
Learning
Vue.JS

Lesson 1: Introduction to Vue.js (Revised)

1. What is Vue.js?

Vue.js is a progressive JavaScript framework for building user interfaces. It's designed to be incrementally adoptable, meaning you can start using it for small parts of your application and gradually expand its usage as needed.

Key features of Vue.js:

- Reactive data binding
- Composable component architecture
- Declarative rendering
- Virtual DOM for efficient updates
- Lightweight and fast

2. Setting Up Your Development Environment

Before we start coding, let's set up your development environment:

1. Install Node.js and npm (Node Package Manager) from https://nodejs.org/
2. Install Vue CLI globally by running this command in your terminal

```
npm install -g @vue/cli
```

3. Create a new Vue project:

```
vue create my-first-vue-app
```

   When prompted, choose "Vue 3" as the preset.

4. Navigate to your project directory:

```
cd my-first-vue-app
```

5. Open the project in your favorite code editor.
6. Start the development server:

```
npm run serve
```

7. Open your browser and go to `http://localhost:8080` to see your Vue app running.

3. Understanding Vue.js Basics

The Vue Instance

Every Vue application starts by creating a new Vue instance with the `createApp` function:

```
import { createApp } from 'vue'
import App from './App.vue'

createApp(App).mount('#app')
```

This code is already in your `main.js` file.

Component Structure

Vue components are typically defined in `.vue` files, which contain three sections:

- `<template>`: The HTML structure of your component
- `<script>`: The JavaScript logic
- `<style>`: The CSS styles (optional)

Template Syntax

Vue uses an HTML-based template syntax that allows you to declaratively bind the rendered DOM to the underlying Vue instance's data.

Example:

```
<template>
  <div>
    <h1>{{ message }}</h1>
  </div>
</template>

<script>
export default {
  name: 'HelloWorld',
  data() {
    return {
      message: 'Hello, Vue!'
    }
  }
}
</script>
```

Declarative Rendering

Vue.js uses a template syntax to declaratively render data to the DOM. Here's a simple example:

```
<template>
  <div>
    <h1>{{ title }}</h1>
    <p>{{ message }}</p>
  </div>
</template>

<script>
export default {
  name: 'HelloWorld',
  data() {
    return {
      title: 'Welcome to Vue.js',
      message: 'This is my first Vue app!'
    }
  }
}
</script>
```

In this example, **`{{ title }}`** and **`{{ message }}`** will be replaced with the values of `title` and `message` from the component's data.

4. Practical Example: Creating a Simple Counter

Let's create a simple counter application to demonstrate basic Vue.js concepts.

Create a new file called MyCounter.vue in your project's src/components/ directory:

```
<template>
  <div>
    <h2>{{ title }}</h2>
    <p>Count: {{ count }}</p>
    <button @click="increment">Increment</button>
    <button @click="decrement">Decrement</button>
  </div>
</template>

<script>
export default {
  name: 'MyCounter',
  data() {
    return {
      title: 'Vue.js Counter',
      count: 0
    }
  },
  methods: {
    increment() {
      this.count++
    },
    decrement() {
      this.count--
    }
  }
}
</script>
```

Now, update your App.vue file to use this component:

```
<template>
  <div id="app">
    <MyCounter />
  </div>
</template>

<script>
import MyCounter from './components/MyCounter.vue'

export default {
  name: 'App',
  components: {
    MyCounter
  }
}
</script>
```

Key points to note:

- We use PascalCase for component names (e.g., `MyCounter`).
- Component files should use multi-word names to avoid conflicts with existing or future HTML elements.
- The `@click` directive is used for event handling.
- Data properties are returned from a function to ensure each component instance has its own data.

5. Vue.js Core Concepts

Directives

Directives are special attributes with the `v-` prefix. They apply special reactive behavior to the rendered DOM. Here are some common directives:

- `v-bind`: Dynamically binds an attribute to an expression. Example: `<a v-bind:href="url">Link</a>` or shorthand `:href="url"`
- `v-if`, `v-else`, `v-else-if`: Conditional rendering. Example: `<p v-if="seen">Now you see me</p>`
- `v-for`: List rendering. Example: `<li v-for="item in items" :key="item.id">{{ item.text }}</li>`
- `v-on`: Attaches an event listener to the element. Example: `<button v-on:click="doSomething">Click me</button>` or shorthand `@click="doSomething"`
- `v-model`: Creates two-way data binding on form inputs. Example: `<input v-model="message">`

v-bind

Use `v-bind` to dynamically bind attributes:

```
<template>
  <div>
    <a v-bind:href="url">{{ linkText }}</a>
    <!-- Shorthand -->
    <img :src="imageUrl" :alt="imageAlt">
  </div>
</template>

<script>
export default {
  data() {
    return {
      url: 'https://vuejs.org',
      linkText: 'Visit Vue.js Website',
      imageUrl: '/path/to/image.jpg',
      imageAlt: 'Vue.js Logo'
    }
  }
}
</script>
```

v-if, v-else, v-else-if

Use these directives for conditional rendering:

```
<template>
  <div>
    <p v-if="score > 90">Excellent!</p>
    <p v-else-if="score > 70">Good job!</p>
    <p v-else>Keep practicing!</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      score: 85
    }
  }
}
</script>
```

v-for

Use `v-for` for list rendering:

```
<template>
  <ul>
    <li v-for="(item, index) in items" :key="item.id">
      {{ index + 1 }}. {{ item.text }}
    </li>
  </ul>
</template>

<script>
export default {
  data() {
    return {
      items: [
        { id: 1, text: 'Learn Vue.js' },
        { id: 2, text: 'Build something awesome' },
        { id: 3, text: 'Share with the community' }
      ]
    }
  }
}
</script>
```

v-on

Use v-on to attach event listeners:

```
<template>
  <div>
    <p>{{ message }}</p>
    <button v-on:click="reverseMessage">Reverse Message</button>
    <!-- Shorthand -->
    <button @mouseover="showTooltip">Hover Me</button>
  </div>
</template>

<script>
export default {
  data() {
    return {
      message: 'Hello Vue.js!'
    }
  },
  methods: {
    reverseMessage() {
      this.message = this.message.split('').reverse().join('')
    },
    showTooltip() {
      alert('This is a tooltip!')
    }
  }
}
</script>
```

v-model

Use v-model for two-way data binding on form inputs:

```
<template>
  <div>
    <input v-model="username" placeholder="Enter your username">
    <p>Your username is: {{ username }}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      username: ''
    }
  }
}
</script>
```

Computed Properties

Computed properties are used for complex logic that you don't want to put directly in your template. They are cached based on their dependencies.

Example:

```
<script>
export default {
  data() {
    return {
      firstName: 'John',
      lastName: 'Doe'
    }
  },
  computed: {
    fullName() {
      return this.firstName + ' ' + this.lastName
    }
  }
}
</script>
```

Computed properties are especially useful for complex calculations or transformations. Here's an expanded example:

```
<template>
  <div>
    <h2>Product: {{ productName }}</h2>
    <p>Price: ${{ price }}</p>
    <p>Quantity: <input type="number" v-model.number="quantity"></p>
    <p>Total: ${{ total }}</p>
    <p>Discount Applied: {{ discountApplied ? 'Yes' : 'No' }}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      productName: 'Super Widget',
      price: 9.99,
      quantity: 1
    }
  },
  computed: {
    total() {
      return (this.price * this.quantity).toFixed(2)
    },
    discountApplied() {
      return this.quantity >= 5
    }
  }
}
</script>
```

In this example, `total` and `discountApplied` are computed properties. They automatically update whenever `price` or `quantity` changes.

6. Exercises

1. Modify the MyCounter component to include a reset button that sets the count back to 0. Hint: You'll need to add a new method and a button that calls this method.
2. Add a new data property called `step` and use it to control how much the count changes with each button click. Hint: You'll need to modify the `data()` function and update the `increment` and `decrement` methods.
3. Create a new component called `MyGreeting.vue` that takes a `name` prop and displays a greeting message. Hint: You'll need to learn about props in Vue.js. The Vue.js documentation has a section on props that will be helpful.

7. Project: Personal Information Card

Create a new component called `MyPersonalCard.vue` that displays a personal information card. The card should include:

- Name
- Job title
- A short bio
- Contact information (email and phone number)
- A button to toggle showing/hiding the contact information

Use Vue.js features like data properties, methods, and v-if directives to implement this functionality.

Hints:

- Start by creating the component structure with a `<template>`, `<script>`, and optionally `<style>` sections.
- Think about what data properties you'll need to store the information and the toggle state.
- Use `v-if` or `v-show` for toggling the visibility of contact information.
- Remember to register and use your new component in `App.vue`.

8. Methods

Methods are functions associated with a Vue instance. They're commonly used for event handling or complex logic. Here's an example:

```
<template>
  <div>
    <p>Random number: {{ randomNumber }}</p>
    <button @click="generateRandomNumber">Generate New Number</button>
  </div>
</template>

<script>
export default {
  data() {
    return {
      randomNumber: 0
    }
  },
  methods: {
    generateRandomNumber() {
      this.randomNumber = Math.floor(Math.random() * 100) + 1
    }
  }
}
</script>
```

In this example, generateRandomNumber is a method that updates the randomNumber data property with a new random value when the button is clicked.

Conclusion

This enhanced lesson provides a comprehensive introduction to Vue.js, covering setup, basic concepts, directives, computed properties, and methods. Each concept is explained with practical examples and code snippets, giving you a solid foundation to start building Vue.js applications.

Remember to practice with the provided exercises and project to reinforce your learning. In the next lesson, we'll explore component communication and lifecycle hooks in more detail.