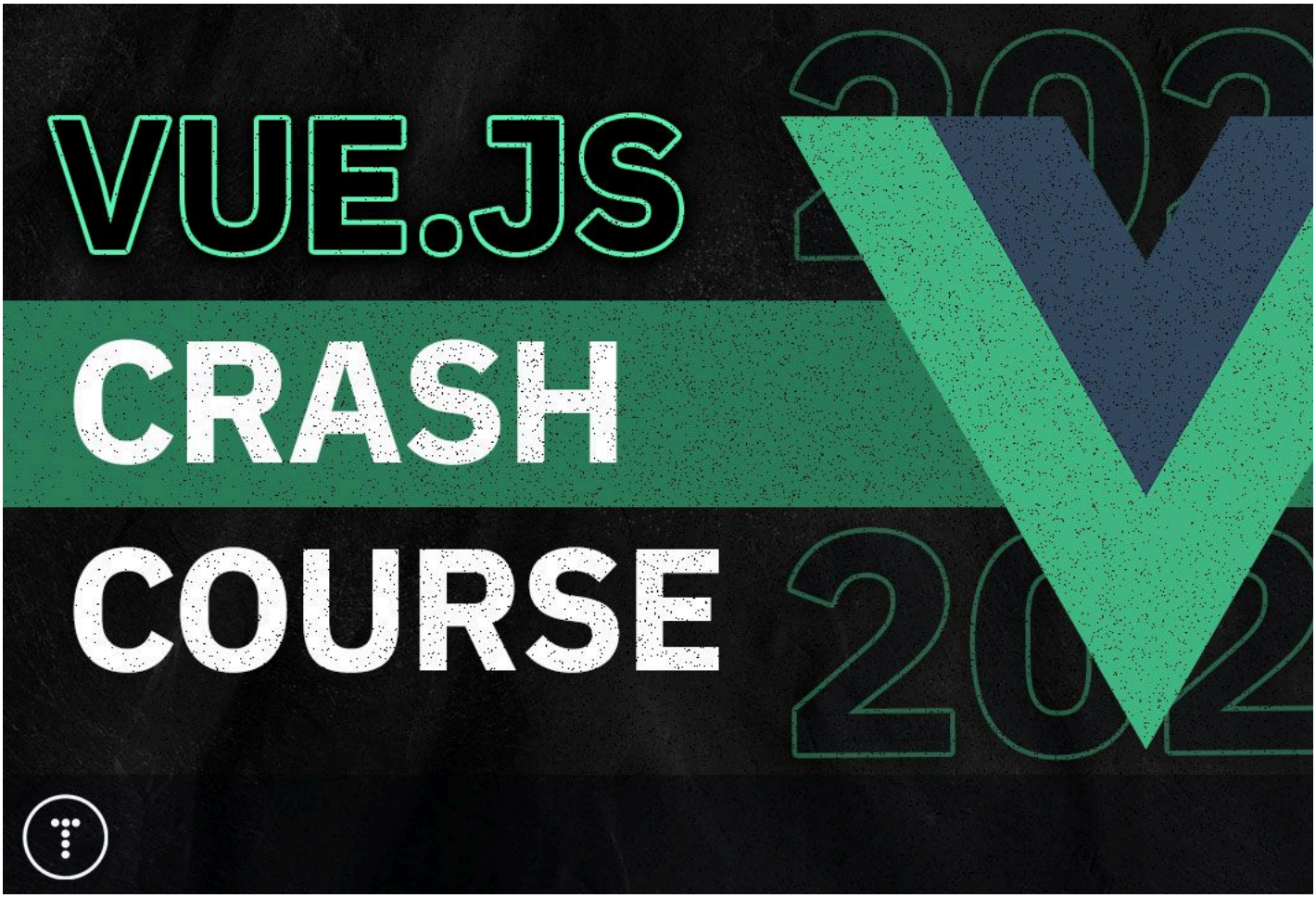
Vue.js Crash Course (Full 2024)

vue Jul 01, 2024



Welcome to my Vue.js crash course. This is a course that is designed to get you up and running with Vue.js as quickly as possible. It is geared toward beginners, so if you've never used Vue.js or you've just dabbled with it, then you're in the right place.

There's going to be a few different parts to this course. First, we'll first go through a few slides and talk about what Vue is and how it works. That won't take very long.

Then I'm going to show you how to get up and running very quickly using the CDN. This is the fastest way to use Vue although you should only use it for really small projects and testing. For larger projects, we use something called "create-vue", which is ultimately what we'll be using.

Next, we'll get setup with create-vue and go over the fundamentals like components, directives, data, methods, etc. We'll look at a very basic task project. I wouldn't even call it an app, it's just a project to experiment with and look at different directives and so on.

Finally, we'll move on to build our main job listing website. Since Vue is a frontend framework, we need a backend for data, so we'll be using JSON Server, which is a library that allows us to create a mock REST API. We'll start with some hardcoded jobs but then we'll add some dynamic data from the API. I'll have the template files with the Tailwind classes available for you in the repo as well. If you took my latest React crash course, it's the same application. That was intentional so that you can compare the two frameworks. I'll also be creating an Angular and a Svelte version.

Here is the repo for the final code: https://github.com/bradtraversy/vue-crash-2024 Here is the full video tutorial: https://youtu.be/VeNfHj6MhgA

So relax, get a coffee or tea and let's learn the Vue.js JavaScript framework.

What Is Vue.js?

Let's start by defining what Vue is. Vue.js is a **progressive JavaScript framework used for building user interfaces and single-page applications**. It's designed to be simple, flexible and incrementally adoptable, meaning you can start using it for small parts of your application and gradually scale up. Vue makes it easy for developers to integrate it into projects of all sizes, whether you're just adding some interactivity to an existing webpage or building a complex web application from scratch. With its reactive data binding and component-based architecture, Vue.js helps developers create dynamic and interactive experiences for users in a straightforward and efficient way.

Vue.js was created by Evan You in 2013. Evan is an independent developer who is currently based in Singapore. At least, that's what I know from his bio. I think the fact that Vue.js wasn't created by a Google or Facebook and is still as popular as it is, speaks volumes about the framework.

Prerequisites

People always ask me "when am I ready to learn a framework?" The answer to that is a tough one because everyone has a different learning abililty and different ways that they learn. I know people that learned JavaScript at the same time as they learned React or Vue. However, I definitely don't think that's optimal for most people. I think you should first HTML, CSS and then the fundamentals of JavaScript. And I don't just mean learn what a variable and a function is. I mean learn to the point where you can build some kind of frontend project without any kind of framework.

I usually recommend the following:

- JS Fundamentals - Loops, functions, objects, etc.
- Events, DOM Manipulation, etc.
- Fetch API & Basic HTTP.
- Arrow functions, high order array methods, destructuring, etc.

I have a course at traversymedia.com and on Udemy called **Modern JavaScript From The Beginning**. It provides everything you need to know and more about JavaScript. So if you want to check that out, I'll have a link in the description.

I would also suggest knowing about NPM (Node Package Manager) because you'll be using this a lot to install 3rd party packages as well as setting up the tools to generate Vue.js projects such as **create-vue**.

The Role Of Frontend Frameworks

Before we talk specifically about Vue, let's talk about the role of frontend frameworks and why they were created.

- **Enhanced User Experience** - Frontend frameworks are designed to make it easier for developers to build user interfaces. They provide a lot of functionality out of the box and allow you to focus on the things that are unique to your application. If you try and build a really interactive interface with vanilla JavaScript, it gets really messy really quick. I'm not saying that you can't do it, but it can be really difficult and it's kind of like reinventing the wheel. -**Organization** - Frontend frameworks are also designed to make it easier for developers to organize their code. The UI is broken into components and each component has it's own state and properties. They're also essential for collaboration. If you have 5 developers create an interface with vanilla JavaScript, you're going to get 5 completely different projects. By using a framework, everyone can be on the same page.
- **Performance** - Frontend frameworks are also optimized for performance. They have a lot of built in features such as virtual DOM, which is a way of rendering the DOM without having to re-render the entire DOM every time. This is a huge performance boost and gives your users a really fast UI.
- **Modularity** - One of the key benefits of frontend frameworks is their modularity. They allow developers to break down their applications into smaller, reusable components. This modular approach not only makes the codebase more manageable and easier to understand but also promotes reusability.

So those are just some of the reasons why we use frontend frameworks. I'm not saying that you should always use one. In fact, that's something I see a lot is people using React or Vue to create a very simple landing page. All that does is over-complicate things. So you have to look at each project individually.

Why Vue?

So why Vue over something like React or Angular?

- **Simplicity & Approachability** - Vue.js is renowned for its simplicity and ease of integration into existing projects. It has a very gentile learning curve and it makes it accessible to developers with varying levels of experience, allowing them to quickly get started and become productive.
- **Flexibility** - Vue.js is designed to be incrementally adoptable, meaning you can introduce it into your project gradually. Whether you need to build a small widget or a full-fledged single-page application (SPA), Vue.js scales effortlessly to meet diverse project requirements. You can also build server-side rendered and static websites with Vue meta frameworks.
- **Performance & Size** - Vue.js offers excellent performance due to its efficient rendering mechanisms, including the virtual DOM. Additionally, Vue's core library is lightweight, which contributes to faster initial load times and better runtime performance. It is known to be one of the fastest frontend frameworks.
- **Component-Based Architecture**: Like other modern frameworks, Vue.js promotes a component-based architecture. Components are self-contained units that can be reused across different parts of your application, fostering maintainability and code reusability.
- **Active Community and Ecosystem**: Vue.js has a really vibrant community and a rich ecosystem of libraries, tools, and plugins. In addition to building single page applications (SPA), you also have meta frameworks like Nuxt.js, which allows you to use server-side rendering (SSR) as well as frameworks like Gridsome, which allows you to build static websites. So Vue has a great ecosystem that is similar to the ecosystem of React.

Choosing Vue.js over React or Angular often boils down to personal preference. You should also take into account what's being used in the industry, especially in your area, but I'm also of the opinion that you should enjoy the framework that you're working with. My advice is always to try the frameworks that interest you. That's why I create these crash courses, so you can get your feet wet and create something and see which one really clicks with you. That's also why I like to create the same app with multiple frameworks. it really gives you a sense of how each framework works.

Vue Components

As with any other frontend JavaScript framework, Vue.js is built around the concept of components. Components are reusable, self-contained pieces of code that can be easily dropped into different projects. Vue components have a very simple structure that is broken into 3 parts.

- The logic/js, which is where you would define any state or data as well as any methods, events, imports, etc.
- The template output, which consists of HTML that will be rendered, however, we can also include dynamic elements in the template such as variables, loops, and conditionals using something called directives. We'll cover these later.
- The style, which is the CSS. You can add "scoped", which means the styling will only pertain to that specific

Here is an example of how Vue components are formatted and structured:

```
<script>
// Vue component definition
export default {
  name: 'SimpleComponent',
  // Optionally, you can include data, methods, computed properties, etc.
};
</script>

<template>
  <div>
    <h2>Hello from Vue.js!</h2>
    <p>This is a simple Vue component.</p>
  </div>
</template>

<style scoped>
/* Scoped CSS for this component */
h2 {
  color: #333;
}
p {
  font-size: 16px;
  line-height: 1.6;
```

```
}
</style>
```

When talking about components and how they're structured, it's important to know that there are two ways to handle the logic in Vue.js. You have the traditional `options API` and the `composition API`. The options API is more straightforward and is a good choice for smaller projects. However, the composition API was released with Vue 3 and is more flexible and allows you to create more complex components. I'll give you an example of both, but overall, I want to use the newer composition API. If you really want to get into the options API, you can also take a look at my older crash course. With both ways, you can define state data, methods, and lifecycle hooks. You can have certain things happen at certain times such as when the component is done loading.

Using Vue.js

There are many ways to use Vue.js. Here are some of the most common.

- **CDN** - The easiest way is to use the CDN. You can include Vue.js in your HTML file and use it, however, I wouldn't suggest this for anything other than a very small and simple project. Maybe for a widget or something.
- **Vue CLI** - The Vue CLI was used for a long time. Its a command line interface to scaffold up a project and it includes a rich collection of official plugins and integrations. However, the Vue CLI is not recommended for new projects anymore and it's in maintenance mode, meaning that it will only receive bug fixes and security updates. If you go to the Vue CLI website, you'll see that it's no longer recommended and they suggest using **create-vue**
- **create-vue** - Create Vue uses the Vite web server and frontend tool. It includes features like hot reloading, out-of-the-box support for TypeScript and other features. It also includes a rich ecosystem of plugins and integrations. We can setup a project with one single command and this is what we'll be using for our job listing app.
- **Nuxt & Gridsome** - Another way to use Vue is with meta frameworks. Just like React has Next.js, Vue has Nuxt.js, which allows you to create server-side-rendered applications. Gridsome is a static-site generator that uses Vue. So these frameworks have their own tools to get setup and they're definitely things I think you should check out but I always suggest learning the core framework first and learn how to build single page applications.

So that's it for the slides. Before we jump in and setup our job listings project, I just want to show you how you can use the CDN to use Vue.js very easily and quickly.

Quick Start With CDN

For anything other than a very simple project, I would suggest that you use something like **create-vue** or **Nuxt**, but you can simply include the CDN, which stands for Content Delivery Network. A CDN is a network of servers that are located all over the world. This means that if you want to use Vue.js, you don't have to download it to your computer. Instead, you can simply include it in your HTML file and use it.

Let's start by just creating a folder called `vue-example` and adding an `index.html` file and add your basic HTML:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vue Example</title>
  </head>
  <body>
    <h1>Hello World</h1>
  </body>
</html>
```

Now add the CDN to the `<head>` of your HTML file:

```
<head>
  <script src="https://unpkg.com/vue@3/dist/vue.global.js"> </script>
</head>
```

I am using VS Code and the Live Server extension, so I will just right click and select "Open With Live Server" and open the project in the browser. You should just see the text "Hello World".

Since we have included the Vue CDN, we can use the Vue.js syntax to create a new Vue app. Add the following code right above the closing tag of the `<body>`:

```
<script>
  const app = Vue.createApp({
    data() {
      return {
        message: 'Hello Vue!',
      };
    },
  });

  app.mount('#app');
</script>
```

Let's go over this code:

- `const app = Vue.createApp()` - We are creating a new Vue app/instance. We are using the `createApp` method, which is a method that is available on the Vue object.
- We are passing in an object that contains the data and methods for our app.
- `data()` - In Vue, the data function is used to define the reactive data properties for the Vue instance. Here, it returns an object with a single property message initialized to 'Hello Vue!'.
- `app.mount('#app')` - We are mounting the Vue instance to the element with the id of app.

Now we can add that element in our HTML:

```
<div id="app">
  <h1>{{ message }}</h1>
</div>
```

The `{{ message }}` is a special Vue syntax that is used to display the value of the message property. This is called interpolation.

We can also add events and methods. And don't worry, I will go over this stuff much more in-depth later. I just want to give you a quick example of how you can use this in your projects.

Let's create a button with an event:

```
<button @event='clickMe'>Click Me</button>
```

Now, we can add a methods option with the `clickMe` method:

```
const app = Vue.createApp({
  data() {
    return {
      message: 'Hello Vue!',
    };
  },
  methods: {
    clickMe() {
      console.log('Button Clicked!')
    }
  }
});

app.mount('#app');
`
```

Now when you click the button, it will fire that method.

This is probably the simplest Vue project that you can create. But it gives you an idea of how you can easily add it to your projects.

create-vue

We can scrap that code and let's create a new project using the `vue-create` tool. If you have ever used React, this is similar to the `create-react-app` tool.

Open a terminal window and run the following:

```
npm create vue@latest vue-crash-2024
```

I am going to choose the following options:

- TypeScript: no
- JSX: no
- Vue Router: I am goping to choose no here because we will implement it ourselves when we need it
- Pinia: no
- Vitest: no
- End to End Testing: no
- ESLint: no
- DevTools: no

It will scaffold up your project.

Open this folder in VS Code or whatever editor you are using.

```
cd vue-crash-2024
code .
```

Vue VS Code Extensions

There are a few Vue.js extensions for VS Code. The ones that I use are the **Vue - Official** and **Vue 3 Snippets** extensions. However, if you find something that you like better, that's fine. Vetur and Volar are quite popular. You just need something that will give you the correct syntax highlighting for .vue files. This is the file extension that Vue components use.

Run The Server

Before we do anything, let's go ahead and install the dependencies and run the dev server. Make sure that you are in the project folder and run the following:

```
npm install
npm run dev
```

By default, this will run on port 5173. I prefer to use port 3000 for my frontend projects, so I am actually going to go into the `vite.config.js` file and change the port to 3000.

```
server: {
  port: 3000,
},
```

Save the file and it should automatically reload the server. Now you can go to `http://localhost:3000` in your browser and you should see the welcome page.

File Structure

Let's take a quick look at the file structure of the project.

package.json

The package.json file is pretty minimal. We just have vue and vue-router since I answered yes to using the router. We also have Vite, which is our server and tooling as well as the Vue plugin as our dev dependencies.

For scripts, we have `dev` to run the dev server, `build` to build the project, and `preview` to preview the production build.

vite-config

The config file is where we have the Vue plugin being initiated. This is where you can add any extra configs, such as the port number, proxies and so on.

index.html

This is the main HTML file. It's pretty simple. We have a div with an id of `app` where our Vue app will be mounted. Since this project uses Vite and not something like Webpack, the actual script is included as a module rather than bundling it all together because Vite includes ES module support.

I am going to change the title:

```
<title>Vue Jobs | Become a Vue Developer</title>
```

public

This is where you can put any static assets such as images, fonts, etc.

src

This is where all of our Vue components and other JavaScript files will go

src/main.js

This is the main entry point for the application. This is where we create the Vue app and mount it to the `#app` div in the `index.html` file.

src/App.vue

This is the main component that will be loaded into the `#app` div in the `index.html` file. It's a single file component that includes the template, script, and style. This is where we will build out our job listing app.

By default, it is bringing in the `HelloWorld` and `TheWelcome` components. Those components are in the `src/components` folder.

Clean Up

Let's clean all of this up so that we have an clean slate to work with. Replace all of the code in the `App.vue` file with the following:

```
<template>
  <h1 class="text-2xl">Vue Jobs</h1>
</template>
```

I added a Tailwind class, but we have not setup Tailwind yet, so it will not do anything.

There is a folder called `components`. Let's delete everything in there. We will be creating our own components.

Tailwind CSS Setup

I am going to use Tailwind CSS for this project. If you are not familiar with Tailwind, it is a utility-first CSS framework that allows you to build custom designs without having to write any CSS. It's very popular and I use it in most of my projects.

To install Tailwind, run the following:

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

This will install Tailwind, PostCSS, and Autoprefixer as dev dependencies. PostCSS is a tool for transforming CSS with JavaScript plugins. Autoprefixer is a PostCSS plugin that parses your CSS and adds vendor prefixes to CSS rules.

Tailwind Config

Now we need to create a Tailwind config file. Run the following:

```
npx tailwindcss init -p
```

This will create a `tailwind.config.js` and a `postcss.config.js` file in the root of your project.

Add the following to your `tailwind.config.js` file:

```
module.exports = {
  content: ['./index.html', './src/**/*.{vue,js,ts,jsx,tsx}'],
  theme: {
    extend: {
      fontFamily: {
        sans: ['Poppins', 'sans-serif'],
      },
      gridTemplateColumns: {
        '70/30': '70% 28%',
      },
    },
  },
  variants: {
    extend: {},
  },
  plugins: [],
};
```

The content array, formerly known as `purge` specifies the files to scan for classes that are used in your project. Tailwind CSS, when used in production, purges unused styles to reduce the final CSS file size.

We are also adding some custom values to the theme object. We are adding a font family and a grid template column. This is a special class for the single job listing page.

Include Tailwind In CSS

Open the file `src/assets/main.css` and add the following:

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

This will add the base, components, and utilities classes to the CSS file from Tailwind.

Now, include that in your `src/main.js` file like this:

```
import '@/assets/main.css';
```

Restart the server and you should see that the Tailwind styles are being applied.

Vue Component Syntax

Before we start putting together our components, I just want to work in the `App.vue` file for a bit, so you can understand how components work. We already discussed the structure of a component in the slides. There is a template, which is the output, the style and the JavaScript logic.

You use the following tag syntax to separate these parts:

```
<script>
// Logic goes here
</script>

<template>
  <!-- HTML goes here -->
  <div>
    <h1 class="text-2xl">Vue Jobs</h1>
  </div>
</template>

<style>
/* Styles go here */
</style>
```

For styling, you would add the `scoped` attribute to the style tag. This will make the styles only apply to the component itself.

```
<style scoped>
h1 {
  color: red;
}
</style>
```

Now only the `h1` tag in this component will be red.

You could also use Sass by adding the `lang` attribute to the style tag:

```
<style lang="scss">
h1 {
  color: red;
}
</style>
```

However, for this to work, we need to install the `sass` package. I am not going to do that, but if you wanted to, you would run the following command:

```
npm install -D sass
```

Since, we are using Tailwind, we don't need to add much custom CSS to this app, so we can remove the style tags completely.

Add Data To A Component

Many components will have state or data associated with them. For example, a component that displays a list of jobs will have a list of jobs in its data. As I mentioned earlier, you can use the options API or the composition API to handle this. The composition API is the recommended way to handle data in Vue. It is more flexible and this is ultimately what we will be using in this project, however, I do want you to know the basics of the Options API because as a Vue developer, you will run into it. At least in the near future. Let's start with the Options API.

Let's start with something simple like a name. Add the following to the script tag in the `App.vue` file:

```
<script>
export default {
  data() {
    return {
      name: 'John Doe',
    };
  },
};
</script>
```

Interpolation

The Options API uses a function named `data` to return the data for the component. In this case, we are returning an object with a single property `name` set to 'John Doe'. Now we can use this data in the template. Replace the `h1` tag in the template with the following:

```
<template>
  <div class="flex flex-col items-center mt-10">
    <h1 class="text-2xl">{{ name }}</h1>
  </div>
</template>
```

You should now see the name in the heading. This is called interpolation and it is a way to use data from the component. We just wrap the variable in double curly braces.

Directives

Vue has something called directives. These are special attributes that you can add to your HTML elements to change the way the component renders. For example, you can use the `v-text` directive to change the text inside the `h1` tag.

Replace the `h1` tag with the following:

```
<h1 class="text-2xl" v-text="name"></h1>
```

What this is doing is updating the H1 tag's textContent property to the value of the `name` property. It will overwrite anything within the tag content. In this particular case, where you just want to show a value, you should use interpolation instead of the `v-text` directive. So let's change it back.

```
<h1 class="text-2xl">{{ name }}</h1>
```

Here are some other common directives:

- `v-if` - Render the element if the expression is true. There is also `v-else` and `v-else-if`.
- `v-for` - Iterate over an array of items and render them.
- `v-bind` - Bind an attribute to a property on the component.
- `v-on` - Bind an event to a function.
- `v-model` - Bind an input to a property on the component.
- `v-show` - Show or hide an element based on the expression.

We'll look at these as well as some others.

v-if

Let's look at an example of using the `v-if` directive. First we will add a new piece of data to the component. Add the following to the script tag:

```
<script>
export default {
  data() {
    return {
      name: 'John Doe',
      status: 'active',
    };
  },
};
</script>
```

Now we can use the `v-if` directive to conditionally render the status. Add the following under the name:

```
<h1 class="text-2xl">{{ name }}</h1>
<p v-if="status === 'active'" class="text-green-700">User is active</p>
```

It will show the name but if you change the `status` property to something else, it will not show the `p` tag.

v-else

You can also use `v-else` to show something else if the condition is false. Add the following to the template:

```
<h1 class="text-2xl">{{ name }}</h1>
<p v-if="status === 'active'" class="text-green-700">User is active</p>
<p v-else class="text-red-700">User is Inactive</p>
```

Now it will show that the user is inactive if the `status` is set to something else.

v-else-if

You can also use `v-else-if` to add another condition:

```
<h1 class="text-2xl">{{ name }}</h1>
<p v-if="status === 'active'" class="text-green-600">User is Active</p>
<p v-else-if="status === 'pending'" class="text-yellow-600">
    User is Pending
  </p>
<p v-else class="text-red-700">User is Inactive</p>
```

Now it will show active if active, pending if pending and inactive if anything else.

v-for

You can also use `v-for` to iterate over an array of items. Let's add an array of tasks to the data:

```
<script>
export default {
  data() {
    return {
      name: 'John Doe',
      status: 'active',
      tasks: ['Task One', 'Task Two', 'Task Three', 'Task Four'],
    };
  },
};
</script>
```

Now we can use `v-for` to loop over the tasks:

```
<h3 class="text-xl my-3">Tasks:</h3>
<ul>
  <li v-for="task in tasks" :key="task" class="flex items-center my-3">{{ task }}</li>
</ul>
```

The `v-for` directive takes the form of `v-for="item in items"`. The `:key` attribute is required when using `v-for` to help Vue keep track of the elements and their state. You should now see the tasks on the page.

v-bind

The `v-bind` directive is used to bind an attribute to a property on the component. For example, you can bind the `href` attribute of an anchor tag to a property on the component. Let's add a link to the data:

```
<script>
export default {
  data() {
    return {
      name: 'John Doe',
      status: 'active',
      tasks: ['Task One', 'Task Two', 'Task Three', 'Task Four'],
      link: 'https://google.com',
    };
  },
};
</script>
```

Now we can bind the `href` attribute of an anchor tag to the `link` property:

```
<a v-bind:href="link" class="text-blue-600">Link to Google</a>
```

We can also shorten this to `:href`:

```
<a :href="link" class="text-blue-600">Link to Google</a>
```

v-on & Methods

The `v-on` directive is used to bind an event to a function/method. For example, you can bind a click event to a function that changes the status. Let's do that.

First, add the button with the following:

```
<button
  v-on:click='toggleStatus'
  class='mt-3 px-4 py-2 bg-blue-500 text-white rounded-md'
>
  Change Status
</button>
```

This will fire off a click event when the button is clicked. Now let's add the method to the data. Remember, we are still using the Options API. I will convert it to the Composition API soon:

```
<script>
export default {
  data() {
    return {
```

```
      name: 'John Doe',
      status: 'active',
      tasks: ['Task One', 'Task Two', 'Task Three', 'Task Four'],
      link: 'https://google.com',
    };
  },
  methods: {
    toggleStatus() {
      if (this.status === 'active') {
        this.status = 'pending';
      } else if (this.status === 'pending') {
        this.status = 'inactive';
      } else {
        this.status = 'active';
      }
    },
  },
};
</script>
```

As you can see, we just have an object called `methods`. Inside that object, we have a method called `toggleStatus`. This method will change the status of the user. Now when the button is clicked, the status will change. We use the `this` keyword to access the data of the component.

v-on **Shorthand**

There is a shorter way to handle events. We can use the `@` symbol to bind an event to a method. For example, we can bind the `click` event to the `toggleStatus` method:

```
<button
  @click="toggleStatus"
  class="mt-3 px-4 py-2 bg-blue-500 text-white rounded-md"
>Change Status</button>
```

We will look at some other directives later, but these are the most common.

Here is the entire code so far:

```
<script>
export default {
  data() {
    return {
      name: 'John Doe',
      status: 'active',
      tasks: ['Task One', 'Task Two', 'Task Three', 'Task Four'],
      link: 'https://google.com',
    };
  },
  methods: {
    toggleStatus() {
      if (this.status === 'active') {
        this.status = 'pending';
      } else if (this.status === 'pending') {
        this.status = 'inactive';
      } else {
        this.status = 'active';
      }
    },
  },
};
</script>

<template>
  <div class="flex flex-col items-center mt-10">
    <h1 class="text-2xl">{{ name }}</h1>
    <p v-if="status === 'active'" class="text-green-600">User is Active</p>
    <p v-else-if="status === 'pending'" class="text-yellow-600">
      User is Pending
```

```
      </p>
      <p v-else class="text-red-700">User is Inactive</p>
      <button
        @click="toggleStatus"
        class="mt-3 px-4 py-2 bg-blue-500 text-white rounded-md"
      >
        Change Status
      </button>
      <h3 class="text-xl my-3">Tasks:</h3>
      <ul>
        <li v-for="task in tasks" :key="task" class="flex items-center my-3">
          {{ task }}
        </li>
      </ul>
      <a v-bind:href="link" class="text-blue-600">Link to Google</a>
    </div>
  </template>
```

Composition API

The Composition API is a new way to handle logic in Vue components. It is more flexible and allows you to create more complex components. It is also more similar to React's hooks. Let's convert the `App.vue` component to use the Composition API.

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

export default {
  setup() {
    const name = ref('John Doe');
    const status = ref('active');
    const tasks = ref(['Task One', 'Task Two', 'Task Three', 'Task Four']);
    const link = ref('https://google.com');

    const toggleStatus = () => {
      if (status.value === 'active') {
        status.value = 'pending';
      } else if (status.value === 'pending') {
        status.value = 'inactive';
      } else {
        status.value = 'active';
      }
    };

    return {
      name,
      status,
      tasks,
      link,
      toggleStatus,
    };
  },
};
</script>
```

With The Composition API, we use the `setup` function to define our reactive properties and methods. We use the `ref` function to create reactive references. What I mean by reactive is that we can change the value of the property and it will automatically update the UI. `ref` takes an inner value and returns a reactive and mutable ref object, which has a single property of .value that points to the inner value. We can directly mutate the value of the ref using the `.value` property. So if you're a React developer, this is similar to how `useState` works. When you modify a ref value, Vue automatically triggers reactivity updates in the template where that ref is used. Yes we could create a variable without ref and it would display initially, but it would not update when the value changes. We need to use ref to make it reactive.

I know at first glance, this seems much more difficult that what we just had. I thought the same thing and hated it. However, after using it in a few projects, I started to see the advantages and how much more flexible it is than just having a rigid object with data and methods.

What I just showed you is also the long form. We can really slim this down.

setup() Function Shorthand

First, we don't have to use the explicit setup function. Instead of actually wrapping everything in the `setup()` function, we can just add setup to the script like this:

```
<script setup>
// ...
</script>
```

Then we can take away the export and the returns and we are left with this:

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

const name = ref('John Doe');
const status = ref('active');
const tasks = ref(['Task One', 'Task Two', 'Task Three', 'Task Four']);
const link = ref('https://google.com');

const toggleStatus = () => {
  if (status.value === 'active') {
    status.value = 'pending';
  } else if (status.value === 'pending') {
    status.value = 'inactive';
  } else {
    status.value = 'active';
  }
};
</script>
```

You can get rid of the Google link bind and link property.

Add Task

Another thing I would like to do is to add a form to add a new task. We can do that by adding a form to the template.

```
<form @submit.prevent="addTask">
      <label for="newTask" class="block text-xl my-3">Add New Task:</label>
      <input
        type="text"
        id="newTask"
        v-model="newTask"
        class="border border-gray-300 rounded-md px-3 py-2 w-full"
      />
      <button
        type="submit"
        class="mt-3 px-4 py-2 bg-green-500 text-white rounded-md"
      >
        Add Task
      </button>
    </form>
```

v-model

The `v-model` directive is used to bind an input to a property on the component. In this case, we are binding the input to the `newTask` property. This will allow us to type in the input and have the value stored in the `newTask` property.

Let's add the property to the script:

```
const newTask = ref('');
```

Now we can add the `addTask` method to the script:

```
const addTask = () => {
  if (newTask.value.trim() !== '') {
    tasks.value.push(newTask.value);
    newTask.value = ''; // Clear input after adding task
  }
};
```

Now the task will get added and since the tasks are reactive, the list will update.

Delete Task

Now, let's add a button to delete the task. Add a button in the list item with the task:

```
<li
  v-for="(task, index) in tasks"
  :key="index"
  class="flex items-center my-3"
>
  <span>{{ task }}</span>
  <button
    @click="deleteTask(index)"
    class="ml-2 px-3 py-1 bg-red-500 text-white rounded-md text-xs"
  >
    x
  </button>
</li>
```

Then add the `deleteTask` method to the script:

```
const deleteTask = (index) => {
  tasks.value.splice(index, 1);
};
```

Vue.js Browser Devtools

I would suggest installing the Vue devtools for your browser. They are available for Chrome and Firefox. I'm not sure about other browsers because I haven't used another browser in over a decade. This will allow you to inspect your Vue components and see the data, props, and state of your components. It's a very useful tool.

You can download and install from the following urls:

- Chrome
- Firefox

Once you install, you should have an option for "Vue" in your devtools. From there, you can click on the components and see props, state, etc.

Lifecycle Hooks

In Vue, we have something called lifecycle hooks. These are functions that are called at different points in the lifecycle of a component. For example, when the component is created, mounted, updated, and destroyed. We can use these hooks to perform actions at these different points. There are a couple of them that were actually removed in Vue 3 because they were replaced by the setup() function, but the main ones are:

- `onBeforeMount` – called before mounting begins
- `onMounted` – called when component is mounted
- `onBeforeUpdate` – called when reactive data changes and before re-render

- `onUpdated` – called after re-render
- `onBeforeUnmount` – called before the Vue instance is destroyed
- `onUnmounted` – called after the instance is destroyed
- `onActivated` – called when a kept-alive component is activated
- `onDeactivated` – called when a kept-alive component is deactivated
- `onErrorCaptured` – called when an error is captured from a child component

These have to be imported to use. This is to keep everything lightweight.

We can use these for things like fetching data when the app/component loads. Let's use the `onMounted` hook to fetch some tasks from the JSONplaceholder API and then add them to the tasks property.

Add the following at the bottom of the script:

```
onMounted(async () => {
  try {
    const response = await fetch('https://jsonplaceholder.typicode.com/todos');
    const data = await response.json();
    tasks.value = data.map((task) => task.title);
  } catch (error) {
    console.error('Error fetching tasks:', error);
  }
});
```

We are fetching the array and then turning into an array of titles then adding it to the tasks property. Now you should see tasks from the API.

Jobs Project

Now that you know about the structure of a component and interpolation and directives, events, etc, let's start our main project. We can go ahead and clear all of this code out of the `App.vue` file. Im actually going to put it into a file called `App2.vue` just to keep it in the repo for you if you need to go back to it.

Theme Files

I wanted this project to look decent, so in the main repo, you will see a folder called `_theme_files`. This is where I put all of the theme files. These are just HTML files with Tailwind classes. I would suggest opening that folder up in another text editor window so it is easily accessible.

Let's start by copying everything in the `index.html` file into the `App.vue` file `<template>` tags.

You'll probably get an error about the logo image. Bring the `logo.png` file from the theme files into `src/assets/img` and add the following to the `App.vue` file:

```
import logo from '@/assets/img/logo.png';
```

You may think we can do something like this:

```
<img src={logo} alt='Logo' />
```

That is what we would do in React, but in Vue, we bind the logo to the image tag like this:

```
<img class='h-10 w-auto' v-bind:src='logo' alt='Vue Jobs' />
```

We can make it shorter with this:

```
<img class="h-10 w-auto" :src="logo" alt="Vue Jobs" />
```

You should now see the logo.

Your First UI Component

Now it's time to start breaking this up into components. Starting from the top, let's create a component for the navigation bar. Create a file at `src/components/Navbar.vue` and cut the `<nav>` and everything in it from the `App.vue` file and paste it into the `Navbar.vue` file in a `<template>` tag:

```
<template>
  <nav class="bg-green-700 border-b border-green-500">
    <div class="mx-auto max-w-7xl px-2 sm:px-6 lg:px-8">
      <div class="flex h-20 items-center justify-between">
        <div
          class="flex flex-1 items-center justify-center md:items-stretch md:justify-start"
        >
          <!-- Logo -->
          <a class="flex flex-shrink-0 items-center mr-4" href="index.html">
            <img class="h-10 w-auto" v-bind:src="logo" alt="Vue Jobs" />
            <span class="hidden md:block text-white text-2xl font-bold ml-2"
              >Vue Jobs</span
            >
          </a>
          <div class="md:ml-auto">
            <div class="flex space-x-2">
              <a
                href="index.html"
                class="text-white bg-green-900 hover:bg-gray-900 hover:text-white rounded-md px-3 py-2"
                >Home</a
              >
              <a
                href="jobs.html"
                class="text-white hover:bg-green-900 hover:text-white rounded-md px-3 py-2"
                >Jobs</a
              >
              <a
                href="add-job.html"
                class="text-white hover:bg-green-900 hover:text-white rounded-md px-3 py-2"
                >Add Job</a
              >
            </div>
          </div>
        </div>
      </div>
    </div>
  </nav>
</template>
```

We also need to now import the logo into this component instead of the `App.vue`. Add this to the top:

```
<script setup>
import logo from '@/assets/img/logo.png';
</script>
```

Now let's import the `Navbar.vue` component into the `App.vue` component:

```
<script setup>
import Navbar from '@/components/Navbar.vue';
</script>
```

Now embed the Navbar component:

```
<template>
  <Navbar />
</template>
```

Hero Component

Next, let's do the Hero component. Create a file at `src/components/Hero.vue` and cut the section with the hero and everything in it from the `App.vue` file and paste it into the `Hero.vue` file in a `<template>` tag:

```
<template>
  <section class="bg-green-700 py-20 mb-4">
    <div
      class="max-w-7xl mx-auto px-4 sm:px-6 lg:px-8 flex flex-col items-center"
    >
      <div class="text-center">
        <h1 class="text-4xl font-extrabold text-white sm:text-5xl md:text-6xl">
          Become a Vue Dev
        </h1>
        <p class="my-4 text-xl text-white">
          Find the Vue job that fits your skills and needs
        </p>
      </div>
    </div>
  </section>
</template>
```

Then import it into the main component and embed it:

```
<script setup>
import Navbar from '@/components/Navbar.vue';
import Hero from '@/components/Hero.vue';
</script>

<template>
  <Navbar />
  <Hero />
  <!-- //... -->
</template>
```

Props

Now let's talk about props. If you have ever used React or Angular, you know what props are, if you haven't, props are a way to pass data from a parent component to a child component. This is how you can make your components reusable. You can pass data, functions, or even other components as props.

I want the Hero component to be able to have a title and subtitle prop that can be passed in and it will display in the output.

In the `App.vue` file, let's pass in two props:

```
<template>
  <Hero title="Test Title" />
</template>
```

We are passing in a prop, which is just a string. Now we need to set up the props in the `Hero.vue` file. Add the following to the script tag:

```
<script setup>
import { defineProps } from 'vue';

defineProps({
```

```
    title: {
      type: String,
      default: 'Become a Vue dev',
    },
  });
  </script>
```

We are importing the `defineProps` function from the `vue` library. This function is used to define props. We are defining a prop called `title` with a type of `String` and a default value of `Become a Vue dev`. Now we can use this prop in the template:

```
<h1 class='text-4xl font-extrabold text-white sm:text-5xl md:text-6xl'>
  {{ title }}
</h1>
```

You should now see "Test Title" as the title.

Let's pass in the subtitle:

```
<template>
  <Hero title="Test Title" subtitle="Test Subtitle" />
</template>
```

Add the following to the `Hero.vue` file:

```
<script setup>
import { defineProps } from 'vue';

defineProps({
  title: {
    type: String,
    default: 'Become a Vue dev',
  },
  subtitle: {
    type: String,
    default: 'Find the Vue job that fits your skills and needs',
  },
});
</script>
```

You should see the subtitle in the output.

That's how we can pass data into other components. Now you can remove the title and subtitle from the embed:

```
<Hero />
```

You should see the default values.

HomeCards Component

Let's create a component at `src/components/HomeCards.vue` and cut the next section and paste into a template in that file:

```
<template>
  <section class="py-4">
    <div class="container-xl lg:container m-auto">
      <div class="grid grid-cols-1 md:grid-cols-2 gap-4 p-4 rounded-lg">
        <div class="bg-gray-100 p-6 rounded-lg shadow-md">
          <h2 class="text-2xl font-bold">For Developers</h2>
          <p class="mt-2 mb-4">
            Browse our Vue jobs and start your career today
          </p>
          <a
```

```
            href="jobs.html"
            class="inline-block bg-black text-white rounded-lg px-4 py-2 hover:bg-gray-700"
          >
            Browse Jobs
          </a>
        </div>
        <div class="bg-green-100 p-6 rounded-lg shadow-md">
          <h2 class="text-2xl font-bold">For Employers</h2>
          <p class="mt-2 mb-4">
            List your job to find the perfect developer for the role
          </p>
          <a
            href="add-job.html"
            class="inline-block bg-green-500 text-white rounded-lg px-4 py-2 hover:bg-green-600"
          >
            Add Job
          </a>
        </div>
      </div>
    </div>
  </section>
</template>
```

Import and embed into `App.vue`:

```
<script setup>
import Navbar from '@/components/Navbar.vue';
import Hero from '@/components/Hero.vue';
import HomeCards from '@/components/HomeCards.vue';
</script>

<template>
  <Navbar />
  <Hero />
  <HomeCards />
  <!-- //... -->
</template>
```

Wrapper Components & <slot>

We can also have components that wrap content. Let's create a component called `Card` that will wrap some text and add some styling to it to make it look like a card.

Create a file at `src/components/Card.vue` and add the following:

```
<template>
  <div class="bg-gray-100 p-6 rounded-lg shadow-md`">
    <slot></slot>
  </div>
</template>
```

The `<slot>` is a placeholder for the content that will be inserted into the component. If you are familiar with React, it is similar to the `{children}` prop. Now we can use this component to wrap the text in the `HomeCards.vue` component:

```
<script setup>
import Card from '@/components/Card.vue';
</script>
```

Now, replace the 2 divs that wrap the content. They have classes of `bg-gray-100` and `bg-green-100`. We can now use the `Card.vue` component to wrap the text:

```
<template>
  <section class="py-4">
```

```
      <div class="container-xl lg:container m-auto">
        <div class="grid grid-cols-1 md:grid-cols-2 gap-4 p-4 rounded-lg">
          <Card>
            <h2 class="text-2xl font-bold">For Developers</h2>
            <p class="mt-2 mb-4">
              Browse our Vue jobs and start your career today
            </p>
            <a
              href="jobs.html"
              class="inline-block bg-black text-white rounded-lg px-4 py-2 hover:bg-gray-700"
            >
              Browse Jobs
            </a>
          </Card>
          <Card>
            <h2 class="text-2xl font-bold">For Employers</h2>
            <p class="mt-2 mb-4">
              List your job to find the perfect developer for the role
            </p>
            <a
              href="add-job.html"
              class="inline-block bg-green-500 text-white rounded-lg px-4 py-2 hover:bg-green-600"
            >
              Add Job
            </a>
          </Card>
        </div>
      </div>
    </section>
  </template>
```

Now the text is wrapped in a card. However, I want the second card to have a green background. So let's make the card component take in a prop called `bg` for background. Add the following to the `Card.vue` file:

```
<script setup>
import { defineProps } from 'vue';

defineProps({
  bg: {
    type: String,
    default: 'bg-gray-100',
  },
});
</script>
```

Now in the template, use that prop:

```
<template>
  <div :class="`${bg} p-6 rounded-lg shadow-md`">
    <slot></slot>
  </div>
</template>
```

Since we are using a dynamic value in the class, we use the `:class`, which is short for `v-bind:class`.

Now, in the `HomeCards.vue` file, we can use the `bg` prop to change the background color for the second card:

```
<Card bg='bg-green-100'>
  <h2 class='text-2xl font-bold'>For Employers</h2>
  <p class='mt-2 mb-4'>
    List your job to find the perfect developer for the role
  </p>
  <a
    href='add-job.html'
    class='inline-block bg-green-500 text-white rounded-lg px-4 py-2 hover:bg-green-600'
  >
    Add Job
```

```
    </a>
  </Card>
```

Working With Data

Now we come to the 'Browse Jobs' section. in a real application, these jobs would come from some kind of database or API. We would make a request to the server and get the data back. For now, we are going to use a JSON file. Later we will use something called JSON Server to create a fake REST API. For now, we are going to create a file named `jobs.json` in the `src` folder and add the following code:

```
[
  {
    "id": 1,
    "title": "Senior Vue Developer",
    "type": "Full-Time",
    "description": "We are seeking a talented Front-End Developer to join our team in Boston, MA. The ideal candidate will have strong skills in HTML, CSS, and J
    "location": "Boston, MA",
    "salary": "$70K - $80K",
    "company": {
      "name": "NewTek Solutions",
      "description": "NewTek Solutions is a leading technology company specializing in web development and digital solutions. We pride ourselves on delivering hi
      "contactEmail": "contact@teksolutions.com",
      "contactPhone": "555-555-5555"
    }
  },
  {
    "id": 2,
    "title": "Front-End Engineer (Vue)",
    "type": "Full-Time",
    "description": "Join our team as a Front-End Developer in sunny Miami, FL. We are looking for a motivated individual with a passion for crafting beautiful an
    "location": "Miami, FL",
    "salary": "$70K - $80K",
    "company": {
      "name": "Veneer Solutions",
      "description": "Veneer Solutions is a creative agency specializing in digital design and development. Our team is dedicated to pushing the boundaries of cr
      "contactEmail": "contact@loremipsum.com",
      "contactPhone": "555-555-5555"
    }
  },
  {
    "id": 3,
    "title": "Vue.js Developer",
    "type": "Full-Time",
    "description": "Are you passionate about front-end development? Join our team in vibrant Brooklyn, NY, and work on exciting projects that make a difference.
    "location": "Brooklyn, NY",
    "salary": "$70K - $80K",
    "company": {
      "name": "Dolor Cloud",
      "description": "Dolor Cloud is a leading technology company specializing in digital solutions for businesses of all sizes. With a focus on innovation and c
      "contactEmail": "contact@dolorsitamet.com",
      "contactPhone": "555-555-5555"
    }
  },
  {
    "id": 4,
    "title": "Vue Front-End Developer",
    "type": "Part-Time",
    "description": "Join our team as a Part-Time Front-End Developer in beautiful Pheonix, AZ. We are looking for a self-motivated individual with a passion for
    "location": "Pheonix, AZ",
    "salary": "$60K - $70K",
    "company": {
      "name": "Alpha Elite",
      "description": "Alpha Elite is a dynamic startup specializing in digital marketing and web development. We are committed to fostering a diverse and inclusi
      "contactEmail": "contact@adipisicingelit.com",
      "contactPhone": "555-555-5555"
    }
  },
  {
    "id": 5,
    "title": "Full Stack Vue Developer",
```

```
    "type": "Full-Time",
    "description": "Exciting opportunity for a Full-Time Front-End Developer in bustling Atlanta, GA. We are seeking a talented individual with a passion for bui
    "location": "Atlanta, GA",
    "salary": "$90K - $100K",
    "company": {
      "name": "Browning Technologies",
      "description": "Browning Technologies is a rapidly growing technology company specializing in e-commerce solutions. We offer a dynamic and collaborative wo
      "contactEmail": "contact@consecteturadipisicing.com",
      "contactPhone": "555-555-5555"
    }
  },
  {
    "id": 6,
    "title": "Vue Native Developer",
    "type": "Full-Time",
    "description": "Join our team as a Front-End Developer in beautiful Portland, OR. We are looking for a skilled and enthusiastic individual to help us create
    "location": "Portland, OR",
    "salary": "$100K - $110K",
    "company": {
      "name": "Port Solutions INC",
      "description": "Port Solutions is a leading technology company specializing in software development and digital marketing. We are committed to providing ou
      "contactEmail": "contact@ipsumlorem.com",
      "contactPhone": "555-555-5555"
    }
  }
]
```

Now we can import the data into the `App` component and use it to render the job listings.

```
import jobData from './jobs.json';
```

Make Data Reactive

Now that we have the data, we need to make it reactive so that the component updates when the data changes. We can use the `ref` function to make the data reactive. Add the following to the script:

```
import { ref } from 'vue';
```

Now add the following to the script:

```
const jobs = ref(jobData);
```

Now the data will be reactive.

We need to loop over the jobs and output the data in the template. ultimately, we will have a component for each job item, but for now, we will just add the HTML here. We need to use the `v-for` directive to loop over the jobs and output the data. This is similar to using `jobs.map()` in React/JavaScript.

Replace the current section with this:

```
  <section class="bg-blue-50 px-4 py-10">
    <div class="container-xl lg:container m-auto">
      <h2 class="text-3xl font-bold text-green-500 mb-6 text-center">
        Browse Jobs
      </h2>
      <div class="grid grid-cols-1 md:grid-cols-3 gap-6">
        <!-- Iterate over jobs array -->
        <div
          v-for="job in jobs"
          :key="job.id"
          class="bg-white rounded-xl shadow-md relative"
        >
          <div class="p-4">
```

```
            <div class="mb-6">
              <div class="text-gray-600 my-2">Full-Time</div>
              <h3 class="text-xl font-bold">Senior Front-End Developer</h3>
            </div>

            <div class="mb-5">
              We are seeking a talented Front-End Developer to join our team in
              Boston, MA. The ideal candidate will have strong skills in HTML,
              CSS, and JavaScript...
            </div>

            <h3 class="text-green-500 mb-2">$70K - $80K / Year</h3>

            <div class="border border-gray-100 mb-5"></div>

            <div class="flex flex-col lg:flex-row justify-between mb-4">
              <div class="text-orange-700 mb-3">
                <i class="fa-solid fa-location-dot text-lg"></i>
                Boston, MA
              </div>
              <a
                href="job.html"
                class="h-[36px] bg-green-500 hover:bg-green-600 text-white px-4 py-2 rounded-lg text-center text-sm"
              >
                Read More
              </a>
            </div>
          </div>
        </div>
      </div>
    </div>
  </section>
```

You should see six jobs because that is how many are in the JSON file.

JobListing Component

Now, let's create a new file at `src/components/JobListing.vue` and add the following:

```
<script setup>
import { defineProps } from 'vue';

const props = defineProps({
  job: Object,
});
</script>

<template>
  <div class="bg-white rounded-xl shadow-md relative">
    <div class="p-4">
      <div class="mb-6">
        <div class="text-gray-600 my-2">{{ job.type }}</div>
        <h3 class="text-xl font-bold">{{ job.title }}</h3>
      </div>

      <div class="mb-5">{{ job.description }}</div>

      <h3 class="text-green-500 mb-2">{{ job.salary }} / Year</h3>

      <div class="border border-gray-100 mb-5"></div>

      <div class="flex flex-col lg:flex-row justify-between mb-4">
        <div class="text-orange-700 mb-3">
          <i class="fa-solid fa-location-dot text-lg"></i>
          {{ job.location }}
        </div>
        <a
          :href="'/job/' + job.id"
          class="h-[36px] bg-green-500 hover:bg-green-600 text-white px-4 py-2 rounded-lg text-center text-sm"
        >
          Read More
```

```
          </a>
        </div>
      </div>
    </div>
  </template>
```

We are specifying a prop called `job`, which will be an object. We are then using that prop in the template to output the job data.

Now, in the `App` component, we need to import the `JobListing` component and pass in each job as a prop. Replace the whole section with the following:

```
<section class="bg-green-50 px-4 py-10">
  <div class="container-xl lg:container m-auto">
    <h2 class="text-3xl font-bold text-green-500 mb-6 text-center">
      Browse Jobs
    </h2>
    <div class="grid grid-cols-1 md:grid-cols-3 gap-6">
      <!-- Iterate over jobs array -->
      <JobListing v-for="job in jobs" :key="job.id" :job="job" />
    </div>
  </div>
</section>
```

We are iterating over the `jobs` array and passing each job as a prop to the `JobListing` component. You should still see the job listings.

JobListings Component

Let's clean this up a bit and put this section and the section under it (View All Jobs link) into a component called `JobListings`. Create a file at `src/components/JobListings.vue` and cut and paste the two sections. We also need to move the imports for the jobs data and `JobListing` component and the `ref` stuff. Be sure to go up one level to the `jobs.json` file. It should look like this:

```
<script setup>
import JobListing from '@/components/JobListing.vue';
import jobData from '../jobs.json';

import { ref } from 'vue';

const jobs = ref(jobData);
</script>

<template>
  <section class="bg-green-50 px-4 py-10">
    <div class="container-xl lg:container m-auto">
      <h2 class="text-3xl font-bold text-green-500 mb-6 text-center">
        Browse Jobs
      </h2>
      <div class="grid grid-cols-1 md:grid-cols-3 gap-6">
        <!-- Iterate over jobs array -->
        <JobListing v-for="job in jobs" :key="job.id" :job="job" />
      </div>
    </div>
  </section>

  <section class="m-auto max-w-lg my-10 px-6">
    <a
      href="jobs.html"
      class="block bg-black text-white text-center py-4 px-6 rounded-xl hover:bg-gray-700"
      >View All Jobs</a
    >
  </section>
</template>
```

Now in the `App.vue`, you should have the following:

```
<script setup>
import Navbar from '@/components/Navbar.vue';
import Hero from '@/components/Hero.vue';
import HomeCards from '@/components/HomeCards.vue';
import JobListings from '@/components/JobListings.vue';
</script>

<template>
  <Navbar />
  <Hero />
  <HomeCards />
  <JobListings />
</template>
```

We have really cleaned up this file.

Limit Job Listings

On the homepage, we only want to show a limit of 3 job listings. Let's add a `limit` prop to the `JobListings` component and set it to `3`. Then, in the `App` component, pass in `limit` as a prop.

Change the `JobListings.vue` script to take a limit prop:

```
<script setup>
import JobListing from '@/components/JobListing.vue';
import jobData from '../jobs.json';

import { ref, defineProps } from 'vue';

const props = defineProps({
  limit: Number,
});

const jobs = ref(jobData);
</script>
```

Now, where we have the `v-for` and `JobListing`, change it to the following:

```
<JobListing
  v-for="(job, index) in jobs.slice(0, limit || jobs.length)"
  :key="job.id || index"
  :job="job"
/>
```

This will limit the jobs to the limit prop value if passed in. If not, then it will just show all the jobs.

Now, in the `App.vue` component, pass in `limit` as a prop:

```
<JobListings :limit="3" />
```

The reason, we add the `:` is because we are binding it and passing in 3 as a number. If we do not use it, then it will be a string.

Now you should only see three listings on the homepage.

showButton Prop

We are going to use this on the home and jobs page. We do not want the "View All Jobs" button on the jobs page. So let's add a prop called `showButton` and only show if that prop is passed in as true:

```
const props = defineProps({
  limit: Number,
  showButton: {
    type: Boolean,
    default: false,
  },
});
```

Now add the condition to the bottom section with the button:

```
<section v-if='showButton' class='m-auto max-w-lg my-10 px-6'>
  <RouterLink
    to='/jobs'
    class='block bg-black text-white text-center py-4 px-6 rounded-xl hover:bg-gray-700'
  >
    View All Jobs
  </RouterLink>
</section>
```

Now add the prop to the embed in the `HomeView`:

```
<JobListings :limit="3" :showButton="true" />
```

computed & Truncated Description

Let's talk about the `computed` function. This is a function that returns a value based on other values. It will run whenever that particular value changes. If you know React, it's kind of like the dependency array in the `useEffect()` hook. When that dependency changes the effect runs. Let's use it to show a truncated version of the description.

Add the following to the `JobListing.vue` component `<script>`:

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

const props = defineProps({
  job: Object,
});

const showFullDescription = ref(false);

const toggleDescription = () => {
  showFullDescription.value = !showFullDescription.value;
};

const truncatedDescription = computed(() => {
  let description = props.job.description;
  if (!showFullDescription.value) {
    description = description.substring(0, 90) + '...';
  }
  return description;
});
</script>
```

We first create a reactive variable called `showFullDescription` and set it to `false`. We then create a function called `toggleDescription` that will toggle the value of `showFullDescription`. We then create a computed property called `truncatedDescription` that will return the full description if `showFullDescription` is `true` and a truncated version if it is `false`. Now replace the following code:

```
<div class="mb-5">{{ job.description }}</div>
```

With this:

```
<div>{{ truncatedDescription }}</div>

<button
  @click="toggleDescription"
  class="text-green-500 hover:text-green-600 mb-5"
>
  {{ showFullDescription ? 'Less' : 'More' }}
</button>
```

Now you can toggle the truncated description.

Icons (PrimeIcons)

We are going to be using some icons in our project. There are many icon libraries. The one that we'll be using is called `PrimeIcons`. Let's install it:

```
npm install primeicons
```

Now replace the following:

```
<i className='fa-solid fa-location-dot text-lg'></i>
```

with the following:

```
<i class='pi pi-map-marker text-orange-700'></i>
```

You should now see the map marker icon. You can view all the available icons here - https://primevue.org/icons/

Vue Router

Up to this point, we have just been using one page/url with all of our components. In a real application, we would have multiple pages and we would need to navigate between them. This is where Vue Router comes in. Vue Router is the official router for Vue.js. We need the router because we will have a separate page for jobs and single job listings as well as the page to add and update listings. You could have chosen to implement the router at the very beginning, but I wanted to show you how to do it yourself.

Let's install Vue Router:

```
npm install vue-router
```

Now we need to set up the router. Create a file at `src/router/index.js` and add the following:

```
import { createRouter, createWebHistory } from 'vue-router';
import HomeView from '@/views/HomeView.vue';

const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),
  routes: [
    {
      path: '/',
      name: 'home',
      component: HomeView,
    },
  ],
});

export default router;
```

We are importing `createRouter` and `createWebHistory` from `vue-router`. We are then creating a router instance with the `createRouter` function. We are passing in the history mode and the routes. We are setting the base URL to `import.meta.env.BASE_URL`, which is the base URL of the app. We are then exporting the router instance. We have one route for the home url to load the home view, which we have not yet created.

Now let's open up the `src/main.js` file and apply the router:

```
import { createApp } from 'vue';
import App from './App.vue';
import '@/assets/main.css';
import 'primeicons/primeicons.css';
import router from './router';

const app = createApp(App);

app.use(router);

app.mount('#app');
```

We have imported the router and then we are using it in the `createApp` function. Now we can use the router in our components.

Views

We are going to create a `views` folder in the `src` folder. This is where we will put the components that will be used as pages. Create a file at `src/views/HomeView.vue` and move everything except the Navbar component from the `App.vue` component into it. The Navbar we want on every page, so that will be stay in the `App.vue` component. The `HomeView.vue` file should look like this:

```
<script setup>
import Hero from '@/components/Hero.vue';
import HomeCards from '@/components/HomeCards.vue';
import JobListings from '@/components/JobListings.vue';
</script>

<template>
  <Hero />
  <HomeCards />
  <JobListings :limit="3" />
</template>
```

In your `App.vue' file, you should now have:

```
<script setup>
import { RouterView } from 'vue-router';
import Navbar from '@/components/Navbar.vue';
</script>

<template>
  <Navbar />
  <RouterView />
</template>
```

The `RouterView` component is a special component that will render the component that is currently active. Anything else in this output will be on every page such as the navbar.

If you run into any weird errors, try deleting the `node_modules` folder and reinstalling the dependencies with `npm install`.

Now you should see everything as it was before.

Jobs View

Let's create a new page and route for the jobs page. This will simply list out all jobs. We can use the `JobListings` component for this.

Create a file at `src/views/JobsView.vue` and add the following:

```
<script setup>
import JobListings from '@/components/JobListings.vue';
</script>

<template>
  <JobListings />
</template>
```

Jobs Route

Now we need to add a route for this page in the `router/index.js` file:

```
import { createRouter, createWebHistory } from 'vue-router';
import HomeView from '@/views/HomeView.vue';
import JobsView from '@/views/JobsView.vue';

const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),
  routes: [
    {
      path: '/',
      name: 'home',
      component: HomeView,
    },
    {
      path: '/jobs',
      name: 'jobs',
      component: JobsView,
    },
  ],
});

export default router;
```

Now if you go to http://localhost:3000/jobs you should see the jobs page.

Links

Now we need to add links to navigate between the pages. We can use the `router-link` component to create links. Let's go into the `Navbar.vue` component and import the `router-link` component:

```
<script setup>
import { RouterLink } from 'vue-router';
</script>
```

Now, you want to replace any `a` tags with the `router-link` component. Let's start with the logo. It should look like this:

```
<RouterLink class='flex flex-shrink-0 items-center mr-4' to='/'>
  <img class='h-10 w-auto' v-bind:src='logo' alt='Vue Jobs' />
  <span class='hidden md:block text-white text-2xl font-bold ml-2'>
    Vue Jobs
  </span>
</RouterLink>
```

Be sure to change `href` to `to` and change `index.html` to `/`.

Now do the navigation links:

```
<div class='md:ml-auto'>
  <div class='flex space-x-2'>
    <RouterLink
      to='/'
      class='text-white bg-green-900 hover:bg-gray-900 hover:text-white rounded-md px-3 py-2'
    >
      Home
    </RouterLink>
    <RouterLink
      to='/jobs'
      class='text-white hover:bg-green-900 hover:text-white rounded-md px-3 py-2'
    >
      Jobs
    </RouterLink>
    <RouterLink
      to='/jobs/add'
      class='text-white hover:bg-green-900 hover:text-white rounded-md px-3 py-2'
    >
      Add Job
    </RouterLink>
  </div>
</div>
```

Active Links

The routing should work, however, the active link is always the home link. We need to add some logic to make sure the active link is the current page.

Let's bring in `useRoute` from the `vue-router` package:

```
import { RouterLink, useRoute } from 'vue-router';
```

Now add the following function:

```
// Function to determine if the link is active
const isActiveLink = (routePath) => {
  const route = useRoute();
  return route.path === routePath;
};
```

We are simply checking if the current route path is equal to the route path we pass in. Now we can use this function to determine if the link is active:

```
 <div class="flex space-x-2">
  <RouterLink
    to="/"
    :class="[
      'text-white',
      isActiveLink('/')
        ? 'bg-green-900'
        : 'hover:bg-gray-900 hover:text-white',
      'rounded-md',
      'px-3',
      'py-2',
    ]"
    >Home</RouterLink
  >
  <RouterLink
    to="/jobs"
    :class="[
      'text-white',
      isActiveLink('/jobs')
        ? 'bg-green-900'
        : 'hover:bg-green-900 hover:text-white',
      'rounded-md',
```

```
        'px-3',
        'py-2',
      ]"
      >Jobs</RouterLink
    >
    <RouterLink
      to="/jobs/add"
      :class="[
        'text-white',
        isActiveLink('/jobs/add')
          ? 'bg-green-900'
          : 'hover:bg-green-900 hover:text-white',
        'rounded-md',
        'px-3',
        'py-2',
      ]"
      >Add Job</RouterLink
    >
  </div>
```

Now the correct link should be active.

View All Jobs Link

Let's add the correct link to the View All Jobs link. In the `JobListings` component, import `RouterLink`:

```
import { RouterLink } from 'vue-router';
```

Then replace the section with the link/button with this:

```
<section class='m-auto max-w-lg my-10 px-6'>
  <RouterLink
    to='/jobs'
    class='block bg-black text-white text-center py-4 px-6 rounded-xl hover:bg-gray-700'
  >
    View All Jobs
  </RouterLink>
</section>
```

We should also edit the link for the "For Developers" section. Open the `src/components/HomeCards.vue` file. Let's change both links to the correct routes:

```
<script setup>
import Card from '@/components/Card.vue';
import { RouterLink } from 'vue-router';
</script>

<template>
  <section class="py-4">
    <div class="container-xl lg:container m-auto">
      <div class="grid grid-cols-1 md:grid-cols-2 gap-4 p-4 rounded-lg">
        <Card>
          <h2 class="text-2xl font-bold">For Developers</h2>
          <p class="mt-2 mb-4">
            Browse our Vue jobs and start your career today
          </p>
          <RouterLink
            to="/jobs"
            class="inline-block bg-black text-white rounded-lg px-4 py-2 hover:bg-gray-700"
          >
            Browse Jobs
          </RouterLink>
        </Card>
        <Card bg="bg-green-100">
          <h2 class="text-2xl font-bold">For Employers</h2>
          <p class="mt-2 mb-4">
```

```
            List your job to find the perfect developer for the role
          </p>
          <RouterLink
            to="/jobs/add"
            class="inline-block bg-green-500 text-white rounded-lg px-4 py-2 hover:bg-green-600"
          >
            Add Job
          </RouterLink>
        </Card>
      </div>
    </div>
  </section>
</template>
```

Not Found Page

Right now, if you visit a route that doesn't exist, you will see a blank page. Let's create a simple not found page. Create a file at `src/views/NotFoundView.vue` and add the following:

```
<script setup>
import { RouterLink } from 'vue-router';
</script>

<template>
  <section class="text-center flex flex-col justify-center items-center h-96">
    <i class="pi pi-exclamation-triangle text-yellow-500 text-7xl mb-5"></i>
    <h1 class="text-6xl font-bold mb-4">404 Not Found</h1>
    <p class="text-xl mb-5">This page does not exist</p>
    <RouterLink
      to="/"
      class="text-white bg-green-700 hover:bg-green-900 rounded-md px-3 py-2 mt-4"
    >
      Go Back
    </RouterLink>
  </section>
</template>
```

Now we need to add a route for this page in the `router/index.js` file:

```
import NotFoundView from '@/views/NotFoundView.vue';

//...

{
  path: '/:catchAll(.*)', // Match any path that hasn't been matched by other routes
  name: 'not-found',
  component: NotFoundView,
},
```

Single Job View

Let's create a single job view. Create a file at `src/views/JobView.vue`. We can copy the section from the theme_files `job.html` file and use it in the template:

```
<template>
  <section class="bg-green-50">
    <div class="container m-auto py-10 px-6">
      <div class="grid grid-cols-1 md:grid-cols-70/30 w-full gap-6">
        <main>
          <div
            class="bg-white p-6 rounded-lg shadow-md text-center md:text-left"
          >
            <div class="text-gray-500 mb-4">Full-Time</div>
            <h1 class="text-3xl font-bold mb-4">Senior Vue Developer</h1>
```

```
            <div
              class="text-gray-500 mb-4 flex align-middle justify-center md:justify-start"
            >
              <i
                class="fa-solid fa-location-dot text-lg text-orange-700 mr-2"
              ></i>
              <p class="text-orange-700">Boston, MA</p>
            </div>
          </div>

          <div class="bg-white p-6 rounded-lg shadow-md mt-6">
            <h3 class="text-green-800 text-lg font-bold mb-6">
              Job Description
            </h3>

            <p class="mb-4">
              We are seeking a talented Front-End Developer to join our team in
              Boston, MA. The ideal candidate will have strong skills in HTML,
              CSS, and JavaScript, with experience working with modern
              JavaScript frameworks such as Vue or Angular.
            </p>

            <h3 class="text-green-800 text-lg font-bold mb-2">Salary</h3>

            <p class="mb-4">$70k - $80K / Year</p>
          </div>
        </main>

        <!-- Sidebar -->
        <aside>
          <!-- Company Info -->
          <div class="bg-white p-6 rounded-lg shadow-md">
            <h3 class="text-xl font-bold mb-6">Company Info</h3>

            <h2 class="text-2xl">NewTek Solutions</h2>

            <p class="my-2">
              NewTek Solutions is a leading technology company specializing in
              web development and digital solutions. We pride ourselves on
              delivering high-quality products and services to our clients while
              fostering a collaborative and innovative work environment.
            </p>

            <hr class="my-4" />

            <h3 class="text-xl">Contact Email:</h3>

            <p class="my-2 bg-green-100 p-2 font-bold">
              contact@newteksolutions.com
            </p>

            <h3 class="text-xl">Contact Phone:</h3>

            <p class="my-2 bg-green-100 p-2 font-bold">555-555-5555</p>
          </div>

          <!-- Manage -->
          <div class="bg-white p-6 rounded-lg shadow-md mt-6">
            <h3 class="text-xl font-bold mb-6">Manage Job</h3>
            <a
              href="add-job.html"
              class="bg-green-500 hover:bg-green-600 text-white text-center font-bold py-2 px-4 rounded-full w-full focus:outline-none focus:shadow-outline mt-4
              >Edit Job</a
            >
            <button
              class="bg-red-500 hover:bg-red-600 text-white font-bold py-2 px-4 rounded-full w-full focus:outline-none focus:shadow-outline mt-4 block"
            >
              Delete Job
            </button>
          </div>
        </aside>
      </div>
    </div>
```

```
    </section>
  </template>
```

Now create the route in the `router/index.js` file:

```
import JobView from '@/views/JobView.vue';

//...

{
  path: '/jobs/:id',
  name: 'job',
  component: JobView,
},
```

If you put `http://localhost:8080/jobs/1` in the browser, you should see the job view.

Let's update the link in the `src/components/JobListing.vue` component:

```
import { RouterLink } from 'vue-router';

//..
  <RouterLink
  :to="'/jobs/' + job.id"
  class="h-[36px] bg-green-500 hover:bg-green-600 text-white px-4 py-2 rounded-lg text-center text-sm"
>
  Read More
</RouterLink>
```

We brought in the router link and added it to the button. Since we used a dynamic route with the job id, we need to use `:to`.

Obviously, right now, the data is just hardcoded HTML. We'll leave it like that for now until we create our backend server and we can fetch the data from there. Let's do that now with JSON Server.

JSON Server & Getting Data

Right now, our data comes from a JSON file. We want to make this work like a real application, so we will be using JSON Server to create a fake REST API. JSON Server is a full fake REST API with zero coding.

First, let's install JSON Server:

```
npm install json-server
```

Now we can use the `jobs.json` file that we have been using. However we need to name the array `jobs`. So open the file and add enclosing curly braces around everything and add a `jobs` name for the array like this:

```
{
  "jobs": [
    {
      "id": 1,
      "title": "Senior Vue Developer",
      "type": "Full-Time",
      "description": "We are seeking a talented Front-End Developer to join our team in Boston, MA. The ideal candidate will have strong skills in HTML, CSS, and
      "location": "Boston, MA",
      "salary": "$70K - $80K",
      "company": {
        "name": "NewTek Solutions",
        "description": "NewTek Solutions is a leading technology company specializing in web development and digital solutions. We pride ourselves on delivering
        "contactEmail": "contact@teksolutions.com",
        "contactPhone": "555-555-5555"
      }
    },
    {
```

```
    "id": 2,
    "title": "Front-End Engineer (Vue)",
    "type": "Full-Time",
    "description": "Join our team as a Front-End Developer in sunny Miami, FL. We are looking for a motivated individual with a passion for crafting beautiful
    "location": "Miami, FL",
    "salary": "$70K - $80K",
    "company": {
      "name": "Veneer Solutions",
      "description": "Veneer Solutions is a creative agency specializing in digital design and development. Our team is dedicated to pushing the boundaries of
      "contactEmail": "contact@loremipsum.com",
      "contactPhone": "555-555-5555"
    }
  },
  {
    "id": 3,
    "title": "Vue.js Developer",
    "type": "Full-Time",
    "description": "Are you passionate about front-end development? Join our team in vibrant Brooklyn, NY, and work on exciting projects that make a difference
    "location": "Brooklyn, NY",
    "salary": "$70K - $80K",
    "company": {
      "name": "Dolor Cloud",
      "description": "Dolor Cloud is a leading technology company specializing in digital solutions for businesses of all sizes. With a focus on innovation and
      "contactEmail": "contact@dolorsitamet.com",
      "contactPhone": "555-555-5555"
    }
  },
  {
    "id": 4,
    "title": "Vue Front-End Developer",
    "type": "Part-Time",
    "description": "Join our team as a Part-Time Front-End Developer in beautiful Pheonix, AZ. We are looking for a self-motivated individual with a passion fo
    "location": "Pheonix, AZ",
    "salary": "$60K - $70K",
    "company": {
      "name": "Alpha Elite",
      "description": "Alpha Elite is a dynamic startup specializing in digital marketing and web development. We are committed to fostering a diverse and inclu
      "contactEmail": "contact@adipisicingelit.com",
      "contactPhone": "555-555-5555"
    }
  },
  {
    "id": 5,
    "title": "Full Stack Vue Developer",
    "type": "Full-Time",
    "description": "Exciting opportunity for a Full-Time Front-End Developer in bustling Atlanta, GA. We are seeking a talented individual with a passion for b
    "location": "Atlanta, GA",
    "salary": "$90K - $100K",
    "company": {
      "name": "Browning Technologies",
      "description": "Browning Technologies is a rapidly growing technology company specializing in e-commerce solutions. We offer a dynamic and collaborative
      "contactEmail": "contact@consecteturadipisicing.com",
      "contactPhone": "555-555-5555"
    }
  },
  {
    "id": 6,
    "title": "Vue Native Developer",
    "type": "Full-Time",
    "description": "Join our team as a Front-End Developer in beautiful Portland, OR. We are looking for a skilled and enthusiastic individual to help us creat
    "location": "Portland, OR",
    "salary": "$100K - $110K",
    "company": {
      "name": "Port Solutions INC",
      "description": "Port Solutions is a leading technology company specializing in software development and digital marketing. We are committed to providing
      "contactEmail": "contact@ipsumlorem.com",
      "contactPhone": "555-555-5555"
    }
  }
]
}
```


The job listings will now break, but that's ok.

JSON Server Script

We need to create an NPM script to run the JSON server. Open the `package.json` file and add the following under scripts:

```
"scripts": {
  "dev": "vite",
  "build": "vite build",
  "preview": "vite preview",
  "server": "json-server --watch src/jobs.json --port 5000" // Add this
},
```

Now, open a new terminal and run `npm run server`.

If you open `http://localhost:5000/jobs` you will see the data.

Fetch Data

Now we need to fetch the data from the JSON server. I will be using the `Axios` library, but you can just as well use the Fetch API to do this. We will be using the `onMounted` lifecycle hook to fetch the data when the component is mounted.

First, let's install Axios:

```
npm install axios
```

Your script should look like this:

```
<script setup>
import JobListing from '@/components/JobListing.vue';
import { RouterLink } from 'vue-router';
import axios from 'axios';

import { ref, defineProps, onMounted } from 'vue';

const props = defineProps({
  limit: Number,
});

const jobs = ref([]);

onMounted(async () => {
  try {
    const response = await axios.get('http://localhost:5000/jobs');
    jobs.value = response.data;
  } catch (error) {
    console.error('Error fetching jobs:', error);
  }
});
</script>
```

So on mount, we are fetching the data from the JSON server and setting it to the `jobs` ref. If there is an error, we are logging it to the console.

Now you should see the job listings again but now they are coming from the server. This is more realistic than just storing them in a file.

Using reactive

How we have this should be fine, but I want to show you a different way by using `reactive` instead of `ref`. If I want to have an object with different values for our state in a single variable, like let's say we want the jobs but we also want a loading value that we can set to true when we are fetching the data and then set it to false when we are done, we can use `reactive` instead of `ref`.

ref vs reactive

Here are some of the differences:

- `reactive()` only takes objects. It does not take primitives like strings, numbers and booleans.
- `ref()` can take objects or primitives. It uses `ref()` under the hood.
- `ref()` has a `.value`property for reassigning, `reactive()` doesn't use `.value` and can't be reassigned

If you are coming from React, the way that I look at it is in React, if you would use a `useState` for something like a string like name, then I would use `ref` in Vue. If you would use an object in `useState` in React, then I would use `reactive` in Vue. That is in no way a set in stone convention, that's just how I personally think about it.

Let's change to the following:

```
<script setup>
import JobListing from '@/components/JobListing.vue';
import { RouterLink } from 'vue-router';
import axios from 'axios';

import { reactive, defineProps, onMounted } from 'vue';

const props = defineProps({
  limit: Number,
});

const state = reactive({
  jobs: [],
  isLoading: true,
});

onMounted(async () => {
  try {
    const response = await axios.get('http://localhost:5000/jobs');
    state.jobs = response.data;
  } catch (error) {
    console.error('Error fetching jobs:', error);
  } finally {
    state.isLoading = false;
  }
});
</script>
```

Now we have a `state` object that contains the `jobs` and `isLoading` values. We can set the `isLoading` value to `true` when we are fetching the data and then set it to `false` when we are done.

Let's install a very simple library to show a spinner. Run the following:

```
npm install vue-spinner
```

Import it at the top:

```
import PulseLoader from 'vue-spinner/src/PulseLoader.vue';
```

We can then change the template to the following:

```
<template>
  <section class="bg-blue-50 px-4 py-10">
    <div class="container-xl lg:container m-auto">
      <h2 class="text-3xl font-bold text-green-500 mb-6 text-center">
        Browse Jobs
      </h2>
      <!-- Show loading spinner or placeholder while isLoading is true -->
      <div v-if="state.isLoading" class="text-center text-gray-500 py-6">
        <PulseLoader />
```

```
      </div>

      <!-- Show job listings when isLoading is false -->
      <div v-else class="grid grid-cols-1 md:grid-cols-3 gap-6">
        <JobListing
          v-for="(job, index) in state.jobs.slice(
            0,
            limit || state.jobs.length
          )"
          :key="job.id || index"
          :job="job"
        />
      </div>
    </div>
  </section>

  <section class="m-auto max-w-lg my-10 px-6">
    <RouterLink
      to="/jobs"
      class="block bg-black text-white text-center py-4 px-6 rounded-xl hover:bg-gray-700"
      >View All Jobs</RouterLink
    >
  </section>
</template>
```

Now it will show `Loading` while we are fetching the data and then it will show the job listings. You don't have to do this, it's just another option.

If you want to slow it down to see the spinner in action, you can use `setTimeout`:

```
onMounted(async () => {
  try {
    const response = await axios.get('http://localhost:5000/jobs');
    // Simulate a 2-second delay
    await new Promise((resolve) => setTimeout(resolve, 2000));
    state.jobs = response.data;
  } catch (error) {
    console.error('Error fetching jobs:', error);
  } finally {
    // Set isLoading to false after the delay
    state.isLoading = false;
  }
});
```

Data For Single Job

The single job view just has hardcoded HTML. Let's fetch the job from our server and display it on the page.

Add the following script to fetch the data and put it into state:

```
<script setup>
import PulseLoader from 'vue-spinner/src/PulseLoader.vue';
import axios from 'axios';

import { reactive, onMounted } from 'vue';
import { useRoute, RouterLink } from 'vue-router';

const route = useRoute();

const jobId = route.params.id;

const state = reactive({
  job: {},
  isLoading: true,
});

onMounted(async () => {
  try {
    const response = await axios.get(`http://localhost:5000/jobs/${jobId}`);
```

```
    state.job = response.data;
  } catch (error) {
    console.error('Error fetching job:', error);
  } finally {
    state.isLoading = false;
  }
});
</script>
```

We can get the id from the URL using the `useRoute` hook. We can then use the `jobId` to fetch the job from the server and put it into the `state` object.

You can open the Vue devtools and check to see if the data is in the state.

For the template, add the following:

```
<template>
  <section v-if="!state.isLoading" class="bg-green-50">
    <div class="container m-auto py-10 px-6">
      <div class="grid grid-cols-1 md:grid-cols-70/30 w-full gap-6">
        <main>
          <div
            class="bg-white p-6 rounded-lg shadow-md text-center md:text-left"
          >
            <div class="text-gray-500 mb-4">{{ state.job.type }}</div>
            <h1 class="text-3xl font-bold mb-4">{{ state.job.title }}</h1>
            <div
              class="text-gray-500 mb-4 flex align-middle justify-center md:justify-start"
            >
              <i class="pi pi-map-marker text-orange-700 mr-2 text-xl"></i>
              <p class="text-orange-700 font-bold">{{ state.job.location }}</p>
            </div>
          </div>

          <div class="bg-white p-6 rounded-lg shadow-md mt-6">
            <h3 class="text-green-800 text-lg font-bold mb-6">
              Job Description
            </h3>

            <p class="mb-4">
              {{ state.job.description }}
            </p>

            <h3 class="text-green-800 text-lg font-bold mb-2">Salary</h3>

            <p class="mb-4">{{ state.job.salary }} / Year</p>
          </div>
        </main>

        <!-- Sidebar -->
        <aside>
          <!-- Company Info -->
          <div class="bg-white p-6 rounded-lg shadow-md">
            <h3 class="text-xl font-bold mb-6">Company Info</h3>

            <h2 class="text-2xl">NewTek Solutions</h2>

            <p class="my-2">
              {{ state.job.company.description }}
            </p>

            <hr class="my-4" />

            <h3 class="text-xl">Contact Email:</h3>

            <p class="my-2 bg-green-100 p-2 font-bold">
              {{ state.job.company.contactEmail }}
            </p>

            <h3 class="text-xl">Contact Phone:</h3>

            <p class="my-2 bg-green-100 p-2 font-bold">
              {{ state.job.company.contactPhone }}
```

```
            </p>
          </div>

          <!-- Manage -->
          <div class="bg-white p-6 rounded-lg shadow-md mt-6">
            <h3 class="text-xl font-bold mb-6">Manage Job</h3>
            <RouterLink
              :to="'/jobs/edit/' + state.job.id"
              class="bg-green-500 hover:bg-green-600 text-white text-center font-bold py-2 px-4 rounded-full w-full focus:outline-none focus:shadow-outline mt-4
              >Edit Job</RouterLink
            >
            <button
              class="bg-red-500 hover:bg-red-600 text-white font-bold py-2 px-4 rounded-full w-full focus:outline-none focus:shadow-outline mt-4 block"
            >
              Delete Job
            </button>
          </div>
        </aside>
      </div>
    </div>
  </section>

  <!-- Loading State -->
  <div v-else class="text-center text-gray-500 py-6">
    <PulseLoader />
  </div>
</template>
```

Now you should see the job data on the single job view page.

Back Button

Let's create a back button component to easily navigate back to the jobs page.

Create a new file called `BackButton.vue` in the `components` folder. Add the following:

```
<script setup>
import { RouterLink } from 'vue-router';
</script>

<template>
  <section>
    <div class="container m-auto py-6 px-6">
      <RouterLink
        to="/jobs"
        class="text-green-500 hover:text-green-600 flex items-center"
      >
        <i class="fas fa-arrow-left mr-2"></i> Back to Job Listings
      </RouterLink>
    </div>
  </section>
</template>
```

Now bring it into the `JobView.vue` component:

```
import BackButton from '@/components/BackButton.vue';
```

Embed it above the section in the template:

```
<template>
  <BackButton />
  <section v-if="!state.isLoading" class="bg-green-50">
    // ...
</template>
```

Proxying Requests

Right now, when we make a request to fetch data, we are making a request to `http://localhost:5000/jobs`. This is because we are running the server on `http://localhost:5000`. However, when we deploy the app, we won't be able to make requests to `http://localhost:5000`. We need to make requests to the same domain that the app is hosted on. We can use a proxy to do this.

Open the `vue.config.js` in the root of the project and add the `proxy` object with the following settings:

```
export default defineConfig({
  plugins: [vue()],
  resolve: {
    alias: {
      '@': fileURLToPath(new URL('./src', import.meta.url)),
    },
  },
  server: {
    port: 3000,
    proxy: {
      '/api': {
        target: 'http://localhost:5000',
        changeOrigin: true,
        rewrite: (path) => path.replace(/^\/api/, ''),
      },
    },
  },
});
```

What this is doing is changing the URL from `http://localhost:5000/jobs` to `http://localhost:3000/api/jobs`. or whatever our domain is.

Now we can change the URL in the `axios` request to `/api/jobs`. Do this in the `src/components/JobListings.vue` file as well as the `src/views/JobView.vue` file.

Add Job Page

Now let's create a page to add a new job. Create a file at `src/views/AddJobView.vue` and add the following for now:

```
<template>This is the add page</template>
```

Add it to the `router/index.js` file:

```
import AddJobView from '@/views/AddJobView.vue';

//...

 {
    path: '/jobs/add',
    name: 'add-job',
    component: AddJobView,
  },
```

Now you should see the add job page at `http://localhost:3000/jobs/add`.

Copy the html from the theme_files `add-job.html` page and paste it into the template:

```
<template>
  <section class="bg-green-50">
    <div class="container m-auto max-w-2xl py-24">
      <div
        class="bg-white px-6 py-8 mb-4 shadow-md rounded-md border m-4 md:m-0"
      >
        <form>
          <h2 class="text-3xl text-center font-semibold mb-6">Add Job</h2>
```

```
<div class="mb-4">
  <label for="type" class="block text-gray-700 font-bold mb-2"
    >Job Type</label
  >
  <select
    id="type"
    name="type"
    class="border rounded w-full py-2 px-3"
    required
  >
    <option value="Full-Time">Full-Time</option>
    <option value="Part-Time">Part-Time</option>
    <option value="Remote">Remote</option>
    <option value="Internship">Internship</option>
  </select>
</div>

<div class="mb-4">
  <label class="block text-gray-700 font-bold mb-2"
    >Job Listing Name</label
  >
  <input
    type="text"
    id="name"
    name="name"
    class="border rounded w-full py-2 px-3 mb-2"
    placeholder="eg. Beautiful Apartment In Miami"
    required
  />
</div>
<div class="mb-4">
  <label for="description" class="block text-gray-700 font-bold mb-2"
    >Description</label
  >
  <textarea
    id="description"
    name="description"
    class="border rounded w-full py-2 px-3"
    rows="4"
    placeholder="Add any job duties, expectations, requirements, etc"
  ></textarea>
</div>

<div class="mb-4">
  <label for="type" class="block text-gray-700 font-bold mb-2"
    >Salary</label
  >
  <select
    id="salary"
    name="salary"
    class="border rounded w-full py-2 px-3"
    required
  >
    <option value="Under $50K">under $50K</option>
    <option value="$50K - $60K">$50 - $60K</option>
    <option value="$60K - $70K">$60 - $70K</option>
    <option value="$70K - $80K">$70 - $80K</option>
    <option value="$80K - $90K">$80 - $90K</option>
    <option value="$90K - $100K">$90 - $100K</option>
    <option value="$100K - $125K">$100 - $125K</option>
    <option value="$125K - $150K">$125 - $150K</option>
    <option value="$150K - $175K">$150 - $175K</option>
    <option value="$175K - $200K">$175 - $200K</option>
    <option value="Over $200K">Over $200K</option>
  </select>
</div>

<div class="mb-4">
  <label class="block text-gray-700 font-bold mb-2"> Location </label>
  <input
    type="text"
    id="location"
    name="location"
    class="border rounded w-full py-2 px-3 mb-2"
```

```
          placeholder="Company Location"
          required
        />
      </div>

      <h3 class="text-2xl mb-5">Company Info</h3>

      <div class="mb-4">
        <label for="company" class="block text-gray-700 font-bold mb-2"
          >Company Name</label
        >
        <input
          type="text"
          id="company"
          name="company"
          class="border rounded w-full py-2 px-3"
          placeholder="Company Name"
        />
      </div>

      <div class="mb-4">
        <label
          for="company_description"
          class="block text-gray-700 font-bold mb-2"
          >Company Description</label
        >
        <textarea
          id="company_description"
          name="company_description"
          class="border rounded w-full py-2 px-3"
          rows="4"
          placeholder="What does your company do?"
        ></textarea>
      </div>

      <div class="mb-4">
        <label
          for="contact_email"
          class="block text-gray-700 font-bold mb-2"
          >Contact Email</label
        >
        <input
          type="email"
          id="contact_email"
          name="contact_email"
          class="border rounded w-full py-2 px-3"
          placeholder="Email address for applicants"
          required
        />
      </div>
      <div class="mb-4">
        <label
          for="contact_phone"
          class="block text-gray-700 font-bold mb-2"
          >Contact Phone</label
        >
        <input
          type="tel"
          id="contact_phone"
          name="contact_phone"
          class="border rounded w-full py-2 px-3"
          placeholder="Optional phone for applicants"
        />
      </div>

      <div>
        <button
          class="bg-green-500 hover:bg-green-600 text-white font-bold py-2 px-4 rounded-full w-full focus:outline-none focus:shadow-outline"
          type="submit"
        >
          Add Job
        </button>
      </div>
    </form>
  </div>
```

```
      </div>
    </section>
  </template>
```

In the script, let's bring in `reactive` and create an object for our form data. We will also have a function to handle the form submit:

```
<script setup>
import { reactive } from 'vue';

const form = reactive({
  type: 'Full-Time',
  title: '',
  description: '',
  salary: '',
  location: '',
  company: {
    name: '',
    description: '',
    contactEmail: '',
    contactPhone: '',
  },
  contact: '',
});

const handleSubmit = (event) => {
  event.preventDefault();
  console.log(form);
};
</script>
```

Now we need to link the form inputs with the data. We can do this with `v-model`:

```
<template>
  <section class="bg-green-50">
    <div class="container m-auto max-w-2xl py-24">
      <div
        class="bg-white px-6 py-8 mb-4 shadow-md rounded-md border m-4 md:m-0"
      >
        <form @submit.prevent="handleSubmit">
          <h2 class="text-3xl text-center font-semibold mb-6">Add Job</h2>

          <div class="mb-4">
            <label for="type" class="block text-gray-700 font-bold mb-2"
              >Job Type</label
            >
            <select
              v-model="form.type"
              id="type"
              name="type"
              class="border rounded w-full py-2 px-3"
              required
            >
              <option value="Full-Time">Full-Time</option>
              <option value="Part-Time">Part-Time</option>
              <option value="Remote">Remote</option>
              <option value="Internship">Internship</option>
            </select>
          </div>

          <div class="mb-4">
            <label class="block text-gray-700 font-bold mb-2"
              >Job Listing Name</label
            >
            <input
              type="text"
              id="name"
              name="name"
              v-model="form.title"
              class="border rounded w-full py-2 px-3 mb-2"
              placeholder="eg. Beautiful Apartment In Miami"
              required
```

```
      />
    </div>
    <div class="mb-4">
      <label for="description" class="block text-gray-700 font-bold mb-2"
        >Description</label
      >
      <textarea
        id="description"
        v-model="form.description"
        name="description"
        class="border rounded w-full py-2 px-3"
        rows="4"
        placeholder="Add any job duties, expectations, requirements, etc"
      ></textarea>
    </div>

    <div class="mb-4">
      <label for="type" class="block text-gray-700 font-bold mb-2"
        >Salary</label
      >
      <select
        id="salary"
        v-model="form.salary"
        name="salary"
        class="border rounded w-full py-2 px-3"
        required
      >
        <option value="Under $50K">under $50K</option>
        <option value="$50 - 60K">$50 - $60K</option>
        <option value="$60 - 70K">$60 - $70K</option>
        <option value="$70 - 80K">$70 - $80K</option>
        <option value="$80 - 90K">$80 - $90K</option>
        <option value="$90 - 100K">$90 - $100K</option>
        <option value="$100 - 125K">$100 - $125K</option>
        <option value="$125 - 150K">$125 - $150K</option>
        <option value="$150 - 175K">$150 - $175K</option>
        <option value="$175 - 200K">$175 - $200K</option>
        <option value="Over $200K">Over $200K</option>
      </select>
    </div>

    <div class="mb-4">
      <label class="block text-gray-700 font-bold mb-2"> Location </label>
      <input
        type="text"
        id="location"
        v-model="form.location"
        name="location"
        class="border rounded w-full py-2 px-3 mb-2"
        placeholder="Company Location"
        required
      />
    </div>

    <h3 class="text-2xl mb-5">Company Info</h3>

    <div class="mb-4">
      <label for="company" class="block text-gray-700 font-bold mb-2"
        >Company Name</label
      >
      <input
        type="text"
        id="company"
        v-model="form.company.name"
        name="company"
        class="border rounded w-full py-2 px-3"
        placeholder="Company Name"
      />
    </div>

    <div class="mb-4">
      <label
        for="company_description"
        class="block text-gray-700 font-bold mb-2"
        >Company Description</label
```

```
            >
            <textarea
              id="company_description"
              v-model="form.company.description"
              name="company_description"
              class="border rounded w-full py-2 px-3"
              rows="4"
              placeholder="What does your company do?"
            ></textarea>
          </div>

          <div class="mb-4">
            <label
              for="contact_email"
              class="block text-gray-700 font-bold mb-2"
              >Contact Email</label
            >
            <input
              type="email"
              id="contact_email"
              v-model="form.company.contactEmail"
              name="contact_email"
              class="border rounded w-full py-2 px-3"
              placeholder="Email address for applicants"
              required
            />
          </div>
          <div class="mb-4">
            <label
              for="contact_phone"
              class="block text-gray-700 font-bold mb-2"
              >Contact Phone</label
            >
            <input
              type="tel"
              id="contact_phone"
              v-model="form.company.contactPhone"
              name="contact_phone"
              class="border rounded w-full py-2 px-3"
              placeholder="Optional phone for applicants"
            />
          </div>

          <div>
            <button
              class="bg-green-500 hover:bg-green-600 text-white font-bold py-2 px-4 rounded-full w-full focus:outline-none focus:shadow-outline"
              type="submit"
            >
              Add Job
            </button>
          </div>
        </form>
      </div>
    </div>
  </section>
</template>
```

@submit.prevent

Usually when we submit a form, we want to prevent the default behavior of the form submission. Instead of adding the `event.preventDefault()`, we can just add the `@submit.prevent` directive to the form. This will prevent the default behavior of the form submission.

Delete the `event.preventDefault()` from the `submit` event handler and add the following to the form tag:

```
<form @submit.prevent="handleSubmit"></form>
```

Handling Form Submission

Now we need to handle the form submission. We will send the data to the server using Axios. We will also redirect the user to the job listings page after the form is submitted.

Add the following to the script:

```
const handleSubmit = async () => {
  const newJob = {
    title: form.title,
    type: form.type,
    location: form.location,
    description: form.description,
    salary: form.salary,
    company: {
      name: form.company.name,
      description: form.company.description,
      contactEmail: form.company.contactEmail,
      contactPhone: form.company.contactPhone,
    },
  };

  try {
    const response = await axios.post('/api/jobs', newJob);
    toast.success('Job Added Successfully');
    router.push(`/jobs/${response.data.id}`);
  } catch (error) {
    console.error('Error adding job:', error);
    toast.error('Failed to add job');
  }
};
```

Vue Toastification

We are using a toast library called `vue-toastification`. Let's install it:

```
npm install vue-toastification@next
```

We need to add the import for Toast and the CSS and use it in our `App.vue` file:

```
import { createApp } from 'vue';
import App from './App.vue';
import Toast from 'vue-toastification'; // Add this line
import 'vue-toastification/dist/index.css'; // Add this line
import '@/assets/main.css';
import 'primeicons/primeicons.css';
import router from './router';

const app = createApp(App);

app.use(router);
app.use(Toast); // Add this line

app.mount('#app');
```

Back in the `AddJobView.vue` file, import `axios`, `useToast` and `useRouter`:

```
import axios from 'axios';
import { useRouter } from 'vue-router';
import { useToast } from 'vue-toastification';
```

And initialize them under the `form` variable:

```
const router = useRouter();
const toast = useToast();
```

Submit the form and you should see a toast message and be redirected to the job listings page.

Deleting a Job

Now we want to be able to delete a job. We already have a button. We need to make it work by sending a delete request to the server.

Let's add a click event to the delete button in the `JobView.vue` file:

```
<button @click="deleteJob"
  class="bg-red-500 hover:bg-red-600 text-white font-bold py-2 px-4 rounded-full w-full focus:outline-none focus:shadow-outline mt-4 block"
>
  Delete Job
</button>
```

Import `useRouter` and `useToast`:

```
import { useRoute, RouterLink, useRouter } from 'vue-router';
import { useToast } from 'vue-toastification';
```

Initialize them:

```
const router = useRouter();
const toast = useToast();
```

Add the `deleteJob` function:

```
const deleteJob = async () => {
  try {
    await axios.delete(`/api/jobs/${jobId}`);
    toast.success('Job Deleted Successfully');
    router.push('/jobs');
  } catch (error) {
    console.error('Error deleting job:', error);
    toast.error('Failed to delete job');
  }
};
```

That's it, you should now be able to delete a job.

Editing a Job

The last major piece of functionality is editing the job. Let's create a new view at `src/views/EditJobView.vue`. Add the following:

```
<template>Edit page</template>
```

Now add the view to the `router/index.js` file:

```
import EditJobView from '@/views/EditJobView.vue';

//..
```

```
{
  path: '/jobs/edit/:id',
  name: 'edit-job',
  component: EditJobView,
},
```

If you go to any job page and click on the edit button, it should bring you to the correct page/view.

Add The Form

Let's just copy the code from the `AddJobView` page and paste it into the `EditJobView` page. Change the heading to say "Edit Job" and the button to say "Update Job".

Fetch Job Data

We need to fetch the data to fill the form. We can get the id from the route params and use it to fetch the job data.

Add the following to the script:

```
import { reactive, onMounted } from 'vue';
import { useRouter, useRoute } from 'vue-router';

const route = useRoute();

const jobId = route.params.id;

const state = reactive({
  job: {},
  isLoading: true,
});
```

Create an `onMounted` that gets the data and fills the job state:

```
onMounted(async () => {
  try {
    const response = await axios.get(`/api/jobs/${jobId}`);
    state.job = response.data;
    // Populate form fields after fetching job data
    form.type = state.job.type;
    form.title = state.job.title;
    form.description = state.job.description;
    form.salary = state.job.salary;
    form.location = state.job.location;
    form.company.name = state.job.company.name;
    form.company.description = state.job.company.description;
    form.company.contactEmail = state.job.company.contactEmail;
    form.company.contactPhone = state.job.company.contactPhone;
  } catch (error) {
    console.error('Error fetching job:', error);
  } finally {
    state.isLoading = false;
  }
});
```

Finally, update the `handleSubmit` to update the job:

```
const handleSubmit = async () => {
  const updatedJob = {
    title: form.title,
    type: form.type,
    location: form.location,
    description: form.description,
    salary: form.salary,
```

```
      company: {
        name: form.company.name,
        description: form.company.description,
        contactEmail: form.company.contactEmail,
        contactPhone: form.company.contactPhone,
      },
    };

    try {
      await axios.put(`/api/jobs/${jobId}`, updatedJob);
      toast.success('Job Updated Successfully');
      router.push(`/jobs/${jobId}`);
    } catch (error) {
      console.error('Error updating job:', error);
      toast.error('Failed to update job');
    }
  };
```

Now change something and update the job listing.

That's it my friends. We have a full CRUD application with Vue 3, the composition API and JSON Server. I hope that you learned a lot from this and you can get started with your own Vue.js projects.