How to Use Vue.js

Trung Nguyen Trong

Atom2021, May 27, 2021

Contents

1. What is Vue.js?

Vue (pronounced /vju:/, like **view**) is a **progressive framework** for building user interfaces. Unlike other monolithic frameworks, Vue is designed from the ground up to be incrementally adoptable. The core library is focused on the view layer only, and is easy to pick up and integrate with other libraries or existing projects. On the other hand, Vue is also perfectly capable of powering sophisticated Single-Page Applications when used in combination with modern tooling and supporting libraries. [2]

Figure 1: Vue Logo

2. Installation

To start Vue.js, I introduce you a tool called Vue CLI. It helps us create projects with certain configuration and built-in tools that give us all those nice features of Vue framework. But the most important tool, which you will need is Node.js. Node.js is a JavaScript runtime which allows you to run JavaScript code outside of the browser. You need to install Node.js on your machine. So you have to visit `https://nodejs.org` to download the latest version. You just download an installer and you can simply walk through that installer and confirm all the defaults.

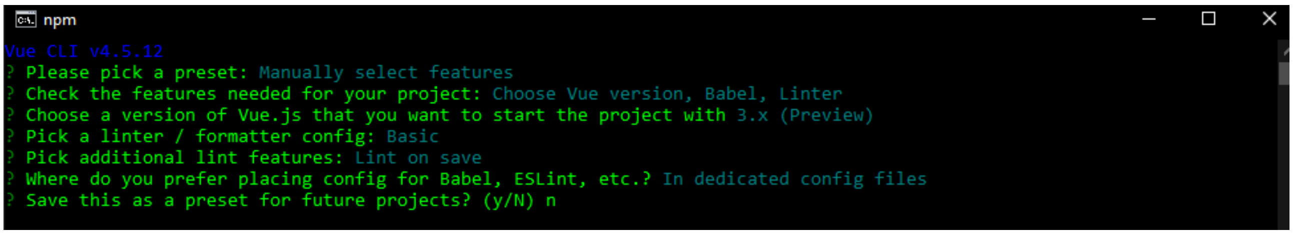
After installing the Node.js, then you install the Vue CLI by running the following command in the terminal or command prompt of your system:

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

Now you will create a new Vue project. You've found a place to contain your project, then you run `vue create`. For example, if you want to create a new project folder called `vue-first-app`, you just run:

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

Then, you will be prompted a couple of questions. You have to answer all the questions by following the guideline on your terminal. You can refer to the answers in the figure 2.



```
Vue CLI v4.5.12
? Please pick a preset: Manually select features
? Check the features needed for your project: Choose Vue version, Babel, Linter
? Choose a version of Vue.js that you want to start the project with 3.x (Preview)
? Pick a linter / formatter config: Basic
? Pick additional lint features: Lint on save
? Where do you prefer placing config for Babel, ESLint, etc.? In dedicated config files
? Save this as a preset for future projects? (y/N) n
```

Figure 2: Create a new project

The action takes a few minutes to succeed, then you can open the folder and start the development server:

```
npm run serve
```

By default, the app will run at `http://localhost:8080/`. If you use Visual Studio Code, there are some useful extensions to help you manage your code professionally. I recommend you to install:

- **Vetur**: Vue tooling for VSCode
- **Prettier**: Code formatter
- **Thunder Client**: Tool for testing Rest API

Let's start coding

2.1 Inside a new project

First of all, use Visual Code to open your generated folder. You can see all the defaults in there. The folder contains a bunch of files and also sub folders. Some folders or files you must care about:

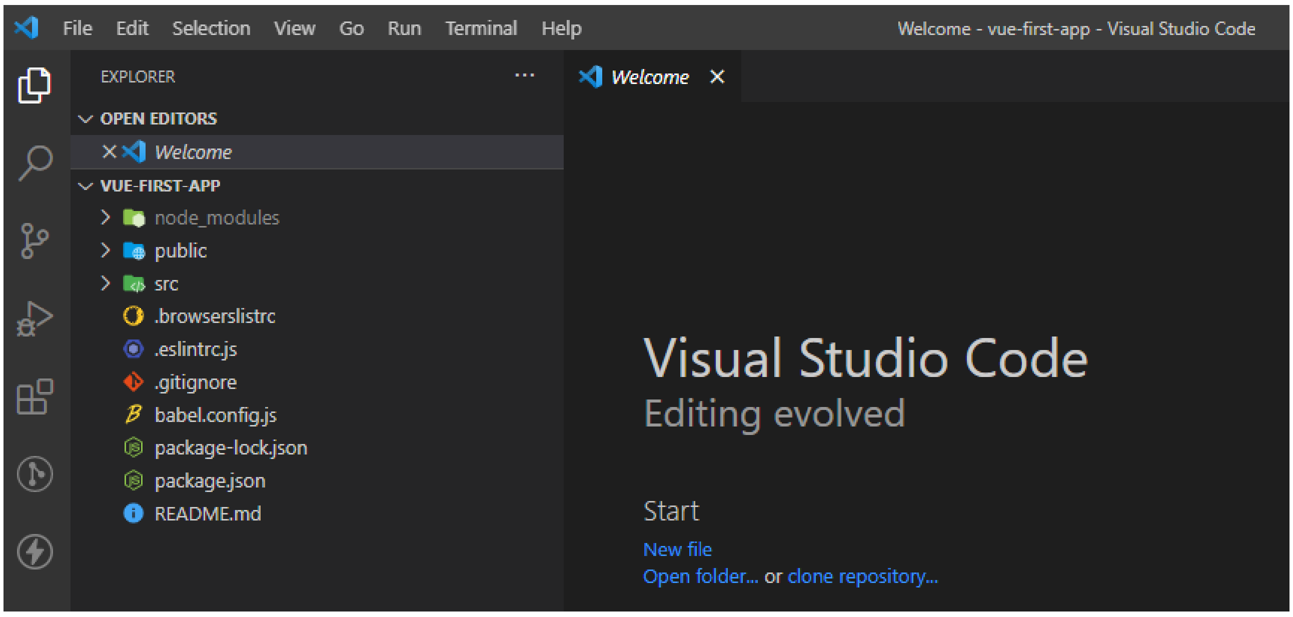


Figure 3: Folder structure

- The `package.json` file defines the project configuration and all dependencies
- The `node_modules` folder contains all dependencies of your project.
- The `src` folder contains your Vue code.

If you run the Vue project, it will load and runs the `main.js` in the `src` folder. In `main.js` file, the project creates a **Vue instance** with a configuration come form the `App.vue` file and mounts it to an element with an id `app`. You can find that id `app` in the `index.html` file in the `public` folder.

The Vue project contains many `.vue` files. The `.vue` file is the Vue component. Vue component have 3 parts:

- `template`: It contains the `HTML` elements.
- `script`: It defines data and all logic that component have to process.

- `style`. It contains all `CSS` or `SCSS` style.

For example, you can explore this structure in `App.vue`.

You have to delete all code in `App.vue`. You will write all other components in `components` folder. Now, your job begins.

3. Core Concepts

In previous section, we know each `.vue` files contain 3 parts: `template`, `script` and `style`. In this section, we will learn about basic structures in Vue framework to build a check list page.

3.1 Template Syntax

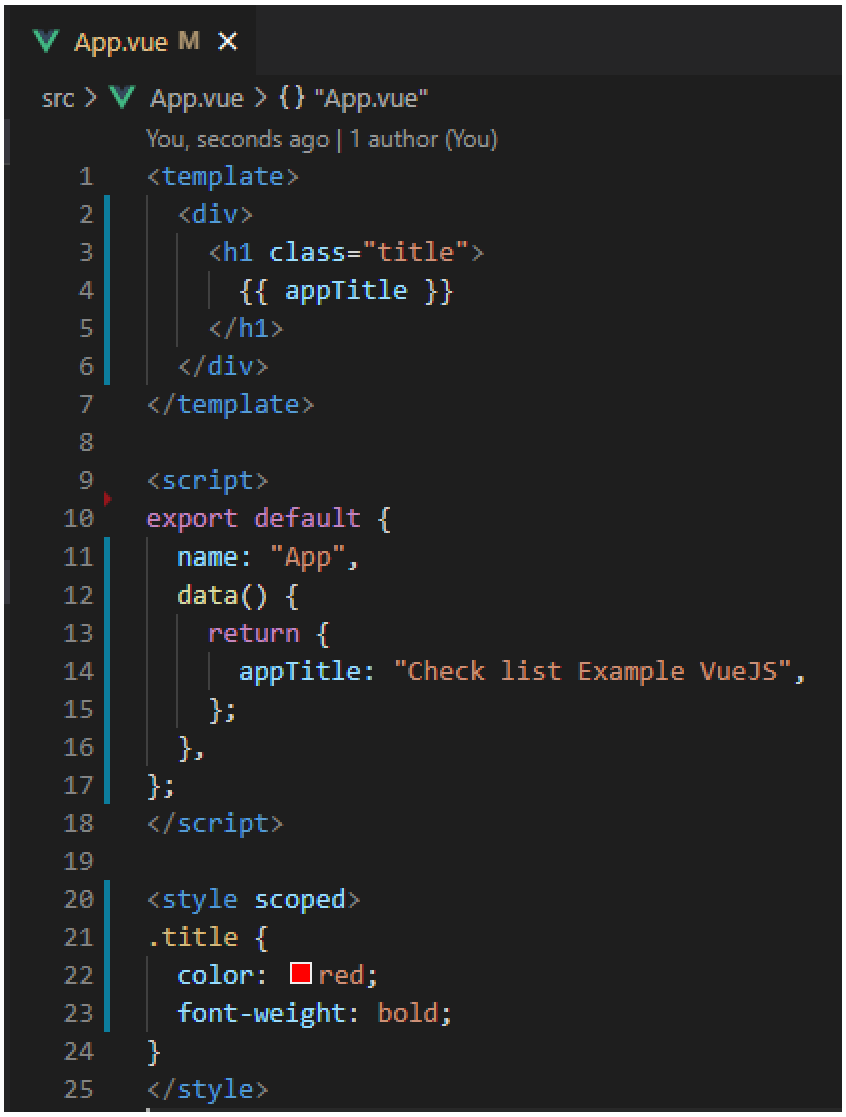
Vue.js uses an HTML-based template syntax that allows you to declaratively bind the rendered DOM to the underlying Vue instance's data. All Vue.js templates are valid HTML that can be parsed by spec-compliant browsers and HTML parsers. Under the hood, Vue compiles the templates into Virtual DOM render functions. Combined with the reactivity system, Vue is able to intelligently figure out the minimal number of components to re-render and apply the minimal amount of DOM manipulations when the app state changes. [2]

3.1.1 Text Interpolation

Let's type some first code:

And then you compile this code, we will see the title of your first Vue application in figure 5. What happened?

In `script` part, the **data** property have a function which returns an object. You can see the value in object exist in the `template` part. When you change the value in object, the view will update. This form is called **text interpolation**. If you want to display the values in **data** properties, write values in double curly braces (the "Mustache" syntax). When the values of **data** change, the view will re-render to match the new values.


App.vue M
src > App.vue > {} "App.vue"
You, seconds ago | 1 author (You)

```
<template>
  <div>
    <h1 class="title">
      {{ appTitle }}
    </h1>
  </div>
</template>

<script>
export default {
  name: "App",
  data() {
    return {
      appTitle: "Check list Example VueJS",
    };
  },
};
</script>

<style scoped>
.title {
  color: red;
  font-weight: bold;
}
</style>
```

Figure 4: Write first code

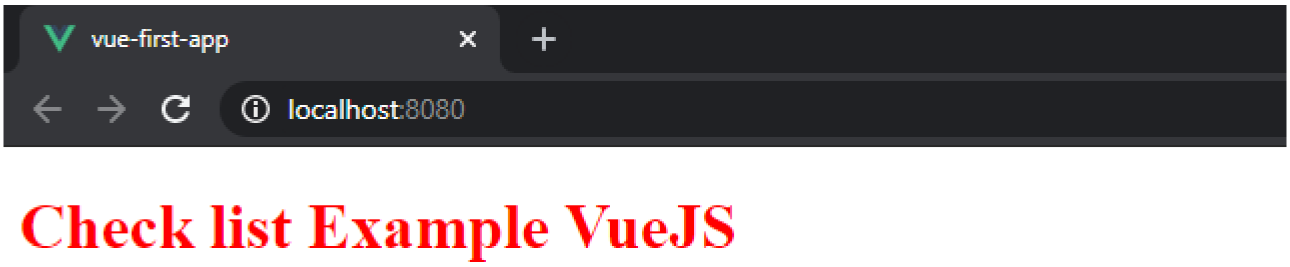
The values in Mustache syntax can also be Javascript expressions

```
{{number + 1}}
```

However, the Mustache syntax only contain one single expression, so the following will not work:

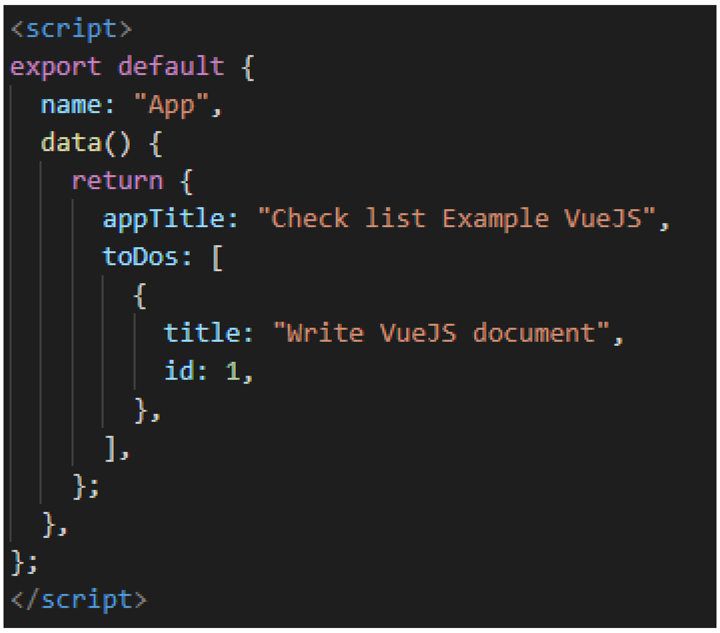
```
{{var a = 1}}
```

To create a to-do list, we just need to declare a value in the **data** property as shown. For instance, I assign an array with 1 object in **data** property to save all jobs we need to do (figure 6).



Check list Example VueJS

Figure 5: First Code complied in Chrome



```
<script>
export default {
  name: "App",
  data() {
    return {
      appTitle: "Check list Example VueJS",
      toDos: [
        {
          title: "Write VueJS document",
          id: 1,
        },
      ],
    };
  },
};
</script>
```

Figure 6:

3.1.2 Directives

Directives are special attributes with the `v-` prefix. Directive attribute values are expected to be a single JavaScript expression (with the exception of `v-for`, which will be discussed later). A directive's job is to reactively apply side effects to the DOM when the value of its expression changes.

Binding HTML attributes

If you want to bind the values in **data** to HTML attributes, you can use `v-bind` directive. For example, you want to set the `id` of a HTML element dynamically:

```
<div v-bind:id="dynamicId"></div>
```

Form

To add an element to the to-do list, we need a form to enter as shown:

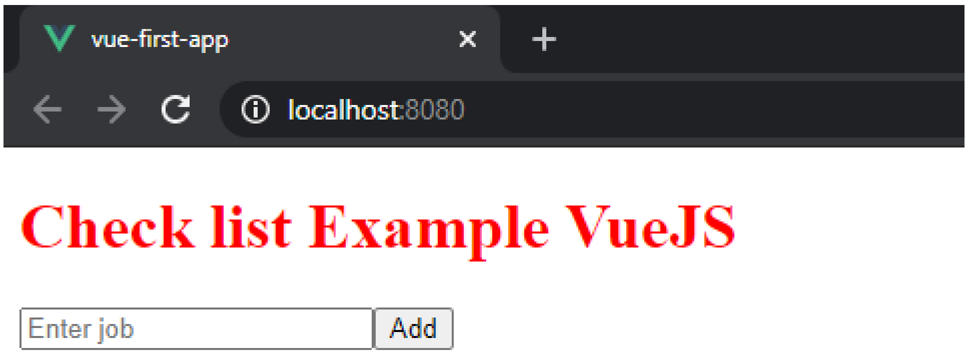
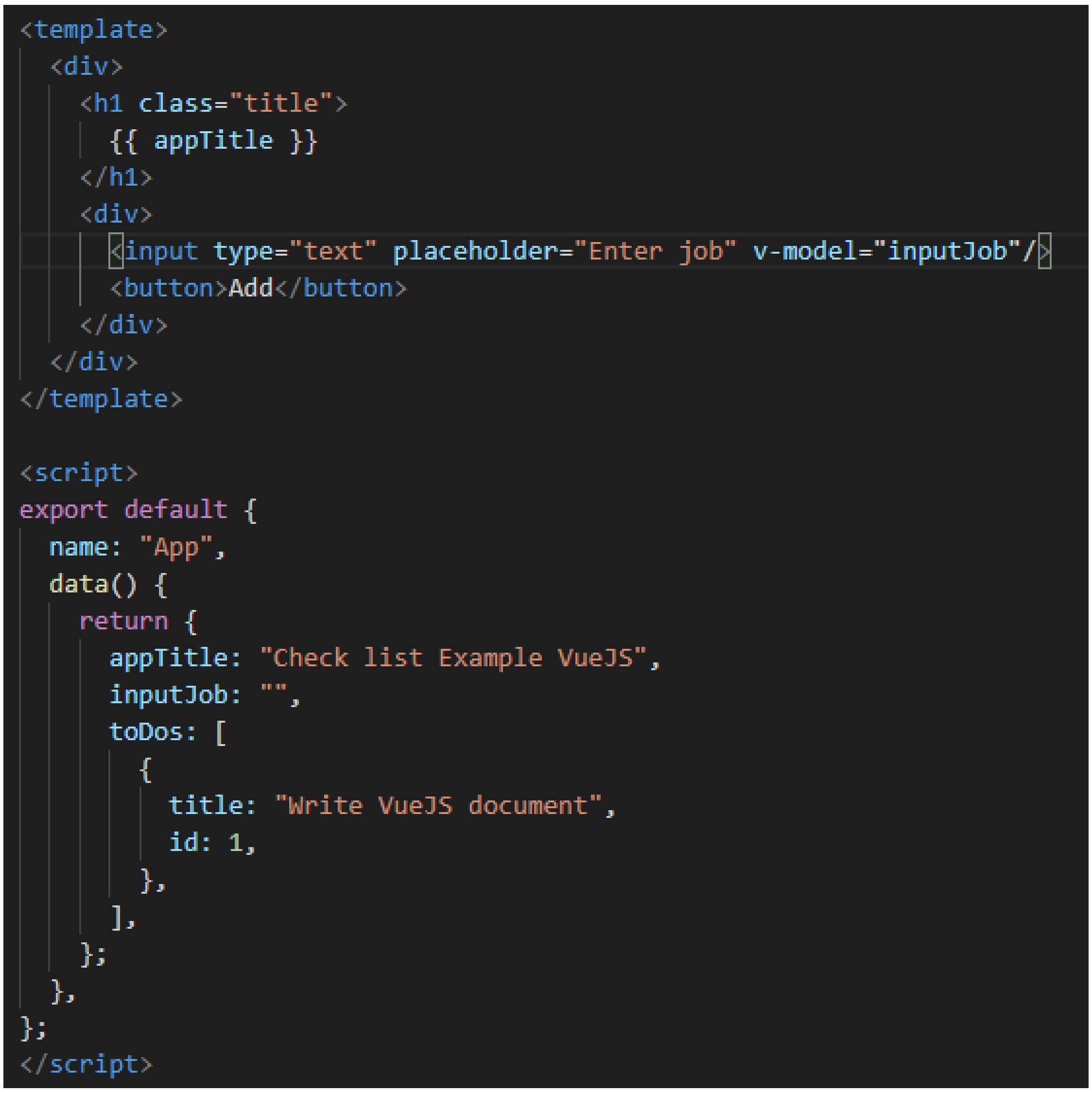


Figure 7: Input form

If we want to get value in input element, there are many ways, in this document we use `v-model` to bind input data into a value in **data** property (figure 8).



```
<template>
  <div>
    <h1 class="title">
      {{ appTitle }}
    </h1>
    <div>
      <input type="text" placeholder="Enter job" v-model="inputJob"/>
      <button>Add</button>
    </div>
  </div>
</template>

<script>
export default {
  name: "App",
  data() {
    return {
      appTitle: "Check list Example VueJS",
      inputJob: "",
      toDos: [
        {
          title: "Write VueJS document",
          id: 1,
        },
      ],
    };
  },
};
</script>
```

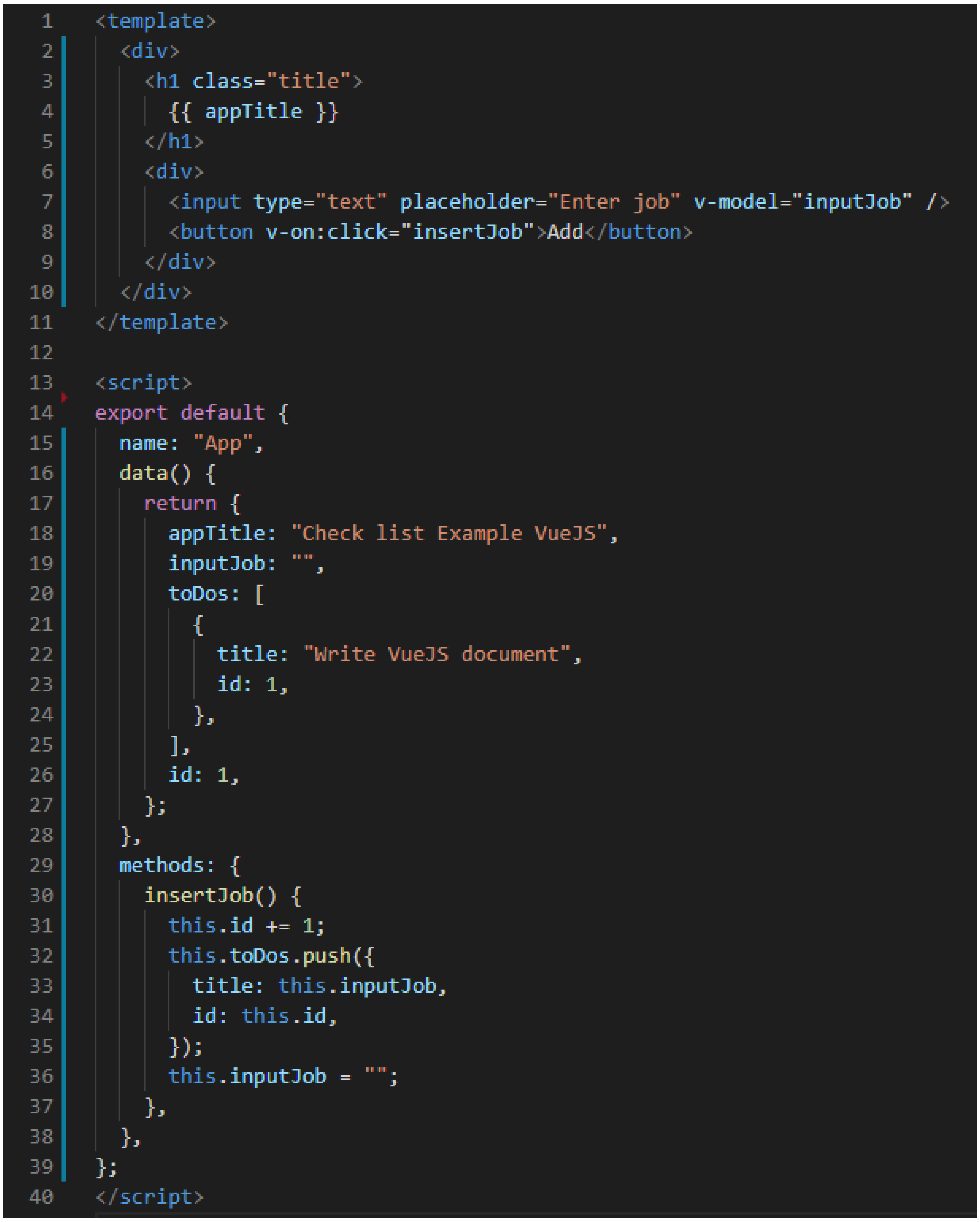
Figure 8: Use `v-model` in a form

Events Handling

Next, we have to catch the event when the ”Add” button is clicked. We use `v-on` with the name of the event placed in the **methods** property, an object containing all the functions used in this component.

In figure 9, we do a lot of things. Firstly, on line 8, we bind a function called `insertJob` by `v-on` with `click` event. Secondly, we create the `insertJob` in **methods** property. To get some values in the **data** as well as some functions in the **methods**, we will use `this` keyword. When the button is clicked, the `insertJob` function will be called and insert the value to the array in the **data** property.

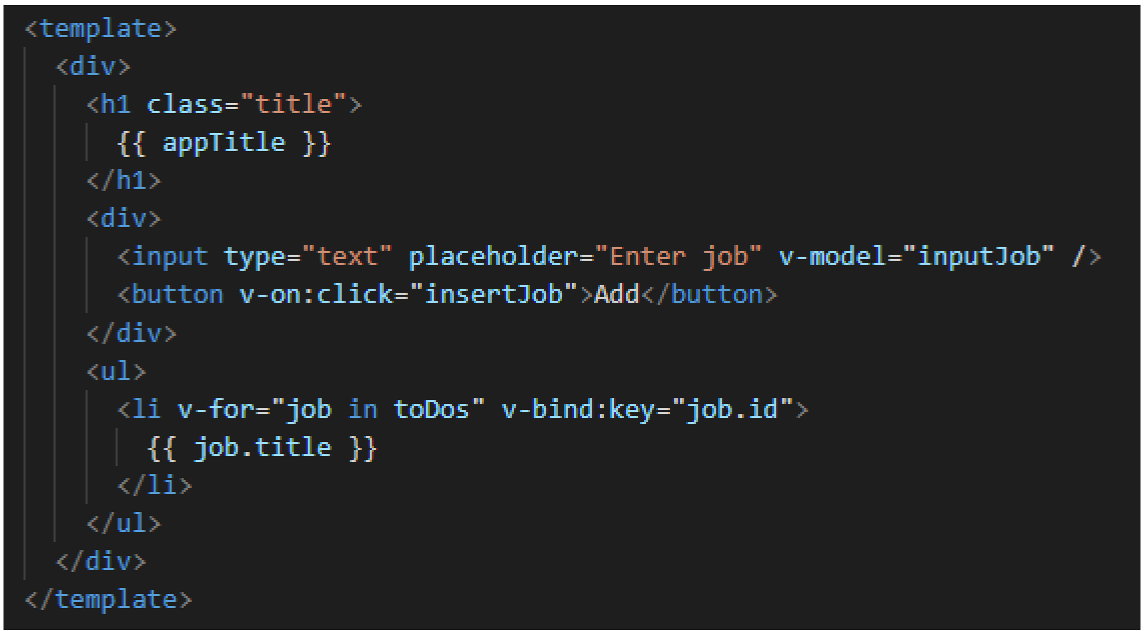
In addition to the click event, you can use `v-on` with events: `scroll`, `keyup`,...



```
<template>
  <div>
    <h1 class="title">
      {{ appTitle }}
    </h1>
    <div>
      <input type="text" placeholder="Enter job" v-model="inputJob" />
      <button v-on:click="insertJob">Add</button>
    </div>
  </div>
</template>

<script>
export default {
  name: "App",
  data() {
    return {
      appTitle: "Check list Example VueJS",
      inputJob: "",
      toDos: [
        {
          title: "Write VueJS document",
          id: 1,
        },
      ],
      id: 1,
    };
  },
  methods: {
    insertJob() {
      this.id += 1;
      this.toDos.push({
        title: this.inputJob,
        id: this.id,
      });
      this.inputJob = "";
    },
  },
};
</script>
```

Figure 9: Use `v-on` to catch click event

Now I want to display all the values in the array to the web page, because the array has a lot of different values so we will need `v-for` to generate the HTML automatically (figure 10).

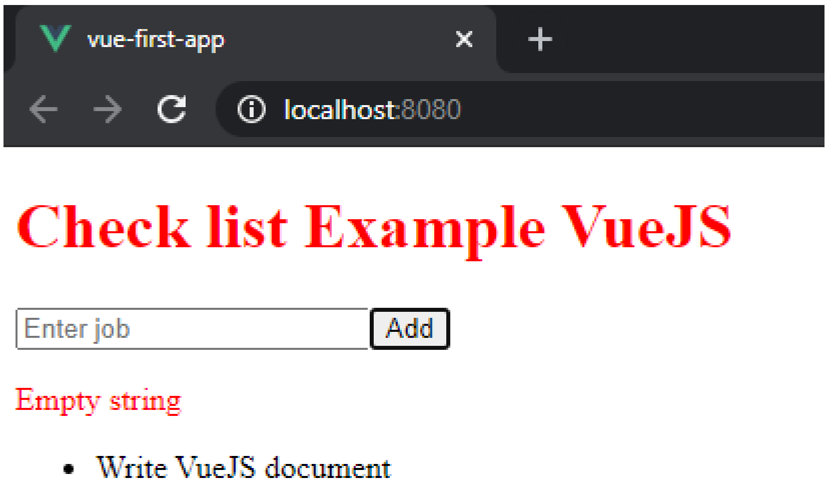


```
<template>
  <div>
    <h1 class="title">
      {{ appTitle }}
    </h1>
    <div>
      <input type="text" placeholder="Enter job" v-model="inputJob" />
      <button v-on:click="insertJob">Add</button>
    </div>
    <ul>
      <li v-for="job in toDos" v-bind:key="job.id">
        {{ job.title }}
      </li>
    </ul>
  </div>
</template>
```

Figure 10: Use `v-for` to display all jobs

Just like in ReactJS, you need to add a key to the element, which is usually the key of that element if taken from the API, restricting the assignment of id as the index of the array, because Vue page re-rendering mechanism will reuse the old elements already in the element tree.

Now, we want to validate the input value when click the button.



vue-first-app

localhost:8080

Check list Example VueJS

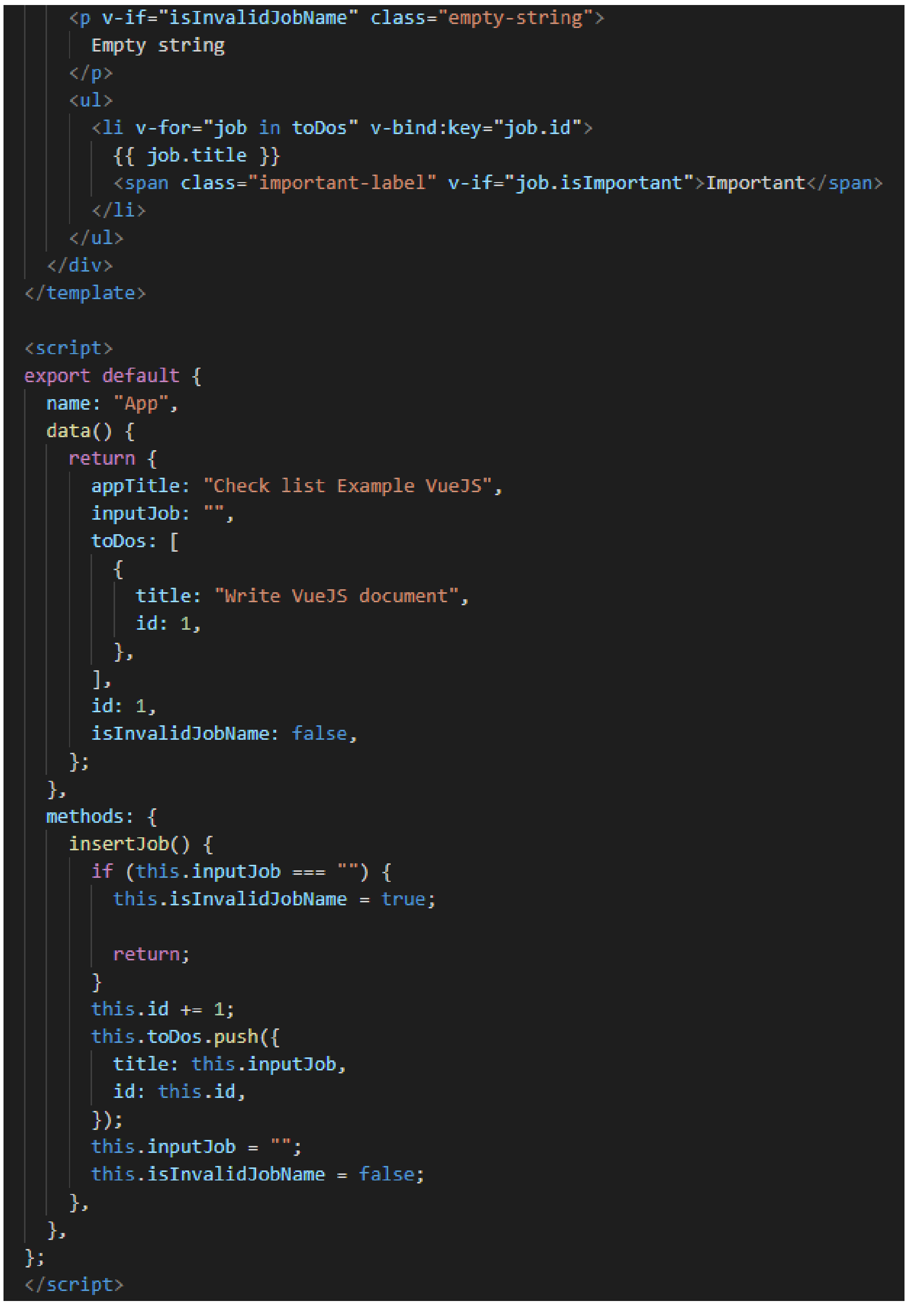
Enter job | Add

Empty string

- Write VueJS document

Figure 11: Validate empty value

Use `v-if` followed by an expression that returns `true` or `false`. If the expression is `true`, the element inside the if will be displayed, and `false` will not display the element.



```
      <p v-if="isInvalidJobName" class="empty-string">
        Empty string
      </p>
      <ul>
        <li v-for="job in toDos" v-bind:key="job.id">
          {{ job.title }}
          <span class="important-label" v-if="job.isImportant">Important</span>
        </li>
      </ul>
    </div>
</template>

<script>
export default {
  name: "App",
  data() {
    return {
      appTitle: "Check list Example VueJS",
      inputJob: "",
      toDos: [
        {
          title: "Write VueJS document",
          id: 1,
        },
      ],
      id: 1,
      isInvalidJobName: false,
    };
  },
  methods: {
    insertJob() {
      if (this.inputJob === "") {
        this.isInvalidJobName = true;

        return;
      }
      this.id += 1;
      this.toDos.push({
        title: this.inputJob,
        id: this.id,
      });
      this.inputJob = "";
      this.isInvalidJobName = false;
    },
  },
};
</script>
```

Figure 12: Use `v-if` to display validated message

Vue also supports the `v-else` structure immediately after the element containing the `v-if`. I would use `v-if` and `v-else` as follows:

```
<div v-if="Math.random() > 0.5"> Now you see me </div>
<div v-else> Now you don't </div>
```

In addition, Vue also supports nested `if`, `else` structure as follows:

```
<div v-if="type === 'A'"> A </div>
<div v-else-if="type === 'B'"> B </div>
<div v-else-if="type === 'C'"> C </div>
<div v-else> Not A/B/C </div>
```

When using `v-if`, with the `true` expression, the element will be added to the tree to be displayed, and if the value is `false`, the DOM will remove that element from the tree. This causes reduced rendering performance for the page.

There is another way, we can use that is to use `v-show`. Like `v-if`, `v-show` takes an expression that returns `true` or `false` if `true` will display the element, and `false` will not. Unlike `v-if`, `v-show` does not have `v-else` with it. In terms of performance, `v-show` simply adds the CSS style `display:none`, which hides the element, but stays on the DOM tree.

Vue provides special shorthands for two of the most often used directives:

- `v-bind`:

  ```
  <!-- full syntax -->
  <a v-bind:href="url"> ... </a>
  <!-- shorthand -->
  <a :href="url"> ... </a>
  ```

- `v-on`:

  ```
  <!-- full syntax -->
  <a v-on:click="doSomething"> ... </a>
  <!-- shorthand -->
  <a @click="doSomething"> ... </a>
  ```

3.2 Interact with Data

Vue provides four basic methods to control data that are

- data: contains data or state of component
- methods: contains functions used in component

- watch
- computed

The first two methods we already know in the previous section, in this part we will learn more about the other two methods.

3.2.1 Computed

```
computed: {
aDouble: vm => vm.a * 2
}
```

In **computed** properties, there are functions that do a certain job. **Computed** properties are cached, and only re-computed on reactive dependency changes. So, we often use computed when one property in the data needs to change when two or more other properties in the data change. Why is there no 1, the answer will be in the watcher subsection.

You can call methods in computed through the syntax:

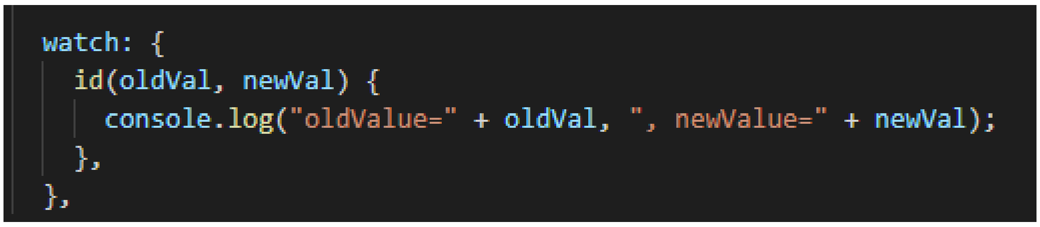
```
<p>{{aDouble}}</p>
```

Even though `aDouble` is a function, you can't add parentheses after the method's name. People often use **computed** when they want to change the style of an element.

3.2.2 Watcher

Watch is an object where keys are expressions to watch and values are the corresponding callbacks. If you want to automatically do something when a value in the **data** changes, use watcher. If you want to log value of `id`, use a watcher as follow:

The names of the methods must match the change value.

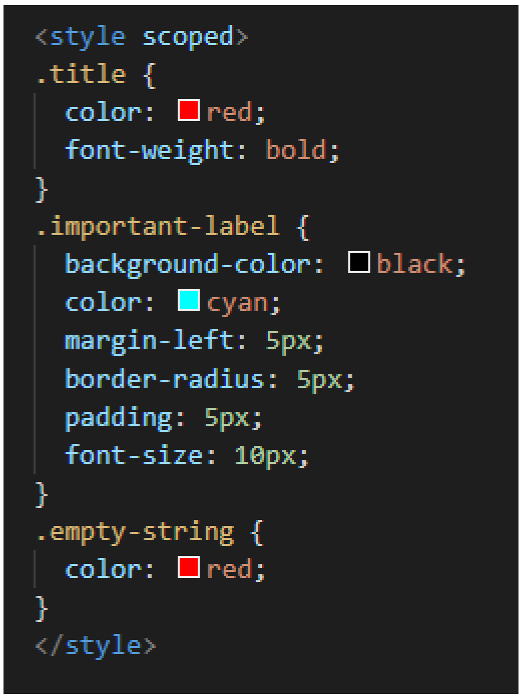


```
watch: {
  id(oldVal, newVal) {
    console.log("oldValue=" + oldVal, ", newValue=" + newVal);
  },
},
```

Figure 13: Use a watcher

3.3 Dynamic Styling and Class

In this section, we will talk about styling the elements of a component. Firstly, we should know that, without the `scoped` attribute, classes will be visible in all of Vue application. That means declaring a class `a` in component `A`, then in component `B` can also use that class `a`. Therefore, to be able to apply the class only to one component, the `scoped` attribute must be added to the `<style>` tag.



```
<style scoped>
.title {
  color: red;
  font-weight: bold;
}
.important-label {
  background-color: black;
  color: cyan;
  margin-left: 5px;
  border-radius: 5px;
  padding: 5px;
  font-size: 10px;
}
.empty-string {
  color: red;
}
</style>
```

Figure 14: Add `scoped` to `<style>` tag

3.3.1 Dynamic Styling, Inline Style

Sometimes, we want to apply CSS style to a specific element, we can use `v-bind` to bind the values in the **data** property.

For example, applying color to the border of a `div` tag we could do as the line of code below, where the value `borderColor` is a value in the **data** property.

```
<div :style="'border-color': borderColor">Example</div>
```

We can replace `borderColor` with an expression containing the conditional operator, or a value in **computed**. In addition to the above usage, we can use the array syntax (See dynamic class subsection).

However, you must limit the use of inline styles because these will overrule all other style values of that element.

3.3.2 Dynamic Class

Classes can also be dynamically binded in three ways:

- *Way 1:* Use expressions contain conditional operator:

  ```
  <div :class="isChange ? 'class-a' : 'class-b'">
  Example
  </div>
  ```

  Where `isChange` is a value in **data**, and `class-a` and `class-b` are classes defined in the `<style>` tag.

- *Way 2:* Use boolean expression:

  ```
  <div :class="'class-a' : true">Example</div>
  ```

- *Way 3:* Use array syntax:

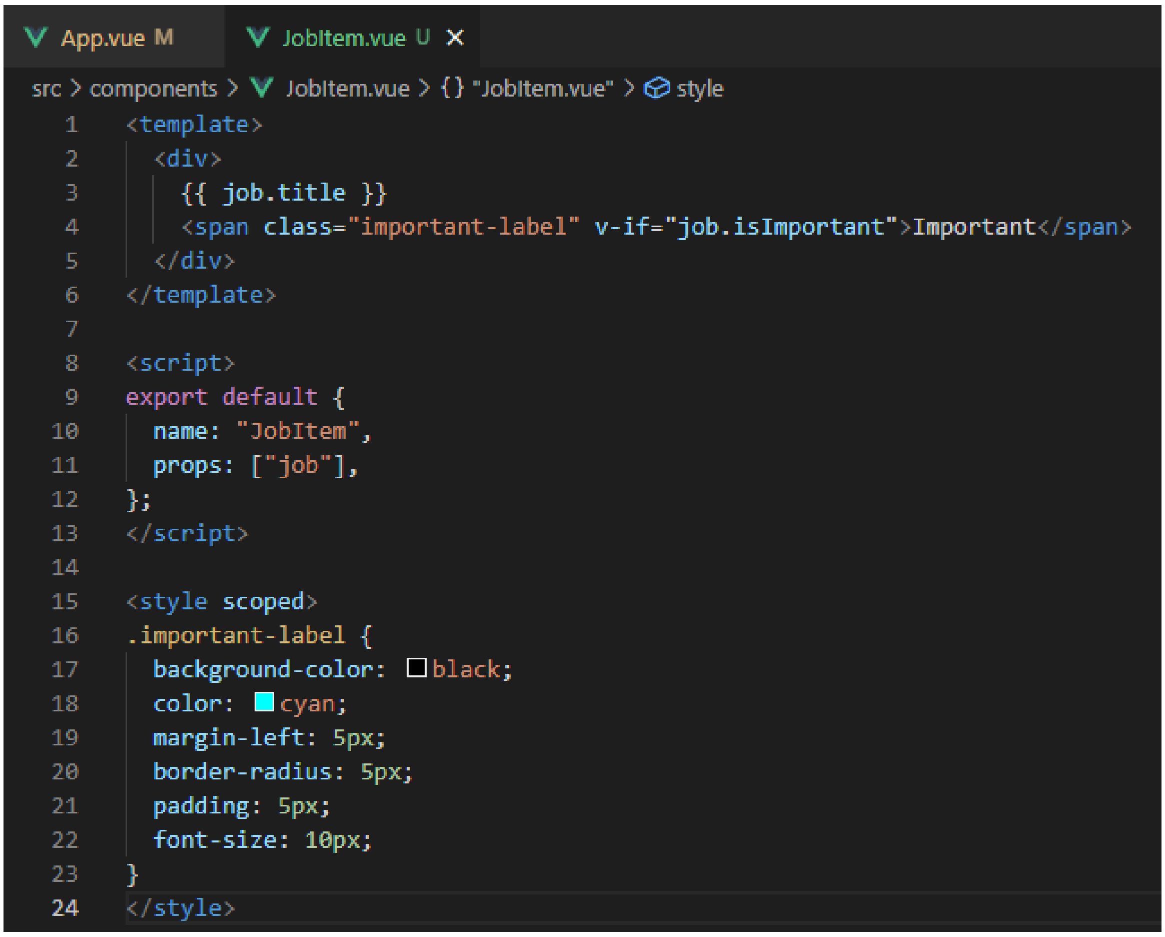
  ```
  <div :class="['class-a']">Example</div>
  ```

4. Components

A Vue application can contain many components, each component is a file `.vue`. Componenent makes it easier to reuse code and manage HTML elements.

4.1 Parent-child Communication

In the figure 10, we can separate the contents of the `<li>` tags into a component named JobItem which is contained in the file `JobItem.vue` (located in the `component` folder).

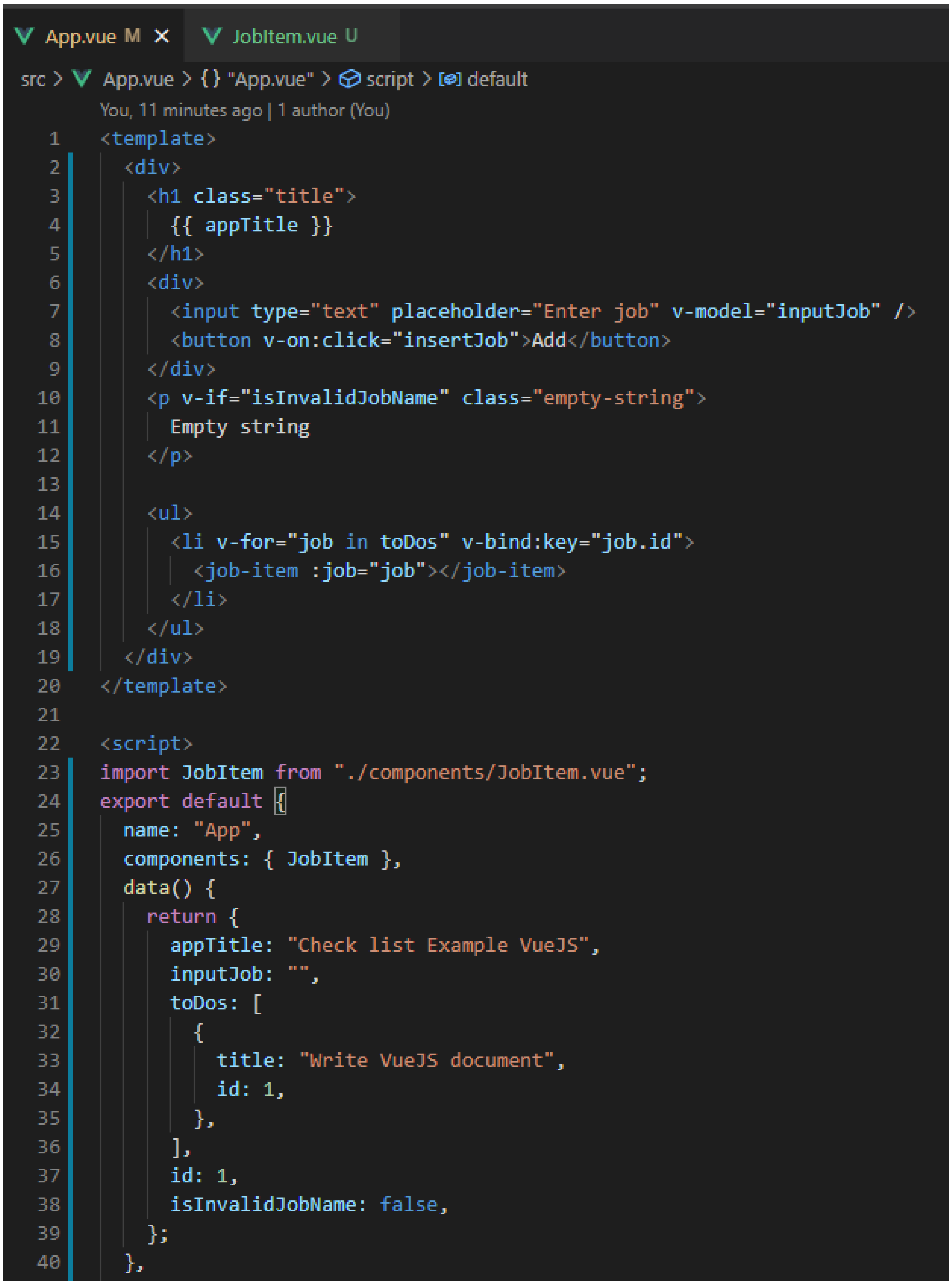


```
<template>
  <div>
    {{ job.title }}
    <span class="important-label" v-if="job.isImportant">Important</span>
  </div>
</template>

<script>
export default {
  name: "JobItem",
  props: ["job"],
};
</script>

<style scoped>
.important-label {
  background-color: black;
  color: cyan;
  margin-left: 5px;
  border-radius: 5px;
  padding: 5px;
  font-size: 10px;
}
</style>
```

Figure 15: File `JobItem.vue`

Figure 15, we see some important things when copying from `App.vue`. You can name the component using the **name** option. The component needs to use `job` passed, so we have to declare the values passed in through the **props** option.

Back to the `App.vue` file, there are certain modifications as shown in the figure 16. To use the component `JobItem`, we must import it and declare the component in **components** option. In the `<template>` tag, we edit to add JobItem and add a value for the `job` attribute with `v-bind`.


App.vue M ×
JobItem.vue U
src > App.vue > {} "App.vue" > script > default
You, 11 minutes ago | 1 author (You)

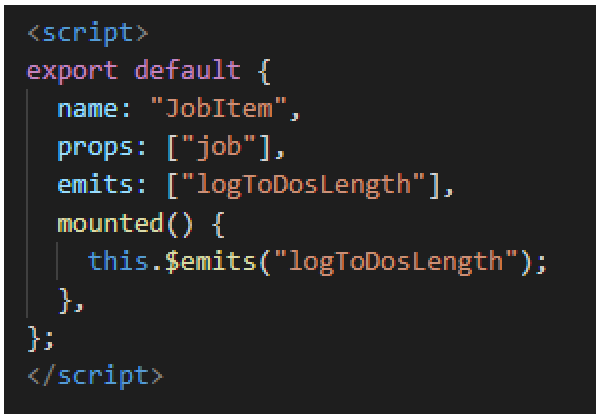
```
<template>
  <div>
    <h1 class="title">
      {{ appTitle }}
    </h1>
    <div>
      <input type="text" placeholder="Enter job" v-model="inputJob" />
      <button v-on:click="insertJob">Add</button>
    </div>
    <p v-if="isInvalidJobName" class="empty-string">
      Empty string
    </p>

    <ul>
      <li v-for="job in toDos" v-bind:key="job.id">
        <job-item :job="job"></job-item>
      </li>
    </ul>
  </div>
</template>

<script>
import JobItem from "./components/JobItem.vue";
export default {
  name: "App",
  components: { JobItem },
  data() {
    return {
      appTitle: "Check list Example VueJS",
      inputJob: "",
      toDos: [
        {
          title: "Write VueJS document",
          id: 1,
        },
      ],
      id: 1,
      isInvalidJobName: false,
    };
  },
```

Figure 16: Use `JobItem` in file `App.vue`

4.2 Child-Parent Communication

Vue does not allow data in the parent component to be changed by changing the value passed from parent to child. Therefore, to change the data already in the parent component, we will use the **emits** option. The emits declaration is the same as **props**, the names in **emits** are the names of the function.

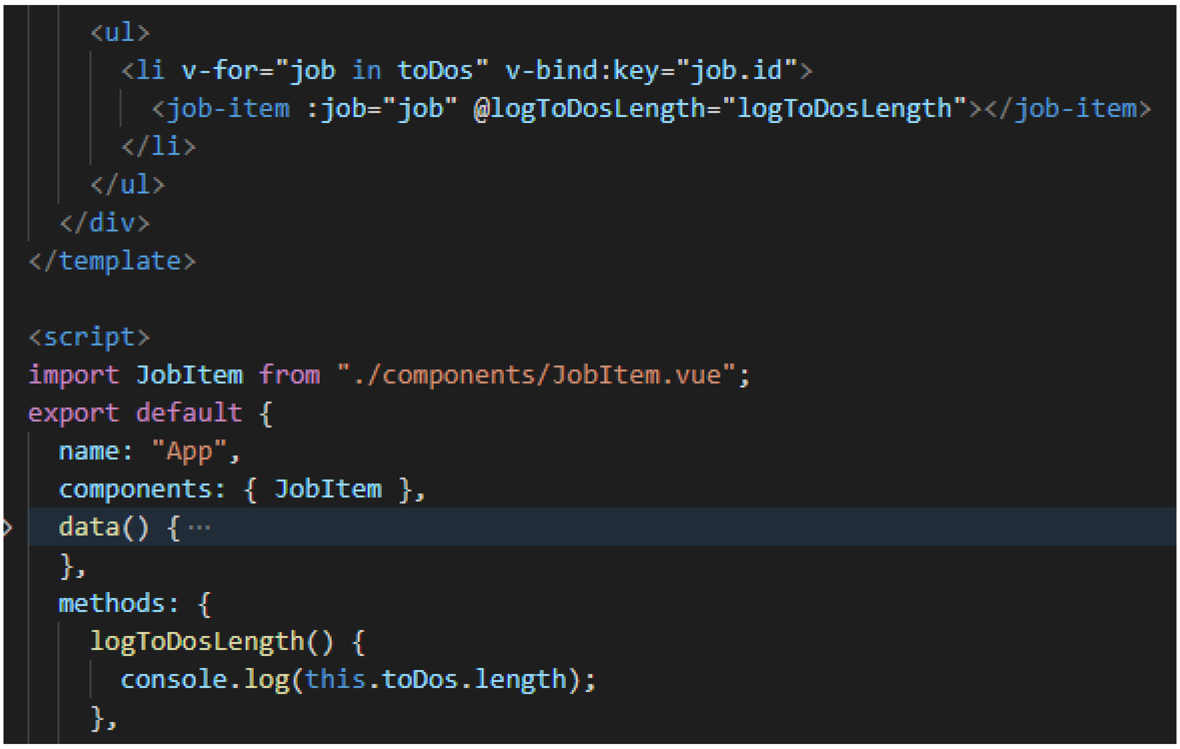
For example, if we want to log the number of jobs to be performed in the `JobItem` component, we need to declare the following:



```
<script>
export default {
  name: "JobItem",
  props: ["job"],
  emits: ["logToDosLength"],
  mounted() {
    this.$emits("logToDosLength");
  },
};
</script>
```

Figure 17:

For `App.vue`, we will bind a function that prints the number of jobs in **methods** by `v-on`.



```
      <ul>
        <li v-for="job in toDos" v-bind:key="job.id">
          <job-item :job="job" @logToDosLength="logToDosLength"></job-item>
        </li>
      </ul>
    </div>
</template>

<script>
import JobItem from "./components/JobItem.vue";
export default {
  name: "App",
  components: { JobItem },
  data() {···
  },
  methods: {
    logToDosLength() {
      console.log(this.toDos.length);
    },
```

Figure 18:

4.3 Provide and Inject

This pair of options are used together to allow an ancestor component to serve as a dependency injector for all its descendants, regardless of how deep the component hierarchy is, as long as they are in the same parent chain.

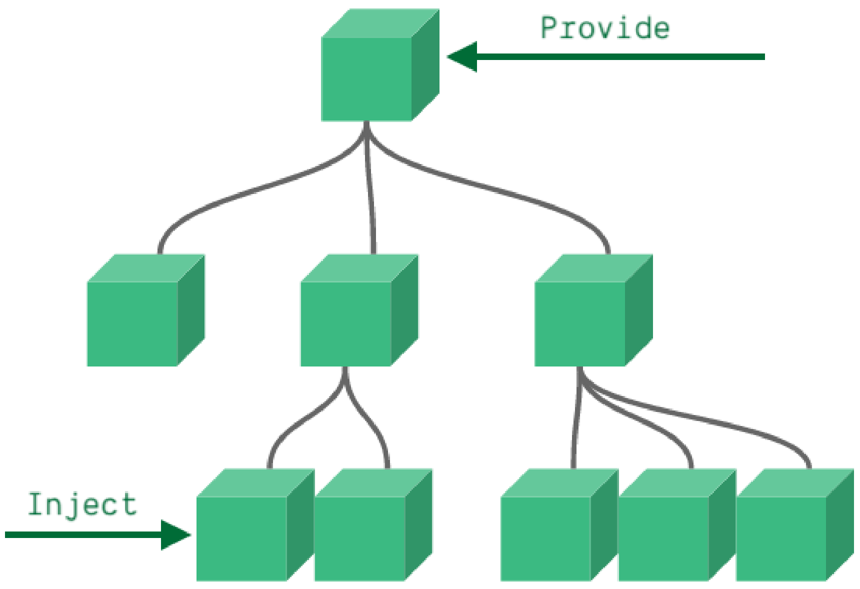


Figure 19: Provide and Inject demonstration

The **provide** option should be an object or a function that returns an object. This object contains the values that are available for injection into its descendants.

```
 // parent component providing 'foo'
provide:{ foo: 'bar' }
```

The **inject** option is defined as same as the **props** option.

```
 // child component injecting 'foo'
inject: ['foo'],
created () {
console.log(this.foo);
}
```

4.4 Lifecycle

Each component in VueJS has a similar lifecycle as shown in the figure 20.

If you want to do something with some stage of the component's lifecycle, you can use the functions shown in the red box in the figure 20.

For example, if you want to load data from the server to save before the component is rendered into a DOM tree, you can simply implement the mounted function and put this function in the component, the order in which it is placed in a component doesn't matter.

5. Summary

In this document, you have done the following:

- Install and use Vue CLI.
- Use simple syntax in Vue framework like v-if, v-on, v-for,...
- Learn about using components through the four properties in a component: data, watch, methods, computed, and also understand parent-child relationships between components.

6. Read more

In addition to the basic knowledge in this document, when working on complex projects in the future, you need to learn more about the following issues:

- Slots: allow you to compose components.
- Routing: by using Vue Router. [1]
- Vuex : serves as a centralized store for all the components in an application, with rules ensuring that the state can only be mutated in a predictable fashion. [3]
- HTTP request

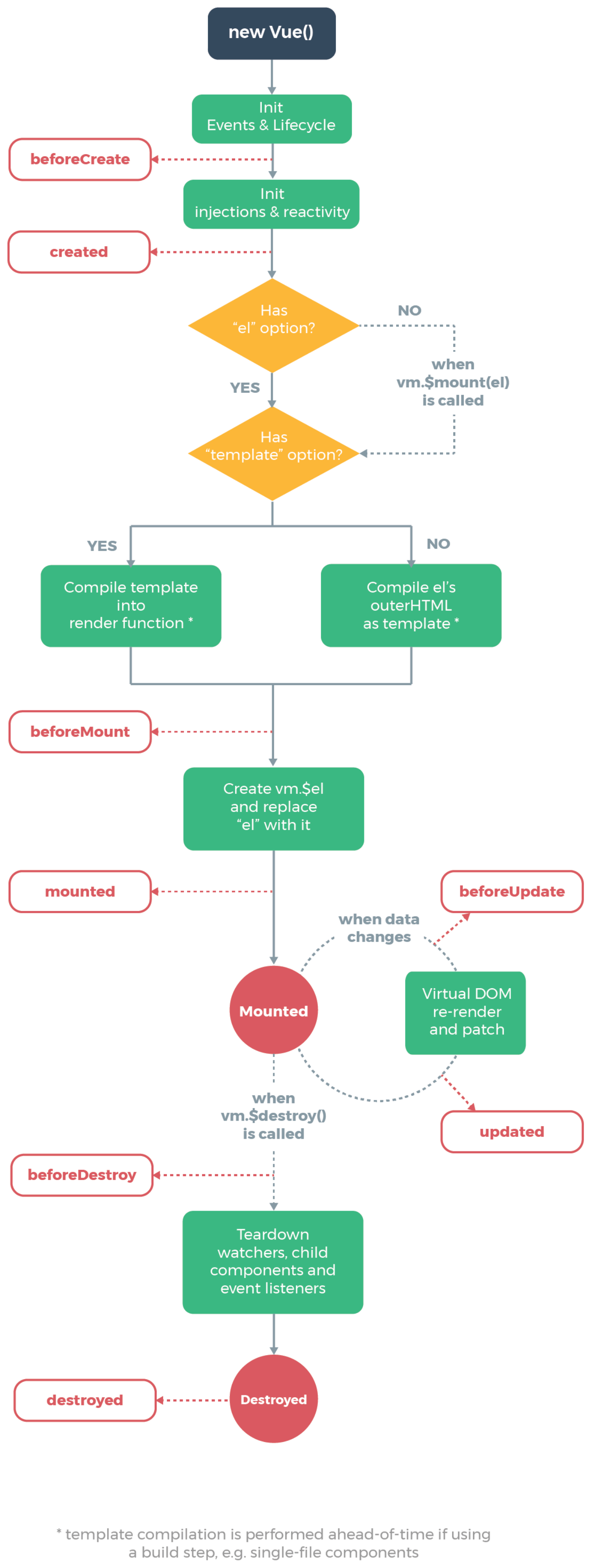

new Vue()
Init Events & Lifecycle
beforeCreate
Init injections & reactivity
created
Has "el" option?
NO
when vm.$mount(el) is called
YES
Has "template" option?
YES
NO
Compile template into render function *
Compile el's outerHTML as template *
beforeMount
Create vm.$el and replace "el" with it
mounted
beforeUpdate
when data changes
Mounted
Virtual DOM re-render and patch
when vm.$destroy() is called
updated
beforeDestroy
Teardown watchers, child components and event listeners
destroyed
Destroyed
* template compilation is performed ahead-of-time if using a build step, e.g. single-file components

Figure 20: The lifecycle diagram

- Composition API
- Mixins: distribute reusable functionalities for Vue components.

References

[1] Evan You. *Vue Router Guide*. URL: https://router.vuejs.org/guide/#javascript. (Accessed: 28 May, 2021).

[2] Evan You. *Vue.js Document*. URL: https://vuejs.org/v2/guide/. (Accessed: 28 May, 2021).

[3] Evan You. *What is Vuex?* URL: https://vuex.vuejs.org/#what-is-a-state-management-pattern. (Accessed: 28 May, 2021).