Introduction to Vue.js

with examples and
hands-on exercises

WEBUCATOR

Version: 1.0.1

The Authors

Jared Dunn

In addition to supporting students in self-paced, online courses and writing and tech editing courseware, Jared is a Full Stack Developer at Webucator. He builds and maintains many of Webucator's internal and external applications. He has a BA in Computer Science and a BA in International and Public Affairs from Brown University.

Chris Minnick

Chris Minnick, the co-founder of WatzThis?, has overseen the development of hundreds of web and mobile projects for customers from small businesses to some of the world's largest companies. A prolific writer, Chris has authored and co-authored books and articles on a wide range of Internet-related topics including HTML, CSS, mobile apps, e-commerce, e-business, Web design, XML, and application servers. His published books include Adventures in Coding, JavaScript For Kids For Dummies, Writing Computer Code, Coding with JavaScript For Dummies, Beginning HTML5 and CSS3 For Dummies, Webkit For Dummies, CIW E-Commerce Designer Certification Bible, and XHTML.

Jared Dunn (Editor)

In addition to supporting students in self-paced, online courses and writing and tech editing courseware, Jared is a Full Stack Developer at Webucator. He builds and maintains many of Webucator's internal and external applications. He has a BA in Computer Science and a BA in International and Public Affairs from Brown University.

Class Files

Download the class files used in this manual at https://static.webucator.com/media/public/materials/classfiles/VUE105-1.0.1-introduction-to-vuejs.zip.

Table of Contents

LESSON 1

Overview and Introduction to Vue.js

Topics Covered

☑ Overview of Vue.js and its core ideas.

☑ Common use cases for Vue.

☑ Fundamentals of Vue components.

☑ Comparing the Composition API and Options API.

☑ Introduction to the PetVue course project and goals.

Introduction

In this lesson, you will be introduced to the powerful JavaScript framework, Vue.js, and discover how it can simplify the development of interactive web applications. We will guide you through Vue's components and key concepts, providing a foundation for building dynamic user interfaces. You'll also learn about PetVue, the pet adoption demo app that you'll be building throughout the course. By the end of this lesson, you'll gain a clear understanding of Vue's usability and the different approaches to creating Vue components, setting the stage for developing advanced Vue applications.

EVALUATION COPY: Not to be used in class.

1.1. Introduction to Vue.js

Before we dive into what Vue.js (sometimes just called Vue) is all about, let's look at a simple demo.

EVALUATION COPY: Not to be used in class.

One of the simplest apps you can build is a counter. A typical counter demo app shows a number and has buttons that the user can use to increment, decrement, and reset that number to 0. If you were tasked with building this app, you might start with some HTML that looks like this:

Demo 1.1: OverviewIntroductionVuejs/Demos/counter-html.html

```
1.   <!DOCTYPE html>
2.   <html lang="en">
3.   <head>
4.   <meta charset="UTF-8">
5.   <title>Counter</title>
6.   </head>
7.   <body>
8.     <h1>Counter</h1>
9.     <p>0</p>
10.    <button>Increment</button>
11.    <button>Decrement</button>
12.    <button>Reset</button>
13.  </body>
14.  </html>
```


At this point, of course, it's just a static web page. To add interactivity, you need to use JavaScript to:

1. Select the `p` element that displays the count so you can change it with the buttons.
2. Select each of the buttons so that you can add event listeners to them.
3. Add event listeners to each of the buttons to add the distinct behaviors that you want (increment, decrement, reset).

Try It Yourself!

If you like, go ahead an see if you can create this yourself before looking at how we do it. You can find the starter HTML at `OverviewIntroductionVuejs/Demos/counter-html.html`.

After adding interactivity with plain "vanilla" JavaScript, the app might look something like the following:

Demo 1.2: OverviewIntroductionVuejs/Demos/counter.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <title>Counter</title>
6.  </head>
7.  <body>
8.    <h1>Counter</h1>
9.    <p id="counter">0</p>
10.   <button id="increment">Increment</button>
11.   <button id="decrement">Decrement</button>
12.   <button id="reset">Reset</button>
13.   <script>
14.     let count = 0; // Initialize count variable
15.
16.     // Get references to the DOM elements
17.     const counterElement = document.getElementById('counter');
18.     const incrementButton = document.getElementById('increment');
19.     const decrementButton = document.getElementById('decrement');
20.     const resetButton = document.getElementById('reset');
21.
22.     // Add event listeners to the buttons
23.     incrementButton.addEventListener('click', () => {
24.       count++; // Increment count
25.       counterElement.textContent = count; // Update displayed count
26.     });
27.     decrementButton.addEventListener('click', () => {
28.       count--; // Decrement count
29.       counterElement.textContent = count; // Update displayed count
30.     });
31.     resetButton.addEventListener('click', () => {
32.       count = 0; // Reset count to 0
33.       counterElement.textContent = count; // Update displayed count
34.     });
35.   </script>
36. </body>
37. </html>
```

Even with such a simple app, I've already written 20+ lines of code and I've had to modify the HTML to make it more accessible to the JavaScript. The program also has to make sure that the `count` variable and the count displayed in the "count" `p` element are in sync with each other by always updating both.

Let's see how Vue can simplify things. Here's the same basic counter applications written with Vue:

Demo 1.3: OverviewIntroductionVuejs/Demos/counter-vue.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <title>Counter - Vue</title>
6.  <!-- Include Vue via CDN script -->
7.  <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
8.  </head>
9.  <body>
10.   <h1>Counter - Vue</h1>
11.   <div id="app">
12.     <p>{{ count }}</p>
13.     <button @click="count++">Increment</button>
14.     <button @click="count--">Decrement</button>
15.     <button @click="count=0">Reset</button>
16.   </div>
17.   <script>
18.     const { createApp, ref } = Vue; // get createApp and ref from Vue
19.
20.     // Create a Vue instance
21.     createApp({
22.       setup() {
23.         const count = ref(0); // Reactive reference to count
24.         return {
25.           count // Expose count to the template
26.         }
27.       }
28.     }).mount('#app') // Mount the Vue instance to the DOM element with id 'app'
29.   </script>
30. </body>
31. </html>
```


You can see right away that the Vue version now has fewer lines of code, but it may not be apparent right away how the Vue code works. Here's what's happened:

1. We included the Vue framework in our app using its CDN (`https://vuejs.org/guide/quick-start.html#using-vue-from-cdn`).

2. We added a `<div id="app"></div>` element around the part of our page that should be rendered with Vue.
3. We imported some important methods from Vue that we use to set up the "app" `div` to render Vue and make our `count` variable reactive. Don't worry about the details here right now. All of this will be explained in depth later on.
4. We added `{{ count }}` to the `p` element to render the `count` variable automatically into the HTML.
5. We removed the now unnecessary `id`s from each element (since we no longer need to grab them directly) and replaced the `addEventListener` calls with Vue's simple `@click` event listener to update the `count` variable directly where those buttons are in the HTML.

A fundamental difference between this Vue.js example and the vanilla JavaScript version is that the Vue application is *reactive*. A reactive web application is one in which the user interface (UI) is automatically updated in response to changes to data. In the case of this simple program, the value of `count` that's displayed in the browser is updated automatically by Vue, without the need for you to explicitly program that functionality.

Vue is one of the most popular JavaScript frameworks today. It sits alongside React and Angular as a top choice for building modern web applications. Many developers like Vue because it is easy to learn, flexible, and has a strong community.

If you are new to building web apps, Vue is a great place to start. If you have experience with other frameworks, you will notice that Vue is designed to be approachable and productive. It can be used for small projects or large, complex applications.

In the rest of this course, you will learn more about Vue's features and how to use them to build real apps. You will see why so many developers choose Vue for their projects. Let's get started!

1.2. Different Uses of Vue

Another reason that Vue is appealing to many developers is that it can be used in a wide variety of ways. In the previous demo (`OverviewIntroductionVuejs/Demos/counter-vue.html`), we used it as a **standalone script**. This is the simplest way to use Vue. The standalone script method is a good choice if your site already has most of its HTML generated by the backend, or if you want to add some

interactivity without a complex setup. By just including a script tag that fetches Vue from a content delivery network (CDN), you get reactivity and easier DOM updates without needing a build step or a big toolchain.

Vue CDN

Documentation on using Vue from the CDN can be found here:

`https://vuejs.org/guide/quick-start.html#using-vue-from-cdn`

The `script` tag for using Vue from the CDN looks like this:

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

Let's look at some other ways that Vue can be used:

❖ 1.2.1. Single-Page Applications (SPAs)

Web apps that require lots of interactivity, real-time updates, and complex state management are good candidates for Single-Page Applications (SPAs). In an SPA, Vue runs the whole frontend. Instead of a new HTML page loading when the user navigates the app, JavaScript code updates the content dynamically inside a single HTML page. This approach is ideal for apps like dashboards or social platforms, where users expect fast, seamless interactions.

This is the exact scenario that Vue and similar JavaScript frameworks were designed to handle, and it is what we are going to focus on for the rest of the course.

❖ 1.2.2. Fullstack and Server-Side Rendering (SSR)

Two caveats of SPAs are:

1. They are not always great for search engine optimization (SEO) because their dynamically rendered content is harder for search engines to index (although search engines have largely solved this problem).
2. If you need interaction with a backend, they require building a separate backend and then connecting to that backend via an API.

To counter these two potential issues, Vue also has a way for building a complete fullstack application with Vue where the content is rendered on the server side. This integrates your backend and your

frontend together into a single app. Because the content is rendered first on the server, it is easier for search engines to index it.

❖ 1.2.3. Other Ways of Using Vue

There are so many cool ways to use Vue! Here are just a few more:

- Creating custom **web components** that can be used like HTML elements on any HTML page, no matter how it was built.
- Creating **mobile and desktop apps**, so you can use Vue even off the web!
- Generating complete **static sites** that load super fast and can be deployed anywhere.

You can read more details about all of this in `Vue's official documentation (https://vuejs.org/guide/extras/ways-of-using-vue.html#ways-of-using-vue)`!

1.3. Introduction to Vue Components

Now that you've seen what Vue can do and the different ways it can be used, let's talk about how you actually write Vue apps in practice. In modern Vue projects, especially when building Single-Page Applications, you'll spend most of your time writing code in Vue components.

Standalone Script Exception
As you have already seen, the exception to this rule is when you are using Vue as a **standalone script**. With a standalone script, you write your Vue directly in your **.html** (or in some cases, your `.js`) files.

A Vue component is a reusable, self-contained block of code that represents a part of your user interface. Each component usually lives in its own file with a `.vue` extension. These files are called "Single File Components" (SFCs).

A `.vue` file is divided into three main sections:

- `<template>`: This is where you write your HTML. The template describes what should appear on the screen.
- `<script>`: This is where you write your JavaScript. The script section handles the logic and data for your component.
- `<style>`: This is where you add CSS to style your component. Styles can be scoped (`<style scoped>`) so they only apply to this component.

The only required section is the `<template>` section. If your component doesn't need any JavaScript or CSS, you can leave those sections out.

Here is a simple example of the basic structure of a Vue component:

Example Vue Component

```
<template>
  <!-- HTML goes here - so I can use HTML comments -->
</template>

<script>
  // JavaScript goes here - so I can use JavaScript comments
</script>

<style>
  /* CSS goes here - so I an use CSS comments */
</style>
```


You can think of components as the building blocks of your app. Each part of your UI (like a button, a form, or even the whole page) can be a component. By breaking your app into components, your code is easier to read, reuse, and maintain.

In the next sections, you'll learn more about the different ways to write the script section of your components, but for now, just know that every Vue app is built from these simple, powerful building blocks: Vue components in `.vue` files.

1.4. Composition API vs Options API

Vue gives you two main ways to write your components: the **Options API** and the **Composition API**. Both let you build interactive, reactive apps, but they have different styles and strengths.

❖ 1.4.1. Options API

The Options API is the original way to write Vue components. It's organized around options (like `data`, `methods`, and `mounted`) that you specify as properties in a JavaScript object. Each option has its own job: `data` holds your reactive state, `methods` defines functions, and `mounted` is a lifecycle hook that runs when your component is ready.

Continuing with the Counter app example, here is the Counter app written as a Vue component in the Options API style:

Demo 1.4: OverviewIntroductionVuejs/Demos/OptionsCounter.vue

```
<template>
  <p>{{ count }}</p>
  <button @click="increment">Increment</button>
  <button @click="decrement">Decrement</button>
  <button @click="reset">Reset</button>
</template>

<script>
  export default {
    name: 'OptionsCounter',
    data() {
        return {
            count: 0
        }
    },
    methods: {
        increment() {
            this.count++;
        },
        decrement() {
            this.count--;
        },
        reset() {
            this.count = 0;
        }
    },
    mounted() {
        console.log('Component mounted');
    },
  }
</script>
```


This style is easy to learn, especially if you're new to JavaScript frameworks or come from an object-oriented background. The Options API keeps your code organized by grouping it by type.

❖ 1.4.2. Composition API

The Composition API is newer. It was added in Vue 3 to solve some of the limitations of the Options API, especially for bigger projects. With the Composition API, you use imported functions like `ref()`

and `onMounted()` to declare state and logic directly in a function. The Composition API gives you much more flexibility with how you organize your code, allowing you to group your code by related logic rather than having to separate it by parts. Here is the Counter app rewritten with the Composition API:

Demo 1.5: OverviewIntroductionVuejs/Demos/CompositionCounter.vue

```
1.  <template>
2.    <p>{{ count }}</p>
3.    <button @click="increment">Increment</button>
4.    <button @click="decrement">Decrement</button>
5.    <button @click="reset">Reset</button>
6.  </template>
7.
8.  <script setup>
9.    import { ref, onMounted } from 'vue';
10.
11.   const count = ref(0);
12.
13.   function increment() {
14.     count.value++;
15.   }
16.   function decrement() {
17.     count.value--;
18.   }
19.   function reset() {
20.     count.value = 0;
21.   }
22.
23.   onMounted(() => {
24.     console.log('Component mounted');
25.   });
26. </script>
```


With the Composition API, you get:

1. **Flexible code organization:** You can group code by what it does, not just by where it fits in an options object.
2. **Easier logic reuse**: You can create reusable functions (called "composables") that let you share logic across components.
3. **Great TypeScript support**: For those of you wanting to write with `TypeScript` (`https://www.typescriptlang.org/`), the Composition API makes it much easier to use TypeScript in your Vue components compared to the Options API.

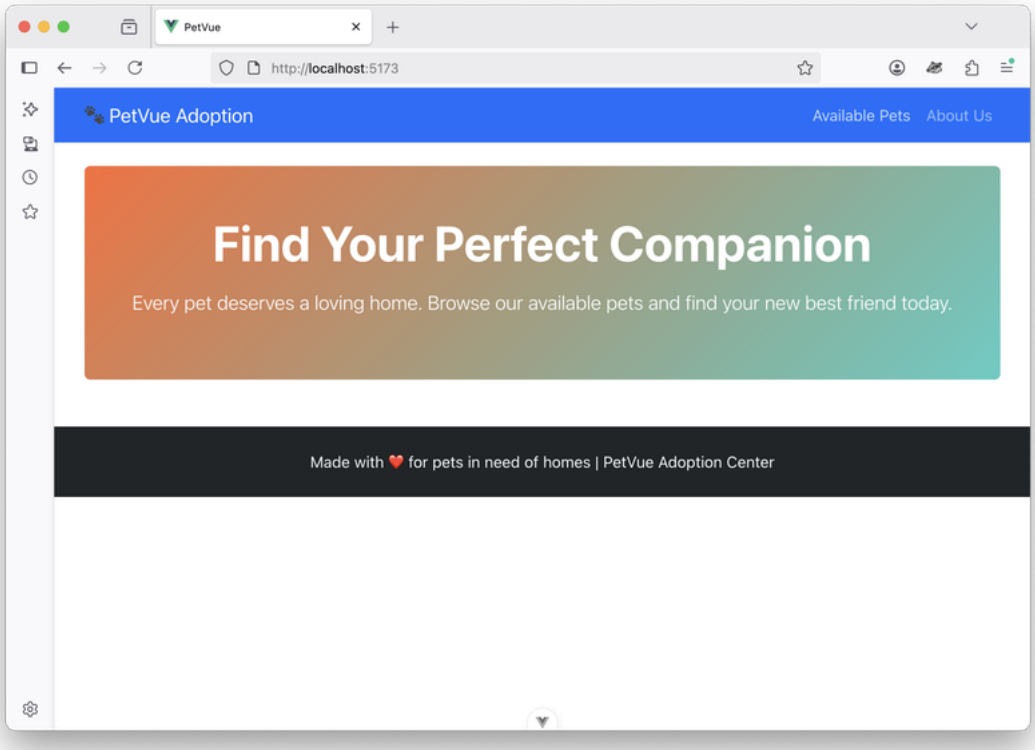
Both the Options API and the Composition API are fully supported and you can even mix them in the same project. For this course, we'll focus on the Composition API. It's the direction Vue is heading, and it gives you the most flexibility and power as your projects get bigger.

If you want to learn more, check out the official Vue docs on `API Styles (https://vuejs.org/guide/introduction.html#api-styles)` and the `Composition API FAQ (https://vuejs.org/guide/extras/composition-api-faq.html)`.

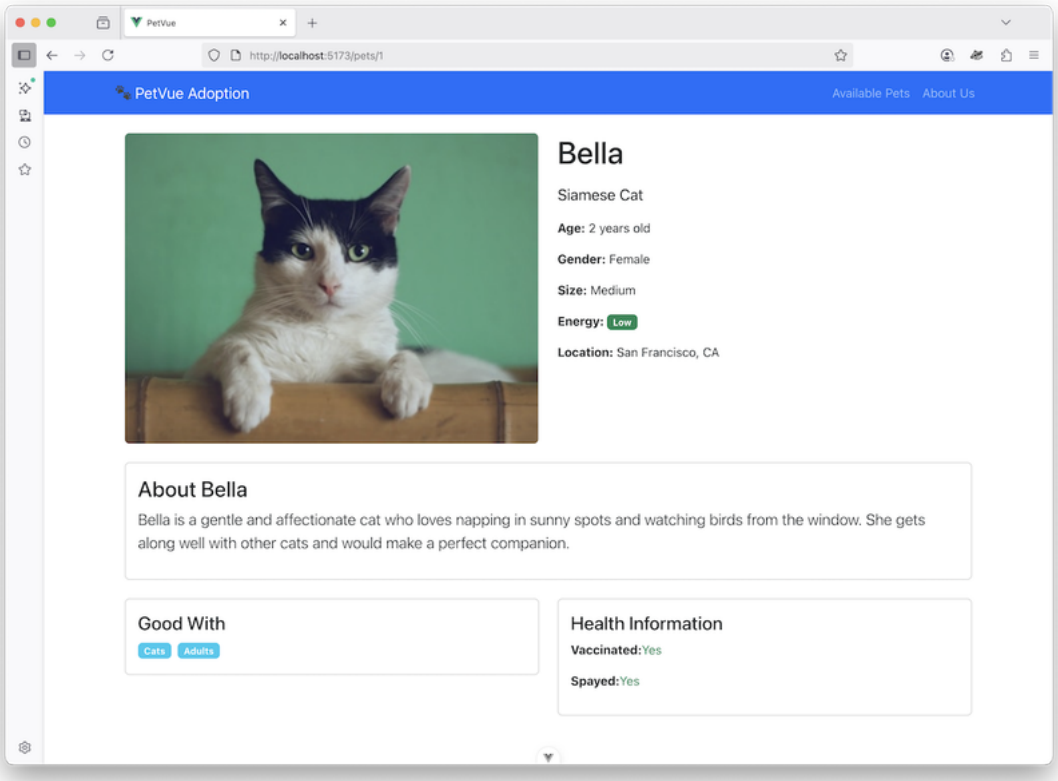
1.5. Course Project Introduction: PetVue

Throughout these lessons, you'll be using the latest Vue syntax and techniques to build an SPA (single-page application) for a fictitious pet adoption service. Most of the course will focus on building a simple web site (called **PetVue**) that shows a list of animals available for adoption. The user can filter the list in a number of ways, and click on individual pets to see more information about them. The main views of PetVue are shown below:

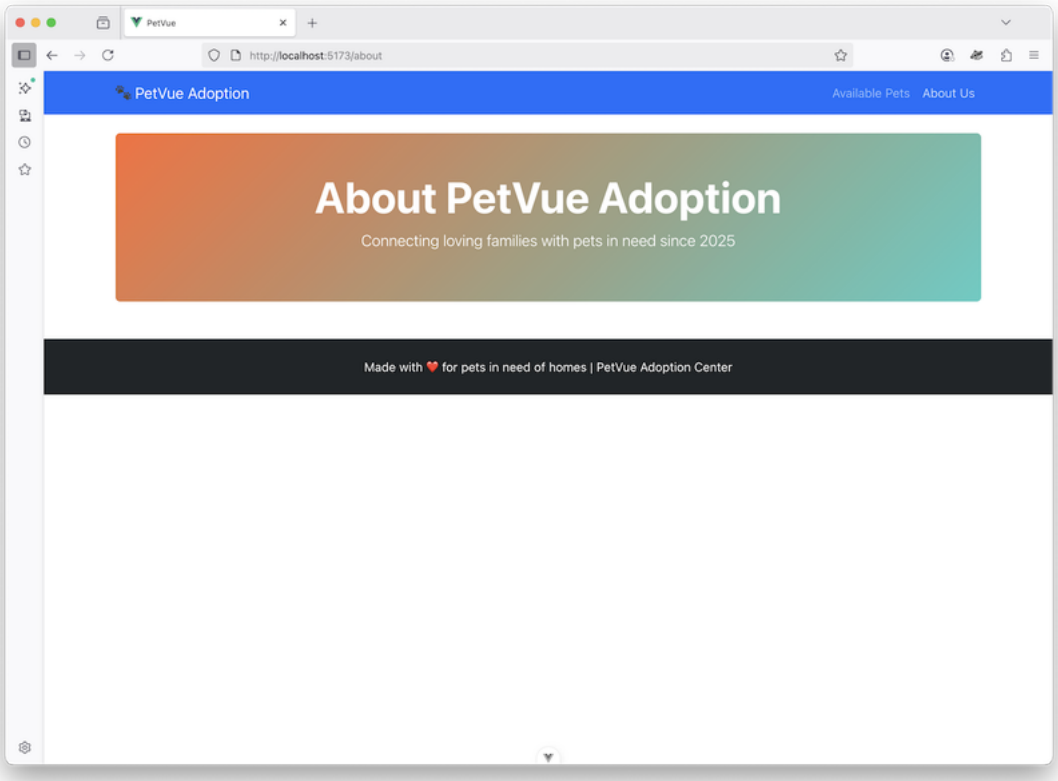
❖ 1.5.1. Available Pets



❖ 1.5.2. Pet Details



❖ 1.5.3. About PetVue



Conclusion

In this lesson, you have explored the fundamentals of Vue.js, understanding its role as a powerful JavaScript framework for building interactive web applications. You've seen how Vue simplifies the development process with its reactivity and easy DOM updates, making it an excellent choice for both small and large-scale projects. You've been introduced to the basic setup of a Vue app and examined the differences between the Composition API and Options API, giving you flexibility in organizing your code. Furthermore, you've previewed the PetVue project, where you'll be applying your Vue knowledge throughout the course.

LESSON 2
Setting Up Vue with Vite

Topics Covered

☑ What single-page applications are and how Vue + Vite support them.

☑ Creating a new Vue project with the Vite scaffolder.

Introduction

In this lesson, we will guide you through setting up your development environment with Vite (pronounced "veet"), a modern build tool that streamlines the process of creating Vue applications. You'll learn how Vite accelerates project setup with instant scaffolding and a fast development server. We'll show you how to adapt the default Vue project into the foundation for the PetVue app, which you'll continue to build throughout the course. By the end of this lesson, you'll understand how Vite supports efficient development of Single-Page Applications (SPAs) with Vue and have a solid base to expand on in future lessons.

EVALUATION COPY: Not to be used in class.

2.1. SPAs with Vue and Vite

Single-Page Applications (SPAs) transform the traditional web browsing experience by loading a single HTML page once, and dynamically updating its content using JavaScript as users interact with the app. Unlike traditional websites, which reload the entire page for each action, SPAs only update the relevant components. This approach results in faster interactions, smoother transitions, and a user experience similar to native apps.

SPAs excel in applications that involve significant interactivity and complex state management, such as social media platforms, dashboards, and real-time data apps.

EVALUATION COPY: Not to be used in class.

❖ 2.1.1. Why Use Vue for SPAs?

Vue.js excels at handling the complexities of SPAs. It offers a straightforward way to build interactive, reactive interfaces that respond instantly to user actions. By managing state effectively and updating content dynamically, Vue simplifies tasks that are challenging in traditional web development.

In an SPA built with Vue:

1. Your page doesn't reload completely during navigation; instead, Vue manages the content seamlessly.
2. Vue's reactive data binding automatically updates the UI, making the interface feel dynamic and responsive.
3. Complex features like real-time updates, dashboards, or interactive forms become intuitive to build and maintain

❖ 2.1.2. The Role of Vite in Building SPAs

While Vue manages the frontend interface logic, you still need an efficient way to structure your project, handle dependencies, and optimize your app for deployment. This is where **Vite** comes into play.

Vite is a modern build tool specifically designed to support frontend frameworks like Vue. It provides:

1. **Rapid Development Server:** Vite offers near-instant startup and hot module replacement, significantly speeding up your development cycle.
2. **Efficient Build Process:** When you're ready for production, Vite bundles and optimizes your app for fast load times, enhancing the user experience.
3. **Seamless Integration:** Vite integrates effortlessly with Vue, simplifying project setup and configuration.

Together, Vue and Vite streamline the entire SPA development process, from writing and testing your code during development, to efficiently deploying your finished application.

Vite Beyond Vue
Vite is not limited to building Vue applications. It can work with many other frameworks (React, Svelte, etc.) and even vanilla JavaScript. You can read more about it at `https://vite.dev/`.

In the next exercise, we will walk you through setting up the PetVue project with Vue and Vite.

Exercise 1: Scaffolding a Vue Project with Vite, Part 1

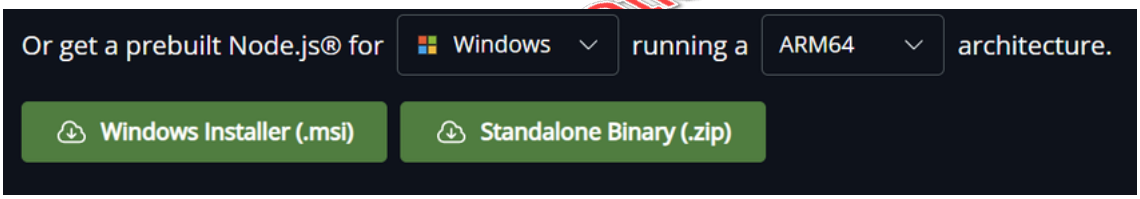
25 to 35 minutes

In this exercise, you will use Vite to set up the Vue application for **PetVue**.

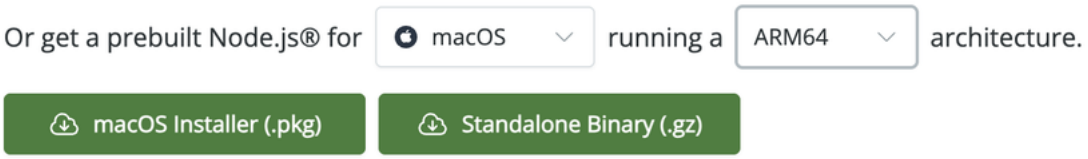
❖ E1.1. Prerequisites

Before you can use Vue or Vite, you need to make sure that you have downloaded Node.js and npm (Node Package Manager).

You can download them both in one package from the official Node.js website: `https://nodejs.org/en/download` . If you're using Windows, look for the button to download the Windows Installer (.msi). The architecture select input will be automatically set to match your computer.

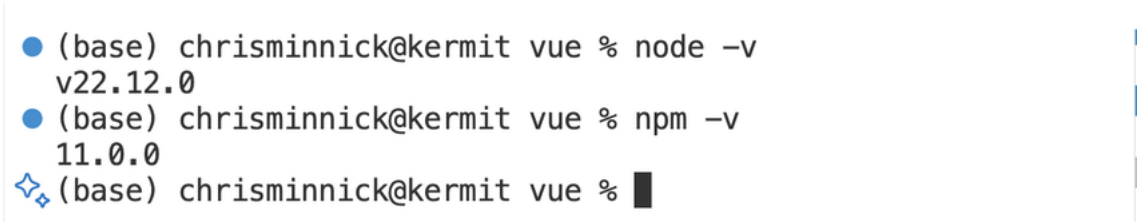


On MacOS, look for the macOS Installer (.pkg).



After the installer has finished downloading, run it. If the installer asks any questions as it's installing, just accept the default options.

Once you have gone through the installer, you can check your Node.js and npm versions by opening the integrated terminal in Visual Studio Code and running the following commands:



Note that the versions that you see may be different, but you should at least have Node version 20 and npm version 10.

❖ E1.2. Scaffolding a Vue Project with Vite

To create the project, you first need to open the terminal at the folder where you want to create the PetVue project. You should do this in your Vue class files folder.

Open in Integrated Terminal

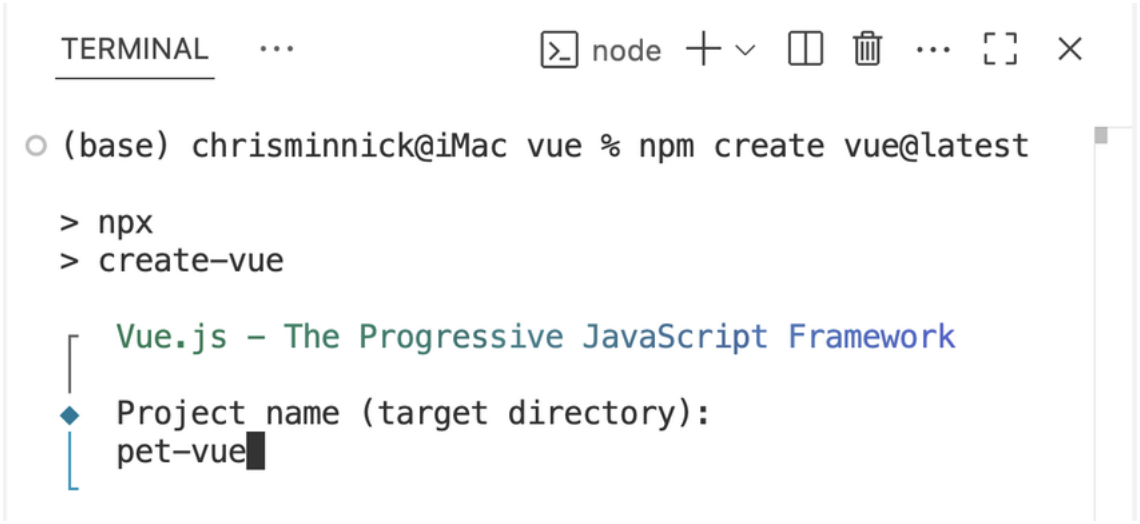
As a reminder, in Visual Studio Code, you can easily open the terminal at any particular folder location by right-clicking on that folder and selecting **Open in Integrated Terminal**:

Follow these steps in the terminal:

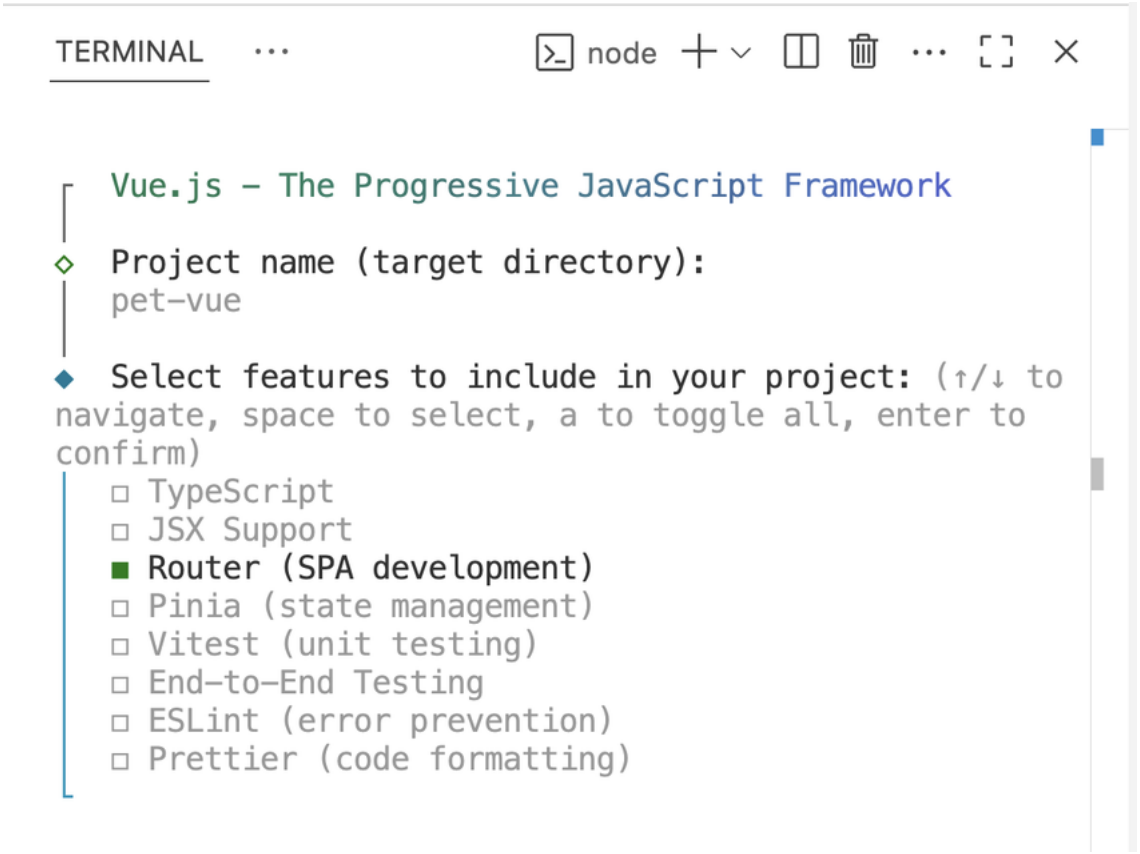
1. Run the create command:

```
npm create vue@latest
```

2. This will prompt you for the name of your project. Type `pet-vue`

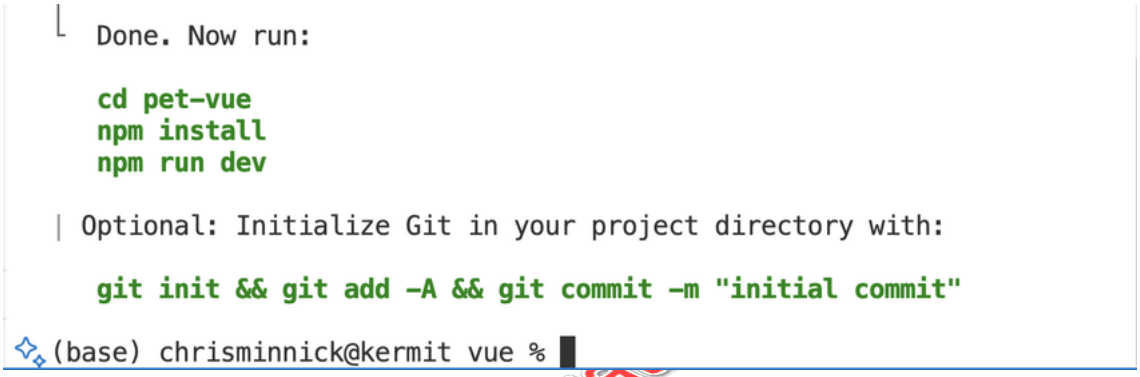


3. You will then be prompted to add additional features to your project. You should add router because we will use this later in the course. Use the down arrow on your keyboard to scroll down to Router, then use the space bar to select it. You don't need to add anything else.



4. Next, you may be asked whether you want to install experimental features. Just press **Enter** to skip installing those
5. Finally, you'll be asked whether you want to skip installing the example code. Choose **No** and press **Enter**.

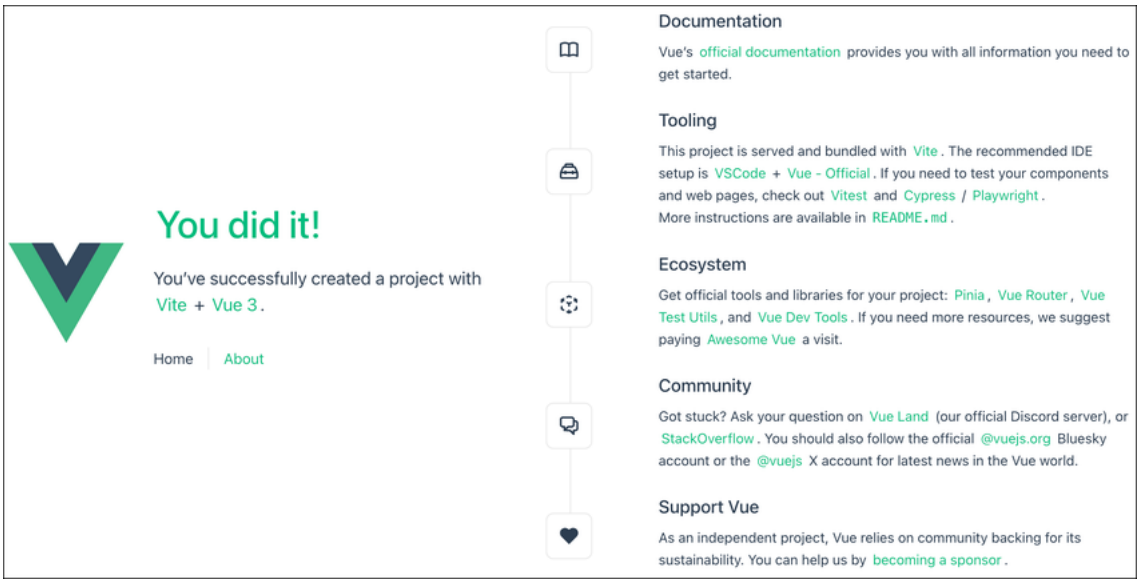
At the end, your terminal should look something like this:



To view the generated site, you can run the suggested commands:

1. `cd pet-vue`
2. `npm install`
3. `npm run dev`

You can then navigate to `http://localhost:5173/` in your browser to view the page.



Starting and Stopping the Dev Server

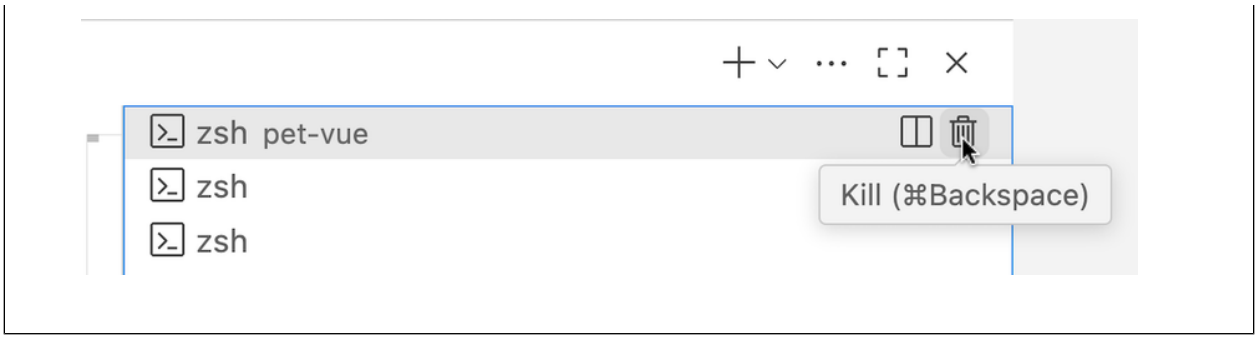
You will always start the dev server by running the **npm run dev** command at the level of the `pet-vue` folder in the terminal. In Visual Studio Code, you can open a terminal at the level of any folder by right clicking on that folder in the Explorer sidebar and selecting **Open in Integrated Terminal**.

Make sure that when you are done with the dev server that you stop it. You can easily run into issues by forgetting to stop the dev server and then trying to start it again when it is already going.

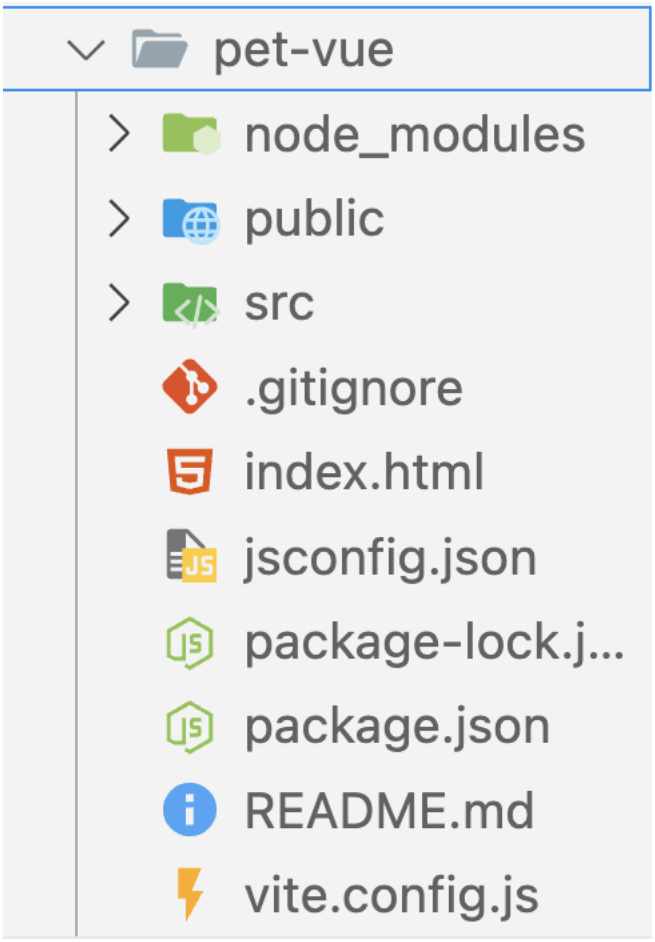
To stop the dev server, press **Ctrl** + **C** in the terminal where the dev server is running. This is what that looks like:

```
  VITE v7.1.2  ready in 1891 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: use --host to expose
  ➜  Vue DevTools: Open http://localhost:5173/__devtools__/ as a separate window
  ➜  Vue DevTools: Press Option(⌥)+Shift(⇧)+D in App to toggle the Vue DevTools
  ➜  press h + enter to show help
^C
(base) chrisminnick@kermit pet-vue %
```

You can also shut down the processes running in a terminal (including the dev server) by killing that terminal:



You should also see the generated `pet-vue` folder in your file structure:



In the next exercise, we will go over the different parts that have been generated within this folder and pare it down to provide you with a base for building the PetVue project.

Solution Files

To help you stay on track, we will provide you with solution files throughout the course. You can compare your own files directly with these solutions and you will even be able to run these apps yourself to test out the behavior in the browser.

For this exercise, you can see the solution files at `settingupvuevite/Solutions/pet-vue-part-1/`.

Note that for generated files (all of the files in this exercise), differences in the versions of Node, npm, Vite, and Vue may create slight differences between these solution files and the ones that you created.

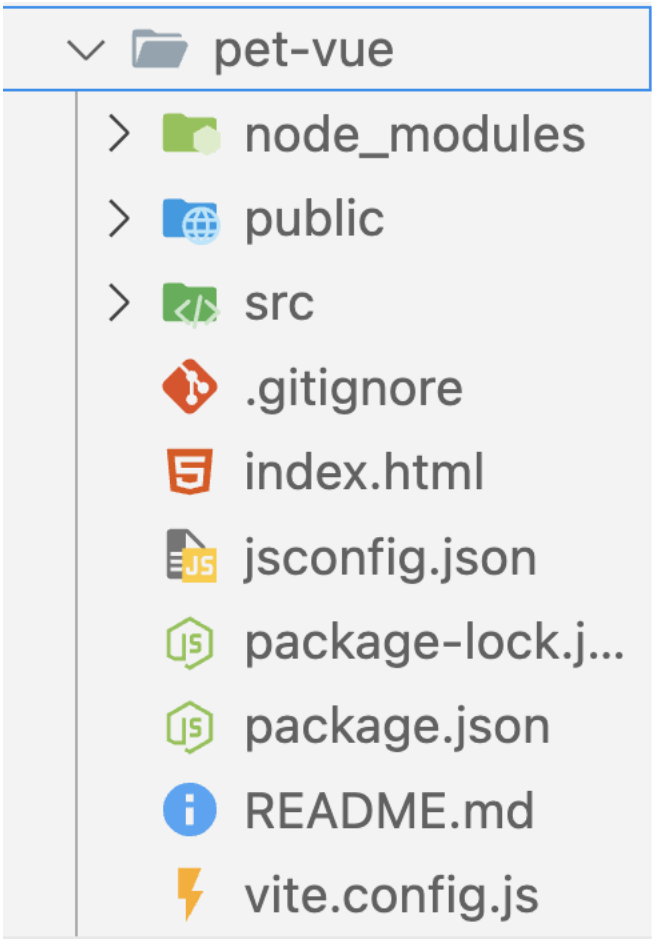
You can view this app in the browser the same way that you viewed your own:

1. First, make sure that the dev server is stopped as described in the **Starting and Stopping the Dev Server** callout.
2. Open a terminal at `pet-vue-part-1` by right-clicking on `pet-vue-part-1` and selecting **Open in Integrated Terminal**.
3. Install the dependencies by running **npm install**. You will only need to do this the first time you run the app.
4. Start the the app by running **npm run dev**.
5. You can now view the **pet-vue-part-1** app in the browser at `http://localhost:5173/`.

Exercise 2: Scaffolding a Vue Project with Vite, Part 2

25 to 30 minutes

In the previous exercise, you scaffolded a default Vue project with Vite, generating the following structure in the `pet-vue` folder:



In this exercise, will break down what each part of this structure is, and then we will guide you through paring it down so that you have a base for your PetVue project. See the **Clean Up Steps** throughout the exercise for guidance on what you need to do to modify the structure for PetVue.

❖ E2.1. Config and Editor Support Features (`.vscode/`, `jsconfig.json` and `vite.config.js`)

The `.vscode/` folder (if present), `jsconfig.json`, and `vite.config.js` files are included to help configure your development environment for a smoother coding experience. The `.vscode/` folder may contain settings specific to Visual Studio Code, such as recommended extensions or workspace preferences, to ensure consistency across different setups.

The `jsconfig.json` file tells your editor how to treat your project as a JavaScript project. It works with features like code `IntelliSense (https://code.visualstudio.com/docs/editing/intellisense)`, which provides smart code suggestions and navigation. You can use `jsconfig.json` to specify which files to include or exclude and to set language options for your project.

The `vite.config.js` file is used by Vite to control how your project is built and served during development. It allows you to set up build options, define aliases for import paths, and configure plugins or other advanced features. While `jsconfig.json` and `.vscode/` focus on editor support and code intelligence, `vite.config.js` controls how your application runs and builds.

You will rarely need to modify `.vscode/`, `jsconfig.json`, or `vite.config.js` for any reason. In this course, you will not need to touch them at all. They are provided to streamline your coding workflow and project setup.

Clean Up Step: Set up Editor Support for Vue in Visual Studio Code

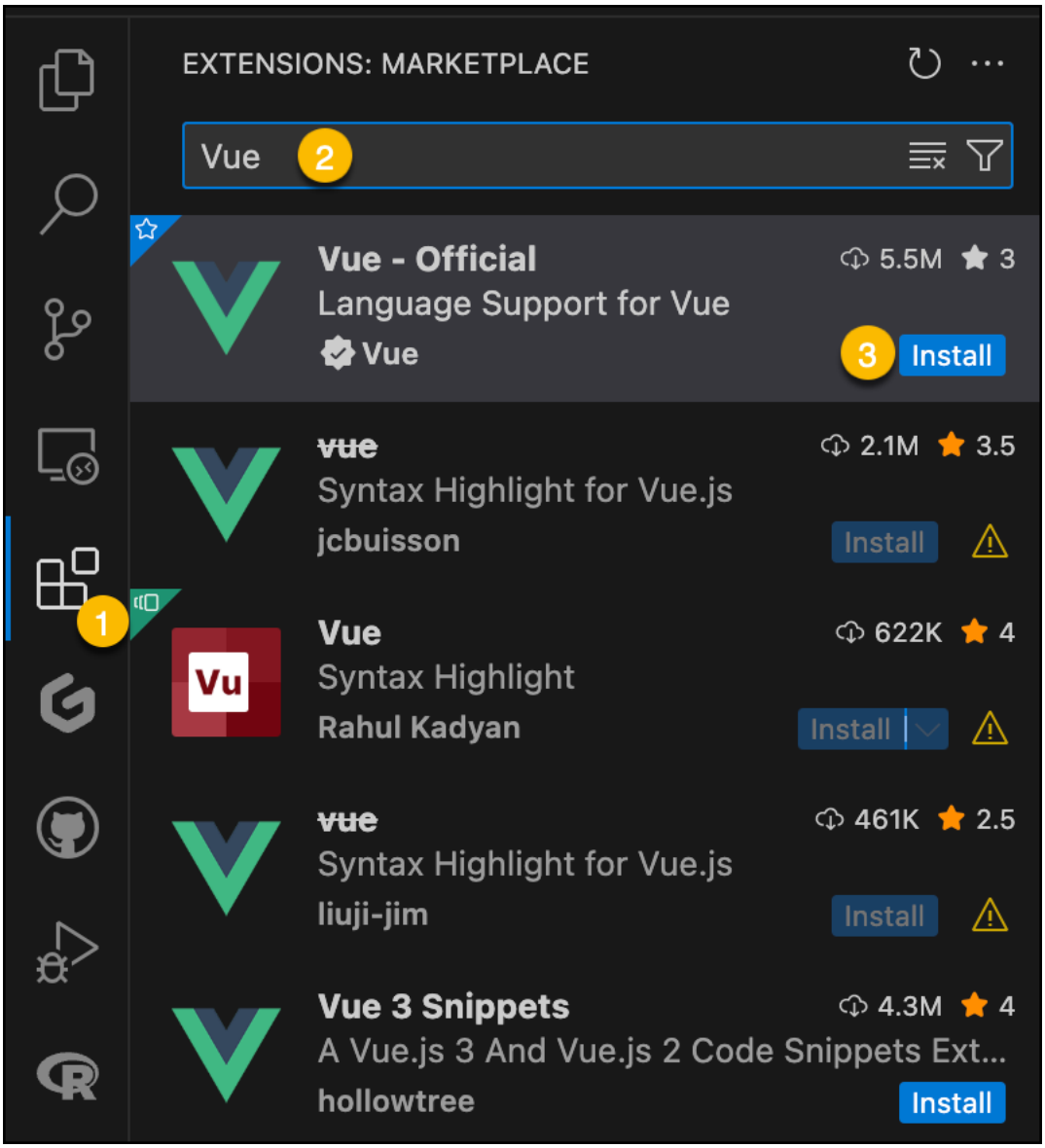
Vue has `an official Visual Studio Code extension (https://marketplace.visualstudio.com/items?itemName=Vue.volar)` that provides additional features to make writing Vue much easier. For example:

- **Syntax highlighting** for `.vue` files, so your code is easier to read and understand.
- **Code navigation** so you can make use of Visual Studio Code's `Go to Definition (https://code.visualstudio.com/docs/editing/editingevolved#_go-to-definition)` and `Peek (https://code.visualstudio.com/docs/editing/editingevolved#_peek)` features among others.
- **Formatting support** to keep your code style consistent.

To install it:

1. Go to the Extensions sidebar in Visual Studio Code.
2. Search for "Vue".

3. And click **Install** on the **Vue - Official** extension (it should be one of the first options if not the very first).



You may need to restart Visual Studio Code in order for it take effect.

❖ E2.2. Dependencies (`node_modules/`, `package.json`, and `package-lock.json`)

The `node_modules/` folder, along with the `package.json` and `package-lock.json` files, play a key role in managing your project's dependencies. These files and the folder work together to keep track of and organize the external code your project relies on. Dependencies are libraries or modules developed by others that your project uses to add features or functionality without having to write everything from scratch. When you ran `npm install`, all of the necessary dependencies for working with Vue and Vite were installed. You will need to install one more dependency, the Bootstrap CSS framework.

You may later choose to install additional dependencies. Every time you install a dependency, it is added to the `node_modules/` folder, listed in `package.json`, and its exact version is recorded in `package-lock.json`.

Clean Up Step: Installing Dependencies

Open a terminal in VSCode (if one isn't already open) and make sure the current directory is `/pet-vue`.

Enter `npm install bootstrap@5` in the terminal (make sure the current directory is `pet-vue`) to install version 5 of Bootstrap.

What is Bootstrap?

Bootstrap (`https://getbootstrap.com`) is a widely used frontend toolkit for building web user interfaces. It provides a responsive grid system, prebuilt components (buttons, forms, navbars, modals, cards), and a large set of utility classes that help you structure layouts, keep styles consistent, and ship features quickly.

Beyond styling, Bootstrap includes JavaScript plugins that add common interface behaviors such as dropdowns, tooltips, carousels, and modals. It smooths out cross-browser differences, follows accessible patterns, and supports theming through CSS variables so you can adapt the look to your brand without starting from scratch.

In a Vue application, you typically apply Bootstrap classes directly in your templates to handle layout and spacing, use its components for standard UI patterns, and opt into its JavaScript where needed. The result is a coherent, responsive interface that's faster to prototype and maintain.

❖ E2.3. Application Content, Styles, and Functionality (`src/` and `index.html`)

The main content, styles, and functionality of your project are organized within the `src/` folder. The `src/` folder holds the core application code, including components, views, and assets. This is where you will spend the most time editing and creating files.

Entry Point (`index.html`, `src/main.js`, and `src/App.vue`)

The entry point of your Vue Vite application consists of three key files: `index.html`, `src/main.js`, and `src/App.vue`. While the `index.html` file is located at the root of your project, not inside the `src/` folder, we are still discussing it here because it has the initial HTML content for your application, including the root element where Vue mounts your app.

The **main.js** file, found in the `src/` folder, serves as the starting point for your JavaScript code. It initializes the Vue application and connects it to the root element defined in `index.html`. App.vue, also in the `src/` folder, acts as the root Vue component that organizes the layout and structure of your application. Together, these files work to load and render your Vue app in the browser.

If you open these files, you may recognize some of the logic from `OverviewIntroductionVuejs/Demos/counter-vue.html`.

Exercise Code 2.1: index.html

```
1.   <!DOCTYPE html>
2.   <html lang="">
3.     <head>
4.       <meta charset="UTF-8">
5.       <link rel="icon" href="/favicon.ico">
6.       <meta name="viewport" content="width=device-width, initial-scale=1.0">
7.       <title>Vite App</title>
8.     </head>
9.     <body>
10.      <div id="app"></div>
11.      <script type="module" src="/src/main.js"></script>
12.    </body>
13.  </html>
```

In `counter-vue.html`, we also had a `div` with id "app".

Exercise Code 2.2: src/main.js

```
1.   import './assets/main.css'
2.
3.   import { createApp } from 'vue'
4.   import App from './App.vue'
5.   import router from './router'
6.
7.   const app = createApp(App)
8.
9.   app.use(router)
10.
11.  app.mount('#app')
```


In `counter-vue.html`, we also used the `createApp()` method to mount our app onto the `#app` element. Previously, we defined the full functionality of our app within the `createApp()` method, but since we are using `Single-File Components (SFCs) (https://vuejs.org/guide/scaling-`

up/sfc.html), the actual functionality of the app is defined within that file rather than in the method call itself.

Exercise Code 2.3: SettingUpVueVite/Solutions/pet-vue-part-1/src/App.vue

```
1.   <script setup>
2.   import { RouterLink, RouterView } from 'vue-router'
3.   import HelloWorld from './components/HelloWorld.vue'
4.   </script>
5.
6.   <template>
7.     <header>
8.       <img alt="Vue logo" class="logo" src="@/assets/logo.svg" width="125"
              height="125" />
9.
10.      <div class="wrapper">
11.        <HelloWorld msg="You did it!" />
12.
13.        <nav>
14.          <RouterLink to="/">Home</RouterLink>
15.          <RouterLink to="/about">About</RouterLink>
16.        </nav>
17.      </div>
18.    </header>
19.
20.    <RouterView />
21.  </template>
22.
23.  <style scoped>
     -------Lines 24 through 84 Omitted-------
85.  </style>
```


In App.vue, you can recognize the script, template, style separation that we discussed in **Introduction to Vue Components**.

Clean Up Step: Change the title in index.html

1. Open index.html.
2. Change the title from "Vite App" to "PetVue".

That's all you'll need to change in index.html!

Clean Up Step: Set up App.vue

1. Open `App.vue`
2. Copy and paste the following code into `App.vue` replacing the existing content:

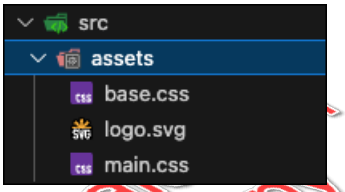
Exercise Code 2.4: src/App.vue

```
<template>
  <div id="app">
    <!-- Header -->
    <nav class="navbar navbar-expand-lg navbar-dark bg-primary">
      <div class="container">
        <router-link to="/" class="navbar-brand">
          <span>PetVue Adoption</span>
        </router-link>
        <div class="navbar-nav ms-auto">
          <router-link to="/" class="nav-link">Available Pets</router-link>
          <router-link to="/about" class="nav-link">About Us</router-link>
        </div>
      </div>
    </nav>

    <!-- Main Content -->
    <main class="container mt-4">
      <router-view />
    </main>

    <!-- Footer -->
    <footer class="bg-dark text-light text-center py-4 mt-5">
      <div class="container">
        <p class="mb-0">
          Made with &#10084; for pets in need of homes | PetVue Adoption Center
        </p>
      </div>
    </footer>
  </div>
</template>
```


Here we are simply refactoring the base `App.vue` to use some Bootstrap styles and to update it from being a Vue welcome page to being the home page of the PetVue site.

Assets (`src/assets/`)

The assets folder is where you store static files like stylesheets, images, and fonts.



The auto-generated `assets/` folder contains some basic styles for the template (`base.css` and `main.css`) and a Vue logo image (`logo.svg`).

Clean Up Step: Remove unnecessary files

1. Delete `logo.svg`. The only files that should be left in the assets folder should be `base.css` and `main.css`. If there are any other files in assets, you can delete them.

2. Open `main.css` and replace its contents with the code below:

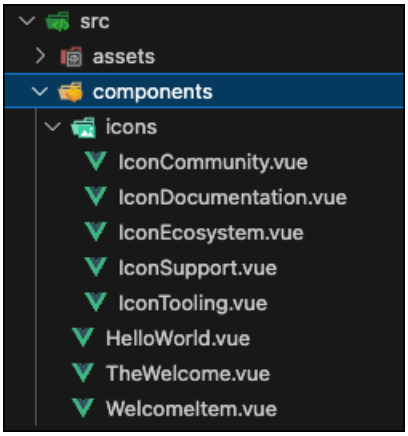
Exercise Code 2.5: src/assets/main.css

```
/* Import base styles and Bootstrap for layout and components */
@import './base.css';
@import 'bootstrap/dist/css/bootstrap.min.css';

/* Define custom color variables for Bootstrap theme */
:root {
  --bs-primary: #ff6b35;
  --bs-secondary: #4ecdc4;
  --bs-success: #28a745;
}

/* Make the navbar brand text bold */
.navbar-brand {
  font-weight: bold;
}

/* Style for pet images in cards: fixed height and cover fit */
.pet-image {
  height: 200px;
  object-fit: cover;
}

/* Style for pet images in detail view: larger fixed height and cover fit */
.pet-detail-image {
  height: 400px;
  object-fit: cover;
}

/* Homepage hero section with gradient background and white text */
.hero-section {
  background: linear-gradient(135deg, var(--bs-primary), var(--bs-secondary));
  color: white;
}
```


This CSS code imports the Bootstrap CSS and creates some styles that will be used as you build the PetVue application.

Components (`src/components/`)

The `components` folder contains reusable Single File Components (SFCs) used throughout your application.



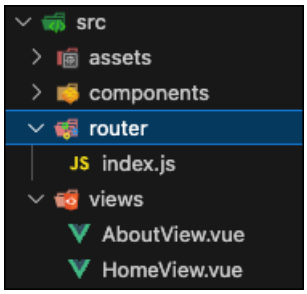
The auto-generated `components/` folder contains several components that show the content on the home page. The icon components are used to render different svg images and `HelloWorld.vue`, `TheWelcome.vue`, and `WelcomeItem.vue` show the main content.

Clean Up Step: Remove unnecessary files

Delete all files and the `icons/` folder within the `components/` folder. You will be starting from scratch creating your own components for the PetVue project.

Routes and Views (`src/router/` and `src/views/`)

`src/router/` contains the Vue Router configuration, which defines how different URLs map to pages in your app. `src/views/` contains page-level SFCs, where each file represents a different page. The routes in the router point to these views.



Clean Up Step: Update HomeView and AboutView

1. Open `AboutView.vue`.
2. Replace the entire content with the below code:

Exercise Code 2.6: src/views/AboutView.vue

```
<template>
  <div class="about-page">
    <!-- Hero Section -->
    <div class="hero-section text-center py-5 mb-5 rounded">
      <h1 class="display-4 fw-bold">About PetVue Adoption</h1>
      <p class="lead">
        Connecting loving families with pets in need since 2025
      </p>
    </div>
  </div>
</template>
```

Note that Vue component files doesn't need to have a `script`, `template`, and `style` block. Having just the `template` block is just fine for something like this About page where there isn't any functionality or CSS.

3. Open `HomeView.vue`.
4. You can replace the entire content with the below code:

Exercise Code 2.7: src/views/HomeView.vue

```
<template>
  <div class="pet-list-page">
    <!-- Hero Section -->
    <div class="hero-section text-center py-5 mb-4 rounded">
      <h1 class="display-4 fw-bold mb-3">Find Your Perfect Companion</h1>
      <p class="lead">
        Every pet deserves a loving home. Browse our available pets and find
        your new best friend today.
      </p>
    </div>
  </div>
</template>
```

❖ E2.4. Source Control with Git (`.gitignore` and `README.md`)

The `.gitignore` and `README.md` files are related to source control with Git, which is a tool for tracking changes to your code over time. Although Git will not be used for source control in this Vue course, it is a very useful tool, especially in environments where multiple developers are working on the same group of files. Because this is so common, Vite auto creates these files when you initialize a project.

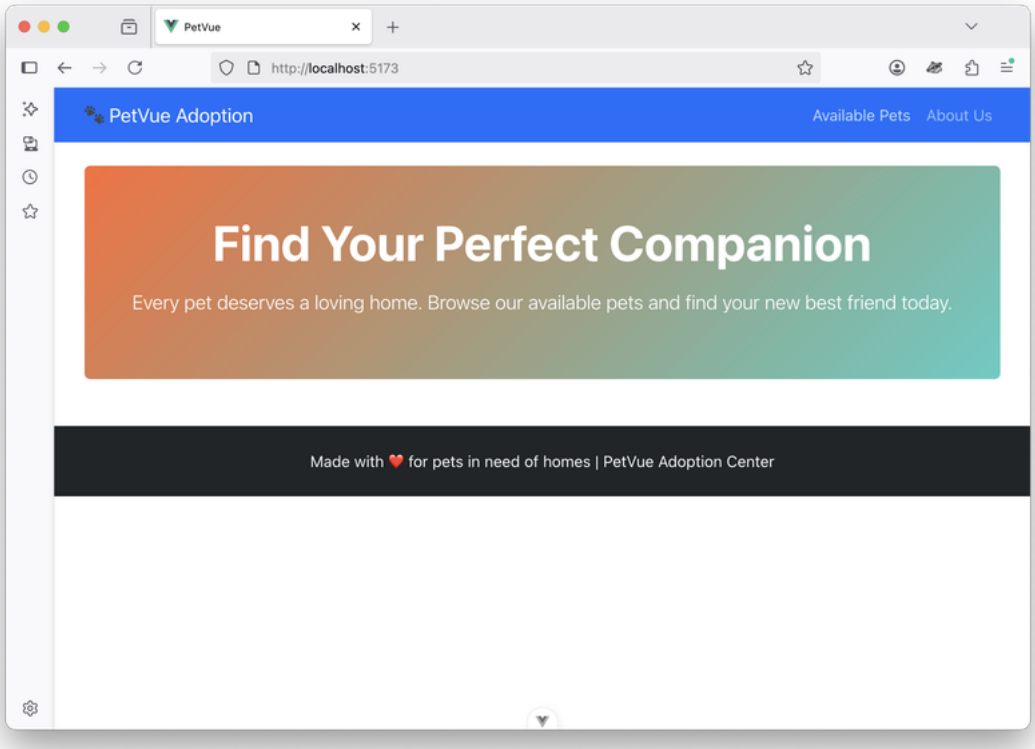
The `.gitignore` file tells Git which files or folders it should not track. This is helpful for keeping temporary files or sensitive information out of your source control history. The `README.md` file is a `markdown document (https://www.markdownguide.org/)` where you can provide information about your project, such as what it does, how to set it up, and any other important notes for developers or users.

❖ E2.5. All Set!

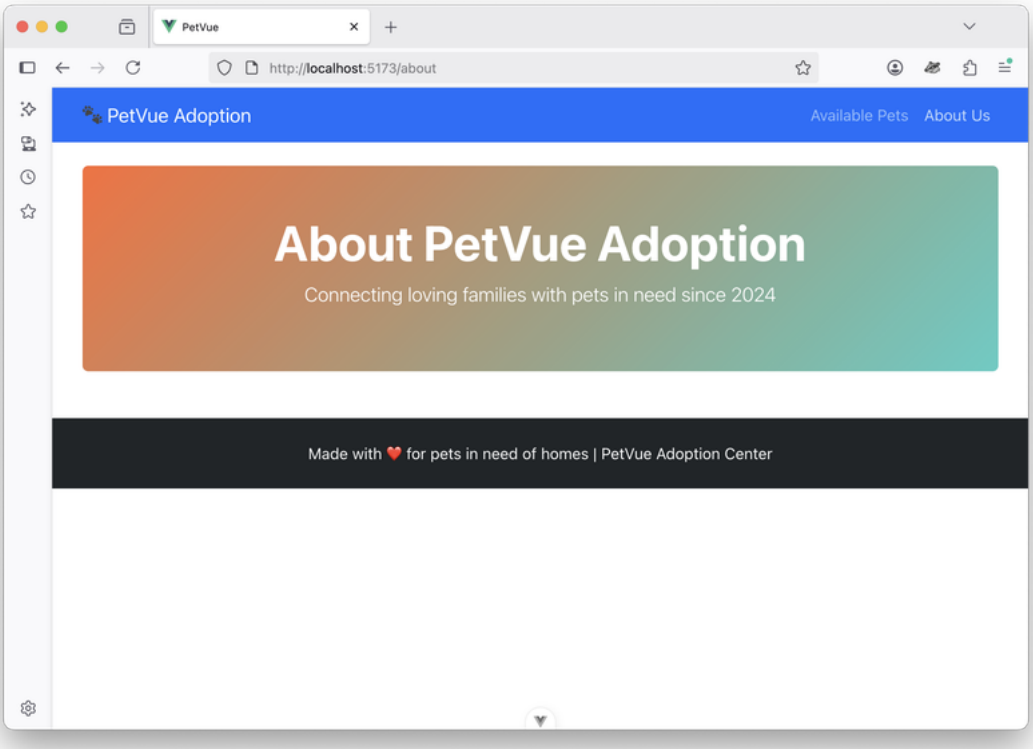
Now, if you run the development server you will see the base of your PetVue project that we will be building on throughout the rest of the course. As a reminder, you can run the development server by running **npm run dev** at the `pet-vue` folder level.

You can find our finished version of **Scaffolding a Vue Project with Vite, Part 2** at `settingupvue vite/Solutions/pet-vue-part-2/`.

Home Page



About Page



Conclusion

In this lesson, we guided you through the setup of a Vue application using Vite, a modern build tool that enhances development efficiency. You learned how to scaffold a Vue project with Vite, transforming it into the PetVue app. Throughout the exercises, we helped you understand the structure of a Vite-based Vue project, set up its development environment, and prepare it for the work you will do throughout the rest of the course. Now, you have a solid foundation in place to dive into the details of how to write Vue code.

LESSON 3
Basic Vue Reactivity

Topics Covered

☑ Reactive data flow and UI updates in Vue.

☑ Creating derived state with computed properties.

☑ Handling user events with v-on.

☑ Binding attributes, classes, and styles using v-bind.

☑ Two-way data binding for forms with v-model.

Introduction

In this lesson, we will dive into the world of Vue reactivity, giving you a comprehensive understanding of how data-driven interfaces are created using Vue's powerful reactivity system. You'll learn to harness Vue's reactive features like `ref` and `reactive` to create dynamic, automatically updating UIs. We will guide you through creating reactive states, computed properties, and event handling, empowering you to build responsive and interactive applications. Let us explore the foundational elements of reactivity that will elevate your Vue development skills.

EVALUATION COPY: Not to be used in class.

3.1. Understanding Reactive Data

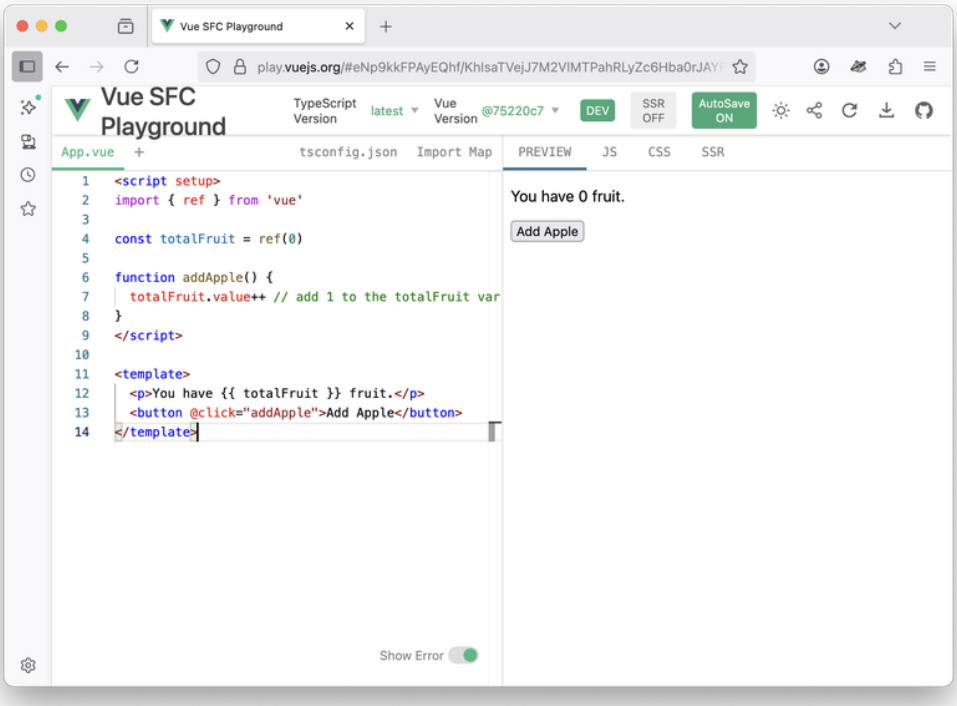
One of the core features that makes Vue powerful and user-friendly is its reactivity system. Reactivity in Vue means that when your application's data changes, the user interface updates automatically to reflect those changes. This allows developers to build interfaces that stay in sync with their data, without having to manually update the DOM.

EVALUATION COPY: Not to be used in class.

Vue SFC Playground

In this reading and throughout the course, we will show code snippets of Vue Single File Components. You can copy and paste these snippets into the `Vue SFC Playground` (`https://play.vuejs.org`). This is a quick way to preview simple Vue code.

Here is what previewing the **ref in the script** code snippet looks like in the Vue SFC Playground:



❖ 3.1.1. Creating Reactive State with `ref`

The primary way to create reactive state in the Vue Composition API is with the `ref` function. `ref` works for any value type, including numbers, strings, booleans, objects, and arrays.

To make a variable reactive with `ref`, first import the function from Vue and set a default value. For example:

ref example

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

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

Now that you have a reactive variable declared in your script, you can easily access it in the template via double curly braces:

ref in the template

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

const totalFruit = ref(0)
</script>

<template>
  <p>You have {{ totalFruit }} fruit.</p>
</template>
```


Accessing reactive variables within the script is slightly different from accessing them in the template. When you access them in the script, you must use the `.value` property:

ref in the script

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

const totalFruit = ref(0)

function addApple() {
  totalFruit.value++ // add 1 to the totalFruit variable
}
</script>

<template>
  <p>You have {{ totalFruit }} fruit.</p>
  <button @click="addApple">Add Apple</button>
</template>
```

Notice that instead of `totalFruit++` we have `totalFruit.value++`. The reason for this slightly confusing bit of syntax is that Vue needs to be able to track changes to the data to handle its reactivity. This is not built into JavaScript by default, so Vue wraps each `ref` variable in an object with a `value` property. It can then tell when the property is accessed or changed. You can `read more in the Vue documentation (https://vuejs.org/guide/essentials/reactivity-fundamentals.html#why-refs)`.

`@click` is Vue's version of `HTML's onclick attribute (https://developer.mozilla.org/en-US/docs/Web/HTML/Reference/Attributes#event_handler_attributes)`. We will go into more detail about this and other event listeners later in this lesson.

❖ 3.1.2. Working with Objects and Arrays

`ref` can work with any variable type including objects and arrays. Here's an example of an object:

ref object

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

const fruitCounts = ref({ apples: 0, bananas: 0 })

function addApple() {
  fruitCounts.value.apples++ // add 1 to the apples property
}
function addBanana() {
  fruitCounts.value.bananas++ // add 1 to the bananas property
}
</script>

<template>
  <p>You have {{ fruitCounts.apples }} apples and {{ fruitCounts.bananas }} bananas.</p>

  <button @click="addApple">Add Apple</button>
  <button @click="addBanana">Add Banana</button>
</template>
```


With objects, notice that we first need to access `.value` before we can access the properties on the object. Arrays are similar:

ref array

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

const fruits = ref([])

function addApple() {
  fruits.value.push("apple")
}
function addBanana() {
  fruits.value.push("banana")
}
</script>

<template>
  <p>{{ fruits }}</p>
  <button @click="addApple">Add Apple</button>
  <button @click="addBanana">Add Banana</button>
</template>
```


Instead of `fruits.push("apple")`, we need to have `fruits.value.push("apple")`.

Using `reactive` as an Alternative to `ref` for Objects and Arrays

Vue also provides a `reactive` function that can make an entire object or array reactive. reactive is nice because you do not need to worry about using the `.value` property. A variable declared with `reactive` can be accessed as is:

reactive example

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

const fruits = reactive(['apple', 'banana'])

function addFruit(fruit) {
  fruits.push(fruit)
}
</script>

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

However, `reactive` has a couple limitations:

1. It only works with objects and arrays, not with primitive values like numbers or strings.

2. If you replace the entire object, you lose the reactivity connection.

   With `ref`, you can reassign the value of the variable, but if you do this with `reactive`, it will break.

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

let fruits = ref(['apple', 'banana'])
fruits.value = ['pear', 'orange'] // valid

let fruits2 = reactive(['apple', 'banana'])
fruits2 = ['pear', 'orange'] // not valid - can create bugs
</script>
```

Similarly, object and array `destructuring` (`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring`) does not work with `reactive` variables either:

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

let fruits = ref(['apple', 'banana'])
const [a, b] = fruits.value // valid

let fruits2 = reactive(['apple', 'banana'])
const [c, d] = fruits2 // not valid - can create bugs
</script>
```

Because of these limitations, `ref` is generally preferred for dealing with reactive state.

3.2. Exploring Computed Properties

In the previous reading, we explored Vue's reactivity system and learned how `ref` and `reactive` (if we use it) allow us to create reactive state in our components. With these tools, Vue automatically updates the DOM whenever our data changes. However, as our applications grow, we often need to derive new values from our reactive state—values that depend on other data and should update automatically whenever their dependencies change. This is where computed properties come in.

❖ 3.2.1. What Are Computed Properties?

A computed property is a special kind of reactive value in Vue that is defined based on other reactive data. They are especially useful when you want to keep your templates simple and avoid duplicating logic. For instance, instead of placing a complex expression directly in your template, you can move it to a computed property

To create a computed property using the Composition API, you use the `computed` function from Vue. This function takes a getter function as its argument and returns a computed `ref`. Here's a simple example:

simple computed example

```
<script setup>
import { ref, computed } from 'vue'
const apples = ref(2)
const bananas = ref(3)

// A computed property that calculates the total number of fruit
const totalFruit = computed(() => apples.value + bananas.value)
</script>

<template>
<p>Apples: {{ apples }}</p>
<p>Bananas: {{ bananas }}</p>
<p>Total fruit: {{ totalFruit }}</p>
<button @click="apples++">Add Apple</button>
<button @click="bananas++">Add Banana</button>
</template>
```


Here, `totalFruit` is automatically updated whenever either `apples` or `bananas` changes.

The functions passed into `computed()` are often written as anonymous functions with the arrow function syntax (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#description) to use fewer lines, but they can be written as regular anonymous functions or as non-anonymous functions. The options below are equivalent to the above arrow syntax:

```
// using a regular anonymous function
const totalFruit = computed(function () {
  return apples.value + bananas.value
})
```

```
// using a non-anonymous function
function addApplesAndBananas() {
  return apples.value + bananas.value;
}
const totalFruit = computed(addApplesAndBananas)
```

❖ 3.2.2. Computed Properties vs. Methods

Both computed properties and methods can be used to derive values from your data, but there is an important difference. Computed properties are cached based on their dependencies. This means a computed property will only re-evaluate when one of its reactive dependencies changes. A method, on the other hand, will run every time the component re-renders, which happens whenever reactive data changes.

For example, consider the following:

method

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

const numbers = ref([1, 2, 3, 4, 5])
const count = ref(0)

function countEvens() {
  console.log("countEvens");
  return numbers.value.filter(n => n % 2 === 0).length
}
</script>

<template>
  <p>Even numbers: {{ countEvens() }}</p>
  <p>Count: {{ count }}</p>
  <button @click="count++">Increment Count</button>
</template>
```

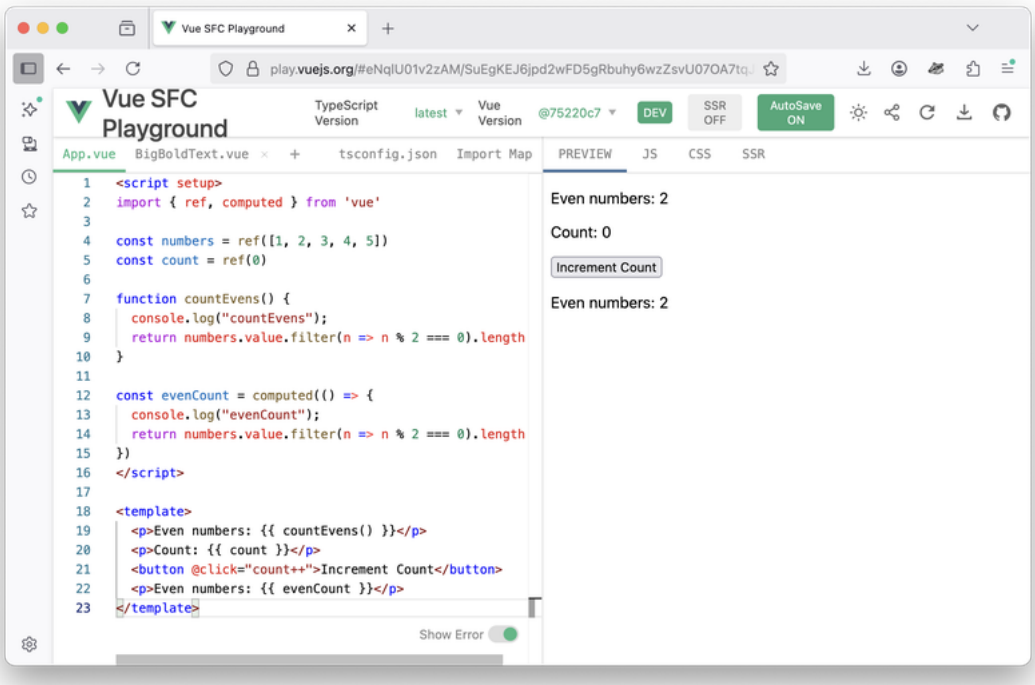

In the above code, `countEvens()` will run on every render even if `numbers` is not changed. So, `countEvens()` will run every time the **Increment Count** button is clicked to increment the `count` state, even though `numbers` (the reactive state that `countEvens` is based on) is not affected by `count`. If we start to do this kind of thing often with expensive calculations, it can negatively impact the runtime of our applications. Using `computed` solves this:

computed

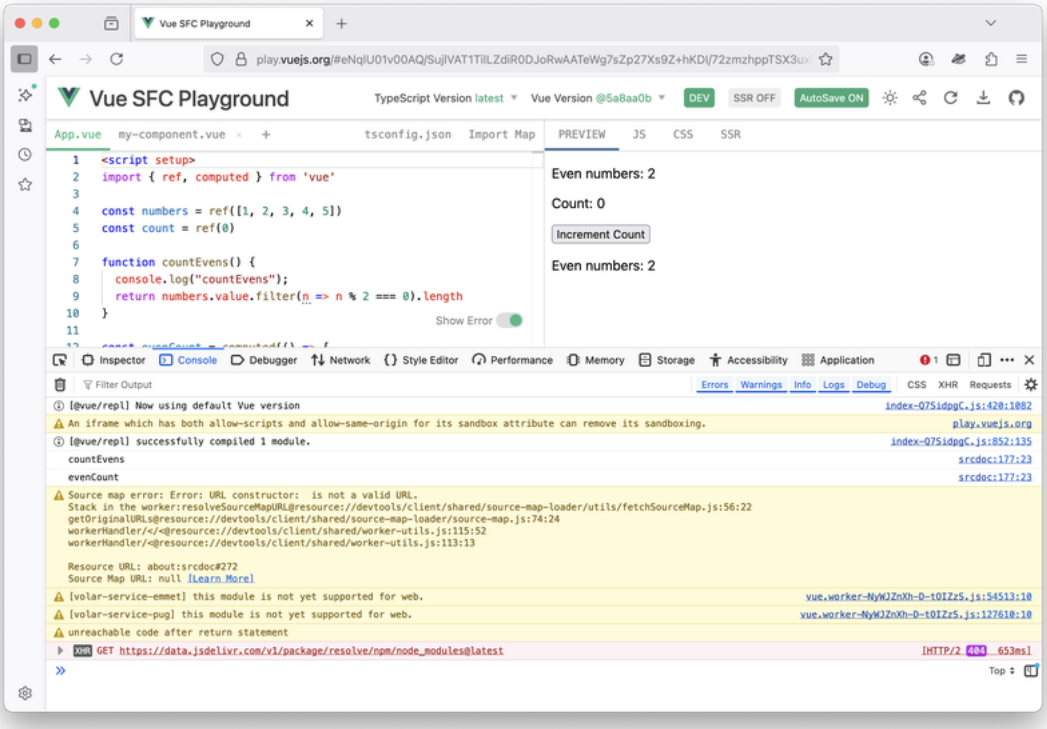
```
<script setup>
import { ref, computed } from 'vue'

const numbers = ref([1, 2, 3, 4, 5])
const count = ref(0)

function countEvens() {
  console.log("countEvens");
  return numbers.value.filter(n => n % 2 === 0).length
}

const evenCount = computed(() => {
  console.log("evenCount");
  return numbers.value.filter(n => n % 2 === 0).length
})
</script>

<template>
  <p>Even numbers: {{ countEvens() }}</p>
  <p>Count: {{ count }}</p>
  <button @click="count++">Increment Count</button>
  <p>Even numbers: {{ evenCount }}</p>
</template>
```


The computed property `evenCount` will only run on load and if `numbers` changes, but `countEvens()` will run every time any state changes. Let's examine this in the `Vue SFC Playground` (`https://play.vuejs.org`):

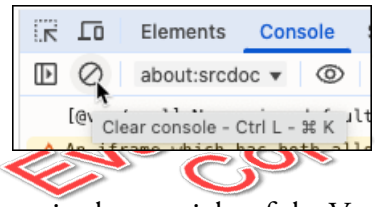
1. Go to the **Vue SFC Playground**: `https://play.vuejs.org`.
2. Copy and paste the above computed demo into **App.vue**:



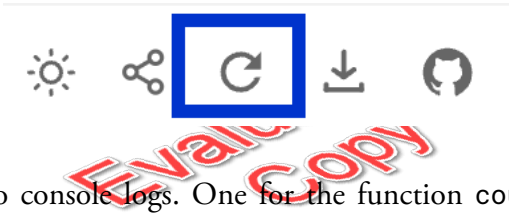
3. Open the **Dev Tool Console** (Right click anywhere on the page and select Inspect element and then go to **Console**):



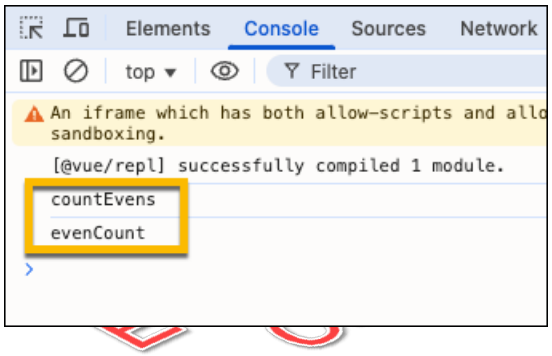
4. You'll see several warning messages and possibly even an error. These have to do with updates that may be required in the Vue SFC Playground, not in your code, and you can ignore them. Clear the **Console** by clicking on the Clear console icon in the top left of the Console:



5. Click the Reload page button in the top right of the Vue SFC Playground:

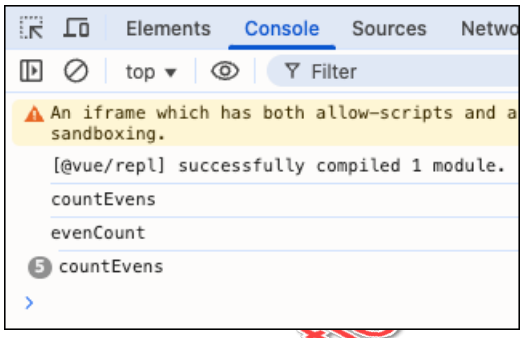


6. You will see the two console logs. One for the function `countEvens()` and one for the computed property `evenCount`:



These are both from the very first render on page load.

7. Try clicking the **Increment Count** button a few times. This will only keep logging `countEvens` because the computed property `evenCount` does not need to be run again:



❖ 3.2.3. Writable Computed Properties

By default, computed properties are read-only. However, you can create a writable computed property by providing both a getter and a setter. This is useful when you want to derive a value from multiple sources but also allow updates that affect the underlying data.

For example: suppose you want to let users edit a date as a string (e.g., "2025-06-02") in an input, but internally manage it as a `Date` object for calculations.

writable computed

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

// Store the date as a Date object
const selectedDate = ref(new Date())

const dateString = computed({
  get() {
    // Format the Date object as "YYYY-MM-DD"
    return selectedDate.value.toISOString().split('T')[0]
  },
  set(newValue) {
    // Parse the string and update the Date object
    if (newValue !== "") {
      const parsed = new Date(newValue)
      selectedDate.value = parsed
    }
  }
})
</script>

<template>
  <label>
    Select date:
    <input type="date" v-model="dateString" />
  </label>
  <p>Stored as: {{ selectedDate.toDateString() }}</p>
</template>
```


In the above demo:

- The `input` field is bound to `dateString` using `v-model` (we will cover this later in the lesson), which displays the date in the format necessary for the "date" `input`.
- When the user changes the date, the setter updates the underlying `selectedDate` object.
- The internal state always stays in sync with the formatted string.

This approach is helpful whenever you need to bridge user-friendly input and more complex internal data structures. Take some time to examine this demo in the Vue SFC Playground just like we did with the previous one!

❖ 3.2.4. Best Practices

When working with computed properties in Vue, following a few best practices will help you write code that is efficient, predictable, and easy to maintain:

1. **Use computed properties for derived values**: Whenever you need a value that depends on other reactive data and should update automatically, reach for a computed property. This keeps your templates clean and your logic centralized.
2. **Keep computed getters pure**: Computed getters should only compute and return a value. Avoid introducing side effects, asynchronous code, or DOM manipulation inside a computed getter. This ensures your computed properties remain predictable and easy to debug.
3. **Reserve writable computed properties for special cases**: Writable computed properties are powerful for two-way binding or when you need to synchronize a derived value with its sources. Use them thoughtfully, as most computed properties should remain read-only.
4. **Prefer methods for non-reactive or uncached logic**: If you need to perform an action or calculation that doesn't depend on reactive state, or if you don't need caching, use a method instead of a computed property.

By following these guidelines, you'll make the most of Vue's reactivity system and keep your components organized and efficient. As you continue building with Vue, leveraging computed properties thoughtfully will lead to more maintainable and robust code.

3.3. Events and v-on

Modern web applications need to respond to user actions, such as clicking a button, submitting a form, or pressing a key. JavaScript handles this with `the addEventListener() method (https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener)`. For example:

addEventListener

```
<!-- HTML -->
<button id="click-me">Click Me!</button>

// JavaScript
const clickMeBtn = document.getElementById("click-me");
clickMeBtn.addEventListener("click", () => alert('You clicked the button!'));
```

You may also see `inline event listeners` (`https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Scripting/Events#inline_event_handlers_%E2%80%94_dont_use_these`) right in the HTML. For instance:

inline event listeners

```
<button onclick="alert('You clicked the button')">Click Me!</button>
```

JavaScript inline event listeners are outdated and should not be used anymore: they can pose some security risks and are disabled on some servers, and they are also much more difficult to maintain in typical vanilla JavaScript applications where it good practice to keep JavaScript and HTML separate from each other.

However, this style of using inline event listeners is useful in a framework like Vue where the JavaScript and HTML are intentionally mixed within the same file. So, Vue provides a new, improved version of JavaScript inline event listeners: `v-on`.

❖ 3.3.1. Listening to Events with `v-on`

The `v-on` directive allows you to attach event listeners to elements in your template. The most common usage is for handling clicks, but you can listen for any standard DOM event. Here's the basic syntax:

v-on syntax

```
<button v-on:click="alert('You clicked the button!')">Click Me!</button>
```

v-on shorthand: @

Because `v-on` is used so often, Vue provides a shorthand (`@`), which you have already seen used in previous demos:

```
<button @click="alert('You clicked the button!')">Click Me!</button>
```

Going forward, we will continue to use `@` for `v-on`.

`v-on` can handle both inline JavaScript as the alert shows in the example above as well as references to functions you have written in the setup script:

```
<script setup>
function handleButtonClick() {
  alert('You clicked the button');
}
</script>

<template>
<button @click="handleButtonClick">Click Me!</button>
</template>
```


Here, we demoed the `"click"` event, but any JavaScript event (`https://w3c.github.io/uievents/#event-types-list`) can be used:

- `submit`
- `mousedown`
- `keyup`
- `input`
- `change`
- `focus`

Placing Event Listeners Appropriately

Make sure that you are attaching each event listener to the appropriate element. For example, `@submit` needs to be put on the `form` element, not the `button` element. Similarly, `@focus` only works on elements that can receive focus, such as inputs, buttons, and links. Placing events on the wrong element will prevent them from firing as expected.

❖ 3.3.2. Passing Arguments to Event Handlers

Event handler functions can also take arguments. To pass arguments, you can simply pass the arguments to the function as you would call it normally:

Passing Arguments

```
<script setup>
function handleButtonClick(msg) {
  alert(msg);
}
</script>

<template>
<button @click="handleButtonClick('You clicked the button')">
  Click Me!
</button>
</template>
```


❖ 3.3.3. Accessing the Event in Event Handlers

Sometimes you may need access to the original `event object (https://developer.mozilla.org/en-US/docs/Web/API/Event)` to either get additional information from the DOM (e.g., which element was clicked, the mouse position, which key was pressed, etc.) or control how the event behaves (e.g., `prevent default behavior (https://developer.mozilla.org/en-US/docs/Web/API/Event/preventDefault)`, `stop event propagation (https://developer.mozilla.org/en-US/docs/Web/API/Event/stopPropagation)`). You can get access to the event object either by using an arrow function or by using the special `$event` variable:

```
<script setup>
function handleButtonClick(event) {
  const buttonId = event.currentTarget.id;
  alert("You clicked the " + buttonId + " button!");
}
</script>

<template>
<!-- Button using arrow function pass event -->
<button id="arrow-function-button"
  @click="(event) => handleButtonClick(event)"
  Click Me!
</button>
<!-- Button using $event variable to pass event -->
<button id="$event-button"
  @click="handleButtonClick($event)">
  Click Me!
</button>
</template>
```


In the code above, clicking the first button will alert "You clicked the arrow-function-button button!" and clicking the second button will alert "You clicked the $event-button button!".

❖ 3.3.4. Event Modifiers

Some event actions are so common that Vue has created special modifiers to make these actions easier. Event modifiers are added to your event listeners as suffixes after a dot, directly in the template (e.g., `@click.stop`). Here are some of the most useful ones:

`.stop`

Use `.stop` to call `event.stopPropagation()` for you. This prevents the event from bubbling up to parent elements. You can read more about `stopPropagation()` in the `MDN documentation (https://developer.mozilla.org/en-US/docs/Web/API/Event/stopPropagation)`.

```
<button id="arrow-key-function-button" @click.stop="handleButtonClick"
  Click Me!
</button>
```

`.prevent`

Use `.prevent` to call `event.preventDefault()` for you, which prevents the default action associated with the event (e.g., submitting a form). You can read more about `preventDefault()` in the MDN documentation (`https://developer.mozilla.org/en-US/docs/Web/API/Event/preventDefault`).

```
<button id="arrow-key-function-button" @click.prevent="handleButtonClick"
  Click Me!
</button>
```

`.once`

Use `.once` to ensure that the event handler will only be triggered at most one time (e.g., only the first click will call the function).

```
<button id="arrow-key-function-button" @click.once="handleButtonClick"
  Click Me!
</button>
```


Modifiers can even be chained together if you need multiple effects:

```
<button id="arrow-key-function-button" @click.prevent.once="handleButtonClick"
  Click Me!
</button>
```

Here, `@click.prevent.once` will both call `event.preventDefault()` and make it so only the first click triggers the function call.

❖ 3.3.5. Key and Mouse Modifiers

Vue makes it easy to listen for specific keys or mouse buttons when handling events. By adding modifiers to your event listeners, you can trigger handlers only under certain conditions, such as when the user presses the **Enter** key or clicks the right mouse button.

Key Modifiers

When listening for keyboard events, you might want to respond only to certain keys. Vue allows you to add key modifiers directly to your event listener. For example, to trigger a method only when the **Enter** key is pressed:

```
<button @keyup.enter="submitForm" type="submit">Submit</button>
```

You can use any valid key name from `the KeyboardEvent.key property (https://developer.mozilla.org/en-US/docs/Web/API/UI_Events/Keyboard_event_key_values)` as a modifier, written in kebab-case. For example for `the ArrowDown key (https://developer.mozilla.org/en-US/docs/Web/API/UI_Events/Keyboard_event_key_values#navigation_keys)`:

```
<input @keyup.arrow-down="onArrowDown" />
```

If you would like to trigger the event on multiple keys, you can chain them. For instance, if we wanted to trigger a function on the press of any arrow key:

```
<input @keyup.arrow-down.arrow-up.arrow-right.arrow-left="onArrowKey" />
```


Vue also provides aliases for some common keys:

- `.delete` (covers both **Delete** and **Backspace**)
- `.down`, `.left.right`, and `.up` (equivalent to `.arrow-down`, `.arrow-left`, `.arrow-right`, and `.arrow-up`)
- `.space` (for the space bar since the key value is `" "`)

You can chain key modifiers with system modifier keys to listen for combinations like **Ctrl** + **Enter**:

```
<input @keyup.ctrl.enter="clearInput" />
```

System modifier keys include:

- `.ctrl`
- `.alt`

- `.shift`
- `.meta` (Command on Mac, Windows key on Windows)

Note that when you chain system modifier keys, the system modifier key must be pressed at the same time as the other key. When you chain regular keys, the event will be triggered if any one of the keys is pressed.

Mouse Button Modifiers

Vue also lets you listen for events from specific mouse buttons using the following modifiers:

- `.left`
- `.right`
- `.middle`

For example, to respond only to right-clicks:

```
<div @click.right="showContextMenu">Open Menu</div>
```


You also pair mouse buttons with system modifier keys:

```
<div @click.ctrl.right="showContextMenu">Open Menu</div>
```

These keyboard and mouse modifiers help you write clear, focused event handlers that respond only to the user actions you care about.

❖ 3.3.6. Demo: Handling Events with `v-on` in the Vue SFC Playground

Let's explore how Vue handles DOM events using the `@` shorthand for `v-on`. We'll build up from a basic click event to using arguments, the event object, and event modifiers.

1. Open the Vue SFC Playground at `https://play.vuejs.org`.

2. Replace any existing code in **App.vue** with the following:

Demo 3.1: BasicVueReactivity/Demos/v-on-demo.vue

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

const name = ref('Vue Student');
const count = ref(0);

function alertUser(message) {
  alert(message);
}

function greetUser(message) {
  alertUser(`${message}, ${name.value}!`);
}

function handleClickWithEvent(event) {
  const id = event.currentTarget.id;
  alertUser(`Event received from element with ID: ${id}`);
}

function incrementCount() {
  count.value++;
  alertUser(`Count incremented to: ${count.value}`);
}

function handleFormSubmit() {
  alertUser('Form submitted without page reload');
}

function fireOnce() {
  alertUser('This message will only show once');
}

function handleEnter() {
  alertUser('Enter key pressed!');
}

function showContextMenu() {
  alertUser('Right-click detected!');
}
</script>

<template>
  <h2>Vue Event Handling Demo</h2>

```

Demo 3.1: BasicVueReactivity/Demos/v-on-demo.vue

```
45.     <section>
46.       <h3>1. Basic Click</h3>
47.       <button @click="() => alertUser('You clicked the button!')">Click Me</button>
48.     </section>
49.
50.     <section>
51.       <h3>2. Calling a Method with Argument</h3>
52.       <button @click="greetUser('Welcome')">Greet</button>
53.     </section>
54.
55.     <section>
56.       <h3>3. Passing the Event Object</h3>
57.       <button id="event-button" @click="handleClickWithEvent">
58.         Click with Event
59.       </button>
60.     </section>
61.
62.     <section>
63.       <h3>4. Using .prevent on a Form</h3>
64.       <form @submit.prevent="handleFormSubmit">
65.         <input type="text" placeholder="Type here" />
66.         <button type="submit">Submit</button>
67.       </form>
68.     </section>
69.
70.     <section>
71.       <h3>5. .stop Modifier (Stops Propagation)</h3>
72.       <div
73.         @click="() => alertUser('Parent div clicked')"
74.         style="padding: 1rem; background: #f0f0f0"
75.       >
76.         <button @click.stop="() => alertUser('Child button clicked')">
77.           Click Child
78.         </button>
79.       </div>
80.     </section>
81.
82.     <section>
83.       <h3>6. .once Modifier</h3>
84.       <button @click.once="fireOnce">Click Me Once</button>
85.     </section>
86.
87.     <section>
88.       <h3>7. Keyboard Event: @keyup.enter</h3>
```

Demo 3.1: BasicVueReactivity/Demos/v-on-demo.vue

```
89.       <input @keyup.enter="handleEnter" placeholder="Press Enter here" />
90.     </section>
91.
92.     <section>
93.       <h3>8. Right Click with .right Modifier</h3>
94.       <div @click.right="showContextMenu" style="padding: 1rem; background: #eef">
95.         Right-click Me!
96.       </div>
97.     </section>
98.
99.     <section>
100.      <h3>9. Button with Counter</h3>
101.      <button @click="incrementCount">Increment Count</button>
102.      <p>Current Count: {{ count }}</p>
103.    </section>
104.
105.  </template>
```


3. Click the "Reload page" button in the top right of the Playground.

This gives you a clean state and clears any previous interactions.

4. Interact with the app. Try each section.
    A. Click the **Click Me** button to alert a message to the user.
    B. Click the **Greet** button to alert a custom greeting using a passed argument.
    C. Click the **Click with Event** button to see which element triggered the event.
    D. Submit the form and see an alert message instead of a page reload.
    E. Click the child button in the gray box — the parent click won't fire due to `.stop`.
    F. Click **Click Me Once** multiple times — the message will only be alerted the first time.
    G. Type in the input field and press **Enter** to trigger a `@keyup.enter` alert.
    H. Right-click the blue box to alert a right-click event message using `.right`.
    I. Use the **Increment Count** button to see the count update and get an alert message.

In this activity, you learned how to:

1. Respond to events using Vue's `@` shorthand (`v-on`).
2. Pass arguments and use the native event object.
3. Prevent default behaviors and stop propagation.
4. Use keyboard and mouse event modifiers.

3.4. v-bind

Vue.js provides a powerful feature for binding data to your template with the `v-bind` directive. This allows you to dynamically update your DOM with JavaScript expressions. The `v-bind` directive can be used to bind attributes, class, style, and more.

❖ 3.4.1. Introduction to v-bind

The basic syntax for `v-bind` is:

```
<img v-bind:src="imageUrl" alt="Description">
```

In the example above, the `src` attribute of an `<img>` tag is bound to the Vue data property `imageUrl`. Any time the `imageUrl` data property changes, the `src` attribute of the image will automatically update to match.

v-bind shorthand

The `v-bind` directive has a shorthand syntax (`:`), which is more common and quicker to type than the full `v-bind` syntax:

```
<img :src="imageUrl" alt="Description">
```

Going forward, we'll use `:` in place of `v-bind`.

❖ 3.4.2. Binding Standard Attributes

`v-bind` can be used to bind any attribute dynamically based on your data:

```
<a :href="linkUrl" :title="tooltip">Open Link</a>
<button :disabled="isSaving">Save</button>
```

For boolean attributes like `disabled`, which in plain HTML are either present or not, Vue adds the attribute when the bound value is `truthy` (`https://developer.mozilla.org/en-US/docs/Glossary/Truthy`) and removes it when it is `falsy` (`https://developer.mozilla.org/en-US/docs/Glossary/Falsy`):

Boolean Attributes

```
<script setup>
  const isButtonADisabled = true;
  const isButtonBDisabled = false;
</script>
<template>
  <button :disabled="isButtonADisabled">Button A</button>
  <!-- renders <button disabled>Button A</button> -->
  <button :disabled="isButtonBDisabled">Button B</button>
  <!-- renders <button>Button B</button>
</template>
```


❖ 3.4.3. Binding Classes and Inline Styles

The `class` and `style` attributes can be bound just like other attributes, but Vue provides extra features that make them more powerful and flexible. You can bind them in three different ways: as a raw string (i.e. just like other attributes), as an object, or as an array. These patterns let you toggle classes and styles dnuamically or combine multiple values.

Binding as a Raw String

The simplest approach is to bind class and style as you would any other attribute: as a raw string. Vue will apply the string directly, just like static HTML:

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

const isActive = ref(false);

// Computed property that reurns a string of class names
const cardClass = computed(() =>
  isActive.value ? 'card active' : 'card'
);

// Static string for inline styles
const cardStyle = "color: red; font-weight: bold; border: solid;"

</script>

<template>
  <div :class="cardClass" :style="cardStyle">
    This is a card.
  </div>
</template>
```


When to use it:

- When you already have a fully composed class list or style string.
- When using a computed property to assemble the final string (e.g., `"btn btn-" + type`).

Limitations:

- You can't easily toggle individual classes or styles without manually building the string yourself.
- Combining multiple sources of classes or styles is more cumbersome compared to objects or arrays.

Binding as an Object

Use an object when you need fine-grained control over which classes or styles are applied.

- **For classes**: the keys are class names, and the values are conditions that must be truthy for Vue to include them.
- **For styles**: the keys are CSS property names, and the values are the CSS values to apply.

```
<!-- Class object -->
<div :class="{ active: isActive, 'text-danger': hasError }"></div>

<!-- Style object -->
<p :style="{ color: activeColor, fontSize: fontSize + 'px' }"></p>
```

If a class name is not a valid JavaScript identifier (for example, it contains a dash, like `text-danger`), wrap it in quotes.

Binding as an Array

Use an array when you want to compose multiple classes or style objects together. Vue applies all truthy entries and ignores falsy ones.

```
<!-- Class array combining strings and object -->
<div :class="['card', extraClass, { active: isActive, 'text-danger': hasError }]"></div>

<!-- Style array merging multiple objects -->
<p :style="[baseStyle, emphasisStyle]">Merged Styles</p>
```


This gives you the most flexibility for combining multiple classes or styles of different formats.

Mixing Static and Dynamic Values

Vue automatically merges static `class` and `style` attributes with those provided through `v-bind`. This means you can freely mix static and dynamic values:

```
<div class="text-danger" :class="{ active: isActive }">
  Static + Dynamic
</div>
```

If `isActive` is true, the element will have both text-danger and active classes; if it is false, it will only have text-danger. The same applies to styles:

```
<p style="font-size: 16px;" :style="{ color: activeColor }">
  Static + Dynamic Styles
</p>
```

Here, the static font size remains in place while Vue dynamically applies the color.

❖ 3.4.4. Binding Attributes with `v-bind` in the Vue SFC Playground

Vue's `v-bind` directive lets you connect your data to HTML attributes so your UI updates automatically when your state changes.

This activity demonstrates the most common uses of `v-bind`: binding standard attributes, classes, styles, and boolean attributes.

1. Open the Vue SFC Playground at `https://play.vuejs.org`.

2. Copy and paste the code below into **App.vue**. Replace any existing code with the following:

Demo 3.2: BasicVueReactivity/Demos/v-bind.vue

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

const imageUrl = ref('https://vuejs.org/images/logo.png');
const isDisabled = ref(false);

// For class examples
const isActive = ref(false);
const hasError = ref(false);
const extraClass = ref('dashed');
const rawClass = computed(() => (isActive.value ? 'card active' : 'card'));

// For style example
const fontSize = ref(16);
const accentColor = ref('darkorange');
</script>

<template>
  <h2>v-bind Demo</h2>

  <!-- 1) Standard binding -->
  <section>
    <h3>1. Standard binding</h3>
    <img :src="imageUrl" alt="Vue Logo" style="height: 80px" />
    <p>Uses <code>:src="imageUrl"</code>.</p>
  </section>

  <!-- 2) Boolean binding -->
  <section>
    <h3>2. Boolean binding</h3>
    <button :disabled="isDisabled">Submit</button>
    <button @click="isDisabled = !isDisabled">Toggle Disabled</button>
  </section>

  <!-- 3) Class binding with a raw string (computed property?) -->
  <section>
    <h3>3. Class binding with a raw string (computed)</h3>
    <div :class="rawClass" class="demo-border pad">rawClass &rightarrow; <code>{{
          rawClass }}</code></div>
    <button @click="isActive = !isActive">Toggle isActive</button>
  </section>

  <!-- 4) Style binding with object -->
  <section>
```

Demo 3.2: BasicVueReactivity/Demos/v-bind.vue

```
      <h3>4. Style binding with object</h3>
      <p :style="{ color: accentColor, fontSize: fontSize + 'px' }" class="demo-
           border pad">
       Object style: <code>{ color: accentColor, fontSize: fontSize + 'px' }</code>
      </p>
      <button @click="fontSize += 2">Increase Font Size</button>
    </section>

    <!-- 5) Class binding with array -->
    <section>
      <h3>5. Class binding with array</h3>
     <div :class="['card', extraClass, { active: isActive, 'text-danger': hasError
           }]" class="demo-border pad">
       Array class: <code>['card', extraClass, { active: isActive, 'text-danger':
           hasError }]</code>
      </div>
      <div>
        <button @click="extraClass = extraClass === 'dashed' ? 'shadow' :
           'dashed'">Cycle extraClass</button>
        <button @click="isActive = !isActive">Toggle isActive</button>
        <button @click="hasError = !hasError">Toggle hasError</button>
      </div>
    </section>
  </template>

  <style scoped>
  /* Basic styles */
  .demo-border { border: 1px solid #ccc; margin: 6px 0; }
  .pad { padding: 8px; }
  button { margin-right: 8px; }

  /* Class examples */
  .card { background: #f9f9f9; padding: 6px; border-radius: 4px; }
  .dashed { border: dashed 1px; }
  .shadow { box-shadow: 0 2px 8px pink; }
  .active { outline: 2px solid green; }
  .text-danger { color: red; }
  </style>
```


3. Click the "Reload page" button in the top right of the Playground to reset the state.

4. Interact with the app and observe the alert messages:
    A. The Vue logo is shown using `:src="imageUrl"` (standard attribute binding).
    B. Click **Toggle Disabled** to enable or disabled the Submit button (boolean attribute binding).
    C. Click **Toggle isActive** to add or remove the `active` class on the box (raw string `class` binding).
    D. Click **Increase Font Size** to dynamically grow the paragraph's font-size of the paragraph (object `style` binding).
    E. Click **Cycle extraClass**, **Toggle isActive**, and **Toggle hasError** to see how multiple classes from an array are combined and conditionally applied.
    F. Click **Toggle Disabled** to enable or disable the submit button.

In this activity, you learned how to use `v-bind` to:

1. Bind standard attributes like `src`.
2. Toggle boolean attributes like `disabled`.
3. Apply classes and styles using a raw string (e.g., through a computed property).
4. Apply classes and inline styles using an object.
5. Combine multiple classes and styles with array syntax.

3.5. Introduction to v-model

The `v-model` directive in Vue.js is a powerful feature that provides a two-way data-binding capability. It allows you to dynamically bind form input values to data properties on your Vue instances. With `v-model`, you can synchronize the data between form inputs and your application's state effortlessly, resulting in more interactive and responsive user interfaces.

❖ 3.5.1. Basic Usage of v-model

The usage of `v-model` is straightforward. It is often used on form elements such as `<input>`, `<textarea>`, and `<select>`. Here's a simple example of using `v-model` with an input field:

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

const message = ref('Hello, Vue!');
</script>

<template>
  <input type="text" v-model="message" />
  <p>Message: {{ message }}</p>
</template>
```


In this example, typing into the input field will automatically update the `message` data property, and the paragraph will reflect the changes immediately.

v-model vs v-bind

The key difference between `v-bind` and `v-model` is that `v-model` creates a *two-way* binding. `v-bind` allows changes made directly to a data property to update wherever that data is displayed. `v-model` allows that *and the reverse*: when the user interacts with the input element (typing in a text field, selecting a checkbox, etc.), the underlying data property is updated automatically, keeping the data and the UI perfectly in sync.

❖ 3.5.2. v-model with Different Input Types

You can use `v-model` with different input types and other HTML elements:

v-model on a checkbox

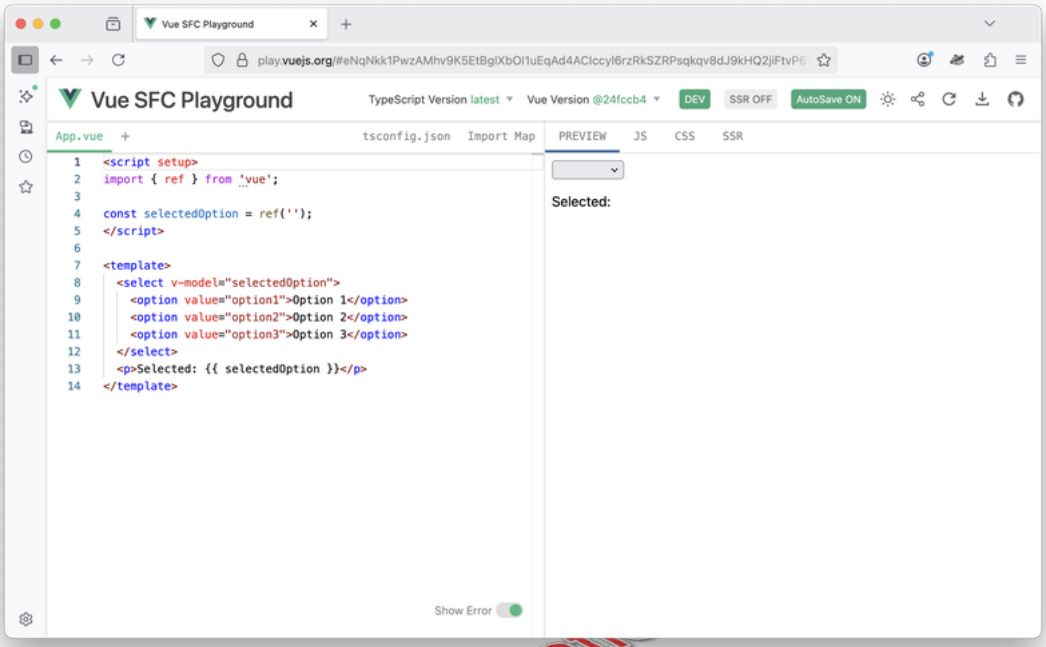
```
<script setup>
import { ref } from 'vue';

const isChecked = ref(false);
</script>

<template>
  <input type="checkbox" v-model="isChecked" />
  <p>Checked: {{ isChecked }}</p>
</template>
```

v-model on a select

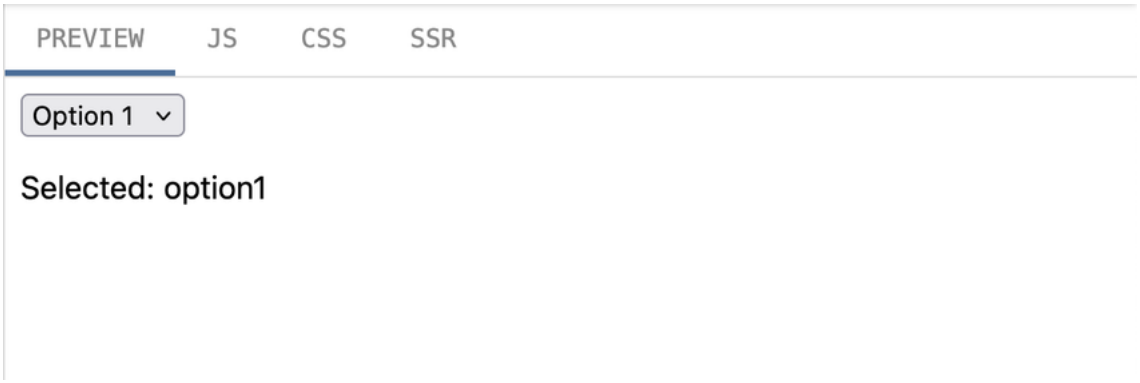
```
<script setup>
import { ref } from 'vue';

const selectedOption = ref('');
</script>

<template>
  <select v-model="selectedOption">
    <option value="option1">Option 1</option>
    <option value="option2">Option 2</option>
    <option value="option3">Option 3</option>
  </select>
  <p>Selected: {{ selectedOption }}</p>
</template>
```


Let's try out 2-way binding of form inputs in the `Vue SFC Playground`:

1. Go to the **Vue SFC Playground**: `https://play.vuejs.org`.
2. Copy and paste the the code from the **v-model on a select** demo into **App.vue**:



3. Choose **Option 1** from the select dropdown and notice that the text appears to the right of **Selected:**.



4. Modify the initial value of the `selectedOption` ref to **option2**, and notice that the value on the select input changes and the value to the right of **Selected:** updates. This is two-way binding in action!

5. Test out the checkbox and text input demos by copying them into the Vue SFC Playground.

❖ 3.5.3. Modifiers

Vue provides modifiers for `v-model` to handle form input edge cases. These modifiers fine-tune the behavior of v-model to suit specific needs.

.lazy

Updates the binding only after `change` events (https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/change_event):


```
<input v-model.lazy="message" />
```

.number

Automatically converts user input to a number:

```
<input v-model.number="age" type="number" />
```

.trim

Automatically trims whitespace from user input:

```
<input v-model.trim="username" />
```

Exercise 3: Building a Pet Adoption Filter System

30 to 45 minutes

In this exercise, you will start to transform the basic PetVue homepage into an interactive pet adoption filtering system. You'll add reactive data management and dynamic filtering capabilities that demonstrate core Vue 3 reactivity concepts.

❖ E3.1. Part 1: Creating the Pet Data Structure

First, we need to create sample pet data to work with throughout this exercise.

1. Create a new folder in your `src/` directory called `data`
2. Create a new file inside the `src/data` folder named `pets.js`
3. Copy and paste the following pet data to the file:

This data structure includes all the properties we'll need for filtering: type, size, energy level, adoption status, and more.

Exercise Code 3.1: src/data/pets.js

```
export const pets = [
{
    id: 1,
    name: 'Bella',
    type: 'Cat',
    breed: 'Siamese',
    age: 2,
    gender: 'Female',
    size: 'Medium',
    bio: 'Bella is a gentle and affectionate cat who loves napping in sunny spots
          and watching birds from the window. She gets along well with other
         cats and would make a perfect companion.',
    image:
      'https://images.unsplash.com/photo-1514888286974-
         6c03e2ca1dba?w=400&h=300&fit=crop&crop=face',
    adopted: false,
    location: 'San Francisco, CA',
    energy: 'Low',
    goodWith: ['cats', 'adults'],
    vaccinated: true,
    spayed: true,
  },
  {
    id: 2,
    name: 'Max',
    type: 'Dog',
    breed: 'Golden Retriever',
    age: 4,
    gender: 'Male',
    size: 'Large',
    bio: "Max is an energetic and friendly dog who loves playing fetch and going
          on long walks. He's great with kids and other dogs, making him perfect
          for an active family.",
    image:
      'https://images.unsplash.com/photo-1552053831-
         71594a27632d?w=400&h=300&fit=crop&crop=face',
    adopted: false,
    location: 'San Francisco, CA',
    energy: 'High',
    goodWith: ['kids', 'dogs', 'cats'],
    vaccinated: true,
    neutered: true,
  },
  {
    id: 3,
```

Exercise Code 3.1: src/data/pets.js

```
40.       name: 'Luna',
41.       type: 'Cat',
42.       breed: 'Maine Coon',
43.       age: 1,
44.       gender: 'Female',
45.       size: 'Large',
46.       bio: 'Luna is a playful kitten with a fluffy coat and bright green eyes. She
               loves interactive toys and is very social with both humans and other
               pets.',
47.       image:
48.         'https://images.unsplash.com/photo-1574158622682-
              e40e69881006?w=400&h=300&fit=crop&crop=face',
49.       adopted: false,
50.       location: 'Oakland, CA',
51.       energy: 'High',
52.       goodWith: ['kids', 'cats', 'dogs'],
53.       vaccinated: true,
54.       spayed: false,
55.     },
56.     {
57.       id: 4,
58.       name: 'Charlie',
59.       type: 'Dog',
60.       breed: 'Labrador Mix',
61.       age: 3,
62.       gender: 'Male',
63.       size: 'Medium',
64.       bio: "Charlie is a sweet and gentle dog who loves cuddles and treats. He's
               perfectly house-trained and would be ideal for a family looking for a
               calm, loving companion.",
65.       image:
66.         'https://images.unsplash.com/photo-1551717743-
              49959800b1f6?w=400&h=300&fit=crop&crop=face',
67.       adopted: false,
68.       location: 'San Jose, CA',
69.       energy: 'Medium',
70.       goodWith: ['kids', 'adults'],
71.       vaccinated: true,
72.       neutered: true,
73.     },
74.     {
75.       id: 5,
76.       name: 'Mia',
77.       type: 'Cat',
78.       breed: 'Tabby',
```

Exercise Code 3.1: src/data/pets.js

```
79.       age: 5,
80.       gender: 'Female',
81.       size: 'Small',
82.       bio: "Mia is a calm and independent cat who enjoys quiet environments. She's
                perfect for someone looking for a low-maintenance companion who still
                enjoys gentle affection.",
83.       image:
84.         'https://images.unsplash.com/photo-1573865526739-
               10659fec78a5?w=400&h=300&fit=crop&crop=face',
85.       adopted: true,
86.       location: 'Berkeley, CA',
87.       energy: 'Low',
88.       goodWith: ['adults'],
89.       vaccinated: true,
90.       spayed: true,
91.     },
92.     {
93.       id: 6,
94.       name: 'Rocky',
95.       type: 'Dog',
96.       breed: 'German Shepherd',
97.       age: 6,
98.       gender: 'Male',
99.       size: 'Large',
100.      bio: "Rocky is a loyal and intelligent dog with protective instincts. He's
                well-trained and would be perfect for an experienced dog owner looking
                for a devoted companion.",
101.      image:
102.        'https://images.unsplash.com/photo-1589941013453-
               ec89f33b5e95?w=400&h=300&fit=crop&crop=face',
103.      adopted: false,
104.      location: 'Palo Alto, CA',
105.      energy: 'Medium',
106.      goodWith: ['adults'],
107.      vaccinated: true,
108.      neutered: true,
109.    },
110.  ]
```

4. Create a script block in `HomeView.vue` and import `ref` from Vue and the pet data from your data file.

HomeView.vue

```
<script setup>
  import { ref } from 'vue';
  import { pets as samplePets } from '../data/pets';
</script>
```

5. Make the pet data reactive using `ref`:

HomeView.vue

```
<script setup>
  import { ref } from 'vue';
  import { pets as samplePets } from '../data/pets';

  // Reactive data
  const pets = ref(samplePets);
</script>
```


❖ E3.2. Part 2: Creating the Filters

Now we'll build the dynamic pet filtering form.

1. Use the same pattern you used to make the `pets` variable reactive to create two new reactive variables in `HomeView.vue`: `selectedType` and `selectedSize`. We'll use these to filter the list of pets.

```
const selectedType = ref('');
const selectedSize = ref('');
```

2. Just below the "hero-section" `div` (before the final closing `</div>` in the `<template>`), add `select` inputs for Pet Type and Size to the template. Use `v-model` to bind them to the new reactive state variables.

```
<div>
  <h5>Filter Pets</h5>
  <div>
    <label for="typeFilter" class="form-label">Pet Type</label>
    <select id="typeFilter" v-model="selectedType" class="form-select">
      <option value="">All Types</option>
      <option value="Dog">Dogs</option>
      <option value="Cat">Cats</option>
    </select>
  </div>
  <div>
    <label for="sizeFilter" class="form-label">Size</label>
    <select id="sizeFilter" v-model="selectedSize" class="form-select">
      <option value="">All Sizes</option>
      <option value="Small">Small</option>
      <option value="Medium">Medium</option>
      <option value="Large">Large</option>
    </select>
  </div>
</div>
```


3. In the `<script>` block, add `computed` to the list of imports from Vue. We'll use a computed property to dynamically remove pets from the list of pets when one of the reactive filters changes.

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

4. Add the filter functionality to the `<script>` block, just below the line that creates the reactive data. To do this:

   A. Create a computed property called `filteredPets`. We want this property to automatically recalculate whenever `pets`, `selectedType`, or `SelectedSize` changes.

   B. Inside the computed property, take advantage of `JavaScript's filter() method (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter)` to filter the `pets` array by `selectedType` and `selectedSize` variables.

C. See our approach below:

```
// Computed property for filtered pets
const filteredPets = computed(() => {

  // Loop through the pets array and find pets who match either filter
  return pets.value.filter((pet) => {

    // Type filter: include all if no type is selected
    const matchesType =
      selectedType.value === '' || pet.type === selectedType.value;

    // Size filter: include all if no size is selcted
    const matchesSize =
      selectedSize.value === '' || pet.size === selectedSize.value;

    // Only return pets that match BOTH filters
    return matchesType && matchesSize;
  });
});
```

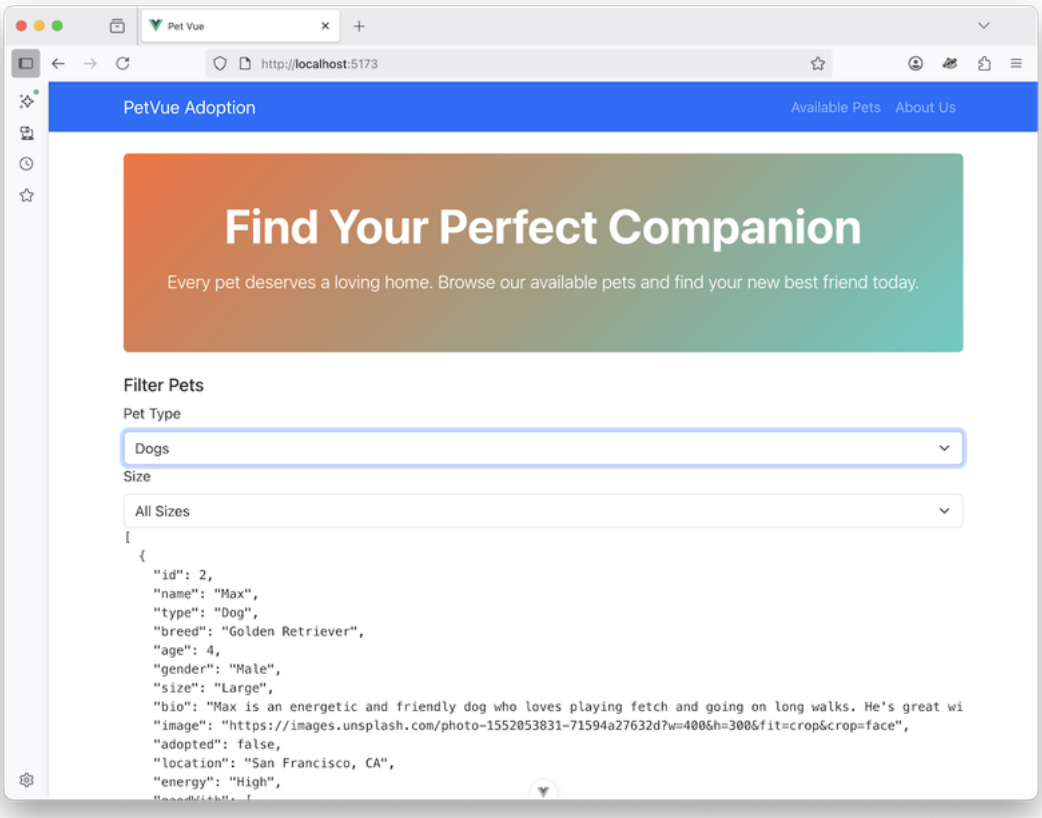

5. To see the filtering take effect, you can convert the value of `filteredPets` to a string using the `JSON.stringify()` method (`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify`) and output it just before the final `</div>` in the template:

```
<pre>{{ JSON.stringify(filteredPets, null, 2) }}</pre>
```

`JSON.stringify(filteredPets, null, 2)` converts the `filteredPets` array into a nicely formatted JSON string, using 2 spaces for indentation so it easy to read.

Wrapping it in a `<pre>` tag tells the browser to preserve whitespace and line breaks, so the spacing from `JSON.stringify` shows up correctly and the array of objects is displayed in a readable, structured way.

6. Start the dev server if it's not already running (using **npm run dev**) and go to `http://localhost:5173` in your browser. You should see the `filteredPets` array display. Try changing the filters to change the pets in the `filteredPets` array.



Here's what your `HomeView.vue` component should look like at this point:

Solution: BasicVueReactivity/Solutions/pet-vue-filter-system/src/views/HomeView.vue

```
<template>
  <div class="pet-list-page">
    <!-- Hero Section -->
    <div class="hero-section text-center py-5 mb-4 rounded">
      <h1 class="display-4 fw-bold mb-3">Find Your Perfect Companion</h1>
      <p class="lead">
        Every pet deserves a loving home. Browse our available pets and find
        your new best friend today.
      </p>
    </div>
    <div>
      <h5>Filter Pets</h5>
      <div>
        <label for="typeFilter" class="form-label">Pet Type</label>
        <select id="typeFilter" v-model="selectedType" class="form-select">
          <option value="">All Types</option>
          <option value="Dog">Dogs</option>
          <option value="Cat">Cats</option>
        </select>
      </div>
      <div>
        <label for="sizeFilter" class="form-label">Size</label>
        <select id="sizeFilter" v-model="selectedSize" class="form-select">
          <option value="">All Sizes</option>
          <option value="Small">Small</option>
          <option value="Medium">Medium</option>
          <option value="Large">Large</option>
        </select>
      </div>
    </div>
  </div>

  <pre>{{ JSON.stringify(filteredPets, null, 2)}}</pre>
</template>

<script setup>
  import { ref, computed } from 'vue';
  import { pets as samplePets } from '../data/pets';

  // Reactive data
  const pets = ref(samplePets);
  const selectedType = ref('');
  const selectedSize = ref('');
```

Solution: BasicVueReactivity/Solutions/pet-vue-filter-system/src/views/HomeView.vue

```
44.
45.     // Computed property for filtered pets
46.     const filteredPets = computed(() => {
47.
48.       // Loop through the pets array and find pets who match either filter
49.       return pets.value.filter((pet) => {
50.
51.         // Type filter: include all if no type is selected
52.         const matchesType =
53.           selectedType.value === '' || pet.type === selectedType.value;
54.
55.         // Size filter: include all if no size is selcted
56.         const matchesSize =
57.           selectedSize.value === '' || pet.size === selectedSize.value;
58.
59.         // Only return pets that match BOTH filters
60.         return matchesType && matchesSize;
61.       });
62.     });
63.   </script>
```


You can see our solution yourself by running the `basicvuereactivity/Solutions/pet-vue-filter-system/` app:

1. Make sure the dev server isn't running.
2. Right click on `pet-vue-filter-system` and click **Open in Integrated Terminal**.
3. Run **npm install** to install dependencies.
4. Run **npm run dev** to run the app.

Conclusion

In this lesson, you learned the core concepts of Vue reactivity, gaining a comprehensive understanding of how to create data-driven interfaces. You learned how to use `ref` and `reactive` to manage reactive state, how computed properties work for deriving dynamic data, and how to effectively handle user events with `v-on`. We also explained the power of directives like `v-bind` and `v-model` to keep your UI in sync with underlying data. Through practical exploration, you have seen how these elements fit

together to enhance your Vue applications, boosting interactivity and responsiveness. As a result, you are now equipped to craft interfaces that are both dynamic and user-friendly.

LESSON 4
Conditionals and Loops

Topics Covered

☑ Conditional rendering with v-if.

☑ Controlling visibility with v-show.

☑ Iterating lists and objects with v-for.

☑ Using keys in v-for for stable rendering.

☑ Combining conditionals and loops in UI components.

4.1. Introduction

In this lesson, you will master conditionals and loops in Vue. You will use `v-if`, `v-else`, and `v-show` to control what renders versus what's merely hidden, choose the right approach for performance and layout, and explore style-based alternatives like visibility and opacity to avoid layout shifts.

You will iterate data with `v-for` across arrays and objects, access indexes, apply stable keys, and nest loops to render structured data. We will guide you through practical demos, and in the exercise, you will enhance your PetVue app by displaying and filtering the pets in a responsive grid.

4.2. Understanding v-if and v-show in Depth

The `v-if` and `v-show` directives can both be used to control the visibility of elements, but they serve different use cases and impact performance in distinct ways. Understanding when to use each will enhance the efficiency of your Vue applications.

❖ 4.2.1. v-if Directive

The `v-if` directive conditionally renders a block. It ensures that the DOM elements associated with a directive exist only if the condition is true. This means that `v-if` renders your element only when the condition is met, causing Vue to destroy and recreate elements as the condition changes.

Using v-if

```
<div v-if="isVisible">
  <p>This text renders only when isVisible is true.</p>
</div>
```


The above example will render the paragraph only when the `isVisible` data property is true.

Performance Considerations

Using `v-if` involves Vue monitoring condition changes and rerendering components, which can be performance-intensive, especially in complex DOM structures. Use `v-if` when the condition is less likely to change frequently.

❖ 4.2.2. v-else Directive

You can create a `v-else` element immediately following a `v-if` element. If present, the `v-else` element will render when the `v-if` condition is false.

The following demo shows the use of `v-if` and `v-else` to conditionally render either a login form or a user profile link.

Demo 4.1: ConditionalsLoopsDynamicStyles/Demos/v-if-else.vue

```
<template>
  <div class="auth">
    <form v-if="!isLoggedIn" @submit.prevent="login" novalidate>
      <label for="email">Email</label>
      <input id="email" v-model="email" type="email" required />

      <label for="password">Password</label>
      <input id="password" v-model="password" type="password" required />

      <button type="submit">Log In</button>
    </form>

    <div v-else>
      <a href="http://example.com/profile">View Profile</a>
    </div>
  </div>
</template>

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

const isLoggedIn = ref(false);
const email = ref('');
const password = ref('');

function login() {
  if (email.value && password.value) {
    isLoggedIn.value = true;
  }
}
</script>

<style scoped>
.auth {
  display: grid;
  gap: 12px;
  max-width: 340px;
}
.auth label {
  display: block;
  font-weight: 600;
  margin-top: 8px;
}
.auth input {
```

Demo 4.1: ConditionalsLoopsDynamicStyles/Demos/v-if-else.vue

```
    width: 100%;
    padding: 8px;
    margin-top: 4px;
    box-sizing: border-box;
  }
  .auth button {
    margin-top: 12px;
  }
  </style>
```

To try out the above demo, go to `http://play.vuejs.org` and paste the code into the **App.vue** component. You'll see a component in the preview pane of the playground that looks like the following:

Enter anything at all into the Email and Password fields and click Login. The preview pane will change to display a fake "View Profile" link, as shown here:

❖ 4.2.3. v-show Directive

The `v-show` directive toggles the visibility of an element via CSS rather than conditional rendering. Unlike `v-if`, elements associated with `v-show` are rendered on the DOM regardless of the condition but are hidden using `display:none;`.

Using v-show

```
<div v-show="isHidden">
  <p>This text is hidden when isHidden is false.</p>
</div>
```

This example keeps the paragraph in the DOM but hides it from the display if `isHidden` is false.

Choosing Between v-show and v-if

`v-if` and `v-show` are both used for conditional rendering, but they work quite differently. When you use `v-if`, the element and its children are created only when the condition is true. On each toggle, Vue creates or destroys the DOM nodes and tears down or reattaches event listeners. It's also lazy: if the condition starts false, nothing gets rendered until it becomes true.

By contrast, `v-show` always renders the element once and then simply toggles its `display` CSS property to show or hide it. This makes `v-show` faster to toggle—especially on frequently changing conditions—but there's a small upfront cost because the element is rendered regardless.

Both directives affect the layout because an element with `display: none` no longer takes up space. If you need to hide something without moving the surrounding elements, you can bind its `visibility` or `opacity` style to a reactive value instead of using `v-if` or `v-show`. When you toggle `visibility: hidden` on an element, it stays in the DOM and still occupies space while remaining invisible. This approach lets you preserve the layout and avoid unwanted shifts while still controlling the element's visibility.

❖ 4.2.4. Best Practices

- When rendering components that are resource-intensive and do not change often, use `v-if`.
- Prefer `v-show` for simple toggling of elements that need to be available in the DOM at all times.

- Toggle the element's visibility or opacity style to hide an element without affecting the layout of the page.

The following demo shows the difference between these three approaches. To try it out, you can copy the code into the Vue SFC Playground (`https://play.vuejs.org`).

Demo 4.2: ConditionalsLoopsDynamicStyles/Demos/conditional-rendering.vue

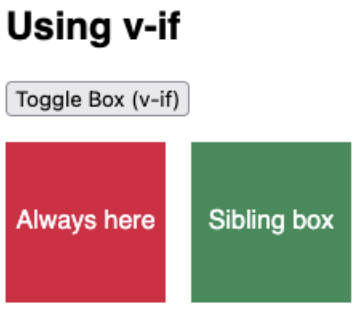
```
<template>
  <div class="demo">
    <!-- v-if removes the element completely -->
    <h2>Using v-if</h2>
    <button @click="toggleIf">Toggle Box (v-if)</button>
    <div class="container">
      <div class="box red">Always here</div>
      <div v-if="showIf" class="box blue">v-if box</div>
      <div class="box green">Sibling box</div>
    </div>

    <!-- v-show toggles display: none, which still affects layout -->
    <h2>Using v-show</h2>
    <button @click="toggleShow">Toggle Box (v-show)</button>
    <div class="container">
      <div class="box red">Always here</div>
      <div v-show="showShow" class="box blue">v-show box</div>
      <div class="box green">Sibling box</div>
    </div>

    <!-- visibility keeps space in the layout -->
    <h2>Using visibility</h2>
    <button @click="toggleVis">Toggle Box (visibility)</button>
    <div class="container">
      <div class="box red">Always here</div>
      <div
        class="box blue"
        :style="{ visibility: visHidden ? 'hidden' : 'visible' }"
      >
        visibility box
      </div>
      <div class="box green">Sibling box</div>
    </div>
  </div>
</template>

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

const showIf = ref(true);
const showShow = ref(true);
const visHidden = ref(false);

```

Demo 4.2: ConditionalsLoopsDynamicStyles/Demos/conditional-rendering.vue

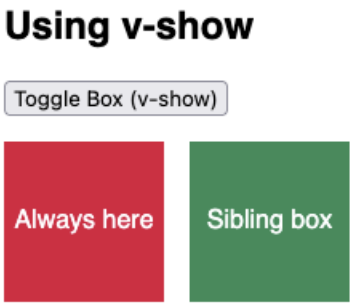
```
44.  const toggleIf = () => {
45.    showIf.value = !showIf.value;
46.  };
47.  const toggleShow = () => {
48.    showShow.value = !showShow.value;
49.  };
50.  const toggleVis = () => {
51.    visHidden.value = !visHidden.value;
52.  };
53.  </script>
54.
55.  <style scoped>
56.  .demo {
57.    font-family: sans-serif;
58.    padding: 1rem;
59.  }
60.
61.  .container {
62.    display: flex;
63.    gap: 1rem;
64.    margin: 1rem 0;
65.  }
66.
67.  .box {
68.    width: 100px;
69.    height: 100px;
70.    color: white;
71.    display: flex;
72.    align-items: center;
73.    justify-content: center;
74.  }
75.
76.  .red {
77.    background: crimson;
78.  }
79.  .blue {
80.    background: royalblue;
81.  }
82.  .green {
83.    background: seagreen;
84.  }
85.  </style>
```

This demo shows three rows of boxes. In each row, the first and third boxes are the same.

In the top row, clicking the button will remove the element using v-if. Notice that the "Sibling" box moves into the space where the middle box was:



In the 2nd row, clicking the button will hide the element with v-show. In this example, the "Sibling" box will also slide over to take up the space that once was taken up by the middle box. The middle box is still there, it's just not taking up any space at the moment.



In the 3rd row, clicking the button will make the 2nd box invisible. It's still there and it still takes up space, you just can't see it:

4.3. Using v-for for Iteration

The `v-for` directive allows you to loop through arrays and objects with ease, providing the ability to create dynamic and data-driven components.

❖ 4.3.1. Basic Usage of v-for

The `v-for` directive is used to render a list of items based on an array. The syntax is straightforward, and you can use it to loop through data collections like arrays or objects. Looping over arrays is particularly useful in generating lists in a user interface.

For example, the following is an array of objects. Each object might represent one item on a to-do list:

```
[
  {id:1, name:'do the laundry'},
  {id:2, name:'mow the lawn'},
  {id:3, name:'get an oil change'}
]
```

To loop over this array and create an HTML list in a Vue application, you can use the following code:

```
<ul>
  <li v-for="item in items" :key="item.id">
    {{ item.name }}
  </li>
</ul>
```


In the above code, the `v-for` directive iterates over the `items` array, rendering a `<li>` for each `item` object.

Notice the :key attribute

The value of the `key` attribute is used internally by Vue as a unique identifier for each element in the array. Giving each element in the array a unique identifier makes it possible for Vue to minimize the number of steps required to update the list later on when you remove, rearrange, or add items to it.

It is best practice to always include a `key` attribute. If it is not included, Vue will try to reuse and patch existing DOM elements based on their order in the list when the underlying data changes. This can lead to unexpected behavior, such as items not updating correctly, or losing their state (for example, text entered into an input field disappearing when the array changes). By providing a unique key, you ensure that Vue can track each item precisely, leading to more predictable and efficient updates.

In this simple demo, we are using the unique `id` of each `item` object as the `key`. You can read more about the `key` attribute here: `https://vuejs.org/guide/essentials/list.html#maintaining-state-with-key`.

❖ 4.3.2. Accessing the Array Index in v-for

Sometimes you may need to access not only the item itself, but also its position in the array. Vue's `v-for` allows you to capture the index of the current item by using a special syntax with parentheses:

```
<li v-for="(item, index) in items" :key="item.id">
  {{ index }} - {{ item.name }}
</li>
```

In the example above, `index` will hold the current item's numeric position in the array (starting from 0).

Accessing the index is common when you want to display an item number in a table:

Displaying Indexes in a Table

```
<table>
  <tr>
    <th>#</th>
    <th>Task</th>
  </tr>
  <tr v-for="(item, index) in items" :key="item.id">
    <td>{{ index + 1 }}</td>
    <td>{{ item.name }}</td>
  </tr>
</table>
```

In this demo, we are displaying `{{ index + 1 }}` so that the item number starts at 1 instead of 0.

Should you use the array index as the key?

The short answer is: **No (most of the time)**.

The purpose of the `:key` is to give Vue a stable and unique identity for each list item so it can correctly update, reorder, or remove DOM elements with minimal work. The array index is not

stable because if items are added, removed, or reordered, the indexes of the remaining items will shift. This can cause the same kinds of issues as when `:key` is not used at all.

Best Practices:

- If you are iterating through an array that you expect will change, always use a stable, unique property from the object such as `id`. If your array is a simple list of numbers or strings and there isn't a unique property, consider converting it to a list of objects:

  ```
  ["red", "green", "blue"]
  ```

  becomes

  ```
  [{id: 1, color: "red"}, {id: 2, color: "green"}, {id: 3, color: "blue"}]
  ```

- If you are unable to convert the list to be a list of objects, consider whether the item itself can be used as the key. If all values in the list of colors are going to be unique, you can use the color itself as the key value:

  ```
  <li v-for="color in colors" :key="color">{{ color }}</li>
  ```

- If the list is truly static (i.e. never changes order, no additions or deletions), using `:key="index"` is relatively save and is fine to use:

  ```
  <li v-for="(color, index) in staticColors" :key="index">{{ color }}</li>
  ```


❖ 4.3.3. Iterating Over Objects

You can also use `v-for` to iterate over the properties of an object.

```
<script setup>
  const myObject = { id: 1, firstName: "John", lastName: "Smith" };
</script>
<template>
  <ul>
    <li v-for="(value, key) in myObject" :key="key">
      {{ key }}: {{ value }}
    </li>
  </ul>
</template>
```


This example demonstrates how to loop through the `myObject`, providing you with both the `key` and the `value` of each object property. It will output a list that looks like:

- id: 1
- firstName: John
- lastName: Smith

❖ 4.3.4. Nesting v-for Loops

Sometimes, you may need to nest `v-for` loops to iterate over a complex data structure. This technique allows you to render items within items, creating multi-level components.

```
<script setup>
const groupedItems = [
  {
    id: 1, title: "Group 1",
    items: [{id: 1, name: "Item 1"},{id: 2, name: "Item 2"}]
  },
  {
    id: 2, title: "Group 2",
    items: [{id: 1, name: "Item 1"},{id: 2, name: "Item 2"}]
  }
];
</script>
<template>
  <ul>
    <li v-for="group in groupedItems" :key="group.id">{{ group.title }}
      <ul>
        <li v-for="item in group.items" :key="item.id">
          {{ item.name }}
        </li>
      </ul>
    </li>
  </ul>
</template>
```


Here, the outer `v-for` iterates over `groupedItems`, and the inner `v-for` loops through each group's `items` array, creating a nested structure. This will output a list like:

- Group 1
  - Item 1
  - Item 2
- Group 2
  - Item 1
  - Item 2

❖ 4.3.5. Using `v-for` to Loop Over Data in the Vue SFC Playground

Vue's `v-for` directive lets you loop through arrays and objects to dynamically render content. In this activity, you'll explore several common ways to use `v-for` to generate content from different types of data.

1. Open the Vue SFC Playground at `https://play.vuejs.org`.

2. Copy and paste the code below into `App.vue`:

Demo 4.3: ConditionalsLoopsDynamicStyles/Demos/v-for.vue

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

// dynamic groceries array
const groceries = ref([
  { id: 1, name: 'Milk' },
  { id: 2, name: 'Eggs' },
  { id: 3, name: 'Bread' },
]);

// static colors array
const colors = ['Red', 'Green', 'Blue'];

// dynamic profile object
const profile = ref({
  name: 'Jane Doe',
  age: 30,
  location: 'New York',
});

// dynamic categories nested array
const categories = ref([
  {
    id: 1,
    title: 'Fruits',
    items: ['Apples', 'Bananas', 'Oranges'],
  },
  {
    id: 2,
    title: 'Vegetables',
    items: ['Carrots', 'Spinach', 'Broccoli'],
  },
]);
</script>

<template>
  <h2>v-for Demo</h2>

  <section>
    <h3>1. Looping Through an Array of Objects</h3>
    <ul>
      <li v-for="item in groceries" :key="item.id">
        {{ item.name }}
      </li>
```

Demo 4.3: ConditionalsLoopsDynamicStyles/Demos/v-for.vue

```
      </ul>
      <p>
        This list uses <code>v-for="item in groceries"</code> with
        <code>:key="item.id"</code>.
      </p>
    </section>

    <section>
      <h3>2. Looping With Index Through a Static List</h3>
      <ul>
        <li v-for="(color, index) in colors" :key="index">
          {{ index }}: {{ color }}
        </li>
      </ul>
      <p>
        This list uses the second argument to access the index:
        <code>v-for="(color, index) in colors"</code>.
      </p>
    </section>

    <section>
      <h3>3. Looping Through Object Properties</h3>
      <ul>
        <li v-for="(value, key) in profile" :key="key">{{ key }}: {{ value }}</li>
      </ul>
      <p>
        This loop uses <code>v-for="(value, key) in profile"</code> to iterate
        over an object.
      </p>
    </section>

    <section>
      <h3>4. Nested v-for Loops</h3>
      <div
        v-for="category in categories"
        :key="category.id"
        style="margin-bottom: 1rem"
      >
        <strong>{{ category.title }}</strong>
        <ul>
          <li v-for="item in category.items" :key="item">
            {{ item }}
          </li>
        </ul>
```

Demo 4.3: ConditionalsLoopsDynamicStyles/Demos/v-for.vue

```
      </div>
      <p>
        This example shows a <code>v-for</code> loop inside another loop for
        grouped data.
      </p>
    </section>
  </template>
```

3. Click the "Reload page" button in the top right of the Playground. This resets the app and displays all the examples with fresh data.
4. Observe the four demos and compare:
    - The **first list** renders objects using `v-for="item in groceries"` and `:key="item.id"`
    - The **second list** shows how to get the **index** using `v-for="(color, index) in colors"`
    - The **third list** loops through an **object's key-value pairs**
    - The **fourth section** shows how to **nest loops** to display grouped items

In this activity, you saw four ways to use `v-for` in Vue:

- Looping through an array of objects
- Using the array index
- Iterating over object properties
- Nesting `v-for` to display grouped or hierarchical data

These patterns give you flexibility when working with different data structures in Vue components.

❖ 4.3.6. Summary

The `v-for` directive is indispensable when working with lists in Vue.js, providing flexible and efficient ways to render data-driven components. By mastering `v-for`, you'll be able to build components that dynamically adapt to data changes, enhancing both functionality and user experience.

Exercise 4: Displaying and Filtering the Pets

25 to 35 minutes

In this exercise, you will use `v-for`, `v-if`, and `v-show` to display the pets and to create a more interactive and visually appealing pet adoption interface.

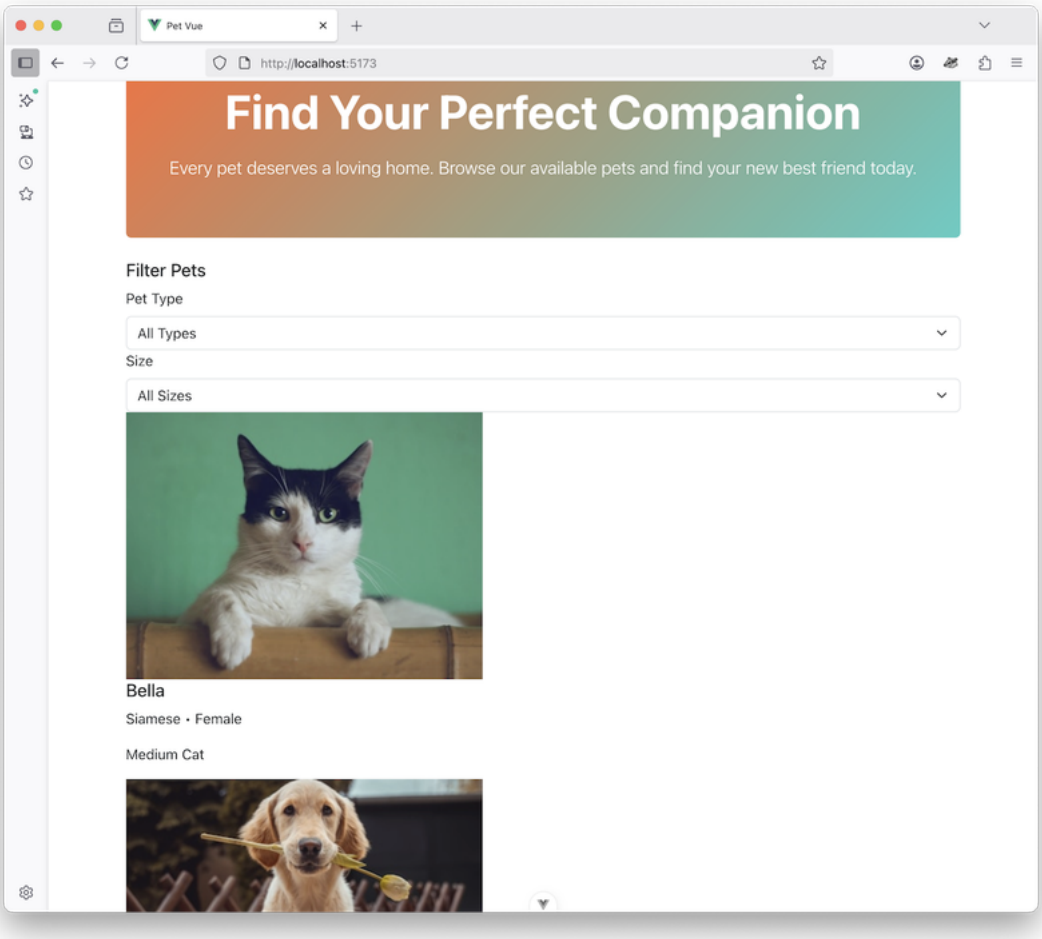
❖ E4.1. Part 1: Display the Pet Data

1. Below the filter select inputs in `HomeView.vue`, use a `<div>` with a `v-for` directive to loop through the `filteredPets` array and output the pet image and some basic info for each pet. The following code replaces the `<pre>` element that was logging the raw `filteredPet` data from the previous lesson.

HomeView.vue

```
<div v-for="pet in filteredPets" :key="pet.id" class="pet-card">
  <img :src="pet.image" :alt="pet.name" />
  <div>
    <h5>{{ pet.name }}</h5>
    <p>{{ pet.breed }} &bull; {{ pet.gender }}</p>
    <p>{{ pet.size }} {{ pet.type }}</p>
  </div>
</div>
```

2. Start the dev server if it's not already running (using **npm run dev**) and go to `http://localhost:5173` in your browser. You should see a list of pets on the home page.

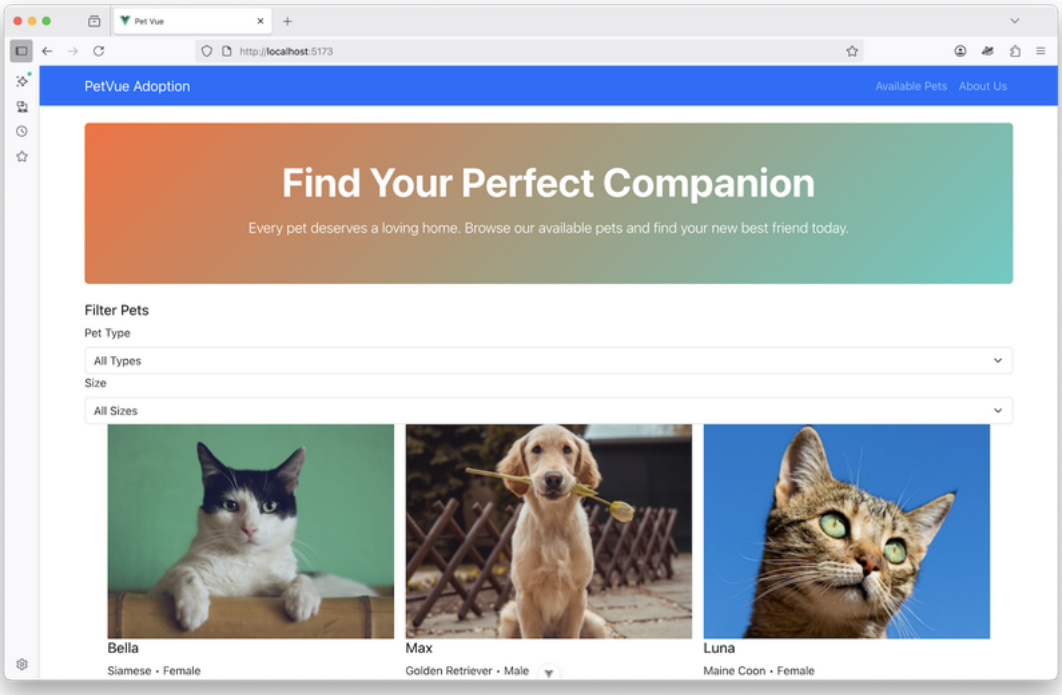


3. Next, you'll style the list of pets as a grid. Add a `<div>` around the `v-for` div and give it an id of petList.

4. Create a `<style>` block at the bottom of `HomeView.vue` and create a new style for the petList. Here's one way you can style the list:

```
<style>
#petList {
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
  gap: 16px;
  justify-content: center;
}
</style>
```

After applying these styles, the homepage of PetVue should look like this:



At this point, you can see how changing the filters changes which pets show up in the grid.

❖ E4.2. Part 2: Add Adoption Status Overlay

Let's add a visual overlay that appears only for adopted pets.

1. In `src/views/HomeView.vue`, add the adoption overlay code right after the `img` tag. The code to insert is shown below. This code uses conditional rendering to display a `<div>` if the pet's adopted status is true.

HomeView.vue

```
<div v-if="pet.adopted" class="adoption-overlay">
  <div class="adoption-message">
    <h6 class="fw-bold mb-1">ADOPTED!</h6>
    <small>Found their forever home</small>
  </div>
</div>
```

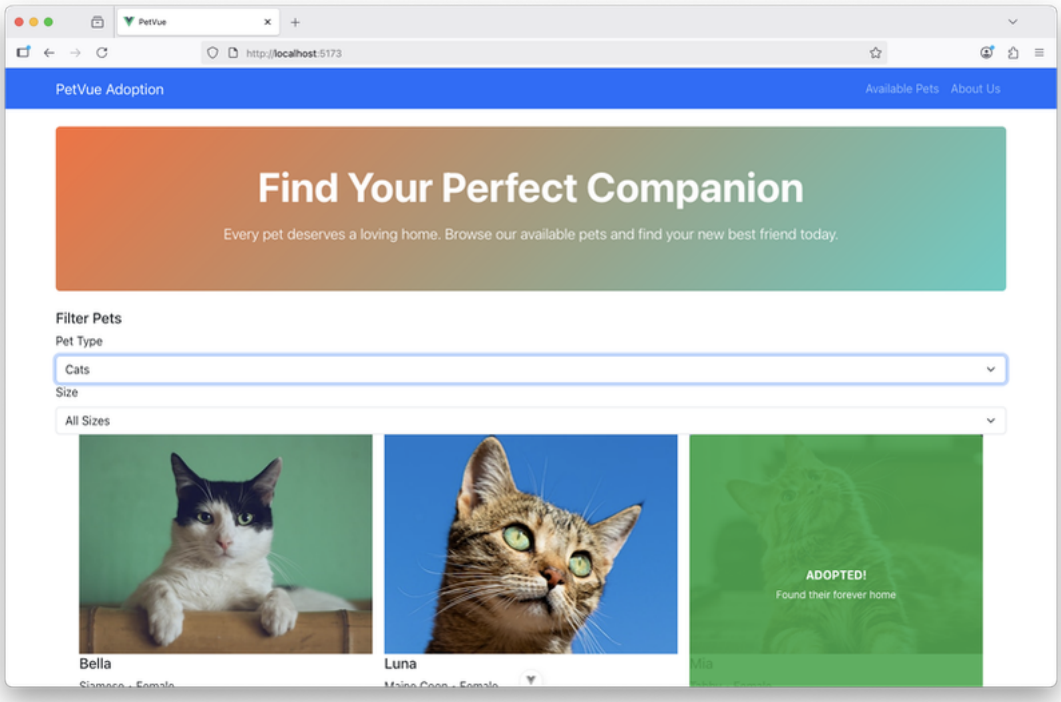
2. Add CSS to your `<style>` section to style the adopted message as an overlay, and add relative position to the pet-card so the adoption overlay will only appear over a single pet card:

HomeView.vue

```
.pet-card {
  position: relative;
}

.adoption-overlay {
  position: absolute;
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;
  background: rgba(40, 167, 69, 0.9);
  display: flex;
  align-items: center;
  justify-content: center;
  color: white;
  text-align: center;
  border-radius: 0.375rem 0.375rem 0 0;
}
```


3. Save your file and check your application. You should see:

- A green overlay with "ADOPTED!" on it appears only on adopted pets
- The overlay covers the entire pet image
- Other pets remain unchanged



Here's the code for `HomeVue.vue` at this point:

Solution:

ConditionalsLoopsDynamicStyles/Solutions/pet-vue-displaying-and-filtering/src/views/HomeView.vue

```
1.   <template>
2.     <div class="pet-list-page">
3.       <!-- Hero Section -->
4.       <div class="hero-section text-center py-5 mb-4 rounded">
5.         <h1 class="display-4 fw-bold mb-3">Find Your Perfect Companion</h1>
6.         <p class="lead">
7.           Every pet deserves a loving home. Browse our available pets and find
8.           your new best friend today.
9.         </p>
10.      </div>
11.      <div>
12.        <h5>Filter Pets</h5>
13.        <div>
14.          <label for="typeFilter" class="form-label">Pet Type</label>
15.          <select id="typeFilter" v-model="selectedType" class="form-select">
16.            <option value="">All Types</option>
17.            <option value="Dog">Dogs</option>
18.            <option value="Cat">Cats</option>
19.          </select>
20.        </div>
21.        <div>
22.          <label for="sizeFilter" class="form-label">Size</label>
23.          <select id="sizeFilter" v-model="selectedSize" class="form-select">
24.            <option value="">All Sizes</option>
25.            <option value="Small">Small</option>
26.            <option value="Medium">Medium</option>
27.            <option value="Large">Large</option>
28.          </select>
29.        </div>
30.      </div>
31.    </div>
32.    <div id="petList">
33.      <div v-for="pet in filteredPets" :key="pet.id" class="pet-card">
34.        <img :src="pet.image" :alt="pet.name" />
35.        <div v-if="pet.adopted" class="adoption-overlay">
36.          <div class="adoption-message">
37.            <h6 class="fw-bold mb-1">ADOPTED!</h6>
38.            <small>Found their forever home</small>
39.          </div>
40.        </div>
41.        <div>
42.          <h5>{{ pet.name }}</h5>
43.          <p>{{ pet.breed }} &bull; {{ pet.gender }}</p>
```

Solution:

ConditionalsLoopsDynamicStyles/Solutions/pet-vue-displaying-and-filtering/src/views/HomeView.vue

```
44.           <p>{{ pet.size }} {{ pet.type }}</p>
45.         </div>
46.       </div>
47.     </div>
48.   </template>
49.
50.   <script setup>
51.     import { ref, computed } from 'vue';
52.     import { pets as samplePets } from '../data/pets';
53.
54.     // Reactive data
55.     const pets = ref(samplePets);
56.     const selectedType = ref('');
57.     const selectedSize = ref('');
58.
59.     // Computed property for filtered pets
60.     const filteredPets = computed(() => {
61.
62.       // Loop through the pets array and find pets who match either filter
63.       return pets.value.filter((pet) => {
64.
65.         // Type filter: include all if no type is selected
66.         const matchesType =
67.           selectedType.value === '' || pet.type === selectedType.value;
68.
69.         // Size filter: include all if no size is selcted
70.         const matchesSize =
71.           selectedSize.value === '' || pet.size === selectedSize.value;
72.
73.         // Only return pets that match BOTH filters
74.         return matchesType && matchesSize;
75.       });
76.     });
77.   </script>
78.
79.   <style>
80.   #petList {
81.     display: flex;
82.     flex-direction: row;
83.     flex-wrap: wrap;
84.     gap: 16px;
85.     justify-content: center;
86.   }
```

Solution:

ConditionalsLoopsDynamicStyles/Solutions/pet-vue-displaying-and-filtering/src/views/HomeView.vue

```
87.
88.   .pet-card {
89.     position: relative;
90.   }
91.
92.   .adoption-overlay {
93.     position: absolute;
94.     top: 0;
95.     left: 0;
96.     right: 0;
97.     bottom: 0;
98.     background: rgba(40, 167, 69, 0.9);
99.     display: flex;
100.    align-items: center;
101.    justify-content: center;
102.    color: white;
103.    text-align: center;
104.    border-radius: 0.375rem 0.375rem 0 0;
105.  }
106.  </style>
```


Conclusion

In this lesson, you learned to control rendering and visibility with `v-if`, `v-else`, and `v-show`, choosing the right approach based on toggle frequency, DOM cost, and layout impact. You also explored using CSS `visibility` and `opacity` to avoid layout shifts. You iterated data with `v-for` across arrays and objects, accessed indexes, applied stable keys, and nested loops for hierarchical structures. We explained how these practices improve performance and predictability in Vue apps.

In the PetVue exercise, you rendered a responsive pet grid with `v-for`, filtered results via a computed list, and added an adoption overlay using `v-if` that appears only for adopted pets.

LESSON 5
Components

Topics Covered

☑ Component fundamentals.

☑ Lifecycle hooks and component update flow.

☑ Passing and validating data with props.

☑ Emitting and handling component events.

☑ Reusing logic with composables.

☑ Slots for content projection and dynamic components for runtime switching.

☑ Refactoring PetVue into smaller, reusable components.

Introduction

In this lesson, you will explore the fundamental concepts of components in Vue.js. We will guide you through the basics of creating and utilizing components, providing a solid foundation for your understanding. You will learn how components interact within an application, explore their lifecycle, and learn how to pass data using props. By the end of this lesson, you will be equipped with the knowledge to create reusable components that can be assembled into interactive web applications.

5.1. Component Basics

Components are an essential part of Vue.js, offering a way to build complex, reusable pieces of an application. They allow developers to encapsulate functionality, structure, and style, which makes development efficient and organized.

❖ 5.1.1. Why Components are Necessary

Components are necessary because they help break down a web application into smaller, manageable pieces, making it easier to develop, test, and maintain. By creating self-contained units of functionality, components promote reusability and scalability.

❖ 5.1.2. Creating Components with Vue.js

Vue Single File Components (SFCs) group are made up of a `<template>` section and optional `<script>` and `<style>` sections. The `<template>` section contains HTML (along with special Vue attributes and dynamic values in double curly braces. The `<script>` section contains JavaScript. The `<style>` section contains CSS code.

Section Order Doesn't Matter

The order in which you put the three sections of an SFC doesn't matter, however once you decide on the order that works best for you, stick with it. Being consistent will make it easier for other developers to read your code.

Here's an example of a Vue SFC with all three sections:

```
<template>
  <div>
    <p>{{ props.message }}</p>
    <p>Count: {{ count }}</p>
    <p>Doubled Count: {{ doubledCount }}</p>
    <button class="button" @click="increment">Increment</button>
  </div>
</template>

<script setup>
import { ref, computed, onMounted } from 'vue';
const props = defineProps(['message']);
const count = ref(0);
const doubledCount = computed(() => count.value * 2);
function increment() {
  count.value++;
}
onMounted(() => {
  console.log('Component mounted!');
});
</script>

<style>
.button {
  width: 200px;
  height: 40px;
  border-radius: 4px;
}
</style>
```


Understanding the setup() Function and <script setup>

In Vue 3, the `setup()` function is a part of the Composition API that serves as the entry point for using composition features within a component. The `setup()` function is executed before a component is created and is used to define reactive properties, lifecycle hooks, and other logic that can be consumed by the component's template or integrated with other components.

Below is an example of using the `setup()` function within a Vue component.

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

export default {
    name: 'ExampleComponent',
    setup() {
        const count = ref(0);

        function increment() {
            count.value++;
        }

        return {
            count,
            increment
        };
    }
};
</script>
```


In this example, the `setup()` function creates a reactive variable `count` and a method `increment` that increases the count. Both of these are returned from the `setup()` function for use within the component's template.

The `<script setup>` syntax is a syntactic sugar added in Vue 3.2 that simplifies the use of the Composition API by allowing developers to use the setup logic directly inside a component's `<script>` block, without needing to explicitly define a `setup()` function. This format reduces boilerplate code and enhances readability. In the following example, we've rewritten the previous example to use the `<script setup>` syntax:

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

const count = ref(0);
function increment() {
  count.value++;
}

};
</script>
```

Because it requires so much less code to accomplish the same result as the full `setup()` function syntax, the `<script setup>` syntax is preferred by most Vue.js developers.

❖ 5.1.3. Scoped Styles

By default, styles inside a `<style>` block in a Vue Single File Component (SFC) are **global** — meaning they apply to the entire app, not just the component where they're written. This can lead to style conflicts when multiple components use the same class names.

To prevent this, Vue provides **scoped styles**. When you add the `scoped` attribute to a `<style>` block, the styles inside it will only apply to the elements in that component. Vue achieves this by automatically adding unique data attributes to elements and matching CSS selectors.

Scoped styles are useful because they:

- Avoid style conflicts between components.
- Make components self-contained and reusable.
- Ensure that changes in one component's styles don't unexpectedly affect others.

Example: Global vs Scoped Styles

Let's look at two components that use the same class name. Notice how scoped styles keep the look of each component independent.

1. Go to the Vue SFC Playground: `https://play.vuejs.org`.
2. Create a new file **RedBox.vue** and paste this code:

Demo 5.1: Components/Demos/RedBox.vue

```
<template>
  <div class="box">I am the RedBox component</div>
</template>

<style scoped>
.box {
  background-color: #f88;
  padding: 1rem;
  border-radius: 6px;
}
</style>
```

This component has a `.box` class styled with a red background, scoped to itself.

3. Create another new file **GreenBox.vue** and paste this code:

Demo 5.2: Components/Demos/GreenBox.vue

```
<template>
  <div class="box">This is the GreenBox component.</div>
</template>

<style scoped>
.box {
  background-color: #8f8;
  padding: 1rem;
  border-radius: 6px;
}
</style>
```

Even though this component also uses the `.box` class, its styles are scoped, so they don't interfere with the `RedBox` styles.

4. Create a new file **BlueBox.vue** and paste this code:

Demo 5.3: Components/Demos/BlueBox.vue

```
<template>
  <div class="box">I am the BlueBox component</div>
</template>

<style scoped>
.box {
  background-color: #88f;
  padding: 1rem;
  border-radius: 6px;
}
</style>
```

5. Open **App.vue** and replace its contents with the following:

Demo 5.4: Components/Demos/ScopedStyles.vue

```
<script setup>
import RedBox from './RedBox.vue';
import GreenBox from './GreenBox.vue';
import BlueBox from './BlueBox.vue';
</script>

<template>
  <h2>Scoped Styles Demo</h2>
  <RedBox />
  <GreenBox />
  <BlueBox />
</template>
```

6. Click the "Reload page" button in the top right of the Playground. You should see one red box, one green box, and one blue box, each styled independently even though all three use the same `.box` class name.
7. Try removing the `scoped` attribute from the style element. How does that change the output?

5.2. Component Lifecycles

Lifecycle hooks allow developers to execute specific code at different stages of a component's lifecycle. These hooks provide powerful opportunities to perform "side effects" such as data fetching, manual DOM manipulation, or integrating third-party libraries.

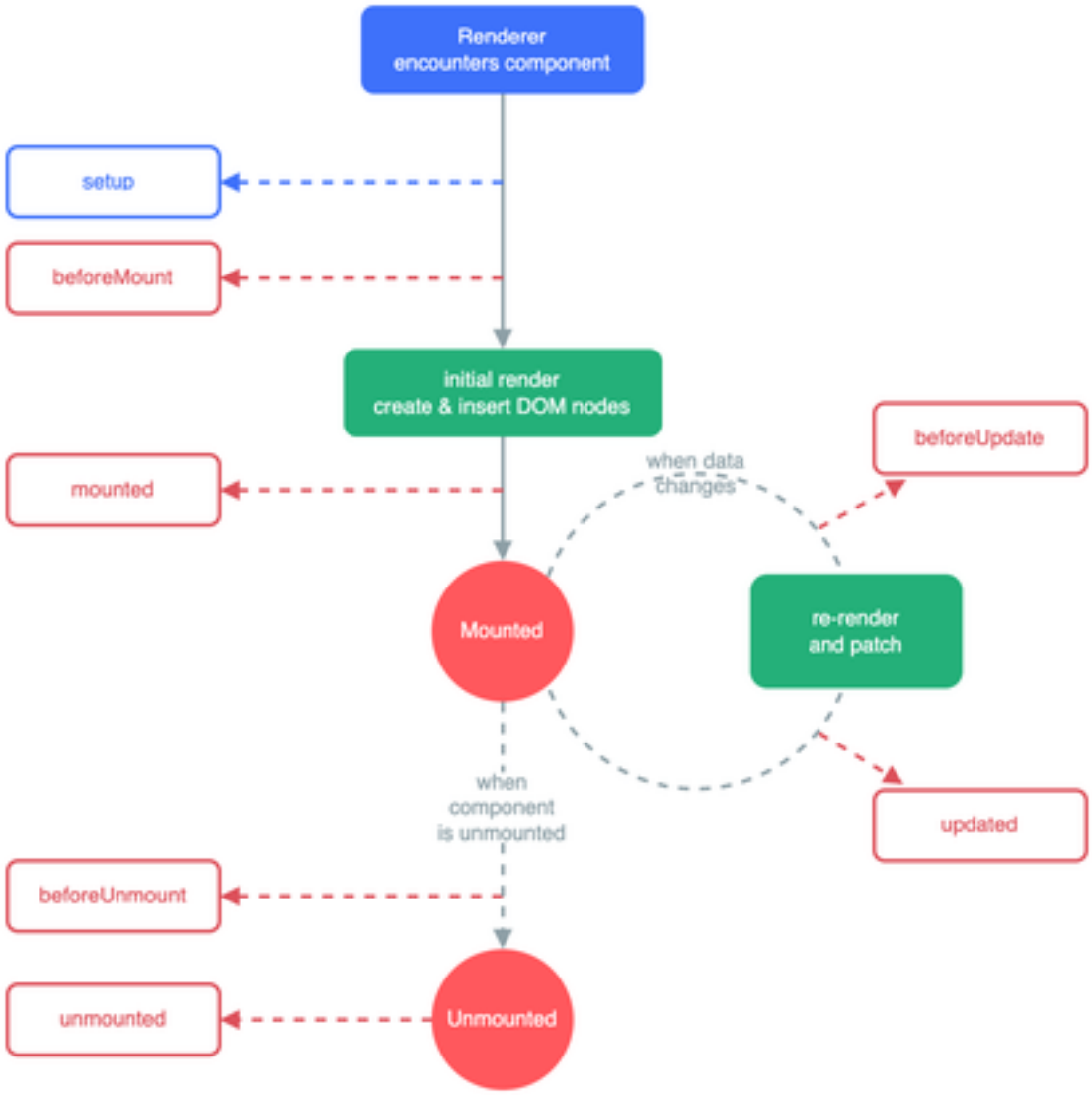
❖ 5.2.1. Understanding a Vue Component's Lifecycle Phases

Vue components go through several distinct phases during their lifecycle, allowing for controlled reaction to changes and rendering. Understanding these phases is key to managing component state and behavior effectively.

The lifecycle begins with the `setup()` function. This function is invoked before the component is fully created, allowing you to define reactive variables and functions in a concise manner. Next is the

`mount` phase, where the component is inserted into the DOM, rendering for the first time. Once a component is mounted, the `update` phase manages changes in the component's data or props, leading to re-renders. Finally, the `unmount` phase handles component removal from the DOM and associated cleanup.

Each phase presents an opportunity to perform specific operations, which can be done using functions called **lifecycle hooks**. The following diagram shows the lifecycle of a component, along with the hooks that run at every phase. In the next section, we'll talk about these hooks and how you can use them.



❖ 5.2.2. Main Lifecycle Hooks

Vue.js has 12 lifecycle hooks in addition to `setup`. Some of these are only useful in very specific cases, which has led to a subset of 6 hooks being referred to as the "main" lifecycle hooks. These are:

- `onBeforeMount()`
- `onMounted()`
- `onBeforeUpdate()`
- `onUpdated()`
- `onBeforeUnmount()`
- `onUnmounted()`

onBeforeMount()

This hook is called right before the component is mounted. It is not as commonly used as `onMounted` but can be useful in certain scenarios.


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

onBeforeMount(() => {
  console.log('Before Mounting');
});
</script>
```

onMounted()

The `onMounted()` hook is called after the component is mounted. It's often used for side effects such as API calls or DOM manipulations.

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

onMounted(() => {
  console.log('Component Mounted');
  // Side effects, like fetching data or manipulating DOM, go here.
});
</script>
```

onBeforeUpdate()

This hook is called right before the component is updated. It allows you to perform actions before the update happens.

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

onBeforeUpdate(() => {
  console.log('Before Update');
});
</script>
```

onUpdated()

Called after the component's reactive state has been updated. This is where you can act on updates.

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

onUpdated(() => {
  console.log('Component Updated');
});
</script>
```


onBeforeUnmount()

This hook is triggered right before a component is unmounted and destroyed. It can be used to perform any teardown that needs to happen.

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

onBeforeUnmount(() => {
  console.log('Before Unmount');
});
</script>
```

onUnmounted()

Called when the component is unmounted. It's ideal for resource cleanup like clearing timers or stopping subscriptions.

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

onUnmounted(() => {
  console.log('Component Unmounted');
  // Cleanup activities, such as stopping intervals or subscriptions, go here.
});
</script>
```


❖ 5.2.3. Observing Lifecycle Hooks in the Vue SFC Playground

This demo illustrates all six main lifecycle hooks, logging in the DevTools Console when they run. Follow these steps to view the demo in the Vue SFC Playground:

1. Go to the **Vue SFC Playground** at `https://play.vuejs.org`.

2. Replay any existing code in **App.vue** with the following:

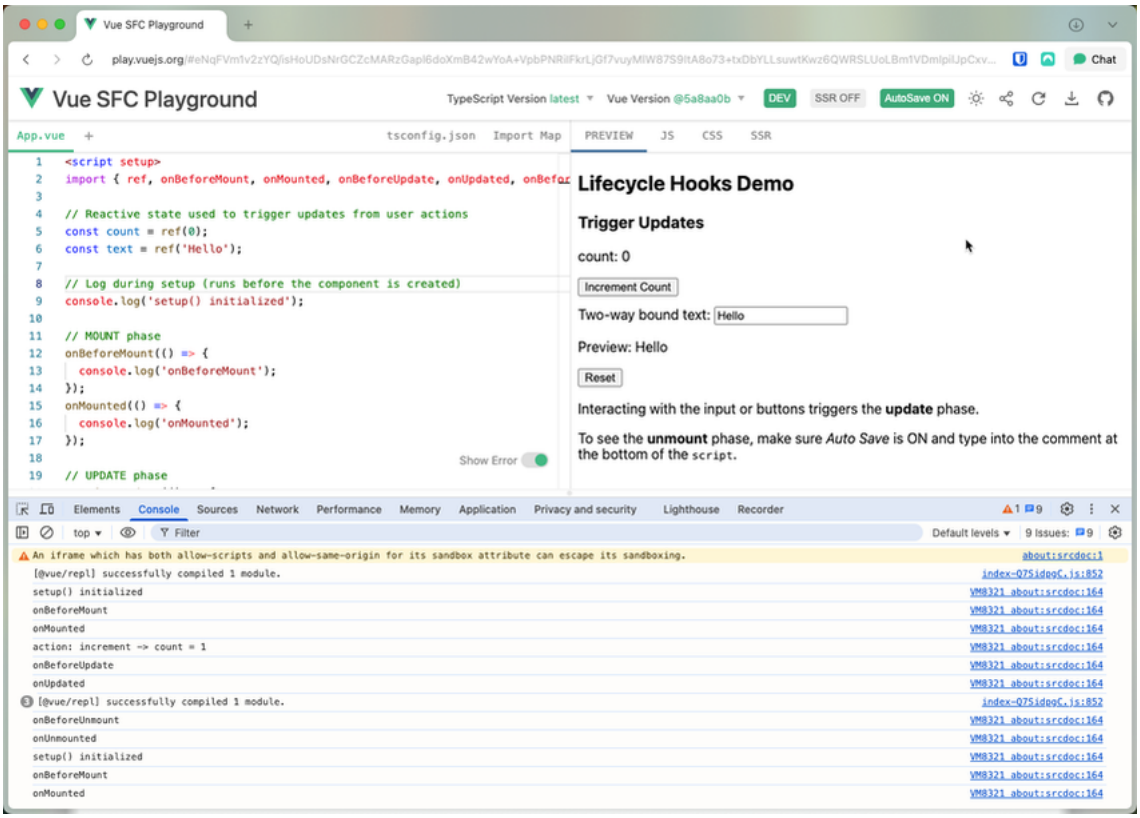
Demo 5.5: Components/Demos/lifecycle.vue

```
1.  <script setup>
2.  import { ref, onBeforeMount, onMounted, onBeforeUpdate, onUpdated, onBeforeUn ↵↵
            mount, onUnmounted } from 'vue';
3.
4.  // Reactive state used to trigger updates from user actions
5.  const count = ref(0);
6.  const text = ref('Hello');
7.
8.  // Log during setup (runs before the component is created)
9.  console.log('setup() initialized');
10.
11. // MOUNT phase
12. onBeforeMount(() => {
13.   console.log('onBeforeMount');
14. });
15. onMounted(() => {
16.   console.log('onMounted');
17. });
18.
19. // UPDATE phase
20. onBeforeUpdate(() => {
21.   console.log('onBeforeUpdate');
22. });
23. onUpdated(() => {
24.   console.log('onUpdated');
25. });
26.
27. // UNMOUNT phase (visible when you edit any code in the script with Auto Save
            ON)
28. onBeforeUnmount(() => {
29.   console.log('onBeforeUnmount');
30. });
31. onUnmounted(() => {
32.   console.log('onUnmounted');
33. });
34.
35. /*
36.   TYPE HERE TO TRIGGER UNMOUNT PHASE (with AutoSave ON):
37.   e.g. add or delete any character in this comment.
38.
39.   Why this works:
40.   - In the Vue SFC Playground, editing the script in App.vue with Auto Save ON
            triggers Hot Module Replacement (HMR).
41.   - HMR unmounts the current component instance, then mounts a new one.
42.   - That fires: onBeforeUnmount -> onUnmounted -> onBeforeMount -> onMounted.
```

Demo 5.5: Components/Demos/lifecycle.vue

```
43.  */
44.  </script>
45.
46.  <template>
47.    <h2>Lifecycle Hooks Demo</h2>
48.
49.    <section>
50.      <h3>Trigger Updates</h3>
51.
52.      <p>count: {{ count }}</p>
53.      <button @click="count++; console.log(`action: increment -> count =
           ${count}`)">
54.        Increment Count
55.      </button>
56.
57.      <div style="margin-top: 0.75rem">
58.        <label>
59.          Two-way bound text:
60.          <input v-model="text" placeholder="Type to trigger update"
61.            @input="console.log(`action: input -> text = ${text}`)"
62.          />
63.        </label>
64.        <p>Preview: {{ text }}</p>
65.      </div>
66.
67.      <button @click="count = 0; text = 'Hello'; console.log('action: reset
           state');">
68.        Reset
69.      </button>
70.
71.     <p>Interacting with the input or buttons triggers the <strong>update</strong>
           phase.</p>
72.      <p>
73.        To see the <strong>unmount</strong> phase, make sure <em>Auto Save</em>
           is ON and type into the
74.        comment at the bottom of the <code>script</code>.
75.      </p>
76.    </section>
77.  </template>
```


3. Open the **DevTools Console** by right-clicking in the preview window and selecting **Inspect**.
4. Go to the **Console** tab in the DevTools to see the messages that are logged by the demo.



You may see several error messages and warnings. These are related to the Vue SFC Playground, not your code, so they are nothing to worry about.

5. Click the "Reload page" button (top right of the playground) You should see: `setup() initialized`, `onBeforeMount`, and `onMounted` in the console.
6. Interact and observe the console
    A. Click **Increment Count** or type in the input to trigger: `onBeforeUpdate` then `onUpdated`.
    B. Click **Reset** to trigger another update cycle.
    C. Type anywhere within the `script` to trigger an unmount and an immediate re-mount. Make sure AutoSave is ON.

5.3. Props

Props are how you pass data from a parent component to a child component is Vue. Using props in Vue's Composition API is straightforward and enables the separation of data handling and presentation logic.

❖ 5.3.1. Defining and Using Props

In Vue, props are passed from parent to child using attributes, and they're declared in the child component using the `defineProps` helper function. Let's take a look at an example of passing and using props.

❖ 5.3.2. Passing Props to Child Components

To pass props to child components, use attribute names. In the following example, the user-name and message-count attributes will pass their values to the `MessageCard` component. In the following example, the `App` component passes the values of `userName` and `userMessageCount` as props.

```
<!-- App.vue -->

<template>
  <MessageCard user-name="Alice" message-count="5" />
</template>
```

Here's what the `MessageCard` component might look like:

```
<!-- MessageCard.vue -->

<template>
  <div class="message-card">
    <h3>Welcome, {{ userName }}!</h3>
    <p>You have {{ messageCount }} messages.</p>
  </div>
</template>

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

// Define props with static typing
const props = defineProps(['userName','messageCount']);
</script>
```

In this `MessageCard` example, the props `userName` and `messageCount` are defined and utilized directly in the template to display static data.

Note that it is conventional to use kebab-case for prop names in the template (e.g., `user-name`, `message-count`) and camelCase in the script when defining/using them (`userName`, `messageCount`).

❖ 5.3.3. Prop Validation

Props can be validated for data types and specific constraints using an object syntax with `defineProps`. Here's an example of validating props:

```
<script setup>
const props = defineProps({
  userName: {
    type: String,
    required: true
  },
  messageCount: {
    type: Number,
    default: 0,
    validator: (value) => value >= 0
  }
});
</script>
```

This setup ensures `userName` is a required string, and `messageCount` is a non-negative number with a default value of 0.

❖ 5.3.4. Static vs. Dynamic Props

Props in Vue can be either static or dynamic. Static props are hard-coded values, while dynamic props reflect changes in the parent component's data.

Static Props

```
<UserProfile user-name="John Doe" />
```

Here `user-name` is a static prop with the value "John Doe".

Dynamic Props


```
<UserProfile :user-name="dynamicUserName" />
```

As you learned in the lesson about Vue reactivity, the colon (`:`) before `user-name` is shorthand for Vue's `v-bind` directive. This shorthand allows you to bind a data property from the parent component to the prop in the child component. Using `v-bind` enables reactive updates; as the value of `dynamicUserName` changes in the parent, the child component will automatically update to reflect these changes.

❖ 5.3.5. Passing and Using props Demo

In this demo, you'll render a list of articles by passing data from a parent component to a child component using props. Follow these steps to see the demo in the Vue SFC Playground:

1. Go to the Vue SFC Playground at `https://play.vuejs.org`.

2. Replay any existing code in **App.vue** with the following:

Demo 5.6: Components/Demos/PassPropsApp.vue

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

// Parent state
const items = ref([
  {
    id: 1,
    title: 'Learn Vue Props',
    description: 'Understand how data flows from parent to child.',
    isFeatured: true,
  },
  {
    id: 2,
    title: 'Use Child Components',
    description: 'Break your app into smaller, reusable pieces.',
    isFeatured: false,
  },
  {
    id: 3,
    title: 'Practice Often',
    description: 'Build small demos to reinforce learning.',
    isFeatured: false,
  },
]);
</script>

<template>
  <h2>Props Demo</h2>
  <p>
    Below, the parent (<code>App.vue</code>) passes props to the child
    (<code>ChildCard.vue</code>).
  </p>

  <section>
    <ChildComponent
      v-for="item in items"
      :key="item.id"
      :title="item.title"
      :description="item.description"
      :isFeatured="item.isFeatured"
    />
  </section>
</template>
```

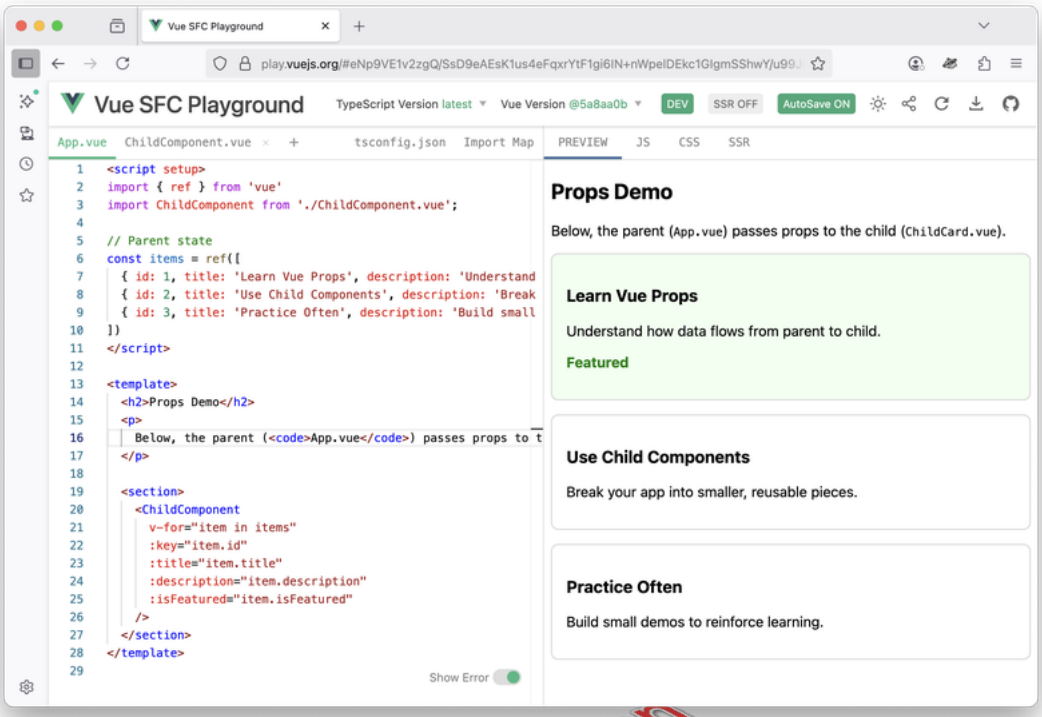
3. Click the + sign to the right of the App.vue tab to create a new component. Name it ChildComponent.vue.

4. Replace the default code in the new ChildComponent.vue component with the following:

Demo 5.7: Components/Demos/ChildComponent.vue

```
<script setup>
// Define props for this component
defineProps({
  title: String,
  description: String,
  isFeatured: Boolean,
});
</script>

<template>
  <div
    :class="{ featured: isFeatured }"
    style="
      border: 1px solid #ccc;
      padding: 1rem;
      border-radius: 8px;
      margin-bottom: 1rem;
    "
  >
    <h3>{{ title }}</h3>
    <p>{{ description }}</p>
    <p v-if="isFeatured" style="color: green; font-weight: bold">Featured</p>
  </div>
</template>

<style scoped>
.featured {
  background-color: #f0fff0;
}
</style>
```


5. Look at the rendered components in the preview pane. You should see a list of three articles with each one populated with the data from the items array.



5.4. Component events

Component events in Vue.js provide a mechanism for communication between components, particularly from child to parent. Custom events enable a child component to send messages to its parent, often indicating a change or action that has occurred.

❖ 5.4.1. Creating Custom Events

Custom events are created in the child component and emitted using the `emit()` method. To define custom events, pass an array of the custom event names you wish to define to the `defineEmits` method. Once you've done that, you can use the `emit()` method to emit the new events, as in the following example:

```
<template>
  <button @click="notifyParent">Click me</button>
</template>

<script setup>
import { defineEmits } from 'vue';

// Declare the 'boop' event
const emit = defineEmits(['boop']);

// Emit 'boop' event when button is clicked
function notifyParent() {
  emit('boop');
}
</script>
```

❖ 5.4.2. Listening for Events

The parent component can listen for events emitted from a child by using the `v-on` directive (or its shorthand, `@`), the same way that a component listens for standard browser events . Here's how the parent can listen to the 'boop' event emitted by the child.


```
<template>
  <child-component @boop="handleBoop" />
</template>

<script setup>
// Handle child component's 'boop' event
function handleBoop() {
  alert('Boop!');
}
</script>
```

❖ 5.4.3. Passing Additional Arguments

Additional data can be passed as arguments when the event is emitted. To send data along with the event name when emitting a message, pass the data as the 2nd argument to the `emit()` method, like this:

```
<template>
  <button @click="notifyParent">Send message</button>
</template>

<script setup>
import { defineEmits } from 'vue';

const emit = defineEmits(['message']);
const message = "Hello, parent!";

function notifyParent() {
  emit('message', message);
}
</script>
```

To access the data in the parent, just specify a parameter in the event handler function, like this:


```
<template>
  <child-component @message="receiveMessage" />
</template>

<script setup>
// Function to handle received message
function receiveMessage(message) {
  alert('Message from child: ' + message);
}
</script>
```

❖ 5.4.4. Validating Events

Custom event validation in Vue can be done using the `defineEmits()` method. Event validation helps ensure that event data follows expected formats or conditions. Here is a simple example of validating an emitted event value.

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

// Validate 'update' event payload
const emit = defineEmits({
  update: (value) => {
    // check that the 'update' value is a number
    if (typeof value === 'number') {
      return true;
    } else {
      console.warn('Invalid data type for update event');
      return false;
    }
  }
});

function updateValue(value) {
  emit('update', value);
}
</script>
```


❖ 5.4.5. Using v-model for Custom Components

`v-model` can be used with custom Vue components to provide the same streamlined two-way binding as native inputs, letting the parent read and update a component's value with minimal boilerplate.

To work, the component must:

- Accept a `modelValue` **prop**.
- Emit an `update:modelValue` **event** when the value changes.

Custom Component Basic Example

```
<!-- Child Component: CustomComponent.vue -->
<template>
  <input :value="modelValue" @input="$emit('update:modelValue', $event.target.value)"
/>
</template>

<script setup>
defineProps({
  modelValue: String,
});
</script>
```

Users of the component can then use `v-model` as they would with native form elements:


```
<!-- Parent Component -->
<CustomComponent v-model="customValue" />
```

Here, `customValue` will always stay in sync with the `<input>` inside the child.

Named v-model Bindings

Sometimes a component needs to expose more than one value. In that case, you can name each `v-model` binding be appending a suffix with a colon.

```
<!-- Parent Component -->
<CustomComponent v-model:firstValue="customValue1" v-model:secondValue="customValue2"
/>
```

`CustomComponent.vue` could look something like:

```
<!-- Child Component: CustomComponent.vue -->
<template>
  <input :value="modelValue"
    @input="$emit('update:firstValue', $event.target.value)" />
  <select :value="modelValue2"
    @change="$emit('update:secondValue', $event.target.value)">
    <option value="foo">Foo</option>
    <option value="bar">Bar</option>
  </select>
</template>

<script setup>
defineProps({
  firstValue: String,
  secondValue: String,
});
</script>
```

The key takeaway here is:

- **By default**, `v-model` uses `modelValue` / `update:modelValue`.
- **Named**`v-model` instances use `v-model:name` in the parent, with `name` as the prop and `update:name` as the event in the child.

❖ 5.4.6. Emitting and Using Custom Events Demo

1. Go to the Vue SFC Playground at `https://play.vuejs.or` (`https://play.vuejs.org`)g.

2. Replace any existing code in **App.vue** with the following:

Demo 5.8: Components/Demos/CustomEventsDemo.vue

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

const logMessages = ref([]);

function log(message) {
  const stamp = new Date().toLocaleTimeString();
  logMessages.value.push(`[${stamp}] ${message}`);
}

function handleBoop() {
  log('Parent received: boop event');
}

function handleMessage(msg) {
  log(`Parent received: message event with payload "${msg}"`);
}
</script>

<template>
  <h2>Custom Events Demo</h2>
  <p>The parent listens for events emitted by the child.</p>

  <CustomEventChild @boop="handleBoop" @message="handleMessage" />

  <section style="margin-top: 1.5rem">
    <h3>Event Log</h3>
    <p>Each emitted event appears below:</p>
    <ul>
      <li v-for="(m, i) in logMessages" :key="i">{{ m }}</li>
    </ul>
  </section>
</template>
```

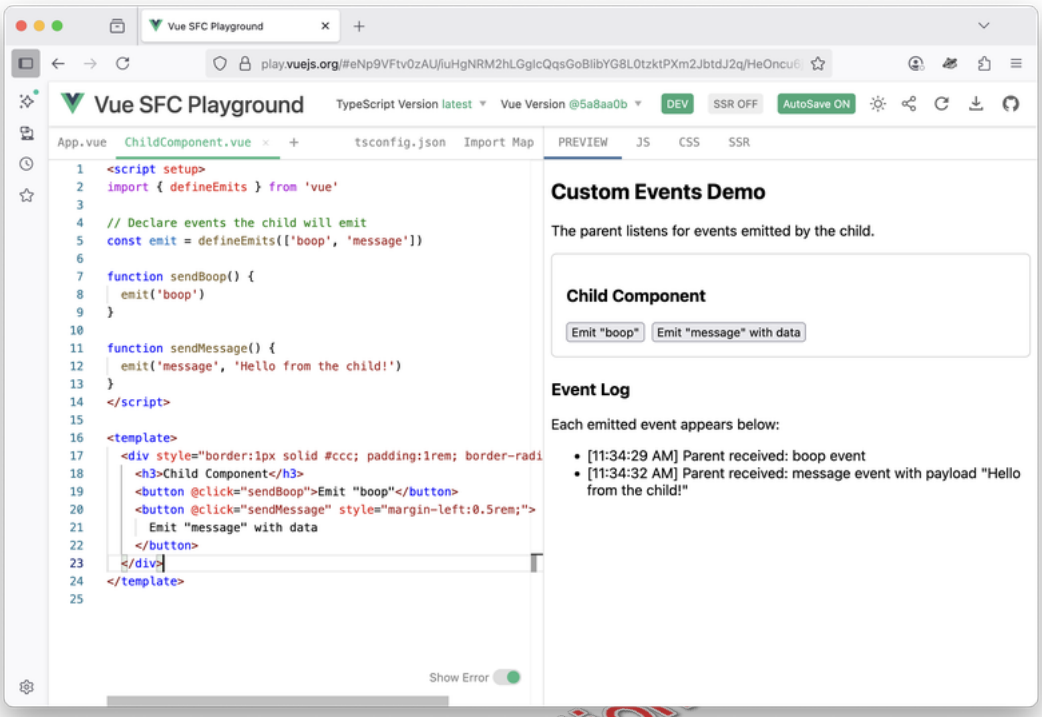

3. Click the **+** sign to the left of the **App.vue** tab to create another component. Name it `CustomEventChild.vue`.

4. Replace the content of the `CustomEventChild.vue` component with the following:

Demo 5.9: Components/Demos/CustomEventChild.vue

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

// Declare events the child will emit
const emit = defineEmits(['boop', 'message']);

function sendBoop() {
  emit('boop');
}

function sendMessage() {
  emit('message', 'Hello from the child!');
}
</script>

<template>
  <div style="border: 1px solid #ccc; padding: 1rem; border-radius: 6px">
    <h3>Child Component</h3>
    <button @click="sendBoop">Emit "boop"</button>
    <button @click="sendMessage" style="margin-left: 0.5rem">
      Emit "message" with data
    </button>
  </div>
</template>
```


5. Try clicking the buttons in the preview pane to emit events and handle them. Watch the messages that are logged to the screen.



6. As a challenge, try creating a new component `CustomModelComponent.vue` and getting `v-model` to work with it!

5.5. Composables

As apps grow, different components often need the same logic (e.g., fetching data, form handling, timers, feature flags, etc.). Copy-pasting that logic into each component can lead to:

- **Duplication** (bugs must be fixed everywhere)
- **Messy components** (hard to read, hard to test)
- **Inconsistent behavior** (slightly different implementations of the same behavior in every component)

Composables let you move that shared logic into a plain JavaScript function and reuse it anywhere in any component. They help keep your components lighter and important logic in clean, testable place.

❖ 5.5.1. What is a Composable?

Composables are plain JavaScript functions (typically named with a `use` prefix) that uses Vue's Composition API to create and return reactive state and helpers.

- It is a function (e.g., `useCounter()`) that runs inside a component's `<script setup>`.
- It can create reactive values with `ref`/`reactive`, derive data with `computed`, and even register lifecycle hooks like `onMounted`.
- It returns whatever the component should use (state, methods, computed values).

Example: a simple toggle tool


```
// composables/useToggle.js
import { ref } from 'vue';

export function useToggle(initial = false) {
  const isOn = ref(initial);
  const toggle = () => (isOn.value = !isOn.value);
  const on = () => (isOn.value = true);
  const off = () => (isOn.value = false);

  return { isOn, toggle, on, off };
}
```

To use a composable in a component, simply import it and call it in the `<script setup>`. It is common for composables to return objects and then to use `object destructuring (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring#destructuring_primitive_values)` to get only the specific state, methods, etc. that you need for the component where you are using the composable.

```
<template>
  <button @click="toggle">{{ isOn ? 'On' : 'Off' }}</button>
</template>

<script setup>
import { useToggle } from './composables/useToggle.js'
const { isOn, toggle } = useToggle(false)
</script>
```

❖ 5.5.2. Lifecycle Hooks Inside Composables

You can use lifecycle hooks inside composables. They will run in the context of the component that called the composable.

```
// composables/useInterval.js
import { onMounted, onUnmounted } from 'vue';

export function useInterval(callback, delay = 1000) {
  let id = null;

  onMounted(() => {
    id = setInterval(callback, delay);
  });

  onUnmounted(() => {
    if (id) clearInterval(id);
  });
}
```


❖ 5.5.3. Per-Instance vs. Shared State

You can choose whether your composable provides state that is isolated per-instance or shared across everywhere it is used.

Per-instance (default): state lives inside the function, so every call creates a new instance of the state.

```
// Per-instance counter (each component gets its own counter)
import { ref } from 'vue';
export function useCounter(initial = 0) {
  const count = ref(initial);
  const inc = () => count.value++;
  return { count, inc };
}
```

Shared (global-ish): state lives at module top-level, so everyone reads/writes the same `ref`.

```
// Shared counter (all components share the same ref)
import { ref } from 'vue';
const shared = ref(0);
export function useSharedCounter() {
  const inc = () => shared.value++;
  return { count: shared, inc };
}
```


Use per-instance when you are sharing similar behavior between components (most cases). Use shared when you want to give multiple components the ability to update the same state.

❖ 5.5.4. Configurable Composables

Composables often accept options to tailor behavior. The following composable accepts an options object to configure behavior: it reads `initial`, `step`, and `autoStart` with defaults, creates a `ref` for `count` seeded from `initial`, and a `computed` value `doubled`. It provides `inc`, `dec`, and `reset` to change the state using `step`. For automation, `startAuto` sets an interval that calls `inc` at the given period after clearing any previous timer, and `stopAuto` cancels it. If `autoStart` is true, the interval

is started via `startAuto` in `onMounted`. The interval is cleaned up in `onUnmounted`. The returned object exposes the reactive state and controls for use in the template.

Demo 5.10: Components/Demos/useCounter.js

```
1.  // composables/useCounter.js
2.  import { ref, computed, onMounted, onUnmounted } from 'vue';
3.
4.  export function useCounter(options = {}) {
5.    const { initial = 0, step = 1, autoStart = true } = options;
6.
7.    const count = ref(initial);
8.    const doubled = computed(() => count.value * 2);
9.
10.   let id = null; // id of the interval
11.   const inc = () => (count.value += step); // increment the count by {step}
12.   const dec = () => (count.value -= step); // decrement the count by {step}
13.   const reset = () => (count.value = initial); // reset the count to {initial}
14.
15.   /**
16.    * Start automatically incrementing
17.    * the count every {interval} milliseconds.
18.    */
19.   function startAuto(interval = 1000) {
20.     stopAuto();
21.     id = setInterval(inc, interval);
22.   }
23.   /**
24.    * Stop the interval from incrementing the count.
25.    */
26.   function stopAuto() {
27.     if (id) {
28.       clearInterval(id);
29.       id = null;
30.     }
31.   }
32.
33.   onMounted(() => {
34.     // start auto incrementing on mounted if autoStart is set
35.     if (autoStart) {
36.       startAuto();
37.     }
38.   });
39.
40.   onUnmounted(() => {
41.     // stop auto incrementing on unmounted
42.     stopAuto();
43.   });
44.
```

Demo 5.10: Components/Demos/useCounter.js

```
45.     return { count, doubled, inc, dec, reset, startAuto, stopAuto };
46.   }
```

Best Practices

- Name composables with a `use` prefix (e.g., `useFetch`, `useForm`).
- Accept configuration via parameters instead of relying on component globals.
- Return a stable, documented object shape so consumers know what to expect.
- Clean up side effects in lifecycle hooks (e.g., clear intervals, abort requests).
- Avoid doing heavy work at import time; do it inside the composable or on demand.

❖ 5.5.5. Using Composables in the Vue SFC Playground

In this demo, you will create a reusable counter composable and use it in multiple components to see both reusability and per-instance state in action.

1. Open the Vue SFC Playground at `https://play.vuejs.org`.
2. Create a new file named **useCounter.js**. Click the plus (**+**) sign next to the **App.vue** tab and paste in the code from the above `useCounter.js` demo.

3. Create another new file named **CounterPanel.vue** and paste the code below.

Demo 5.11: Components/Demos/CounterPanel.vue

```
<template>
  <div class="panel">
    <h3>{{ title }}</h3>
    <p>
      Count: <strong>{{ count }}</strong> (x2 = {{ doubled }})
    </p>

    <div class="controls">
      <button @click="dec">-</button>
      <button @click="inc">+</button>
      <button @click="reset">Reset</button>
      <button @click="startAuto(500)">Auto +</button>
      <button @click="stopAuto">Stop</button>
    </div>
  </div>
</template>

<script setup>
import { defineProps } from 'vue';
import { useCounter } from './useCounter.js';

const props = defineProps({
  title: { type: String, default: 'Counter' },
  initial: { type: Number, default: 0 },
  step: { type: Number, default: 1 },
  autoStart: { type: Boolean, default: true },
});

const { count, doubled, inc, dec, reset, startAuto, stopAuto } = useCounter({
  initial: props.initial,
  step: props.step,
  autoStart: props.autoStart,
});
</script>

<style>
.panel {
  border: 1px solid #ddd;
  border-radius: 8px;
  padding: 12px;
  margin: 8px 0;
}
.controls button {
  margin-right: 6px;
```

Demo 5.11: Components/Demos/CounterPanel.vue

```
45.     padding: 6px 10px;
46.   }
47.   </style>
```

4. Replace the contents of **App.vue** with the following to render multiple counters using the same composable:

Demo 5.12: Components/Demos/ComposableApp.vue

```
1.    <script setup>
2.    import CounterPanel from './CounterPanel.vue';
3.    import { useCounter } from './useCounter.js';
4.
5.    const { count, inc, dec, reset } = useCounter({
6.      step: 1,
7.      initial: 10,
8.      autoStart: false
9.    });
10.
11.   </script>
12.
13.   <template>
14.     <div class="app">
15.       <h2>Composable Demo: useCounter</h2>
16.       <p>
17.         <strong>{{ count }}</strong> is a counter that you can
18.         <button @click="inc">increment</button>,
19.         <button @click="dec">decrement</button>, and
20.         <button @click="reset">reset</button>.
21.       </p>
22.       <CounterPanel title="Auto Counter" :initial="0" :step="1" :auto-start="true"
              />
23.       <CounterPanel title="Bigger Auto Counter" :initial="100" :step="5" :auto-
              start="true" />
24.     </div>
25.   </template>
```

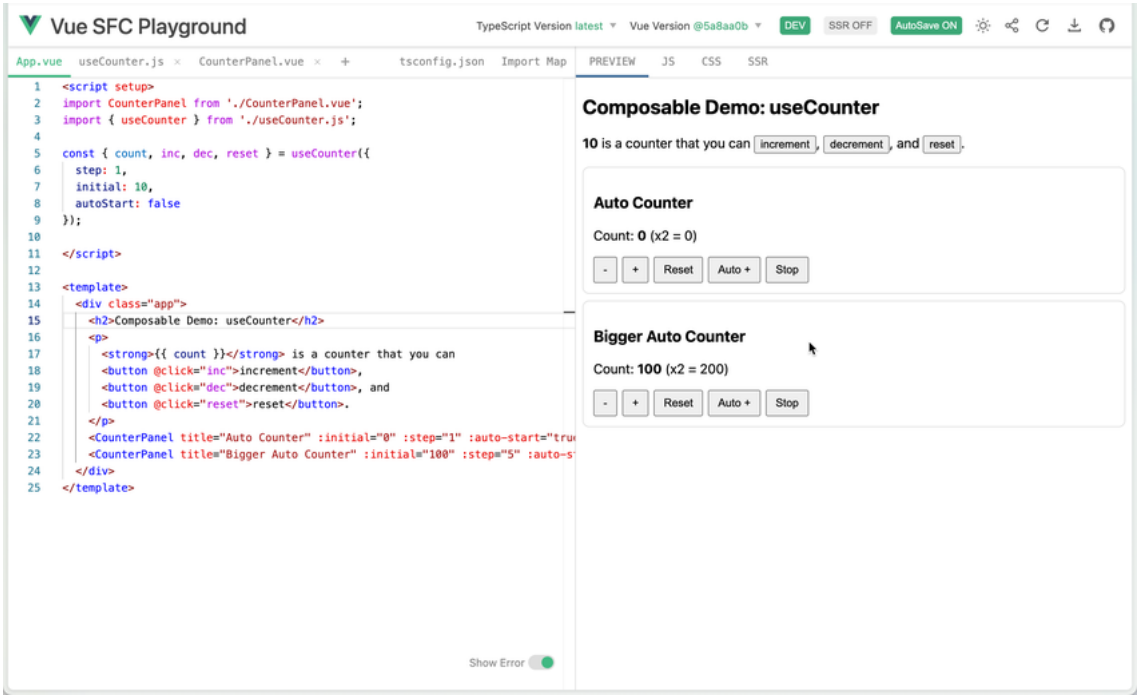

5. **Preview the app.** Notice:

   A. That we use the `useCounter` composable in both the `CounterPanel` component and in **App.vue**.

   B. We are able to reuse the composable while still customizing its exact behavior in each component. For instance, in App.vue, we don't allow the use of the auto-incrementing

behavior. The two instances of `CounterPanel` also have different `initial` values and `step` values.

C. Each instance of `useCounter` (in **App.vue** and each `CounterPanel` instance) creates independent state.



5.6. Advanced Component Features: slots and dynamic components

This activity focuses on enhancing the flexibility and reusability of your Vue.js components using slots and dynamic components.

❖ 5.6.1. Understanding Slots

Slots allow you to insert custom content into components, providing a way to create component templates that can be populated with dynamic content.

Basic Slots

Basic slots let you insert content into a predefined location within a component. To create a slot, use the `<slot>` component and (optionally) provide content between its start and end tags that will be used when no content is passed to the slot, like this:

```
<template>
  <div class="container">
    <slot>Default Content</slot>
  </div>
</template>
```

In your parent component, any content between a component with a slot's start and end tags will be passed to that child element and rendered in the slot. The following example will cause the words "Custom Content" to be rendered inside the `<slot>` of the `MyComponent` component.

```
<MyComponent>
  <p>Custom Content</p>
</MyComponent>
```


To test out using slots, go to the Vue SFC Playground (`https://play.vuejs.org`) and create a new component by clicking the plus symbol above the left-hand editor. Name the new component `BigBoldText.vue` and copy the following code to the new component.

Demo 5.13: Components/Demos/BigBoldText.vue

```
1.   <template>
2.     <div class="big-bold">
3.       <slot>Default Content</slot>
4.     </div>
5.   </template>
6.   <style scoped>
7.    .big-bold {
8.     font-size: 48px;
9.     font-weight: bold;
10.   }
11.  </style>
```

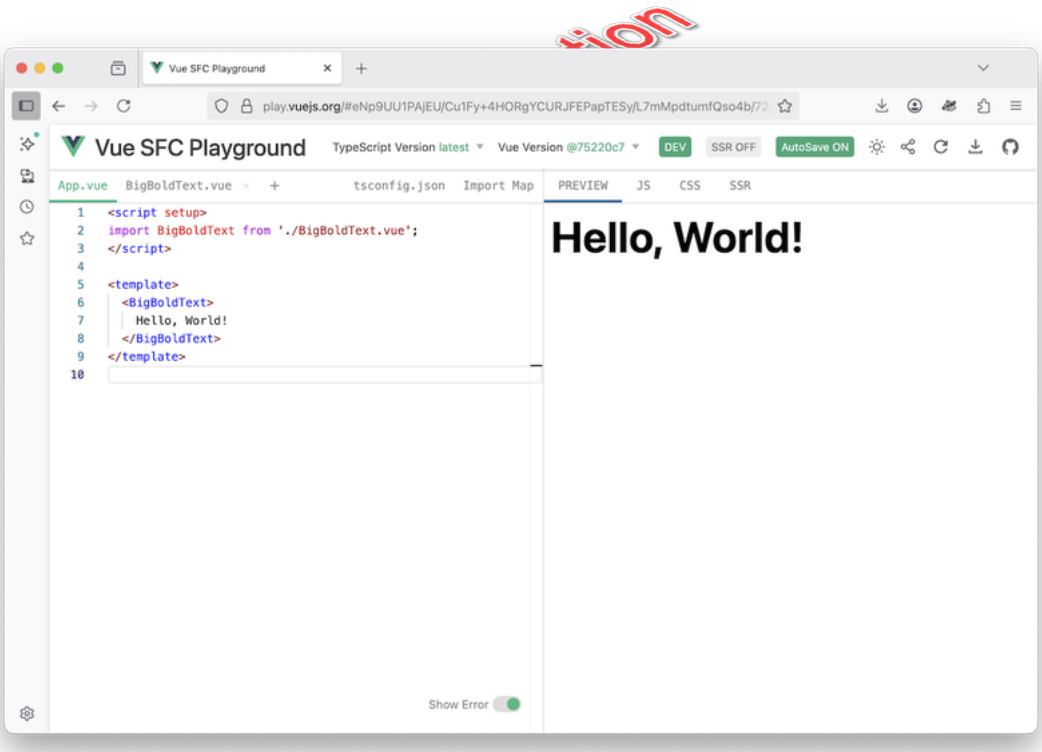
In **App.vue**, import the new component and render an instance of it in the `<template>`, using a beginning and ending tags (as in the example above). The content you put between the start and end tags will render in the child component's `<slot>` element.

You can copy this code for `App.vue`:

Demo 5.14: Components/Demos/App.vue

```
<script setup>
import BigBoldText from './BigBoldText.vue';
</script>

<template>
  <BigBoldText>
    Hello, World!
  </BigBoldText>
</template>
```

When you preview the app in the Vue SFC Playground, you'll see that any text you put between the start and end tags of the `<BigBoldText>` element will be rendered using the styles in the BigBoldText component.



❖ 5.6.2. Named Slots

Named slots enable the use of multiple slots within the same component, each identified with a unique name.

```
<template>
  <div class="wrapper">
    <header>
      <slot name="header">Default Header</slot>
    </header>
    <section>
      <slot>Default Content</slot>
    </section>
  </div>
</template>
```

The parent component assigns content to named slots as follows:


```
<MyComponent>
  <template v-slot:header>
    <h1>Custom Header</h1>
  </template>
  <p>Main Content</p>
</MyComponent>
```

In this demo, `<h1>Custom Header</h1>` will be placed into the `<header>` and `<p>Main content</p>` will be placed into the `<section>`.

❖ 5.6.3. Dynamic Components

Dynamic components allow you to render different components conditionally, using a single `<component>` element with the `is` attribute. This feature is useful for situations where component structures need to change based on user interaction or external data.

Using Dynamic Components

```
<template>
  <component :is="componentNameMap[current]" />
</template>

<script setup>
import { ref } from 'vue';
import ComponentA from './ComponentA.vue';
import ComponentB from './ComponentB.vue';

// Map component string names to Components
const componentNameMap = { ComponentA, ComponentB };

// Current component name
const current = ref('ComponentA');
</script>
```

`<component :is="...">` renders whatever component you pass it. Here, `current` is a ref holding a string (`'ComponentA'` or `'ComponentB'`), and `componentNameMap` is a simple registry that maps those strings to the actual imported component objects. In JavaScript, if the key and variable name are the same, you can use shorthand object notation:


```
// The following are all equivalent to each other
{ ComponentA, ComponentB }
{ ComponentA: ComponentA, ComponentB: ComponentB }
{ 'ComponentA': ComponentA, 'ComponentB: ComponentB }
```

Passing `componentNameMap[current]` ensures `:is` receives a real component (not just a string), so switching current at runtime cleanly swaps what's rendered. The map also acts as a safe list of allowed components.

Dynamic Components Demo

Here's a complete example of dynamic component selection based on user interaction. Follow these steps:

1. Go to the Vue SFC Playground at `https://play.vuejs.org`.

2. Replace any existing code in `App.vue` with the following:

Demo 5.15: Components/Demos/MainComponent.vue

```
<script setup>
import { ref } from 'vue';
import RedBox from './RedBox.vue';
import GreenBox from './GreenBox.vue';
import BlueBox from './BlueBox.vue';

// Track the current component being displayed
const current = ref('RedBox');

// Map names to component objects
const components = { RedBox, GreenBox, BlueBox };
</script>

<template>
  <h2>Dynamic Components Demo</h2>
  <p>Click the buttons to switch which component is displayed below.</p>

  <div style="margin-bottom: 1rem">
    <button @click="current = 'RedBox'">Show Red</button>
    <button @click="current = 'GreenBox'">Show Green</button>
    <button @click="current = 'BlueBox'">Show Blue</button>
  </div>

  <!-- Dynamic component rendering -->
  <component :is="components[current]" />
</template>

<style scoped>
button {
  margin-right: 0.5rem;
  padding: 0.4rem 0.75rem;
  border-radius: 6px;
  border: 1px solid #ccc;
  cursor: pointer;
}
</style>
```


3. Click the + sign to the right of the `App.vue` tab to create a 2nd component. Name it `RedBox.vue`, and paste in the following code (unless you still have the `RedBox.vue`, `Green`

`Box.vue`, and `BlueBox.vue` components from the Component Basics activity, in which case you can use those):

Demo 5.16: Components/Demos/RedBox.vue

```
<template>
  <div class="box">I am the RedBox component</div>
</template>

<style scoped>
.box {
  background-color: #f88;
  padding: 1rem;
  border-radius: 6px;
}
</style>
```


4. Click the `+` sign to the right of the `RedBox.vue` tab and name the new component `GreenBox.vue`. Paste the following code into `GreenBox.vue`:

Demo 5.17: Components/Demos/GreenBox.vue

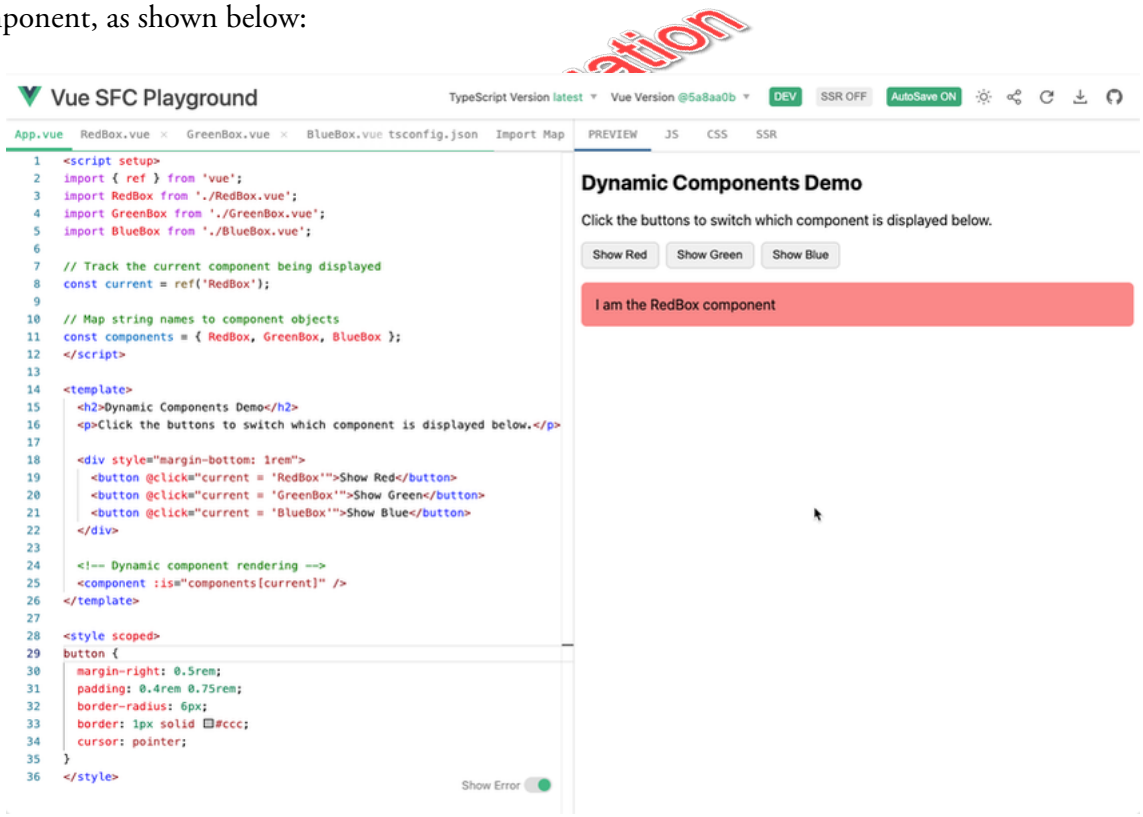
```
<template>
  <div class="box">This is the GreenBox component.</div>
</template>

<style scoped>
.box {
  background-color: #8f8;
  padding: 1rem;
  border-radius: 6px;
}
</style>
```

5. Click the + sign to the right of the `GreenBox.vue` tab and name the new component `BlueBox.vue`. Paste the following code into `BlueBox.vue`:

Demo 5.18: Components/Demos/BlueBox.vue

```
<template>
  <div class="box">I am the BlueBox component</div>
</template>

<style scoped>
.box {
  background-color: #88f;
  padding: 1rem;
  border-radius: 6px;
}
</style>
```

In this example, clicking the buttons will cause a different component to render into the dynamic component, as shown below:



Exercise 5: Breaking PetVue into Components

30 to 50 minutes

In this exercise, you will learn how to break down a large, complex Vue component into smaller, reusable pieces. You'll extract a `PetCard` component from the existing `HomeView.vue`, making your code more organized, maintainable, and reusable.

❖ E5.1. The Problem: HomeView.vue is Getting Too Large

Look at your current `HomeView.vue` file. Although it's not too complex or long at this point (at somewhere around 100 lines of code), it currently handles multiple responsibilities:

- Displaying the hero section
- Managing filter controls and logic
- Rendering individual pet cards

This violates the **Single Responsibility Principle** and makes the code harder to:

- Understand and maintain
- Test individual features
- Reuse in other parts of the application
- Collaborate on with other developers

❖ E5.2. Our Refactoring Plan

Going by the principle of single responsibility, `HomeView.vue` ideally should be broken into 3 or 4 components:

1. **HomeView.vue** (orchestrator) - Manages overall page structure and data flow.
2. **PetFilter.vue** (component) - Handles all filtering UI and logic.
3. **PetCard.vue** (component) - Handles individual pet display and interactions.
4. **HeroSection.vue** (component) - Since this is part of the Home view, and is unlikely to be used elsewhere, the case for this being its own component isn't as strong as for the others. But, once you learn how, extracting it won't be difficult to do.

In this exercise, we'll walk you through the process of extracting the `PetCard` component from `HomeView`. Extracting `PetFilter` and the `HeroSection` will use the same process, and we encourage you to try those on your own as a challenge exercise.

❖ E5.3. Step 1: Create the PetCard Component

1. Create a new file `src/components/PetCard.vue` and create the structure for a new component:

PetCard.vue

```
<template>

</template>

<script setup>

</script>

<style scoped>

</style>
```

2. Copy and paste the `<div>` with the `v-for` from the pet list section of the `HomeView.vue` template and paste it into the `<template>` of `PetCard.vue`:

PetCard.vue

```
<template>
  <div v-for="pet in filteredPets" :key="pet.id" class="pet-card">
    <img :src="pet.image" :alt="pet.name" />
    <div v-if="pet.adopted" class="adoption-overlay">
      <div class="adoption-message">
        <h6 class="fw-bold mb-1">ADOPTED!</h6>
        <small>Found their forever home</small>
      </div>
    </div>
    <div>
      <h5>{{ pet.name }}</h5>
      <p>{{ pet.breed }} &bull; {{ pet.gender }}</p>
      <p>{{ pet.size }} {{ pet.type }}</p>
    </div>
  </div>
</template>
```


3. The `PetCard.vue` component will be for a single pet-card, so we can get rid of the `v-for` code:

PetCard.vue

```
<template>
  <div class="pet-card">
    <!-- Same as above -->
  </div>
</template>
```

4. In the `<script>` block, define the `pet` prop that will be received by `PetCard` and mark it as required:

PetCard.vue

```
<script setup>
const props = defineProps({
  pet: {
    type: Object,
    required: true,
  },
});
</script>
```

5. Cut the pet card-related styles from the `<styles>` block in HomeView.vue and paste them into `PetCard.vue`:

PetCard.vue

```
<style scoped>
.pet-card {
  position: relative;
}

.adoption-overlay {
  position: absolute;
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;
  background: rgba(40, 167, 69, 0.9);
  display: flex;
  align-items: center;
  justify-content: center;
  color: white;
  text-align: center;
  border-radius: 0.375rem 0.375rem 0 0;
}
</style>
```


What this component does:

- **Props**: Receives a `pet` object from its parent component.

- **Encapsulates**: All HTML and styling needed to render one pet card, including the ADOPTED overlay logic.
- **Reusable**: Parent components (e.g., `HomeView`) can drop this same `PetCard` into different lists, grids, or detail pages.

❖ E5.4. Step 2: Update HomeView to Use the New Component

1. Replace the `<div>` with the `v-for` from `HomeView.vue` and replace it with a `PetCard` element that has a `v-for` attribute:

HomeView.vue

```
<!-- Pet Grid - Now using PetCard components -->
<div id="petList">
  <PetCard v-for="pet in filteredPets" :key="pet.id" :pet="pet" />
</div>
```

2. Add a `v-if` directive to the `<div>` wrapped around the `PetCard` component that will only try to render the pet grid if there's at least 1 pet matching the current filter.

HomeView.vue

```
<div id="petList" v-if="filteredPets.length > 0">
  <PetCard v-for="pet in filteredPets" :key="pet.id" :pet="pet" />
</div>
```

3. Import `PetCard` into `HomeView.vue`

HomeView.vue

```
<script setup>
import { ref, computed } from 'vue';
import { pets as samplePets } from '../data/pets';

import PetCard from '../components/PetCard.vue';
```

❖ E5.5. Step 4: Test the Refactored Components

Start your development server with **npm run dev** and test:

1. **Filtering still works** - The filter controls should function exactly as before.

2. **Pet cards display correctly** - All visual styling and interactions should work.
3. **No console errors** - Check browser dev tools for any errors.

What you accomplished:

- **Separation of Concerns**: Each component has a single, clear responsibility.
- **Reusability**: PetCard can now be used in other parts of your app.
- **Maintainability**: Easier to find and fix issues in specific components.
- **Component Communication**: Props pass data down, events pass data up.

Solution: Components/Solutions/pet-vue-components/src/components/PetCard.vue

```
<template>
  <div class="pet-card">
    <img :src="pet.image" :alt="pet.name" />
    <div v-if="pet.adopted" class="adoption-overlay">
      <div class="adoption-message">
        <h6 class="fw-bold mb-1">ADOPTED!</h6>
        <small>Found their forever home</small>
      </div>
    </div>
    <div>
      <h5>{{ pet.name }}</h5>
      <p>{{ pet.breed }} &bull; {{ pet.gender }}</p>
      <p>{{ pet.size }} {{ pet.type }}</p>
    </div>
  </div>
</template>

<script setup>
const props = defineProps({
  pet: {
    type: Object,
    required: true,
  },
});
</script>

<style scoped>
.pet-card {
  position: relative;
}

.adoption-overlay {
  position: absolute;
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;
  background: rgba(40, 167, 69, 0.9);
  display: flex;
  align-items: center;
  justify-content: center;
  color: white;
  text-align: center;
```

Solution: Components/Solutions/pet-vue-components/src/components/PetCard.vue

```
44.     border-radius: 0.375rem 0.375rem 0 0;
45.   }
46.   </style>
```


❖ E5.6. Challenge: Extract the PetFilter Component

Follow a similar process as above to extract the logic related to the filters into a new component `src/components/PetFilter.vue`. The most difficult part of this challenge will be setting up the custom events. Be sure to review the **Component Events** reading, particularly the section on setting up `v-model`.

Challenge Solution: Components/Solutions/pet-vue-components/src/components/PetFilter.vue

```
<template>
  <div class="pet-filter">
    <h5>Filter Pets</h5>

    <!-- Pet Type Filter -->
    <div>
      <label for="typeFilter" class="form-label">Pet Type</label>
      <select id="typeFilter" class="form-select" v-model="selectedType"
        @change="emitFiltered"
      >
        <option value="">All Types</option>
        <option value="Dog">Dogs</option>
        <option value="Cat">Cats</option>
      </select>
    </div>

    <!-- Pet Size Filter -->
    <div>
      <label for="sizeFilter" class="form-label">Size</label>
      <select id="sizeFilter" class="form-select" v-model="selectedSize"
        @change="emitFiltered"
      >
        <option value="">All Sizes</option>
        <option value="Small">Small</option>
        <option value="Medium">Medium</option>
        <option value="Large">Large</option>
      </select>
    </div>
  </div>
</template>

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

// Define props
const props = defineProps({
  pets: { type: Array, required: true },
  modelValue: { type: Array, required: true },
});

// Define emits for v-model
const emit = defineEmits(['update:modelValue']);

```

Challenge Solution: Components/Solutions/pet-vue-components/src/components/PetFilter.vue

```
// Local filter state
const selectedType = ref('');
const selectedSize = ref('');

// Computed property for filtered pets
  const filteredPets = computed(() => {

    // Loop through the pets array and find pets who match either filter
    // REPLACE pets.value with props.pets
    return props.pets.filter((pet) => {

      // Type filter: include all if no type is selected
      const matchesType =
        selectedType.value === '' || pet.type === selectedType.value;

      // Size filter: include all if no size is selected
      const matchesSize =
        selectedSize.value === '' || pet.size === selectedSize.value;

      // Only return pets that match BOTH filters
      return matchesType && matchesSize;
    });
  });

// Emit updated filtered list upward
function emitFiltered() {
  emit('update:modelValue', filteredPets.value);
}

// Sync initial filtered list on mount
// Not strictly necessary, but ensures parent has initially filtered list
// if the defaults are changed in the future
onMounted(emitFiltered);
</script>
```

Steps to Extract the PetFilter Component:

1. Create a new file: `src/components/PetFilter.vue`.
2. Move the filter HTML (the two `select` elements) from `HomeView.vue` into the new component's `<template>`.

3. In the `<script setup>`, define props so the component receives the full `pets` array and the `modelValue` for `v-model`.
4. Transfer the filtering logic and state from `HomeView.vue` into `PetFilter.vue`:
    A. Move the reactive state (`selectedType` and `selectedSize`).
    B. Move the computed property (`filteredPets`).
5. Update the filtering logic to work with the `pets` prop instead of the `pets` ref (`props.pets.filter` instead of `pets.value.filter`).
6. Update the two `select` elements to emit `'update:modelValue'` with the filtered list whenever the filters change.
    A. We created an `emitFiltered()` method so that we didn't have to repeat logic.
    B. You can also call the method once on mounted. This will ensure that the parent and child components are in sync.
7. In `HomeView.vue`, replace the "pet-filter" `div` with the new `PetFilter` component (make sure it is imported):

```
<PetFilter :pets="samplePets" v-model="filteredPets" />
```

Solution: Components/Solutions/pet-vue-components/src/views/HomeView.vue

```
<template>
  <div class="pet-list-page">
    <!-- Hero Section -->
    <div class="hero-section text-center py-5 mb-4 rounded">
      <h1 class="display-4 fw-bold mb-3">Find Your Perfect Companion</h1>
      <p class="lead">
        Every pet deserves a loving home. Browse our available pets and find
        your new best friend today.
      </p>
    </div>
    <!-- Filters extracted into PetFilter component -->
    <PetFilter :pets="samplePets" v-model="filteredPets" />
  </div>
  <div id="petList" v-if="filteredPets.length > 0">
    <!-- Pet cards extracted into PetCard component -->
    <PetCard v-for="pet in filteredPets" :key="pet.id" :pet="pet" />
  </div>
</template>

<script setup>
  import { ref } from 'vue';
  import { pets as samplePets } from '../data/pets';
  import PetCard from '../components/PetCard.vue';
  import PetFilter from '../components/PetFilter.vue';

  const filteredPets = ref(samplePets);
</script>

<style>
#petList {
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
  gap: 16px;
  justify-content: center;
}
</style>
```

Conclusion

In this lesson, we guided you through the exploration of Vue.js components, covering the essentials of creating and utilizing components efficiently. You learned about the importance of breaking down a complex application into reusable components, the lifecycle phases, and how to manage component communication using props and events. We also explained advanced features such as slots and dynamic components, enriching your capability to build flexible and dynamic Vue applications. By the end of this lesson, you have acquired the knowledge to construct maintainable and scalable applications using components, harnessing the full potential of Vue.js.

LESSON 6
Vue Routing

Topics Covered

☑ Installing and configuring Vue Router.

☑ Defining routes, views, and navigation links.

☑ Creating nested routes and child views.

☑ Using dynamic route parameters and passing props.

☑ Programmatic navigation with the router.

☑ Adding and wiring routes in the PetVue app.

Introduction

In this lesson, we will introduce you to Vue Routing, an essential part of building single-page applications with Vue.js. You'll learn how to set up and configure Vue Router to handle navigation within your application, allowing users to seamlessly transition between different views without full page reloads. We will guide you through the basics of creating routes, mapping them to components, and programmatically navigating between views. You'll also learn about creating more complex routes with nested routes and dynamic routes. By the end of this lesson, you'll have a solid understanding of how to implement routing in your Vue projects, enhancing the user experience and optimizing your application's performance.

6.1. Vue Router Basics

Routing is a critical concept in single-page applications (SPAs) as it allows users to navigate through different views or components without requiring a full page reload. SPAs dynamically update content as users interact with the application, providing a seamless and efficient user experience akin to that of a desktop application. Effective routing contributes to maintaining state, organizing application structure, and optimizing performance.

❖ 6.1.1. Setting Up and Using Vue Router

Installing Vue Router

To begin using Vue Router, you first need to install it into your Vue.js project. This can be done using npm with the following command:

```
npm install vue-router
```

Or by installing it when you set up the project using Vite.

Creating a Router Instance

After installation, you need to create a router instance in your main entry file (e.g., `main.js`). This is usually done by creating a separate file and importing it into main.js. For example, in the default Vite with Vue Router configuration, `main.js` looks like this:

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

import { createApp } from 'vue'
import App from './App.vue'
import router from './router'

const app = createApp(App)

app.use(router)

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

The router import points to where all the Router setup code is written. In this case, it's in `src/router/index.js`. Open `src/router/index.js`, and you'll see 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,
    },
    {
      path: '/about',
      name: 'about',
      // route level code-splitting
      // this generates a separate chunk (About.[hash].js) for this route
      // which is lazy-loaded when the route is visited.
      component: () => import('../views/AboutView.vue'),
    },
  ],
})

export default router
```

The router file imports functions for creating routes and tracking the history (where the user has been previously, which makes the "back" button work). Next, it imports the **HomeView** component. The **HomeView** component is the only one that needs to be imported right away. As you can see in the comment above the `AboutView` component, the other components can be loaded as they are navigated to by using "lazy loading." Feel free to delete the three comment lines in the **AboutView** route after you've read them.

The `createRouter()` function takes as its argument an object with a `history` property and a `routes` property. The `history` property will likely be the same for every Vue app you create. The `routes` property is an array containing a list of objects that specify a path, a name for the route, and a component.

Mapping Routes to Components

Each route in the router configuration maps a path to a component. When a user navigates to a specific path, the corresponding component is displayed. To display the component specified by the routing configuration object, you use the `<router-view>` tag. For example, when you're creating the top-level

navigation for an SPA, which will switch out part or all of the home view when the path changes, you use `<router-view>` in the `App.vue` file, like in this simplified example:

```
<!-- App.vue -->
<template>
  <div id="app">
    <nav>
      <router-link to="/">Home</router-link>
      <router-link to="/about">About</router-link>
    </nav>
    <router-view/>
  </div>
</template>
```

So, clicking the **Home** link will swap out `<router-view/>` with `HomeView` and clicking the **About** link will swap it out with the `AboutView`.

Programmatic Routing with Vue Router

You can implement programmatic routing by accessing the router instance using the `useRouter` function. This function provides the `push` method for navigating to different routes. Programmatic routing is often used when the page should redirect based on the result of running a function or conditional code. For example, when a user attempts to log into an app, the app needs to redirect them to the `/users` route if their login was successful, or to a **/forgotpassword** route if their login was unsuccessful.

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

const router = useRouter();

const handleLogin = (isSuccessful) => {
  if (isSuccessful) {
    router.push('/users');
  } else {
    router.push('/forgotpassword');
  }
};
</script>
```

Programmatic routing is covered in more detail later in this lesson (in the section titled **Programmatic Routing**).

6.2. Nested Routes and Dynamic Routing

Understanding nested routes and dynamic routing is essential for creating complex, structured Vue.js applications. With Vue Router, you can define routes within routes (also known as nested routes) and create dynamic paths that accept route parameters.

❖ 6.2.1. Nested Routes

Nested routes allow you to create child routes within a parent route, providing a way to structure your application's content hierarchically. This supports complex user interfaces that require sub-navigation.

Defining Nested Routes

To define nested routes, use a children property in the route configuration. The children property contains the subroutes, which are configured using the same properties as their parents.

```
const routes = [
  {
    path: '/user/:id',
    component: UserComponent,
    children: [
      {
        path: 'profile',
        component: UserProfile
      },
      {
        path: 'posts',
        component: UserPosts
      }
    ]
  }
];
```

Using `<router-view>` for Nested Routes

To display nested components, use a `<router-view>` within the component rendered by the top-level component. This child `<router-view>` will render the components for the nested routes.


```
<template>
  <div>
    <h1>User ID: {{ $route.params.id }}</h1>
    <router-view></router-view>
  </div>
</template>
```

❖ 6.2.2. Dynamic Routing

Dynamic routing allows you to create routes that include dynamic segments, capturing values from the URL and making the route more flexible.

Defining Dynamic Routes

Dynamic segments are defined using a colon (`:`) followed by a parameter name in the path. This can be accessed within the component via `$route.params`.

```
const routes = [
  { path: '/user/:id', component: UserComponent }
];
```

In this example, `/user/:id` means any URL like `/user/123` or `/user/johndoe` will match this route.

Accessing Route Parameters

Route parameters are available on the `$route` object. Here is an example of how you might use them in a component:

```
<template>
  <div>
    <h1>User {{ $route.params.id }}</h1>
  </div>
</template>

<script>
export default {
  created() {
    console.log(this.$route.params.id);
  }
}
</script>
```


Implementing nested routes and dynamic routing can significantly enhance the interactivity and usability of your Vue.js applications by providing more structured and personalized navigational experiences.

6.3. Programmatic Routing

Programmatic routing lets you navigate users in response to events or logic (e.g., after login, on form submit, on a condition). With the Composition API, you access the router via `useRouter()` and issue

navigation commands such as `router.push`, `router.replace`, and `router.go`. This section explains how they work, when to use each, and how to pass dynamic route params, queries, and hashes.

❖ 6.3.1. Getting the Router Instance

```
<script setup>
import { useRouter, useRoute } from 'vue-router'

const router = useRouter()   // for navigating
const route = useRoute()     // for reading current route info (params, query, hash)
</script>
```

❖ 6.3.2. router.push: Navigate by Adding a New History Entry

`router.push` changes the current route and adds a new entry to the browser's history stack (so the Back button returns to the previous route). It supports several forms:

1) Push by string path

```
router.push('/about')
```

2) Push by named route

Named routes are the most reliable way to pass params for dynamic routes.

```
router.push({ name: 'about' })
```

3) Push with dynamic params

Given a route like `{ path: '/user/:id', name: 'user', component: UserView }`:

```
router.push({ name: 'user', params: { id: '42' } }) // navigates to /user/42
```

Notes about params:

- When using **string path**, `params` are ignored. Use a named route to pass `params`.
- You must provide all required dynamic segments (e.g., `:id`), otherwise navigation fails.

- Params are strings by default (e.g., numbers become strings in the URL).

4) Push with query and hash

You can also pass a hash and a query in the object passed to `router.push`.

A query is the set of key–value pairs appended to a URL after the question mark. For example, `/products?category=books&page=2`. In Vue Router, pass a plain object to `query`; values are serialized into the URL and read back as strings via `route.query`.

A hash is the fragment identifier appended after the pound sign. For example, `/products#reviews`. It typically points to an in-page anchor. In Vue Router, set `hash` to a string that starts with `#`, like `#reviews`.

```
router.push({
  name: 'search',
  query: { q: 'vue router', page: '2' }, // becomes ?q=vue%20router&page=2
  hash: '#tips'                          // becomes #tips
})
```


5) Push with a full path string that includes params and query

When you already have or must construct an exact URL, for example in the case of server-supplied URLs or routes that don't map cleanly to a named route, you can pass the URL as a string containing a full path.

This method gives you full control over path segments, query, and hash and preserves them exactly as built.

```
router.push('/user/42/profile?tab=security#two-factor')
```

❖ 6.3.3. router.replace: Navigate Without Adding a History Entry

`router.replace` works like `push` but it **replaces** the current entry in the browser history instead of adding a new one. Use it for redirects where pressing Back should not return to the pre-redirect page (e.g., after login, after completing an onboarding step, or canonical URL normalization).

```
<script setup>
import { useRouter } from 'vue-router'
const router = useRouter()

function afterLoginRedirect() {
  // User logs in; we don't want Back to go back to /login
  router.replace({ name: 'dashboard' })
}
</script>
```

❖ 6.3.4. router.go, router.back, router.forward: Move Through History

`router.go(n)` moves forward or backward in the browser history by `n` steps. Negative values move backward, positive values move forward. These are essentially wrappers over the native history API.

```
router.go(-1)   // same as router.back()
router.go(1)    // same as router.forward()
router.back()
router.forward()
```


Use cases:

- Implement a custom Back button: `router.back()`.
- Return multiple steps after a multi-screen flow: `router.go(-2)`.

Note: If the history stack includes external pages, `go/back/forward` may navigate outside your app. These calls do not return a navigation failure object.

❖ 6.3.5. Common Pitfalls and Best Practices

- Use **named routes** for dynamic params: `router.push({ name: 'user', params: { id: '123' } })`. If you pass a `path`, `params` are ignored.
- Provide all required params for routes with dynamic segments, or navigation will fail.
- Use `router.replace` for redirects where Back should not return to the previous route (e.g., after login).
- Use `router.go/back/forward` for history-based navigation; be aware these may navigate outside the app if the history stack contains external pages.

Exercise 6: Adding Routes to PetVue

25 to 35 minutes

In this exercise, you will learn how to use Vue Router to create navigation between different views in your Vue application. You'll build a detailed pet profile page and implement navigation from the pet list to individual pet details.

❖ E6.1. Step 1: Verify that Vue Router is installed and configured

1. Verify that Vue Router is installed. We installed Vue Router during the creation of our application (using **npm create vue@latest**). So, there should be nothing more for you to do. You can verify that Vue Router is installed by opening `package.json` and looking for `vue-router` in the dependencies array, as shown below. Note that your version of `vue-router` is likely different than the version shown in this screenshot.

```
    "preview": "vite preview"
  },
  "dependencies": {
    "bootstrap": "^5.3.8",
    "vue": "^3.5.18",
    "vue-router": "^4.5.1"
  },
  "devDependencies": {
    "@vitejs/plugin-vue": "^6.0.1",
    "vite": "^7.0.6"
```

2. If `vue-router` doesn't show up in your dependencies object in `package.json`, install it by running **npm install vue-router** at the root of your `pet-vue` project.
3. PetVue already contains two routes, which are already defined in `src/router/index.js`. Open this file to see these routes.

router/index.js

```
const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),
  routes: [
    {
      path: '/',
      name: 'home',
      component: HomeView,
    },
    {
      path: '/about',
      name: 'about',
      component: () => import('../views/AboutView.vue'),
    },
  ],
})
```

❖ E6.2. Step 2: Update the Router Configuration

Now let's add a new route to the router configuration.

1. Update `src/router/index.js` to include the new route:

router/index.js

```
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,
    },
    {
      path: '/about',
      name: 'about',
      component: () => import('../views/AboutView.vue'),
    },
    // Add this new route
    {
      path: '/pets/:id',
      name: 'pet-detail',
      component: () => import('../views/PetDetailView.vue'),
      props: true,
    },
  ],
});

export default router;
```


What this route configuration does:

- **Dynamic route parameter**: `:id` captures the pet ID from the URL
- **Lazy loading**: Uses `import()` to load the `PetDetailView` component lazily.
- **Props**: `props: true` automatically passes route params as component props
- **Named route**: `pet-detail` can be referenced by name in navigation

❖ E6.3. Step 3: Update HomeView to Navigate to PetDetailView

Now let's modify `HomeView` to navigate to the `PetDetailView` view when a pet card is clicked.

1. In your `src/views/HomeView.vue`, add the following function to the `<script>` section:

HomeView.vue

```
function handlePetClicked(pet) {
  // Navigate to pet detail page using the router
  router.push({
    name: 'pet-detail',
    params: { id: pet.id.toString() },
  });
};
```

2. Add the router import at the top of your script section in `HomeView.vue`:

HomeView.vue

```
// Add this import alongside your existing imports
import { useRouter } from 'vue-router';

// Add this line in your script setup section
const router = useRouter();
```

3. Add a custom event `pet-clicked` to the `PetCard` component:

   A. In `HomeView.vue`, update `PetCard` with `@pet-clicked="handlePetClicked(pet)"`:

```
<!-- HomeView.vue -->
<PetCard v-for="pet in filteredPets" :key="pet.id" :pet="pet"
      @pet-clicked="handlePetClicked(pet)" />
```

B. In `PetCard.vue`, capture click events on the "pet-card" `div` and emit the `pet-clicked` event:

```
<!-- PetCard.vue -->
<template>
  <div class="pet-card" @click="emit('pet-clicked')">
    ...
  </div>
</template>

<script setup>
const props = defineProps({
  pet: {
    type: Object,
    required: true,
  },
});

const emit = defineEmits(['pet-clicked']);
</script>
```


What this does:

- **useRouter()** gives us access to the router instance
- **router.push()** programmatically navigates to a new route
- **Named route navigation** uses the route name instead of the path
- **Route parameters** pass the pet ID in the URL

❖ E6.4. Step 4: Create the PetDetailView View

Next, we'll create a new view for displaying individual pet details.

1. Create a new file `src/views/PetDetailView.vue` and scaffold a new single file component:

PetDetailView.vue

```
<template>

</template>

<script setup>

</script>

<style>

</style>
```

2. In the `<script>` block, import `computed` from Vue, `useRoute` from Vue Router, and the pets data file:

PetDetailView.vue

```
import { computed } from 'vue';
import { useRoute } from 'vue-router';
import { pets as samplePets } from '../data/pets';
```


3. Get the route instance, which will give us the pet number to locate from the pet data.

```
const route = useRoute();
```

4. Get the pet id from the route parameter and use it to find the correct pet:

```
const pet = computed(() => {
  const petId = parseInt(route.params.id);
  return samplePets.find((p) => p.id === petId);
});
```

5. Now that your component has access to a variable named pet that contains all the data for a single pet, you can create the template and use any of the properties available. Here's a simple

list of all a pet's attributes. Use this as a starting point to style the pet detail page any way you like:

```
<template>
<pre>
id: {{pet.id}}
name: {{pet.name}}
type: {{pet.type}}
breed: {{pet.breed}}
age: {{pet.age}}
gender: {{pet.gender}}
size: {{pet.size}}
bio: {{pet.bio}}
image: {{pet.image}}
adopted: {{pet.adopted}}
location: {{pet.location}}
energy: {{pet.energy}}
goodWith: {{pet.goodWith}}
vaccinated: {{pet.vaccinated}}
spayed: {{pet.spayed}}
</pre>
</template>
```


6. Use **npm run dev** to start the development server and preview PetVue in a browser. Click on any pet to open their detail page, and make sure that the details are showing up.
7. Try manually changing the URL to a route for a pet ID that doesn't exist, such as `http://localhost:5173/pet/1000`. You should just see just blank space under the header where the pet details normally render. Can you figure out how to use conditional rendering to display an error message, such as "Pet not found" instead?

Solution: VueRouting/Solutions/pet-vue-routing/src/views/HomeView.vue

```
<template>
  <div class="pet-list-page">
    <!-- Hero Section -->
    <div class="hero-section text-center py-5 mb-4 rounded">
      <h1 class="display-4 fw-bold mb-3">Find Your Perfect Companion</h1>
      <p class="lead">
        Every pet deserves a loving home. Browse our available pets and find
        your new best friend today.
      </p>
    </div>
    <!-- Filters extracted into PetFilter component -->
    <PetFilter :pets="samplePets" v-model="filteredPets" />
  </div>
  <div id="petList" v-if="filteredPets.length > 0">
    <!-- Pet cards extracted into PetCard component -->
    <PetCard v-for="pet in filteredPets" :key="pet.id" :pet="pet"
      @pet-clicked="handlePetClicked(pet)" />
  </div>
</template>

<script setup>
  import { ref } from 'vue';
  import { useRouter } from 'vue-router';
  import { pets as samplePets } from '../data/pets';
  import PetCard from '../components/PetCard.vue';
  import PetFilter from '../components/PetFilter.vue';

  const filteredPets = ref(samplePets);

  const router = useRouter();

  function handlePetClicked(pet) {
    console.log('Pet clicked:', pet);
    // Navigate to pet detail page using the router
    router.push({
      name: 'pet-detail',
      params: { id: pet.id.toString() },
    });
  };
</script>

<style>
#petList {
  display: flex;
```

Solution: VueRouting/Solutions/pet-vue-routing/src/views/HomeView.vue

```
45.     flex-direction: row;
46.     flex-wrap: wrap;
47.     gap: 16px;
48.     justify-content: center;
49.   }
50.   </style>
```

Solution: VueRouting/Solutions/pet-vue-routing/src/components/PetCard.vue

```
<template>
  <div class="pet-card" @click="emit('pet-clicked')">
    <img :src="pet.image" :alt="pet.name" />
    <div v-if="pet.adopted" class="adoption-overlay">
      <div class="adoption-message">
        <h6 class="fw-bold mb-1">ADOPTED!</h6>
        <small>Found their forever home</small>
      </div>
    </div>
    <div>
      <h5>{{ pet.name }}</h5>
      <p>{{ pet.breed }} &bull; {{ pet.gender }}</p>
      <p>{{ pet.size }} {{ pet.type }}</p>
    </div>
  </div>
</template>

<script setup>
const props = defineProps({
  pet: {
    type: Object,
    required: true,
  },
});

const emit = defineEmits(['pet-clicked']);
</script>

<style scoped>
.pet-card {
  position: relative;
}

.adoption-overlay {
  position: absolute;
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;
  background: rgba(40, 167, 69, 0.9);
  display: flex;
  align-items: center;
  justify-content: center;
```

Solution: VueRouting/Solutions/pet-vue-routing/src/components/PetCard.vue

```
    color: white;
    text-align: center;
    border-radius: 0.375rem 0.375rem 0 0;
  }
  </style>
```

Solution: VueRouting/Solutions/pet-vue-routing/src/router/index.js

```
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,
    },
    {
      path: '/about',
      name: 'about',
      component: () => import('../views/AboutView.vue'),
    },
    {
      path: '/pets/:id',
      name: 'pet-detail',
      component: () => import('../views/PetDetailView.vue'),
      props: true,
    },
  ],
})

export default router
```

Solution: VueRouting/Solutions/pet-vue-routing/src/views/PetDetailView.vue

```
<template>
  <div class="container my-4">
    <!-- Not found / invalid ID -->
    <div v-if="!pet" class="alert alert-warning">
      Pet not found. <a class="alert-link" @click.prevent="$router.back()">Go back</a>.
    </div>

    <div v-else class="row g-4">
      <!-- Left: Image -->
      <div class="col-lg-7">
        <img :src="pet.image" :alt="pet.name" class="img-fluid rounded w-100" />
      </div>

      <!-- Right: Summary -->
      <div class="col-lg-5">
        <h1 class="h2 mb-1">{{ pet.name }}</h1>
        <p class="text-muted mb-3">{{ pet.breed }} {{ pet.type }}</p>

        <dl class="row mb-3">
          <dt class="col-4">Age:</dt>
          <dd class="col-8">{{ pet.age }} years old</dd>

          <dt class="col-4">Gender:</dt>
          <dd class="col-8">{{ pet.gender }}</dd>

          <dt class="col-4">Size:</dt>
          <dd class="col-8">{{ pet.size }}</dd>

          <dt class="col-4">Energy:</dt>
          <dd class="col-8">
            <span class="badge rounded-pill" :class="energyBadgeClass">
              {{ pet.energy }}
            </span>
          </dd>

          <dt class="col-4">Location:</dt>
          <dd class="col-8">{{ pet.location }}</dd>
        </dl>
      </div>
    </div>

```

Solution: VueRouting/Solutions/pet-vue-routing/src/views/PetDetailView.vue

```
      <!-- About -->
      <div v-if="pet" class="card my-4">
        <div class="card-body">
          <h5 class="card-title mb-2">About {{ pet.name }}</h5>
          <p class="mb-0">{{ pet.bio }}</p>
        </div>
      </div>

      <!-- Good With + Health -->
      <div v-if="pet" class="row g-4">
        <div class="col-lg-6">
          <div class="card h-100">
            <div class="card-body">
              <h5 class="card-title">Good With</h5>
              <div class="d-flex flex-wrap gap-2">
                <span
                  v-for="(tag, i) in pet.goodWith"
                  :key="i"
                  class="badge rounded-pill bg-info text-dark"
                >
                  {{ tag }}
                </span>
              </div>
            </div>
          </div>
        </div>

        <div class="col-lg-6">
          <div class="card h-100">
            <div class="card-body">
              <h5 class="card-title">Health Information</h5>
              <ul class="list-unstyled mb-0">
                <li class="mb-2">
                  <strong class="me-1">Vaccinated:</strong>
                  <span :class="pet.vaccinated ? 'text-success' : 'text-danger'">
                    {{ pet.vaccinated ? 'Yes' : 'No' }}
                  </span>
                </li>
                <li>
                  <strong class="me-1">Spayed:</strong>
                  <span :class="pet.spayed ? 'text-success' : 'text-danger'">
                    {{ pet.spayed ? 'Yes' : 'No' }}
                  </span>
```

Solution: VueRouting/Solutions/pet-vue-routing/src/views/PetDetailView.vue

```
85.                 </li>
86.               </ul>
87.             </div>
88.           </div>
89.         </div>
90.       </div>
91.
92.     </div>
93.   </template>
94.
95.   <script setup>
96.   import { computed } from 'vue'
97.   import { useRoute } from 'vue-router'
98.   import { pets as samplePets } from '../data/pets'
99.
100.  const route = useRoute()
101.
102.  // Find the pet based on the ID from the route params.
103.  const pet = computed(() => {
104.    const petId = Number(route.params.id)
105.    return samplePets.find(p => p.id === petId)
106.  })
107.
108.  // Map energy to Bootstrap badge colors.
109.  const energyBadgeClass = computed(() => {
110.    if (!pet.value) return 'bg-secondary'
111.    const level = (pet.value.energy || '').toLowerCase()
112.    return level === 'low'
113.      ? 'bg-success'
114.      : level === 'medium'
115.      ? 'bg-warning text-dark'
116.      : level === 'high'
117.      ? 'bg-danger'
118.      : 'bg-secondary'
119.  })
120.  </script>
```


Conclusion

In this lesson, you explored the fundamental aspects of Vue Routing, learning how to implement navigation between different components in your Vue.js applications. We guided you through the

installation and configuration of Vue Router, creating routes and managing dynamic and nested routing. You gained hands-on experience by adding routes to PetVue and testing navigation flows. By mastering these skills, you're now equipped to create seamless and interactive single-page applications with enhanced user experiences.