the correct arguments.

Sometimes you don’t want randomness to break your tests… randomly.

Sometimes you'd prefer not to make actual network requests to your server.

Sometimes your server isn’t even running.

Sometimes, you're working with time and the times—they are *a*-changing.

Test doubles.

Secret agents for your tests.

Mocks, spies, and stubs.

These are fake methods and values that you can use so that you can pin down the thing you’re actually trying to test.

Putting Things Back

Leave no trace.

- **Clear**: You've created some complex mock logic, and now you're retracing steps, clearing call history to test cleanly.
- **Reset**: You made a mess with return values or `.mockImplementation`—and now you just want to start over without rebuilding the mock.
- **Restore**: You're done mocking, you want to reinstate the original functionality, and walk away like nothing ever happened.

Putting Things Back

Leave no trace.

- `fn.mockClear()`: Clears out all of the information about how it was called and what it returned. This is effectively the same as setting `fn.mock.calls` and `fn.mock.results` back to empty arrays.

- `fn.mockReset()`: In addition to doing what `fn.mockClear()`, this method replaces the inner implementation with an empty function.

- `fn.mockRestore()`: In addition to doing what `fn.mockReset()` does, it replaces the implementation with the original functions.

Putting Things Back

Leave no trace—in bulk.

- `vi.clearAllMocks`: Clears out the history of calls and return values on the spies, but does *not* reset them to their default implementation. This is effectively the same as calling `.mockClear()` on each and every spy.

- `vi.resetAllMocks`: Calls `.mockReset()` on all the spies. It will replace any mock implementations with an empty function.

- `vi.restoreAllMocks`: Calls `.mockRestore()` on each and every mock. This one returns the world to it's original state.

Example

examples/scratchpad

Spying on console.log.

Exercise

examples/element-factory

Spying on alert.

Exercise

examples/guessing-game

Preventing randomness.

Exercise

examples/element-factory

Revisiting that login form.

Spying on and mocking existing functions.

Mocking environment variables.

Example

examples/log-jam

Let’s look at a function that should only log in development.

Exercise

examples/log-jam

Enforcing an API key.

Mocking dependencies.

You've got to draw the line somewhere.

You don't want to or need to test other people's code.

Sometimes that code has side effects.

It might make network requests.

It might read or write to the file system.

You don't want to have to deal with any of that—nor should you.

Example

examples/element-factory

Mocking `sendToServer` in Log Jam.

Exercise

examples/element-factory

Mocking data fetching.

Time traveling.

Example

examples/scratch-pad

Stop and start time.

Mock Service Worker

Just mock out the whole network stack already.

Let's look at using Mock Service Worker.

Exercise

examples/directory

Mock out a request with Mock Service Worker.

Dependency Injection

Too much mocking? Try dependency injection!

Just pass in the things you need.

This is one of those areas, where just refactoring your code can make it easier to test.

If your code relies on functionality that you pass in, then it’s a lot easier to pass in things that suit your purpose.

I’ve never seen breaking your code into too many small, well-named, easy-to-test functions.

Example

examples/directory

Dependency injection.

Testing from the browser perspective.

Example

examples/task-list

Testing our task list.

Exercise

examples/accident-counter

Testing our accident counter.

Avenues for Further Study

Onwards…

There is still more to learn.

So, we also have this course called **Enterprise UI Development**.

We go deeper into some of the topics we covered today.

We also cover.

Running up you tests with a pre-commit hook.

Running your tests in a continuous integration environment with Github Actions.

Setting up a code coverage tooling.