

Testy jednostkowe w aplikacjach frontendowych!

Spis treści

Wprowadzenie.....	2
Środowisko	2
W skrócie - co testować, czego nie testować?	2
Gdzie umieścić testy? Jak je nazywać?	2
Jak zacząć	3
Przejdźmy do kodu!.....	4
Pierwszy test!.....	4
Dodajemy propsy	5
Testujemy V-Model	6
Dodajemy atrybut data do wyszukiwania elementów	6
Mockujemy!.....	7
Testujemy Pinię!.....	9
Sprawdzamy pokrycie, dopisujemy testy	9
Podsumowanie	11
Przydatne linki	11
Dziękuję za przeczytanie!	11

Wprowadzenie

Temat testowania jest dość obszerny, dlatego w tym dokumencie zostaną poruszone tylko podstawy testowania w kontekście aplikacji frontendowych. W związku z tym że pracowałem głównie z Order Process, skupiłem się na testowaniu komponentów Vue. Wykorzystałem do tego Vitest, który jest oparty na Jest.js i wszystkie przykładowe testy zamieszczone w tym dokumencie są oparte o te biblioteki. Podczas pisania tych testów posiłkowałem się dokumentacją obu bibliotek, i jako że jedna jest oparta o drugą to po zapoznaniu się z jedną nie powinno być również problemu z korzystaniem z drugiej. Do tego dokumentu została stworzona aplikacja ToDo List która została pokryta testami. Po kilku akapitach wstępu przejdę do omówienia przykładowych testów.

Środowisko

Vitest najprościej jest dodać na etapie tworzenia projektu Vite przez wybranie odpowiednich opcji w CLI. Jeżeli chcemy to zrobić manualnie, będziemy potrzebować takich pakietów jak: `@vue/test-util`, `jsdom` oraz `vitest`. Vitest dostosuje się do konfiguracji projektu przez odczytanie pliku konfiguracyjnego Vite. Więcej o konfiguracji vitest można znaleźć [tutaj](#).

W skrócie - co testować, czego nie testować?

Podczas robienia researchu napotkałem wiele podejść do testowania. Jeżeli mowa o testach jednostkowych, powinniśmy się skupić aby przetestować czy komponenty robią to do czego zostały stworzone ale nie w jaki sposób to robią. Przykładowo jeżeli mamy komponent wyświetlający dane na podstawie jakiegoś filtra, nie testujemy funkcji związanych z filtrowaniem, testujemy czy komponent wyświetlił przefiltrowane dane. Jeżeli komponent współpracuje z jakimś Pinia store, Pinia store powinien zostać zamockowany i powinno się przetestować czy komponent uruchamia odpowiednie metody. Pinia store powinien zostać przetestowany oddzielnie.

Gdzie umieścić testy? Jak je nazywać?

Jest parę podejść do tego gdzie należy umieszczać testy, nie ma jednego poprawnego, wszystkie posiadają różne wady i zalety. To jakie podejście zastosujemy w projekcie zależy głównie od preferencji zespołu. Podczas researchu spotkałem się najczęściej z dwoma podejściami. Jedno podejście zakłada aby pliki testowe znajdowały się obok plików komponentów. Drugie podejście zakłada aby utworzyć specjalny katalog `__tests__` w katalogu `src` i tam odwzorować strukturę projektu i umieszczać testy. Ja zdecydowałem się na ten drugi sposób ze względu na to że wolę aby kod aplikacji był oddzielony od testów, a przez odwzorowaną strukturę łatwo jest znaleźć pliki testowe dla konkretnych komponentów. Co do nazewnictwa, najczęściej pliki testowe posiadają suffix „spec” lub „test” np.: „component.spec.ts” lub „component.test.ts”.

Jak zacząć

Aby uruchomić testy wystarczy wpisać w konsoli po prostu „vitest”. Vitest uruchomi się, uruchomi wszystkie znalezione testy i będzie obserwował zmiany w kodzie umożliwiając nam Test Driven Development.

```
PROBLEMY DANE WYJŚCIOWE KONSOLA DEBUGOWANIA TERMINAL
> vitest

DEV v0.32.0 C:/Users/Norbert/Desktop/to-do-app-vitest
✓ src/_tests_/unit/components/UI/VCheckbox.spec.ts (2)
✓ src/_tests_/unit/views/ToDoListView.spec.ts (6)
✓ src/_tests_/integration/views/ListView.spec.ts (3)
✓ src/_tests_/unit/components/UI/VButton.spec.ts (3)
✓ src/_tests_/unit/components/UI/VToggle.spec.ts (3)
✓ src/_tests_/unit/components/ListItem.spec.ts (3)
✓ src/_tests_/unit/components/UI/VTextInput.spec.ts (2)
✓ src/_tests_/unit/components/Logo.spec.ts (1)
✓ src/_tests_/unit/stores/ItemsStore.spec.ts (4)

Test Files 9 passed (9)
Tests 27 passed (27)
Start at 12:14:18
Duration 4.55s (transform 341ms, setup 3ms, collect 1.84s, tests 545ms, environment 6.31s, prepare 1.58s)

Waiting for file changes...
press h to show help, press q to quit
```

Za każdym razem kiedy zmienimy coś w kodzie testu lub komponentu, Vitest uruchomi na nowo tylko pojedynczy plik testowy, bez testowania reszty komponentów, co jest szybkie i wygodne.

```
PROBLEMY DANE WYJŚCIOWE KONSOLA DEBUGOWANIA TERMINAL
RERUN src/_tests_/unit/components/Logo.spec.ts x1
✓ src/_tests_/unit/components/Logo.spec.ts (1)

Test Files 1 passed (1)
Tests 1 passed (1)
Start at 12:16:17
Duration 127ms

Waiting for file changes...
press h to show help, press q to quit
```

Jeżeli któryś z testów zawiedzie otrzymamy raport:

```
PROBLEMY DANE WYJŚCIOWE KONSOLA DEBUGOWANIA TERMINAL
DEV v0.32.0 C:/Users/Norbert/Desktop/to-do-app-vitest
✓ _tests_/unit/stores/ItemsStore.spec.ts (4)
✓ _tests_/unit/components/UI/VToggle.spec.ts (3)
> _tests_/unit/views/ToDoListView.spec.ts (6)
  > ToDoList (6)
    ✓ Should render
    ✓ Should have items visible in the list
    ✗ Should have addItem method called with right arguments
    ✓ Form shouldn't call addItem method in pinia store if text input is empty
    ✓ Should have removeItem and setIsDone triggered by list item
    ✓ Should react on filters
✓ _tests_/unit/components/ListItem.spec.ts (3)
✓ _tests_/unit/components/UI/VButton.spec.ts (3)
> _tests_/unit/components/UI/VTextInput.spec.ts (2)
  > VTextInput (2)
    ✓ Should render
    ✗ Should update v-model
✓ _tests_/unit/components/UI/VCheckbox.spec.ts (2)
✓ _tests_/unit/components/Logo.spec.ts (1)

Failed Tests 2

FAIL _tests_/unit/views/ToDoListView.spec.ts > ToDoList > Should have addItem method called with right arguments
AssertionError: expected "spy" to be called at least once
   52 |     const calls = vi.mocked(useToDoStore()).addItem.mock.calls;
   53 |     expect(vi.mocked(useToDoStore()).addItem).toHaveBeenCalled();
   54 |     expect(calls[0][0]).toEqual({ value: testValue, isDone: false });
     |                               ^
   55 |   });
   56 |

[1/2]-

FAIL _tests_/unit/components/UI/VTextInput.spec.ts > VTextInput > Should update v-model
AssertionError: expected 'initialText' to be 'test' // Object.is equality
- Expected - 0
+ Received + 1

- 'test'
+ 'initialText'

> _tests_/unit/components/UI/VTextInput.spec.ts:28:41

[2/2]-

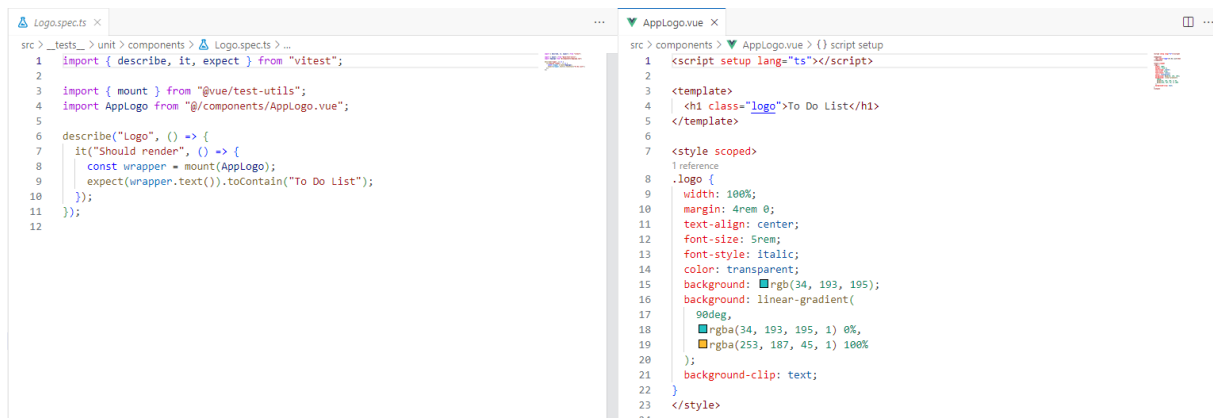
Test Files 2 failed | 6 passed (8)
Tests 2 failed | 22 passed (24)
Start at 12:26:05
Duration 4.04s (transform 285ms, setup 1ms, collect 1.43s, tests 490ms, environment 4.63s, prepare 1.18s)
```

Przejdźmy do kodu!

Stworzone przeze mnie repozytorium z kodem można znaleźć [tutaj](#)! Od tej pory stopniowo będę omawiał napisane przeze mnie testy od najprostszych do najbardziej skomplikowanych, przy okazji opisując i tłumacząc co za co odpowiada. Na koniec podam linki do dokumentacji niektórych elementów.

Pierwszy test!

Pierwszy najprostszy test sprawdzający czy komponent został wyrenderowany.



Jak widać komponent to prosty nagłówek z napisem To Do List. Aby przetestować czy komponent został wyrenderowany musimy zaimportować kolejno: describe, it, expect z „vitest” oraz mount z „vue test utils” i oczywiście sam komponent.

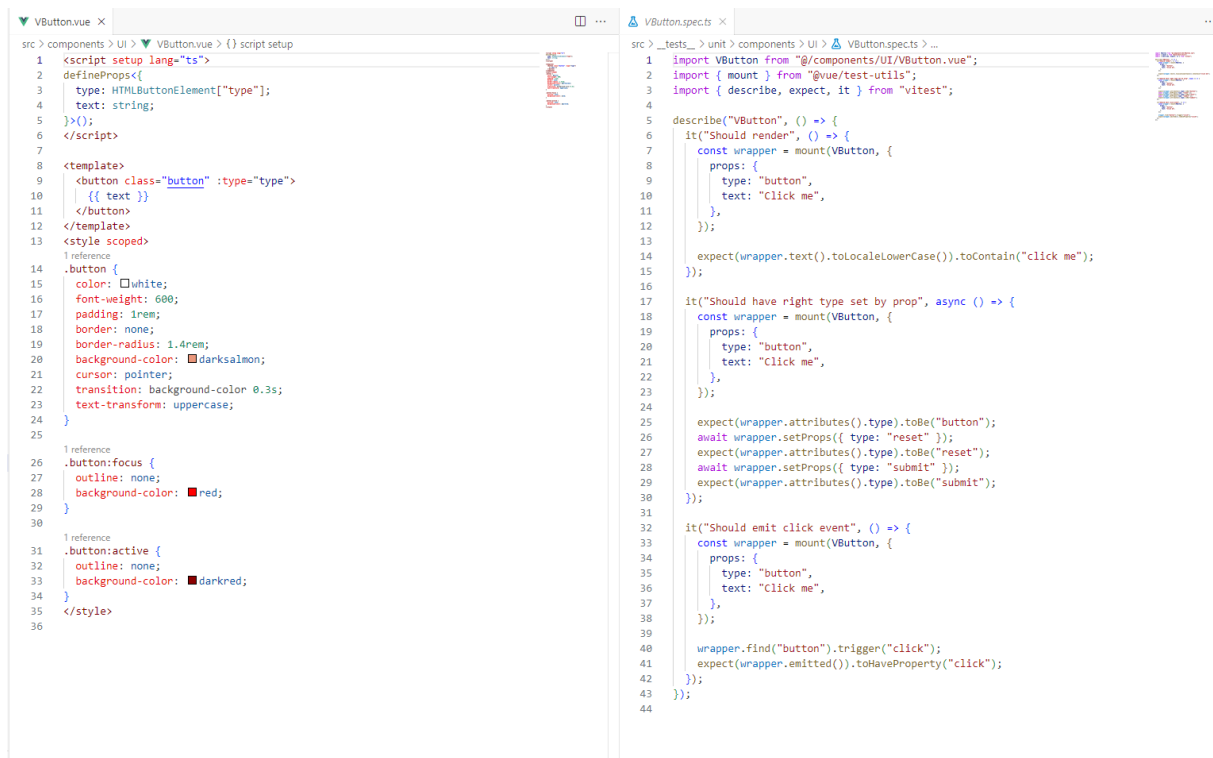
Funkcja „describe” służy głównie do grupowania i organizowania testów. Pierwszy argument tej funkcji to po prostu opis tego co testujemy. W powyższym przypadku jest to komponent „Logo”. Drugi argument to funkcja zawierająca już testy jednostkowe.

Funkcja „it” definiuje konkretny test. Pierwszy argument to opis mówiący co konkretnie sprawdza test. W tym przypadku „Should render”. Drugi argument to funkcja testująca.

Funkcja „mount” montuje komponent do zmiennej „wrapper”. Za pomocą metody „wrapper.text()” wyciągamy z zamontowanego komponentu widoczny w nim tekst. [Całą dokumentację wrappera można znaleźć tutaj](#).

Funkcja „expect” służy do sprawdzania czy warunki zostały spełnione. Jest bardzo wiele operatorów porównania a w tym przypadku został wykorzystany operator „toContain” który sprawdza czy dany string jest substringiem „wrapper.text()”. Jest dużo operatorów porównania. [Wszystkie można znaleźć w dokumentacji Vitest](#). Kolejne zostaną poruszone podczas omawiania kolejnych testów.

Dodajemy propsy



```
src > components > UI > VButton.vue > {} script setup
1 <script setup lang="ts">
2   defineProps<
3     type: HTMLButtonElement["type"];
4     text: string;
5   >();
6 </script>
7
8 <template>
9   <button class="button" :type="type">
10     {{ text }}
11   </button>
12 </template>
13 <style scoped>
14   1 reference
15   .button {
16     color: white;
17     font-weight: 600;
18     padding: 1rem;
19     border: none;
20     border-radius: 1.4rem;
21     background-color: darksalmon;
22     cursor: pointer;
23     transition: background-color 0.3s;
24     text-transform: uppercase;
25   }
26   1 reference
27   .button:focus {
28     outline: none;
29     background-color: red;
30   }
31   1 reference
32   .button:active {
33     outline: none;
34     background-color: darkred;
35   }
36 </style>
37
src > _tests_ > unit > components > UI > VButton.specs > ...
1 import VButton from "@components/UI/VButton.vue";
2 import { mount } from "@vue/test-utils";
3 import { describe, expect, it } from "vitest";
4
5 describe("VButton", () => {
6   it("Should render", () => {
7     const wrapper = mount(VButton, {
8       props: {
9         type: "button",
10         text: "click me",
11       },
12     });
13     expect(wrapper.text().toLocaleLowerCase()).toContain("click me");
14   });
15
16   it("Should have right type set by prop", async () => {
17     const wrapper = mount(VButton, {
18       props: {
19         type: "button",
20         text: "click me",
21       },
22     });
23
24     expect(wrapper.attributes().type).toBe("button");
25     await wrapper.setProps({ type: "reset" });
26     expect(wrapper.attributes().type).toBe("reset");
27     await wrapper.setProps({ type: "submit" });
28     expect(wrapper.attributes().type).toBe("submit");
29   });
30
31   it("Should emit click event", () => {
32     const wrapper = mount(VButton, {
33       props: {
34         type: "button",
35         text: "click me",
36       },
37     });
38
39     wrapper.find("button").trigger("click");
40     expect(wrapper.emitted()).toHaveLength(1);
41   });
42 });
43
44
```

Drugim komponentem jest prosty przycisk. Jak widać do komponentu możemy przekazać props podczas montowania.

Pierwszy test sprawdza czy komponent się renderuje.

Drugi test sprawdza czy komponent reaguje na props i czy przycisk otrzymuje odpowiedni typ. Wrapper posiada metodę „attributes()” która zwraca atrybuty komponentu/elementu. Dzięki czemu za pomocą matchera „toBe” możemy sprawdzić czy atrybut typ posiada wartość „button”. Matcher „toBe” służy do porównywania prymitywnych wartości. Jeżeli chcemy porównać dwa obiekty, czy posiadają tę samą strukturę, powinniśmy skorzystać z matchera „toEqual”. Wrapper posiada też asynchroniczną metodę „setProps” która pozwala zmieniać props. Ważne jest aby zawsze korzystać z tej metody z wykorzystaniem operatora „await” ponieważ w innym wypadku testy zostaną wykonane zanim property zostanie zmienione w komponencie.

Trzeci test sprawdza czy przycisk emituje „click event”. Używając metody „trigger” jesteśmy w stanie wykonywać akcje na elemencie. W tym wypadku została wykonana akcja „click”. Aby sprawdzić czy komponent wyemitował event używamy metody „emitted()”. Zwróci to obiekt w którym możemy zobaczyć jakie eventy zostały wyemitowane przez komponent. Aby sprawdzić czy „click event” znajduje się w obiekcie używamy matchera „toHaveLength” który sprawdza czy obiekt posiada klucz „click”.

W tych testach została też wykorzystana metoda „find” do znalezienia konkretnego elementu w komponencie. Metoda „find” wykorzystuje składnię metody „querySelector”.

Testujemy V-Model

```
src > components > UI > VTextInput.vue > {} script setup
1 <script setup lang="ts">
2 interface IProps {
3   modelValue: string;
4   placeholder?: string;
5 }
6
7 interface IEmits {
8   (e: "update:modelValue", value: string): void;
9 }
10
11 defineProps<IProps>();
12 const emit = defineEmits<IEmits>();
13 </script>
14
15 <template>
16 <input
17   class="text-input"
18   :placeholder="placeholder"
19   :value="modelValue"
20   @input="(e) => emit('update:modelValue', (e.target as HTMLInputElement).value)"
21   type="text"
22 />
23 </template>

src > _tests_ > unit > components > UI > VTextInput.specs > ...
17
18 it("Should update v-model", async () => {
19   const wrapper = mount(VTextInput, {
20     props: {
21       modelValue: "initialText",
22       "onUpdate:modelValue": (e: string) => {
23         wrapper.setProps({ modelValue: e });
24       },
25     });
26
27   await wrapper.find("input").setValue("test");
28   expect(wrapper.props("modelValue")).toBe("test");
29 });
30
31
```

Na powyższym teście widzimy komponent „TextInput” który jest prostym polem tekstowym. Test sprawdza czy komponent wpisuje dane do v-model. Jako props przekazujemy modelValue z przykładową wartością oraz po przecinku funkcję która mówi co ma się dzieć kiedy wartość modelValue się zmienia. Za pomocą metody „setValue” ustawiamy wpisujemy wartość do pola tekstowego i sprawdzamy czy dane propsa „modelValue” zostały uaktualnione.

Dodawamy atrybut data do wyszukiwania elementów

W bardziej złożonych komponentach może wystąpić sytuacja że ciężko nam będzie znaleźć jakiś element. Częstą praktyką jest dodawanie atrybutu data-testid do elementu żeby łatwiej było go odszukać przy pomocy metody „find”. Przykładowy test wykorzystujący tą praktykę.

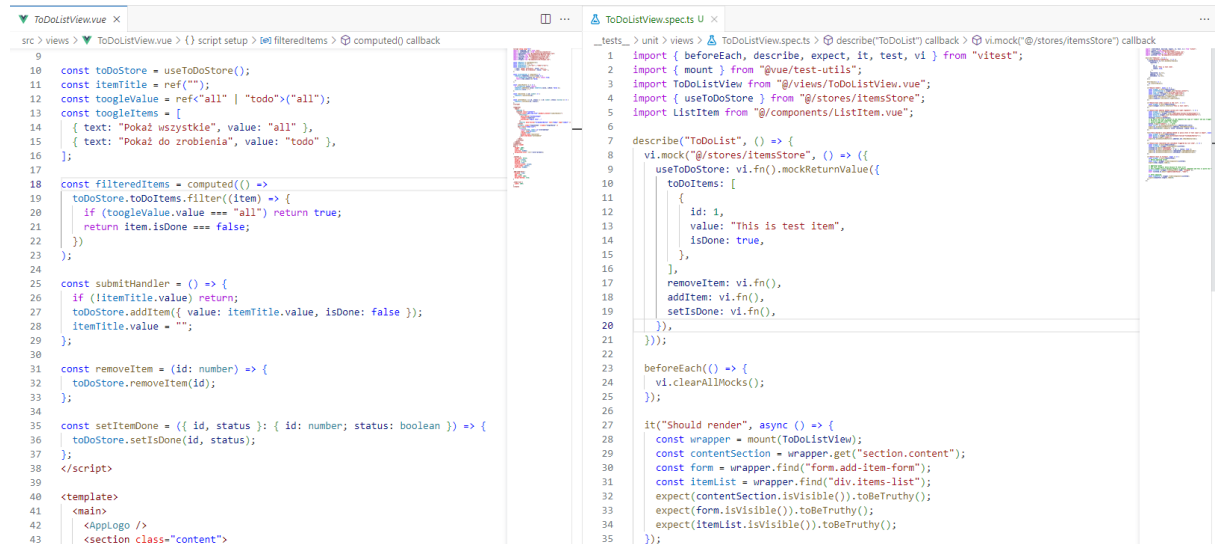
```
src > components > UI > VToggle.vue > {} script setup
1 <script setup lang="ts" generic="T">
2   defineProps<{
3     modelValue: T;
4     items: Array<{ text: string; value: T }>;
5   }>();
6
7   const emit = defineEmits<{
8     (e: "update:modelValue", value: T): void;
9   }>();
10 </script>
11
12 <template>
13   <div class="toggle">
14     <button
15       v-for="(item, i) in items"
16       :class="{
17         'toggle-button',
18         modelValue === item.value && 'toggle-button--active',
19       }"
20       :key="`btn-${i}`"
21       :data-testid="`button-${item.value}`"
22       @click="emit('update:modelValue', item.value)"
23     >{{ item.text }}
24   </button>
25 </div>
26 </template>

src > _tests_ > unit > components > UI > VToggle.specs > ...
39
40 it("Should have only one active button with active class", async () => {
41   const wrapper = mount(VToggle, {
42     props: {
43       modelValue: "value",
44       "onUpdate:modelValue": (e: string) => {
45         wrapper.setProps({ modelValue: e });
46       },
47       items: [
48         { text: "Test", value: "value" },
49         { text: "Test2", value: "value2" },
50       ],
51     });
52
53   const firstButton = wrapper.find('button[data-testid="button-value"]');
54   const secondButton = wrapper.find('button[data-testid="button-value2"]');
55
56   expect(firstButton.classes()).toContain("toggle-button--active");
57   await secondButton.trigger("click");
58
59   expect(secondButton.classes()).toContain("toggle-button--active");
60
61   const activeEl = wrapper.findAll("button.toggle-button--active");
62   expect(activeEl.length).toBe(1);
63 });
64
```

Test sprawdza czy komponent posiada tylko jeden aktywny przycisk. Jak widać komponent posiada data-testid dla każdego przycisku. Za pomocą metody „find” odnajdujemy przyciski, następnie sprawdzamy czy aktywny przycisk posiada klasę „active”. Następnie wykonywana jest akcja „click” na drugim przycisku po czym sprawdzamy czy drugi przycisk posiada klasę „active” i czy jest tylko jeden element z tą klasą.

Mockujemy!

W bardziej skomplikowanych komponentach, będziemy musieli zamockować część logiki czy np. zapytania do api. W tym celu użyjemy „[vi](#)”. Vi dostarcza wiele różnych narzędzi do mockowania, w przykładzie została wykorzystana metoda „mock”.



Jak widać na powyższym screenie w widoku `ToDoListView` używamy `todoStore`, i odwołujemy się do niego w różnych miejscach w komponencie.

Aby zamockować `todoStore`, w bloku „describe” wykorzystujemy „[vi.mock](#)”. Jako pierwszy argument przekazujemy ścieżkę do mockowanego pliku, a w drugim argumentcie zwracamy obiekt który odzwierciedla to co zwraca Pinia store. Mimo wszystko nadal jest tu zaimportowany „`useToDoStore`” który będzie potrzebny później. Tutaj została też wykorzystana metoda „[vi.fn\(\)](#)” która tworzy „funkcję szpiega” która pozwala nam sprawdzić czy została wywołana, ile razy, z jakimi argumentami itd.

Oprócz tego pojawia się jeszcze hook „beforeEach”, który uruchamia kod przed każdym pojedynczym testem. „clearAllMocks” jak nazwa wskazuje, czyści stan naszych mocków. Do dyspozycji mamy jeszcze hooki „afterEach, beforeAll, afterAll”. „afterEach” uruchamia kod po każdym teście, „beforeAll” i „afterAll” uruchamiają kod przed wszystkimi testami i po wszystkich testach.

Na poniższym screenie widać część testów wioku ToDoListView.

```
ToDoListView.spectrs 3, U x
__tests__ > unit > views > ToDoListView.spectrs > ...
42 it("Should have addItem method called with right arguments", () => {
43   const wrapper = mount(ToDoListView);
44   const textInput = wrapper.find('input[data-testid="formTextInput"]');
45   const button = wrapper.find('button[data-testid="formSubmitButton"]');
46   const testValue = "value";
47   textInput.setValue(testValue);
48   // Be aware that even if button in the template has type of "submit" and you trigger "click" event on element
49   // Form will not get submitted! Example:
50   // button.trigger("click") - not work!
51   button.trigger("submit"); // - work!
52   const calls = vi.mocked(useToDoStore()).addItem.mock.calls;
53   expect(vi.mocked(useToDoStore()).addItem).toHaveBeenCalled();
54   expect(calls[0][0]).toEqual({ value: testValue, isDone: false });
55 });
56
57 test("Form shouldn't call addItem method in pinia store if text input is empty", async () => {
58   const wrapper = mount(ToDoListView);
59   const button = wrapper.find('button[data-testid="formSubmitButton"]');
60   await button.trigger("submit");
61   expect(vi.mocked(useToDoStore()).addItem).not.toHaveBeenCalled();
62 });
63
64 it("Should have removeItem and setIsDone triggered by list item", () => {
65   const wrapper = mount(ToDoListView);
66   const listItem = wrapper.findComponent(ListItem);
67   listItem.vm.$emit("removeItem", 1);
68   listItem.vm.$emit("setItemDone", { id: 1, status: true });
69   expect(vi.mocked(useToDoStore()).removeItem).toHaveBeenCalled();
70   expect(vi.mocked(useToDoStore()).setIsDone).toHaveBeenCalled();
71 });
72
73 it("Should react on filters", async () => {
74   const wrapper = mount(ToDoListView);
75   // Before applying filter
76   const items = wrapper.findAllComponents(ListItem);
77   expect(items.length).toBe(1);
78
79   // Applying filter
80   // Not just (Vtoogle) below because TS shows error
81   // This probably occurs because Vtoogle is generic component and this is quite new feature in Vue, can be not completely supported!
82   const listItem = wrapper.findComponent({ name: "Vtoogle" });
83   await listItem.vm.$emit("update:modelValue", "todo");
84
85   // After applying
86   const itemsAfter = wrapper.findAllComponents(ListItem);
87   expect(itemsAfter.length).toBe(0);
88 });
89 });
```

Większość widocznego kodu była już omówiona ale można tutaj zauważyć wykorzystanie „[vi.mocked](#)”. Dzięki `vi.mocked` możemy wyciągnąć informację z naszego mocka. Jako argument musimy przekazać mockowany element i właśnie dlatego „`useToDoStore`” wciąż musi być zaimportowany.

Skupiając się na drugiej części pierwszego testu. Za pomocą „`vi.mocked`” wyciągamy informacje o funkcji „`addItem`” która została również zamockowana przy użyciu „`vi.fn()`”. W związku z tym mamy dostęp do pewnych informacji. Matcher „`toHaveBeenCalled()`” sprawdza czy funkcja została wywołana. Wyciągamy też tablicę „`calls`” która posiada informację na temat wywołań. Za pomocą matchera „`toEqual`” sprawdzamy czy obiekt który został przekazany jako argument funkcji ma odpowiednią zawartość.

Drugi test jest w bloku „`test`” zamiast „`it`”. Jest to praktycznie to samo i może być używane przemiennie. „`It`” nie do końca pasowało do opisu testu który napisałem więc wykorzystałem „`test`”. W tym teście można też zauważyć „`expect().not.HaveBeenCalled()`”. [Not](#) po prostu odwraca testowany warunek.

W trzecim teście można zauważyć że po odnalezieniu konkretnego komponentu we wrapperze możemy emitować z niego zdarzenia. Reszta sprawdza czy odpowiednie akcje zostały wywołane w Pinia store.

Ostatni test sprawdza czy działa opcja filtrowania. Jak widać nie testujemy logiki która odpowiada za testowanie, testujemy tylko czy filtr zmienił ilość elementów.

Testujemy Pinię!

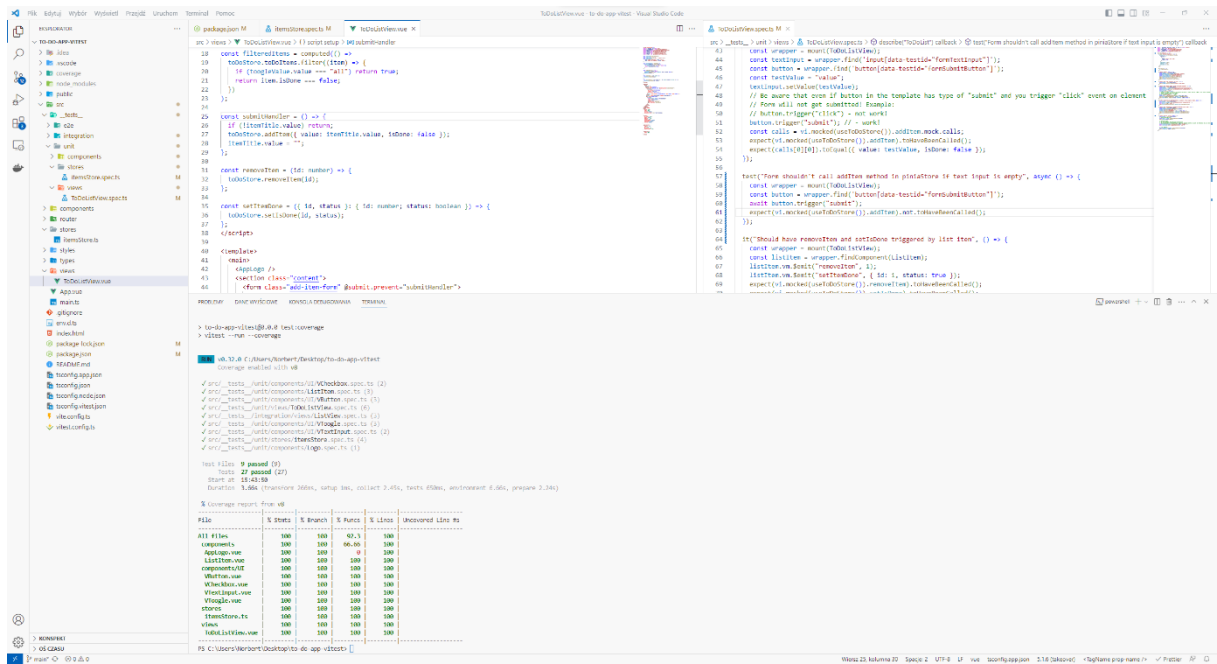
```
itemsStore.spec.ts x
src > __tests__ > unit > stores > itemsStore.spec.ts > ...
1 import { useToDoStore } from "@stores/itemsStore";
2 import { setActivePinia, createPinia } from "pinia";
3 import { describe, it, expect } from "vitest";
4
5 // Testing data store
6 describe("ToDoItem store", () => {
7   // Pinia store can't work without an instance
8   // So we need to create pinia instance
9   setActivePinia(createPinia());
10  const toDoStore = useToDoStore();
11  const testItemData = { value: "Test item", isDone: false };
12  let testItemId: number;
13
14  it("Should have method to add item to the store", () => {
15    testItemId = toDoStore.addItem(testItemData);
16    expect(toDoStore.toDoItems.length).toBe(1);
17  });
18
19  it("Should have method to modify isDone property in stored item", () => {
20    toDoStore.setIsDone(testItemId, true);
21    const item = toDoStore.toDoItems.find((i) => i.id === testItemId);
22    expect(item?.isDone).toBe(true);
23  });
24
25  it("Should not crash if isDone property is set on nonexisting item", () => {
26    expect(() => toDoStore.setIsDone(1, true)).not.toThrow();
27  });
28
29  it("Should have method to remove the item", () => {
30    toDoStore.removeItem(testItemId);
31    expect(toDoStore.toDoItems.length).toBe(0);
32  });
33 });
34
```

Pinia store nie może działać bez aplikacji, dlatego musimy stworzyć instancję Pinii. Jak widać na powyższym screenie importujemy `setActivePinia` i `createPinia` oraz tworzymy instancję Pinii. Reszta to proste testy sprawdzające logikę pinia store.

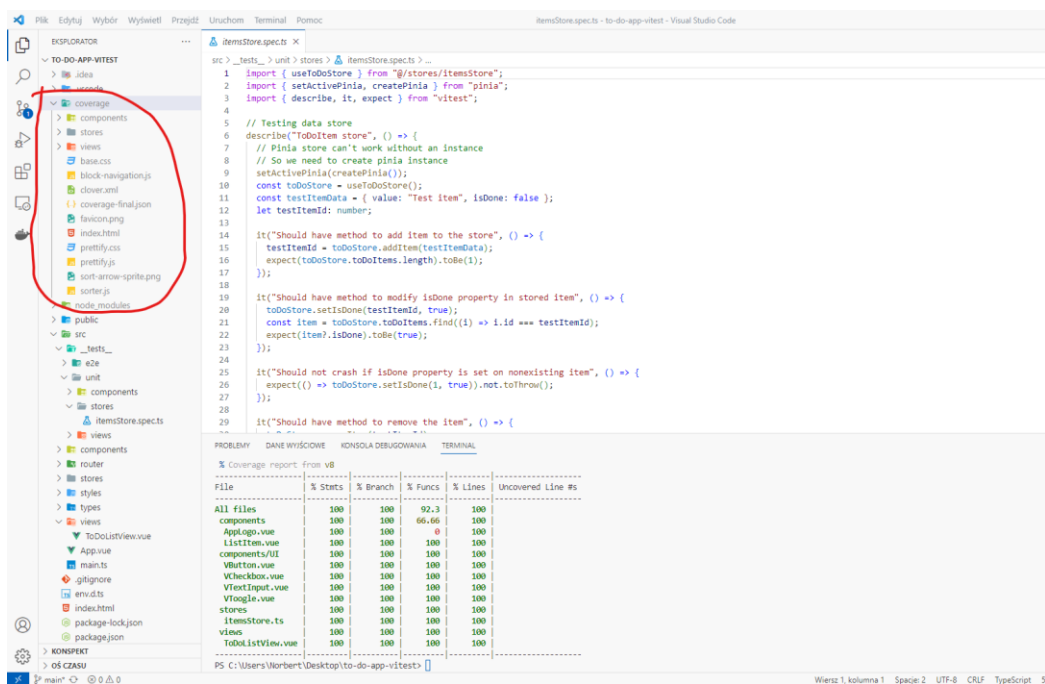
Sprawdzamy pokrycie, dopisujemy testy

[illegible]

Vitest oraz Jest.js posiadają opcje wygenerowania raportu pokrycia kodu testami, co może też dać nam wskazówkę czego nie przetestowaliśmy. Po uruchomieniu vitest z flagą coverage otrzymamy mniej więcej taki raport. Jak widać na screenie wyżej, „ToDoListView” nie posiada testu pokrywającego kod w linii 26. Zapomniałem napisać testu, który sprawdzi co się stanie jeżeli „todo item” nie będzie miał żadnej zawartości. Po dopisaniu odpowiedniego testu, który jest widoczny w pliku „spec”, linia 57, coverage ma już 100%.



Flaga coverage wygeneruje nam również plik HTML do podglądu raportu z testów.



All files

100% Statements

356/356

100% Branches

36/36

92.3% Functions

15/15

100% Lines

356/356

Press n or j to go to the next uncovered block, b, p or k for the previous block.

Filter

File	Statements	Branches	Functions	Lines
components	100%72/72	100%72/72	2/266.66%	2/3100%72/72
components/ui	100%135/135	100%135/135	3/3100%	2/2100%135/135
stores	100%30/30	100%30/30	7/7100%	3/3100%30/30
views	100%93/93	100%93/93	12/12100%	5/5100%93/93

Podsumowanie

Jak napisałem wcześniej, temat testowania jest dosyć obszerny. Ciężko jest poznać wszystkie matchery w krótkim czasie, a tym bardziej ciężko by je było tutaj opisać. Tak samo z mockowaniem. „Vi” ma tyle możliwości że np. mock w rozdziale „Mockujemy!” na pewno można zrobić na inne sposoby. Na pewno nie wszystko zostało tutaj poruszone ale mimo wszystko mam nadzieję że wiedza zawarta w tym pliku to jakaś baza do szybkiego startu w pisaniu testów. W razie szukania jakichś rozwiązań w internecie, polecam szukanie z hasłami Vitest oraz Jest, oraz sprawdzenie dokumentacji obu bibliotek, ponieważ są one podobne i mogą podsunąć rozwiązanie.

W katalogu `__test__/integration/views/` jest jeden przykładowy test integracyjny. Który sprawdza integrację `ToDoView` z `ToDoStore`. Warto również na niego spojrzeć.

Przydatne linki

Dorzucam parę przydatnych linków które mogą się przydać. Zanim jeszcze zacząłem tworzyć to do listę, napisałem parę podstawowych testów. Można jest znaleźć w tym branchu: „[jest-basics](#)”. Jest tam bardzo mało kodu ale np. jest test sprawdzający Timeout co też może się przydać.

Inne linki:

- Testowanie pinia: <https://pinia.vuejs.org/cookbook/testing.html>
- 40 minutowy film na YouTube o pisaniu testów dla Vue: <https://www.youtube.com/watch?v=FJRuG85tXV0>

Jest:

- Dokumentacja: <https://jestjs.io/docs/getting-started>

Vitest:

- Mock: <https://vitest.dev/api/mock.html>
- Vi Utility: <https://vitest.dev/api/vi.html>
- Expect i Matchery: <https://vitest.dev/api/expect.html>

Vue test utils:

- Api: <https://v1.test-utils.vuejs.org/api/>
- Wrapper: <https://v1.test-utils.vuejs.org/api/wrapper/>

Vue Mastery:

- Unit testing: <https://www.vuemastery.com/courses/unit-testing-vue-3/what-to-test-vue-3/>

Polecam też przejrzeć Vue Mastery, był tam albo jeszcze jeden albo dwa kursy odnośnie testowania. Przypominam że login i hasło do Vue Mastery znajduje się na Notion :)

Dziękuję za przeczytanie!

Norbert Bednarczyk dla Riotters, 15.08.2023.