1. Learning Outcomes

1.1 Understanding Reactivity in a Web App

- **Explanation:**
  - Define reactivity in the context of a web application.
  - Discuss the concept of data-binding and how it enables automatic UI updates.
  - Explore the reactivity system in VueJS, including the reactivity caveats.
- **Activities:**
  - Create a simple Vue component to demonstrate reactivity.
  - Modify data properties and observe how the UI updates reactively.

1.2 Ability to develop a frontend application using HTML, CSS, JavaScript

- **Explanation:**
  - Emphasize the role of VueJS as a progressive framework, allowing integration into existing projects.
  - Discuss the HTML template syntax, styling with CSS, and enhancing functionality with JavaScript within Vue components.
- **Activities:**
  - Build a basic VueJS application incorporating HTML, CSS, and JavaScript.
  - Customize the styling of components using scoped CSS.

1.3 Understanding how routing works, and how to use it

- **Explanation:**
  - Introduce Vue Router and its significance in single-page applications.
  - Explain the concept of routing, navigation guards, and nested routes.
- **Activities:**
  - Set up Vue Router in a project.
  - Create multiple pages and link them using router links.
  - Implement navigation guards for authentication or data validation.

1.4 Making API Requests with Fetch API (replacing state management)

- **Explanation:**
  - Introduce the Fetch API for making asynchronous HTTP requests.
  - Discuss the basics of making GET, POST, and other types of requests.
  - Explain how to handle responses and errors.
- **Activities:**

  - Integrate the Fetch API into a VueJS application to retrieve data from a mock API.
  - Implement data binding to display the fetched data in the UI.

General Tips for Day 1 Facilitation:

- **Hands-On Exercises:** Continue to incorporate hands-on exercises for each learning outcome, focusing on using the Fetch API.
- **Real-world Examples:** Utilize real-world examples to illustrate the practical applications of VueJS and API integration.
- **Peer Collaboration:** Encourage participants to collaborate and discuss their solutions during the activities.
- **Q&A Sessions:** Allocate time for questions and answers to address any confusion or concerns related to API integration.
- **Resources for Further Learning:** Provide additional resources and references for participants interested in diving deeper into VueJS and API interactions.
- **Feedback Loop:** Maintain a feedback loop for participants to share their thoughts on the learning materials and session structure.

HTML And CSS Refresher

HTML

What is HTML?

HTML stands for HyperText Markup Language. It is the standard markup language for creating web pages. It describes the structure of a web page. HTML consists of a series of elements, which are the building blocks of HTML pages.

HTML elements are represented by tags. HTML tags label pieces of content such as "heading", "paragraph", "table", and so on. Browsers do not display the HTML tags, but use them to render the content of the page.

Type this into a text-editor and save this as .html

```
<!DOCTYPE html>
<html>
<head>
<title>Page Title</title>

</head>
<body>
<h1>This is a Heading</h1>
<p>This is a paragraph.</p>
</body>
</html>
```

Html texts are display from top to bottom. The first line is the doctype, which tells the browser what type of document it is. The second line is the html tag, which tells the browser that this is an html document. The third line is the head tag, which contains the title of the page. The fourth line is the body tag, which contains the content of the page. The fifth line is the h1 tag, which is a heading. The sixth line is the p tag, which is a paragraph.

HTML Tags (the important ones)

Headings

Headings are defined with the h1 to h6 tags. h1 defines the most important heading. h6 defines the least important heading.

```
<h1>This is heading 1</h1>
<h2>This is heading 2</h2>
<h3>This is heading 3</h3>
```

This is heading 1

This is heading 2

This is heading 3

Headers, Paragraphs, and Lists

a standard html website has a header, a footer, and a main content. The header contains the title of the page, and the navigation bar. The footer contains the contact information, and the social media links. The main content contains the main content of the page.

```
<header>
  <h1>Company Name</h1>
  <nav>
    <a href="#">Home</a>
    <a href="#">About</a>
    <a href="#">Contact</a>
  </nav>
</header>

<main>
  <h1>Page Title</h1>
  <p>Page content goes here.</p>
</main>

<footer>
```

```
  <p>Company Name</p>
  <p>123 Street Name</p>
  <p>City, State, Country</p>
</footer>
```

Company Name

Home About Contact

Page Title

Page content goes here.

Company Name

123 Street Name

City, State, Country

Lists

There are two types of list in html, ordered lists and unordered lists. Ordered lists are numbered lists, while unordered lists are bulleted lists.

```
<ul>
  <li>Coffee</li>
  <li>Tea</li>
  <li>Milk</li>
</ul>

<ol>
  <li>Coffee</li>
  <li>Tea</li>
  <li>Milk</li>
</ol>
```

- Coffee
- Tea
- Milk

1. Coffee
2. Tea
3. Milk

Links

Links are defined with the *a* tag. The *href* attribute specifies the URL of the page the link goes to.

```
<a href="www.google.com" target="_blank">Click Me</a>
```

Click Me

The target attribute specifies where to open the linked document. The *target*="*_blank*" attribute specifies to open the linked document in a new tab.

Images

Images are defined with the *img* tag. The *src* attribute specifies the URL of the image.

```
<img src="image.jpg" alt="Image">
```

Image

Tables

Tables are defined with the *table* tag. The *tr* tag defines a row in the table. The *td* tag defines a cell in the table. The *th* tag defines a header cell in the table.

```
<table>
    <tr>
        <th>Firstname</th>
        <th>Lastname</th>
        <th>Age</th>
    </tr>
    <tr>
        <td>Jill</td>
        <td>Smith</td>
        <td>50</td>
    </tr>
    <tr>
        <td>Eve</td>
        <td>Jackson</td>
        <td>94</td>
    </tr>
</table>
```

Firstname	Lastname	Age
Jill	Smith	50
Eve	Jackson	94

Forms

Standard HTML forms are defined with the *form* tag. The *input* tag defines an input field where the user can enter data. The *label* tag defines a label for an input field. The *textarea* tag defines a multi-line input field. The *select* tag defines a drop-down list. The *option* tag defines an option in a drop-down list. The *button* tag defines a clickable button.

```
<form>
    <label for="fname">First name:</label><br>
    <input type="text" id="fname" name="fname"><br>
    <label for="lname">Last name:</label><br>
    <input type="text" id="lname" name="lname">
</form>
```

First name:

Last name:

CSS

Cascading Style Sheets (CSS) is a style sheet language used for describing the presentation of a document written in a markup language such as HTML. CSS is a cornerstone technology of the World Wide Web, alongside HTML and JavaScript.

Create a file called style.css and save it in the same folder as your html file. Add this code to your html file

```
body {
  background-color: lightblue;
}

h1 {
  color: white;
  text-align: center;
}
```

```
<head>
     <link rel="stylesheet" href="style.css">
</head>
<body>
        <h1>Hello World</h1>
</body>
```

Inline styles

You can also put styles directly in your html file. This is called inline styles. Add this code to your html file

```
<body style="background-color:lightblue;">
        <h1 style="color:white;text-align:center;">Hello World</h1>
</body>
```

CSS Selectors

CSS selectors are used to "find" (or select) the HTML elements you want to style. We can divide CSS selectors into five categories: simple selectors, pseudo-class selectors, attribute selectors, pseudo-element selectors, and combinators.

Simple Selectors

Simple selectors select elements based on name, id, class, attribute, and pseudo-class.

```


/* Selects all <p> elements */
p {
  text-align: center;
  color: red;
}

/* Selects the element with id="demo" */

#demo {
  text-align: center;
  color: red;
}

/* Selects all elements with class="intro" */

.intro {
  text-align: center;
  color: red;
}

/* Selects all <a> elements with a target attribute */
```

```css
a[target] {
  background-color: yellow;
}

/* Selects all <a> elements with a target="_blank" attribute */

a[target="_blank"] {
  background-color: yellow;
}

/* Selects every <p> element that is the first child of its parent */

p:first-child {
  color: red;
}

/* Selects every <p> element that is the last child of its parent */

p:last-child {
  color: red;
}

/* Selects every <p> element that is the first <p> element of its parent */

p:nth-child(1) {
  color: red;
}

/* Selects every <p> element that is the second <p> element of its parent */

p:nth-child(2) {
  color: red;
}

/* Selects every <p> element that is the last <p> element of its parent */

p:nth-child(3) {
  color: red;
}

/* Selects every <p> element that is the second child of its parent */

p:nth-of-type(2) {
  color: red;
}

/* Selects every <p> element that is the last child of its parent */

p:last-of-type {
  color: red;
}

/* Selects every <p> element that is the only child of its parent */
```

```css
p:only-child {
  color: red;
}

/* Selects every <p> element that is the only <p> element of its parent */

p:only-of-type {
  color: red;
}

/* Selects every <p> element that is the first <p> element of its parent, if <p>
is the first child of its parent */

p:first-child:first-of-type {
  color: red;
}
```

And many more. CSS is a very powerful tool for styling your web pages. And can be very difficult to understand at first. But with practice, you will be able to master it.

Tailwind CSS

Tailwind CSS is a utility-first CSS framework for rapidly building custom user interfaces. It is a highly customizable, low-level CSS framework that gives you all of the building blocks you need to build bespoke designs without any annoying opinionated styles you have to fight to override.

Tailwind CSS is a great tool to have in your web development arsenal as it allows you to build beatiful and responsive websites in a short amount of time.

For this workshop, we will be using Tailwind CSS generously. You can learn more about Tailwind CSS at https://tailwindcss.com/

Beginning VueJS

Some short notes

VueJS is a JavaScript *framework for building Single Page Application for the web. It uses HTML, CSS, JSS and uses both component-based and declarative programming model to build user interfaces. It is a progressive framework, which means that it is both flexible and *incrementally* adoptable. These are the uses for VueJS

- Enhancing static HTML without a build step
- Embedding as Web Components on any page
- Single-Page Application (SPA)
- Fullstack / Server-Side Rendering (SSR)
- Jamstack / Static Site Generation (SSG)
- Targeting desktop, mobile, WebGL, and even the terminal *A framework is just a structured way of writing JavaScript code. Different frameworks does the same things (which is building interactive web

application) differently, with different goals in mind.

Before we begin

1. Install NPM, as this will be the only way we will install JS library, frameworks, and packages throughout this workshop

Setting up a VueJS project

For this workshop, we will be using Vite.

What is Vite?

I copied this from Wikipedia : **Vite** (French: [vit], like "veet") is a local development server written by Evan You, the creator of Vue.js, and used by default by Vue and for React project templates. It has support for TypeScript and JSX. It uses Rollup and esbuild internally for bundling.

It monitors files as they're being edited and upon file save the web browser reloads the code being edited through a process called Hot Module Replacement (HMR) which works by just reloading the specific file being changed using ES6 modules (ESM) instead of recompiling the entire application.

Vite provides built-in support for server-side rendering (SSR). By default, it listens on TCP port 5173. It is possible to configure Vite to serve content over HTTPS and proxy requests (including WebSocket) to a back-end web server (such as Apache HTTP Server or lighttpd).

Vite for VueJS

Open *PowerShell* and run the command *npm create vite@latest* and press ENTER. This should prompt you to setup a name for your project, and select VueJS when the prompt is shown. Afterwards, you may choose TypeScript, or JavaScript (depending on how much you like to suffer). Vite will then create a new folder containg your VueJS project scaffold. Before you start, make sure to run *cd* into the project folder Vite just made and run the command *npm i* to install any missing dependencies for your project.

After setting up your VueJS project, you should find folder named *src* and a file named *App.vue*. This is where we will be writing our VueJS code. Open *App.vue* and you should see this code

```
<script setup>
import HelloWorld from './components/HelloWorld.vue'
</script>

<template>
  <div>
    <a href="https://vitejs.dev" target="_blank">
      <img src="/vite.svg" class="logo" alt="Vite logo" />
    </a>
    <a href="https://vuejs.org/" target="_blank">
      <img src="./assets/vue.svg" class="logo vue" alt="Vue logo" />
    </a>
  </div>
  <HelloWorld msg="Vite + Vue" />
</template>
```

```
<style scoped>
.logo {
  height: 6em;
  padding: 1.5em;
  will-change: filter;
  transition: filter 300ms;
}
.logo:hover {
  filter: drop-shadow(0 0 2em #646cffaa);
}
.logo.vue:hover {
  filter: drop-shadow(0 0 2em #42b883aa);
}
</style>
```

VueJS uses a component-based programming model, which means that we will be writing our code in components. Components are reusable pieces of code that can be used to build a user interface. In this case, we have a component named *HelloWorld* which is imported from the file *HelloWorld.vue*. We will be writing our code in this file.

Our HelloWorld component looks like this

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

defineProps({
  msg: String,
})

const count = ref(0)
</script>

<template>
  <h1>{{ msg }}</h1>

  <div class="card">
    <button type="button" @click="count++">count is {{ count }}</button>
    <p>
      Edit
      <code>components/HelloWorld.vue</code> to test HMR
    </p>
  </div>

  <p>
    Check out
    <a href="https://vuejs.org/guide/quick-start.html#local" target="_blank"
      >create-vue</a
    >, the official Vue + Vite starter
  </p>
  <p>
```

```
      Install
      <a href="https://github.com/vuejs/language-tools" target="_blank">Volar</a>
      in your IDE for a better DX
    </p>
    <p class="read-the-docs">Click on the Vite and Vue logos to learn more</p>
  </template>

  <style scoped>
  .read-the-docs {
    color: #888;
  }
  </style>
```

The file has three sections, the *script* section, the *template* section, and the *style* section. The *script* section is where we will be writing our JavaScript code. The *template* section is where we will be writing our HTML code. The *style* section is where we will be writing our CSS code.

Understanding Reactivity in a Web App

In the *script* section, we have this code

```
const count = ref(0)
```

This is a variable named *count* that is initialized to 0. This variable is reactive, which means that if the value of *count* changes, the UI will automatically update to reflect the new value. This is the concept of reactivity in VueJS. In order to use this variable in our HTML code, we need to use the *mustache* syntax. The *mustache* syntax is a way to display the value of a variable in HTML. It looks like this

```
<p>{{ count }}</p>
```

So if you were to put it in our *template* section, it would look like this

```
<template>
<p>{{ count }}</p>
</template>
```

In contrast to regular 'ol JavaScript, we don't need to use the *document.getElementById()* function to get the element we want to change. We can just use the *mustache* syntax to display the value of our variable in HTML. This is the concept of data-binding in VueJS. To appreciate the power of reactivity, we need to see it in regular JavaScript. Open *index.html* and add this code

```
<p id="count"></p>
```

This is a paragraph element with an id of *count*. Now, open *main.js* and add this code

```
let count = 0
const countElement = document.getElementById("count")
countElement.innerHTML = count
```

This is a variable named *count* that is initialized to 0. This variable is not reactive, which means that if the value of *count* changes, the UI will not automatically update to reflect the new value.

To change the value using say a button in regular Javascript, we need to use the *document.getElementById()* function to get the element we want to change. We can then use the *innerHTML* property to change the value of the element. This is the concept of data-binding in regular JavaScript.

See the sample code

```
let count = 0
const countElement = document.getElementById("count")
countElement.innerHTML = count
function increment() {
    count++
    countElement.innerHTML = count
}
```

Based on this code, we can see that using regular javascript to build a web application is tedious and time-consuming. This is why we use frameworks like VueJS to build web applications.

If we write this in VueJS, it would look like this

```
<script setup>
import { ref } from 'vue'
let count = ref(0)
</script>
<template>
<p>{{ count }}</p>
<button @click="count++">Increment</button>
</template>
```

The *@click* is a shorthand for *v-on:click*. This is a directive that listens to the click event on the button. When the button is clicked, the count variable is incremented by 1. The UI will automatically update to reflect the new value of count. You can write your own functions in the script tag, and replace the *count++* with your own function. For example

```
<script setup>
import { ref } from 'vue'
let count = ref(0)
```

```
function Increment() {
    count++
}
</script>
<template>
<p>{{ count }}</p>
<button @click="Increment">Increment</button>
</template>
```

Ability to develop a frontend application using HTML, CSS, JavaScript

We start this part by first creating our own Vue component file. Create a new file named *Counter.vue* in the *src/components* folder. In this file, we will be writing our VueJS code. Open *Counter.vue* and you should see this code

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

const count = ref(0)
</script>

<template>
  <div>
    <button type="button" @click="count++">count is {{ count }}</button>
  </div>
</template>
```

This is a component named *Counter* that is imported from the file *Counter.vue*. We will be writing our code in this file. In order to use this component in our App.vue file, we need to import it. Open *App.vue* and add this code

```
<script setup>
import Counter from './components/Counter.vue'
</script>
```

In our template section, we can now use the *Counter* component. Add this code to the template section

```
<template>
  <div>
    <Counter />
  </div>
</template>
```

Styling with CSS

You can see that our component is looking quite boring. We can add styles to our component to make it look better. Open *Counter.vue* and add this code to the style section

```
<style scoped>
button {
    background-color: #4CAF50; /* Green */
    border: none;
    color: white;
    padding: 15px 32px;
    text-align: center;
    text-decoration: none;
    display: inline-block;
    font-size: 16px;
}
</style>
```

This is a style for our button. The *scoped* attribute means that the style will only be applied to the component it is defined in. This is the concept of scoped CSS in VueJS.

Enhancing functionality with JavaScript within Vue components

Let's take at a Todo List and see how we can implement it using VueJS. Create a new file named *TodoList.vue* in the *src/components* folder. In this file, we will be writing our VueJS code. Open *TodoList.vue* and you should see this code

```
<script setup>
import { ref } from 'vue'
const todos = ref([])
const todo = ref("")
function addTodo() {
    todos.value.push(todo.value)
    todo.value = ""
}
</script>
<template>
    <div>
        <input type="text" v-model="todo" />
        <button @click="addTodo">Add Todo</button>
        <ul>
            <li v-for="todo in todos">{{ todo }}</li>
        </ul>
    </div>
</template>
```

Here in our *script* section, we have a list of todos, and a todo variable that is used to store the value of the input field. We also have a function named *addTodo* that is used to add the todo to the list of todos. In our *template* section, we have an input field that is bound to the todo variable. We also have a button that is bound to the *addTodo* function. We also have a list that is bound to the list of todos.

The v-model directive is used to bind the value of the input field to the todo variable. The v-for directive is used to loop through the list of todos and display them in the list. As usual, our @click directive is used to bind the button to the *addTodo* function.

What are props?

In our *HelloWorld* component, we have this code

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

defineProps({
  msg: String,
})

const count = ref(0)
</script>
```

This is a variable named *msg* that is initialized to an empty string. This variable is a prop, which means that it is passed from the parent component to the child component. In this case, the parent component is *App.vue* and the child component is *HelloWorld.vue*. We can pass the value of *msg* from *App.vue* to *HelloWorld.vue* like this

```
<template>
  <div>
    <HelloWorld msg="Vite + Vue" />
  </div>
</template>
```

Multiple props can be passed from the parent component to the child component. For example

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

defineProps({
  msg: String,
  count: Number
})

const count = ref(0)
</script>
```

This is particularly useful when you are breaking down components into smaller ones. For example, you can break down a Todo List into a Todo Item component. This is how you would do it

```
<script setup>
import TodoItem from './components/TodoItem.vue'
import { ref } from 'vue'

let todo = ref("")
const todos = ref([])
function addTodo() {
    todos.value.push(todo.value)
    todo.value = ""
}
</script>

<template>
    <div>
        <input type="text" v-model="todo" />
        <button @click="addTodo">Add Todo</button>
        <TodoItem v-for="todo in todos" :todo="todo" />
    </div>
</template>
```

For the TodoItem component, we can build it like this

```
<script setup>
import { ref } from 'vue'
defineProps({
todos: Array
})
</script>
<template>
    <ul>
        <li v-for="todo in todos">{{ todo }}</li>
    </ul>
</template>
```

You have now successfully built a Todo List using VueJS. You can now add more features to your Todo List, like a delete button, or a checkbox to mark a todo as completed.

Before we move to routing, and the Fetch API, let's look at a better way of styling our components using Tailwind CSS.

Styling with Tailwind CSS

Tailwind CSS is a collection of utility classes that can be used to style your components. It is a utility-first CSS framework, which means that it is composed of many small utility classes that can be combined to build any design. It is also highly customizable, which means that you can configure it to suit your needs.

Let's take a look back at our Counter component. We can style it using Tailwind CSS like this

```
<template>
  <div>
    <button type="button" @click="count++" class="bg-blue-500 hover:bg-blue-700
text-white font-bold py-2 px-4 rounded">count is {{ count }}</button>
  </div>
</template>
```

This is a button that is styled using Tailwind CSS. You can see that it is quite long and tedious to write. This is why we use Tailwind CSS IntelliSense. Tailwind CSS IntelliSense is an extension for VSCode that provides instant Tailwind CSS support in your editor. It provides features like autocomplete, syntax highlighting, and linting for Tailwind CSS. You can install it from the VSCode Marketplace.

For more information on Tailwind CSS, you can visit their website at https://tailwindcss.com/

Tailwind CSS is a great tool to have in your web development arsenal as it allows you to build beatiful and responsive websites in a short amount of time.

Understanding how routing works, and how to use it

What is routing?

Routing in the context of Single Page Applications is a way to navigate between pages. In a Single Page Application, the browser does not reload the entire page when the user navigates to a different page. Instead, it only loads the content that is needed for that page. This is the concept of routing in Single Page Applications. Traditionally, routing is done on the server-side. The server handles the request and returns the appropriate page. In Single Page Applications, routing is done on the client-side. The client handles the request and returns the appropriate page.

The advantage of routing on the client-side is that it is faster and more responsive. Our web application no longer needs to wait for the server to respond before loading the page. The disadvantage is that it is not SEO-friendly. This is because the server does not know what page the user is on, so it cannot index the page. This is why we need to use server-side rendering to make our Single Page Applications SEO-friendly.

We don't need to worry about SEO for this workshop, so we will be using client-side routing.

What is Vue Router?

Vue Router helps link between the browser's URL/History and Vue's components allowing for certain paths to render whatever view is associated with it.

Setting up Vue Router

To use Vue Router, we need to install it first. Open *PowerShell* and run the command *npm install vue-router@4*. This will install Vue Router in our project. Now, we need to import it in our *main.js* file. Open *main.js* and add this code

```
import { createApp } from 'vue'
import App from './App.vue'
import {createRouter,createWebHashHistory} from "vue-router";
import Home from "./components/Home.vue";
const routes = [
     { path: '/', component: Home },
]

// 3. Create the router instance and pass the `routes` option
// You can pass in additional options here, but let's
// keep it simple for now.
const router = createRouter({
     // 4. Provide the history implementation to use. We are using the hash
history for simplicity here.
     history: createWebHashHistory(),
     routes, // short for `routes: routes`
})
createApp(App).use(router).mount('#app')
```

In our App.vue, we need to add a router-view component. This is where our components will be rendered. Open *App.vue* and add this code

```
<template>
  <div>
    <router-view />
  </div>
</template>
```

And now our routing works. To change the page, we need to add a router-link component. Open *Home.vue* and add this code

```
<template>
  <div>
    <router-link to="/about">About</router-link>
  </div>
</template>
```

This is a link to the about page. When the user clicks on the link, the page will change to the about page. We also need to add a router-view component to the about page. Open *About.vue* and add this code

```
<template>
  <div>
    <h1>About Page</h1>
  </div>
</template>
```

This is a simple about page. You can add more pages to your application by creating a new component and adding it to the routes array in *main.js*.

What are navigation guards?

This is a work in progress. We'll add it soon

Making API Requests with Fetch API

What is the Fetch API?

The Fetch API provides an interface for fetching resources (including across the network). It is a more powerful and flexible replacement for XMLHttpRequest. During ancient times, we used XMLHttpRequest to make API requests. It was a tedious and time-consuming process.

This was how we used it.

```
var request = new XMLHttpRequest()
request.open('GET', 'https://api.github.com/users', true)
request.onload = function () {
  // Begin accessing JSON data here
  var data = JSON.parse(this.response)
  if (request.status >= 200 && request.status < 400) {
    data.forEach((user) => {
      console.log(user.login)
    })
  } else {
    console.log('error')
  }
}
request.send()
```

Although, this problem was somewhat remedied by the introduction of jQuery. jQuery made it easier to make API requests, but it was still tedious and time-consuming.

```
$.ajax({
  url: 'https://api.github.com/users',
  dataType: 'json',
  success: function(data) {
    data.forEach((user) => {
      console.log(user.login)
    })
  }
});
```

The difference between the two is that jQuery is a library, while the Fetch API is a browser API. This means that the Fetch API is built into the browser, while jQuery is not. This is why we need to install jQuery before we can use it. This is how we use the Fetch API.

```
fetch('https://api.github.com/users')
  .then(response => response.json())
  .then(data => {
    data.forEach((user) => {
      console.log(user.login)
    })
  })
  .catch(err => console.log(err))
```

See the difference? The Fetch API is much easier to use than XMLHttpRequest. This is why we will be using the Fetch API for this workshop.

Use the Fetch API to make API requests

Let's take a look at how we can use the Fetch API to make API requests. Open *Home.vue* and add this code

```
<script setup>
import { ref } from 'vue'
let users = ref([])
fetch('https://api.github.com/users')
  .then(response => response.json())
  .then(data => {
    users.value = data
  })
  .catch(err => console.log(err))
</script>
<template>
  <div>
    <ul>
      <li v-for="user in users">{{ user.login }}</li>
    </ul>
  </div>
</template>
```

This is a list of users that is fetched from the GitHub API. Fetch API will be used to make API requests in this workshop. Especially after we built our Go RESTFul API.

POST Requests with Fetch API

We can also use the Fetch API to make POST requests. Let's take a look at how we can do that. Open *Home.vue* and add this code

```
<script setup>
import { ref } from 'vue'
let users = ref([])
let username = ref("")
function addUser() {
  fetch('https://api.github.com/users', {
    method: 'POST',
    body: JSON.stringify({
      username: username.value
    }),
    headers: {
      'Content-type': 'application/json; charset=UTF-8',
    },
  })
    .then((response) => response.json())
    .then((json) => {
      users.value.push(json)
    })
}

</script>
<template>
  <div>
    <input type="text" v-model="username" />
    <button @click="addUser">Add User</button>
    <ul>
      <li v-for="user in users">{{ user.login }}</li>
    </ul>
  </div>
</template>
```

Although this probably wouldn't work, as the GitHub API requires authentication. But you get the idea 😃.