Introduction to automated frontend testing

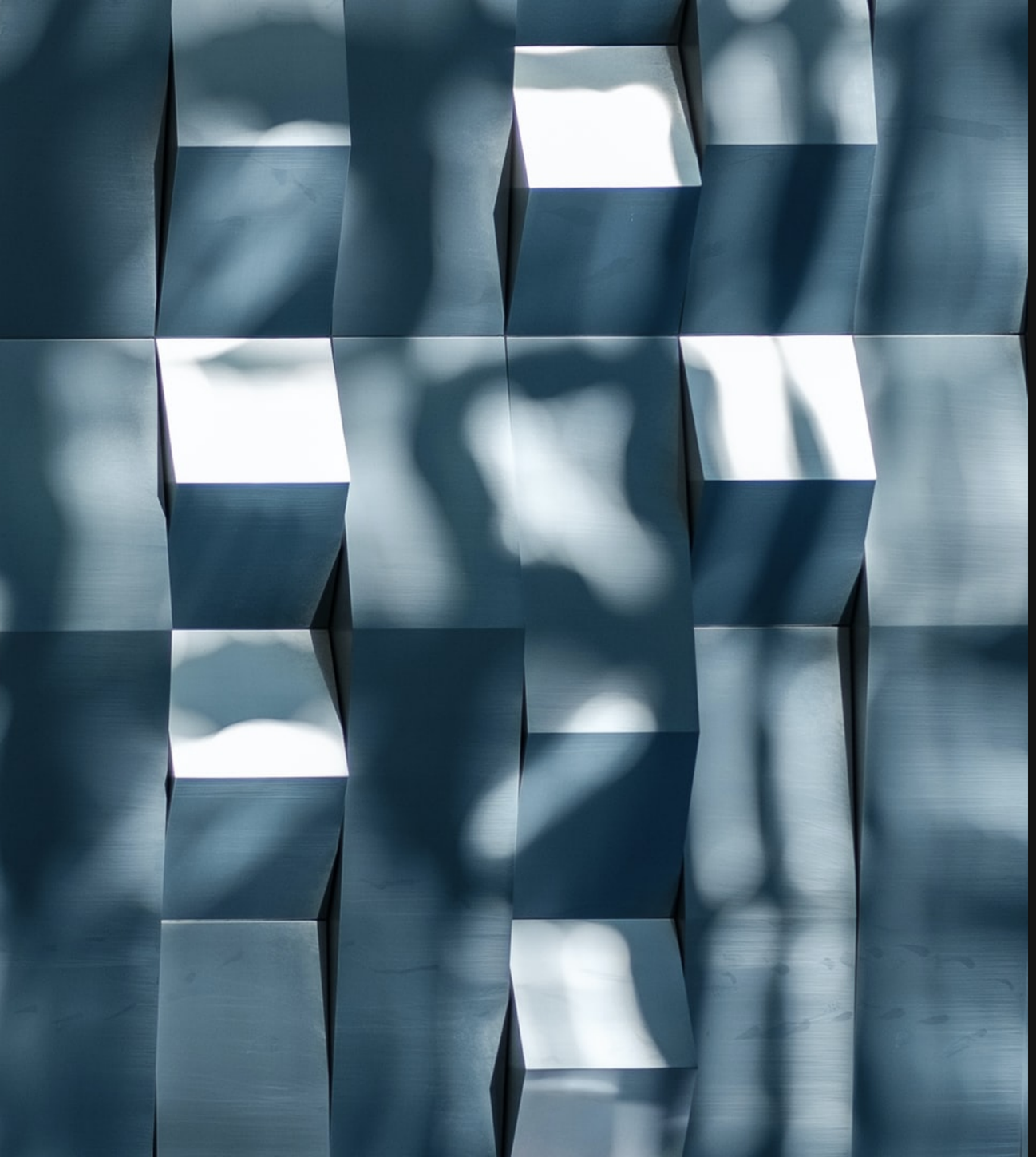


Table of contents

What is automated frontend testing?

What is automated frontend testing?

Testing - method to check whether the actual software product does not match expected requirements.

- correctness
- performance
- security
- accessibility

Automated testing - code, that when executed, automatically tests other code.

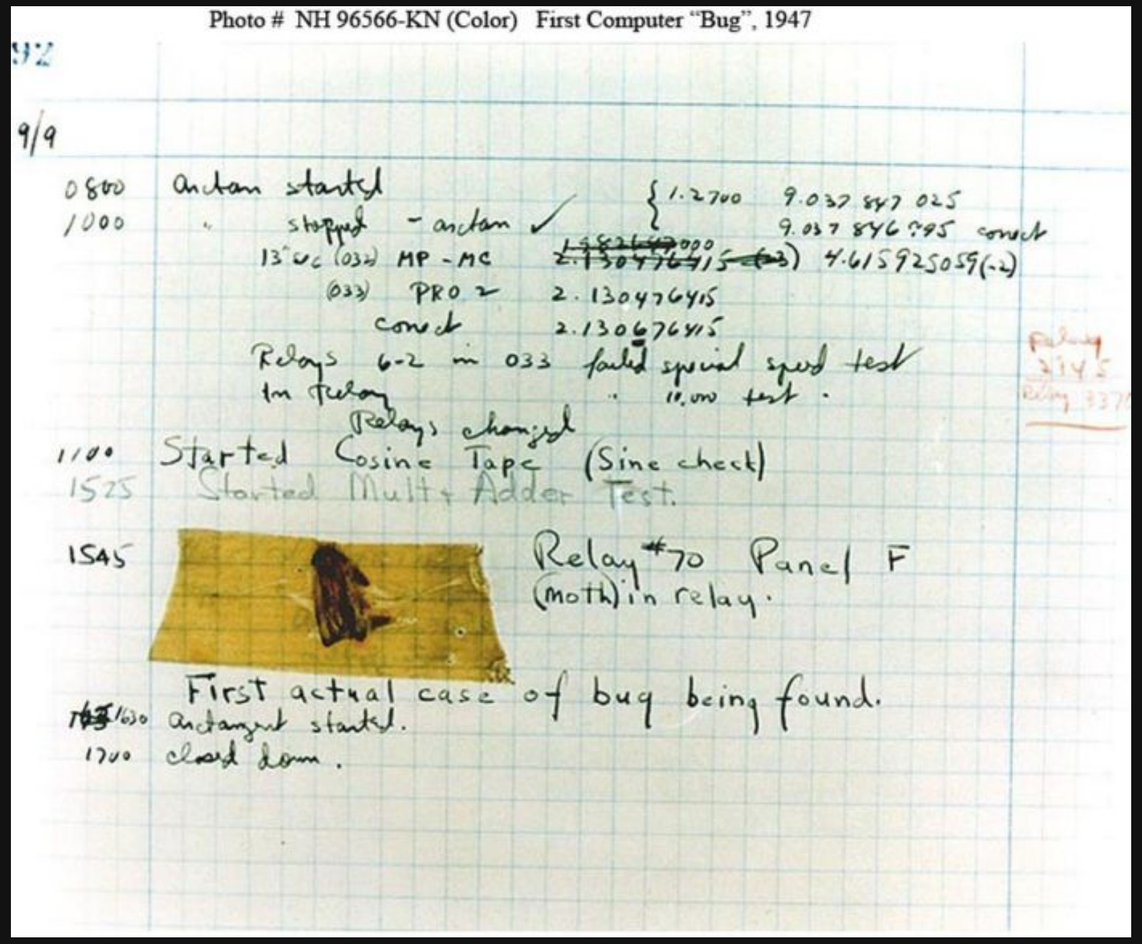
Automated frontend testing - automated testing software, that tests front end applications.

- validating what people see on the screen
- validating application behavior cross-browsers and cross-platforms
- detecting client side performance issues
- detecting accessibility issues

Tests allow us to find errors in the code and increase its quality, but they do not guarantee correctness.

Why should we test?

Why should we test?



Computer Mark II, Grace Hopper, Harvard, 1947

Not having automated tests results in many pain points:

- increased risk of functionalities not working according to the specification
- repeating manual tests
- introducing regressions
- lower code quality

Automated tests can solve most of the problems mentioned above.

Tests pyramid

Tests pyramid

Type	Description	Tools
Unit	check that individual unit of code produces expected output or side effects (it can be function, class, composable, component...)	Jest, Vitest
Integration/Component	check that your component mounts, renders, can be interacted with, and behaves as expected.	Vue Test Utils, Vue Testing Library, Cypress Component Testing
End to End	check features that span multiple pages and make real network requests against your production-built Vue application. These tests often involve standing up a database or other backend	Cypress, Selenium, Playwright

This division is not very strict. Depending on how we write tests, component testing can be considered as unit or integration testing.

Today we will focus on Vitest + Vue Test Utils.

Vitest

Vitest

- unit testing framework
- vite powered
- jest compatible
- smart watch mode
- c8 code coverage
- recommended to use by official vue docs

Getting started:

```
npm install -D vitest
```

Run with hot reloading:

```
vitest
```

Run once:

```
vitest run
```

Run subset of tests:

```
vitest foobar
```

Run once and generate coverage:

```
vitest run --coverage
```

Vitest

describe, it:

```
const myBeverage = { delicious: true, sour: false };
describe('my beverage', () => {
  it('is delicious', () => {
    expect(myBeverage.delicious).toBeTruthy();
  });

  test('is not sour', () => {
    expect(myBeverage.sour).toBeFalsy();
  });
});
```

afterAll, afterEach, beforeAll, beforeEach:

```
describe('test suite using store', () => {
  beforeEach(() => {
    initializeStore();
  });
  it('test using store', () => { ... });
});

afterAll(() => {
  cleanup();
});
```

matchers - equality:

```
expect(2 + 2).toBe(4);
expect({ one: 1 }).toEqual({ one: 1 });
expect({ one: 1 }).not.toBe({ one: 1 });
```

matchers - truthiness:

```
expect(null).toBeNull();
expect(undefined).toBeUndefined();
expect('').toBeFalsy()
expect(1).toBeTruthy();
```

Vitest

matchers - numbers:

```
expect(4).toBeGreaterThan(3);
expect(3.5).toBeGreaterThanOrEqual(3.5);
expect(2).toBeLessThan(5);
```

matchers - floats, strings, iterables:

```
expect(0.2 + 0.1).not.toBe(0.3);
expect(0.2 + 0.1).toBeCloseTo(0.3);
expect('Christoph').toMatch(new RegExp('stop'));
expect([1, 1, 2, 3, 5, 8, 13]).toContain(5);
expect([1, 1, 2, 3, 5, 8, 13]).toHaveLength(7);
```

functions:

```
const rept = vi.fn((str, num) ⇒ str.repeat(num));
rept('a', 3);
expect(rept).toHaveBeenCalled();
expect(rept).toHaveBeenCalledWith('a', 3);
expect(rept).toHaveBeenCalledWith(expect.anything(), 3);
expect(rept).toHaveReturnedWith('aaa');
```

mocking globals variables:

```
const IntersectMock = vi.fn(() ⇒ ({
  disconnect: vi.fn(),
  observe: vi.fn(),
  unobserve: vi.fn(),
}));
vi.stubGlobal('IntersectionObserver', IntersectMock);
```

Vitest

mocking modules:

```
// use-object.js
export function useObject() {
  return { method: () => true };
}
// file.js
import { useObject } from 'use-object';
const obj = useObject();
obj.method();
```

```
// file.spec.js
import { useObject } from 'some-path';
vi.mock('some-path', () => ({
  useObject: vi.fn(() => ({
    method: vi.fn(),
  })),
}));
```

snapshot testing:

```
import { mount } from '@vue/test-utils';
import Loader from '@/Loader.vue';

describe('Loader.vue', () => {
  it('match snapshot', () => {
    const wrapper = mount(Loader, {
      propsData: { loading: true },
    });
    expect(wrapper.html()).toMatchSnapshot();
  });
});
```

```
exports[`Loader.vue > match snapshot 1`] = `
"<div class=\\"text-center\\">
  <div class=\\"v-progress-circular\\">
    <i class=\\"v-icon material-icons\\">loading</i>
  </div>
  <p class=\\"pt-2 mb-0\\">Loading</p>
</div>"
`;
```

Vue Test Utils

Vue Test Utils

- official component testing utility library for Vue.js
- aimed to simplify testing Vue.js components
- provides some methods to mount and interact with Vue components in an isolated manner

Getting started:

```
npm install -D @vue/test-utils
```

mount:

```
import { mount } from '@vue/test-utils';
import Foo from './Foo.vue';

describe('Foo', () => {
  it('renders a div', () => {
    const wrapper = mount(Foo);
    expect(wrapper.contains('div')).toBe(true);
  });
});
```

shallowMount:

```
import { shallowMount } from '@vue/test-utils';
import Foo from './Foo.vue';

describe('Foo', () => {
  it('renders red div', () => {
    const wrapper = shallowMount(Foo, {
      propsData: {
        color: 'red',
      },
    });
    expect(wrapper.props().color).toBe('red');
  });
});
```

Vue Test Utils

mocks:

```
import { mount } from '@vue/test-utils';
import Foo from './Foo.vue';

describe('Foo', () => {
  it('contains the correct route', () => {
    const $route = { path: 'projects/9/apps/321' };
    const wrapper = mount(Foo, {
      mocks: {
        $route,
      },
    });
    expect(wrapper.vm.$route.path).toBe($route.path);
  });
});
```

stubs:

```
import { mount } from '@vue/test-utils';
import App from '@/App.vue';


describe('App.vue', () => {
  it('match snapshot', () => {
    const wrapper = mount(App, {
      stubs: ['Toolbar', 'ErrorDialog', 'Notifications'],
    });
    expect(wrapper.html()).toMatchSnapshot();
  });
});
```

Vue Test Utils

createLocalVue:

```
import Vuex from 'vuex';
import { mount, createLocalVue } from '@vue/test-utils';
import Foo from '@/components/Foo.vue';

const localVue = createLocalVue();
localVue.use(Vuex);

const store = new Vuex.Store({
  state: { username: 'alice' },
});

describe('Foo', () => {
  it('renders a username using Vuex store', () => {
    const wrapper = mount(Foo, {
      store,
      localVue,
    });

    expect(wrapper.find('.name').text()).toBe('alice');
  });
});
```

Vue Test Utils

attributes():

```
import { mount } from '@vue/test-utils';
import Foo from './Foo.vue';

const wrapper = mount(Foo);
expect(wrapper.attributes().id).toBe('foo');
expect(wrapper.attributes('id')).toBe('foo');
```

classes():

```
import { mount } from '@vue/test-utils';
import Foo from './Foo.vue';

const wrapper = mount(Foo);
expect(wrapper.classes()).toContain('bar');
expect(wrapper.classes('bar')).toBe(true);
```

destroy():

```
import { mount } from '@vue/test-utils';
import Foo from './Foo.vue';

const wrapper = mount(Foo);
wrapper.destroy();
```

emitted():

```
import { mount } from '@vue/test-utils';

it('emit demo', async () => {
  const wrapper = mount(Component);
  wrapper.vm.$emit('foo');
  wrapper.vm.$emit('foo', 123);
  /* { foo: [[], [123]] } */
  expect(wrapper.emitted().foo).toBeTruthy();
  expect(wrapper.emitted('foo')).toBeTruthy();
  expect(wrapper.emitted().foo.length).toBe(2);
  expect(wrapper.emitted().foo[1]).toEqual([123]);
});
```

Vue Test Utils

find():

```
import { mount } from '@vue/test-utils';
import Foo from './Foo.vue';

const wrapper = mount(Foo);
const div = wrapper.find('div');
```

findAll():

```
import { mount } from '@vue/test-utils';
import Foo from './Foo.vue';

const wrapper = mount(Foo);
const div = wrapper.findAll('div').at(5);
```

findComponent(), findAllComponents():

```
import { mount } from '@vue/test-utils';
import List from './List.vue';
import ListItem from './ListItem.vue';

const wrapper = mount(List);
const second = wrapper.findAllComponents(ListItem).at(2);
```

exists():

```
import { mount } from '@vue/test-utils';
import Foo from './Foo.vue';

const wrapper = mount(Foo);
expect(wrapper.exists()).toBe(true);
expect(wrapper.find('not-exist').exists()).toBe(false);
```

html():

```
import { mount } from '@vue/test-utils';
import Foo from './Foo.vue';

const wrapper = mount(Foo);
expect(wrapper.html()).toBe('<div><p>Foo</p></div>');
```

isVisible():

```
import { mount } from '@vue/test-utils';
import Foo from './Foo.vue';

const wrapper = mount(Foo);
expect(wrapper.isVisible()).toBe(true);
```

Vue Test Utils

setData():

```
import { mount } from '@vue/test-utils';
import Foo from './Foo.vue';

it('setData demo', async () => {
  const wrapper = mount(Foo);
  await wrapper.setData({ foo: 'bar' });
  expect(wrapper.vm.foo).toBe('bar');
});
```

setValue():

```
import { mount } from '@vue/test-utils';
import Foo from './Foo.vue';

it('setValue demo', async () => {
  const wrapper = mount(Foo);
  const textInput = wrapper.find('input[type="text"]');
  await textInput.setValue('some value');
  expect(textInput.element.value).toBe('some value');
});
```

trigger():

```
import { mount } from '@vue/test-utils';
import Foo from './Foo';

it('trigger demo', async () => {
  const clickHandler = vi.fn();
  const wrapper = mount(Foo, {
    propsData: { clickHandler }
  });
  await wrapper.trigger('click');
  expect(clickHandler).toHaveBeenCalled();
});
```

Vue Test Utils

createWrapper:

```
import { createWrapper } from '@vue/test-utils';
import MySelect from './MySelect.vue';

describe('MySelect', () => {
  it('when choosing an item emits change event', () => {
    const wrapper = mount(Select, {
      attachTo: document.body,
    });
    const root = createWrapper(document.body);
    wrapper.find('.select').trigger('click');
    root.find('.item:nth-child(2)').trigger('click');
    expect(wrapper.emitted('change')).toBeTruthy();
    wrapper.destroy();
  });
});
```

```
import { createWrapper } from '@vue/test-utils';
import MySelect from './MySelect.vue';

describe('MySelect', () => {
  it('when choosing an item emits change event', () => {
    const wrapper = mount(Select, {
      attachTo: document.body,
    });

    wrapper.find('.select').trigger('click');
    document
      .querySelector('.item:nth-child(2)')
      .dispatchEvent(
        new MouseEvent('click', {
          bubbles: true
        }),
      );
    expect(wrapper.emitted('change')).toBeTruthy();
    wrapper.destroy();
  });
});
```

Tips and tricks

Tips and tricks

1. Understand what to test and structure your test accordingly

```
describe('methods', () => {
  /* Testing every method in isolation */
});

describe('computed', () => {
  /* Testing every computed property in isolation */
});

describe('template', () => {
  /* Testing what is rendered */
});
```

✗ WRONG! We should test component public interface:

- rendered template
- emitted events
- side effects like vuex actions, router, etc.
- connection with children

Tips and tricks

1. Understand what to test and structure your test accordingly

```
<template>
  <main>
    <div v-if="loading">Loading ... </div>
    <template v-else>
      <p v-if="error">Something went wrong!</p>
      <div v-else>Loaded!</div>
    </template>
  </main>
</template>
```

```
describe('when loading', () => {
  it.todo('renders "Loading ... " text');
  it.todo('does not render error message');
  it.todo('does not render data');
});
describe('when there is an error', () => {
  it.todo('does not render "Loading ... " text');
  it.todo('renders error message');
  it.todo('does not render data');
});
```

Tips and tricks

2. Start with component factory

```
describe('TodoList', () ⇒ {
  it('when enabled, adding a new item causes update', () ⇒ {
    const wrapper = mount(TodoList, {
      propsData: { disabled: false },
      stubs: ['TodoItem'],
      mocks: {
        $todoListStore: {
          addTodo: vi.fn(),
          removeTodo: vi.fn(),
        },
      },
    })
  });
  it('when disabled, adding a new item does not cause update', () ⇒ {
    const wrapper = mount(TodoList, {
      propsData: { disabled: true },
      // ... all other options the same
  });
});
```

✗ WRONG! It's better to extract the component mounting to another function.

Tips and tricks

2. Start with component factory

```
describe('TodoList', () => {
  let wrapper;
  const createComponent = (propsData) => {
    wrapper = mount(TodoList, {
      propsData,
      stubs: ['TodoItem'],
      mocks: {
        $todoListStore: {
          addTodo: vi.fn(),
          removeTodo: vi.fn(),
        },
      },
    });
  };
  it('when enabled, adding a new item causes update', () => {
    createComponent({ disabled: false });
  });
  it('when disabled, adding a new item does not cause update', () => {
    createComponent({ disabled: true });
  });
});
```

Tips and tricks

3. Use helpers to find elements and components

```
describe('My component test', () => {
  let wrapper;

  it('test 1', () => {
    createComponent();
    wrapper.find('button[type="submit"]').trigger('click');
  });

  it('test 2', () => {
    createComponent();
    wrapper.find('button[type="submit"]').trigger('click');
  });
});
```

What if we change something in the confirmation button?

What if we replace it with custom component?

✗ WRONG! Unreadable and very vulnerable to implementation changes.

Tips and tricks

3. Use helpers to find elements and components

```
describe('My component test', () => {
  let wrapper;
  const findSubmitButton = () => wrapper.find('button[type="submit"]');

  it('test 1', () => {
    createComponent();
    findSubmitButton().trigger('click');
  });

  it('test 2', () => {
    createComponent();
    findSubmitButton().trigger('click');
  });
});
```

Tips and tricks

4. Do not test component internals

```
<div>
  <p>Count:</p><span data-testid="count">{{ count }}</span>
  <p>Double count:</p><span data-testid="double">{{ double }}</span>
  <button data-testid="increment-button" @click="incrementCount">Increment</button>
</div>
```

```
export default {
  data() {
    return { count: 0 };
  },
  computed: {
    double() {
      return this.count * 2;
    },
  },
  methods: {
    incrementCount() {
      this.count++;
    },
  },
};
```

Tips and tricks

4. Do not test component internals

```
describe('Counter', () => {
  it('calls correct method on button click', () => {
    createComponent();
    vi.spyOn(wrapper.vm, 'incrementCount');

    findIncrementButton().trigger('click');

    expect(wrapper.vm.incrementCount).toHaveBeenCalled();
    expect(wrapper.vm.count).toBe(1);
  });

  it('calculates double correctly', () => {
    createComponent({ data: { count: 1 } });
    expect(wrapper.vm.double).toBe(2);
  });
});
```

✗ WRONG! We are not testing component output. Very vulnerable to implementation changes.

Tips and tricks

4. Do not test component internals

```
describe('Counter', () => {
  it('increases the count on Increment button click', async () => {
    createComponent();
    expect(findCount().text()).toBe('0');
    await findIncrementButton().trigger('click');
    expect(findCount().text()).toBe('1');
  });

  it('calculates double correctly', async () => {
    createComponent({ data: { count: 1 } });
    expect(findDouble().text()).toBe('2');
  });
});
```

- forget about wrapper.vm
- do not spy on methods
- if we change implementation details, like method name or computed name, test should pass

Tips and tricks

5. Follow the user

```
describe('Counter', () => {
  it('increases the double on increasing count', async () => {
    createComponent({ data: { count: 1 } });
    expect(findDouble().text()).toBe('2');
    wrapper.vm.count = 2;
    expect(findDouble().text()).toBe('4');
  });
});
```

```
describe('Counter', () => {
  it('increases the double on increasing count', async () => {
    createComponent({ data: { count: 1 } });
    expect(findDouble().text()).toBe('2');
    wrapper.setData({ count: 2 });
    expect(findDouble().text()).toBe('4');
  });
});
```

✗ WRONG!

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

Tips and tricks

5. Follow the user

```
describe('Counter', () ⇒ {
  it('increases the double on increasing count', async () ⇒ {
    createComponent({ data: { count: 1 } });
    expect(findDouble().text()).toBe('2');
    await findIncrementButton().trigger('click');
    expect(findDouble().text()).toBe('4');
  });
});
```

Tips and tricks

6. Query elements semantically

Bad:

```
wrapper.find('.js-foo');
wrapper.find('.btn-primary');
wrapper.find('.qa-foo-component');
wrapper.find('#unique');
```

Better:

```
wrapper.find('[data-testid="my-foo-id"]');
wrapper.find({ ref: 'foo'});
```

Best:

```
wrapper.findByText('Count');
wrapper.findByRole('heading');
wrapper.findByRole('link', { name: 'Click Me' });
wrapper.findByLabelText('Password');
wrapper.findByTitle('Delete');
wrapper.findComponent(FooComponent);
```

More on that:

- Vue Testing Library
- Frontend Test Element Locators
- GitLab's "extendedWrapper"

Tips and tricks

7. Treat child components as black boxes

```
<div>
  <MyButton :double-count="double" @click="incrementCount">Increment</MyButton>
  <p>Count:</p><span data-testid="count">{{ count }}</span>
</div>
```

```
import MyButton from './MyButton.vue';
export default {
  components: { MyButton },
  data() {
    return { count: 0 };
  },
  computed: {
    double() { return this.count * 2; },
  },
  methods: {
    incrementCount() {
      this.count++;
    },
  },
};
```

Tips and tricks

7. Treat child components as black boxes

```
const findChildButton = () => wrapper.findComponent(MyButton);
const findCount = () => wrapper.find('[data-testid="child-counter"]');

it('passes a correct prop to child button', () => {
  createComponent({ data: { count: 4 } });
  expect(findChildButton().props('double-count')).toBe(8);
});

it('updates count on child button "click" event', async () => {
  createComponent();
  expect(findCount().text()).toBe('0');
  findChildButton().vm.$emit('click');
  await nextTick();
  expect(findCount().text()).toBe('1');
});
```

Tips and tricks

8. Mock Vuex in the component

```
<div class="username">{{ username }}</div>
```

```
export default {
  name: 'Username',
  data() {
    return {
      username: this.$store.state.username,
    };
  },
};
```

Tips and tricks

8. Mock Vuex in the component

```
describe('Username', () => {
  let wrapper;
  const createComponent = () => {
    Vue.use(Vuex);
    const store = new Vuex.Store({
      state: {
        username: 'alice',
      },
    });
    wrapper = mount(Username, {
      store,
    });
  };

  it('renders a username using a real Vuex store', () => {
    createComponent();
    expect(findUsername().text()).toBe('alice');
  });
});
```

✗ WRONG! We are polluting global Vue instance. Now all tests will have Vuex.

Tips and tricks

8. Mock Vuex in the component

```
describe('Username', () => {
  let wrapper;
  const createComponent = () => {
    const localVue = createLocalVue();
    localVue.use(Vuex);
    const store = new Vuex.Store({
      state: {
        username: 'alice',
      },
    });
    wrapper = mount(Username, {
      store, localVue,
    });
  };
  it('renders a username using a real Vuex store (correct way)', () => {
    createComponent();
    expect(findUsername().text()).toBe('alice');
  });
});
```

✗ WRONG! What if our store implementation is incorrect? Our test will fail.

Tips and tricks

8. Mock Vuex in the component

```
describe('Username', () => {
  let wrapper;
  const createComponent = () => {
    wrapper = mount(Username, {
      mocks: {
        $store: {
          state: { username: 'alice' },
        },
      },
    });
  };
  it('renders a username using mocked Vuex store', () => {
    createComponent();
    expect(findUsername().text()).toBe('alice');
  });
});
```

Tips and tricks

9. Wait in tests correctly

```
const askTheServer = async () => {
  const { data } = await axios.get('/endpoint');
};
```

The best is to just await the call:

```
it('waits for an ajax call', async () => {
  await askTheServer();
  expect(something).toBe('done');
});
```

waitForPromises helper to flush all pending promises:

```
function waitForPromises() {
  return new Promise((resolve) => {
    requestAnimationFrame(resolve);
  });
}
```

Tips and tricks

9. Wait in tests correctly

Wait until a vue component is re-rendered:

```
import { nextTick } from 'vue';

it('waits for vue to re-render the component', async () ⇒ {
  wrapper.setProps({ value: 'new value' });
  await nextTick();
  expect(wrapper.text()).toBe('new value');
});
```

Tips and tricks

10. Follow Jest best pratices

Prefer toBe over toEqual:

```
expect(1).toEqual(1); // ✗
expect(1).toBe(1); // ✓
```

Prefer more befitting matchers:

```
expect(arr.length).toBe(1); // ✗
expect(foo).toBe(undefined); // ✗
expect(arr).toHaveLength(1); // ✓
expect(foo).toBeUndefined(); // ✓
```

Be careful when using toBeTruthy or toBeFalsy:

```
expect(null).toBeFalsy(); // ✗
expect(null).toBe(false); // ✓
```

Be careful when using toBeDefined:

```
expect(wrapper.find('foo')).toBeDefined(); // ✗
expect(wrapper.find('foo').exists()).toBe(true); // ✓
```

Tips and tricks

1. Understand what to test and structure your test accordingly
2. Start with component factory
3. Use helpers to find elements and components
4. Do not test component internals
5. Follow the user
6. Query elements semantically
7. Treat child components as black boxes
8. Mock Vuex in the component
9. Wait in tests correctly
10. Follow Jest best pratices