後悔しないための

# Vue

## コンポーネント設計

nakajmg著

# 後悔しないための Vue コンポーネント設計

nakajmg　著

2018-10-08 技術書典 5 版　　PonyHead 発行

# はじめに

## 本書について

本書は、筆者が業務および個人プロジェクトで得た Vue.js を使う際の知見を文書化したものです。扱っている内容は主に次のようなものになります。

- コンポーネントの設計およびプロジェクトのディレクトリ構成
- テストしやすい/しづらいコンポーネントとは
- コンポーネントの何をテストするか
- 単体テストの書き方

## 対象読者

本書が想定する主な対象読者は、Vue.js を使ってる、もしくは Vue.js を使ってシングルページアプリケーションを作成したいと考えている方になります。中でも、次のような方はとくに学びを得られる部分があるかもしれません。

- コンポーネントの設計に自信がない方
- コンポーネントの分類で悩んでる方
- コンポーネントのアンチパターンを知りたい方
- テストの書き方がわからない方

## 動作環境など

本書の内容は、次の環境での動作を前提としています。

- OS: macOS High Sierra
- Vue CLI: v3.0.1
- Vue.js: v2.5.17

- vue-router": v3.0.1
- vuex: v3.0.1
- Node.js: v10.7.0
- npm: v6.1.0

本書で紹介するコードは、.vueによるシングルファイルコンポーネントでの開発を想定しています。

本書で紹介しているコードは、次のリポジトリから自由にダウンロード/clone して使えます。

https://github.com/nakajmg/testable-vue-component-sample

## 免責事項

本書で使われている**テスト**という単語は、コンポーネントの単体テスト（ユニットテスト）および、コンポーネント同士の結合テスト（インテグレーションテスト）を指しています。E2E テストやリグレッションテストについては触れていませんのでご了承ください。

また、本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について、著者はいかなる責任も負いません。

# 目次

# 第 1 章

# なぜテストを書くのか

「なぜテストを書くのか」「なぜテストが必要なのか」「テストの有用性」などなど、テストの必要性を説く文章は検索すればいくらでも出てくるでしょう。本書では「なぜテストが必要なのか」といった一般化したテーマの内容は書きません。そういったものが読みたい方は、検索してください。

この章では筆者である「私」がなぜテストを書くようになったかを綴っています。興味がなければ次の章へと進んでください。

## 1.1　なぜ「私」はテストを書くようになったのか

筆者が Vue.js を使いだしたのは v0.10 のころからで、以来 4 年以上に渡って大小さまざまなプロジェクトで使ってきました。しかしながら、テストに真剣に向き合うようになったのはここ 1 年以内くらいのことです。

きっかけは Backbone.js で組まれた既存の SPA を、Vue.js で作り直す案件での経験です。この案件は仕様書の更新がされておらず、**既存アプリの動作=仕様**になっているような状態でした。

この案件では、ベースの実装をほぼ全て筆者が行いました。CSS は現行のやつをそのまま使う、という判断ミスもありますが、複雑怪奇な CSS に引きづられ、コンポーネントの分割がかなり雑になり、テストも書きませんでした。なんとか要件を満たすように実装が完了したこの案件は、運用フェイズへと入りました。

運用フェイズが数ヶ月したころ、筆者が他の新規案件を担当することになり、同僚が運

用を代わることになりました。

そのときになって改めてコードを見てみると、コンポーネントにべったりと張り付いたさまざまな依存、めちゃくちゃな Store と無数の getters、流れがわかりづらい初期化処理などなど、とても人に引き渡せるようなコードではありませんでした。

テストがない状況で引き継がれた当人にとっては、小さなバグの修正だとしても、何が正しい動作で、どこに影響が出るかを把握するのは難しいでしょう。これは通常の JavaScript ではなく、Vue や React のような UI コンポーネントではなおさらです。

せめてテストだけでも書いておけば、しっかりコンポーネントを分割できる判断ができればと、申し訳なさと後悔を覚えました。

### 1.1.1 他人に迷惑をかけるコードを書きたくない

テストを書けば全てが解決するとはまったく思いませんが、テストがあるだけで解決できるような問題や状況はあると思います。

せめて自分が書いたコードで他人に迷惑をかけないように、テストを書いていこうと。そしてコンポーネントをよりよい状態に分割できる知識を身に着けようと思いました。

ここでいう**他人**とは、自分以外だけでなく、自分も含まれています。書いたあと 1 ヶ月も経てば、書いたコードのことは覚えてないことがほとんどです。テストを書かずに動くものが作れたとしても、それは未来の自分への借金、もしくは他の人へとコストを押し付けてることにしかならないと筆者は考えています。

# 第 2 章

# テストしやすいコンポーネントと、テストしづらいコンポーネント

この章では、どういったコンポーネントがテストしやすく、どういったコンポーネントがテストしづらいのかについて、筆者の経験からの考えを紹介します。

## 2.1 テストしやすい/しづらいコンポーネント

筆者が考えるテストしやすいコンポーネントは、次のようなコンポーネントです。

- 機能が少ない
- 依存がない/少ない
- 状態をもたない

また、これらの項目の１つでも逆をいくコンポーネントは、テストしづらいといえます。

これらの項目が、どうテストのしやすさと結びつくのかについて、解説します。

## 2.2 機能を少なくシンプルに保つ

コンポーネントの機能が少なければ少ないほど、テストはしやすいです。ここでいう**機能**とは、methodsの数ではなく、コンポーネントが担う役割のことです。

propsで受け取った値を表示するだけでなく、dataで自身の状態を操作したり、レンダリングのために computedで複雑な計算をいくつもしたり、このようなコンポーネントは**機能**が多く、テストしづらいといえます。

もしコンポーネントの機能が多いなと感じたら、それはコンポーネントの役割がうまく分割できていないサインかもしれません。役割を見極めて、ちょうどいい粒度でコンポーネントを分割できると、アプリケーションの構造としても、テストのためにもよいです。

## 2.3 依存は少なく

コンポーネントが依存してるものが少なければ少ないほど、テストはしやすいです。

依存の数は、そのコンポーネントの再利用性にも関わってきます。たとえば、Vuex の Store や、Router にべったり依存したようなコンポーネントは、他の場所や用途で使うのは難しいでしょう。

▼リスト 2.1　依存の多いコンポーネント

```
<template>
  <div>
    <div
      v-for="item in items"
      :key="item.id"
    >
      <span>{{item.label}}</span>
      <button @click="clickItem(item)">
        {{item.label}}
      </button>
    </div>
  </div>
</template>

<script>
import { mapState } from "vuex"
export default {
  name: "BigComponent",
  computed: {
    ...mapState(["items"]),
  },
  methods: {
    clickItem(item) {
      this.$router.push({
        name: "itemDetail",
        params: {
          name: item.name,
          id: item.id,
        },
      })
    },
  },
}
</script>
```

依存を少なくするには、コンポーネントの役割を分割することが大事になります。たとえばリスト 2.1 例では、Store から値を取り出すコンポーネントと、値を表示するコンポーネントに分けることで、値を表示するコンポーネントは他の場所でも使えるようになるでしょう。

▼リスト 2.2　Store から値を取り出すコンポーネント: Container.vue

```
<template>
  <div>
    <Item
      v-for="item in items"
      :key="item.id"
      v-bind="item"
      @clickItem="onClickItem"
    />
  </div>
</template>

<script>
import { mapState } from "vuex"
export default {
  name: "BigComponent",
  computed: {
    ...mapState(["items"]),
  },
  methods: {
    onClickItem(item) {
      this.$router.push({
        name: "itemDetail",
        params: {
          name: item.name,
          id: item.id,
        },
      })
    },
  },
}
</script>
```

▼リスト 2.3　値を表示するコンポーネント: Item.vue

```
<template>
  <div>
    <span>{{label}}</span>
    <button @click="clickItem">
      {{label}}
    </button>
```

```
    </div>
  </template>

  <script>
  export default {
    name: "Item",
    proops: {
      id: Number,
      name: String,
      label: String,
    },
    methods: {
      clickItem() {
        this.$emit({
          id: this.id,
          name: this.name,
        })
      },
    },
  }
  </script>
```

このように、コンポーネントはできるだけ依存の少ない状態にしておくと、変更/修正がしやすく、テストもしやすくなります。

コンポーネントの分割については第 3 章「コンポーネントを分類する」と第 4 章「ディレクトリ構成とコンポーネントの分類」にて解説します。

## 2.4 なるべく状態を持たせない

状態を持たないコンポーネントはテストしやすいです。逆をいうと、状態を持っているコンポーネントはテストしづらいです。

**状態をもっているコンポーネント**とは、dataを利用しているコンポーネントや、propsの特定の値によって振る舞いが大きく変わるようなコンポーネントを指します。後者のような、値によって振る舞いが変わるような状況は**状態を作り込んでいる**と表現されることもあります。

### 2.4.1 data を使うパターン

筆者はコンポーネントでの dataの利用を最小限にするように努めています。筆者が思う dataを使うのに適した状況は、アプリケーション全体の振る舞いに関与しない、その

コンポーネントに閉じた状態を扱う場合だと考えています。

たとえば、次のようなクリックしたら展開されるメニューを制御するときは、状態がコンポーネントに閉じていて、アプリケーションには影響を与えません。

▼リスト 2.4　コンポーネントに閉じた状態

```
<template>
  <nav>
    <div v-show="menuOpened">
      <router-link to="/">Root</router-link>
      <router-link to="/foo">Foo</router-link>
      <router-link to="/bar">Bar</router-link>
    </div>
    <button @click="toggleMenu">
      Open Menu
    </button>
  </nav>
</template>

<script>
export default {
  name: "Menu",
  data() {
    return {
      menuOpened: false,
    }
  },
  methods: {
    toggleMenu() {
      this.menuOpened = !this.menuOpened
    },
  },
}
</script>
```

リスト 2.4 のようなパターン以外に、フォームの入力項目なども dataに格納します。フォームの入力は一時的な入力で、コンポーネントに閉じた状態ですので、dataでもつのが適しているでしょう。

コンポーネントの状態を絶対の悪として、すべての値を Vuex Store に格納するような使い方をしてる方がたまに見受けられますが、コンポーネントに閉じた状態までも制限するような使い方は、無用な複雑性を生み出す結果に繋がりかねません。適材適所を見極めて、dataとうまく付き合っていくのが、状態管理疲れしないためにも必要かと思います。

### 2.4.2　状態を作り込まない

propsで状態を作り込んでしまうパターンとしてよくあるのは、propsで渡された mod

eや typeといったプロパティの値によって、そのコンポーネントの動作が大きく変わるものが挙げられます。このような作りにしてしまうと、コンポーネント内部で if (mode === "edit") {...}や ... といった、条件によって変わるものが増えていき、コンポーネントが複雑なものになります。

このような、状態を作り込んでしまうパターンは、**機能が見た目的に似ている場合**に起こりやすいです。ありがちな例として、次のようなリンクとボタン、2つの機能を切り替えられるコンポーネントを見てみましょう。

▼リスト 2.5　LinkOrButton.vue

```
<template>
  <div class="LinkOrButton">
    <a class="LinkOrButton_Link"
      v-if="href"
      :href="href"
    >
      {{label}}
    </a>
    <button class="LinkOrButton_Button"
      v-if="clickHandler"
      @click="clickHandler"
    >
      {{label}}
    </button>
  </div>
</template>

<script>
export default {
  name: "LinkOrButton",
  props: {
    label: String,
    href: String,
    clickHandler: Function,
  },
}
</script>
```

リスト 2.5 では、propsで受け取る値によって、v-ifで表示の切り替えをしています。これは hrefと clickHandlerがこのコンポーネントの**状態**として機能しています。

「リンクにもボタンにも使えていいじゃん。見た目一緒だし」と思う方もいるかもしれませんが、**見た目が似ていても、役割が別なら別のコンポーネント**として作成するのが、アプリケーションにも、テストにもいいやり方です。

たとえコンポーネントを作成するときにコストがかかったとしても、拡張/修正といっ

た作業を行うときに、払った分のコストに見合うようなリターンが得られると筆者は考えています。

「**見た目が似ていても、役割が別なら別のコンポーネント**」、大事なことなので何度でもいいます。

## 2.5 props の型指定で避けたほうがいい型

筆者は普段、コンポーネントの propsの型指定を、できるだけ次のいずれかの型で指定するようにしています。

- Number
- String
- Array
- Object
- Boolean

多くの場合、アプリケーションのデータは API から JSON で受け取って使います。そしてこれらは JSON で表現が可能な型です。言い換えると、これ以外の型を指定した場合、propsで渡す前に変換の作業が必要になります。

たとえば Dateを指定してる場合、コンポーネントにわたす前に new Date(dateTime) といった変換を行うことになり、テストが複雑になります。

▼リスト 2.6　props で Date を受け取る場合

```
import mockItem from "./mockData/item.json"

describe('MyComponent', () => {
  it('props', () => {
    const wrapper = mount(MyComponent, {propsData: {
      ...mockItem,
      date: new Date(mockItem.date)
    }})
    //...
  })
})
```

これはデータの取得時だけでなくデータの保存時も同様で、API を使って JSON 形式で保存する場合、Dateから元の形式へと再度変換する必要がでてきます。日時を表示するような場合には、コンポーネントは文字列か数値として受け取り、表示する際に new D

ateや日時操作系のライブラリで変換するようにすることをお勧めします。

## 2.6 親子コンポーネント間のやりとり

Vue コンポーネントの親子間のやりとりは、親コンポーネントが propsでデータを渡し、子コンポーネントからは$emitによって親へメッセージを送るパターンが推奨[*1]されています。

これは **props down events up** と呼ばれているパターンで、Vue.js の文脈では有名なものです。このパターンで親子コンポーネント間のやり取りの手順を一定にすると、データのやり取りは親の v-onと子の$emitに注目すればよくなり、コードの理解が簡単になります。一貫した手順はコンポーネントのテストのコストを下げる助けにもなるでしょう。

### 2.6.1 props で Function を渡す

**props down events up** が推奨される一方で、propsでは Functionを受け取ることができます。

たとえば、次のリスト 2.7 とリスト 2.8 は動作的には同じにようなものになります。

▼リスト 2.7　props down, events up

```
<template>
  <Child
    v-for="item in items" v-bind="item"
    v-on:clickChild="onClickChild"
  />
</template>

<script>
export default {
  methods: {
    onClickChild(item) {/* ... */}
  },
  components: { Child },
}
</script>
```

▼リスト 2.8　props down, events up

[*1] Vue.js スタイルガイド https://jp.vuejs.org/v2/style-guide/index.html

```
<template>
  <Child
    v-for="item in items" v-bind="item"
    :clickHandler="childClickHandler"
  />
</template>

<script>
export default {
  methods: {
    childClickHandler(item) {/* ... */}
  },
  components: { Child },
}
</script>
```

リスト 2.7 では Childで何かがクリックされたときに$emitで親に **clickChild イベント**を伝えて、親がそのイベントを v-onで購読して関数を実行します。対して、リスト 2.8 では、propsでクリックされたときに実行する関数を受け取り、クリックされたときに Childコンポーネントがその関数を実行します。

動作的には同じになりますが、親子コンポーネント間でやり取りをする方法としては異なるものです。

ではどちらの方法で親子間のやりとりをするのがいいのか、という疑問が湧いてくるわけですが、筆者は**親子コンポーネント間でのやり取りの方法がプロジェクト内で統一されていればどちらでもいい**と考えています。propsで Functionを渡すパターンは React ではごく普通の方法です。

どちらを使えばいいのかは、プロジェクトに関わる人の属性によっても変わるでしょう。React の経験者が多ければ Functionを渡す方法を全面的に採用するのは理にかなっていることだと思います。大事なのは**プロジェクト内でコンポーネント間のやり取りにルールを設ける**ことだと筆者は考えています。

ただし、**props の型指定で避けたほうがいい型**で紹介したように、コンポーネントの設計次第で、テストの際には空の Functionやモック関数を渡す必要がでてくることは気に留めておく必要があります。

## 2.7 Store の getters に注意

API から受け取ったデータを、Vuex の Store に格納して使う場面があります。このとき、gettersで View に寄せたようなデータを作成して利用していると、テストの際に一手間増えることに注意が必要です。

たとえば次のような gettersの場合を考えてみます。

▼リスト 2.9　view に寄せた getters

```
export default {
  todos(state) {
    return state.originalTodos.map(todo => {
      // todo を加工する処理
      return todo
    })
  },
}
```

リスト 2.9 では、元の originalTodosに対して、加工する処理を行っています。gettresに定義したこの todosを使うコンポーネントは、テストを行う際に、この gettersの処理を通す必要があります。

### 2.7.1 rootState と rootGetters の参照は危険信号

gettersが、state以外の値を参照しているときは特に注意が必要です。それは namespacedなモジュールの gettersから、他の namespacedなモジュールの stateや gettersを、rootGettersや rootStateを使って参照している場合です。

stateだけを加工する処理であれば、gettersを importして使うのも容易ですが、namespacedなモジュールを rootGettersや rootStateで参照しているケースは、テストデータの作成も非常に難しいものになります。

▼リスト 2.10　やばい getters

```
  yabaiGetter(state, getters, rootState, rootGetters) {
    const userTodos = getters.todos.filter(todo => {
      return todo.author === rootState.userName
    })
    return userTodos.filter(todo => {
      return new Date(todo.date).getMonth()
```

```
        === rootGetters["dateModule/currentDate"]getMonth()
    })
  },
```

リスト 2.10 は極端な例ですが、この getters自身や、この gettersを使うコンポーネントのテストを想像してみてください。1つのテストのために、state、getters、rootGetters、rootStateを再現するようなモックデータを用意しなければならなくなります。テスト用に完璧な Store を準備すればいいと思うかもしれませんが、コンポーネントもしくはこの gettersが変更されたときのコストを考えると、無用な複雑性を作りこんでいると筆者は考えます。

もし namespacedなモジュールを、rootGettersや rootStateから使わなければならないようなコードを書いている場合は、Store のデータやモジュールの分け方を、再考することを筆者の苦い経験からお勧めします。

## 2.8 ライフサイクルフックに直接処理を書かない

コンポーネントの各種ライフサイクルフックに、thisの値を参照したり$emitしたりといった処理をべた書きすることもあるかと思います。しかしながら、これがテストを妨げる要因となることがあります。

Vue コンポーネントのテストで使う vue-test-utils[*2]は、ライフサイクルのメソッドを mock 関数に差し替えることができません。ですので、ライフサイクルのメソッド内に処理がべた書きされている場合、その処理を止めたりスキップしたりといったことができません。

▼リスト 2.11　ライフサイクルにべた書き

```
mounted() { // mock 化できない
  this.wrapperClientHeight = this.$refs.wrapper.clientHeight
},
```

通常の methodsは mock 化できるので、ライフサイクルの中から methodsに定義した関数を実行するようにしておくことで、この問題を回避できます。

▼リスト 2.12　ライフサイクルからメソッド呼び出し

[*2] https://vue-test-utils.vuejs.org/ja/

```
mounted() {
  this.setWrapperClientHeight()
},
methods: {
  setWrapperClientHeight() { // mock 化できる
    this.wrapperClientHeight = this.$refs.wrapper.clientHeight
  }
}
```

vue-test-utils の issue や PRs を見る限り vue-test-utils の開発陣は、Vue の内部動作に関わる部分を変更するような路線には進みたくないようです。[*3]

ライフサイクルの処理は、少々めんどうでも関数として切り出しておくことをお勧めします。

[*3] https://github.com/vuejs/vue-test-utils/pull/167#issuecomment-359250469

# 第 3 章

# コンポーネントを分類する

Vue.js のプロジェクトでは、Vue コンポーネントを`/components`ディレクトリに格納することが多いかと思います。しかしながら、一口にコンポーネントと言っても、その機能や役割はさまざまです。

とくに、大きさ（粒度）の異なるコンポーネントをすべて同じ階層に入れてしまうと、大きなコンポーネントに引きづられてコンポーネントの分割が適切に行えないことも起こりえます。

## 3.1 コンポーネントの種類を知る

フロントエンドのコンポーネント指向な開発において、よく知られたコンポーネント分類のパターンがあります。それはコンポーネントを **Presentational Component** と **Container Component** の 2 つに分けて考えるパターンです。

### 3.1.1 Presentational Component

Presentational Component は主に `props`で受け取った値を表示する役割を担います。Presentational Component には、多くの場合次のような制約を持たせます。

- アプリケーションの構造や機能に依しない
  - Store や Router などの存在を知らず、依存しない
- データの読み込みや、変更方法を定義しない
- ライフサイクルフックを極力使わない
- コンポーネント独自の状態を持たせない
  - Form や、そのコンポーネントに閉じた状態を扱う場合を除く

Vue コンポーネントの場合、Presentational Component は受け取った値を表示し、コンポーネントでアクションが起こった場合には`$emit`を使って上位のコンポーネントに処理を委ねる、という形を取ることになるかと思います。

### 3.1.2 Container Component

Container Component はアプリケーションの振る舞いを定義する役割を担います。主な役割は次のようなものです。

- 下位のコンポーネントの振る舞いを定義する
- Store のデータや commit/action を扱う

Container Component はデータの読み込み方法や、Store や Router などのアプリケーション内の機能を知っています。読み込んだデータは`props`を経由してコンポーネントに渡し、コンポーネントからのイベントで Store の commit や Router の機能などを呼び出します。

## 3.2 ２種類で足りる？

コンポーネントの種類について、Presentational Component と Container Component という 2 つの大きな分類について解説しましたが、実際のプロジェクトでは、この 2 つだけでコンポーネントをきれいに分割するのは困難でしょう。分類の粒度が大きく、人によって捉え方が異なったものになるのがコンポーネント指向開発の難しいところです。

次章である第 4 章「ディレクトリ構成とコンポーネントの分類」では、筆者がプロジェクトを進める上で採用しているディレクトリ構造と、それに基づいたコンポーネントの分類について解説します。

# 第 4 章

# ディレクトリ構成とコンポーネントの分類

筆者は普段、コンポーネントを次の４つのディレクトリに分けて、アプリケーションを構築しています。

- `basics`
- `components`
- `containers`
- `pages`

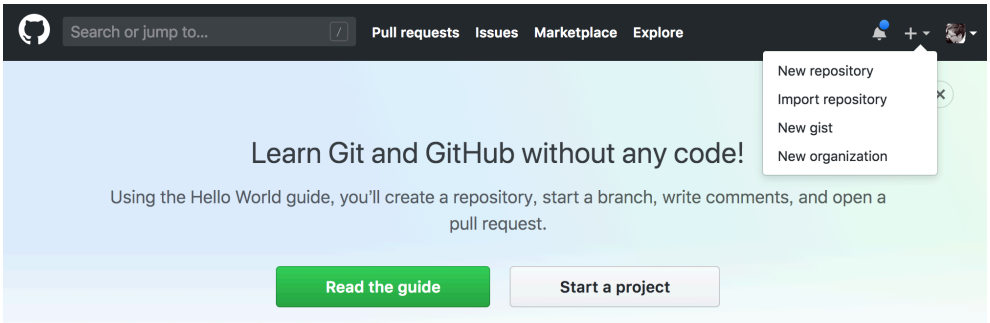
これらのディレクトリ名は、コンポーネントの分類と対応しています。エディタのファイルツリーなどでコンポーネントの大きさ（粒度）順に並ぶように命名しており、basics が一番小さく、pages が一番大きいコンポーネントの単位になります。

このあとの紹介ではそれぞれのディレクトリに入れるコンポーネントを、ディレクトリ名から `s`を取り除いた形で次のように呼ぶことにします。

- **basic**
- **component**
- **container**
- **page**

## 4.1 UI のサンプル

それぞれのコンポーネントの分類は、GitHub.com のトップページの**ヘッダー**を分解していく形で解説します。




▲図 4.1　GitHub のヘッダー

## 4.2 basics ディレクトリ

basics ディレクトリに格納する **basic** は、単体で機能が成立するようなコンポーネントの最小単位です。筆者が図 4.1 のヘッダーから **basic** に分割するとしたら、次の項目を **basic** とします。

- ロゴ
- 検索入力
- 通知のベルマーク
- ＋アイコン
- ユーザーアイコン

**Pull requests** などメニューの項目１つ１つを分解すれば **basic** になりそうですが、メニューの項目を単体で使うことはなかなかないように思います。同じように、メニューを展開する下向き矢印のアイコンも、クリックで開くメニューとセットとしてコンポーネントに切り出します。

最初から限界まで小さい単位でコンポーネントを考えると、無用な複雑性と冗長性を生み出す原因になると筆者は考えています。ですので、単体で機能するものを見極めて

**basic** として定義し、それらを **component** などで組み合わせていくのが、コンポーネント分割でのベストな方法ではないかと思います。

## 4.3 components ディレクトリ

components ディレクトリに格納する **component** は、**basic** または **component** を組み合わせたり、協調動作させるのが役割になります。

図 4.1 のヘッダーの中では、次の項目を **component** とします。

- メニュー
- 下向き矢印のアイコンと、クリックで開くメニュー
- 通知やアイコンと、クリックで開くメニューのセット

## 4.4 containers ディレクトリ

containers ディレクトリに格納する **container** は、自身より小さいコンポーネントである **component** と **basic** にデータを受け渡すのが主な役割です。特徴としては次のようなものです。

- Store を直接参照できる
- データの読み込みや変更を行える
- **component** 同士を協調させる
- **component** と **basic** を含むことができる
- **component** や **basic** のレイアウトを行う

図 4.1 のヘッダーの中では、次の項目を **container** とします。

- ヘッダー全体

## 4.5 pages ディレクトリ

pages ディレクトリに格納する **page** は、Vue Router の `routes`に指定するコンポーネントです。**page** の特徴は次のようなものになります。

- Store や Router を直接参照できる
- データの読み込みや変更を行える
- **page** 以外のコンポーネントを含むことができる

- **container** のレイアウトを行う

**page** はアプリケーション全体の機能を知っていて、必要に応じて Store の値を操作したり、Router の機能を使います。必要であれば格納するコンポーネントのレイアウト用のスタイルを追加します。

図 4.1 全体が **page** になります。

## Nuxt.js の pages ディレクトリ

Nuxt.js を使ったことがある方であれば、pages ディレクトリに馴染みがあるかと思います。本書で解説してる pages ディレクトリは、筆者が Nuxt.js から影響を受けて名付けたものですので、ほぼ同一のものだと捉えてください。

# 第 5 章

# なにをテストするか

Vue コンポーネントは、JavaSciprt だけでなく HTML と CSS も含む UI コンポーネントです。通常の JavaScript であれば、関数の入力と出力をテストの対象としますが、Vue コンポーネントの場合、コンポーネントに与える `props`や `computed`、テンプレートからのイベントで呼び出される `methods`など、コンポーネント全体の振る舞いがテストの対象となります。

コンポーネントのテストは、細かくやろうと思えばキリがないくらいに項目が挙げられます。しかし、テストを細かくやりすぎると、テストを追加/更新していくのがつらくなりがちです。コンポーネントをテストがしやすい状態を保つのも大事なことですが、うまく手を抜いていくこともテストを継続していくためには必要だと筆者は考えています。

## 5.1 テストの対象

筆者が普段コンポーネントのテスト項目としているものは、`template`と `script`の 2 つがメインです。`style`による見た目に関わる部分については、項目として挙げません。

レンダリング結果の HTML については第 7 章「テストを書く」で解説するスナップショットによるテストを行いますが、よほどプロダクトのコアに関わる部分でない限り、リグレッションテストなどは行いません。

条件によって CSS のクラスを付け替えるのは、`template`のテストの範囲内ですので、テストの対象としています。

## 5.2 コンポーネントのテスト項目

コンポーネントのテスト項目とするのはおもに次のようなものです。

- templateで次の項目が正しく動作するか
  - 正しくレンダリングされているか
  - v-on
  - v-bind:class
  - v-bind:attrs
- propsが正しく受け取れるか
- methodsが正しく動作するか
- $emitでイベントが正しく発火するか
- slotが正しく動作するか
- コンポーネント同士が正しく協調しているか

項目の最後にある**コンポーネント同士が正しく協調しているか**は結合テストを雑に総称したもので、それ以外はいわゆる単体テストになります。

単体テストでコンポーネント単体の動作をしっかり担保して、結合テストでコンポーネント同士の協調動作を確認します。

## 5.3 Vuex のテスト

本書では詳しくは解説しませんが、必要であれば Vuex Store の mutationsや actions、gettersについてもテストを行います。これら Vuex の機能は、それぞれはただの関数ですので、通常の関数のテストと同じように行っていきます。

## 5.4 どうやってテストするか

なにをテストするかについて紹介しましたが、では実際にテストを書こうと思うと手が止まってしまうかもしれません。次章の第 6 章「テスト実行環境の構築」と第 7 章「テストを書く」では、プロジェクトにテスト環境を構築する方法と、コンポーネントのテストの書き方について紹介します。

# 第 6 章

# テスト実行環境の構築

本章ではテスト環境の構築と、テストの書き方について解説します。本章の解説で使用するソースコードは次のリポジトリからダウンロード/clone できます。

`https://github.com/nakajmg/testable-vue-component-sample`

それでは Vue コンポーネントのテストを実行するための環境を構築していきましょう。

## 6.1 Vue CLI を使った環境構築

これから新しいプロジェクトを作成するのであれば、Vue CLI v3[*1]を使った環境構築を強く推奨します。Vue CLI を使ったテスト環境の構築はとても簡単です。まだテストを書いたことがない方でも、Vue CLI を使えばテスト実行環境の構築と、テストの実行がすぐに行えます。

### 6.1.1 Vue CLI でプロジェクトを作成する

次のコマンドは、Vue CLI によるプロジェクトの新規作成を行うものです。

```
$ npx vue create my-project
```

このコマンドを実行すると、次のように対話形式でプリセットの選択を求められます。

```
Vue CLI v3.0.1
? Please pick a preset: (Use arrow keys)
> ts (vue-router, vuex, sass, babel, typescript, pwa, eslint, unit-jest)
```

[*1] `https://cli.vuejs.org/`

```
  js (vue-router, vuex, sass, babel, pwa, eslint, unit-jest)
  default (babel, eslint)
  Manually select features
```

tsと jsプリセットにある unit-jestという項目に、Jest というテストランナーを使った環境構築を含んでいます。この 2 つのどちらかを選択すると、テスト環境を含む新規プロジェクトがすぐに作成されます。

プリセットではなく自分で使用したいものを選ぶ場合は、**Manually select features** を選択して、次に表示される **Unit Testing** を選択します。

```
Vue CLI v3.0.1
? Please pick a preset: Manually select features
? Check the features needed for your project:
 ◉ Babel
 ◯ TypeScript
 ◯ Progressive Web App (PWA) Support
 ◯ Router
 ◯ Vuex
 ◯ CSS Pre-processors
 ◉ Linter / Formatter
>◉ Unit Testing
 ◯ E2E Testing
```

項目を選んでいくと、テストに使うテストランナーの選択が表示されるので、**Jest** を選択します。

```
? Pick a unit testing solution:
  Mocha + Chai
> Jest
```

次の選択として各種設定を **package.json** に書くか、それぞれをファイルとして出力するかが表示されます。

```
? Where do you prefer placing config for Babel, PostCSS, ESLint, etc.? (Use arrow ke
> In dedicated config files
  In package.json
```

Jest の設定は編集する頻度がそれほど高くないので、プロジェクト直下にファイルが増えるのが嫌な方は **package.json** を選択するとよいかと思います。

### 6.1.2 テストの実行

Vue CLI で作成したプロジェクトの **package.json** には、`scripts`に `test:unit`というコマンドが用意されているので、これを実行するとテストが実行できます。

実行すると次のようなテスト結果が出力されます。

```
$ npm run test:unit

> my-project@0.1.0 test:unit /path/to/test/my-project
> vue-cli-service test:unit

 PASS  tests/unit/HelloWorld.spec.js
  HelloWorld.vue
    ✓ renders props.msg when passed (20ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.629s
Ran all test suites.
```

## 6.2 Vue CLI UI を使う

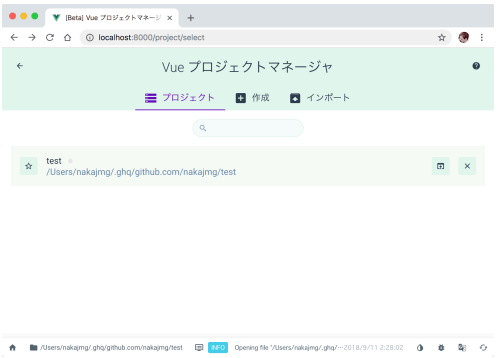
Vue CLI には、コマンドラインからだけでなく GUI で選択しながらプロジェクトを作成する方法である **Vue CLI UI** というものが用意されています。

### 6.2.1 Vue CLI UI の実行

次のコマンドで Vue CLI UI を実行します。

```
$ npx vue ui
```

実行すると図 6.1 のようにローカルサーバが起動して、ブラウザで開かれます。Vue CLI UI ではプロジェクトの作成だけでなく、プロジェクトの管理や、npm からのパッケージのインストールなどが GUI 上で行なえます。



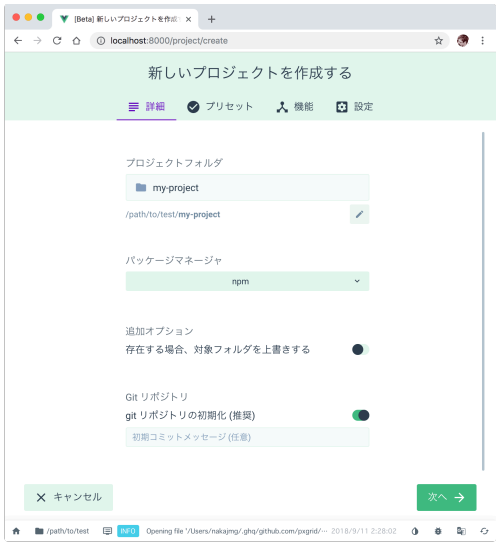

▲図 6.1　Vue CLI UI プロジェクト一覧

**プロジェクトの作成**

次に**作成**タブを選択すると図 6.2 のようにディレクトリを選択と、プロジェクトを作成するボタンが表示されます。

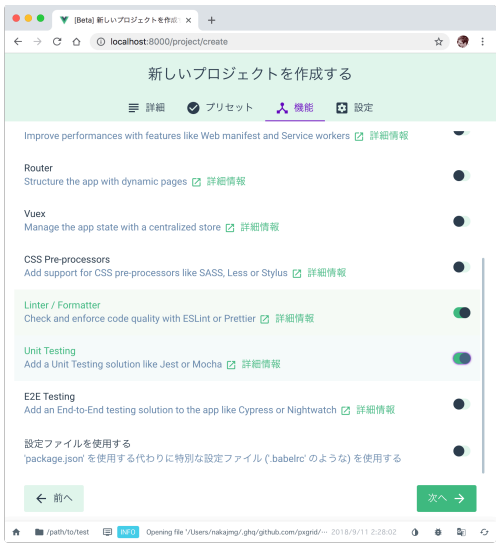

▲図 6.2　Vue CLI UI プロジェクト作成

プロジェクト作成のボタンを押すと、図 6.3 のように新しいプロジェクト作成の画面になります。プロジェクト名を入力し、好みに応じてオプションを選択していきます。



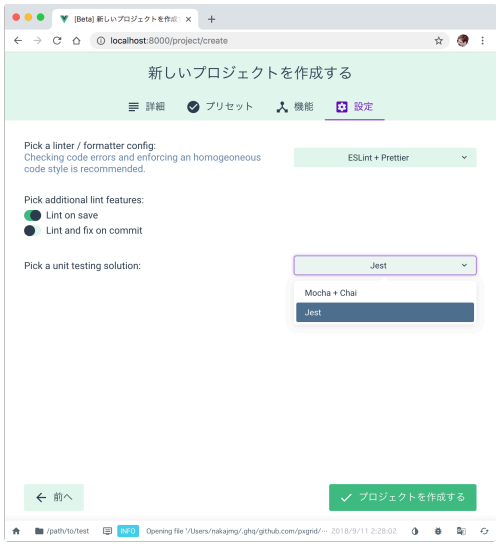

▲図 6.3　Vue CLI UI 新しいプロジェクトの作成

機能の選択で **Unit Testing** を有効にして、次に進みます。




▲図 6.4　Vue CLI UI 機能の選択

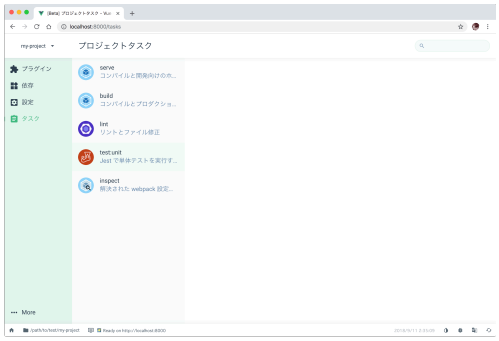
**Pick a unit testing solution** で Jest を選択してプロジェクトの作成は完了です。




▲図 6.5　Vue CLI UI Jest を選択する

必要なプラグインのインストールが完了すると、プロジェクト内のタスク（npm scripts）の一覧が表示されます。Vue CLI UI では、**package.json** の `scripts`にあるタスクを、GUI で起動することができます。

### テストの実行




▲図 6.6　Vue CLI UI タスク一覧

テストを実行する場合には、コマンドラインから `npm run test:unit`を実行するか、GUI から `test:unit`を実行します。

## 6.3 テストのサンプル

Vue CLI でテスト環境を構築すると、testディレクトリに HelloWorld.spec.jsというファイルが生成されます。このファイルは src/components/HelloWorld.vueのテストファイルです。

▼リスト 6.1　テストのサンプル

```
import { shallowMount } from "@vue/test-utils";
import HelloWorld from "@/components/HelloWorld.vue";

describe("HelloWorld.vue", () => {
  it("renders props.msg when passed", () => {
    const msg = "new message";
    const wrapper = shallowMount(HelloWorld, {
      propsData: { msg }
    });
    expect(wrapper.text()).toMatch(msg);
  });
});
```

リスト 6.1 は、コンポーネントの propsに渡した値が、期待どおりにレンダリングされるかをテストしています。このテストファイルを読めば、テストの書き方をなんとなく把握できるかと思います。

環境が構築できたら次は実際のテストの書き方です。

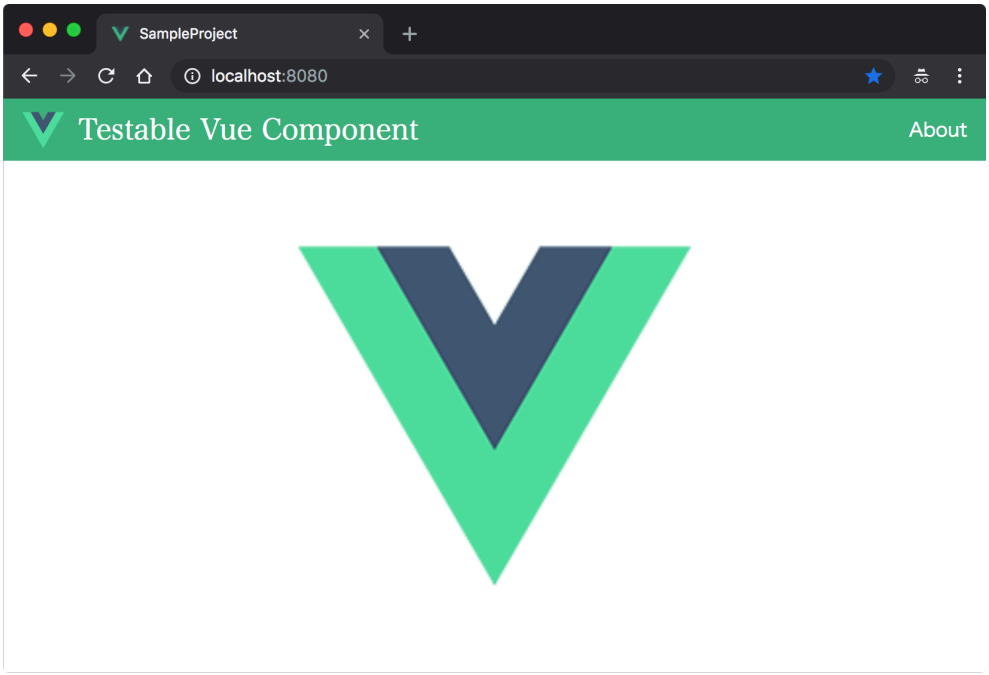
# 第 7 章

# テストを書く

本章では Vue コンポーネントのテストの書き方について紹介します。テストを実行するテストランナーには **Jest** を、コンポーネントを操作するライブラリとして **vue-test-utisl** を使います。

## 7.1 サンプルアプリケーション

テストを書いていく対象として、サンプルのアプリケーションを用意しました。




▲図 7.1　top

機能としては、ヘッダーのコンポーネントで、/ページと/aboutページを切り替えるだけのシンプルなものになっています。

このサンプルのアプリケーションは、次のリポジトリから自由にダウンロード/clone できます。

```
https://github.com/nakajmg/testable-vue-component-sample
```

解説では主要なテストケースに限定しています。全てのテストケースはソースコードか、付録 A「テストコード」を参照してください。

## 7.2　テストの実行方法

テストの実行は、次のコマンドで行います。

```
$ npm run test:unit
```

### Vue CLI で作成したプロジェクトのテストでエラーが起きるときは

Vue CLI v3 で作成したプロジェクトで、npm run test:unitを実行したときに、次のようなエラーがでる場合があります。

▼リスト 7.1　テスト実行時のエラー

```
SyntaxError: Unexpected string
      at ScriptTransformer._transformAndBuildScript
      (node_modules/jest-runtime/build/script_transformer.js:403:17)
```

その場合、jest.config.jsに次のように cache:　falseを加えてみてください。

▼リスト 7.2　jest.config.js

```
module.exports = {
  // ... 省略
  cache: false,
}
```

これでも直らない場合には、次の issue のコメントにある対策をいくつか試してみてください。
Default unit tests are failing - https://github.com/vuejs/vue-cli/issues/1879

## 7.3　ディレクトリとファイル構成

サンプルアプリケーションの/src配下の、ディレクトリ構造と、ファイル構成は次のようになっています。

```
./src
├── App.vue
├── assets
├── basics
```

```
│   ├── Logo.vue
│   └── SiteTitle.vue
├── components
│   ├── Menu
│   │   └── MenuItem.vue
│   └── Menu.vue
├── containers
│   └── GlobalHeader.vue
├── pages
│   ├── About.vue
│   └── Root.vue
├── router.js
└── store
    └── index.js
```

今回テストの対象とするのは、次の４つのディレクトリに格納している**.vue** ファイルです。

- **basiscs**
- **components**
- **containers**
- **pages**

各ディレクトリの役割については第 4 章「ディレクトリ構成とコンポーネントの分類」にて紹介していますので、まだ読んでいない場合は先に目を通しておくのをお勧めします。

## 7.4 Jest の使い方と機能

コンポーネントのテストを書く前に、テストランナー兼アサーションライブラリの Jest について、使い方と機能を軽く紹介します。

### 7.4.1 テストランナーとは

テストランナーとは、テストを実行してくれるツールです。`.spec.js`もしくは`.test.js`という拡張子のファイルをテストファイルとして認識して、テストケースをすべて、または個別に実行してくれます。

### 7.4.2 アサーション（expect）

アサーションとは、プログラムが意図したとおりの動作をしていない場合に、メッセージを表示したり、例外を投げたりするものです。意図したとおりに動作 J している場合に

は何も起きません。

Jest にはアサーションを行うための機能が備わっており、テストケースの実行時にはグローバル変数として expectという関数が使えます。次のリスト 7.3 は expectの使用例です。

▼リスト 7.3　expect の使用例

```
const wrapper = mount(MyComponent) // Vue コンポーネントのコンパイル
expect(wrapper.vm.isSelected).toBe(true)
```

リスト 7.3 では、wrapper.vm.isSelectedが trueかどうかを確認しています。もしテスト実行時にこの値が falseとなっていた場合、ここでテストが失敗し、期待した値になっていないことを通知するメッセージが表示されます。テストが失敗することを**テストがコケる**、**テストが落ちる**といった表現をすることがあります。

expectの機能については、都度解説していきますが、詳細を知りたい方は公式のドキュメント[*1]で確認ください。

### 7.4.3　describe と it

テストはある程度の細かさに分けた複数のテストケースで構成します。このとき使うのが describe()と it()です。describeはテストケースをまとめる単位として、itはテストケースそのものになります。

リスト 7.4 はテストの例です。describe()のコールバック関数として個別のテストケースである it()をグループ分けします。

▼リスト 7.4　testsample

```
describe("MyComponent.vue", () => {
  it("props のテスト", () => {
    const wrapper = shallowMount(MyComponent, { propsData: {text: "test"} })
    expect(wrapper.props().text).toBe("test")
  })
})
```

describe()は describe()を内包することができるので、テストケースを意味のある単位でグループ分けすることで、似たような対象を検証するテストケースのまとまりを作

[*1] https://jestjs.io/docs/ja/expect

ることができます。

もし Jest や JavaScript のテストについてあまり詳しくない場合は、一度公式ドキュメントの**テストのセットアップ**[*2]について目をとおしておくことをお勧めします。

> **it と test**
>
> Jest では `it()` と `test()` は同一のものですが、JavaScript の単体テストの習慣から筆者は `it` を好んで使っています。どちらを使っても動作は変わりませんので、好みの方を使ってください。

### 7.4.4 スナップショットテスト

Jest はスナップショットと呼ばれる、コンポーネントのレンダリング結果を比較する機能があります。比較するのは正しいレンダリング結果として保存したスナップショットで、このスナップショットは初回テスト時には存在しないので、テストケースを追加した初回にはスナップショットを作成するコマンドを実行します。

正しいレンダリング結果をスナップショットとして保存すると、以降のテストではこのスナップショットとレンダリング結果を比較します。もしコンポーネントにレンダリング結果が変わるような変更を入れた場合、このスナップショットテストは失敗します。変更が意図したものであればスナップショットを更新します。もし意図してない変更であった場合にはバグの発生を意味するので、修正する必要があることがわかります。

スナップショットテストは、レンダリング周りの単純なテストあればほとんどのケースをカバーできるので、テスト作成のコストを大幅に減らせる可能性を秘めています。スナップショットテストの実行方法については実際のテストケースで紹介します。

## 7.5 vue-test-utils

**vue-test-utils** は、Vue コンポーネントをテストするためのユーティリティライブラリです。`props` を渡してコンポーネントをマウントする機能や、コンポーネントの子コン

[*2] `https://jestjs.io/docs/ja/setup-teardown`

ポーネントをダミーに置き換える機能、methodsの関数を上書きする機能など、テストの際に便利な機能が揃っています。

### 7.5.1 mount と shallowMount

Vue コンポーネントのテストでは、まず mountか shallowMaountを使って、Vue コンポーネントをマウントするところから始めます。どちらも Vue コンポーネントをマウントする関数ですが、次のような違いがあります。

- mount: 子コンポーネントに実際のコンポーネントを使う
- shallowMount: 子コンポーネントをダミーコンポーネントに差し替える

コンポーネント単体のテストには、基本的に shallowMountを使います。子コンポーネントをそのまま使うと、子コンポーネントの動作まで検証するようなテストになってしまう可能性が高くなります。

また、子コンポーネントの機能などが変わったときに、親コンポーネントもテストまでもが影響を受けてしまいます。コンポーネント単体のテストでは基本的に shallowMaountを使い、コンポーネント同士で連携が必要な動作を検証するときにだけ mountを使うようにするとよいでしょう。

### 7.5.2 コンポーネントラッパー

mountと shallowMountを実行すると、マウントされたコンポーネントと、コンポーネントを検証するための関数を含むラッパーオブジェクトを返します。コンポーネントの Vue インスタンス（vm）や仮想 DOM は、このコンポーネントラッパーを通して取得/検証を行います。

▼リスト 7.5　コンポーネントラッパーの作成

```
const wrapper = shallowMount(SiteTitle, { propsData })
```

リスト 7.5 の wrapperがコンポーネントラッパーです。wrapperには Vue インスタンスの値を取得する関数だけでなく、Vue インスタンスの値を上書きする関数も用意されています。

### 7.5.3 使用頻度の高い関数

使用頻度の高いコンポーネントラッパーの関数を紹介します。すべての関数の詳細については、公式のドキュメント[*3]を参照ください。

#### props() と setProps()

`props()`は Vue インスタンスの `props`を取得する関数です。`setProps()`は Vue インスタンスに `props`で値を渡す関数です。

作成した Vue インスタンスの `props`の値を確認するときや、`props`の値を更新したいときに使います。

#### setMethods()

`setMethods()`は Vue インスタンスの `methods`を上書きする関数です。ボタンをクリックしたときや、イベントが発生したときに呼ばれる関数を mock に差し替える、といった使い方をします。

#### emitted()

`emitted()`は Vue インスタンスで`$emit()`されたイベントと、`$emit()`の際に引数に渡された値を取得できます。

#### find() と findAll()

`find()`と `findAll()`は、Vue インスタンスの DOM の中から、セレクタに一致する DOM ノードまたは Vue コンポーネントを取得するのに使います。たとえば、ボタンがクリックされたときの検証をしたい場合などでは `wrapper.find("button")`でボタン要素を探します。

#### trigger()

`trigger()`は `find()`などで探した DOM ノードに対して、DOM イベントを発火します。たとえば `wrapper.find("button").trigger("click")`とすれば、コンポーネントの中から最初に見つかった `button`要素に対してクリックイベントを発火できます。

[*3] https://vue-test-utils.vuejs.org/ja/api/wrapper/

## 7.6 basic のテスト

それでは実際にテストを書いていきます。まずは一番小さいコンポーネントの単位である **basic** のテストです。**basic** は一番小さいコンポーネントですので、テストケースもシンプルになる傾向があります。もし **basic** のコンポーネントのテストが「なんかめんどくさいな」となったら、コンポーネントの分割がうまくいっていない可能性があるので、見直してみるとよいかもしれません。

### 7.6.1 SiteTitle.vue

テストの対象は`SiteTitle.vue`です。このコードはリスト 7.6 のようになっています。

▼リスト 7.6　SiteTitle.vue

```
<template>
  <div class="SiteTitle">
    {{title}}
  </div>
</template>

<script>
export default {
  name: "SiteTitle",
  props: {
    title: String,
  },
}
</script>
```

#### SiteTitle.vue のテスト

`SiteTitle.vue`に対してのテストは次のリスト 7.7 のようになります。

▼リスト 7.7　SiteTitle.spec.js

```
import { shallowMount } from "@vue/test-utils"
import SiteTitle from "@/basics/SiteTitle.vue"
describe("SiteTitle.vue", () => {
  // props で渡す値の準備
  const propsData = {
    title: "test title",
  }
  it("props", () => {
    // コンポーネントラッパーの作成
    const wrapper = shallowMount(SiteTitle, { propsData })
    // Vue インスタンスの props が propsData と同じか
```

```
    expect(wrapper.props()).toEqual(propsData)
  })
  describe("template", () => {
    it("snapshot", () => {
      const wrapper = shallowMount(SiteTitle, { propsData })
      // レンダリング結果が前回のスナップショットと同じか
      expect(wrapper.vm.$el).toMatchSnapshot()
    })
  })
})
```

const wrapper = shallowMount(SiteTitle, { propsData )}でコンポーネントに propsで値を渡して、expect(wrapper.props()).toEqual(propsData)で propsを期待どおりに受け取れているかを確認しています。

### 7.6.2 スナップショットテストの実行

expect(wrapper.vm.$el).toMatchSnapshot()はスナップショットテストです。スナップショットとレンダリング結果を比較して、一致しなければこのテストケースは失敗します。

テストケースの初回実行時にはスナップショットが存在しないので、**npm** でスナップショットを作成するコマンドを実行します。

```
$ npm run test:update-snapshot
```

以降はこのスナップショットとレンダリング結果が比較されます。

**vue-cli-service** や **jest** コマンドを直接実行する場合には、-uオプションをつけるとスナップショットを更新できます。

## 7.7 component のテスト

次は **component** のテストです。**component** も Store や Router に依存しないように作成しているので、**basic** のテストと基本は変わりません。

ここからは主要なテストケースに絞って紹介します。

### 7.7.1 Menu.vue と MenuItem.vue

次は **component** のテストです。テストの対象は Menu.vueと MenuItem.vueです。

Menu.vueと MenuItem.vueは、それぞれリスト 7.8 とリスト 7.9 のようになっています。

▼リスト 7.8　Menu.vue

```
<template>
  <div class="Menu">
    <MenuItem class="Menu_Item"
      v-for="item in items"
      :key="item.label"
      v-bind="item"
      @clickMenuItem="onClickMenuItem"
    />
  </div>
</template>

<script>
import MenuItem from "./Menu/MenuItem.vue"
export default {
  name: "Menu",
  components: {
    MenuItem,
  },
  props: {
    items: Array,
  },
  methods: {
    onClickMenuItem({ name }) {
      this.$emit("clickMenuItem", { name })
    },
  },
}
</script>
```

▼リスト 7.9　MenuItem.vue

```
<template>
  <div class="MenuItem">
    <span class="MenuItem_Label"
      @click="clickMenuItem"
    >
      {{label}}
    </span>
  </div>
</template>

<script>
export default {
  name: "MenuItem",
  props: {
    label: String,
    name: String,
  },
  methods: {
    clickMenuItem() {
```

```
      this.$emit("clickMenuItem", { name: this.name })
    },
  },
}
</script>
```

### methods のテスト

Menu.vueの methodsをテストします。

▼リスト 7.10　methods のテスト

```
describe("methods", () => {
  it("clickMenuItem", () => {
    const wrapper = mount(Menu, { propsData })
    // onClickMenuItem() の実行
    wrapper.vm.onClickMenuItem(menuItems[0])
    // "clickMenuItem"イベントが emit されているか
    expect(wrapper.emitted("clickMenuItem")).toBeTruthy()
    // emit 時の引数が期待している値と同じか
    expect(wrapper.emitted("clickMenuItem")[0][0]).toEqual({
      name: menuItems[0].name,
    })
  })
})
```

Vue インスタンスの methodsは、コンポーネントラッパーの vmプロパティから参照/実行できます。リスト 7.10 では、wrapper.vm.onClickMenuItem()で関数を実行して、wrapper.emitted("clickMenuItem")が trueと評価される値かどうかで"clickMenuItem"イベントが発火しているかどうかを確認しています。

### MenuItem との結合テスト

Menu.vueは子コンポーネントとして MenuItem.vueを使用しています。ここでは@clickMenuItem="onClickMenuItem"としている箇所をテストしたいので、リスト 7.11 のようなテストケースを作成します。

▼リスト 7.11　子コンポーネントからの$emit のテスト

```
// mount による結合テスト
it("@clickMenuItem=onClickMenuItem", () => {
  const mock = jest.fn()
  const wrapper = mount(Menu, { propsData })
  // onClickMenuItem を mock に差し替える
  wrapper.setMethods({
    onClickMenuItem: mock,
  })
```

```
  // MenuItem コンポーネントを探す
  const menuItem = wrapper.find(MenuItem)
  // "clickMenuItem"イベントを emit する関数を実行
  menuItem.vm.clickMenuItem()
  // mock 実行時に期待する引数が渡されているか
  expect(mock).toHaveBeenCalledWith({
    name: menuItems[0].name,
  })
})
```

まず Vue インスタンスから find()で MenuItemコンポーネントの Vue インスタンスを探し、toHaveBeenCalledWith()で mock関数が引数に同じ値を指定して実行されているかを確認します。

このテストケースでは、"onClickMenuItem"イベントが子コンポーネントから発火したときに onClickMenuItem()が実行されるかどうかだけが関心の対象です。onClickMenuItem()の中身がどういった処理なのかは関係ないので、onClickMenuItemは jest.fn()で作成したモック関数 mockに置き換えます。

### MenuItem のテスト

MenuItem.vueの主要なテストは、DOM イベントが発火したときに関数が実行されるかを確認するテストです。テストケースはリスト 7.12 のようになります。

▼リスト 7.12　click イベントのテスト

```
it("@click=clickMenuItem", () => {
  // モック関数の作成
  const mock = jest.fn()
  const wrapper = shallowMount(MenuItem, { propsData })
  // clickMenuItem() を mock に置き換える
  wrapper.setMethods({
    clickMenuItem: mock,
  })
  // DOM を探してクリックする
  wrapper.find(".MenuItem_Label").trigger("click")
  // mock が呼ばれているか
  expect(mock).toHaveBeenCalled()
})
```

このテストケースでは、リスト 7.11 と同じように jest.fn()で作成したモック関数で clickMenuItemを置き換えます。そして find()を使って DOM 要素を探して、trigger()によって"click"イベントを発火します。さいごに toHaveBeenCalled()で mockが実行されたかを確認します。

テスト全体のソースは付録 A「テストコード」の「A.2 Menu.spec.js」と「A.3 MenuItem.spec.js」を参照ください。

## 7.8 container のテスト

次に **conteinr** のテストです。**conteinr** は Store に依存しているので、Store のモックを作成します。

### 7.8.1 GlobalHeader.vue

テストの対象は `GlobalHeader.vue`です。このコードはリスト 7.13 のようになっています。

▼リスト 7.13　GlobalHeader.vue

```
<template>
  <header class="GlobalHeader">
    <Logo class="GlobalHeader_Logo" />
    <span class="GlobalHeader_SiteTitle"
      @click="navigateRoot"
    >
      <SiteTitle :title="siteTitle"/>
    </span>
    <Menu class="GlobalHeader_Menu"
      :items="menuItems"
      @clickMenuItem="onClickMenuItem"
    />
  </header>
</template>

<script>
import { mapState } from "vuex"
import Logo from "../basics/Logo.vue"
import SiteTitle from "../basics/SiteTitle.vue"
import Menu from "../components/Menu.vue"
export default {
  name: "GlobalHeader",
  components: {
    Logo,
    SiteTitle,
    Menu,
  },
  computed: {
    ...mapState(["siteTitle", "menuItems"]),
  },
  methods: {
    onClickMenuItem({ name }) {
      this.$emit("navigate", { name })
    },
    navigateRoot() {
      this.$emit("navigate", { name: "root" })
```

```
    },
  },
}
</script>
```

### 7.8.2 テストで Vuex を使う

GlobalHeader.vueは、mapStateなど Vuex の機能に依存しています。テストを行うには Vuex Store を再現します。

テストで Vuex を使う場合、他のテストケースの Vue コンストラクタに影響を与えないように[*4]、リスト 7.14 のように、vue-test-utils の createLocalVueを使用して localVueを作成して使う必要があります。

▼リスト 7.14　localVue の作成と Vuex のインストール

```
import { shallowMount, createLocalVue } from "@vue/test-utils"
import Vuex from "vuex"
const localVue = createLocalVue()
localVue.use(Vuex)
```

そしてテストケースの実行前に storeが毎回同じ状態になるように beforeAllで storeを初期化します。

▼リスト 7.15　Store の初期化

```
describe("ContainerComponent.vue", () => {
  let store

  // 全てのテストの前に store を初期化
  beforeAll(() => {
    store = new Vuex.Store({
      state: {
        siteTitle: "test site title",
        menuItems,
      },
    })
  })
  it("some test", () => {
    // Vue インスタンス作成時に store と localVue を渡す
    const wrapper = shallowMount(ContainerComponent, { store, localVue })
  })
})
```

[*4] https://vue-test-utils.vuejs.org/ja/guides/using-with-vuex.html

初期化した storeはコンポーネントラッパーの作成時に localVueと一緒に渡して使います。

#### Store を使ったテスト

Store を使ったテストはリスト 7.16 のようになります。

▼リスト 7.16　Vue インスタンスに store を渡す

```
beforeAll(() => {
  store = new Vuex.Store({
    state: {
      siteTitle: "test site title",
      menuItems,
    },
  })
})

describe("methods", () => {
  it("clickMenuItem", () => {
    // Vue インスタンス作成時に store と localVue を渡す
    const wrapper = shallowMount(GlobalHeader, { store, localVue })
    wrapper.vm.onClickMenuItem(menuItems[0])
    expect(wrapper.emitted("navigate")).toBeTruthy()
    expect(wrapper.emitted("navigate")[0][0]).toEqual({
      name: menuItems[0].name,
    })
  })
})
```

テスト全体のソースは付録 A「テストコード」の「A.4 GlobalHeader.spec.js」を参照ください。

## 7.9　page のテスト

### 7.9.1　Root.vue

テストの対象は Root.vueです。このコードはリスト 7.17 のようになっています。

▼リスト 7.17　Root.vue

```
<template>
  <div class="Root">
    <GlobalHeader @navigate="onNavigate"/>
    <Logo class="Root_Logo"/>
  </div>
```

```
</template>

<script>
import GlobalHeader from "../containers/GlobalHeader.vue"
import Logo from "../basics/Logo.vue"
export default {
  name: "Root",
  components: {
    GlobalHeader,
    Logo,
  },
  methods: {
    onNavigate({ name }) {
      this.$router.push({ name })
    },
  },
}
</script>
```

### 7.9.2 Vue Router のテスト

Vue Router のテストも「7.8.2 テストで Vuex を使う」の説で紹介したように、他のテストケースへの影響を避けるために localVueを使ってインストールする必要があります。

ただし、$routerの関数の実行や、$routeの値を参照するような何かをテストしたい場合には、Vue Router をインストールせずにリスト 7.18 のように$routerや$routeをモックする[*5]ことで行います。

▼リスト 7.18　routermock

```
let $router
let $route
beforeAll(() => {
  // $router を模したオブジェクトの作成

  $router = {
    push: jest.fn(),
  }
  $route = {
    path: "/about",
    name: "about"
  }
})

it("test router", () => {
```

[*5] https://vue-test-utils.vuejs.org/ja/guides/using-with-vue-router.html

```
  const wrapper = shallowMount(PageComponent, {
    mocks: {
      $route,
      $router,
    },
  })
})
```

### $router.push のテスト

onNavigateが実行されたとき、$router.push()が呼ばれることを確認します。

▼リスト 7.19

```
let store
let $router
beforeAll(() => {
  store = new Vuex.Store({
    state: {
      siteTitle: "test site title",
      menuItems,
    },
  })
  // $router のモックを作成
  $router = {
    push: jest.fn(), // $router.push をモック関数にしておく
  }
})
describe("methos", () => {
  it("onNavigate", () => {
    // Vue インスタンス作成時に$router プロパティを注入する
    const wrapper = shallowMount(Root, { store, localVue, mocks: { $router } })
    wrapper.vm.onNavigate({ name: "root" })
    // mock 化した$router.push が呼ばれているか
    expect($router.push).toHaveBeenCalledWith({
      name: "root",
    })
  })
})
```

テスト全体のソースは付録 A「テストコード」の「A.5 Root.spec.js」を参照ください。

これで 4 種類のコンポーネントのテストは完了です。

# 付録 A

# テストコード

全てのテストコードは次のリポジトリからダウンロード/clone できます。
https://github.com/nakajmg/testable-vue-component-sample

## A.1 SiteTitle.spec.js

▼リスト A.1 SiteTitle.spec.js

```
import { shallowMount } from "@vue/test-utils"
import SiteTitle from "@/basics/SiteTitle.vue"
describe("SiteTitle.vue", () => {
  // props で渡す値の準備
  const propsData = {
    title: "test title",
  }
  it("props", () => {
    // コンポーネントラッパーの作成
    const wrapper = shallowMount(SiteTitle, { propsData })
    // Vue インスタンスの props が propsData と同じか
    expect(wrapper.props()).toEqual(propsData)
  })
  describe("template", () => {
    it("snapshot", () => {
      const wrapper = shallowMount(SiteTitle, { propsData })
      // レンダリング結果が前回のスナップショットと同じか
      expect(wrapper.vm.$el).toMatchSnapshot()
    })
  })
})
```

## A.2 Menu.spec.js

▼リスト A.2　Menu.spec.js

```
import { mount } from "@vue/test-utils"
import Menu from "@/components/Menu.vue"
import MenuItem from "@/components/Menu/MenuItem.vue"
import menuItems from "../_mockData/menuItems.json"
describe("Menu.vue", () => {
  const propsData = {
    items: menuItems,
  }
  it("props", () => {
    const wrapper = mount(Menu, { propsData })
    expect(wrapper.props()).toEqual(propsData)
  })
  describe("methods", () => {
    it("clickMenuItem", () => {
      const wrapper = mount(Menu, { propsData })
      // onClickMenuItem() の実行
      wrapper.vm.onClickMenuItem(menuItems[0])
      // "clickMenuItem"イベントが emit されているか
      expect(wrapper.emitted("clickMenuItem")).toBeTruthy()
      // emit 時の引数が期待している値と同じか
      expect(wrapper.emitted("clickMenuItem")[0][0]).toEqual({
        name: menuItems[0].name,
      })
    })
  })
  describe("template", () => {
    it("snapshot", () => {
      const wrapper = mount(Menu, { propsData })
      expect(wrapper.vm.$el).toMatchSnapshot()
    })
  })

  // mount による結合テスト
  it("@clickMenuItem=onClickMenuItem", () => {
    const mock = jest.fn()
    const wrapper = mount(Menu, { propsData })
    // onClickMenuItem を mock に差し替える
    wrapper.setMethods({
      onClickMenuItem: mock,
    })
    // MenuItem コンポーネントを探す
    const menuItem = wrapper.find(MenuItem)
    // "clickMenuItem"イベントを emit する関数を実行
    menuItem.vm.clickMenuItem()
    // mock 実行時に期待する引数が渡されているか
    expect(mock).toHaveBeenCalledWith({
      name: menuItems[0].name,
    })
  })
})
```

## A.3 MenuItem.spec.js

▼リスト A.3　MenuItem.spec.js

```
import { shallowMount } from "@vue/test-utils"
import MenuItem from "@/components/Menu/MenuItem.vue"
import menuItems from "../../_mockData/menuItems.json"
describe("MenuItem.vue", () => {
  const propsData = menuItems[0]
  it("props", () => {
    const wrapper = shallowMount(MenuItem, { propsData })
    expect(wrapper.props()).toEqual(propsData)
  })
  describe("methods", () => {
    it("clickMenuItem", () => {
      const wrapper = shallowMount(MenuItem, { propsData })
      // clickMenuItem() の実行
      wrapper.vm.clickMenuItem()
      // "clickMenuItem"イベントが emit されているか
      expect(wrapper.emitted("clickMenuItem")).toBeTruthy()
      // emit 時の引数が期待している値と同じか
      expect(wrapper.emitted("clickMenuItem")[0][0]).toEqual({
        name: propsData.name,
      })
    })
  })
  describe("template", () => {
    it("@click=clickMenuItem", () => {
      // モック関数の作成
      const mock = jest.fn()
      const wrapper = shallowMount(MenuItem, { propsData })
      // clickMenuItem() を mock に置き換える
      wrapper.setMethods({
        clickMenuItem: mock,
      })
      // DOM を探してクリックする
      wrapper.find(".MenuItem_Label").trigger("click")
      // mock が呼ばれているか
      expect(mock).toHaveBeenCalled()
    })
    it("snapshot", () => {
      const wrapper = shallowMount(MenuItem, { propsData })
      expect(wrapper.vm.$el).toMatchSnapshot()
    })
  })
})
```

## A.4 GlobalHeader.spec.js

▼リスト A.4　GlobalHeader.spec.js

```
import { shallowMount, mount, createLocalVue } from "@vue/test-utils"
import GlobalHeader from "@/containers/GlobalHeader.vue"
import Menu from "@/components/Menu.vue"
import Vuex from "vuex"
import menuItems from "../_mockData/menuItems.json"
// local の Vue を作る
const localVue = createLocalVue()
// local の Vue に Vuex をインストール
localVue.use(Vuex)
describe("GlobalHeader.vue", () => {
  let store

  // 毎テストケース実行前に store を初期化する
  beforeAll(() => {
    store = new Vuex.Store({
      state: {
        siteTitle: "test site title",
        menuItems,
      },
    })
  })

  describe("methods", () => {
    it("clickMenuItem", () => {
      // Vue インスタンス作成時に store と localVue を渡す
      const wrapper = shallowMount(GlobalHeader, { store, localVue })
      wrapper.vm.onClickMenuItem(menuItems[0])
      expect(wrapper.emitted("navigate")).toBeTruthy()
      expect(wrapper.emitted("navigate")[0][0]).toEqual({
        name: menuItems[0].name,
      })
    })
    it("navigateRoot", () => {
      const wrapper = shallowMount(GlobalHeader, { store, localVue })
      wrapper.vm.navigateRoot()
      expect(wrapper.emitted("navigate")).toBeTruthy()
      expect(wrapper.emitted("navigate")[0][0]).toEqual({
        name: "root",
      })
    })
  })

  describe("template", () => {
    it("@click=navigateRoot", () => {
      const mock = jest.fn()
      const wrapper = shallowMount(GlobalHeader, { store, localVue })
      wrapper.setMethods({
        navigateRoot: mock,
      })
      wrapper.find(".GlobalHeader_SiteTitle").trigger("click")
      expect(mock).toHaveBeenCalled()
    })
    it("@clickMenuItem=onClickMenuItem", () => {
      const mock = jest.fn()
      const wrapper = shallowMount(GlobalHeader, { store, localVue })
      wrapper.setMethods({
        onClickMenuItem: mock,
      })
```

```
      const menu = wrapper.find(Menu)
      menu.vm.$emit("clickMenuItem")
      expect(mock).toHaveBeenCalled()
    })
    it("snapshot", () => {
      const wrapper = mount(GlobalHeader, { store, localVue })
      expect(wrapper.vm.$el).toMatchSnapshot()
    })
  })
})
```

## A.5 Root.spec.js

▼リスト A.5　Root.spec.js

```
import { createLocalVue, shallowMount } from "@vue/test-utils"
import Vuex from "vuex"
import menuItems from "../_mockData/menuItems.json"
import Root from "@/pages/Root.vue"
import GlobalHeader from "@/containers/GlobalHeader.vue"
const localVue = createLocalVue()
localVue.use(Vuex)
describe("Root.vue", () => {
  let store
  let $router
  beforeAll(() => {
    store = new Vuex.Store({
      state: {
        siteTitle: "test site title",
        menuItems,
      },
    })
    // $router のモックを作成
    $router = {
      push: jest.fn(), // $router.push をモック関数にしておく
    }
  })
  describe("methos", () => {
    it("onNavigate", () => {
      // Vue インスタンス作成時に$router プロパティを注入する
      const wrapper = shallowMount(Root, { store, localVue, mocks: { $router } })
      wrapper.vm.onNavigate({ name: "root" })
      // mock 化した$router.push が呼ばれているか
      expect($router.push).toHaveBeenCalledWith({
        name: "root",
      })
    })
  })
  describe("template", () => {
    it("@navigate=onNavigate", () => {
      const mock = jest.fn()
      const wrapper = shallowMount(Root, { store, localVue, mocks: { $router } })
      wrapper.setMethods({
```

```
        onNavigate: mock,
      })
      const globalHeader = wrapper.find(GlobalHeader)
      // "navigate"イベントを emit する
      globalHeader.vm.$emit("navigate")
      expect(mock).toHaveBeenCalled()
    })
    it("snapshot", () => {
      const wrapper = shallowMount(Root, { store, localVue, mocks: { $router } })
      expect(wrapper.vm.$el).toMatchSnapshot()
    })
  })
})
```

# あとがき

読んでいただきありがとうございました。当初の予定より書きたいことが多くなり、そこそこのボリュームでお送りしましたが、いかがでしたでしょうか。

本書はこれまで Vue.js や他のライブラリを使ってきて貯まっていた、頭の中にある「こうしたらヨサソウ」を文書化したものです。もともとは Vue Fes Japan の CFP として送ったネタですが、落選して行き場を失っていたので、こうして 1 冊の本としてまとめられてよかったです。

筆者は普段 CodeGrid という技術系メディアで執筆していますが、Web と紙媒体という違いを意識してしまい、少し筆の進みが遅くなったりもしました。ですが、スパイダーマンのゲームを早くやりたいという一心でなんとか書ききることができました。Marvel に深く感謝します。あと執筆中にゲームをしながら励ましてくれた妻にも感謝します。

繰り返しになりますが本書を手にとっていただきありがとうございます。本書の何か 1 つでも開発の役に立てば幸いです。

# 著者紹介

**サークル: PonyHead**

**著者: じまぐ**

**Twitter: @nakajmg**

**表紙: ゆりえってぃ**

**Twitter: @yuriettys**

# 後悔しないための Vue コンポーネント設計

2018 年 10 月 8 日　技術書典 5 版 v1.0.0
著　者　nakajmg
発行所　PonyHead

(C) 2018 PonyHead