by AndЯas

What is Vue.js?

- Progressive JS framework for building user interfaces & SPAs
- Designed to be simple, flexible and incrementally adoptable
- Used for projects of all sizes
- Reactive data-binding & component-based architecture

Vue Components

- Reusable, self-contained pieces of code
- Includes the logic/JS, dynamic HTML output & scoped styling
- Options API vs Composition API

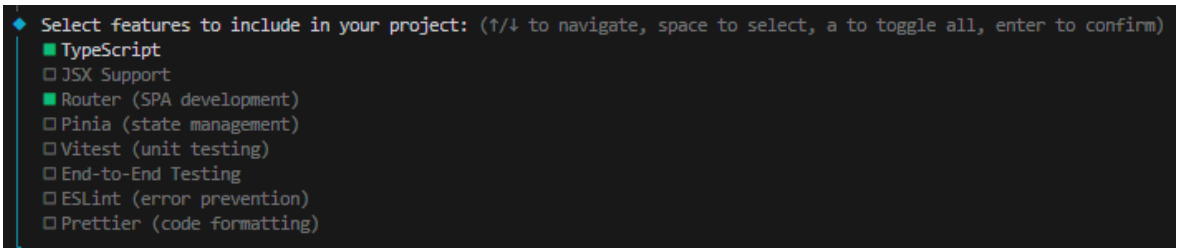
```
<script>
  // JavaScript/Logic
</script>

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

<style scoped>
  /* CSS Styling */
</style>
```

Getting Setup

- CDN - Include the script tag with the CDN url
- Vue CLI - Command line interface for setting up Vue projects. This is no longer recommended
- Create Vue - Uses Vite, which includes features like hot-reloading, out of the box TypeScript and an ecosystem of plugins
- Nuxt.js & Gridsome - SSR & SSG frameworks that use Vue

Setup

$ npm create vue@latest projectname



cd project folder
$ npm i

change vite config (optional to start server 3000)

```
// https://vite.dev/config/
export default defineConfig({
  plugins: [vue(), vueDevTools()],
  server: {
    port: 3000, // Set the port to 3000
    open: true, // Open the browser when the server starts
  },
  resolve: {
    alias: {
      "@": fileURLToPath(new URL("./src", import.meta.url)),
    },
  },
});
```

start server:
$ npm run dev

V-if, V-for, V-bind (:href), V-on (options api)

```vue
<script>
export default {
  data() {
    return {
      HelloWorld: 'Hello World',
      status: true,
      tasks: ["task1", "task2", "task3"],
      link: "https://vuejs.org",
    }
  },
}
</script>

<template>
  <h1>{{ HelloWorld }}</h1>
  <p v-if="status">This is a paragraph</p>
  <p v-else>This is another paragraph</p>
  <ul>
    <li v-for="(task, index) in tasks" :key="index">
      {{ task }}
    </li>
  </ul>
  <p>
    <a v-bind:href="link">Link</a>
    // or more simply
    <a :href="link">Link</a>
  </p>
  <button @click="status = !status">Toggle Paragraph</button>
</template>
```

Sidenote:

Interpolation refers to the process of inserting or embedding variables or expressions into a string. The {{ }} syntax is commonly used for interpolation in **templating languages** and **frontend frameworks** (like Handlebars, EJS, Vue.js, Angular, etc.).

- {{ }} is a placeholder for dynamic content.
- It tells the system: "Insert this variable or expression here."
- Widely used in frontend frameworks and templating engines.

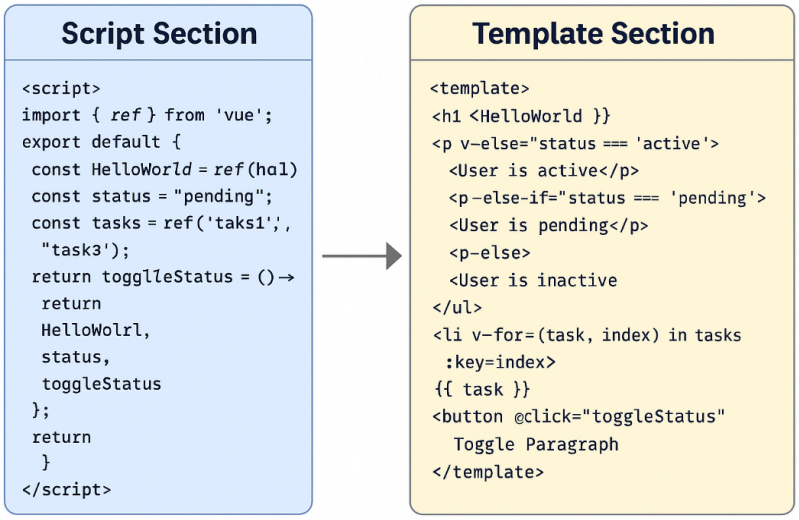
V-on, option 2: Creating methods, and have 3 paragraph tags but (@click is more used then V-on, but does the same thing)

```vue
    methods: {
    toggleStatus() {
      if (this.status === "active") {
        this.status = "pending";
      } else if (this.status === "pending") {
        this.status = "inactive";
      } else {
        this.status = "active";
      }
    }
  },
};

</script>

<template>
  <h1>{{ HelloWorld }}</h1>
  <p v-if="status === 'active'">User is active</p>
  <p v-else-if="status === 'pending'">User is pending</p>
  <p v-else>User is inactive</p>
  <ul>
    <li v-for="(task, index) in tasks" :key="index">
      {{ task }}
    </li>
  </ul>
  <p>
    <a v-bind:href="link">Link 1</a>
    // or more simply
    <a :href="link">Link 2</a>
  </p>
  <button v-on:click="toggleStatus">Toggle Paragraph</button>
</template>
```

V-if, V-for, V-on (composition api), longer way to doing it

This is a Vue 3 component that demonstrates the use of reactive properties and event handling in the less commonly used longer version using the Composition API. It uses the Composition API to define a setup function that initializes data and methods. The setup function returns an object that exposes the reactive properties and methods to the template. The template section uses Vue's directives to conditionally render content based on the status property. We need to import ref from vue that shows its reactive props, and change this.status to status.value.

```vue
<script>
import { ref } from 'vue';
// This is a Vue 3 component that demonstrates the use of reactive properties and event handling.
// It uses the Composition API to define a setup function that initializes data and methods.
export default {
  setup() {
    const HelloWorld = ref('Hello World');
    let status = ref('pending'); // Changed to let
    const tasks = ref(["task1", "task2", "task3"]);

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

<template>
  <h1>{{ HelloWorld }}</h1>
  <p v-if="status === 'active'">User is active</p>
  <p v-else-if="status === 'pending'">User is pending</p>
  <p v-else>User is inactive</p>
  <ul>
    <li v-for="(task, index) in tasks" :key="index">
      {{ task }}
    </li>
  </ul>
  <button @click="toggleStatus">Toggle Paragraph</button>
</template>
```



V-if, V-for, V-on (composition api), shorter more commonly used way

This is a Vue 3 component that demonstrates the use of reactive properties and event handling in the more concise commonly used Composition API style. Setup moved to the script tag, and due to this both the export and return can be removed.

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

    const HelloWorld = ref('Hello World');
    let status = ref('pending'); // Changed to let
    const tasks = ref(["task1", "task2", "task3"]);

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

</script>

<template>
  <h1>{{ HelloWorld }}</h1>
  <p v-if="status === 'active'">User is active</p>
  <p v-else-if="status === 'pending'">User is pending</p>
  <p v-else>User is inactive</p>
  <ul>
    <li v-for="(task, index) in tasks" :key="index">
      {{ task }}
    </li>
  </ul>
  <button @click="toggleStatus">Toggle Paragraph</button>
</template>
```

Form adding new tasks, del button removing tasks

- `v-model="newTask"` binds the `<input>` to the `newTask` variable reactively
- When the form is submitted (`@submit.prevent="addTask"`):
  - `.prevent` stops the default page reload.
  - the `addTask` function is called.

```
const HelloWorld = ref('Hello World');
let status = ref('pending'); // Changed to let
const tasks = ref(["task1", "task2", "task3"]);
const newTask = ref('');


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

const addTask = () => {
  if (newTask.value.trim() !== '') {
    tasks.value.push(newTask.value);
    newTask.value = '';
  }
};

const deleteTask = (index) => {
  tasks.value.splice(index, 1); // Remove the task at the specified index \
};

</script>

<template>
  <h1>{{ HelloWorld }}</h1>
  <p v-if="status === 'active'">User is active</p>
  <p v-else-if="status === 'pending'">User is pending</p>
  <p v-else>User is inactive</p>
  <h3>Tasks:</h3>
  <ul>
    <li v-for="(task, index) in tasks" :key="index">
      <span> {{ task }}
        <button @click="deleteTask(index)">x</button>
      </span>
    </li>
  </ul>
  <form @submit.prevent="addTask">
    <label for="newTask">New Task:</label>
    <input type="text" id="newTask" name="newTask" v-model="newTask" />
    <button type="submit">Add Task</button>
  </form>
  <br />
  <button @click="toggleStatus">Toggle Paragraph</button>
</template>
```

Lifecycle methods

- 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

Some examples:

Fetching data

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

Img import / display

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

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

Passing props

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

```
defineProps({
    title: String,
});
```

<-- this is also correct but with the example below if we are not passing a prop to Hero it will use the defined default option

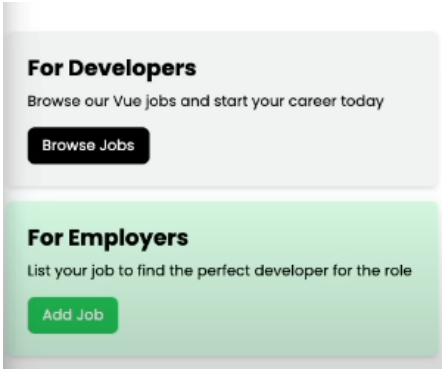
```
<script setup lang="ts">
import { defineProps } from 'vue';

defineProps({
    title: {
        type: String,
        default: 'Become a Vue Dev',
    },
});
</script>


<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">
                    {{ title }}
                </h1>
```

Reusable component, color as props

- `defineProps()` declares that the component accepts a `bgColor` **prop**, which should be a `String`.
- If **no prop** is passed, it defaults to `"bg-gray-100"` (a light gray Tailwind background).
- `:class="`${bgColor} ...`"` dynamically applies the background color class.
- `<slot></slot>` allows **content injection** from the parent component. <<-- this is like {children} in react

Carrd.vue

```vue
<script setup lang="ts">
defineProps({
    bgColor: {
        type: String,
        default: 'bg-gray-100',
    },
});
</script>

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



HomeCards.vue

```vue
<script setup lang="ts">
import Card from './Card.vue';
</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 bgColor="bg-gray-100">
                    <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 bgColor="bg-green-100">···
                </Card>
            </div>
        </div>
    </section>
</template>
```

Mapping and array of object to a reusable component

We have a jobs.json which contains an array of objects

- Import static job data from a local JSON file (`jobs.json`).
- Wrap it in `ref()` to make it **reactive** (if needed later, like for filtering or search).
- Import `JobListing.vue`, which is the **child component** to render individual jobs.

JobListings.vue

```vue
<script setup lang="ts">
import jobData from '../assets/jobs.json';
import JobListing from './JobListing.vue';
import { ref } from 'vue';


const jobs = ref(jobData);
console.log(jobData);


</script>

<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>
            <div class="grid grid-cols-1 md:grid-cols-3 gap-6">
                <JobListing v-for="job in jobs" :key="job.id" :job="job" />
            </div>
        </div>
    </section>
</template>
```

- The component **receives a single `job` object** as a prop.
- This `job` object has properties like `title`, `type`, `description`, `location`, `salary`, etc.

JobListings.vue

```vue
<script setup lang="ts">
defineProps({
    job: {
        type: Object,
        required: true,
    }
})
</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 }}</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
                    <i class="fa-solid fa-loca
                    {{ job.location }}
                </div>
                <a :href="'/job/' + job.id"
                    class="h-[36px] bg-green-5
                    text-sm">
                    Read More
                </a>
            </div>
        </div>
    </div>
</template>
```

`jobs.json`	Holds job data (like a mini backend for now)
`jobs = ref(jobData)`	Makes job list reactive
`<JobListing />`	A reusable child component to display each job
`:job="job"`	Passes data from parent to child via props
`defineProps()`	Used in child to receive props cleanly

Adding props to reusable component loop and optional button

- Limiting job listings by the number passed as prop otherwise default to the whole array
- Show button optional, default to false, not showing

App.vue

```vue
<template>
  <Navbar />
  <Hero />
  <HomeCards />
  <JobListings :limit="3" :showButton="true" />
</template>
```

JobListings.vue

```vue
defineProps({
    limit: Number,
    showButton: {
        type: Boolean,
        default: false,
    },
})

const jobs = ref(jobData);
console.log(jobData);

</script>

<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>
            <div class="grid grid-cols-1 md:grid-cols-3 gap-6">
                <JobListing v-for="job in jobs.slice(0, limit || jobs.length)" :key="job.id" :job="job" />
            </div>
        </div>
    </section>
    <section v-if="showButton" class="m-auto max-w-lg my-10 px-6">
        <a href="/jobs" class="block bg-black text-white text-center py-4 px-6 rounded-xl hover:bg-gray-700">View
            All Jobs</a>
    </section>
</template>
```

computed()

`computed()` lets you define a **value that depends on other reactive values** — and it **automatically updates** when those dependencies change. It's like a smarter `ref()` that *recalculates* only when needed.
It's similar to react useEffect dependency array.

Example follow below

Toggle description using computed()

- **Show a shortened job description by default**, and allow the user to **toggle to view the full description** (and back).
- This is a **reactive flag (ref)** used to track whether the full job description is shown (`true`) or just a preview (`false`).
- If `showFullDescription` is `false`, the description is cut to 90 characters with `"..."` appended.
    - If `true`, it returns the full description.
    - This computed property **automatically recalculates** when `showFullDescription` changes.
    - Displays either the **truncated** or **full** description.
- Button toggle simply flips the Boolean value on button click — toggling between **shortened** and **full** text.
    - Button text changes based on the current state (`More` / `Less`).

```
import { defineProps, ref, computed } from 'vue'

const props = defineProps({
    job: {
        type: Object,
        required: true,
    }
})

const showFullDescription = ref(false)

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

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

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

Install icons

$npm install primeicons

main.ts(or js)

```ts
import { createApp } from "vue";
import App from "./App.vue";
import "./assets/main.css";
import "primeicons/primeicons.css";

const app = createApp(App);

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

```vue
<div class="text-orange-700 mb-3">
    <i class="pi pi-map-marker"></i>
    {{ job.location }}
</div>
```

Router (manual setup)

Usually it would come bundled using create-vue if we opt in. However we can set it up manually:

- $ npm install vue-router
- Create folder in src called router and an index.js
- Create folder in src called views and an HomeView.vue
- Main.ts(js) we use the router
- App.vue (main component) use RouterView which is like an outlet

router.ts

```ts
import { createRouter, createWebHistory } from "vue-router"

import HomeView from "../views/HomeView.vue";

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

export default router; // create a router object
```

HomeView.vue

```vue
<script setup lang="ts">
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" :showButton="true" />
</template>
```

main.ts

```ts
import { createApp } from "vue";
import App from "./App.vue";
import "./assets/main.css";
import "primeicons/primeicons.css";

const app = createApp(App);

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

||---->>

```ts
import { createApp } from "vue";
import router from "./router";
import "./assets/main.css";
import "primeicons/primeicons.css";

import App from "./App.vue"; // import the App component

const app = createApp(App); // Pass the App component here
app.use(router); // use the router object
app.mount("#app");
```

App.vue

```vue
<script setup lang="ts">
import Navbar from './components/Navbar.vue';
import { RouterView } from 'vue-router';
</script>

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

Dynamic links in Nav using useRoute and ternary op

Client-side routing with no page reloads (like w anchor tags).
Highlighted nav link based on the current route (`bg-green-900`).
Dynamic classes using `useRoute` and conditional logic.

1. Add navigation links (`Home`, `Jobs`, `Add Job`).
    a. **RouterLink** — for Navigation
2. Highlight the **currently active page** with a different background color (e.g. `bg-green-900`).
    a. **useRoute()** — Getting the Current Route
3. Dynamic Class Binding
    a. Uses Vue's **computed class binding** with an array.
    b. If the route is active: Adds `'bg-green-900'` for highlighting the link.
    c. If not: Adds `'hover:bg-gray-900'` for hover effect only.
    d. Remaining classes (`text-white`, `px-3`, etc.) are always applied

```vue
<script setup lang="ts">
import { RouterLink, useRoute } from 'vue-router';
import logo from '../assets/img/logo.png';

const isActiveLink = (routePath: string) => {
    const route = useRoute();
    return route.path === routePath;
};

</script>

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

Dynamic links to individual pages and not found

Element	Purpose
`<RouterLink :to="...">`	Creates a dynamic link to a job's detail page
`/jobs/:id`	Defines the dynamic route and loads `JobView.vue`
`:id`	Is a route parameter passed in the URL
`:catchAll(.*)`	Catches any unmatched routes and shows a "Not Found' page

JobsView.vue

```vue
<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>
```

Router.vue

```
{
  path: "/jobs/:id",
  name: "job",
  component: JobView,
},
{
  path: "/:catchAll(.*)",
  name: "not-found",
  component: NotFoundView,
},
```

Running local mock server

Using json server to load our jobs like a back end, it would still run locally (it still loads from the same json file, but json-server sets it up) but we would send get **post,put delete,requests**

$ npm install json-server

Edit jobs.json

Change the array of jobs to and object that has jobs that contains the array:

```
[
  {
    "id": 1,
    "title": "Senior Vue Developer",
    "type": "Full-Time",
```

||==>>

```
{
  "jobs":[
  {
    "id": 1,
    "title": "Senior Vue Developer",
    "type": "Full-Time",
```

Edit package.json to run our local server

```
"scripts": {
  "dev": "vite",
  "build": "run-p type-check \"build-only {@}\" --",
  "preview": "vite preview",
  "server": "json-server --watch src/assets/jobs.json --port 5000",
  "build-only": "vite build",
  "type-check": "vue-tsc --build"
},
```

To run:

$ npm run dev

```
PS C:\Users\rohad\Documents\GitHub\applied> npm run dev

> vue-tutorial@0.0.0 dev
> vite


  VITE v6.2.6  ready in 2503 ms

  ➜  Local:   http://localhost:3000/
  ➜  Network: use --host to expose
  ➜  Vue DevTools: Open http://localhost:3000/__devtools__/ as a separat
e window
  ➜  Vue DevTools: Press Alt(⌥)+Shift(⇧)+D in App to toggle the Vue DevT
ools
```

$ npm run server

```
PS C:\Users\rohad\Documents\GitHub\applied> npm run server

> vue-tutorial@0.0.0 server
> json-server --watch src/assets/jobs.json --port 5000

--watch/-w can be omitted, JSON Server 1+ watches for file changes by de
fault
JSON Server started on PORT :5000
Press CTRL-C to stop
Watching src/assets/jobs.json...

ᕙ( ◡‿◡ )ᕗ
Serving ./public directory if it exists
```

Change fetch to axios (optional)

$ npm i axios

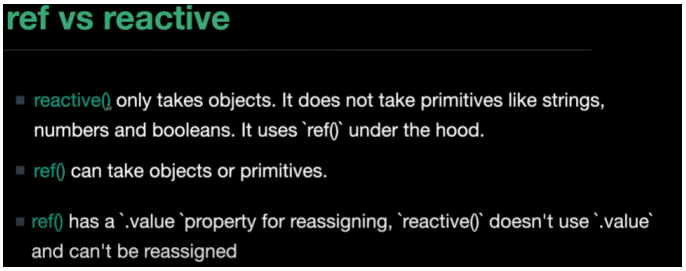
Remove/edit JobListings.vue as we will have date server by json server instead, we will hook on the onMounted life-cycle method to get the data

```
    <script setup lang="ts">
-   import jobData from '../assets/jobs.json';
    import JobListing from './JobListing.vue';
    import { RouterLink } from 'vue-router';
    import { ref } from 'vue';
    @@ -12,7 +11,7 @@ defineProps({
        },
    })

-   const jobs = ref(jobData);
+   const jobs = ref([]);
```

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

Ref vs reactive

In react term it would be like
useState(“something”), useState(“something2”) **vs** useState({first: “something”, second: “something”})



Refactoring code to use json-server and reactive (also create a TS interface for the object), and isLoading boolean in the state object is used to track whether the data (jobs) is currently being fetched from the API. It helps manage the loading state of the component and can be used to display a loading indicator or spinner while the data is being retrieved.

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

// Define the Job interface
interface Job {
    id: number;
    title: string;
    description: string;
    type: string;
    salary: string;
    location: string;
}

defineProps({
    limit: Number,
    showButton: {
        type: Boolean,
        default: false,
    },
});

// Define the state with the Job type
const state = reactive<{
    jobs: Job[];
    isLoading: boolean;
}>({
    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>

<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>
            <!-- Loading spinnger while loading true -->
            <div v-if="state.isLoading" class="text-center text-gray-500 py-6">
                <PulseLoader />
            </div>
            <!-- Display jobs when loading is false -->
            <div v-else class="grid grid-cols-1 md:grid-cols-3 gap-6">
                <JobListing v-for="job in state.jobs.slice(0, limit || state.jobs.length)" :key="job.id" :job="job" />
            </div>
        </div>
    </section>
```

useRoute for individual job listing

Set up JobView, so each individual listing will display the job using their id.

router/index.ts

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

JobView (as using TS set up interfaces) to receive the object via json-server (mind company also and object)

```
"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 JavaScript, with
  experience working with modern JavaScript frameworks such as Vue or Angular.",
  "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 high-quality products
    and services to our clients while fostering a collaborative and innovative work
    environment.",
    "contactEmail": "contact@teksolutions.com",
    "contactPhone": "555-555-5555"
  }
},
```

Defines the shape of a `Job` and its `Company`.

```
const route = useRoute();
const jobId = route.params.id as string;

interface Company {
    name: string;
    description: string;
    contactEmail: string;
    contactPhone: string;
}

interface Job {
    id: number;
    title: string;
    description: string;
    company: Company; // Use the Company interface
    type: string;
    salary: string;
    location: string;
}
```

`reactive`

- `Partial<Job>` allows initializing an empty object (some or all properties can be missing).
- `isLoading` is `true` until data is fetched.

onMounted

- When the component mounts, it fetches job data using the ID from the URL.
- On success, it saves the job info into `state.job`.
- Whether success or failure, it sets `isLoading` to `false`.

```
const state = reactive<{
    job: Partial<Job>; // Allow an empty object initially
    isLoading: boolean;
}>({
    job: {}, // Initialize as an empty object
    isLoading: true,
});

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

Displays the UI w conditional rendering

```
<section v-if="!state.isLoading">
    ...
</section>

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

In full example next page:

```vue
<script setup lang="ts">
import { reactive, onMounted } from 'vue';
import PulseLoader from 'vue-spinner/src/PulseLoader.vue';
import { useRoute } from 'vue-router';
import axios from 'axios';

const route = useRoute();
const jobId = route.params.id as string;

interface Company {
    name: string;
    description: string;
    contactEmail: string;
    contactPhone: string;
}

interface Job {
    id: number;
    title: string;
    description: string;
    company: Company; // Use the Company interface
    type: string;
    salary: string;
    location: string;
}

const state = reactive<{
    job: Partial<Job>; // Allow an emp
    isLoading: boolean;
}>({
    job: {}, // Initialize as an empty
    isLoading: true,
});

onMounted(async () => {
    try {
        const response = await axios.g
        state.job = response.data; //
    } catch (error) {
        console.error('Error fetching
    } finally {
        state.isLoading = false;
    }
});
</script>
```

```vue
<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
                        <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-cent
                            <i class="fa-solid fa-location-dot text-lg text-orange-70
                            <p class="text-orange-700">{{ state.job.location }}</p>
                        </div>
                    </div>

                    <div class="bg-white p-6 rounded-lg shadow-md mt-6">···
                    </div>
                </main>

                <!-- Sidebar -->
                <aside>
                    <!-- Company Info -->
                    <div v-if="state.job.company" class="bg-white p-6 rounded-lg shad
                        <h3 class="text-xl font-bold mb-6">Company Info</h3>
                        <h2 class="text-2xl">{{ state.job.company.name }}</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.compa
                    </div>

                    <!-- Manage -->
                    <div class="bg-white p-6 rounded-lg shadow-md mt-6">···
                    </div>
                </aside>
            </div>
        </div>
    </section>
    <div v-else="state.isLoading" class="text-center text-gray-500 py-6">
        <PulseLoader />
    </div>
</template>
```

Proxying

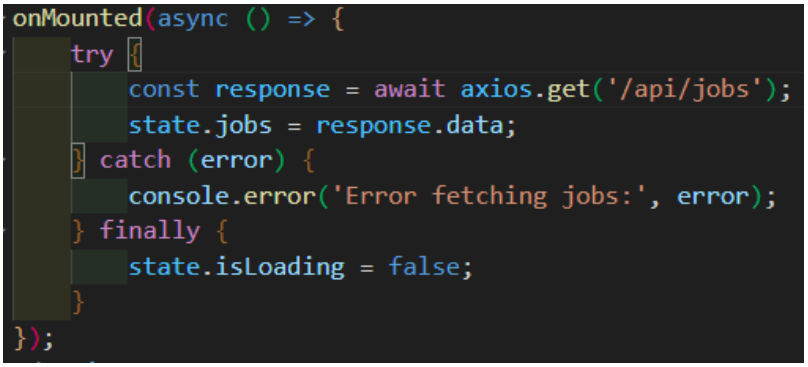
We have several requests that goes to the same http://localhost:5000/ with different endings. We could set this up in vite

- **Proxying** avoids CORS issues during local development (Cross-Origin Resource Sharing).
- It makes API calls feel like they're to the same origin (the frontend server).
- You configure it in `vue.config.js` (Vue CLI) or `vite.config.ts` (Vite).
- In production, your real server (e.g., Nginx or Node.js) handles routing.

vite.config.ts

```ts
// https://vite.dev/config/
export default defineConfig({
  plugins: [vue(), vueDevTools()], // Combined all plugins into one array
  server: {
    port: 3000, // Set the port to 3000
    open: true, // Open the browser when the server starts
    proxy: {
      // Proxy configuration
      "/api": {
        target: "http://localhost:5000", // Target API URL
        changeOrigin: true, // Change the origin of the host header to the target URL
        rewrite: (path) => path.replace(/^\/api/, ""), // Rewrite the path to remove the /api prefix
      },
    },
  },
  resolve: {
    alias: {
      "@": fileURLToPath(new URL("./src", import.meta.url)),
    },
  },
});
```

With this now we can replace the code to be



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

Addlisting

Creating AddJob.vue in Views and adding it to router

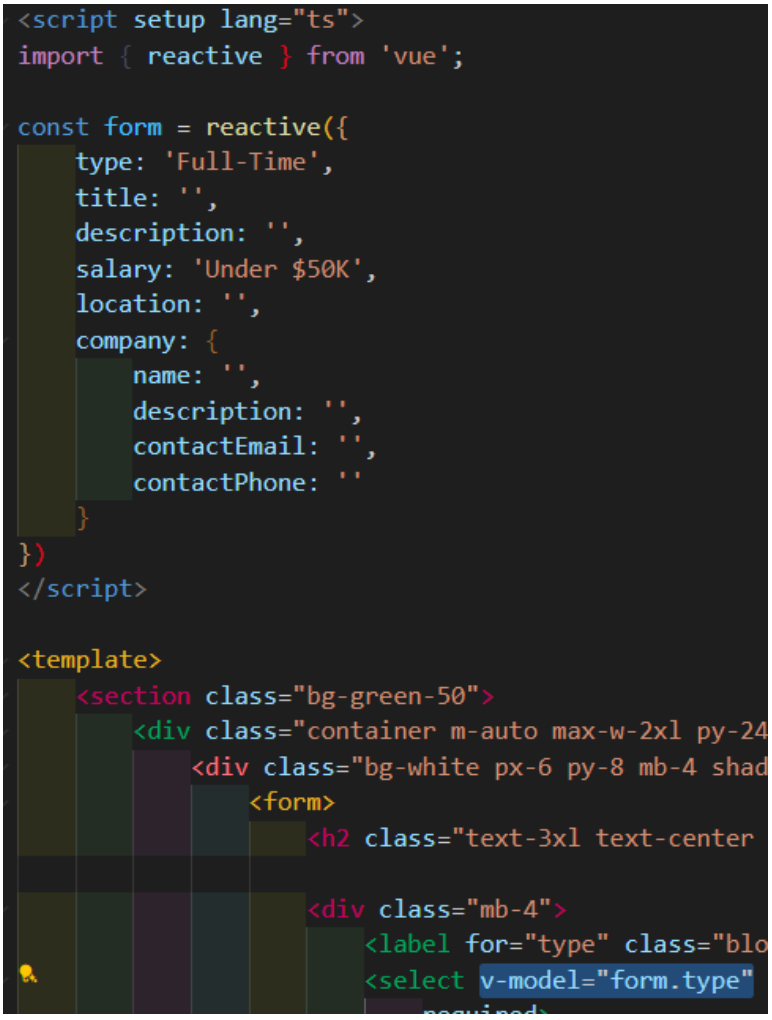
```
{
  path: "/jobs/add",
  name: "add-job",
  component: AddJob,
},
```

Setting up AddJob.vue, we set the type to Full-Type to have an initial value, this has to match one of the options in the selection, otherwise it won't show. All inputs/textareas added with v-model.

`v-model` is a **two-way binding directive** in Vue. It connects **form input elements** (like `input`, `textarea`, `select`) with your Vue component's data, so:

- The UI updates **when the data changes**
- The data updates **when the user types or selects something**

It's sugar for listening to `input` events and updating a variable.



In this case, we don't strictly need to create an interface because Vue's reactive and v-model bindings will work without explicit type definitions.

handleSubmit async function

- Builds a job object from form input
- Sends it to the backend with `axios.post`
- On success: redirects to the newly created job's detail page (In Vue (with **Vue Router**), `router.push()` is used to **programmatically navigate** to a new route — the same way you'd do it by clicking a `<RouterLink>`.)
- On error: logs the error (to be improved)

```
const form = reactive({
    type: 'Full-Time',
    title: '',
    description: '',
    salary: 'Under $50K',
    location: '',
    company: {
        name: '',
        description: '',
        contactEmail: '',
        contactPhone: ''
    }
})

const handleSubmit = async () => {
    const newJob = {
        title: form.title,
        type: form.type,
        description: form.description,
        salary: form.salary,
        location: form.location,
        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);
        // @todo - show toast message
        router.push(`/jobs/${response.data.id}`);
    } catch (error) {
        // @todo - handle error properly
        console.error('Error fetching job:', error);
    }
}
</script>

<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" class="w-full max-w-lg mx-auto">
                    <h2 class="text-3xl text-center font-semibold mb-6">Add Job</h2>
```

Toastification

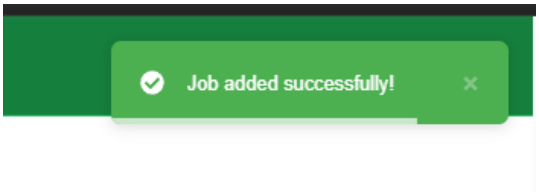
npm i vue-toastification@next

main.ts

```
import { createApp } from "vue";
import router from "./router";
import "./assets/main.css";
import "primeicons/primeicons.css";
import Toast from "vue-toastification";
import "vue-toastification/dist/index.css";

import App from "./App.vue"; // import the App component

const app = createApp(App); // Pass the App component here
app.use(router); // use the router object
app.use(Toast); // use the Toast plugin
app.mount("#app");
```

AddJobsView.vue

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

const form = reactive({ ···
})

const toast = useToast();

const handleSubmit = async () => {
    const newJob = { ···
    }
    try {
        const response = await axios.post(`/api/jobs`, newJob);
        toast.success('Job added successfully!');
        router.push(`/jobs/${response.data.id}`);
    } catch (error) {
        toast.error('Failed to add job.');
        console.error('Error fetching job:', error);
    }
}
```



Delete listing

JobView.vue

```vue
<script setup lang="ts">
import { reactive, onMounted } from 'vue';
import PulseLoader from 'vue-spinner/src/PulseLoader.vue';
import BackButton from '@/components/BackButton.vue';
import { useRoute, RouterLink, useRouter } from 'vue-router';
import axios from 'axios';
import { useToast } from 'vue-toastification';

const route = useRoute();
const router = useRouter();
const toast = useToast();
const jobId = route.params.id as string;

interface Company {...
}

interface Job {...
}


const state = reactive<{
    job: Partial<Job>; // Allow an empty object initially
    isLoading: boolean;
}>({
    job: {}, // Initialize as an empty object
    isLoading: true,
});

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

onMounted(async () => {
    try {
        const response = await axios.get(`/api/jobs/${jobId}`);
        state.job = response.data; // Assign the fetched job data
    } catch (error) {
        console.error('Error fetching job:', error);
    } finally {
        state.isLoading = false;
    }
});
</script>
```

Edit listing

- Fetches a specific job by ID (from the route).
- Pre-fills a form with that job's data.
- Allows editing and updating the job.
- Submits the update with an HTTP PUT request.
- Provides user feedback via toasts.

JobView.vue

```
<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 block">
    Edit
    Job</RouterLink>
```

Router.ts

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

EditJobView.vuew

```
<script setup lang="ts">
import { reactive, onMounted } from 'vue';
import { useRoute, useRouter } from 'vue-router'; // useRoute to get the job ID from the route params
import { useToast } from 'vue-toastification';
import axios from 'axios';

const route = useRoute();
const router = useRouter();
const jobId = route.params.id as string; // Get the job ID from the route params
const toast = useToast();
```

imports

- `reactive`: Makes objects reactive to changes.
- `onMounted`: Runs code when the component is mounted.
- `useRoute`: Access current route (needed for job ID).
- `useRouter`: Used for navigation (e.g., redirect after update).
- `useToast`: For showing success or error messages.
- `axios`: For making HTTP requests.

Route & Router setup

- Extracts the `jobId` from the route (`/jobs/:id`).
- Sets up `toast` for notification messages.

Form state (two-way bound to inputs)

- This is the form data the user can change.
- It's initialized to default/empty values.

App-level state

- Holds the fetched job data and a loading flag.
- Used to populate the form on initial load.

```vue
<script setup lang="ts">
import { reactive, onMounted } from 'vue';
import { useRoute, useRouter } from 'vue-router';
import { useToast } from 'vue-toastification';
import axios from 'axios';

const route = useRoute();
const router = useRouter();
const jobId = route.params.id as string; // Get t
const toast = useToast();


const form = reactive({
    type: 'Full-Time',
    title: '',
    description: '',
    salary: 'Under $50K',
    location: '',
    company: {
        name: '',
        description: '',
        contactEmail: '',
        contactPhone: ''
    }
})

const state = reactive({
    job: {
        id: 0,
        title: '',
        description: '',
        company: {
            name: '',
            description: '',
            contactEmail: '',
            contactPhone: ''
        },
        type: '',
        salary: '',
        location: ''
    },
    isLoading: true,
});
```

```vue
const handleSubmit = async () => {
    const updatedJob = {
        title: form.title,
        type: form.type,
        description: form.description,
        salary: form.salary,
        location: form.location,
        company: {
            name: form.company.name,
            description: form.company.description,
            contactEmail: form.company.contactEmail,
            contactPhone: form.company.contactPhone
        }
    }
    try {
        const response = await axios.put(`/api/jobs/${jobId}`, updatedJob);
        toast.success('Job updated successfully!');
        router.push(`/jobs/${response.data.id}`);
    } catch (error) {
        toast.error('Failed to update job.');
        console.error('Error updating job:', error);
    }
}

onMounted(async () => {
    try {
        const response = await axios.get(`/api/jobs/${jobId}`);
        state.job = response.data; // Assign the fetched job data
        // Populate the form with the fetched job data
        form.title = state.job.title;
        form.type = state.job.type;
        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;
    }
});
</script>
```


Fin