

„ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej”

Politechnika Wrocławska
Wydział Informatyki i Zarządzania



Politechnika
Wrocławska

Zaawansowane Technologie Webowe

Laboratorium

Temat: Javascript frontend framework - Vue
Opracował: mgr inż. Piotr Jóźwiak
Data: lipiec 2020
Liczba godzin: 2 godziny

Table of Contents

Wstęp	3
Instalacja	3
Rozpoczęcie pracy nad projektem	4
Punkt wejścia oraz anatomia plików .vue	5
Tworzenie komponentu	6
Praca z formularzami	10
Event listeners	13
Metody komponentu	13
Emitowanie event'u do parent'a	14
Odbieranie event'u z child komponentu	14
Podstawowa walidacja formularza	15
Conditionals	18
Pobieranie danych z REST API	19

Wstęp

Aktualnie technologie Javascript'owe przeżywają swój złoty wiek. Coraz więcej firm tworzy całkowicie dynamiczne aplikacje internetowe, które często wypierają tradycyjne aplikacje desktopowe. Jest to związane z bardzo dużym rozwojem technologii webowych zorganizowanych wokół *HTML5* oraz *Javascript*.

Aby móc przyjrzeć się jednej z technologii Javascript pod nazwą Vue zaimplementujemy prosty przykład, w którym będziemy dodawać kontakty do książki adresowej. Pozwoli to nam przedstawić podstawowe założenia Javascriptowego frameworka Vue.

Czym jest Vue?

- Vue jest opensourcowym frameworkiem Javascript do implementowania front-endu
- Vue jest warstwą View w modelu MVC (*Model View Controller*)
- Vue jest jedną z najpopularniejszych bibliotek wykorzystywanych na świecie
- Inaczej niż tacy rywale jak Ract czy Angular został zaimplementowany przez zwykłego programistę zamiast wielkiej organizacji jaką jest Facebook czy Google.

Instalacja

Aby móc skorzystać z frameworka Vue najprościej jest podłączyć bibliotekę poprzez odpowiednie podlinkowanie w kodzie HTML do CDN (Content Delivery Network). To rozwiązanie ma swoje plusy, które już omawialiśmy przy okazji Bootstrapa. Aby wykorzystać tę metodę wystarczy dodać poniższy link w sekcji `<head>` html:

```
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
```

Jest to bardzo szybkie rozwiązanie, jednak jego podstawową wadą jest to, że zbudowanie całej hierarchii plików leży po stronie programisty. Zatem w sytuacji, gdy rozpoczynamy pracę nad nowym projektem lepiej jest skorzystać z Vue CLI (<https://cli.vuejs.org/>), które dostarczy nam wielu dodatkowych narzędzi – wraz z serwerem WWW, na którym będziemy mogli uruchamiać naszą aplikację webową w trakcie developmentu. Vue w tym celu wykorzystuje ekosystem Node. Zatem najpierw należy zainstalować **npm**. Następnie instalujemy Vue poprzez polecenie:

```
npm i -g @vue/cli @vue/cli-service-global
```

Po zainstalowaniu Vue CLI możemy skorzystać z polecenia **vue** do utworzenia nowego projektu:

```
vue create address-book-vue-app
```

Vue CLI w trakcie tworzenia projektu zada jedno pytanie, na które w naszym przypadku odpowiadamy opcją: **default (babel, eslint)**.

```
cmd - vue create vue-books-app
```

```
Vue CLI v4.4.6
? Please pick a preset:
> default (babel, eslint)
  Manually select features
```

Kiedy instalacja dobiegnie końca, możemy uruchomić serwer dla naszej aplikacji. W tym celu należy przejść do folderu, który utworzył Vue CLI z naszym projektem oraz wydać polecenie:

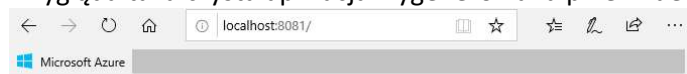
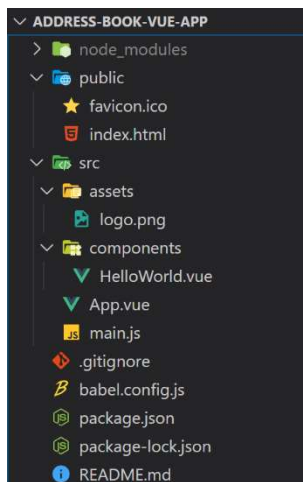
```
cd address-book-vue-app
npm run serve
```

Po uruchomieniu serwera możemy zobaczyć, jak wygląda taka czysta aplikacja wygenerowana przez Vue CLI. Adres aplikacji został wyświetlony w shellu, gdzie uruchomiliśmy serwer WWW. To okno musi pozostać uruchomione przez cały czas pracy nad projektem. Zazwyczaj adres aplikacji to <http://localhost:8080>:

Jeśli powyższa strona jest widoczna w naszej przeglądarce, to oznacza że środowisko jest gotowe do dalszej pracy. Jeśli korzystamy z **Visual Studio Code** warto zainstalować plugin **Vetur**, aby mieć poprawne kolorowanie składni oraz formatowanie.

Dobrym rozwiązaniem będzie także zainstalowanie wtyczki do przeglądarki o nazwie **Vue DevTools**. Umożliwia ona sprawdzenie informacji o komponentach aplikacji. Jest to bardzo pomocne narzędzie dla deweloperów do debugowania aplikacji.

Wtyczka jest dostępna zarówno dla **Chrome** jak i **Firefox**.



Rozpoczęcie pracy nad projektem

Przyjrzyjmy się teraz plikom projektu przygotowanym przez Vue CLI. W folderze z aplikacją odnajdziemy folder o nazwie **public**, który zawiera plik **index.html** oraz folder **src** z plikiem **main.js**, które tworzą punkt wejściowy do aplikacji. Czymś nowym na pewno wydać się mogą pliki z rozszerzeniem **.vue**. W wygenerowanym przykładzie odnajdujemy dwa takie pliki **App.vue** oraz **HelloWorld.vue**. Pierwszy to plik z komponentami, drugi jest plikiem przykładowym odpowiedzialnym za wygenerowanie widoku

strony, który pojawił się już w przeglądarce zaraz po uruchomieniu pustego projektu.

Punkt wejścia oraz anatomia plików .vue

W pliku **src/main.js** powoływany jest framework Vue do życia. Zjrzyjmy do tego pliku:

```
import Vue from 'vue'
import App from './App.vue'

Vue.config.productionTip = false

new Vue({
  render: h => h(App),
}).$mount('#app')
```

Framework w momencie uruchomienia „montuje się” w pliku **index.html** do taga, który posiada id **app**. Znajdziemy ten element w pliku **index.html**. Jest to miejsce, w którym Vue będzie „wstrzykiwało” widok naszego serwisu.

Zadaniem programisty jest oprogramowywanie plików z rozszerzeniem **.vue**. Taki plik zawsze składa się z trzech elementów:

- **<template>** - definicja wyglądu komponentu w języku HTML
- **<script>** - obsługa logiki komponentu w języku Javascript
- **<style>** - ostylowanie komponentu w CSS

Przykładowy pusty komponent wygląda tak:

```
<template></template>

<script>
  export default {
    name: 'component-name',
  }
</script>

<style scoped></style>
```

Przy pierwszym zetknięciu może się to wydawać dość dziwne, przecież od dłuższego czasu promuje się podział implementacji na HTML, CSS oraz JavaScript. Jednakże z punktu widzenia programowania komponentów (nawet wizualnych) wygodniejszym jest utrzymanie kompletu kodu w jednym miejscu zamiast zapisywać go w trzech oddzielnych położeniach. Wynikło to z praktyki na przestrzeni lat pracy z frontendami.

Dane oraz logika działania komponentu znajduje się wewnątrz tagu **<script>**. Minimalnym elementem jest property **name**. Definicje stylu komponentu związane z jego lokalnym widokiem znajdować się będą

wewnątrz tagu **<style>**. Na szczęście mamy tutaj możliwość skorzystania z zawężenia działania tych stylów tylko do danego komponentu poprzez użycie słowa **scoped**.

Aby uprościć pracę, a zarazem nadać jakiś choćby minimalny styl przykładowi, to skorzystamy z gotowego minimalistycznego zestawu definicji. W tym przykładzie główny nacisk stawiamy na funkcjonalność, a nie na wygląd. W tym celu należy dodać w sekcji **<head>** pliku **index.html** poniższe odwołanie:

```
<link rel="stylesheet" href="https://unpkg.com/primitive-ui/dist/css/main.css" />
```

Tworzenie komponentu

Przejdźmy do omówienia sposobu tworzenia komponentu w Vue. Komponent odpowiada za definicję pewnej niewielkiej funkcjonalności, w tym wypadku wizualnej. Stworzymy komponent, który będzie wyświetlał tabelę z danymi osób zapisanych w książce kontaktów.

W tym celu stworzymy plik **src/components/PersonsTable.vue**. Na początek wypełnimy go statycznymi danymi:

```
<template>
  <div id="persons-table">
    <table>
      <thead>
        <tr>
          <th>Imię i nazwisko</th>
          <th>email</th>
          <th>telefon</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>Jan Kowalski</td>
          <td>jan.kowalski@example.pl</td>
          <td>+48 603-788-987</td>
        </tr>
        <tr>
          <td>Maria Koniuszy</td>
          <td>m.kon@example.com</td>
          <td>+48 730-889-966</td>
        </tr>
      </tbody>
    </table>
  </div>
</template>

<script>
  export default {
```

```

    name: 'persons-table',
  }
</script>

<style scoped></style>

```

W Vue konwencja nazewnictwa plików oraz importów zakłada użycie stylu **PascalCase**, np. **PersonsTable**, ale użycie komponentu w szablonie wymaga skorzystania ze stylu **kebab-case**, np.: **<persons-table>**. O rodzajach stylów kodowania możesz przeczytać więcej tutaj: <https://medium.com/better-programming/string-case-styles-camel-pascal-snake-and-kebab-case-981407998841>.

Aby wyświetlić tabelę z kontaktami musimy wyeksportować **PersonsTable** oraz zaimportować go w pliku **App.vue**. W imporcie możemy skorzystać ze znaku **@** aby wskazać folder **src**. Aby poinformować **App.vue** jakie komponenty są dostępne musimy dodać je do właściwości **components**. Teraz możemy wstawić komponent do kodu html poprzez tag **<persons-table>**. Wszystkie powyższe zmiany oraz definicja kilku globalnych stylów została przedstawiona na poniższym listingu dla pliku **src/App.vue**:

```

<template>
  <div id="app" class="small-container">
    <h1>Znajomi</h1>

    <persons-table />
  </div>
</template>

<script>
  import PersonsTable from '@components/PersonsTable.vue'

  export default {
    name: 'app',
    components: {
      PersonsTable,
    },
  }
</script>

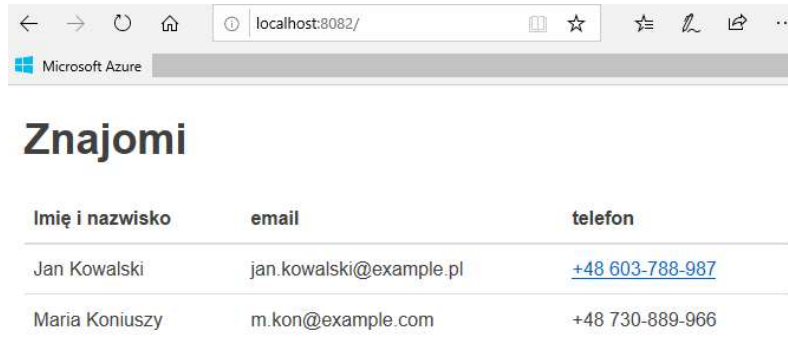
<style>
  button {
    background: #009435;
    border: 1px solid #009435;
  }

  .small-container {
    max-width: 680px;
  }
</style>

```

```
}  
</style>
```

Po uwzględnieniu zmian w przeglądarce powinniśmy zobaczyć poniższą stronę:



Imię i nazwisko	email	telefon
Jan Kowalski	jan.kowalski@example.pl	+48 603-788-987
Maria Koniuszy	m.kon@example.com	+48 730-889-966

W kolejnym kroku zajmiemy się dostarczeniem danych do naszego komponentu. Jak większość frameworków JavaScript'owych dane dostarcza się w postaci obiektów i tablic. W tym celu utworzymy tablicę **persons** wewnątrz metody odpowiedzialnej za dostarczanie danych. Jak łatwo się domyślić nazwa tej metody to **data()**. Jeśli mieliśmy już do czynienia z **React'em**, to funkcja **data()** jest podobna w swym założeniu do Reactowego **state**. Metodę tą dodajemy w pliku **App.vue** według poniższego schematu:

```
import PersonsTable from '@/components/PersonsTable.vue'  
  
export default {  
  name: 'app',  
  components: {  
    PersonsTable,  
  },  
  data() {  
    return {  
      persons: [  
        {  
          id: 1,  
          name: 'Adam Słodowy',  
          email: 'adam.slodowy@zrobtosam.pl',  
          phone: '+48 787 774 664'  
        },  
        {  
          id: 2,  
          name: 'Michał Studencki',  
          email: 'ms@student.pwr.edu.pl',  
          phone: '+48 600 565 454'  
        },  
      ],  
    }  
  }  
}
```



```

      id: 3,
      name: 'Kamila Napokaz',
      email: 'kami2003@h2.pl',
      phone: '+48 609 554 987'
    },
  ],
},
},
}

```

Warto zwrócić uwagę, że każdy element tablicy posiada swój unikalny identyfikator. Zdefiniowaliśmy dane na poziomie **App.vue**, ale chcemy przekazać je do komponentu **PersonsTable**. Można tego dokonać poprzez przekazanie danych w dół obiektu z wykorzystaniem `propertisa`. Atrybuty zaczynające się znakiem `:` umożliwiają przekazywanie danych poprzez bindowanie. Zatem w pliku **App.vue** zdefiniujemy to połączenie w poniższy sposób:

```
<persons-table :personsSource="persons"/>
```

Oznacza to zbindowanie danych z tablicy **persons** definiowanej w **App.vue** do atrybutu **personsSource** zdefiniowanego w komponencie **PersonsTable.vue**. Przejdźmy do zdefiniowania tego atrybutu na komponencie w poniższy sposób:

```

<script>
  export default {
    name: 'persons-table',
    props: {
      personsSource: Array,
    },
  }
</script>

```

W powyższym atrybucie **personsSource** otrzymywać będziemy w komponencie **PersonsTable** dane do wyświetlenia w tablicy. Domyślnie jest to pusta tablica.

Teraz kiedy mamy już dane wewnątrz komponentu musimy zająć się ich wyświetleniem. Do tego celu wykorzystamy pętlę **v-for** w pliku **PersonsTable.vue** w poniższy sposób:

```

<template>
  <div id="persons-table">
    <table>
      <!-- ...thead... -->
      <tbody>
        <tr v-for="person in personsSource" :key="person.id">
          <td>{{ person.name }}</td>
          <td>{{ person.email }}</td>

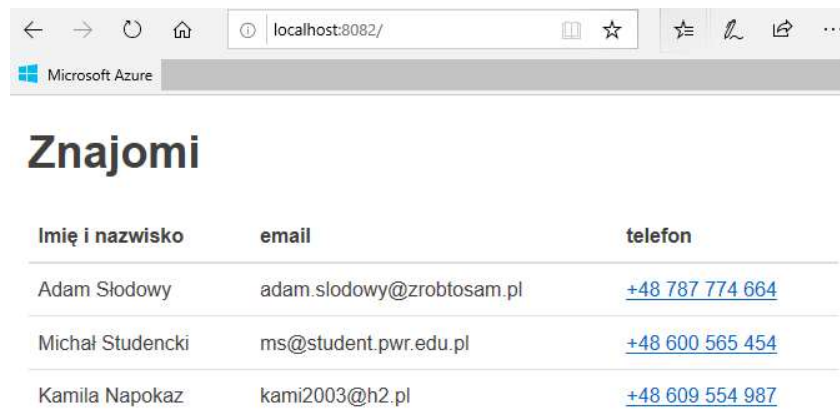
```

```

    <td>{{ person.phone }}</td>
  </tr>
</tbody>
</table>
</div>
</template>

```

Vue tak jak i React posiada wymaganie do unikalnego identyfikowania elementów tablicy. Dlatego skorzystaliśmy z atrybutu **:key** na wierszu tablicy, aby ustawić tę unikalną wartość. Możemy teraz sprawdzić wynik naszych zmian. W przeglądarce otrzymaliśmy dane pochodzące ze zdefiniowanej tablicy:



The screenshot shows a web browser window with the address bar at localhost:8082/. The page title is "Znajomi". Below the title is a table with three columns: "Imię i nazwisko", "email", and "telefon". The table contains three rows of data.

Imię i nazwisko	email	telefon
Adam Słodowy	adam.slodowy@zrobtosam.pl	+48 787 774 664
Michał Studencki	ms@student.pwr.edu.pl	+48 600 565 454
Kamila Napokaz	kami2003@h2.pl	+48 609 554 987

W ten oto sposób skończyliśmy definiowanie komponentu wyświetlającego dane, który realizuje część „Read” z aplikacji CRUD.

Praca z formularzami

Kolejnym bardzo ważnym zagadnieniem jest możliwość modyfikacji danych. W tym celu przyjrzymy się funkcjonalności odpowiedzialnej za dodawanie nowej osoby do naszej książki adresowej z wykorzystaniem formularza.

Na początek utworzymy komponent odpowiedzialny za tę część funkcjonalności. Definicja tego elementu znajdzie się w pliku **src/components/PersonForm.vue**. Definicja będzie składała się z widoku w postaci formularza oraz definicji danych:

```

<template>
  <div id="person-form">
    <form>
      <label>Imię i nazwisko</label>
      <input type="text" />
      <label>Email</label>
      <input type="text" />
      <label>Telefon</label>
      <input type="text" />
      <button>Dodaj kontakt</button>
    </form>
  </div>
</template>

```

```

    </form>
  </div>
</template>

<script>
  export default {
    name: 'person-form',
    data() {
      return {
        person: {
          name: '',
          email: '',
          phone: '',
        },
      }
    },
  },
}
</script>

<style scoped>
  form {
    margin-bottom: 2rem;
  }
</style>

```

Teraz musimy dodać component do **App.vue**. Importujemy komponent oraz wskazujemy jego miejsce wyświetlania w **<template>**:

```

<template>
  <div id="app" class="small-container">
    <h1>Znajomi</h1>

    <person-form />
    <persons-table :personsSource="persons"/>
  </div>
</template>

<script>
  import PersonsTable from '@components/PersonsTable.vue'
  import PersonForm from '@components/PersonForm.vue'

  export default {
    name: 'app',
    components: {
      PersonsTable,

```

```

    PersonForm,
  },
  data() {
    // ...
  },
}
</script>

```

Po zmianach formularz wyświetla się w poniższy sposób:

Imię i nazwisko	email	telefon
Adam Słodowy	adam.slodowy@zrobtosam.pl	+48 787 774 664
Michał Studencki	ms@student.pwr.edu.pl	+48 600 565 454
Kamila Napokaz	kami2003@h2.pl	+48 609 554 987

Na tę chwilę formularz nie potrafi jeszcze dodać danych do książki, ponieważ nie zaimplementowaliśmy jego obsługi. W tym celu skorzystamy z **v-model**, o którym można więcej przeczytać tutaj: <https://vuejs.org/v2/guide/forms.html>. Jest to wbudowana funkcjonalność w framework Vue służąca m. in. do obsługi zdarzeń **onchange** związanych z formularzami. **V-model** definiuje połączenie pomiędzy kontrolką html **<input>** a danymi zdefiniowanymi w metodzie **data()**. Zmodyfikujmy plik **PersonForm.vue** według poniższego schematu:

```

<template>
  <div id="person-form">
    <form>
      <label>Imię i nazwisko</label>
      <input v-model="person.name" type="text" />
      <label>Email</label>
      <input v-model="person.email" type="text" />
      <label>Telefon</label>
    </form>
  </div>
</template>

```

```

    <input v-model="person.phone" type="text" />
    <button>Dodaj kontakt</button>
  </form>
</div>
</template>

```

Event listeners

Do obsłużenia formularza musimy dodać event **onsubmit**. Zrobimy to z wykorzystaniem **v-on:submit** lub równoważnej składni **@submit**. Konwencja jest taka sama dla innych zdarzeń, np. **onclick** będzie definiowany w sposób **v-on:click** lub **@click**. Event **submit** posiada także dodatkową funkcjonalność o nazwie **prevent**, którego użycie jest równoważne z dodaniem **event.preventDefault()** wewnątrz funkcji **submit**. Skorzystamy z tej funkcjonalności, ponieważ nie chcemy wykorzystywać domyślnej formy obsługi formularzy w postaci wygenerowania żądania **GET/POST** z przeglądarki. Wprowadźmy powyższe zmiany do pliku **PersonForm.vue** wskazując na funkcję **handleSubmit** jako odpowiedzialną za przetworzenie formularza:

```

<form @submit.prevent="handleSubmit">
  <!-- ... -->
</form>

```

Metody komponentu

Teraz utworzymy pierwszą metodę na komponencie. Poniżej funkcji **data()**, można dodać obiekt **methods**, który będzie zawierać wszelkie customowe metody. Dodajmy brakującą funkcję **handleSubmit** związaną z obsługą formularza w pliku **PersonForm.vue**. Na tę chwilę nie będzie jeszcze ona nic robić:

```

export default {
  name: 'person-form',
  data() {
    return {
      person: {
        name: '',
        email: '',
        phone: '',
      },
    }
  },
  methods: {
    handleSubmit() {
      console.log('uruchomiono handleSubmit')
    },
  },
}

```

Emitowanie event'u do parent'a

Teraz po wciśnięciu przycisku dodawania kontaktu w konsoli przeglądarki pojawi się odpowiedni komunikat informujący o działaniu funkcji **handleSubmit**. W ten sposób wiemy, że formularz działa. Musimy jeszcze przekazać dane do **App**. Zrobimy to z wykorzystaniem **\$emit**. Emit rozsyła informację o nazwie zdarzenia oraz o danych zdarzenia do nadrzędnego komponentu. Korzystamy z tej funkcjonalności według schematu:

```
this.$emit('name-of-emitted-event', dataToPass)
```

W naszym przypadku, stworzymy event o nazwie **add:person** oraz prześlemy obiekt **this.person**:

```
handleSubmit() {
  this.$emit('add:person', this.person)
},
```

Teraz możemy przetestować działanie formularza. Zanim jednak prześlemy dane z niego, to uruchommy wtyczkę DevTools w przeglądarce. Przejdźmy do zakładki **Vue**, a następnie do sekcji **events**. Zatwierdźmy dane formularza i zaobserwujmy co się zarejestruje we wtyczce:

The image shows a web application on the left and the Vue DevTools event log on the right. The application, titled "Znajomi", has three input fields: "Imię i nazwisko" (containing "Nowa Osoba"), "Email" (containing "iam.new@here.eu"), and "Telefon" (containing "+48 111 222 333"). A green "Dodaj kontakt" button is at the bottom. The DevTools event log on the right shows an event named "add:person" emitted by "<PersonForm>". The event info section shows the payload as an array containing an object with the form data: {email: "iam.new@here.eu", name: "Nowa Osoba", phone: "+48 111 222 333"}.

Jak widać na powyższym obrazku, dane z naszego formularza zostały wyemitowane w postaci zdarzenia. Wprowadzone informacje w formularz znajdują się w obiekcie **payload**.

Odbieranie event'u z child komponentu

Doprowadziliśmy aplikację do stanu, w którym **person-form** wysyła zdarzenie. Teraz musimy odebrać to zdarzenie w komponencie nadrzędnym, aby móc je przetworzyć. W ogólności przechwytywanie zdarzeń na danym komponencie definiuje się według poniższego schematu:

```
<component @name-of-emitted-event="methodToCallOnceEmitted"></component>
```

W takim razie wprowadźmy te zmiany w pliku **App.vue**:

```
<person-form @add:person="addPerson"/>
```

Teraz musimy utworzyć metodę **addPerson** w pliku **App.vue**. Standardowo dodajemy ją do sekcji **methods** za metodą **data()**:

```
methods: {  
  addPerson(person) {  
    this.persons = [...this.persons, person]  
  }  
},
```

Muszę tutaj wtrącić jedną ważną uwagę. W powyższej metodzie obsługującej dodanie nowego pracownika jest pewien zasadniczy błąd. Błąd ten na tę chwilę jest pomijalny, bo w prawdziwej sytuacji dodanie pracownika będzie się odbywało po stronie backendu. Jednakże warto mieć to w świadomości, że nowo dodany pracownik w powyższy sposób nie otrzymuje unikalne id. Jeśli ten kod miałby być w pełni funkcjonalny, to należałoby zadbać o nadanie tego identyfikatora.

Spróbujmy dodać nową osobę do książki. Po wprowadzeniu danych do formularza i zatwierdzeniu go pojawia się nowy wpis w tablicy znajomych. Oczywiście ten wpis będzie żył tak długo jak będzie otwarta przeglądarka. Dane nie są zapisywane w trwały sposób. Za to odpowiada backend, którego jeszcze nie omówiliśmy.

Podstawowa walidacja formularza

Trudno mówić o dobrze zaimplementowanym formularzu bez napisania choćby podstawowej walidacji wprowadzonych danych. Zatem przyjrzyjmy się jak wygląda to zagadnienie w Vue. Do zbudowania mechanizmu walidacji wykorzystamy **computed properties** (<https://vuejs.org/v2/guide/computed.html>), które są funkcjami automatycznie wyliczanymi w momencie wystąpienia zmiany w modelu. Wykorzystanie tego mechanizmu pozwoli uniknąć umieszczania skomplikowanej logiki w szablonie Vue. Walidacja w przykładzie będzie jedynie sprawdzać czy pola nie są puste, jednakże bardzo łatwo będzie z tego miejsca rozbudować funkcjonalność o bardziej skomplikowany mechanizm. Zdefiniujmy te property w pliku **PersonForm.vue** poniżej **methods**:

```
computed: {  
  invalidName() {  
    return this.person.name === ''  
  },  
  
  invalidEmail() {  
    return this.person.email === ''  
  },  
  
  invalidPhone() {  
    return this.person.phone === ''  
  }  
}
```

```
    },
  },

```

Dodatkowo dodamy kilka flag określających stan formularza **PersonForm.vue**. Flaga **submitting** wskazuje stan komponentu oznaczający, że formularz jest w trakcie przesyłania. Flaga **error** wskazuje, że formularz posiada błędy, natomiast flaga **success** oznacza, że formularz nie ma błędów. Implementacja wygląda tak:

```
data() {
  return {
    submitting: false,
    error: false,
    success: false,
    person: {
      name: '',
      email: '',
      phone: '',
    },
  },
}

```

Funkcja obsługująca zatwierdzenie formularza także musi zostać zmieniona. Na początek dodamy funkcję **clearStatus()**, która wyzeruje stan walidacji. Następnie w trakcie obsługi formularza sprawdzimy za pośrednictwem computed properties czy formularz posiada błędy. Przykładowa implementacja w pliku **PersonForm.vue** wygląda następująco:

```
methods: {
  handleSubmit() {
    this.submitting = true
    this.clearStatus()

    //check form fields
    if (this.invalidName || this.invalidEmail || this.invalidPhone) {
      this.error = true
      return
    }

    this.$emit('add:person', this.person)

    //clear form fields
    this.person = {
      name: '',
      email: '',
      phone: '',
    }
  }
}

```



```

    this.error = false
    this.success = true
    this.submitting = false
  },

  clearStatus() {
    this.success = false
    this.error = false
  },
},

```

Pozostało jeszcze zdefiniowanie stylów dla komunikatów o błędzie lub sukcesie przetwarzania formularza. W tym celu w pliku **PersonForm.vue** dodajemy poniższe definicje stylów:

```

<style scoped>
  form {
    margin-bottom: 2rem;
  }

  [class*='-message'] {
    font-weight: 500;
  }

  .error-message {
    color: #d33c40;
  }

  .success-message {
    color: #32a95d;
  }
</style>

```

Na koniec musimy jeszcze wprowadzić modyfikacje do szablonu formularza. Celem jest ustawienie klasy **has-error** dla pola formularza, które nie zostało wypełnione. Wykorzystamy do tego atrybut **:class==**, który zapewnia, że **class** będzie traktowany jako część JavaScript zamiast zwykły tekst html. Dodatkowo wprowadzimy funkcjonalność wyzerowania stanu formularza za każdym razem, gdy użytkownik wprowadzi jakąś zmianę poprzez zdarzenia **@focus** oraz **@keypress**. Przykładowa implementacja wygląda następująco:

```

<form @submit.prevent="handleSubmit">
  <label>Imię i nazwisko</label>
  <input
    v-model="person.name"
    type="text"
    :class="{ 'has-error': submitting && invalidName }"
  >

```

```

    @focus="clearStatus"
    @keypress="clearStatus"
  />
<label>Email</label>
<input
  v-model="person.email"
  type="text"
  :class="{ 'has-error': submitting && invalidEmail }"
  @focus="clearStatus"
/>
<label>Telefon</label>
<input
  v-model="person.phone"
  type="text"
  :class="{ 'has-error': submitting && invalidPhone }"
  @focus="clearStatus"
  @keypress="clearStatus"
/>
<p v-if="error && submitting" class="error-message">
  Proszę wypełnić wskazane pola formularza
</p>
<p v-if="success" class="success-message">
  Dane poprawnie zapisano
</p>
<button>Dodaj kontakt</button>
</form>

```

Conditionals

Pragnę zwrócić uwagę na atrybut **v-if**. Jest to mechanizm conditionals z frameworka Vue. W tym przypadku odpowiada on za wyświetlenie tagu **<p>** po spełnieniu warunku zapisanego w wartości tego atrybutu. Mechanizm ten posiada także atrybuty **v-else** oraz **v-else-if**, które, jak łatwo się domyślić, implementują funkcjonalność wyrażenia warunkowego. Więcej o tej funkcjonalności można przeczytać tutaj: <https://vuejs.org/v2/guide/conditional.html>.

Teraz już funkcjonalność formularza jest kompletna. Możemy ją przetestować i zobaczyć rezultaty. Poniżej przedstawiam widok formularza dla poprawnie wprowadzonych danych oraz błędnego procesowania:

Znajomi

Imię i nazwisko

Email

Telefon

Dane poprawnie zapisano

Dodaj kontakt

Imię i nazwisko	email	telefon
Adam Słodowy	adam.slodowy@zrobtosam.pl	+48 787 774 664
Michał Studencki	ms@student.pwr.edu.pl	+48 600 565 454
Kamila Napokaz	kami2003@h2.pl	+48 609 554 987
Nikodem Kompletny	pisz.do.mnie@lonet.pl	+44 4487730032

Znajomi

Imię i nazwisko

Email

Telefon

Proszę wypełnić wskazane pola formularza

Dodaj kontakt

Imię i nazwisko	email	telefon
Adam Słodowy	adam.slodowy@zrobtosam.pl	+48 787 774 664
Michał Studencki	ms@student.pwr.edu.pl	+48 600 565 454
Kamila Napokaz	kami2003@h2.pl	+48 609 554 987

Pobieranie danych z REST API

Na koniec chciałbym poruszyć jeszcze jedno bardzo ważne zagadnienie. Aby przedstawiony przykład posiadał jakąkolwiek wartość musimy omówić jeszcze podłączenie frontendu do backendu. Ze względu na ograniczenia czasowe laboratorium, omówimy jedynie pobranie danych z serwera i wyświetlenie ich na stronie internetowej. W tym celu skorzystamy z JSON placeholder (<https://jsonplaceholder.typicode.com/>) API, który umożliwia pobranie przykładowych danych do dewelopmentu aplikacji. W ten oto sposób nie musimy się przejmować skąd weźmiemy backend do naszego przykładu. Wykorzystamy żądanie GET z adresu <https://jsonplaceholder.typicode.com/users>, jako źródło danych do tabeli znajomych.

Ogólne schemat funkcjonalności odwołania do REST API został przedstawiony na poniższym przykładzie:

```

async asynchronousMethod() {
  try {
    const response = await fetch('url')
    const data = await response.json()

    // do something with `data`
  } catch (error) {
    // do something with `error`
  }
}

```

Jest to przykład definicji asynchronicznej metody, która generuje żądanie pod adres URL oraz wczytuje odpowiedź w postaci JSON.

Zatem zamienimy naszą dotychczasową implementację z prepopulowaną tablicą osób na wersję z żądaniem GET do serwera. W tym celu dodamy funkcję `getPersons()` do pliku `App.vue` wewnątrz sekcji `methods`. Implementacja tej funkcji wygląda następująco:

```
methods: {  
  //...  
  async getPersons() {  
    try {  
      const response = await fetch('https://jsonplaceholder.typicode.com/users')  
      const data = await response.json()  
      this.persons = data  
    } catch (error) {  
      console.error(error)  
    }  
  },  
},
```

Następnie dodajemy metodę `mounted()` do komponentu `App.vue`. `Mounted` jest funkcjonalnością związaną z cyklem życia formatki. Więcej o tym można przeczytać tutaj: <https://vuejs.org/v2/guide/instance.html#Lifecycle-Diagram>. `Mounted` jest wykonywane w momencie, gdy komponent jest już kompletny i wstawiony do drzewa `DOM`. Jest to funkcjonalność często wykorzystywana do ładowania danych po wejściu na stronę internetową. Wykorzystamy `mounted` do uruchomienia funkcji `getPersons` celem wczytania danych z serwera. Przykładowa implementacja poniżej:

```
export default {  
  name: 'app',  
  components: {  
    PersonsTable,  
    PersonForm,  
  },  
  data() {  
    return {  
      persons: []  
    }  
  },  
  //...  
  
  mounted() {  
    this.getPersons()  
  },  
}
```

Teraz po wejściu do serwisu tablica z osobami jest wypełniona przykładowymi danymi pobranymi z serwera backaendowego:

Imię i nazwisko	email	telefon
Leanne Graham	Sincere@april.biz	1-770-736-8031 x56442
Ervin Howell	Shanna@melissa.tv	010-692-6593 x09125
Clementine Bauch	Nathan@yesenia.net	1-463-123-4447
Patricia Lebsack	Julianne.OConner@kory.org	493-170-9623 x156
Chelsey Dietrich	Lucio_Hettinger@annie.ca	(254)954-1289
Mrs. Dennis Schulist	Karley_Dach@jasper.info	1-477-935-8478 x6430
Kurtis Weissnat	Telly.Hoeger@billy.biz	210.067.6132
Nicholas Runolfsdottir V	Sherwood@rosamond.me	586.493.6943 x140
Glenna Reichert	Chaim_McDermott@dana.io	(775)976-6794 x41206
Clementina DuBuque	Rey.Padberg@karina.biz	024-648-3804

W ten oto sposób przyjrzelśmy się sposobowi generowania żądań REST API. Oczywiście do pełnej funkcjonalności należy jeszcze zaimplementować metod związanych z pełną obsługą