Vue Todo List Project Notes

Haoru

November 2025

1 Project Structure Introduction

1.1 Project Introduction

This project is a self-learning exercise to understand the basics of Vue 3 by building a simple Todo List application. The main goals are:

- To get familiar with Vue Single File Components (`.vue` files).
- To understand the `template`, `script`, and `style` sections in `App.vue`.
- To practice basic layout and styling using CSS.
- To understand how global CSS is imported and applied.

The project is created using Vite as a modern frontend build tool.

1.2 Installing Vue via npm

To create a Vue project from scratch with Vite, the following commands are used:

Listing 1: Create and run the Vue project

```
npm create vite@latest
cd vue-todo-list
npm install
npm run dev
```

After running the development server, the application can be opened in a browser at a URL similar to:

Listing 2: Default dev server URL

```
http://localhost:5173
```

1.3 Project Structure

The important project files are:

- `src/App.vue` – the main (and currently only) Vue component.
- `src/main.js` – the entry point of the Vue application.
- `src/style.css` – the global CSS file, including the gradient background.
- `index.html` – the root HTML container where Vue mounts.

1.4 Global CSS Setup

A simple global gradient background is defined in `src/style.css`:

Listing 3: src/style.css

```
body {
  background: linear-gradient(
    to right,
    rgb(113, 65, 168),
    rgba(44, 114, 251, 1)
  );
}
```

Because this file is imported in `main.js`, the gradient background is applied to the whole page.

1.5 App Entry File

The entry file `src/main.js` looks like this:

Listing 4: src/main.js

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

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

Explanation:

- `createApp` creates a Vue application.
- `App.vue` is imported as the root component.
- `style.css` applies global styling (including background).
- `mount('#app')` attaches Vue to the page.

1.6 Base Component: App.vue

The main UI of the Todo List is defined in `src/App.vue`. At this stage, the component focuses on layout and styling:

- Title: ToDo List
- Input box
- Add Todo button
- A (mock) Completed todo layout

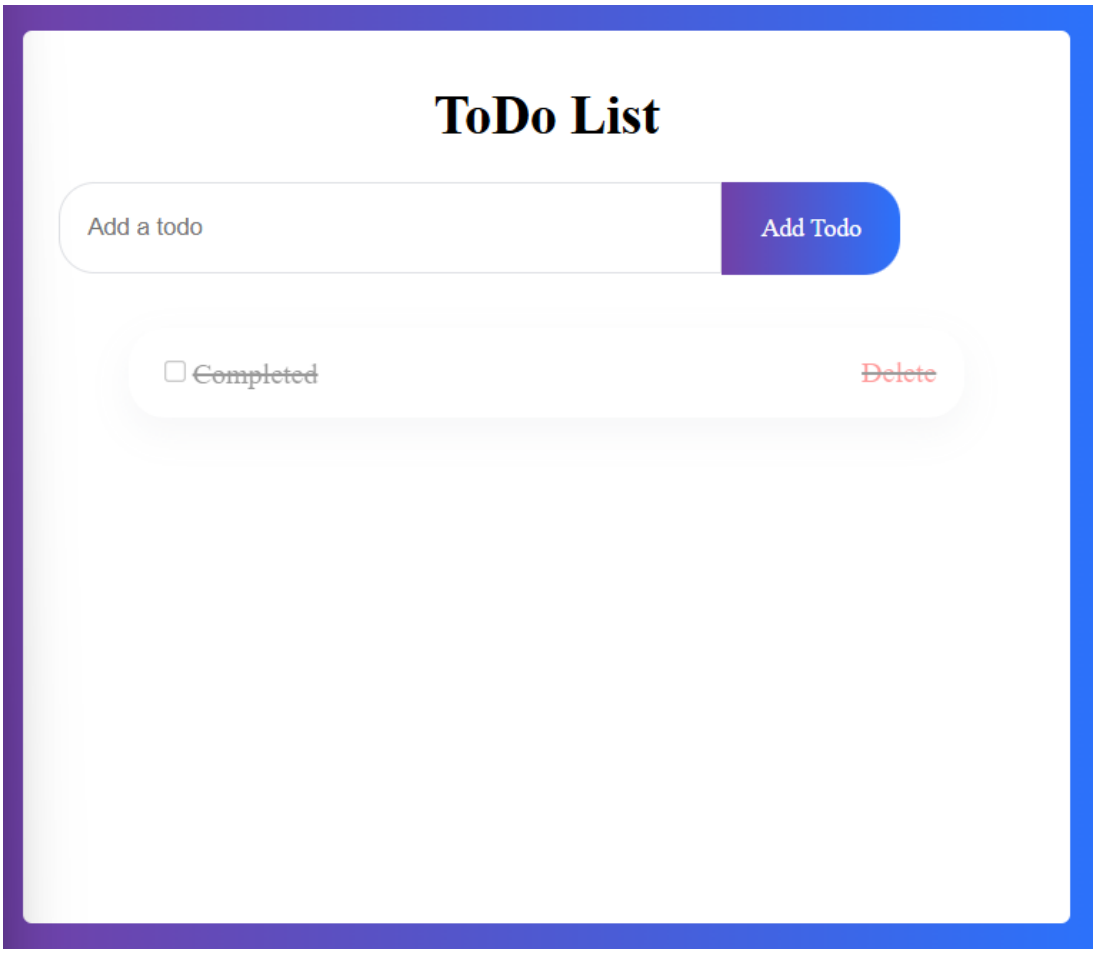

Figure 1: Current UI of the Todo List Component

1.6.1 Script Block

Listing 5: App.vue – <script setup>

```
<script setup>

</script>
```

1.6.2 Template Block

Listing 6: App.vue – <template>

```
<template>
  <div class="todo-app">
    <div class="title">ToDo List</div>

    <div class="todo-form">
      <input
        type="text"
        class="todo-input"
        placeholder="Add a todo"
      />
      <div class="todo-button">Add Todo</div>
    </div>

```

```
    <div class="completed">
      <div>
        <input type="checkbox" />
        <span class="name">Completed</span>
      </div>
      <div class="del">Delete</div>
    </div>
  </div>
</template>
```

1.6.3 Style Block

Listing 7: App.vue – <style scoped>

```
<style scoped>
.todo-app {
  box-sizing: border-box;
  margin-top: 40px;
  margin-left: 1%;
  padding-top: 30px;
  width: 98%;
  height: 500px;
  background: #ffffff;
  border-radius: 5px;
}

.title {
  text-align: center;
  font-size: 30px;
  font-weight: 700;
}

.todo-form {
  display: flex;
  margin: 20px 0 30px 20px;
}

.todo-button {
  width: 100px;
  height: 52px;
  border-radius: 0 20px 20px 0;
  text-align: center;
  background: linear-gradient(
    to right,
    rgb(113, 65, 168),
    rgba(44, 114, 251, 1)
  );
  color: #fff;
  line-height: 52px;
  cursor: pointer;
  font-size: 14px;
  user-select: none;
}

.todo-input {
  padding: 0px 15px;
  border-radius: 20px 0 0 20px;
  border: 1px solid #dfe1e5;
```

```
  outline: none;
  width: 60%;
  height: 50px;
}

.item {
  box-sizing: border-box;
  display: flex;
  align-items: center;
  justify-content: space-between;
  width: 80%;
  height: 50px;
  margin: 8px auto;
  padding: 16px;
  border-radius: 20px;
  box-shadow: rgba(149, 157, 165, 0.2) 0px 8px 20px;
}

.del {
  color: red;
}

.completed {
  box-sizing: border-box;
  display: flex;
  align-items: center;
  justify-content: space-between;
  width: 80%;
  height: 50px;
  margin: 8px auto;
  padding: 16px;
  border-radius: 20px;
  box-shadow: rgba(149, 157, 165, 0.2) 0px 8px 20px;
  text-decoration: line-through;
  opacity: 0.4;
}
</style>
```

1.7 Current Progress

At this stage, the project has:

- A fully styled UI container for the todo application.
- An input box and a visible "Add Todo" button (UI only, no logic yet).
- A mock "Completed" item preview.
- Scoped CSS applied cleanly.
- Global background from `style.css`.

2 Add New Functions

2.1 First Interactive Feature: Logging a Constant Value on Click

In this step, the Todo List application gains its first interactive behavior: clicking the *Add Todo* button now triggers a function that prints a constant value (defined inside the Vue component) to the browser console.

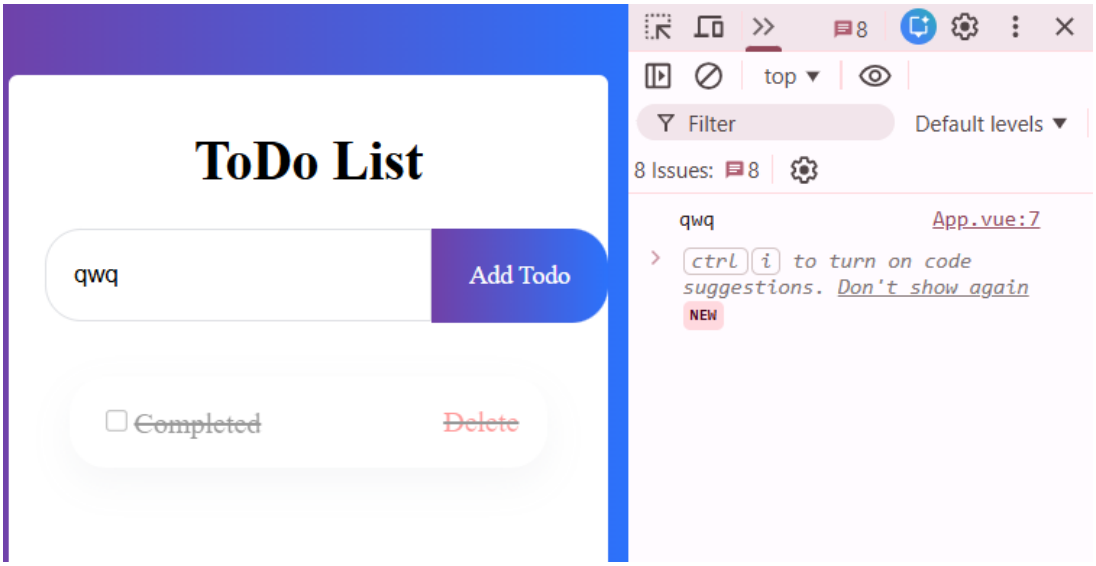

Figure 2: Constant value printed from the Vue component

2.1.1 Goal

- Create a reactive constant using `ref()`.
- Trigger a function when the user clicks a button.
- Print the constant value to the console.

2.1.2 Key Concepts Learned

- **ref():** creates a reactive variable.
- **@click:** shorthand for `v-on:click`.
- **ref.value:** required to access the stored value.

2.1.3 Updated <script setup> Block

Listing 8: App.vue – <script setup> (updated)

```
<script setup>
import { ref } from 'vue'
const value = ref('qwq')

function add() {
  console.log(value.value)
}
</script>
```

2.1.4 Updated <template> Block

Two minor changes were introduced:

- A constant `value` was referenced instead of user input.
- The `@click="add"` event triggers the log action.

Listing 9: App.vue – <template> (updated)

```
<template>
  <div class="todo-app">
    <div class="title">ToDo List</div>

    <div class="todo-form">
      <input
        v-model="value"
        type="text"
        class="todo-input"
        placeholder="Add a todo"
      />
      <div @click="add" class="todo-button">Add Todo</div>
    </div>
  </div>
</template>
```

2.1.5 Result

When clicking the **Add Todo** button, the browser console prints the constant string stored inside `value`:

Listing 10: Console output example

```
qwq
```

This confirms that:

- The `ref` API works correctly.
- Button events trigger JavaScript logic.
- Reactive values require `.value` for access.

2.2 Rendering a Static Todo List with `v-for`

In this step, the Todo List application is updated to display multiple todo items based on a predefined static array. Instead of rendering only a single placeholder item, Vue's `v-for` directive is used to loop through a list of objects.

2.2.1 Goal

- Create a static `ref` array containing multiple todos.
- Remove the hard-coded single item.
- Use `v-for` to render all items in the array.

2.2.2 Key Concepts Learned

- **ref([]):** allows an array to be reactive.
- **v-for:** loops over arrays inside Vue templates.
- **item in list:** the simplest form of iteration syntax.

2.2.3 Updated <script setup> Block

Here we define a static array of todo objects, each with a `text` description and `isComplete` state.

Listing 11: App.vue – <script setup> (static list)

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

const value = ref("qwq");

const list = ref([
  { isCompleted: false, text: "go shopping" },
  { isCompleted: false, text: "study" },
  { isCompleted: false, text: "clean garden" }
]);

function add() {
  console.log(value.value);
}
</script>
```

2.2.4 Updated <template> Block

The following code replaces a single static item with multiple rendered items using `v-for`:

Listing 12: App.vue – <template> (list rendering)

```
<template>
  ...

    <div v-for="item in list" class="completed">
      <div>
        <input type="checkbox" />
        <span class="name">Completed</span>
      </div>
      <div class="del">Delete</div>
    </div>
  </div>
</template>
```

2.2.5 Result

At this stage, Vue successfully loops through the static list and renders three repeated todo items on the page. However, before updating the template, each todo still displayed the same placeholder text — `Completed`.

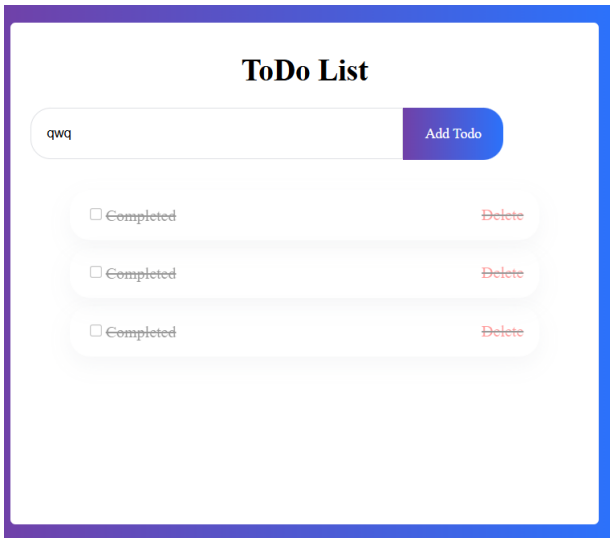

Figure 3: Static todo list rendered using `v-for`

To correctly display the text from each todo object, Vue's templating syntax must be used. This is done with double curly braces:

Listing 13: Using {{ }} to display todo text

```
<span class="name">{{ item.text }}</span>
```

This syntax inserts the value of `item.text` into the DOM. Once updated, each todo is rendered using its own text:

- go shopping
- study
- clean garden

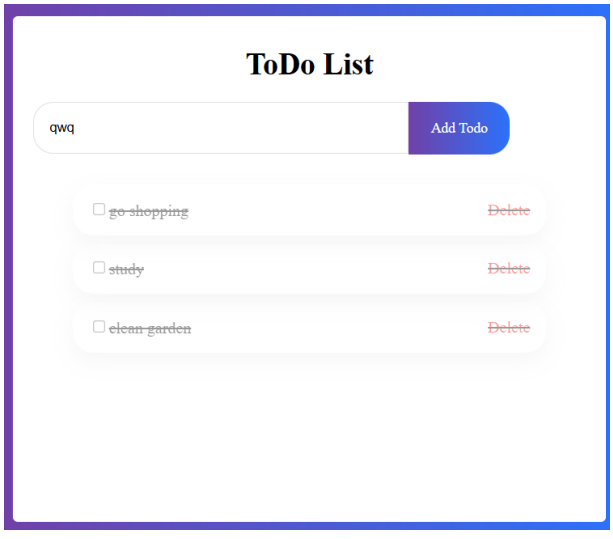

Figure 4: Final- Static todo list rendered using `v-for` 2

This confirms that:

- Vue can render a list using `v-for`.
- Each repeated DOM block receives its own `item` scope.
- Expressions inside {{ }} are evaluated and printed.

2.3 Dynamic Style Switching Based on Completion Status

In this part, we implement dynamic visual updates for each todo item based on its completion state. This includes:

- Using Vue's `:class` syntax to bind CSS classes conditionally.
- Using `v-model` to toggle the `isCompleted` property.
- Automatically switching between "active" and "completed" styles.

2.3.1 Dynamic Class Binding

Vue supports dynamic CSS class assignment via `:class`, which is shorthand for `v-bind:class`. It allows us to evaluate JavaScript expressions inside the template, instead of treating the class as a literal string.

Listing 14: Conditional class binding using `:class`

```
<div v-for="item in list"
    :class="item.isCompleted ? 'completed' : 'item'">
```

- `class="..."` — static, never changes
- `:class="..."` — dynamic, reactive at runtime

When `item.isCompleted` is true, the item gets the `completed` class (strikethrough + faded opacity). Otherwise it keeps the normal `item` class.

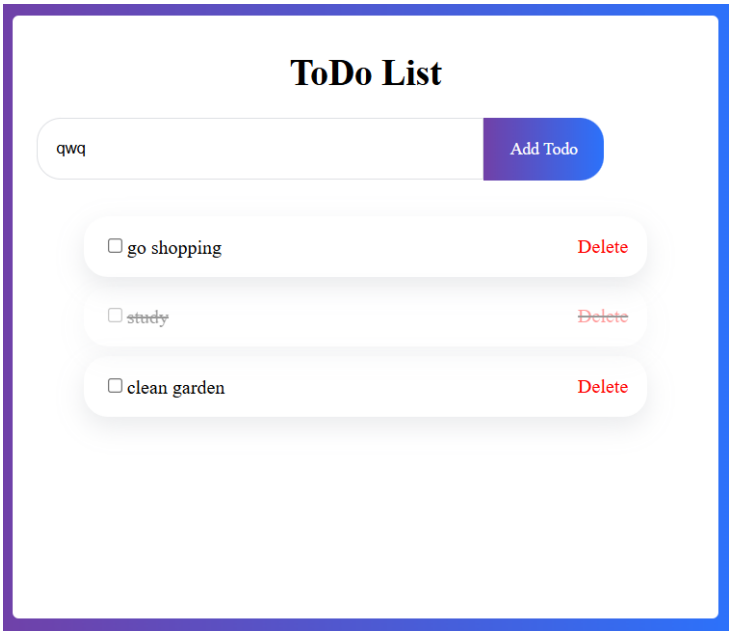

Figure 5: Items change appearance based on `isCompleted`

2.3.2 Making the Checkbox Interactive with `v-model`

Initially, each checkbox was static and not linked to the data model:

Listing 15: Before: static checkbox

```
<input type="checkbox" />
```

By enabling two-way data binding, clicking the checkbox now directly toggles `item.isCompleted`:

Listing 16: After: interactive checkbox with `v-model`

```
<input v-model="item.isCompleted" type="checkbox" />
```

Why this works

- `v-model` on a checkbox binds to a boolean.
- Clicking switches between `true` and `false`.
- The ternary expression in `:class` instantly re-evaluates.
- Vue reactivity updates the UI automatically—no extra code needed.

2.3.3 Final Template

Listing 17: Fully reactive todo list items

```
<div
  v-for="item in list"
  :class="item.isCompleted ? 'completed' : 'item'"
>
  <div>
    <input v-model="item.isCompleted" type="checkbox" />
    <span class="name">{{ item.text }}</span>
  </div>
  <div class="del">Delete</div>
</div>
```

2.3.4 Visual Result

Each todo item now fully reflects its internal state:

- Unchecked → normal style
- Checked → faded & strikethrough style

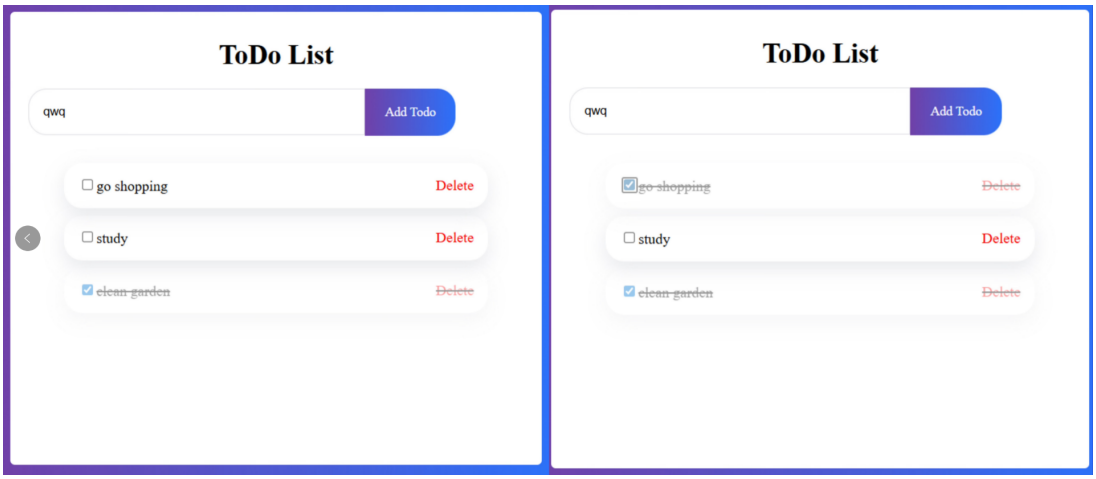

Figure 6: Toggling completed state by clicking the checkbox

2.4 Adding and Deleting Todo Items

With rendering and style switching complete, we now implement the core data operations of a Todo List application:

- Adding a new todo item
- Deleting an existing todo item

Both actions work by modifying the reactive array `list` defined in the <`script setup`> block.

2.4.1 Adding a New Todo Item

The Add button is already bound to the `add()` function via:

```
<div @click="add" class="todo-button">Add Todo</div>
```

This means that clicking the button will call the function below, located inside the <script setup> section:

Listing 18: add() function inside <script setup>

```
function add() {
  // Get user input and remove whitespace at the beginning and end
  const text = value.value.trim();

  // If the input is empty, stop here and do nothing
  if (!text) return;

  // Add a new todo item to the list
  list.value.push({
    isCompleted: false,
    text,
  });

  // Clear the input field after adding
  value.value = "";
}
```

Explanation

- `value.value` retrieves the current text from the input
- `trim()` removes extra whitespace
- Empty input is ignored to prevent blank todos
- A new object is pushed into the reactive array `list`
- `isCompleted` defaults to `false`
- After pushing, the input box is cleared

2.4.2 Deleting a Todo Item

We define a delete function inside <script setup> that removes one item by index:

Listing 19: del() function inside <script setup>

```
function del(index) {
  // Remove the todo item at the specified index
  list.value.splice(index, 1);
}
```

Explanation

- `splice()` mutates the array
- `index` is passed from the template
- Vue reactivity ensures the UI updates instantly

2.4.3 Template Update for Deletion

To pass the index into `del()` we extend the v-for syntax:

Listing 20: v-for with item and index

```
<div v-for="(item, index) in list"
    :class="item.isCompleted ? 'completed' : 'item'">
```

Then we bind the delete button inside the <`template`> block:

Listing 21: Delete button inside <template>

```
<div @click="del(index)" class="del">Delete</div>
```

Summary

- The `add()` function pushes a new todo object into the array.
- The `del()` function removes one item based on its index.
- `v-for` gives access to both `item` and `index`.
- UI updates automatically because `list` is a reactive ref.

At this stage, the Todo List application supports fully working:

- Adding todos
- Checking todos as complete
- Removing todos
- Reactive UI updates