Realizacja aplikacji internetowych

2025/26

Instrukcja laboratoryjna

Framework Vue

Prowadzący: Tomasz Goluch

Wersja: 1.0

I. Wprowadzenie

Cel: Przekazanie podstawowych informacje o zasobach laboratorium.

Laboratorium odbywa się na maszynach fizycznych (komputery laboratoryjne albo własne laptopy), z wykorzystaniem preferowanego edytora lokalnego (np. Visual Studio, Visual Studio Code, Rider, WebStorm. Nie ma żadnych przeciwskazań aby laboratorium uruchomić w oparciu o kontenery Docker, szczególnie z wykorzystaniem Docker Compose.

II. Instalacja

Cel: Zapoznanie z dostępnymi sposobami instalacji Vue.js.

Aby korzystać z framework'a wystarczy umieścić znacznik `script` pobierający wybraną wersję biblioteki z serwera CDN. W przypadku pozostawić ciąg `vue@3.X.X` albo `vue@next` zostanie on przetłumaczony na najnowszą wersję:

```
<script src="https://unpkg.com/vue@3.X.X/dist/vue.global.js"></script>
<script src="https://unpkg.com/vue@next"></script>
<script src="https://cdn.jsdelivr.net/npm/vue@3.X.X/dist/vue.global.js"></script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/vue/3.X.X/vue.global.js"></script>
<script src=" https://cdn.jsdelivr.net/npm/vue@next/dist/vue.global.js
"></script>
```

Jeśli chcemy korzystać z takich dobrodziejstw Vue.js jak komponenty, wtyczki, gotowe szablony aplikacji musimy zainstalować `vue-cli` albo `Vite` do czego wymagane będzie środowisko wykonawcze `Node.js`[1] wraz menedżerem pakietów `npm`. Można alternatywnie wykorzystać innego menedżera pakietów jak: `Yarn`[2], `PNPm`[3].

Instalację Node.js warto rozpocząć do instalacji Manager'a Node.js NVM. Pozwala on na łatwe przełączanie się pomiędzy różnymi wersjami, co pozwala – przykładowo – przetestować rozwijany moduł korzystając z najnowocześniejszej wersji Node.js, bez konieczności odinstalowywania stabilnej wersji. Wersję dla Windows można pobrać tutaj: https://github.com/coreybutler/nvm-windows/releases/ (oczywiście wybieramy najnowszą wersję). Po zainstalowaniu można korzystać z następujących komend:

`nvm install latest` – instalacja najnowszej wersji Node.js (wymagane uprawnienia administratora);
`nvm ls` – informacja o zainstalowanych wersjach i ustawionej jako aktualnie używana;
`nvm use X.X.X` – określenie która wersja ma być używana (wymagane uprawnienia administratora).

Jeśli preferujemy `Yarn'a` albo `pnpm` można któryś doinstalować:

`npm install --global yarn` lub `npm install -g pnpm`

Przydatnym narzędziem jest GUI dla npm'a:

[1] Strona projektu Node.js: https://nodejs.org/en/
[2] Strona projektu Yarn: https://classic.yarnpkg.com/lang/en/
[3] Strona projektu Pnpn: https://pnpm.io/

`npm install -g npm-gui`, uruchamiane komendą `npm-gui`.

Pozwala na intuicyjne zarządzania zależnościami wymienionymi w pliku `package.json`. W tle będzie ono transparentnie używać poleceń `npm`, `pnpm` lub `yarn` do instalowania, usuwania lub aktualizowania zależności.

W kolejnym kroku tworzymy nowy projekt. Zaleca się aby wykorzystać komendę `npm create-vue` albo alias: `npm create vue@latest`[4], która bazuje na Vite. Jest to zmiana w stosunku do wcześniej wykorzystywanej komendy `vue create <nazwa-projektu` która opierała się na VueCLI i webpacku. Vite nie opiera się o system modułów Node (CommonJS) a o natywne moduły ECMAScript (ESM)[5]. Zapewnia on znacznie lepsze środowisko programistyczne dzięki niezwykle szybkiemu uruchamianiu i szybkości wymiany gorących modułów znanych jako mechanizm HMR (ang. Hot Module Replacement), pozwalających na przeładowanie aplikacji po zmianie kodu. Pozwala na zastosowanie SSR (ang. Server Side Rendering). W tym celu wykorzystamy wspomniany kreator `npm create-vue`, który pozwala na szczegółową konfigurację projektu. Inicjalizacja szablonu z: TS, Router, Pinia, Vitest, Cypress, ESLint i Prettier:

```
> npx
> create-vue

Vue.js - The Progressive JavaScript Framework

✓ Project name: ... rai-lab
✓ Add TypeScript? ... No / Yes
✓ Add JSX Support? ... No / Yes
✓ Add Vue Router for Single Page Application development? ... No / Yes
✓ Add Pinia for state management? ... No / Yes
✓ Add Vitest for Unit Testing? ... No / Yes
✓ Add an End-to-End Testing Solution? » Cypress
✓ Add ESLint for code quality? » Yes
✓ Add Prettier for code formatting? ... No / Yes

Scaffolding project in D:\_TMP\rai-lab...

Done. Now run:

  cd rai-lab
  npm install
  npm run format
  npm run dev
```

Instalacja i uruchomienie nowego projektu (wymagane uprawnienie administratora):

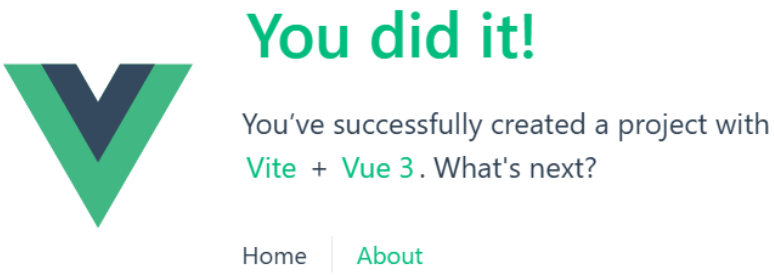
```
cd <nazwa_projektu>
npm|yarn install
```

[4] https://github.com/vuejs/create-vue
[5] https://blog.logrocket.com/commonjs-vs-es-modules-node-js/

```
npm|yarn run format
npm|yarn run dev
```

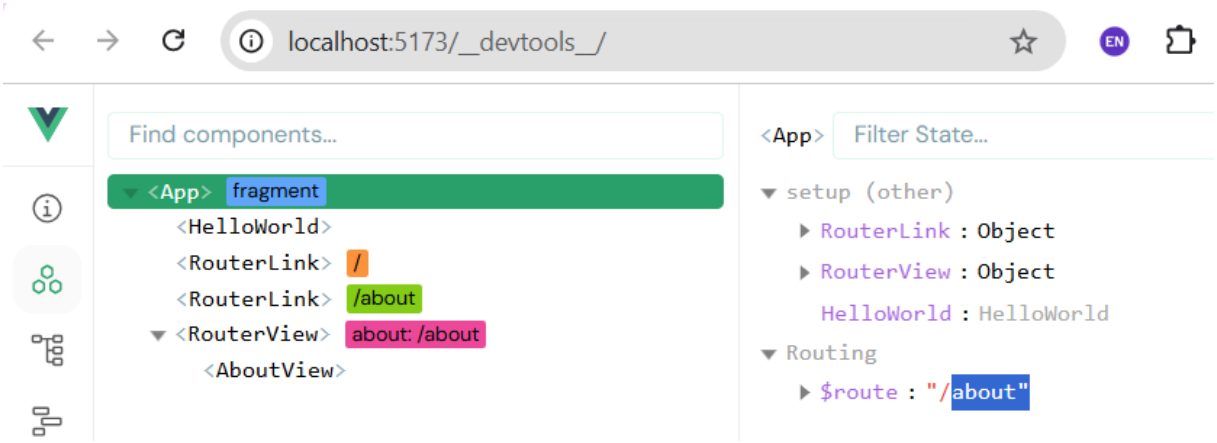
```
VITE v5.4.11  ready in 5673 ms

➜  Local:   http://localhost:5173/
➜  Network: use --host to expose
➜  Vue DevTools: Open http://localhost:5173/__devtools__/ as a separate window
➜  Vue DevTools: Press Alt(⌥)+Shift(⇧)+D in App to toggle the Vue DevTools

➜  press h + enter to show help
```

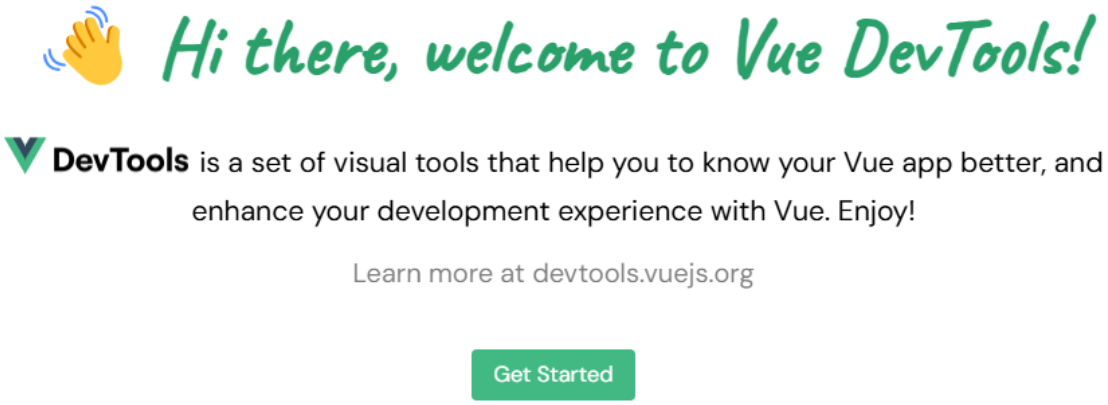
Aplikacja powinna być widoczna w przeglądarce pod lokalnym adresem z domyślnym portem 5173: http://localhost:5173:



Zatrzymanie projektu (powrót do powłoki): Q + enter

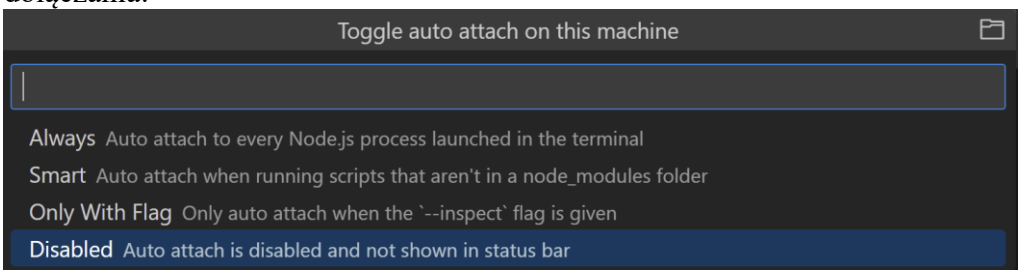
W celu łatwiejszego debugowania aplikacji możemy wykorzystywać okno z narzędziami dewelopera DevTools (Alt+Shift+D) w oknie aplikacji albo w osobnym oknie dodając suffix __devtools__/ do adresu:





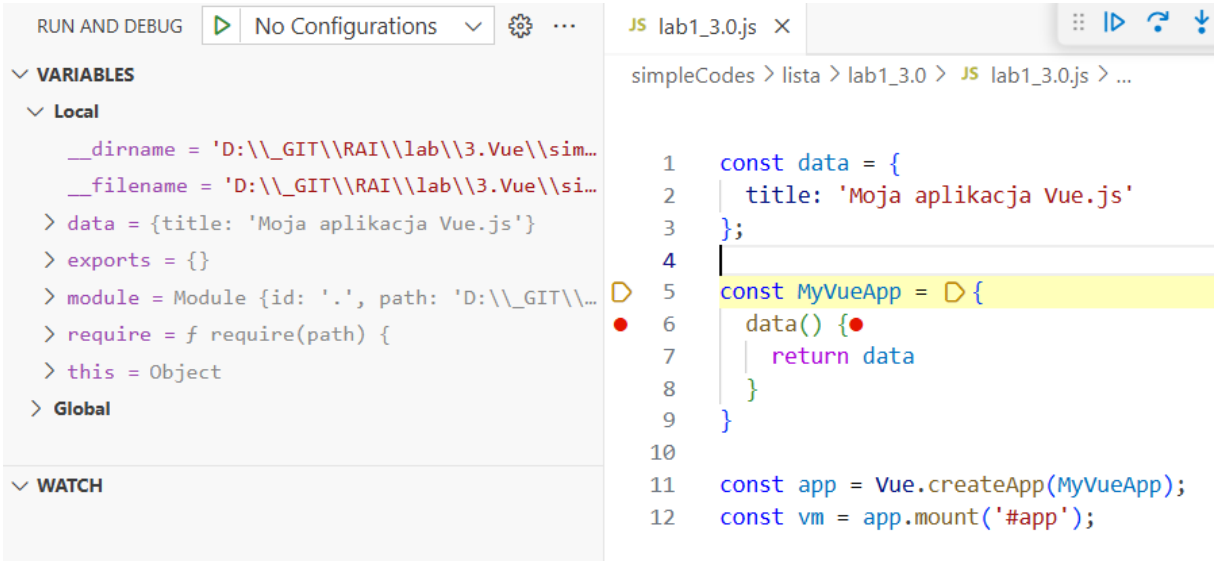
Projekt Vue.js może być rozwijany z wykorzystaniem wielu IDE, jednym z lepszych wyborów jest VS Code. Otwarcie projektu z poziomu IDE ogranicza się do wskazania folderu z aplikacją (zawierającego plik package.json) w tym celu możemy posłużyć się skrótem klawiaturowym (Ctrl+K, Ctrl+O). Uruchomienie z wiersza poleceń jest jeszcze łatwiejsze i wystarczy komenda `code <ścieżka_do_folderu_projektu>`, zatem będąc w interesującym nas folderze wystarczy komenda: `code .`.

Nieocenioną zaletą VS Code jest wbudowana obsługa debugowania środowiska uruchomieniowego Node.js[6] dzięki której, można debugować JS oraz inne języki, transpilowane do JS, np. TypeScript. W przypadku prostych projektów, kiedy plik projektu package.json nie istnieje albo nie jest skonfigurowany, możemy skorzystać z Toggle Auto Attach z palety komend (Ctrl+Shift+P). Do wyboru mamy trzy tryby automatycznego dołączania:



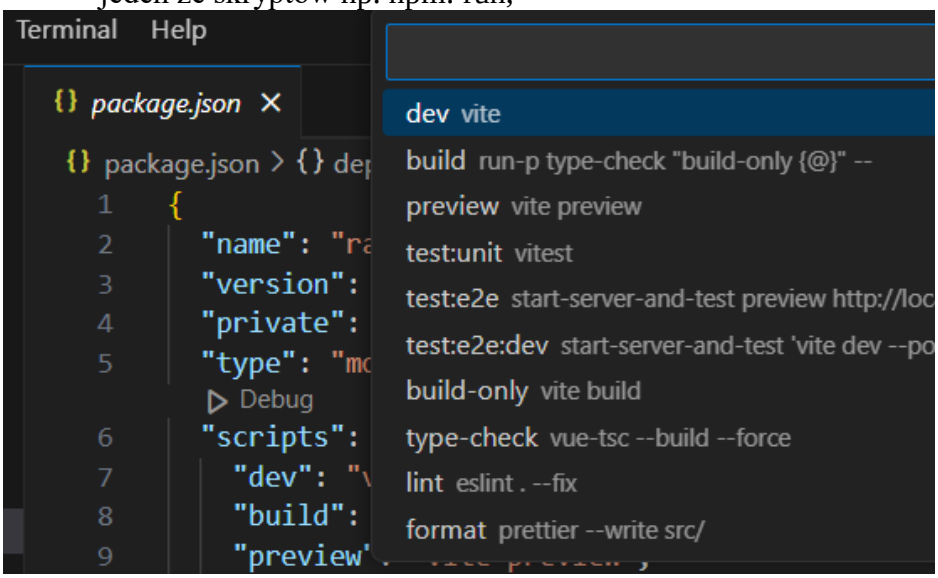
Teraz wystarczy uruchomić Debbuger (F5):

[6] https://code.visualstudio.com/docs/nodejs/nodejs-debugging

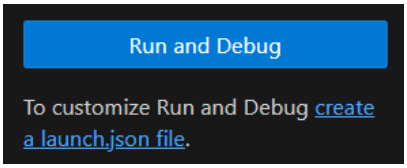


W przypadku istnienia konfiguracji w pliku package.json, np. po instalacji projektu:

- skrótem klawiszowym (Ctrl + `) otwieramy PowerShell'a w oknie terminal w IDE a następnie podajemy odpowiednią komendę, np.: `npm run dev`;
- VS Code automatycznie wykrywa zadania dla npm. Po otwarciu pliku package.json nad kluczem "scripts" pojawi się komenda Debug – po jej kliknięciu możemy wybrać jeden ze skryptów np. npm: run;



- (rekomendowane[7]) w sekcji Run and Debug (Ctrl+Shift+D) możemy dostosować parametry uruchamiania i debuggowania klikając na odnośnik: …create a launch .json file:

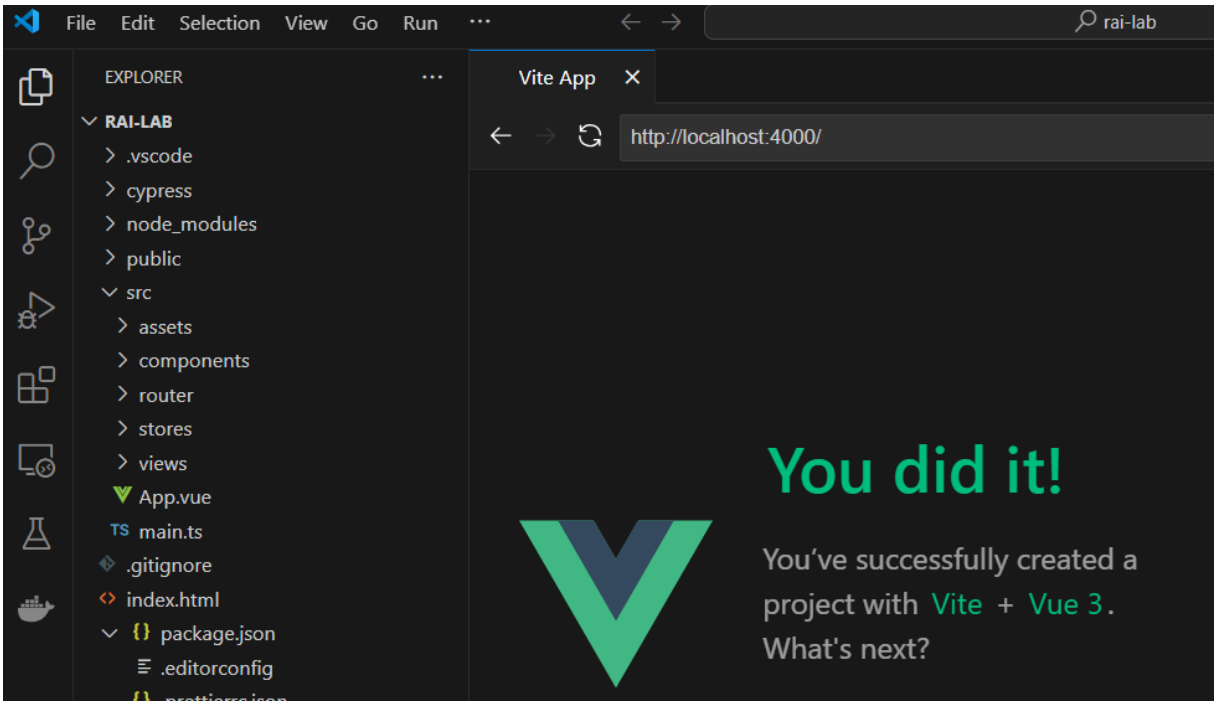


Przykład pliku launch.json z mieszaną konfiguracją uruchamiającą program w node.js i przeglądarkę na wybranym url'u:

[7] https://code.visualstudio.com/docs/editor/debugging#_launch-configurations

```
{
  "version": "0.2.0",

  "configurations": [
    {
      "type": "chrome",
      "request": "launch",
      "name": "localhost (Edge)",
      "url": "http://localhost:5173",
      "webRoot": "${workspaceFolder}"
    },
    {
      "command": "npm run dev",
      "name": "Launch Program",
      "request": "launch",
      "type": "node-terminal",
      "cwd": "${workspaceFolder}"
    }
  ],
  "compounds": [
    {
      "name": "Lunch program in Edge",
      "configurations": ["Launch Program", "localhost (Edge)"],
      "stopAll": true
    }
  ]
}
```

W przypadku projektu Vite można zainstalować wtyczkę VS Code for Vite ⚡ – nowy serwer deweloperski będzie startował zaraz po otwarciu projektu.



Jeśli chcemy uruchomić projekt Vue w Visual Studio to potrzebna będą nam dwa pliki:

- plik projektu *.esproj – określa jaka wersja SDK ma zostać pobrane z repozytorium Nuget, komendę uruchomieniową z package.json, gdzie są przechowywane testy i jaki framework będzie używany, gdzie będą przechowywany zbudowany projekt i czy przy każdym uruchomieniu go budować:

```
<Project Sdk="Microsoft.VisualStudio.JavaScript.Sdk/1.0.1184077">
  <PropertyGroup>
    <StartupCommand>npm run dev</StartupCommand>
    <JavaScriptTestRoot>.\</JavaScriptTestRoot>
    <JavaScriptTestFramework>Jest</JavaScriptTestFramework>
    <!-- Allows the build (or compile) script located on package.json to run
on Build -->
    <ShouldRunBuildScript>false</ShouldRunBuildScript>
    <!-- Folder where production build objects will be placed -->
    <BuildOutputFolder>$(MSBuildProjectDirectory)\dist</BuildOutputFolder>
  </PropertyGroup>
</Project>
```

- plik .vscode\launch.json – określający typ przeglądarki i adres url:

```
        {
        "version": "0.2.0",
        "configurations": [
            {
            "type": "edge",
            "request": "launch",
            "name": "localhost (Edge)",
            "url": "https://localhost:5173",
            "webRoot": "${workspaceFolder}"
            },
            {
            "type": "chrome",
            "request": "launch",
            "name": "localhost (Chrome)",
            "url": "https://localhost:5173",
            "webRoot": "${workspaceFolder}"
            }
        ]
        }
```

III. Tworzenie aplikacji Vue.js

Cel: Utworzenie prostej aplikacji Vue.js.

Na początek utworzymy prostą aplikację. Nie trzeba jeszcze instalować `vue-cli` albo `Vite` wystarczy sama biblioteka. W pliku HTML np. `labVUE.html`. tworzymy element DOM aplikacji Vue:

```
<!DOCTYPE html>
<script src="https://unpkg.com/vue"></script>
<html>
  <body>
    <div id="app" class="demo">
      <h2>{{title}}</h2>
      <input v-model="title">
    </div>
  </body>
  <script src="labVUE.js"></script>
</html>
```

W pliku JS np. `labVUE.js` dodajmy kod komponentu który jest używany jako punkt startowy do renderowania, gdy montujemy aplikację:

```
const data = {
  title: 'Moja aplikacja Vue.js'
};

const MyVueApp ={
  data() {
    return data
  }
}
```

Tworzymy instancję aplikacji Vue przekazując do funkcji `createApp` globalnego API (funkcje dostępne poprzez globalny obiekt Vue) powyższy kod i montujemy go w DOM za pomocą metody `mount` z lokalnego API.

```
const app = Vue.createApp(MyVueApp);
const vm = app.mount('#app')
```

Ponieważ `createApp` jak i większość metod aplikacji zwraca obiekt aplikacji możemy łańcuchować wołania metod (fluent API) z lokalnego API:

```
const vw = Vue.createApp(MyVueApp).mount('#app')
```

Po uruchomieniu powinniśmy otrzymać następujący wynik:

Moja aplikacja Vue.js

Metoda `mount` wyjątkowo zamiast instancji aplikacji zwraca instancję głównego komponentu, warto więc wcześniej przypisać odwołanie do obiektu w zmiennej, może okazać się przydatne. Przykładowo, w Vue.js 3.X wycofano filtry znane z 2.X, ale teraz można wykorzystać globalne właściwości[8]:

[8] https://v3.vuejs.org/api/application-config.html#globalproperties

```
const myApp = Vue.createApp(MyVueApp)
myApp.config.globalProperties.$filters = {
  capitalize(value) {
    return value.toUpperCase()
  }
}
const vm = myApp.mount('#app')
```

Wykorzystanie filtru:

```
<p> {{ $filters.capitalize('To jest jakiś napis') }} </p>
```

Wynik:

TO JEST JAKIŚ NAPIS

Właściwości globalne muszą być zdefiniowane przed zamontowaniem widoku w DOM w którym następuje ich użycie, inaczej dostaniemy ostrzeżenie: Property "$filters" was accessed during render but is not defined on instance.

Aplikacja Vue była inspirowana wzorcem architektonicznym MVVM. Instancja zwrócona przez `mount` w pewnym sensie spełnia założenia widok-modelu realizującego wiązanie danych pomiędzy modelem danych zwracanym w metodzie `data()` a widokiem jako elementem DOM z atrybutem `id="#app".`

Vue pozwala na umieszczanie wyrażeń JS w podwójnych nawiasach klamrowych:

```
<p> {{ "Sufit z 7,45 to: " + Math.ceil(7.45) }} </p>
```

Wynik:

Sufit z 7,45 to: 8

IV. Wybrane dyrektywy Vue[9]

Cel: Zapoznanie z działaniem wybranych dyrektyw Vue.js.

Dyrektywy to specjalne atrybuty rozpoczynające się prefixem `v-`. dodające reaktywne zachowanie do renderowanego DOM. Dyrektywa `v-model` pozwala na dwukierunkowe wiązanie danych, współpracuje z elementami: `input`, `select`, `textarea`. Dodajmy element `input`[10]:

```
<div id="app" class="container">
    <h2>{{title}}</h2>
    <input v-model="title">
</div>
```

[9] https://vuejs.org/v2/api/#Directives

[10] W przypadku problemów z wiązaniem sprawdź czy dodany element jest wewnątrz elementu `<div id="app" class="container">`?

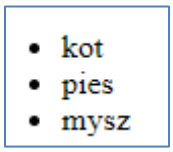
Efekt powinien być następujący:

Warto zauważyć, że kod koncentruje się tylko na logice a wszystkie manipulacje na DOM są delegowane do Vue.
Dyrektywa v-for[11] pozwala na iteracyjne przetwarzanie, np. elementów tablicy. Dodajmy tablicę stringów do naszego modelu:

```
const data = {
  title: 'Moja aplikacja Vue.js',
  tab: ['kot', 'pies', 'mysz']
};
```

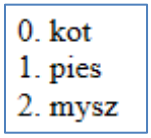
Aby je wyświetlić na stronie wystarczy dodać do zwykłej listy wspomnianą dyrektywę:

```
<div id="app" class="demo">
  <h2>{{title}}</h2>
  <input v-model="title">
  <ul>
    <li v-for="pet in tab">{{ pet }}</li>
  </ul>
</div>
```



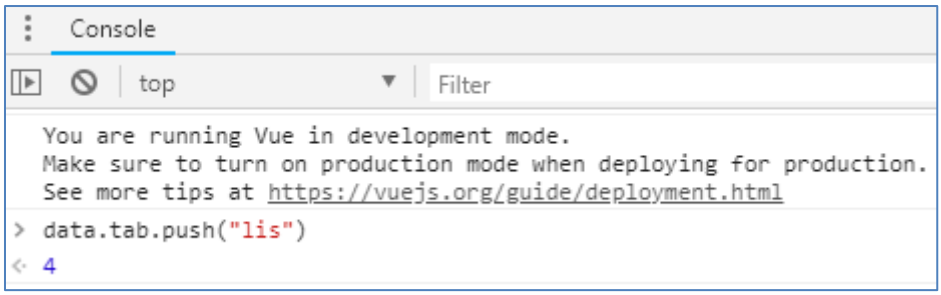
Vue zapewnia dostęp do bieżącego indeksu tablicy (index), który można wykorzystywać w wyrażeniach:

```
      <ul>
        <div v-for="(pet, index) in tab">{{ index }}. {{ pet }}</div>
      </ul>
```



Dodając w konsoli nowy element do tablicy natychmiast powinien zostać wyświetlony:

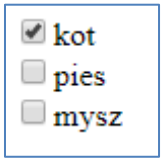
[11] https://vuejs.org/v2/guide/list.html



Bardzo prawdopodobne, że nasza aplikacja będzie pracować na bardziej złożonych danych, np. tabela obiektów zawierające przynajmniej dwie właściwości:

```
const data = {
  title: 'Moja aplikacja Vue.js',
    tab: [{ name: 'kot', checked: true },
        { name: 'pies', checked: false },
        { name: 'mysz', checked: false }]
};
```

```
<li v-for="pet in tab">
    <label>
        <input type="checkbox" v-model="pet.checked">{{pet.name}}
    </label>
</li>
```



Dyrektywa `v-bind:<nazwa_atrybutu>` pozwala na powiązanie atrybutu elementu z wyrażeniem:

```
const data = {
      ...
  path: "https://vuejs.org/images/",
  filename: "logo.png"
};
```

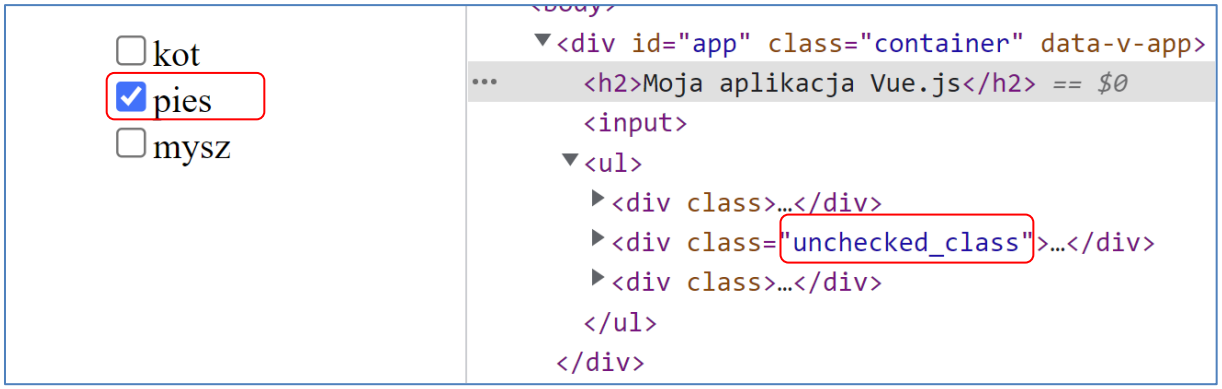
```
<img v-bind:src="path+filename" height="42"/>
```

Posiada wersję skróconą, składającą się z pojedynczego znaku `:`.

```
<img :src="path+filename" height="42"/>
```

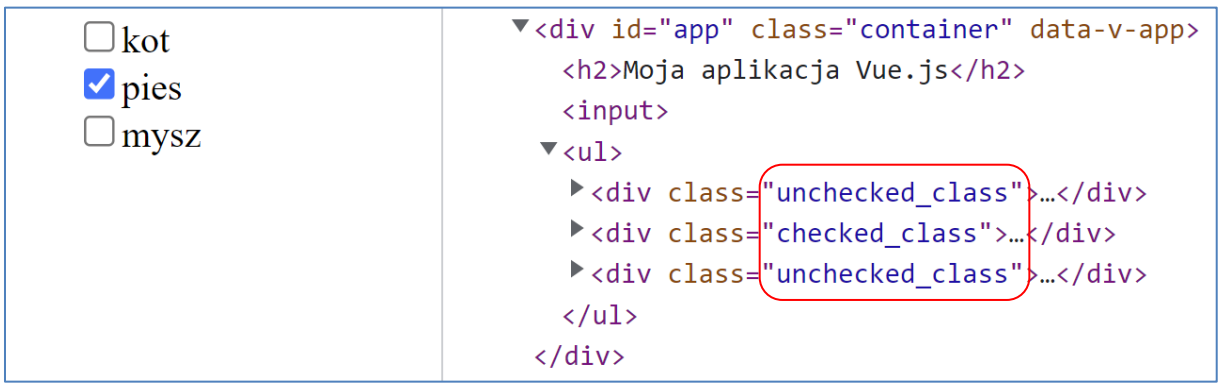
Kiedy wiążemy atrybuty `class` lub `style`, możemy to uzależnić od warunku:

```
<ul>
    <div v-for="pet in tab" :class="{'unchecked_class':pet.checked}">
        <div class="checkbox">
            <label>
                <input type="checkbox" v-model="pet.checked">{{pet.name}}
            </label>
        </div>
    </li>
</ul>
```



Możemy też przełączać się między różnymi klasami/stylami:

```
<ul>
    <li v-for="pet in tab" :class="[pet.checked?'checked_class':'unchecked_class']">
        <div class="checkbox">
            <label>
                <input type="checkbox" v-model="pet.checked">{{pet.name}}
            </label>
        </div>
    </li>
</ul>
```



Dyrektywa `v-if` pozwala na warunkowe wyświetlanie elementu. Warunkiem jest dowolne wyrażenie i może zawierać właściwości modelu danych.

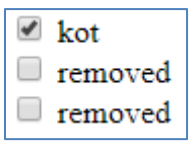
```
<li v-for="pet in tab"
:class="[pet.checked?'checked_class':'unchecked_class']">
    <div class="checkbox">
        <label>
            <input type="checkbox" v-model="pet.checked">
            <span v-if="pet.checked"> {{pet.name}}</span>
        </label>
    </div>
</li>
```

Dyrektywa `v-show` pozwala na to samo co dyrektywa `v-if` ale zawsze renderuje element w DOM. Jeśli warunek nie jest spełniony to ukryje element (właściwość CSS `display: none`) Zalecana podczas częstych zmian stanu.

```
<li v-for="pet in tab"
:class="[pet.checked?'checked_class':'unchecked_class']">
    <div class="checkbox">
        <label>
            <input type="checkbox" v-model="pet.checked">
            <span v-show="pet.checked"> {{pet.name}}</span>
        </label>
    </div>
</li>
```

Wraz z dyrektywa v-if można używać dyrektywy v-else, która wyświetli element w przypadku niespełnionego warunku:

```
<div class="checkbox">
    <label>
        <input type="checkbox" v-model="pet.checked">
        <span v-if="pet.checked"> {{pet.name}} </span>
        <span v-else > removed </span>
    </label>
</div>
```



Dyrektywa v-on pozwala na obsługę zdarzeń:

```
  methods: {
    increment(event) {
      this.counter += 1; // 'this' odniesie się do 'data'
    }
  }
```

```
<hr>
<button v-on:click="increment">Dodaj 1</button>
<p>Przycisk naciśnięto {{ counter }} razy.</p>
```

W przypadku krótszych metod możemy umieścić ich definicję bezpośrednio w dyrektywie:

```
<button @click="counter++">(Wersja inline) Dodaj 1</button>
```

Podobnie jak miało to miejsce w przypadku dyrektywy v-bind, v-on również posiada wersję skróconą, składającą się ze pojedynczego znaku @.

W prosty sposób możemy tworzyć niestandardowe dyrektywy:

```
app.directive('random-color', {
  mounted: (el) => el.style.color = `#${Math.floor(Math.random() *
    16777215).toString(16)}`
});
```

Zastosowanie:

```
<h2 v-random-color>{{title}}</h2>
```

Efekt:

Moja aplikacja Vue.js

Kolejny przykład z wykorzystaniem wartości przekazanej do dyrektywy:

```
app.directive('ceil', {
  mounted: (el, binding) => el.innerHTML = Math.ceil(binding.value)
});
```

Wykorzystanie i wynik:

```
Sufit z 7,45 to : <span v-ceil='7.45'></span>
```

Sufit z 7,45 to : 8

Obiekt definicji dyrektywy może udostępniać kilka opcjonalnych uchwytów: created, beforeMount, mounted, beforeUpdate, updated, beforeUnmount, unmounted. W przykładach wykorzystano jedynie mounted.

V. Właściwości Obliczone (Computed Properties) i Obserwatory (Watchers)

Cel: Zapoznanie z właściwościami obliczonymi (computed properties) do tworzenia danych pochodnych i z obserwatorami (watch) do reagowania na zmiany danych.

Właściwości obliczone pozwalają na deklaratywne tworzenie wartości, które zależą od innych danych. Są one buforowane (cachowane) – oznacza to, że funkcja obliczająca uruchomi się ponownie tylko wtedy, gdy zmieni się jedna z jej zależności. Jest to znacznie wydajniejsze niż wywoływanie metody w szablonie.

Fragment JS (do dodania do pliku `labVUE.js`):

```
const data = {
  // ... (istniejące dane: title, tab, counter, itd.)
  firstName: 'Jan',
  lastName: 'Kowalski'
};

const MyVueApp = {
  data() { /* ... */ },
  methods: { /* ... */ },

  // Nowa właściwość
  computed: {
    // Ta funkcja będzie wywołana tylko gdy zmieni się
    // this.firstName lub this.lastName
    fullName() {
      console.log('Obliczanie fullName...');
      return `${this.firstName} ${this.lastName}`
    }
  }
};
```

W obiekcie `data` dodajemy `firstName` i `lastName`. Do obiektu `MyVueApp` dodajemy nową właściwość `computed`.

Fragment HTML (do dodania do `div#app` w pliku `labVUE.html`):

```
  <input v-model="firstName" placeholder="Imię">
  <input v-model="lastName" placeholder="Nazwisko">

  <!-- Używamy 'fullName' jak zwykłej danej -->
  <p>Pełne imię: {{ fullName }}</p>
```

Wartość `fullName` będzie automatycznie aktualizowana przy zmianie któregokolwiek z pól `input`, a konsola pokaże log tylko wtedy, gdy wartość faktycznie się zmieni.

Obserwatory pozwalają na wykonanie "efektów ubocznych" (np. pobieranie danych z API, operacje na DOM) w odpowiedzi na zmianę konkretnej właściwości danych.

Fragment JS (do dodania do pliku `labVUE.js`):

```
const MyVueApp = {
  data() { /* ... */ },
```

```
  methods: {
    increment(event) { /* ... */ },

    // Nowa metoda
    getAnswer() {
      this.answer = 'Myślę...';
      // Symulacja zapytania do API
      setTimeout(() => {
        this.answer = 'Odpowiedź brzmi: 42.';
      }, 1500);
    }
  },
  computed: { /* ... */ },

  // Nowa właściwość
  watch: {
    // Ta funkcja będzie wywołana za każdym razem,
    // gdy zmieni się 'question'
    question(newQuestion, oldQuestion) {
      if (newQuestion.includes('?')) {
        this.getAnswer();
      }
    }
  }
};
```

Dodajemy nowe właściwości do `data` oraz nową metodę `getAnswer` i właściwość `watch` do `MyVueApp`.

Fragment HTML (do dodania do `div#app` w pliku `labVUE.html`):

```
  <p>
    Zadaj pytanie:
    <input v-model="question" />
  </p>
  <p>{{ answer }}</p>
```

Po wpisaniu znaku zapytania w polu `input`, właściwość `answer` zmieni się na "Myślę...", a po 1.5 sekundy na "Odpowiedź brzmi: 42.".

VI. Komponenty Vue

Cel: Zapoznanie z procesem tworzenia komponentów Vue.js.

Komponenty są ważnym mechanizmem pozwalający na podział funkcjonalności na mniejsze części i na łatwe ich ponowne użycie.

Podczas tworzenia i rejestracji komponentu jako pierwszy parametr przekazujemy nazwę nowego elementu HTML reprezentującego nasz komponent. Najlepiej z wykorzystaniem notacji Kebab case, ponieważ HTML jest językiem case insensitive:

```
<script>
    const app = Vue.createApp();
    app.component('my-component', {
        template: '<h2>To jest komponent</h2>'
    })
    app.mount('#app');
</script>
```

Użycie komponentu (html):

```
<div id='app'>
    <my-component></my-component>
</div>
```

Wynik:

Umieszczanie szablonu w postaci łańcucha znaków jest złą praktyką. Szablony powinny być definiowane w kodzie HTML i służy do tego element `template`:

```
<template id="component_id">
    <h2>To jest komponent</h2>
</template>
```

W komponencie umieszczamy odwołanie do szablonu.

```
app.component('my-component', {
    template: '#component_id'
});
```

Komponent może posiadać własne dane, jednak należy pamiętać, że właściwość `data` musi być zaimplementowana w postaci metody:

```
app.component('my-component', {
    template: '#component_id',
    data: function () {
        return {
            msg: 'To jest komponent'
        }
    }
});
```

W Vue wszystkie instancje komponentów zawierają referencję do tego samego obiektu, pozwala to aby metody, definicje obliczonych właściwości i uchwyty cyklu życia były tworzone, przechowywane i uruchamiane, tylko raz dla każdej instancji komponentu. Dzięki użyciu funkcji definiującej dane, każda instancja komponentu w momencie tworzenia

instancji komponentu otrzyma swój własny, unikatowy obiekt danych. W przypadku zdefiniowana danych jako wartości zamiast funkcji otrzymamy błąd:

```
▲ ▶[Vue warn]: The data option must be a function. Plain object usage is no longer supported.  vue:2260
    at <MyComponent>
    at <App>
```

Teraz możemy skorzystać w szablonie z wiązania:

```
<template id="component_id">
    <h2>{{msg}}</h2>
</template>
```

Wszystkie komponenty posiadają dostęp do globalnego zasięgu aplikacji, dodajmy zatem właściwość `data` w aplikacji (tutaj również dane musimy zwracać przy pomocy metody):

```
const app = Vue.createApp({
    data: function () {
        return {
            gMsg: "global"
        }
    },
});
```

Aby móc zobaczyć te globalne dane, należy jawnie definiować w komponencie – za pomocą atrybutu `props` – które właściwości modelu danych komponentu nadrzędnego albo globalne będą widoczne:

```
app.component('my-component', {
    template: '#component_id',
    data: function () {
        return {
            msg: 'To jest'
        }
    },
    props: ['msg2']
});
```

Nasz dotychczasowy komponent będzie teraz w widoku wyświetlał dodatkowe dane z komponentu nadrzędnego:

```
<template id='component_id'>
    <h2>{{msg}} {{msg2}}</h2>
</template>
```

Do wiązania właściwości modelu danych z instancją komponentu służy dyrektywa `v-bind`:

```
<my-component :msg2='gMsg'></my-component>
```

Wynik:

Komponenty można zagnieżdżać w sobie. Nic nie stoi na przeszkodzie aby komponent składał się z wielu różnych komponentów.
Dodajmy szablon zawierający kontrolkę `input`:

```
<template id="inp_id">
    <input v-bind:value="msgIn" />
</template>
```

Implementacja i rejestracja komponentu pozwalającego na wprowadzanie danych:

```
app.component('inp-component', {
    template: '#inp_id',
    props: ['msgIn'],
});
```

Szablon multikomponentu zawierający dwa powyższe. Dane komponentu `inp-component` są powiązane z komponentem `my-component` właściwością `mMsg`:

```
<template id='multi_comp_id'>
    <div>
        <inp-component :in_msg="mMsg" ></inp-component>
        <my-component :msg2="mMsg"></my-component>
    </div>
</template>
```

Utworzenie i rejestracja multikomponentu, będzie on zawierał jako dane właściwość `mMsg` do komunikacji pomiędzy dwoma komponentami:

```
app.component('multi-component', {
    template: '#multi_comp_id',
    data: function () {
        return {
            mMsg: 'komponent'
        }
    },
})
```

Użycie mulitikomponentu w kodzie jest bardzo proste i czytelne:

```
<div id='app'>
    <multi-component></multi-component>
</div>
```

Jeśli chcielibyśmy aby tekst kontrolki `input` po uruchomieniu zawierał napis odpowiadający zmiennej `mMsg`, naszego multikomponentu musimy powiązać `:value` kontrolki z zewnętrzną właściwością widoczną w komponencie `inp-component`, w poniższym przypadku nadajmy jej nazwę. `in_msg`:

```
<template id="inp_id">
    <input :value="in_msg" @input="onInput" />
</template>
```

```
app.component('inp-component', {
    template: '#inp_id',
    props: ['in_msg'],
});
```

W kolejnym kroku należy właściwość `in_msg` powiązać ze zmienną `mMsg`:

```
<template id='multi_comp_id' @inpupdated="updatemsg" >
    <div>
        <inp-component :in_msg="mMsg" ></inp-component>
        <my-component :msg2="mMsg"></my-component>
    </div>
</template>
```

Niestety dane z komponentu nie przepływają pomiędzy komponentami tak ja byśmy tego oczekiwali:

Nadal jednak wpisanie czegoś w inpucie nie wyzwala żadnych zmian, dzieje się tak ponieważ domyślnie dane propagowane są tylko w kierunku rodzic → potomek. Aby móc przesłać dane w drugą stronę komponent `inp-component` musi emitować zdarzenie `inpupdated`:

```
app.component('inp-component', {
    template: '#inp_id',
    props: ['in_msg'],
    methods: {
        onInput(event) {
            this.$emit('inpupdated', event.target.value)
            console.log(event.target.value)
        }
    },
    emits: ['inpupdated']
});
```

W szablonie komponentu `inp-component` wiążemy metodę `onInput` ze zdarzeniem `input`:

```
<template id="inp_id">
    <input :value="in_msg" @input="onInput" />
</template>
```

W komponencie nadrzędnym musimy obsłużyć zdarzenie:

```
<template id='multi_comp_id' @inpupdated="updatemsg" >
    <div>
        <inp-component :in_msg="mMsg" ></inp-component>
        <my-component :msg2="mMsg"></my-component>
    </div>
</template>
```

Należy jeszcze zaimplementować zachowanie metody `updatemsg` . Teraz zachowanie się komponentów powinno być poprawne:

Zdarzenia emitowane przez komponent nie są bąbelkują, tak jak ma to miejsce w przypadku natywnych zdarzeń DOM. Można nasłuchiwać tylko zdarzeń emitowanych przez bezpośredni komponent podrzędny. Jeśli zachodzi potrzeba komunikacji między komponentami siostrzanymi lub głęboko zagnieżdżonymi, należy użyć zewnętrznej magistrali zdarzeń lub globalnego rozwiązania do zarządzania stanem.

Sloty to "miejsca" w szablonie komponentu, w które rodzic może wstrzyknąć dowolną treść HTML. Pozwala to na tworzenie elastycznych i reużywalnych komponentów, np. okien modalnych, kart czy paneli.

1. Komponent potomny (definicja slotu):

Tworzymy komponent `fancy-button`, który przyjmuje treść za pomocą znacznika `<slot>`.

```
app.component('fancy-button', {
  template: `
    <button class="fancy-btn">
      <!-- Treść przekazana przez rodzica pojawi się tutaj -->
      <slot></slot>
    </button>
  `
});
```

2. Komponent nadrzędny (użycie slotu):

Teraz możemy użyć komponentu i przekazać do niego treść (np. tekst lub ikonę).

```
<div id="app">
  <!-- Cokolwiek jest pomiędzy znacznikami, trafi do <slot> -->
  <fancy-button>
    Kliknij mnie!
  </fancy-button>

  <fancy-button>
    <!-- Możemy wstawić tu dowolny HTML -->
    <span style="font-weight: bold;">Zatwierdź</span>
  </fancy-button>
</div>
```

Zobaczymy dwa przyciski, jeden z tekstem "Kliknij mnie!", a drugi z pogrubionym tekstem "Zatwierdź".

Możemy stworzyć własny komponent, na którym będzie można używać dyrektywy `v-model`, podobnie jak na natywnym elemencie `<input>`.

Użycie `v-model` na komponencie jest "skrótem" składniowym. Domyślnie, jest to to samo, co ręczne wykonanie dwóch operacji:

1. Przekazanie do komponentu propsa o nazwie `modelValue`.
2. Nasłuchiwanie na zdarzenie, które komponent emituje, o nazwie `update:modelValue`.

1. Komponent potomny (implementacja `v-model`):

Aby nasz komponent był kompatybilny z `v-model`, musi implementować ten protokół (akceptować prop `modelValue` i emitować zdarzenie `update:modelValue`). Tworzymy komponent `custom-input`, który to robi.

```
app.component('custom-input', {
  // 1. Akceptuj prop 'modelValue'
  props: ['modelValue'],
  // 2. Deklaruj emitowane zdarzenie
  emits: ['update:modelValue'],
  // 3. W szablonie, zwiąż :value i emituj @input
  template: `
    <input
      :value="modelValue"
      @input="$emit('update:modelValue', $event.target.value)"
      placeholder="Własny input"
    >
  `
});
```

2. Komponent nadrzędny (użycie `v-model`):

Teraz możemy używać `v-model` na naszym komponencie tak, jak na zwykłym `<input>`.

```
const data = {
  message: 'Cześć Vue!'
  // ... reszta danych
};
```

```
<div id="app">
  <!-- Ten v-model jest teraz połączony z naszym komponentem -->
  <custom-input v-model="message"></custom-input>

  <p>Wiadomość to: {{ message }}</p>
</div>
```

Wpisywanie tekstu w `custom-input` będzie na bieżąco aktualizować właściwość `message` w komponencie nadrzędnym.

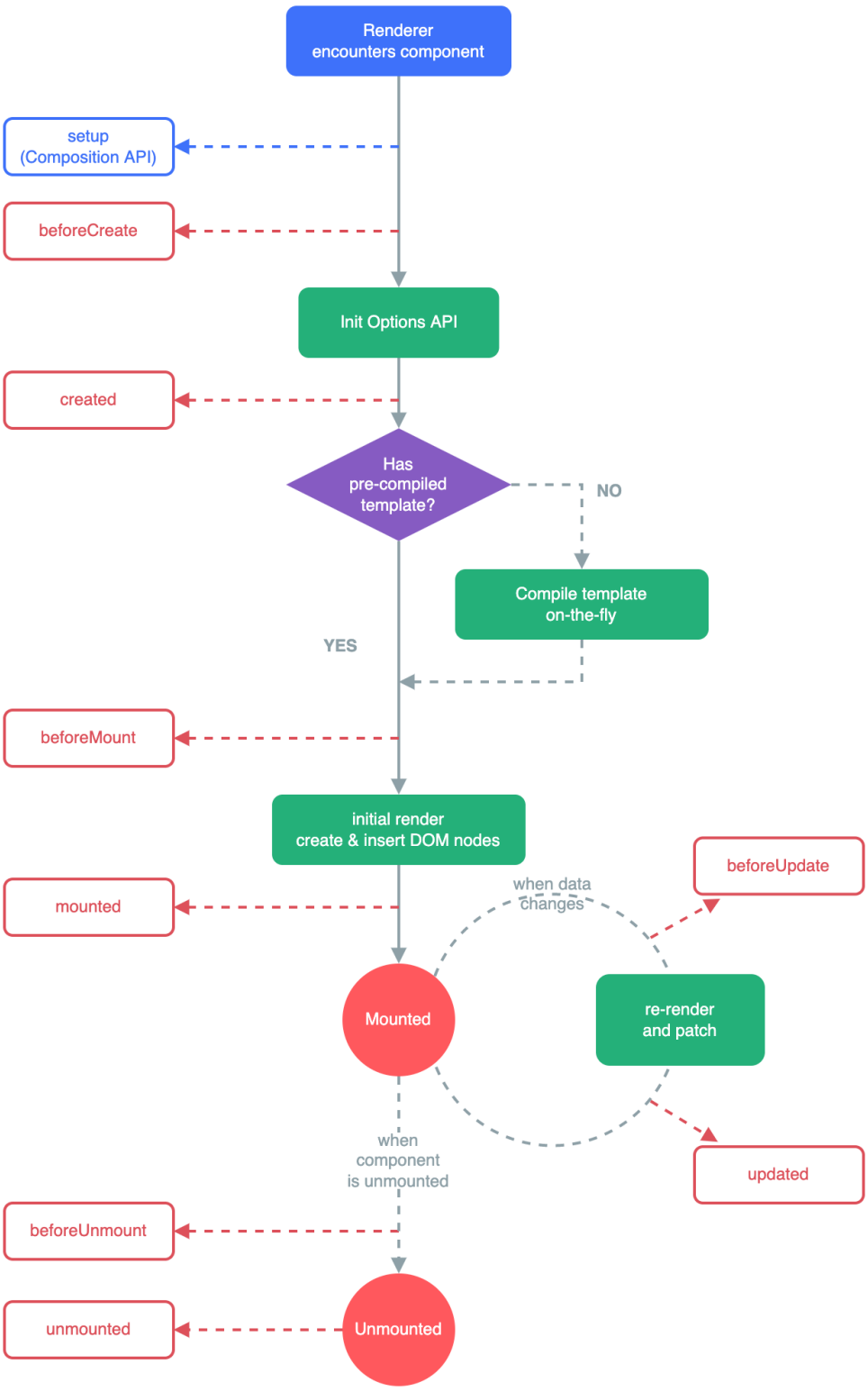
VII. Cykl życia komponentu

Cel: Zapoznanie z cyklem życia komponentów Vue.js.

Każda instancja komponentu podczas tworzenia przechodzi wiele kroków inicjalizacyjnych, takich jak:

- konfiguracja wiązania danych,
- kompilacja szablonu,
- montowanie i odmontowanie instancji do DOM,
- aktualizacja DOM w przypadku zmiany powiązanych danych
- niszczenie.

W trakcie wykonywane są również haki cyklu życia, czyli funkcje dające użytkownikom możliwość dodawania własnego kodu na określonych etapach. Należy pamiętać, że są one wywoływane z kontekstem wskazującym na bieżącą aktywną instancję, która je wywołuje, nie należy zatem używać funkcji strzałkowych ponieważ nie posiadają one this'a. Będzie on traktowany jak każda inna zmienna i leksykalnie przeszukiwany przez zakresy nadrzędne aż do znalezienia. Może to skutkować błędami takimi jak; Uncaught TypeError: Cannot read property of undefined or Uncaught TypeError: this.myMethod is not a function.



Rysunek 1 źródło: https://vuejs.org/assets/lifecycle.MuZLBFAS.png

VIII. Komponenty jedno plikowe Vue

Cel: Zapoznanie z procesem tworzenia komponentów jedno plikowych Vue.js.

W bardziej złożonych projektach preferowane jest zamykanie kodu HTML, JS i styli powiązanych z jednym komponentem w jednym pliku, tzw. komponent jedno plikowy. Rozszerzenie takiego pliku to .vue. Dzięki pośredniej fazie kompilacji możemy użyć preprocesorów takich jak Pug, Babel czy Stylus aby tworzyć czytelniejsze i bogato wyposażone komponent. Przykład prostego, kompletnego komponentu `App.vue`:

```
<template>
    <h2>{{ msg }}</h2>
</template>

<script>
export default {
  data () {
    return {
      msg: 'To jest komponent'
    }
  }
}
</script>

<style scoped>
</style>
```

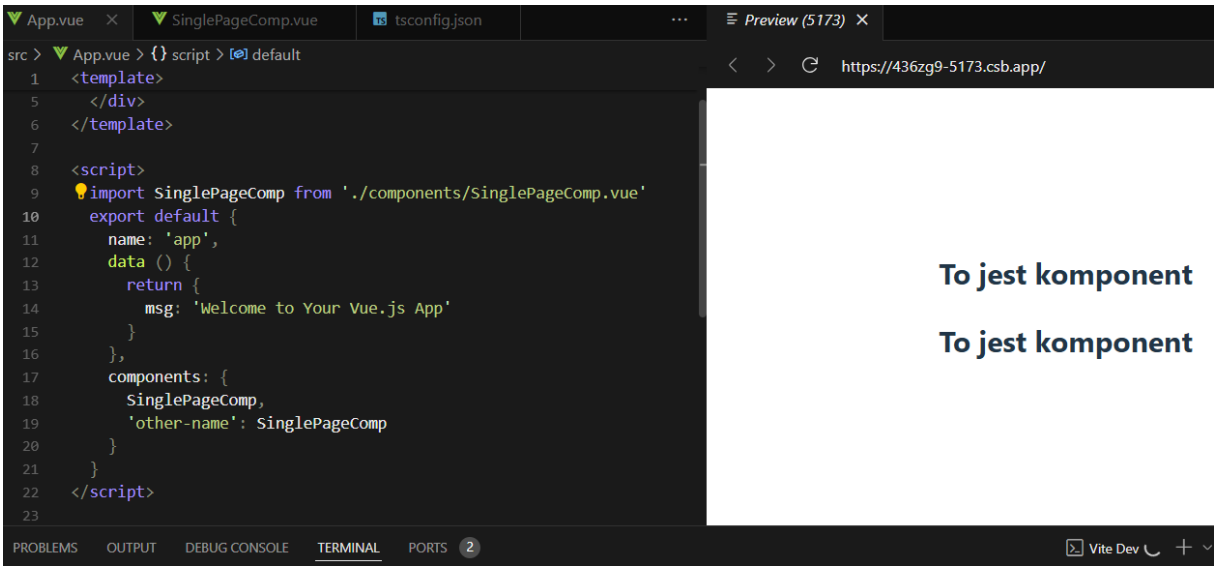
Domyślnie styl komponentu widoczny jest globalnie, do zawężenia jego zasięgu na komponent służy atrybut `scoped`. Działanie komponentu można sprawdzić przy używając edytora online.

Import komponentu wewnątrz komponentu `App.vue`:

```
<template>
  <div id="app">
    <single-page-comp></single-page-comp>
    <other-name></other-name>
  </div>
</template>

<script>
  import SinglePageComp from './SinglePageComp.vue'
  export default {
    name: 'app',
    data () {
      return {
        msg: 'Welcome to Your Vue.js App'
      }
    },
```

```
    components: {
      SinglePageComp,
      'other-name': SinglePageComp
    }
  }
</script>
```

Widok uruchomionej aplikacji:



IX. Composition API

Cel: Zapoznanie z różnicą między Options API a Composition API.

Options API to podstawowa i opisana w poprzednich podpunktach metoda tworzenia komponentów Vue. Polega ona na użyciu zestawu opcji, takich jak dane, obliczone właściwości i metody, w celu zdefiniowania zachowania i stanu komponentu. Jego zaletą jest dobra dokumentacja oraz prostota przez co jest dedykowany dla osób dopiero rozpoczynających przygodę z frameworkiem, czyli dla studentów. W miarę rozrastania się projektu, kiedy komponenty stają się coraz bardziej złożone zarządzanie dużą liczbą opcji może stać się uciążliwe. Może to prowadzić do zjawiska znanego jako „eksplozja opcji", w którym komponent staje się bardzo trudny do utrzymania. Drugim problemem jest słaba możliwość dzielenia logiki między komponentami. Mamy do dyspozycji albo powielanie kodu lub abstrakcje zwane mixin'ami, które posiadają wiele wad.

Composition API to alternatywny sposób budowania komponentów wprowadzony w Vue 3.0, pozwalający rozwiązać niektóre ograniczenia Options API. Pozwala używać funkcjonalnego, reaktywnego stylu programowania do budowania komponentów i oferuje bardziej elastyczny i ekspresyjny sposób definiowania zachowania komponentów. W odróżnieniu od Options API umożliwia to korzystanie z zaawansowanych funkcji JS, takich jak async/await, Promises oraz bibliotek stron trzecich, takich jak RxJS, do budowania komponentów. Niestety nie jest ono kompatybilne wstecz z wersjami Vue 2.6 i starszymi.

Jednym z zalet jest możliwość ponownego wykorzystania logiki w wielu komponentach. W tym celu należy wyizolować kod do zewnętrznego pliku jako Composable, co pozwoli wykorzystać go w osobnych komponentach, a każdy z nich będzie zarządzał własnym

stanem. Poniższy kod pozwala na śledzenie pozycji myszy `onMounted` to zaczep cyklu życia, który rejestruje callback, który zostanie wywołany po zamontowamu komponentu. Metoda `addEventListener` ustawia funkcję update, która będzie wołana za każdym razem gdy nastąpi zdarzenie mousemove:

```
<template>
  Pozycja myszy to: {{ x }}, {{ y }}
</template>

<script setup>
import { ref, onMounted, onUnmounted } from 'vue'
const x = ref(0);
const y = ref(0);
function update(event) {
  x.value = event.pageX
  y.value = event.pageY
}
onMounted(() => window.addEventListener('mousemove', update))
onUnmounted(() => window.removeEventListener('mousemove', update))
</script>
```

Zgodnie z konwencją używamy prefixu `use` do oznaczenia Composable. Mogą one przyjmować jako argumenty wejściowe ref lub funkcje getter, co pozwala na dynamiczne reagowanie na zmiany tych wartości. Zwracają obiekty zawierające zmienne ref, co zapewnia, że wartości te pozostają reaktywne.

```
import { ref, onMounted, onUnmounted } from 'vue'

export function useMousePosition() {
    const x = ref(0)
    const y = ref(0)

    function update(event) {
        x.value = event.pageX
        y.value = event.pageY
    }

    onMounted(() => window.addEventListener('mousemove', update))
    onUnmounted(() => window.removeEventListener('mousemove', update))

    return { x, y }
}
```

Teraz można łatwo re-użyć w komponencie:

```
<template>
  Pozycja myszy to: {{ x }}, {{ y }}
</template>

<script setup>
```

```
import { useMousePosition } from '@/composables/use-mouse-position.js'
const { x, y } = useMousePosition()
</script>
```

Pobieranie danych z API jest dobrym kandydatem na Composable. Metoda `useFetch` pozwala zarządzać stanem asynchronicznym, takim jak pobieranie danych z serwera. Metoda `fetchData` jest udostępniona pod zmienioną nazwą `refetch` aby było jasne, że pobieranie zostało już wywołane.

```
export function useFetch(url) {
    const data = ref(null);
    const error = ref(null);
    const loading = ref(false);

    async function fetchData(url) {
        loading.value = true;
        error.value = null;

        try {
            const response = await fetch(url);
            data.value = await response.json();
        } catch (err) {
            error.value = err;
        } finally {
            loading.value = false;
        }
    }

    fetchData(url);

    return {
        data,
        error,
        loading,
        refetch: fetchData,
    };
}
```

W celu wykorzystania computed property w Composition API używamy funkcji `computed` importowanej z vue.

```
<script setup>
import { ref, computed } from 'vue'

const firstName = ref('Jan')
const lastName = ref('Kowalski')

const fullName = computed(() => {
  console.log('Obliczanie fullName...');
```

```
  return `${firstName.value} ${lastName.value}`
})
</script>
```

```
<template>
  <input v-model="firstName" placeholder="Imię">
  <input v-model="lastName" placeholder="Nazwisko">
  <p>Pełne imię: {{ fullName }}</p>
</template>
```

W celu wykorzystania obserwatorów w Composition API używamy funkcji `watch` importowanej z `vue`.

```
<script setup>
import { ref, watch } from 'vue'

const question = ref('')
const answer = ref('Pytania zwykle kończą się znakiem "?" ;-)')

// Obserwator
watch(question, (newQuestion, oldQuestion) => {
  if (newQuestion.includes('?')) {
    getAnswer();
  }
})

function getAnswer() {
  answer.value = 'Myślę...';
  setTimeout(() => {
    answer.value = 'Odpowiedź brzmi: 42.';
  }, 1500);
}
</script>
```

```
<template>
  <p>
    Zadaj pytanie:
    <input v-model="question" />
  </p>
  <p>{{ answer }}</p>
</template>
```

X. Vue Router[12]

Vue Router pozwala na łatwe pisanie aplikacji SPA (Single-page Application). Instalacja to dodanie kolejnej biblioteki dostępnej np.:

[12] https://router.vuejs.org/guide/

```
<script src="https://unpkg.com/vue-router@4"></script>
```

Jeśli tworzymy projekt za pomocą kreatora `npm create vue@latest`
to jedną z opcji będzie automatyczna instalacja router'a.

Po instalacji wystarczy zmapować nasze komponenty na ścieżki i poinformować Vue Router, gdzie je renderować:

```
<div id="app">
    <h1>Hello App!</h1>
    <p>
      <router-link to="/foo">Go to Foo</router-link>
      <router-link to="/bar">Go to Bar</router-link>
    </p>
    <router-view></router-view>
  </div>
```

```
import { createRouter, createWebHistory } from 'vue-router';

// Definicje komponentów (mogą być proste obiekty)
const Foo = { template: '<div>To jest komponent Foo</div>' }
const Bar = { template: '<div>To jest komponent Bar</div>' }

// Definicja tablicy
const routes = [
  { path: '/foo', component: Foo },
  { path: '/bar', component: Bar }
]
const router = createRouter({
  history: createWebHistory(),
  routes,
});
const app = Vue.createApp({});
app.use(router); // Rejestracja pluginu
app.mount('#app');
```

Wynik:

XI. Wtyczki Vue[13]

Cel: Zapoznanie z procesem tworzenia wtyczek Vue.js.

[13] https://vuejs.org/v2/guide/plugins.html

Wtyczki – jak ma to miejsce w innych produktach – pozwalają na poszerzanie możliwości frameworka: globalne metody, właściwości lub zasoby: nowe dyrektywy, filtry, efekty przejścia itp…

Instalacja wtyczki:

```
npm install|yarn add <nazwa-wtyczki> --save-dev
```

Użycie w instancji aplikacji:

```
import { createApp } from 'vue';
import App from './App.vue';
import VueSomePlugin from 'vue-some-plugin';

const app = createApp(App);
app.use(VueSomePlugin); // Użycie w instancji aplikacji
app.mount('#app');
```

Jako przykład zainstalujmy wtyczkę udostępniającą łatwą w użyciu i wydajną tabelę pozwalającą na: sortowanie, filtrowanie kolumn, paginacja itp.:[14]

```
npm install|yarn add --save vue-good-table-next
```

Import wtyczki lokalnie wewnątrz komponentu:

```
<script lang="ts">
import { defineComponent } from 'vue' // Lub import { ref } jeśli nie używasz
defineComponent
import { VueGoodTable } from 'vue-good-table-next'
import 'vue-good-table-next/dist/vue-good-table-next.css'

// 1. Zaimportuj mapState z Pinia
import { mapState } from 'pinia'
// 2. Zaimportuj swój store (ścieżka jest przykładowa)
import { useContactsStore } from '@/stores/contactsStore'

export default defineComponent({
  components: {
    VueGoodTable
  },
  data() {
    return {
      // Definicja kolumn (przykładowo)
      columns: [
        { label: 'Name', field: 'name' },
        { label: 'Phone', field: 'phone' },
      ]
    }
```

[14] https://borisflesch.github.io/vue-good-table-next/guide/

```
  },
  computed: {
    // 3. Zamapuj stan ze store'a Pinia do tego komponentu
    ...mapState(useContactsStore, ['contactsMode', 'contactsList'])
  }
})
</script>
```

Użycie wewnątrz szablonu:

```
<template>
  <div v-if="contactsMode" class="content">
    <h2>Contacts:</h2>
    <vue-good-table
      :columns="columns"
      :rows="contactsList"
    />
  </div>
</template>
```

Alternatywa przy użyciu `<script setup>`:

```
<script setup lang="ts">
import { ref } from 'vue'
import { VueGoodTable } from 'vue-good-table-next'
import 'vue-good-table-next/dist/vue-good-table-next.css'

// 1. Zaimportuj i użyj store'a
import { useContactsStore } from '@/stores/contactsStore'
const contactsStore = useContactsStore();

// 2. Zdefiniuj kolumny
const columns = ref([
  { label: 'Name', field: 'name' },
  { label: 'Phone', field: 'phone' },
]);
</script>
```

Użycie wewnątrz szablonu:

```
<template>
  <div v-if="contactsStore.contactsMode" class="content">
    <h2>Contacts:</h2>
    <vue-good-table
      :columns="columns"
      :rows="contactsStore.contactsList" />
  </div>
</template>
```

Implementacja komponentu ze slotem:

```
<!-- FancyButton.vue -->
<template>
  <button class="fancy-btn">
    <slot></slot>
  </button>
</template>
<style>
.fancy-btn { background: #42b983; color: white; padding: 10px; border: none; }
</style>
```

Implementacja komponentu kompatybilnego z `v-model`:

```
<!-- CustomInput.vue -->
<script setup>
// 1. Akceptuj prop 'modelValue'
defineProps(['modelValue'])
// 2. Deklaruj emitowane zdarzenie
defineEmits(['update:modelValue'])
</script>

<template>
  <!-- 3. W szablonie, zwiąż :value i emituj @input -->
  <input
    :value="modelValue"
    @input="$emit('update:modelValue', $event.target.value)"
    placeholder="Własny input"
  >
</template>
```

XII. Stan aplikacji Vue (Pinia)

Cel: Zapoznanie z zarządzaniem stanem aplikacji Vue.js.

Wzrost złożoności dużych aplikacji często może być spowodowany dużą liczą elementów stanów rozproszonych na wiele komponentów oraz interakcje między nimi. Z pomocą przychodzi wzorzec Single source of truth (SSOT) realizowany przez Pinia. Magazyn Pinia składa się z:

- stanów – odpowiednik danych w komponencie, obiekt reprezentujący początkowy stan aplikacji, Jest zdefiniowany jako funkcja, która zwraca stan początkowy. Dzięki temu Pinia może działać zarówno po stronie serwera, jak i klienta. Wykorzystując TypeScript można zdefiniować stan za pomocą interfejsu i wpisać wartość zwracaną przez `state`:

```
interface UserInfo {
    name: string
    age: number
}

    interface State {
```

```
    userList: UserInfo[]
    user: UserInfo | null
}

export const useUserStore = defineStore('user', {
    state: (): State => {
        return {
            userList: [],
            user: null,
        }
    },
})
```

Domyślnie można bezpośrednio odczytywać i zapisywać stan, uzyskując do niego dostęp poprzez instancję magazynu (`state`).

```
const store = useCounterStore()
store.count++
```

- getterów – odpowiednik obliczonych wartości w komponencie, w większości przypadków będą polegać tylko na stanie. W przypadku potrzeby wykorzystania innych getterów dostęp do całej instancji magazynu udostępnia `this`, ale w takim przypadku konieczne jest zdefiniowanie typu zwracanego przez funkcję (dotyczy znanego ograniczenia w TypeScript):

```
export const useCounterStore = defineStore('counter', {
    state: () => ({
        count: 0,
    }),
    getters: {
        // automatically infers the return type as a number
        doubleCount(state) {
            return state.count * 2
        },
        // the return type **must** be explicitly set
        doublePlusOne(): number {
            // autocompletion and typings for the whole store ✨
            return this.doubleCount + 1
        },
    },
})
```

Dostęp do gettera można uzyskać bezpośrednio w instancji magazynu:

```
<template>
  <p>Double count is {{ store.doubleCount }}</p>
</template>
```

- akcji – odpowiednik metod w komponencie. Mogą być asynchroniczne, uzyskują dostęp do całej instancji magazynu za pośrednictwem `this`:

```
export const useCounterStore = defineStore('counter', {
    state: () => ({
        count: 0,
    }),
    actions: {
        // since we rely on `this`, we cannot use an arrow function
        increment() {
            this.count++
        },
        randomizeCounter() {
            this.count = Math.round(100 * Math.random())
        },
    },
})
```

Akcje wywoływane są w taki sam sposób jak zwykłe funkcje i metody:

```
<template>
  <!-- Even on the template -->
  <button @click="store.randomizeCounter()">Randomize</button>
</template>
```

Magazyn można zaimportować do dowolnego komponentu: `import { useCounterStore } from './store'` i używać (również w innych magazynach): `const st = useStore()`.

Instalacja Pinia:

```
npm install pinia | yarn add pinia
```

XIII. Tailwind CSS

Cel: Zapoznanie z frameworkiem Tailwind .

Tailwind CSS to popularny framework do budowania interfejsów użytkownika w miejsce tradycyjnych metod korzystania z arkuszy stylów CSS, oferuje dynamiczną konfigurację poprzez zastosowanie klas bezpośrednio w kodzie HTML. Jest prosty w użyciu, i bardzo elastyczny w dostosowywaniu stylu do indywidualnych potrzeb projektu. Dostarcza ogromny zestaw predefiniowanych klas, obejmujących szeroki zakres stylów.

Instalacja Tailwind i jego zależności[15]:

```
npm install -D tailwindcss@latest postcss@latest autoprefixer@latest
```

Wygenerowanie plików `tailwind.config.js` i `postcss.config.js`:

[15] Get started with Tailwind CSS: https://tailwindcss.com/docs/installation/using-vite

```
npx tailwindcss init -p
```

Plik `tailwind.config.js` z katalogu głównym projektu:

```
module.exports = {
    content: [
      "./index.html",
      "./src/**/*.{vue,js,ts,jsx,tsx}",
    ],
    darkMode: false, // or 'media' or 'class'
    theme: {
      extend: {},
    },
    plugins: [],
  }
```

Plik `postcss.config.js` zawierający skonfigurowane `tailwindcss` i `autoprefixer`:

```
module.exports = {
    plugins: {
      tailwindcss: {},
      autoprefixer: {},
    },
  }
```

W celu dołączenia Tailwind w CSS należy utworzyć plik ./src/index.css i użyć dyrektywy @tailwind, aby uwzględnić podstawowe style, komponenty i narzędzia Tailwind:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Tailwind zamieni te dyrektywy w czasie kompilacji na wszystkie style, które generuje na podstawie skonfigurowanego systemu projektowania. W dokumentacji frameworka opisano m.in:

- dodawanie własnych stylów (w tym stylów bazowych i nowych narzędzi) – https://tailwindcss.com/docs/adding-custom-styles;
- ponowne używanie stylów (w tym wyodrębnianie komponentów) – https://tailwindcss.com/docs/reusing-styles.

Finalnie należy zaimportować plik CSS w pliku ./src/main.js:

```
import { createApp } from 'vue'
import App from './App.vue'
import './index.css'
```

```
createApp(App).mount('#app')
```

XIV. Testowanie aplikacji Vue

Cel: Zapoznanie z testowaniem aplikacji Vue.js.

W skład strategii testowania aplikacji Vue, dostępne są trzy poziomy:

1. **testy jednostkowe** dla których rekomendowanym środowiskiem jest Vitest albo Jest. Ten pierwszy jest rozwijany przez zespół odpowiedzialny za Vue/Vite przez co przy minimalnym wysiłku łatwo integruje się z projektami opartymi na Vite i jest bardzo szybki. Przykład testu[16]:

```
// sum.js
export function sum(a, b) {
    return a + b
  }

  // sum.test.js
  import { expect, test } from 'vitest'
  import { sum } from './sum.js'

  test('adds 1 + 2 to equal 3', () => {
    expect(sum(1, 2)).toBe(3)
  })
```

   Zazwyczaj testy jednostkowe nie różnią się od zwykłych testów dotyczących JS/TS, wyjątkiem specyficznym dla Vue może być testowanie funkcji wielokrotnego użytku, czyli Composable's:

```
// counter.js
import { ref } from 'vue'

export function useCounter() {
  const count = ref(0)
  const increment = () => count.value++

  return {
    count,
    increment
  }
}

// counter.test.js
import { useCounter } from './counter.js'

test('useCounter', () => {
  const { count, increment } = useCounter()
  expect(count.value).toBe(0)
```

[16] https://vitest.dev/guide/

```
  increment()
  expect(count.value).toBe(1)
})
```

2. testy komponentów, które można je uznać za formę testowania integracyjnego – Sprawdzają one poprawność montowania, renderowania, interakcji i zachowania kontenera. Testy powinny koncentrować się na publicznych interfejsach komponentu (przeważnie są to emitowane zdarzenia, właściwości i sloty[17]), a nie na wewnętrznych szczegółach implementacji.
   W poniższym przykładzie w pierwszym wierszu następuje import funkcji testowych z Vitest. Drugi wiersz importuje płytką wersję montowania, gdzie „montowanie" oznacza ładowanie pojedynczego komponentu w celu jego przetestowania. Vue Test Utils udostępnia dwie metody pozwalające na tworzenie wrapper'a komponentu Vue:

   - `shallowMount()` — z zaślepionymi komponentami podrzędnymi, szybsza i idealna do testów jednostkowych;
   - `mount()` — z zamontowanymi wszystkimi komponentami podrzędnymi, przydatna w przypadku gdy należy uwzględnić testowanie zachowania komponentów podrzędnych.

   W trzecim wierszu następuje import testowanego komponentu.

```
import { describe, it, expect } from 'vitest'
import { shallowMount } from '@vue/test-utils'
import WeatherHeader from '../WeatherHeader.vue'
```

W kolejnym wierszu znajduje się blok `describe`, który definiuje zestaw testów jednostkowych i może on zawierać wiele testów jednostkowych, przy czym każdy z nich jest definiowany przez blok `it`, którego pierwszym argumentem jest opis funkcji testowej i powinien być krótkim opisem tego, co robi ten konkretny test. Właściwe testowanie odbywa się od kolejnego wiersza i w pierwszym kroku następuje płytkie zamontowanie komponentu Vue z wykorzystaniem wrappera zaślepiającego ewentualne komponenty podrzędne. Pozwala on testować wszystkie aspekty kodu HTML wygenerowanego przez komponent Vue i wszystkie jego właściwości (takie jak dane). Jako drugi parametr przekazywane do `props` komponentu. W drugim kroku następuje rzeczywiste sprawdzenie czy tytuł wygenerowany przez komponent to „Vue Project". Ponieważ następuje porównanie ciągów, zaleca się użycie funkcji `toMatch()`[18].

```
describe('WeatherHeader.vue Test', () => {
  it('renders message when component is created', () => {
    // render the component
    const wrapper = shallowMount(WeatherHeader, {
      props: {
        title: 'Vue Project'
      }
```

[17] https://vuejs.org/guide/components/slots
[18] Inne opcje dostępne w Vitest do wykonywania kontroli opisane są tutaj: https://testdriven.io/blog/vue-unit-testing/

```
    })

    // check that the title is rendered
    expect(wrapper.text()).toMatch('Vue Project')
  })
})
```

Pełną listę dostępnych sprawdzeń można znaleźć w dokumentacji API Vitest.

3. testy End-to-End pozwalają na całościowe testowanie aplikacji. Nie ma atrap ani imitacji, testowany jest rzeczywisty system. Można sprawdzać takie aspekty aplikacji jak:
   - API,
   - kod po stronie klienta/serwera,
   - bazy danych,
   - obciążenie serwera.

   Zapewniona jest wysoka jakość integracji systemu.
   Przykładowy test E2E : https://nightwatchjs.org/blog/introducing-component-testing-in-nightwatch/.