Tip top JavaScript Testing

@Jack_Franklin

How to write good tests

Tests are only as useful as the *effort you put into them*.

It's important your application code is well written and maintainable.

You don't write *tests for your tests*.

So your test code should be 👌

Unless you write tests for your tests

But of course then you need tests for your tests for your tests

What makes a great unit test?

Browse — Thread

Thread Online Ltd. [GB] | https://www.thread.com/items/browse/new_in?sorting=recommended&search_id=d0572d44-7edb-439e-930d-50f06b428810

Superuser menu ... | Thread Drive - Go... | Carbon | MEGA Help | How to Install Zsh... | Email Templates –... | 2-Step Verification | JACK FRANKLIN L... | Other Bookmarks

Ideas 15 Shop

1 10

ALL ITEMS SALE

New in | Everything | Jack's Essentials | Coats & Jackets | Blazers | Suits | Knitwear & Sweaters | Shirts | T-shirts & Polos | Jeans | Trousers & Chinos | Shorts | Shoes | Accessories | Socks & Underwear

FILTER: Brands Colours Price

SORT BY: Personalised to me

Pier One
Smart Lace Shoes
£54.99
EU 46 / UK 12

Levi's
511® Slim Fit Jeans
£70
W34 L34

Barbour
Liddesdale Quilted Jacket
£99.95
M

Superdry
Military Parka Jacket
£99.99
M

```
it('clearly says what is being tested', () => {
  // 1. Setup

  // 2. Invoke the code under test

  // 3. Assert on the results of step 2.
})
```

```
describe('finding items in a price range', () => {
  it('returns the right set of items', () => {
    const dummyItems = [{ name: 'shirt', price: 2000 }, ...]

    const result = itemFinder(dummyItems).min(1000).max(5000)

    expect(result).toEqual(...)
  })
})
```

You should be able to look at a single `it` test and know everything

Tests should have no external dependencies

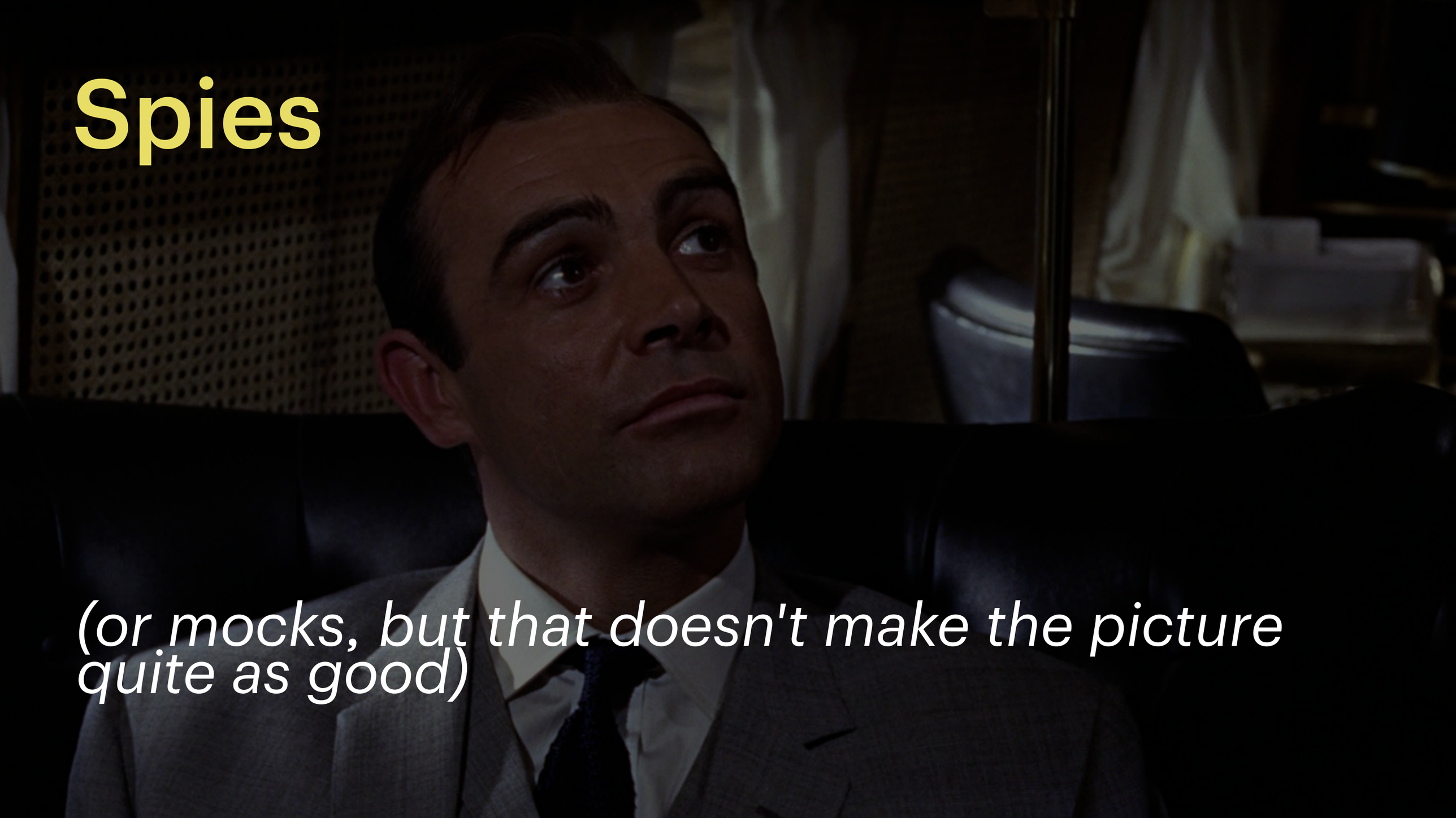
Keeping *unit* tests as a *unit*

If your tests can fail without *any of the code you're testing changing*, **your test is not properly** *isolated*.

This is a *huge cause of confusion and frustration* **in large code bases.**

🤷‍♀️ ***I changed*** `user.js` ***but tests in*** `blog_posts.js` ***broke*** ‼️

Spies



(or mocks, but that doesn't make the picture quite as good)

Mocking

Fake a function's implementation for the purpose of a test.

thread-events

Logging a user's actions across the site:

1. User `123` clicked on `home_feed`
2. User `123` added item `456` to the cart
3. User `123` added a new shipping address

```
import threadEventsLogger from 'thread-events-logger'

const processUserClickOnItem = (item) => {
  threadEventsLogger.log('item_click', { item_id: item.id })

  // does other stuff here too
}
```

threadEventsLogger: **external dependency!**

```
describe('when the user clicks on the item', () => {
  it('logs a thread-event', () => {
    // what goes here?
  })
})
}
```

Option 1:

```
processUserClickOnItem({ id: 123 })
expect(frontendEventsLog[frontendEventsLog.length - 1]).toEqual({
  type: 'item_click',
  data: { item_id: item.id },
})
```

Option 2:

```
processUserClickOnItem({ id: 123 })
expect(threadEventsLogger.log).toHaveBeenCalledWith('item_click', {
  item_id: 123,
})
```

Option 2 relies on `threadEventsLogger` being thoroughly unit tested itself.

don't test things twice

Avoiding awkward browser interactions in tests with mocks

```
const redirectUser = user => {
  if (user.authenticated) {
    window.location.assign('/home')
  } else if (...) {
    ...
  } else {
    ...
  }
}
```

```
test('if the user is logged in they are taken to home', () => {
  jest.spyOn(window.location, 'assign').mockImplementation(() => {});

  redirectUser({ authenticated: true })

  expect(window.location.assign).toHaveBeenCalledWith('/home')
})
```


Tidying up after yourself

Mocks won't be automatically cleared between tests

```
test('if the user is logged in they are taken to home', () => {
  jest.spyOn(window.location, 'assign').mockImplementation(() => {})
  redirectUser({ authenticated: true })
  expect(window.location.assign).toHaveBeenCalledWith('/home')
})

test('if the user is not logged out we do not redirect them', () => {
  redirectUser({ authenticated: false })
  expect(window.location.assign).not.toHaveBeenCalled()
})
```

!! the second test is going to fail!

```
beforeEach(() => {
  jest.clearAllMocks()
})
```


Mocks are an essential tool in a developers' testing toolkit

beforeEach

`beforeEach` is a great way to run code before each test

But it can make a test hard to work with or debug.

```
it('filters the items to only shirts', () => {
  const result = filterItems(items, 'shirts')
  expect(result).toEqual(...)
})
```

Where is `items` coming from?

```
let items
beforeEach(() => {
  items = [{ name: 'shirt', ... }, ... ]
})
```

3 parts to a good test

```
it('filters the items to only shirts', () => {
  const items = [{ name: 'shirt', ... }, ... ]

  const result = filterItems(items, 'shirts')

  expect(result).toEqual(...)
})
```

```
it('...', () => {
  const items = [{ name: 'shirt', ... }, ... ]
  ...
})
it('...', () => {
  const items = [{ name: 'shirt', ... }, ... ]
  ...
})
it('...', () => {
  const items = [{ name: 'shirt', ... }, ... ]
  ...
})
it('...', () => {
  const items = [{ name: 'shirt', ... }, ... ]
  ...
})
it('...', () => {
  const items = [{ name: 'shirt', ... }, ... ]
  ...
})
it('...', () => {
  const items = [{ name: 'shirt', ... }, ... ]
  ...
})
it('...', () => {
  const items = [{ name: 'shirt', ... }, ... ]
  ...
})
```

Functions for test data

```
const getItems = () => [...]

it('...', () => {
  const items = getItems()
  ...
})
```

👍 creation of test data is done for each test

👍 if the data has to change, we have one place to change it

Having consistent test data

It's important that test data resembles your real data.

You'll have a few domain objects that turn up in lots of tests.

At Thread we have the `Item` object.

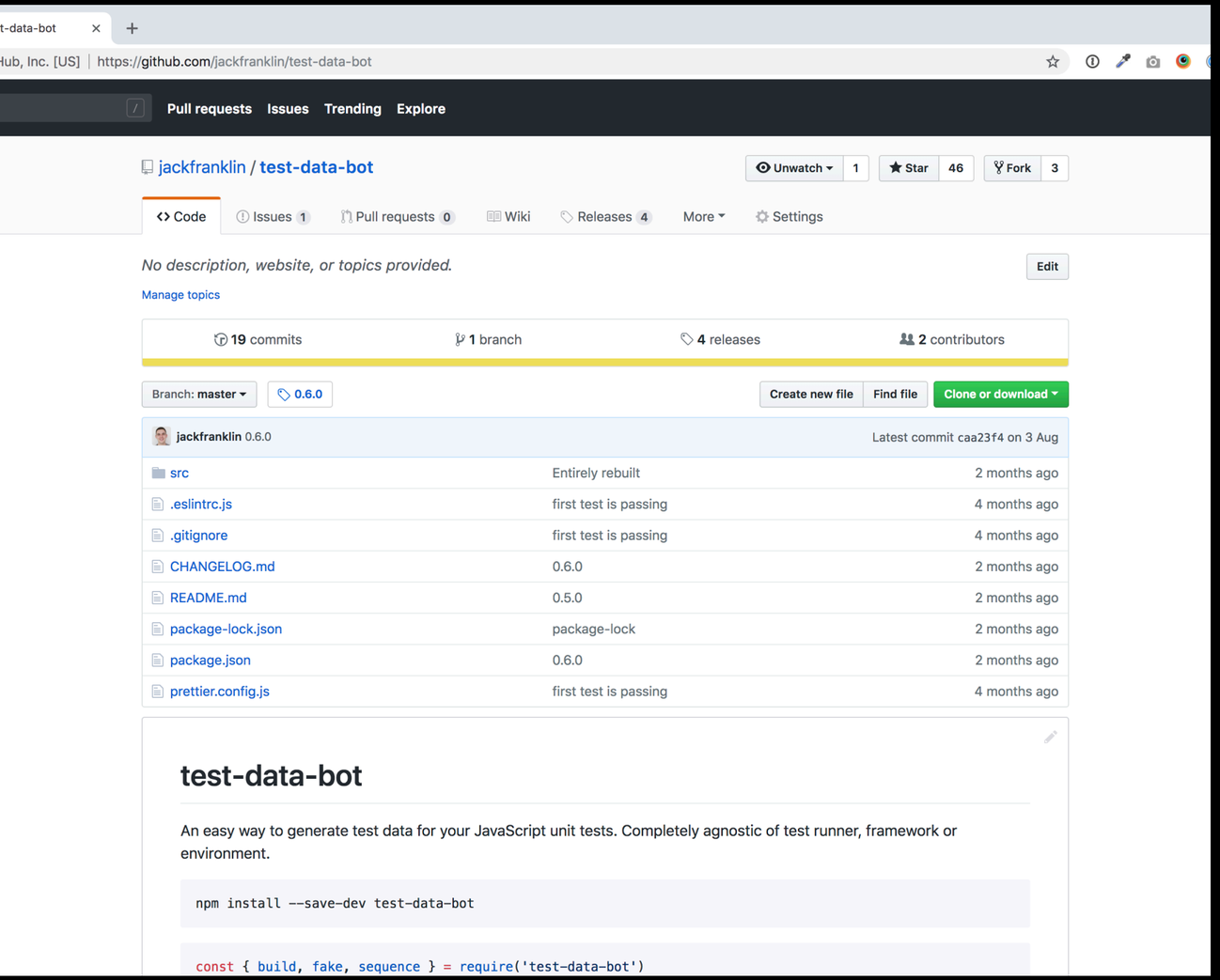
in every test...

```
const dummyItem = {...}
```

Then, one day:

🤓 ***All items returned from our API have a new property: 'buttonType'***

Now you have lots of outdated tests.



We can solve this with *factories*.

https://github.com/jackfranklin/test-data-bot

```
export const itemBuilder = build('Item').fields({
  brand: fake(f => f.company.companyName()),
  colour: fake(f => f.commerce.color()),
  images: {
    medium: arrayOf(fake(f => f.image.imageUrl()), 3),
  },
  is_thread_own_brand: bool(),
  name: fake(f => f.commerce.productName()),
  price: fake(f => parseFloat(f.commerce.price())),
  sentiment: oneOf('neutral', 'positive', 'negative'),
  on_sale: bool(),
  slug: fake(f => f.lorem.slug()),
});
```

```
import { itemBuilder } from 'frontend/lib/factories'

const dummyItem = itemBuilder()

const dummyItemWithName = itemBuilder({ name: 'Oxford shirt' })
```

Avoiding brittle tests

When I change the code I have to change the tests as well, so all tests do is double the amount of work I have!

objects have a contract: a public API that they provide

```
it('filters the items to only shirts', () => {
  const shirtFinder = new ShirtFinder({ priceMax: 5000 })

  expect(shirtFinder.__foundShirts).toEqual([])

  expect(shirtFinder.getShirts()).toEqual([])
})
```

You should be able to rewrite code *without changing all your tests.*

Test things by calling them just like you do in real life

When writing new tests, check that they fail!

Can you spot the problem with this test?

```
describe('finding items in a price range', () => {
  it('returns the right set of items', () => {
    const dummyItems = [{ name: 'shirt', price: 2000 }, ...]

    const result = itemFinder(dummyItems).min(1000).max(5000)
  })
})
```

Many test frameworks will pass a test without an assertion!

```
expect.assertions(2)
```

If you write a test and it passes first time, *try to break it*

small

feedback

loops

Write code
Check if it worked
Write code
Check if it worked
Write code
Check if it worked
Write code
Check if it worked
Write code
Check if it worked
Write code
Check if it worked
Write code
Check if it worked

Write code

this time needs to be short

Check if it worked

Write code

wait 5 seconds for webpack

manually refresh the browser

click the button you're working on

go back to your editor

Check if it worked

Write code

hit save in your editor

Run tests

Fixing bugs with short feedback loops

There's a bug where the price filtering max price limit is not used

1. Prove it in a failing test

```
it('filters by max price correctly', () => {
  const items = [{ name: 'shirt', price: 3000 }]
  expect(itemFinder({ maxPrice: 2000})).toEqual([])
})
```

TEST FAILURE:

Expected [], got [{ name: 'shirt', price: 3000 }]

2. Fix the bug *without changing the test*

3. Rerun the test

```
TEST PASSED
Expected [], got []
```

Confident refactoring

Red
Green
Refactor

1: Write the test and see it fail.

2: Write whatever code it takes to make it pass.

3: Rewrite the code until you're happy, using the tests to guide you.

1: Write the test and see it fail.

2: Write whatever code it takes to make it pass.

3: Rewrite the code until you're happy, using the tests to guide you.

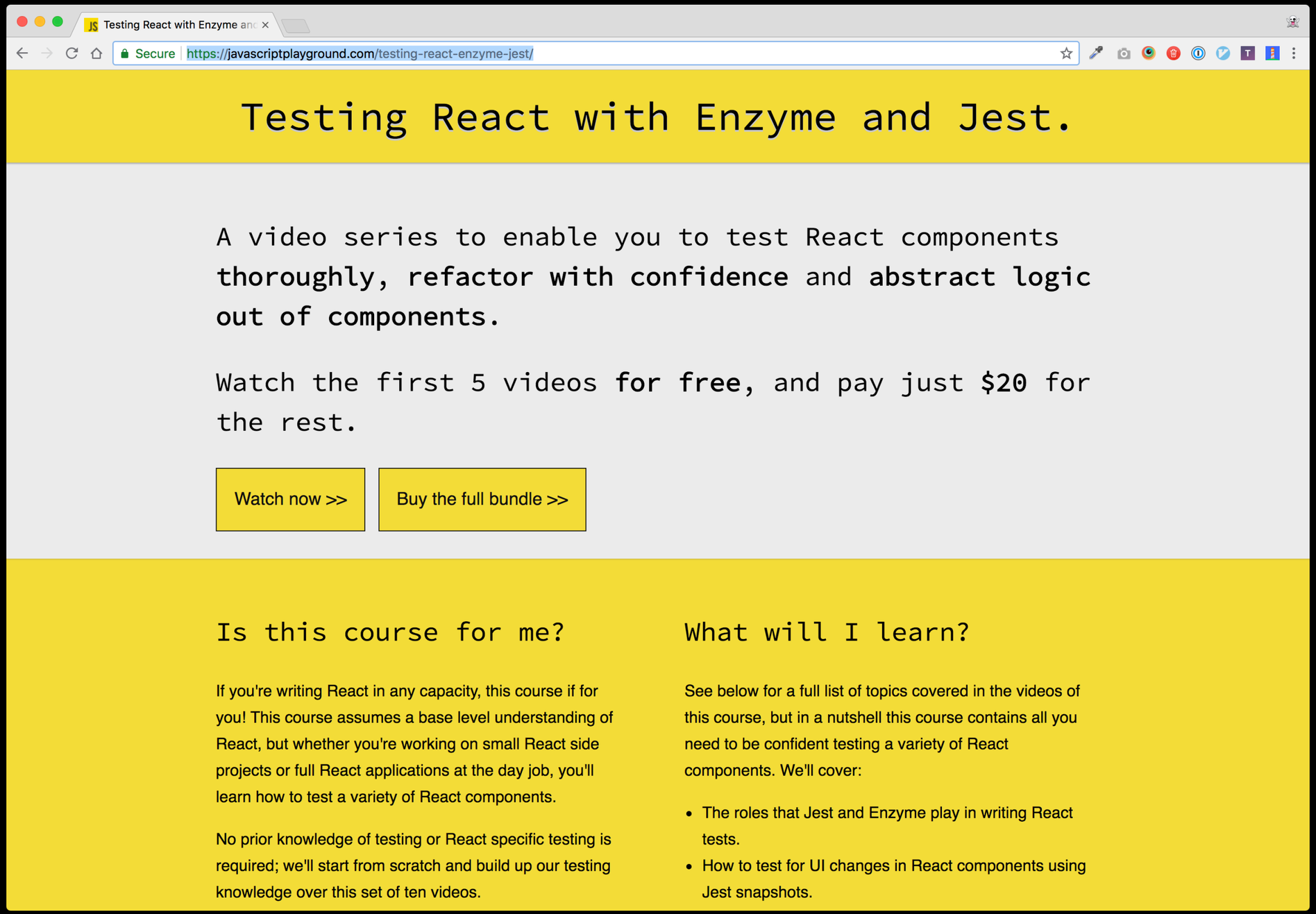
1: Write the test and see it fail.

2: Write whatever code it takes to make it pass.

3: Rewrite the code until you're happy, using the tests to guide you.

You should feel *slightly uncomfortable* when you have a failing test.

Testing React


Testing React with Enzyme and
Secure
https://javascriptplayground.com/testing-react-enzyme-jest/
Testing React with Enzyme and Jest.
A video series to enable you to test React components thoroughly, refactor with confidence and abstract logic out of components.
Watch the first 5 videos for free, and pay just $20 for the rest.
Watch now >>
Buy the full bundle >>
Is this course for me?
If you're writing React in any capacity, this course if for you! This course assumes a base level understanding of React, but whether you're working on small React side projects or full React applications at the day job, you'll learn how to test a variety of React components.
No prior knowledge of testing or React specific testing is required; we'll start from scratch and build up our testing knowledge over this set of ten videos.
What will I learn?
See below for a full list of topics covered in the videos of this course, but in a nutshell this course contains all you need to be confident testing a variety of React components. We'll cover:
The roles that Jest and Enzyme play in writing React tests.
How to test for UI changes in React components using Jest snapshots.

<ShamelessPlug>

You should buy my course on Testing React!

javascriptplayground.com/testing-react-enzyme-jest/


40% off!
What a deal!
Use JACKFRIDAY to get 40% off 🎉
(for today only!)

There is only one rule for testing React components

How can I test hooks in React?

You don't


CONTRACT

A React component's contract is *what it shows to the user*.

So test your components *as if you are a user*.

```
it('closes when you click', () => {
})
```

YOUR TEST

COMPONENT UNDER TEST

Which test is better?

```
const wrapper = mount(<Button />)
wrapper.find('a').simulate('click')
expect(wrapper.getState().isDisabled).toEqual(true)
```

Or:

```
const wrapper = mount(<Button />)
wrapper.find('button').simulate('click')
expect(wrapper.find('button').prop('disabled')).toEqual(true)
```

👎 Reaches into the component to read some state

```
expect(wrapper.getState().isDisabled).toEqual(true)
```

👍 Reads the component as the user would.

```
expect(wrapper.find('button').prop('disabled')).toEqual(true)
```

You can use Enzyme, react-testing-library or any alternative.

If you test as the user, you'll have good tests.

⁉ The exact framework doesn't actually matter that much.

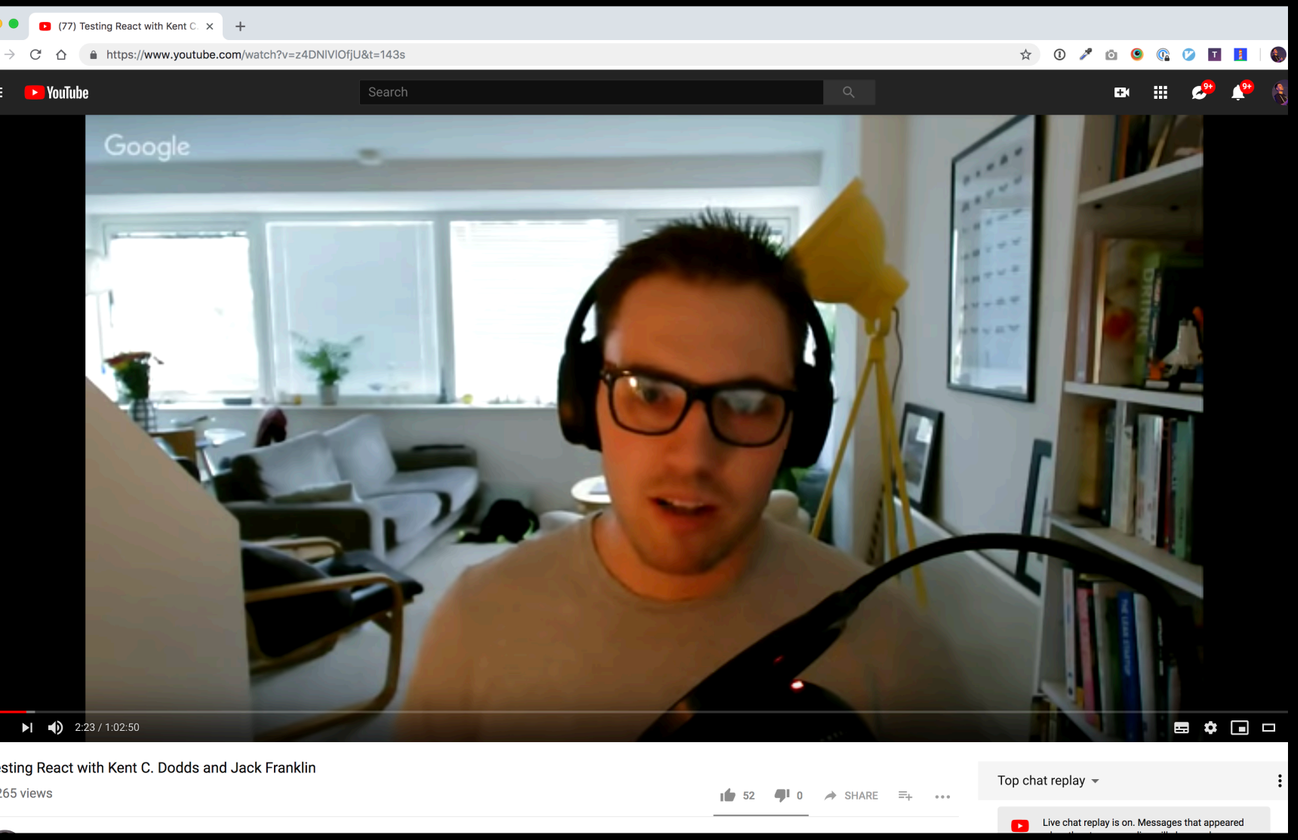
To conclude...

1. Remember what makes a good test: *setup*, *invocation*, *assertion*

2. Avoid brittle tests: test the *public contract*, not internal details.

3. When it comes to React, *the framework doesn't matter if you test like a user*

If you liked this, you might like...



https://www.youtube.com/watch?v=z4DNlVlOfjU

...with Kent C. Dodds (😎) and myself

Come and find me if you have questions, or tweet @Jack_Franklin