Testing Functional React Components

A journey from Enzyme to React Testing Library

Jordan Cooperman

Twitter: @jordantomax
Email: jordan@kobee.io

Being a JavaScript programmer is like being a dog

jQuery

A little here, a little there

Backbone

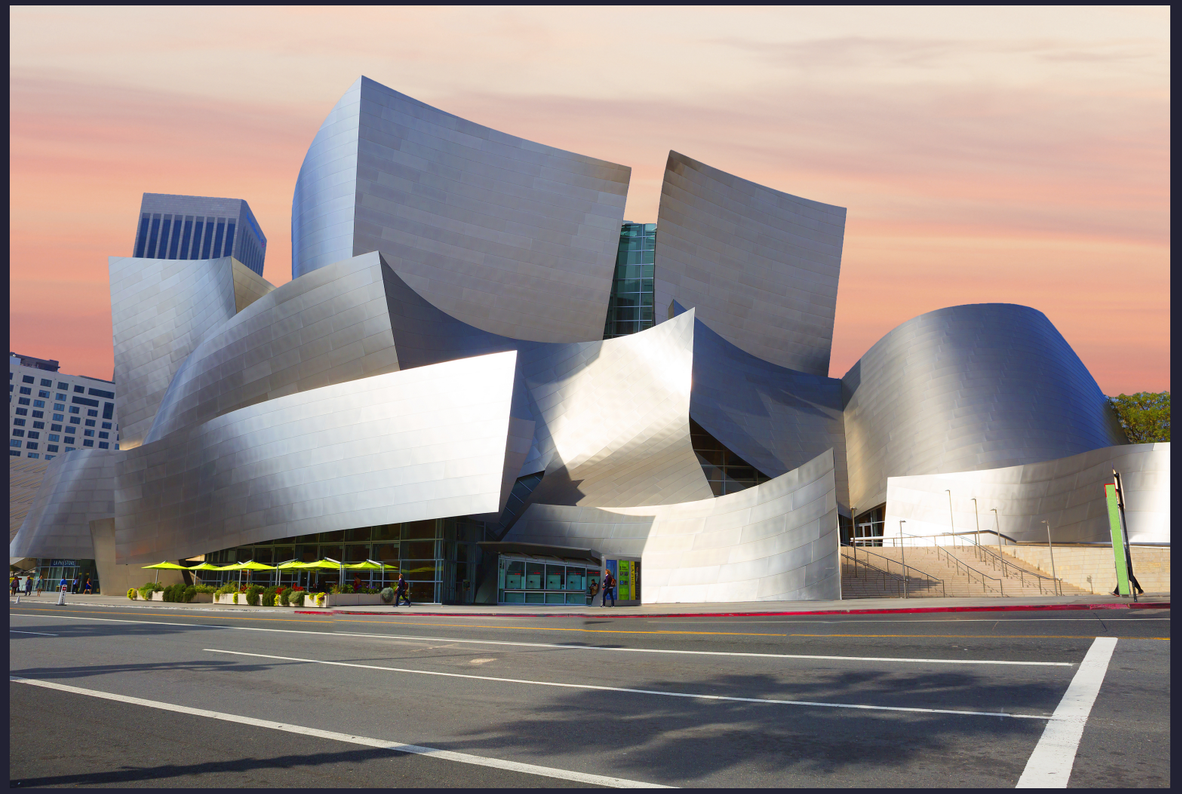
kind of organized?

Angular

Full framework, opinionated and complicated

React

Simple, flexible, encapsulated



Unit testing with React is easy

Code coverage

Code coverage

- 90%?

Code coverage

- 90%?
- 70%?

Code coverage

- 90%?
- 70%?
- 50%?

Code coverage

- 90%?
- 70%?
- 50%?
- Don’t know?

Let's write some bad tests!

First, let's look at the application that we're testing

```
class List extends React.Component {
  todoCreate (e) { ... }
  handleChange (e, i) { ... }
  handleRemove (i) { ... }

  render () {
    return (
      <div>
        <button onClick={this.todoCreate}>Add a todo</button>

        <ul>
          {this.state.todos.map((todo, i) => {
            return <ListItem
                handleChange={(e) => this.handleChange(e, i)}
                handleRemove={() => this.handleRemove(i)}
                key={i}
                todo={todo}
              />
          })}
        </ul>
      </div>
    )
  }
}
```

```
describe('List', () => {

  // ...

  it('adds a todo when clicking add', async () => {
    const wrapper = shallow(<List />)
    await wrapper.find('button').props().onClick()
    expect(wrapper.instance().state.todos.length).toEqual(1)
  })

  // ...

})
```

```
describe('List', () => {

  // ...

  it('adds a todo when clicking add', async () => {
    const wrapper = shallow(<List />)
    await wrapper.find('button').props().onClick()
    expect(wrapper.instance().state.todos.length).toEqual(1)
  })

  // ...

})
```

```
describe('List', () => {

  // ...

  it('adds a todo when clicking add', async () => {
    const wrapper = shallow(<List />)
    await wrapper.find('button').props().onClick()
    expect(wrapper.instance().state.todos.length).toEqual(1)
  })

  // ...

})
```

```
describe('List', () => {

  // ...

  it('adds a todo when clicking add', async () => {
    const wrapper = shallow(<List />)
    await wrapper.find('button').props().onClick()
    expect(wrapper.instance().state.todos.length).toEqual(1)
  })

  // ...

})
```

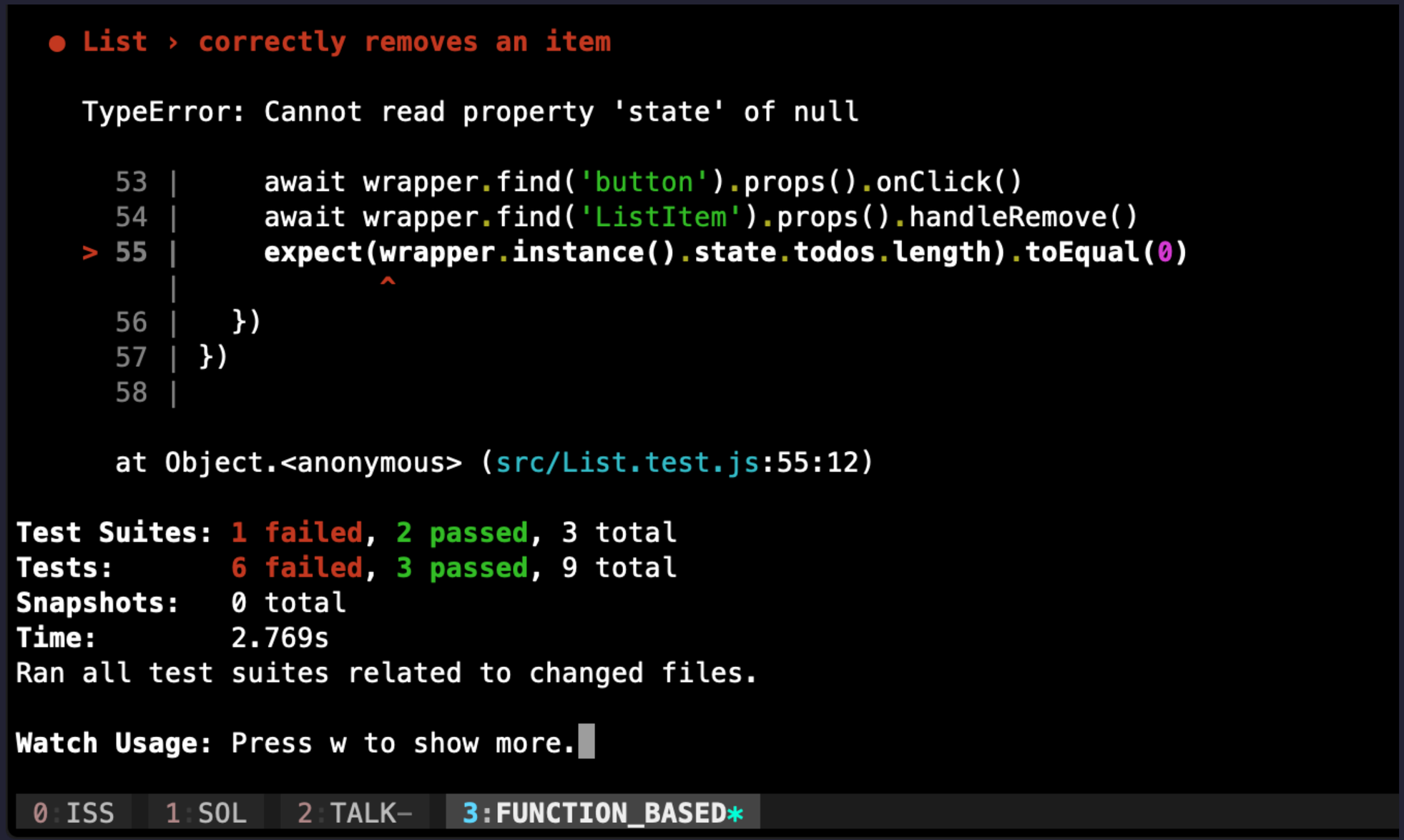
What happens if we re-write the component with the latest React hotness?

```
function List () {
  const [todos, setTodos] = React.useState([])

  function create (e) { ... }
  function handleChange (e, i) { ... }
  function handleRemove (i) { ... }

  return (
    <div>
      <button onClick={create}>Add a todo</button>

      <ul>
        {todos.map((todo, i) => {
          return (
            <ListItem
              handleChange={(e) => handleChange(e, i)}
              handleRemove={() => handleRemove(i)}
              key={i}
              todo={todo}
            />
          )
        })}
      </ul>
    </div>
  )
}
```


● List › correctly removes an item
TypeError: Cannot read property 'state' of null
53 | await wrapper.find('button').props().onClick()
54 | await wrapper.find('ListItem').props().handleRemove()
> 55 | expect(wrapper.instance().state.todos.length).toEqual(0)
56 | })
57 | })
58 |
at Object.<anonymous> (src/List.test.js:55:12)
Test Suites: 1 failed, 2 passed, 3 total
Tests: 6 failed, 3 passed, 9 total
Snapshots: 0 total
Time: 2.769s
Ran all test suites related to changed files.
Watch Usage: Press w to show more.
0:ISS 1:SOL 2:TALK- 3:FUNCTION_BASED*


I knew it was too good to be true

```
describe('List', () => {

  // ...

  it('adds a todo when clicking add', async () => {
    const wrapper = shallow(<List />)
    await wrapper.find('button').props().onClick()
    expect(wrapper.find('ListItem').length).toEqual(1)
  })

  // ...

})
```

```
describe('List', () => {

  // ...

  it('adds a todo when clicking add', async () => {
    const wrapper = shallow(<List />)
    await wrapper.find('button').props().onClick()
    expect(wrapper.find('ListItem').length).toEqual(1)
  })

  // ...

})
```


Terminal — tmux — 83×29
PASS src/App.test.js
PASS src/List.test.js
PASS src/ListItem.test.js
Test Suites: 3 passed, 3 total
Tests: 7 passed, 7 total
Snapshots: 0 total
Time: 1.902s, estimated 2s
Ran all test suites.
Watch Usage
› Press f to run only failed tests.
› Press o to only run tests related to changed files.
› Press q to quit watch mode.
› Press p to filter by a filename regex pattern.
› Press t to filter by a test name regex pattern.
› Press Enter to trigger a test run.
0:ISS 1:SOL 2:TALK- 3:FUNCTION_BASED* 13/08 11:59:35

No

```
it('adds a todo when clicking add', async () => {
  const wrapper = shallow(<List />)
  await wrapper.find('button').props().onClick()
  expect(wrapper.find('ListItem').length).toEqual(1)
})
```

- Shallow render not explicit in what is not rendered
- Calling a method from props does not mirror user behavior
- 'ListItem' is a developer defined named which does not appear in the dom

```
it('adds a todo when clicking add', async () => {
  const wrapper = shallow(<List />)
  await wrapper.find('button').props().onClick()
  expect(wrapper.find('ListItem').length).toEqual(1)
})
```

- Shallow render not explicit in what is not rendered
- Calling a method from props does not mirror user behavior
- 'ListItem' is a developer defined named which does not appear in the dom

```
it('adds a todo when clicking add', async () => {
  const wrapper = shallow(<List />)
  await wrapper.find('button').props().onClick()
  expect(wrapper.find('ListItem').length).toEqual(1)
})
```

- Shallow render not explicit in what is not rendered
- Calling a method from props does not mirror user behavior
- 'ListItem' is a developer defined named which does not appear in the dom

```
it('adds a todo when clicking add', async () => {
  const wrapper = shallow(<List />)
  await wrapper.find('button').props().onClick()
  expect(wrapper.find('ListItem').length).toEqual(1)
})
```

- Shallow render not explicit in what is not rendered
- Calling a method from props does not mirror user behavior
- 'ListItem' is a developer defined named which does not appear in the dom

What to do?

What to do?

- Fully render, mock explicitly instead of shallow rendering

What to do?

- Fully render, mock explicitly instead of shallow rendering
- Trigger real click event instead of calling onClick directly

What to do?

- Fully render, mock explicitly instead of shallow rendering
- Trigger real click event instead of calling onClick directly
- Look at the actual DOM, not developer defined names

Let's write better tests

The problem

You want to write maintainable tests for your React components. As a part of this goal, you want your tests to avoid including implementation details of your components and rather focus on making your tests give you the confidence for which they are intended. As part of this, you want your testbase to be maintainable in the long run so refactors of your components (changes to implementation but not functionality) don't break your tests and slow you and your team down.

This solution

The `react-testing-library` is a very lightweight solution for testing React components. It provides light utility functions on top of `react-dom` and `react-dom/test-utils`, in a way that encourages better testing practices. Its primary guiding principle is:

> The more your tests resemble the way your software is used, the more confidence they can give you.

Installation

This module is distributed via npm which is bundled with node and should be installed as one of your project's `devDependencies`:

```
npm install --save-dev @testing-library/react
```

https://github.com/testing-library/react-testing-library

```
describe('List', () => {

  // ...

  it('adds a todo when clicking add', async () => {
    const wrapper = render(<List />)
    act(() => { fireEvent.click(wrapper.getByText('Add a todo')) })
    expect(wrapper.container.querySelector('li')).not.toBeNull()
  })

  // ...

})
```

```
describe('List', () => {

  // ...

  it('adds a todo when clicking add', async () => {
    const wrapper = render(<List />)
    act(() => { fireEvent.click(wrapper.getByText('Add a todo')) })
    expect(wrapper.container.querySelector('li')).not.toBeNull()
  })

  // ...

})
```

```
describe('List', () => {

  // ...

  it('adds a todo when clicking add', async () => {
    const wrapper = render(<List />)
    act(() => { fireEvent.click(wrapper.getByText('Add a todo')) })
    expect(wrapper.container.querySelector('li')).not.toBeNull()
  })

  // ...

})
```

```
describe('List', () => {

  // ...

  it('adds a todo when clicking add', async () => {
    const wrapper = render(<List />)
    act(() => { fireEvent.click(wrapper.getByText('Add a todo')) })
    expect(wrapper.container.querySelector('li')).not.toBeNull()
  })

  // ...

})
```

Not bad

Let's also ListItem which requires prop data

```
import ListItem from './ListItem'

describe('ListItem', () => {
  it('shows an input to edit when clicking name', async () => {
    const props = {
      handleChange: jest.fn(),
      todo: {
        name: 'todo',
        complete: false
      }
    }
    const wrapper = render(<ListItem {...props} />)
    await act(async () => fireEvent.click(wrapper.container.querySelector('li > span')))
    expect(wrapper.getByPlaceholderText('Edit todo')).toBeDefined()
  })
})
```

```
import ListItem from './ListItem'

describe('ListItem', () => {
  it('shows an input to edit when clicking name', async () => {
    const props = {
      handleChange: jest.fn(),
      todo: {
        name: 'todo',
        complete: false
      }
    }
    const wrapper = render(<ListItem {...props} />)
    await act(async () => fireEvent.click(wrapper.container.querySelector('li > span')))
    expect(wrapper.getByPlaceholderText('Edit todo')).toBeDefined()
  })
})
```

We can do better with factories

Wherever there is structured data, there should be factories

Rosie

build passing

Rosie is a factory for building JavaScript objects, mostly useful for setting up test data. It is inspired by factory_girl.

To use Rosie you first define a *factory*. The *factory* is defined in terms of *attributes*, *sequences*, *options*, *callbacks*, and can inherit from other factories. Once the factory is defined you use it to build *objects*.

Define the data structure

Using generated data makes your tests more meaningful

```
function randomFromArray (array) {
  return array[Math.floor(Math.random() * array.length)]
}

Factory.define('todo')
  .attr('name', () => faker.lorem.word())
  .attr('complete', () => randomFromArray([true, false]))
```

Use the factory

```
import ListItem from './ListItem'

describe('ListItem', () => {
  it('shows an input to edit when clicking name', async () => {
    const props = {
      handleChange: jest.fn(),
      todo: Factory.build('todo')
    }
    const wrapper = render(<ListItem {...props} />)
    await act(async () => fireEvent.click(wrapper.container.querySelector('li > span')))
    expect(wrapper.getByPlaceholderText('Edit todo')).toBeDefined()
  })
})
```

Use the factory

```
import ListItem from './ListItem'

describe('ListItem', () => {
  it('shows an input to edit when clicking name', async () => {
    const props = {
      handleChange: jest.fn(),
      todo: Factory.build('todo')
    }
    const wrapper = render(<ListItem {...props} />)
    await act(async () => fireEvent.click(wrapper.container.querySelector('li > span')))
    expect(wrapper.getByPlaceholderText('Edit todo')).toBeDefined()
  })
})
```

What about integration tests?

https://twitter.com/rauchg/status/807626710350839808

Contract tests with factories

Using standard interfaces to create consistent failure in cases where services change

(Martin Fowler explains contract tests well https://martinfowler.com/bliki/ContractTest.html)

Example: Router

With React Router

We're using React Router, but we may switch in the future.

(Just an example, I <3 React Router!)

Let's write tests to ensure that our tests fail when the shape of our router changes

The component

```
import { withRouter } from 'react-router'

function List ({ history }) {
  function goBack () { ... }

  // This is a small piece of the List component
  // Which will be wrapped in the withRouter HOC

  return (
    <button onClick={goBack}>
      Go back
    </button>
  )
}

export default withRouter(List)
```

The test

```
jest.mock('react-router', () => ({
  withRouter: Component => Component
}))

Factory.define('withRouter')
  .attr('history', () => ({
    goBack: jest.fn()
  }))

describe('List', () => {
  it('correctly goes back', async () => {
    const props = {
      ...Factory.build('withRouter')
    }
    const wrapper = render(<List {...props} />)
    await act(async () => fireEvent.click(wrapper.getByText('Go back')))
    expect(props.history.goBack).toHaveBeenCalled()
  })
})
```

The test

```
jest.mock('react-router', () => ({
  withRouter: Component => Component
}))

Factory.define('withRouter')
  .attr('history', () => ({
    goBack: jest.fn()
  }))

describe('List', () => {
  it('correctly goes back', async () => {
    const props = {
      ...Factory.build('withRouter')
    }
    const wrapper = render(<List {...props} />)
    await act(async () => fireEvent.click(wrapper.getByText('Go back')))
    expect(props.history.goBack).toHaveBeenCalled()
  })
})
```

The test

```
jest.mock('react-router', () => ({
  withRouter: Component => Component
}))

Factory.define('withRouter')
  .attr('history', () => ({
    goBack: jest.fn()
  }))

describe('List', () => {
  it('correctly goes back', async () => {
    const props = {
      ...Factory.build('withRouter')
    }
    const wrapper = render(<List {...props} />)
    await act(async () => fireEvent.click(wrapper.getByText('Go back')))
    expect(props.history.goBack).toHaveBeenCalled()
  })
})
```

The test

```
jest.mock('react-router', () => ({
  withRouter: Component => Component
}))

Factory.define('withRouter')
  .attr('history', () => ({
    goBack: jest.fn()
  }))

describe('List', () => {
  it('correctly goes back', async () => {
    const props = {
      ...Factory.build('withRouter')
    }
    const wrapper = render(<List {...props} />)
    await act(async () => fireEvent.click(wrapper.getByText('Go back')))
    expect(props.history.goBack).toHaveBeenCalled()
  })
})
```

Now changing our router isn't so scary

We can change the shape of the withRouter factory, which would cause all of our tests to fail.

This gives us confidence in making decisions like replacing a library.

To summarize

To summarize

- Test user facing behavior (i.e. text, style that is visible to the user)

To summarize

- Test user facing behavior (i.e. text, style that is visible to the user)
- Write unit tests to test general use cases

To summarize

- Test user facing behavior (i.e. text, style that is visible to the user)
- Write unit tests to test general use cases
- Use contract tests to make large changes less scary

To summarize

- Test user facing behavior (i.e. text, style that is visible to the user)
- Write unit tests to test general use cases
- Use contract tests to make large changes less scary
- Use factories where there is structured data

To summarize

- Test user facing behavior (i.e. text, style that is visible to the user)
- Write unit tests to test general use cases
- Use contract tests to make large changes less scary
- Use factories where there is structured data
- Use factories to create consistent contracts

Tools and Resources

https://github.com/jordantomax/talks/tree/master/testing-functional-components
https://github.com/stalniy/bdd-lazy-var
https://github.com/rosiejs/rosie
https://github.com/marak/Faker.js/
https://github.com/testing-library/react-testing-library https://kentcdodds.com/blog/testing-implementation-details
https://martinfowler.com/bliki/ContractTest.html

Thank you

@jordantomax
jordan@kobee.io
kobee.io