

Wrocław University of Science and Technology



Advanced Web Technologies

Lab

Topic: Javascript frontend framework – Vue

Prepared by: mgr Piotr Jozwiak

Date: July 2020

Number of hours: 2 hours

Table of Contents

Entry	3
Installation	3
Starting work on the project.....	4
Entry point and anatomy of .vue files.....	5
Component creation	6
Working with forms	10
Event listeners.....	13
Component methods	13
Sending an event to the parent	14
Receiving an event from a child component	14
Basic form validation.....	15
conditions.....	18
Retrieving data from REST API	19

Entry

Currently, Javascript technologies are experiencing a golden age. More and more companies are creating completely dynamic web applications that often displace traditional desktop applications. This is related to the very large development of web technologies organized around HTML5 and Javascript.

To be able to look at one of the Javascript technologies called Vue, we will implement a simple example in which we will add contacts to the address book. This will allow us to present the basic assumptions of the Vue Javascript framework.

What is Vue?

- Vue is an open source Javascript framework for implementing front-ends
- Vue is a View layer in the MVC (Model View Controller) model
- Vue is one of the most popular libraries used in the world
- Unlike rivals such as Ract or Angular, it was implemented by an ordinary programmer instead of a large organization such as Facebook or Google.

Installation

To be able to use the Vue framework, the easiest way is to connect the library by appropriate linking in the HTML code to the CDN (Content Delivery Network). This solution has its advantages, which we have already discussed on the occasion of Bootstrap. To use this method just add the link below in the section `<head>html:`

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

This is a very quick solution, but its main disadvantage is that it is the programmer's responsibility to build the entire file hierarchy. Therefore, when we start working on a new project, it is better to use Vue CLI (<https://cli.vuejs.org/>), which will provide us with many additional tools - along with a web server on which we will be able to run our web application during development. Vue uses the Node. So first you need to install npm. Then we install Vue with the command:

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

AfterAfter installing the Vue CLI, we can use the vue command to create a new project:

```
createaddress-book-vue-app
```

While creating the project, Vue CLI will ask one question, which in our case we answer with the option: default (babel, eslint). :

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

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

When the installation is complete, we can start the server for our application. To do this, go to the folder that created Vue CLI with our project and issue the command:

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

After starting the server, we can see what such a clean application generated by Vue CLI looks like. The address of the application was displayed in shell where we started the server. This window must remain open while working on the project. Typically, the address of the application is <http://localhost:8080>:

If the above page is visible in our browser, it means that the environment is ready for further work. If we use Visual Studio Code, it is worth installing the Vetur plugin to have correct syntax coloring and formatting.

It will also be a good idea to install a browser plug-in called Vue DevTools. It allows you to check information about application components. It is a very helpful tool for developers to debug applications.

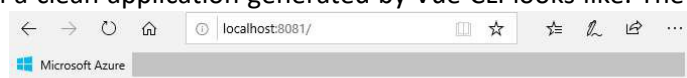
ADDRESS-BOOK-VUE-APP

- > node_modules
- > public
 - ★ favicon.ico
 - index.html
- > src
 - > assets
 - logo.png
 - > components
 - ✓ HelloWorld.vue
 - ✓ App.vue
 - main.js
 - .gitignore
 - babel.config.js
 - package.json
 - package-lock.json
 - README.md

Plug

Isa

available both
for both Chrome and Firefox.



Welcome to Your Vue.js App

For a guide and recipes on how to configure / customize this project, check out the [vue-cli documentation](#).

Installed CLI Plugins

[babel](#) [eslint](#)

Essential Links

[Core Docs](#) [Forum](#) [Community Chat](#) [Twitter](#) [News](#)

Ecosystem

[vue-router](#) [vuex](#) [vue-devtools](#) [vue-loader](#) [awesome-vue](#)

Starting work on the project

Now let's take a look at the project files prepared by the Vue CLI. In the folder with the application, we will find a folder named public, which contains the index.html file and the src folder with the main.js file, which form the entry point to the application. Files with the .vue extension may seem something new. In the generated example, we find two such files App.vue and HelloWorld.vue. The first is the component file, the second is the sample file responsible for generating the view

page that already appeared in the browser as soon as the empty project was launched.

Entry point and anatomy of .vue files

INsrc/main.js file, the Vue framework is brought to life. Let's look at this file:

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

Vue.config.productionTip = false

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

framework at launch "mounts" in the index.html file to the tag that has the app's id. We will find this element in the index.html file. This is where Vue will inject the view of our site.

The programmer's task is to program files with the .vue extension. Such a file always consists of three elements:

- **<template>**-component appearance definition in HTML
- **<script>**- service component logic in Javascript
- **<style>**- styling component in CSS

example of an empty component looks like this:

```
<template></template>

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

<style scoped></style>
```

At first glance, it may seem quite strange, after all, the division of implementation into HTML, CSS and JavaScript has been promoted for a long time. However, from the point of view of programming components (even visual ones), it is more convenient to keep a complete set of code in one place instead of storing it in three separate locations. This resulted from practice over the years of working with frontends.

The data and logic of the component operation is inside the <script> tag. The minimum element is the property name. The component style definitions associated with its local view will be located

inside the `<style>` tag. Fortunately, here we have the opportunity to narrow down the operation of these styles only to a given component by using the word `scoped`.

In order to simplify the work, and at the same time to give some minimal style to the example, we will use a ready-made minimalist set of definitions. In this example, the main emphasis is on functionality, not appearance. To do this, add the following reference to the `<head>` section of the `index.html` file:

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

Component creation

Let's move on to discussing how to create a component in Vue. The component is responsible for defining some small functionality, in this case visual. Let's create a component that will display a table with the data of people saved in the contact book.

To do this, we create the file `src/components/PersonsTable.vue`. To begin with, we'll fill it with static data:

```
<template>
  <div id="persons table">
    <table>
      <thead>
        <tr>
          <th>First name and last name</th>
          <th>e-mail</th>
          <th>Telephone</th>
        </tr>
      </thead>
      <body>
        <tr>
          <etc>Jan Kowalski</etc>
          <etc>jan.kowalski@example.pl</etc>
          <etc>+48 603-788-987</etc>
        </tr>
        <tr>
          <etc>Maria Koniuszy</etc>
          <etc>m.kon@example.com</etc>
          <etc>+48 730-889-966</etc>
        </tr>
      </body>
    </table>
  </div>
</template>

<script>
  export default{
```

```

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

<style scoped></style>

```

In Vue, the file naming convention and imports use the PascalCase style, e.g.

PersonsTable, but using the component in the template requires using the kebab-case style, e.g.:

<persons table> You can read more about types of coding styles

here: <https://medium.com/better-programming/string-case-styles-camel-pascal-snake-and-kebab-case-981407998841>.

To display the table with contacts we need export PersonsTable and import it in the App.vue file. In the import, we can use the @ sign to indicate the src folder. To tell App.vue what components are available, we need to add them to properties components. Now we can insert the component into the html code via the <persons-table> tag. All the above changes and the definition of a few global styles are presented in the following listing for the src/App.vue file:

```

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

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

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

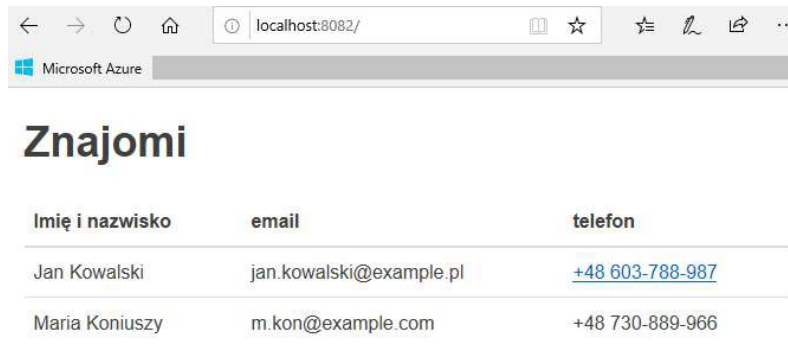
  export default {
    name: 'app',
    components: {
      PersonsTable,
    },
  }
</script>

<style>
  .small-container {
    background-color: #009435;
    border: 1px solid #009435;
  }

```

```
}
</styles>
```

After taking the changes into account in the browser, we should see the following page:



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 two rows of data.

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

In the next step, we will provide data to our component. Like most JavaScript frameworks, data is provided in the form of objects and arrays. To do this, we'll create a persons array inside the method responsible for providing the data. As you can easily guess, the name of this method is `data()`. If we've already dealt with React, then the `data()` function is similar in its assumption to React's state. We add this method in the `App.vue` file according to the following scheme:

```
import PersonsTable from '@components/PersonsTable.vue'

export
  default {
    name: 'app',
    component:
      s: {
        PersonsTable,
      },
    data() {
      return {
        persons: [
          {
            id: 1,
            name: 'Adam Malt',
            'e-mail': 'adam.slodowy@zrobtosam.pl',
            'phone': '+48 787 774 664'
          },
          {
            id: 2,
            name: 'Michael Student',
            'e-mail': 'ms@student.pwr.edu.pl',
            'phone': '+48 600 565 454'
          }
        ]
      }
    }
  }
}
```



```

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

```

Note that each array element has a unique ID. We have defined the data at the App.vue level, but we want to pass it to the PersonsTable component. This can be done by passing data down the object using properties. Attributes starting with a character : enable the transfer of data by binding. So in the App.vue file, let's define this connection as follows:

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

This means binding data from the persons array defined in App.vue to the personsSource attribute defined in the PersonsTable.vue component. Let's define this attribute on the component as follows:

```

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

```

In the personsSource attribute above we will receive data to be displayed in the table in the PersonsTable component. By default it is an empty array.

Now that we have the data inside the component, we need to display it. To do this, we will use the v-for loop in the PersonsTable.vue file as follows:

```

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

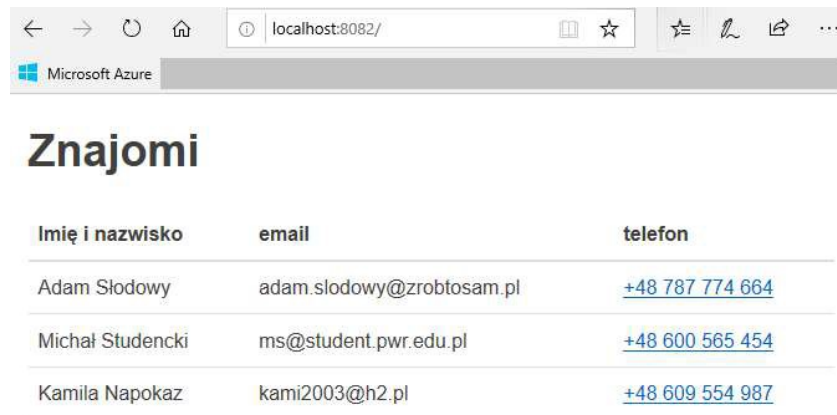
```

```

      <etc>{{person.telephone}}</etc>
    </tr>
  </body>
</table>
</div>
</template>

```

Vue, like React, has a requirement to uniquely identify array elements. So we used the :key attribute on an array row to set this unique value. We can now check the result of our changes. In the browser, we received data from the defined array:



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

This is how we ended up defining the data display component that implements the part "Read" from the CRUD application.

Working with forms

Another very important issue is the ability to modify data. For this purpose, let's look at the functionality responsible for adding a new person to our address book using a form.

To begin with, we will create a component responsible for this part of the functionality. The definition of this element will be in the file `src/components/PersonForm.vue`. The definition will consist of a view in the form of a form and data definition:

```

<template>
  <div id="person-form">
    <forms>
      <label>First name and last name</label>
      <input type="text" />
      <label>E-mail</label>
      <input type="text" />
      <label>Telephone</label>
      <input type="text" />
      <button>add contact</button>
    </forms>
  </div>
</template>

```

```

    </forms>
  </div>
</template>

<script>
  export default{
    name:'person-
    form',date() {
      return{peop
        le:{
          name:'',
          e-mail:'',
          phone:'',
        },
      }
    },
  }
</script>

<stylescoped>
  forms{
    margin-bottom:2 rem;
  }
</styles>

```

Now we need to add a component to App.vue. We import the component and indicate its display location in <template>:

```

<template>
  <div id="app" class="small-container">
    <h1>Friends</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',component
    s:{

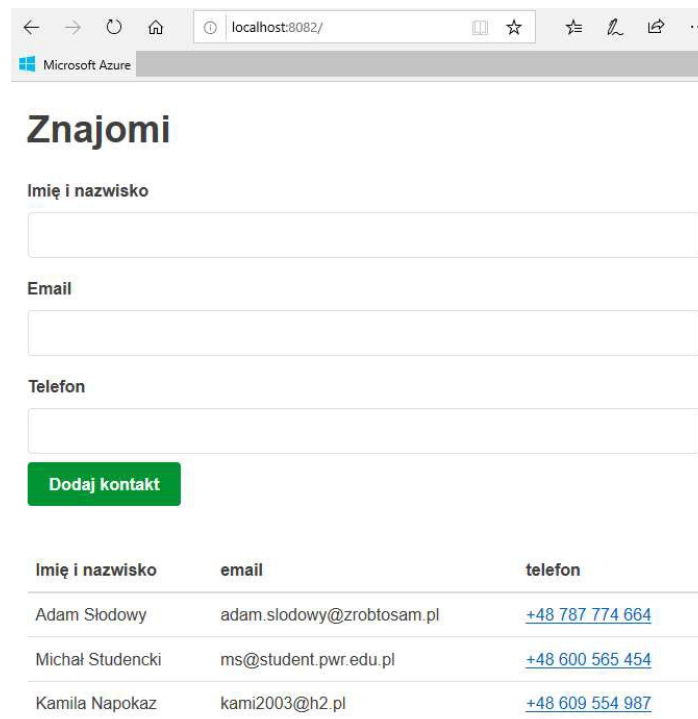
```

```

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

```

After the changes, the form displays as follows:



Imię i nazwisko	email	telefon
Adam Słodowy	adam.slodowy@zrobτοςam.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

At the moment, the form is not able to add data to the book yet, because we have not implemented its support. For this we will use the v-model, which you can read more about here: <https://vuejs.org/v2/guide/forms.html>. This is a built-in functionality in the Vue framework that serves, among others, to handle onchange events related to forms. The V-model defines the connection between the html <input> control and the data defined in the data() method. Let's modify the PersonForm.vue file as shown below:

```

<template>
  <div id="person-form">
    <forms>
      <label>First name and last name</label>
      <input v-model="person.name" type="text" />
      <label>E-mail</label>
      <input v-model="person.e-mail" type="text" />
      <label>Telephone</label>
    </forms>
  </div>
</template>

```

```

    <input v-model="person.telephone" type="text" />
    <button>add contact</button>
  </forms>
</div>
</template>

```

Event listeners

To handle the form, we need to add an onsubmit event. We will do this using `v-on:submit` or the equivalent `@submit` syntax. The convention is the same for other events, e.g. `onclick` will be defined as `v-on:click` or `@click`. Event submit also has an additional functionality called `prevent`, which is equivalent to adding `event.preventDefault()` inside the submit function. We will use this functionality because we do not want to use the default form of form handling in the form of generating a GET / POST request from the browser. Let's make the above changes to the `PersonForm.vue` file pointing to the `handleSubmit` function as responsible for processing the form:

```

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

```

Component methods

Now we will create the first method on the component. Below the `data()` function, you can add a `methods` object that will contain any custom methods. Let's add the missing `handleSubmit` function related to handling the form in the `PersonForm.vue` file. For now, she won't do anything yet:

```

export default {
  name: 'person-
form', data() {
    return {
      peop
      le: {
        name: '',
        e-mail: '',
        phone: '',
      },
    },
  },
  methods: {
    handleSubmit() {
      console.log('handleSubmit started')
    },
  },
}

```

Sending an event to the parent

Now, after pressing the add contact button in the browser console, an appropriate message will appear informing about the operation of the handleSubmit function. This way we know that the form works. We still need to pass data to App. We will do this using \$emit. Emit sends information about the event name and event data to the parent component. We use this functionality according to the scheme:

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

In our case, we will create an event called add:person and pass this.person object:

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

Now we can test how the form works. However, before we send data from it, let's run the DevTools plugin in the browser. Let's go to the Vue tab and then to the events section. Let's validate the form data and observe what will register in the plugin:

The screenshot shows a web application on the left and the Vue DevTools interface on the right. The web 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 button labeled 'Dodaj kontakt' is at the bottom. The DevTools interface on the right shows the 'Vue' tab selected, with the 'events' section expanded. It displays an event named 'add:person' emitted by '<PersonForm>'. The event info shows a payload array containing an object with the following properties: 'email' (iam.new@here.eu), 'name' (Nowa Osoba), and 'phone' (+48 111 222 333).

As you can see in the picture above, the data from our form was emitted in the form of an event. The information entered into the form is in the payload object.

Receiving an event from a child component

We have brought the application to the state where person-form sends the event. Now we need to receive this event in the parent component to be able to process it. In general, capturing events on a given component is defined as follows:

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

Then let's make these changes to the App.vue file:

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

Now we need to create a method **addPerson** in the App.vue file. By default, we add it to the section **methods** after the **data()** method:

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

I must make one important point here. There is a fundamental error in the above method of adding a new employee. This error is negligible at the moment, because in a real situation, adding an employee will take place on the backend side. However, it is worth bearing in mind that a newly added employee does not receive a unique id in the above way. If this code was to be fully functional, then care should be taken to assign this identifier.

Let's try to add a new person to the book. After entering the data into the form and confirming it, a new entry appears in the friends wall. Of course, this entry will live as long as the browser is open. The data is not stored permanently. This is the responsibility of the backend, which we have not yet discussed.

Basic form validation

It is difficult to talk about a well-implemented form without writing at least a basic validation of the entered data. So let's take a look at this issue in Vue. To be built we will use the validation mechanism

computed

properties (<https://vuejs.org/v2/guide/computed.html>), which are functions automatically calculated when a change in the model occurs. Using this mechanism will avoid placing complicated logic in the Vue template. Validation in the example will only check if the fields are not empty, but it will be very easy to extend the functionality with a more complicated mechanism from this place. Let's define these properties in the PersonForm.vue file below methods:

```
computed:{invalid
  Name() {
    return this.person.name=== ''
  },

  invalidEmail() {
    return this.person.e-mail=== ''
  },

  invalidPhone() {
    return this.person.telephone=== ''
  }
},
```

```
    },
  },
},
```

In addition, we will add a few flags defining the state of the PersonForm.vue form. The submitting flag indicates the state of the component, meaning that the form is being submitted. The error flag indicates that the form has errors, while the success flag indicates that the form has no errors. The implementation looks like this:

```
date()
{return{
  submitting:false,e
rror:false,succes
s:false,people:{
  name:'',
  e-mail:'',
  phone:'',
},
}
},
```

The form validation function also needs to be changed. First, we'll add the clearStatus() function to clear the validation state. Then, when handling the form, we will check via computed properties whether the form has errors. An example implementation in the ProsonForm.vue file looks like this:

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

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

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

  //clear form
  fieldsthis.person=
  {
    name:'',
    e-mail:'',
    phone:'',
```



```

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

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

```

It remains to define the styles for the error or success messages of the form processing. To do this, add the following style definitions to the PersonForm.vue file:

```

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

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

  .error-
  message{color:#d33
  c40;
  }

  .success-
  message{color:#3
  2a95d;
  }

```

Finally, we still need to make modifications to the form template. The goal is to set a has-error class for a form field that has not been filled out. For this we will use the :class== attribute, which ensures that class will be treated as part of JavaScript instead of plain html text. In addition, we will introduce the functionality of resetting the form state each time the user makes a change via the @focus and @keypress events. An example implementation looks like this:

```

<forms @submit.prevent="handleSubmit">
  <label>First name and last name</label>
  <input
    v-
    model="person.name" type=
    "text"
  >

```

```

    @focus="clearStatus"@key
    press="clearStatus"
  / />
<label>E-mail</label>
<input
  v-model="person.e-
  mail"type="text"
  :class="{ 'has-error': submitting&&invalidEmail}"
  @focus="clearStatus"
/ />
<label>Telephone</label>
<input
  v-
  model="person.telephone"
  type="text"
  :class="{ 'has-error': submitting&&invalidPhone}"
  @focus="clearStatus"
  @keypress="clearStatus"
/ />
<pv-if="error&&submitting"class="error-
  message">Please fill in the indicated fields of
  the form
</p>
<pv-if="success"class="success-
  message">Data saved correctly
</p>

```

conditions

I want to point out the v-if attribute. This is the conditionals mechanism from the Vue framework. In this case, it is responsible for displaying the <p> tag after the condition stored in the value of this attribute is met. This mechanism also has the v-else and v-else-if attributes, which, as you can easily guess, implement the functionality of the conditional expression. You can read more about this functionality here: <https://vuejs.org/v2/guide/conditional.html>.

Now the functionality of the form is complete. We can test it and see the results. Below is a view of the form for correctly entered data and incorrect processing:

Znajomi

Imię i nazwisko

Email

Telefon

Dane poprawnie zapisano

Dodaj kontakt

Imię i nazwisko	email	telefon
Adam Słodowy	adam.slodowy@zrobτοςam.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@zrobτοςam.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

Retrieving data from REST API

Finally, I would like to raise one more very important issue. In order for the presented example to have any value, we still need to discuss connecting the frontend to the backend. Due to lab time constraints, we will only cover downloading data from the server and displaying it on the website. For this we will use a JSON placeholder (<https://jsonplaceholder.typicode.com/>) API that allows you to download sample data for application development. This way we don't have to worry about where we get the backend for our example. We will use a GET request from the address <https://jsonplaceholder.typicode.com/users>, as the data source for the friends table.

The general diagram of the REST API reference functionality is shown in the example below:

```

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

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

```

This is an example of an asynchronous method definition that generates a request to a URL and reads the response in JSON.

So let's replace our current implementation with a prepopulated table of people with a version with a GET request to the server. To do this, we will add the `getPersons()` function to the `App.vue` file inside the `methods` section. The implementation of this function looks like this:

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

Then we add the `mounted()` method to the `App.vue` component. `Mounted` is a functionality related to the life cycle of the form. You can read more about it here: <https://vuejs.org/v2/guide/instance.html#Lifecycle-Diagram>. `Mounted` is executed when the component is complete and inserted into the DOM tree. This is a functionality often used to load data after entering a website. We will use `mounted` to run the `getPersons` function to load data from the server. Example implementation below:

```
export
default {name: '
app', component
s: {
  PersonsTable,
  PersonForm,
},
data()
{return {
  persons: []
}
},
//...

mounted()
{this.getPersons(
)
```

Now, after entering the site, the table with people is filled with sample data downloaded from the backend server:

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

This is how we looked at how to generate REST API requests. Of course, for full functionality, you still need to implement methods related to full support