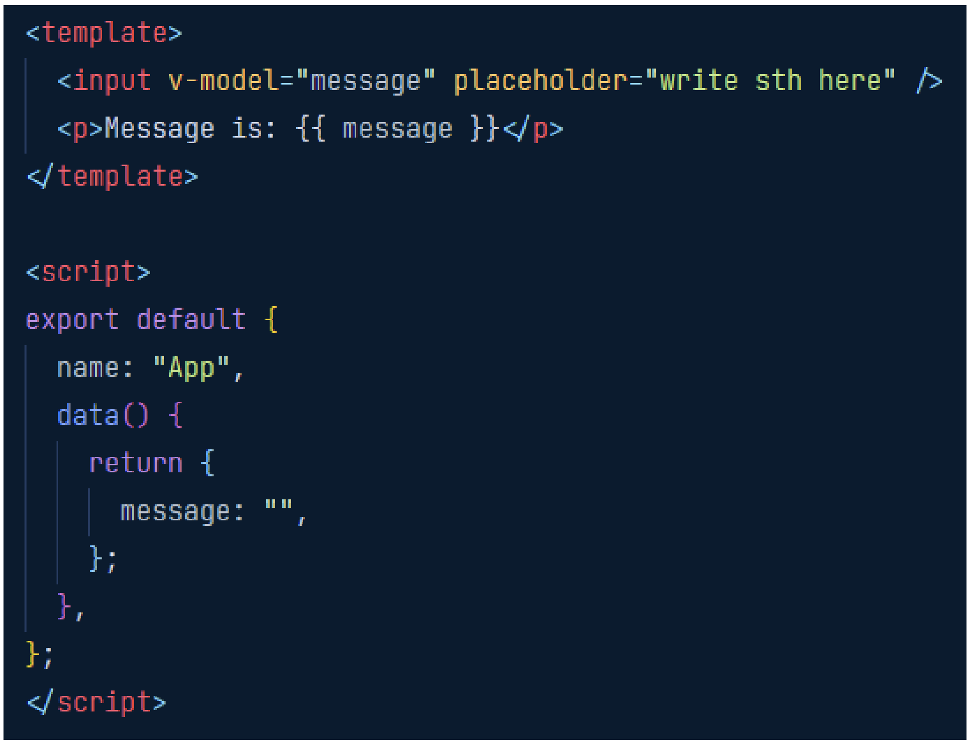
Advanced Web Programming

Lab7 - Vue - Forms, Vue Router

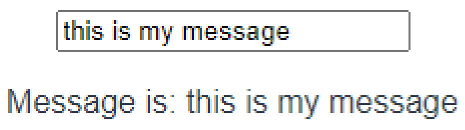
Section 1. Forms

Form Input Bindings

You can use the v-model directive to create two-way data bindings on form input, textarea, and select elements. It automatically picks the correct way to update the element based on the input type. Although a bit magical, v-model is essentially syntax sugar for updating data on user input events, plus special care for some edge cases.



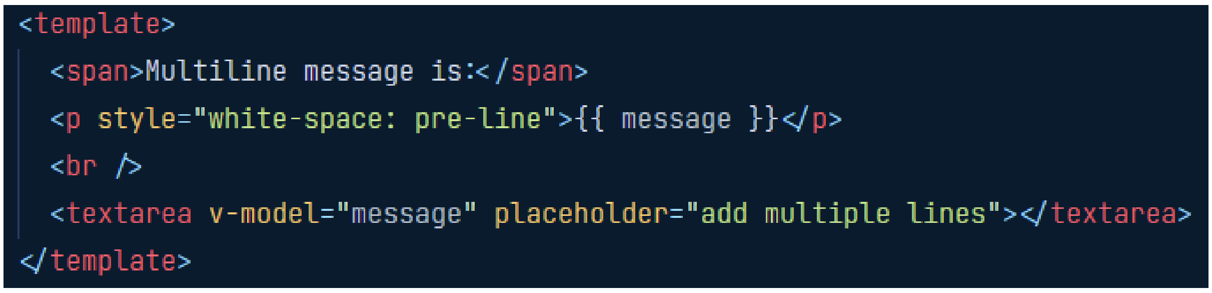
```
<template>
  <input v-model="message" placeholder="write sth here" />
  <p>Message is: {{ message }}</p>
</template>

<script>
export default {
  name: "App",
  data() {
    return {
      message: "",
    };
  },
};
</script>
```



```
this is my message

Message is: this is my message
```

If you want to provide initial value , you should declare it on the JavaScript side, inside the data option of your component.

Multiline text

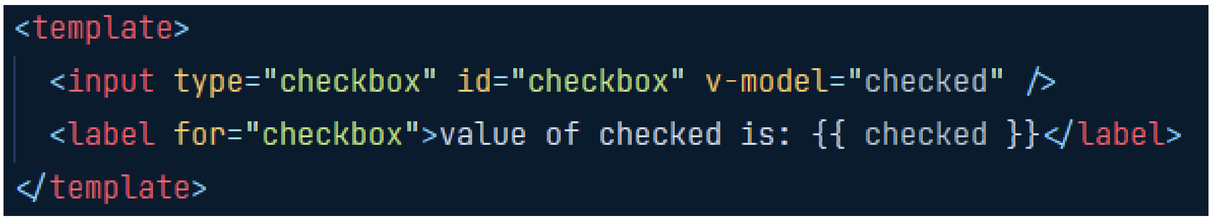


```
<template>
  <span>Multiline message is:</span>
  <p style="white-space: pre-line">{{ message }}</p>
  <br />
  <textarea v-model="message" placeholder="add multiple lines"></textarea>
</template>
```

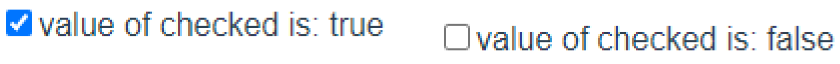
It is mandatory for element displaying multiline text to have style attribute
white-space: pre-line | pre | pre-wrap

Single checkbox

Single checkbox value is represented with true / false value in data()



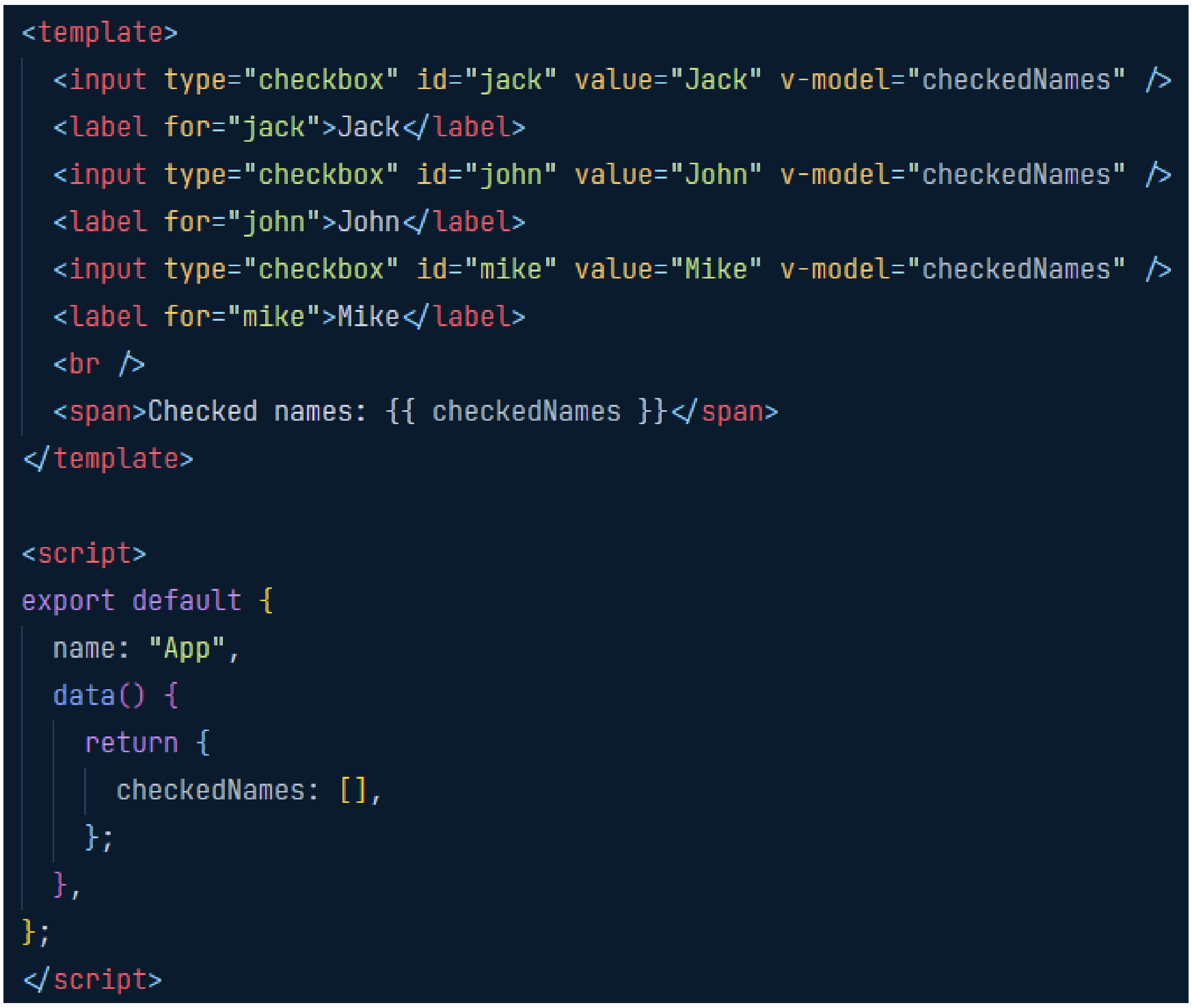
```
<template>
  <input type="checkbox" id="checkbox" v-model="checked" />
  <label for="checkbox">value of checked is: {{ checked }}</label>
</template>
```



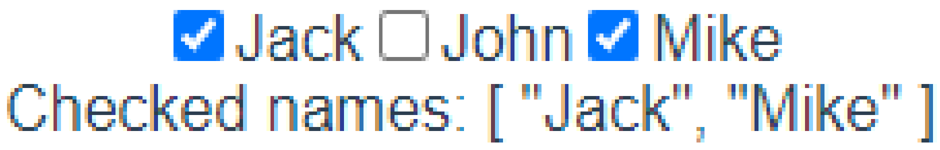
☑ value of checked is: true ☐ value of checked is: false

Multiple checkboxes

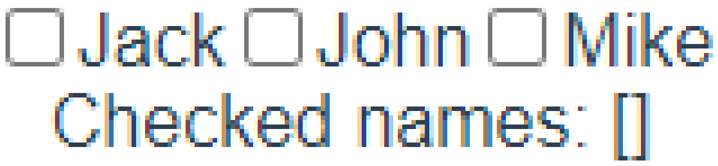
Multiple checkboxes, bound to the same array:



```
<template>
  <input type="checkbox" id="jack" value="Jack" v-model="checkedNames" />
  <label for="jack">Jack</label>
  <input type="checkbox" id="john" value="John" v-model="checkedNames" />
  <label for="john">John</label>
  <input type="checkbox" id="mike" value="Mike" v-model="checkedNames" />
  <label for="mike">Mike</label>
  <br />
  <span>Checked names: {{ checkedNames }}</span>
</template>

<script>
export default {
  name: "App",
  data() {
    return {
      checkedNames: [],
    };
  },
};
</script>
```

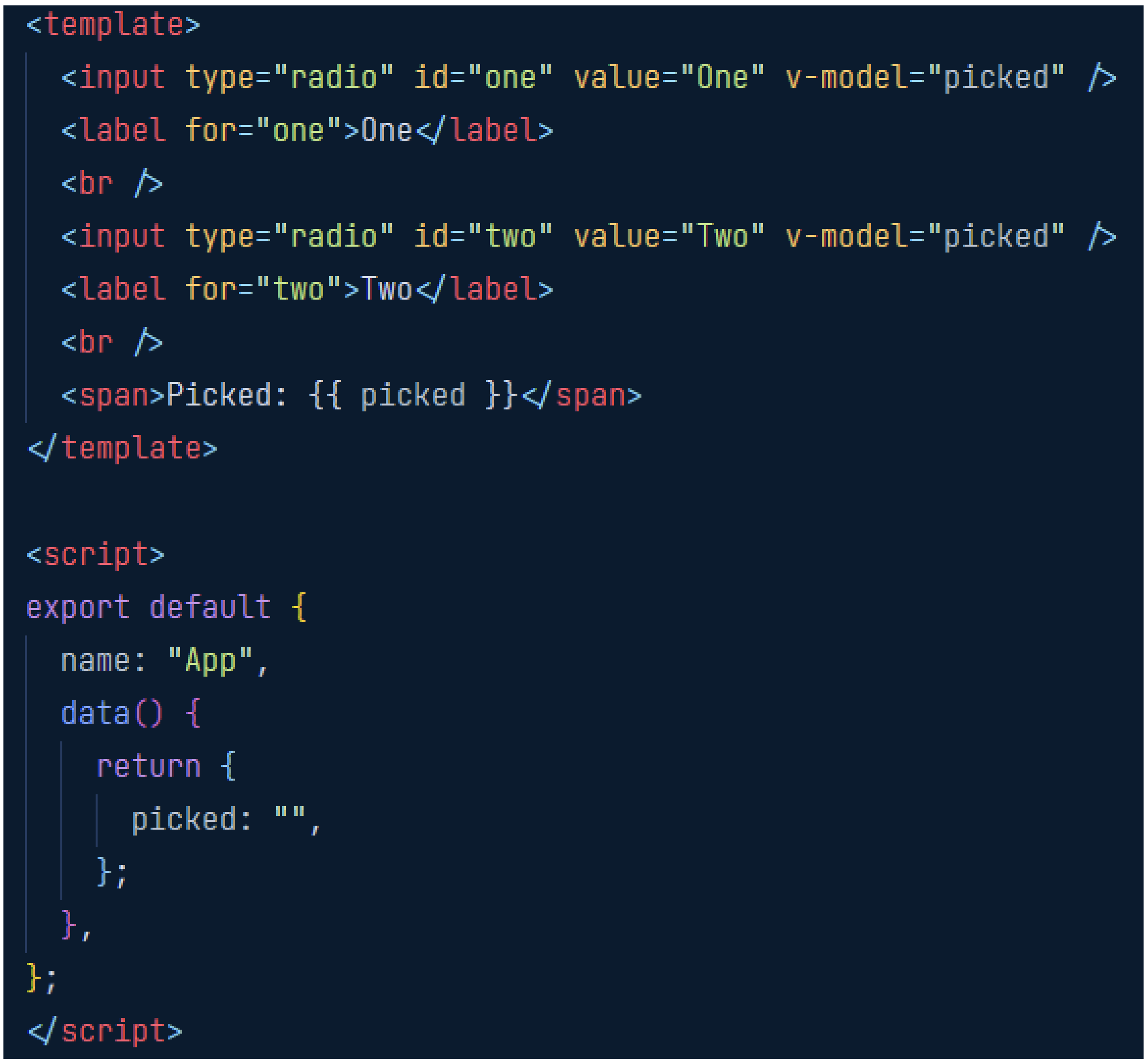


```
☑ Jack ☐ John ☑ Mike
Checked names: [ "Jack", "Mike" ]
```

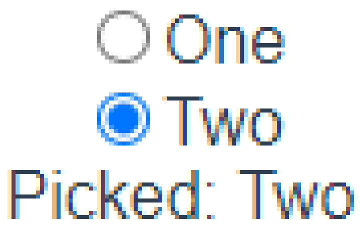


```
☐ Jack ☐ John ☐ Mike
Checked names: []
```

Radio

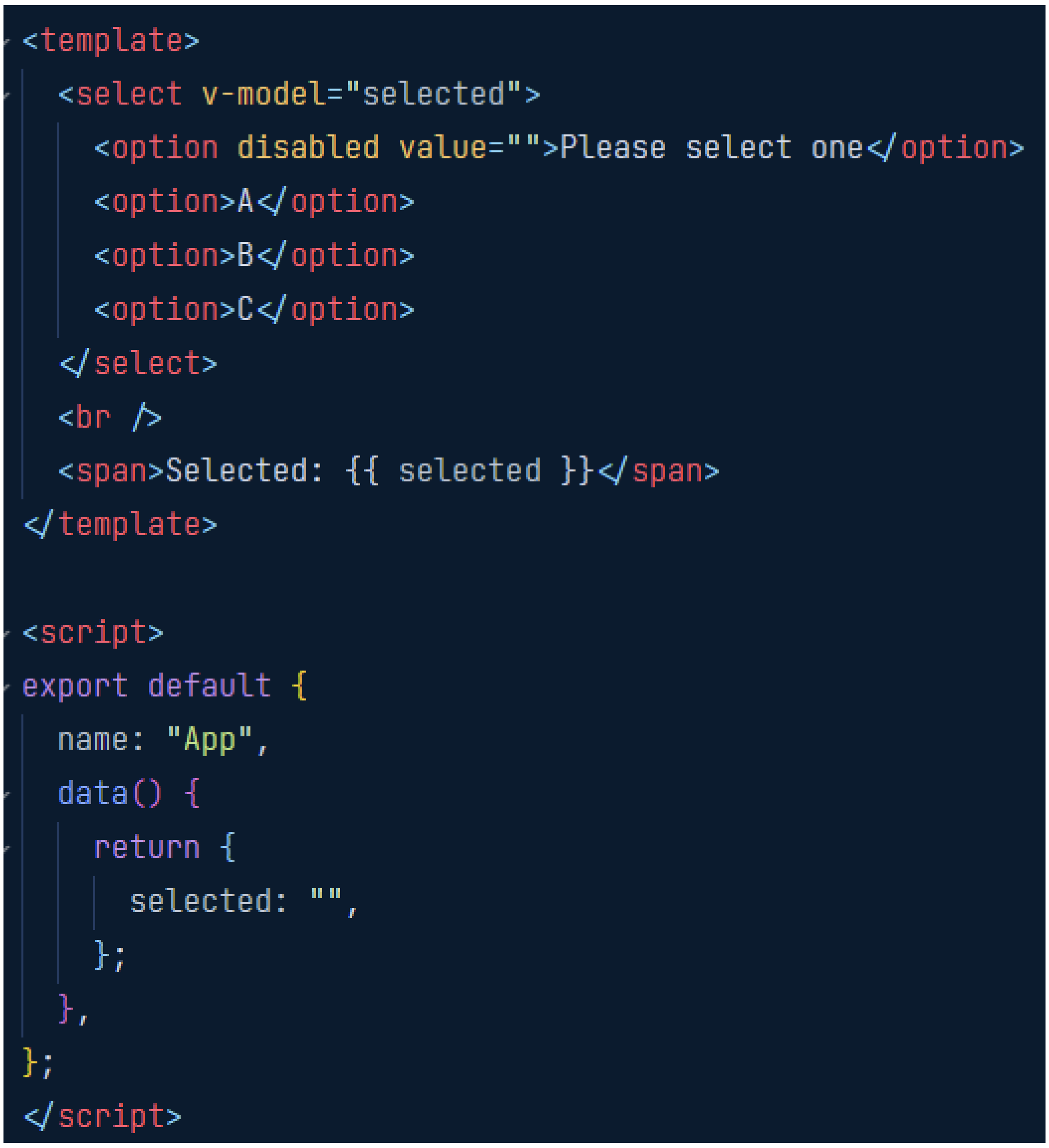


```
<template>
  <input type="radio" id="one" value="One" v-model="picked" />
  <label for="one">One</label>
  <br />
  <input type="radio" id="two" value="Two" v-model="picked" />
  <label for="two">Two</label>
  <br />
  <span>Picked: {{ picked }}</span>
</template>

<script>
export default {
  name: "App",
  data() {
    return {
      picked: "",
    };
  },
};
</script>
```

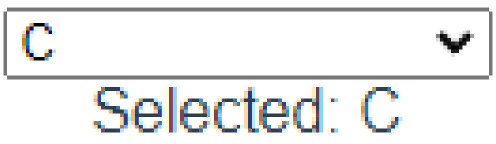


```
○ One
● Two
Picked: Two
```

Select

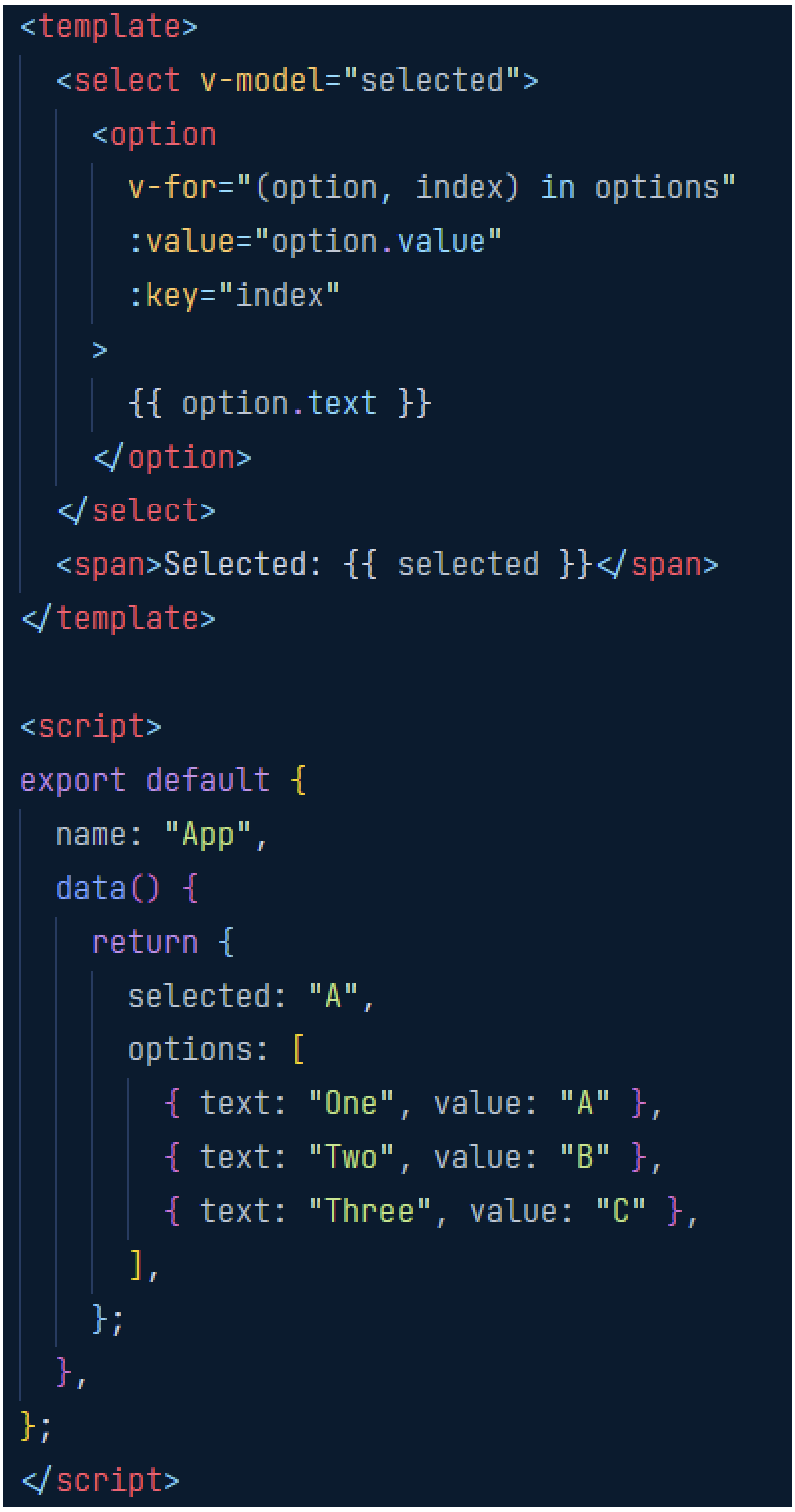


```
<template>
  <select v-model="selected">
    <option disabled value="">Please select one</option>
    <option>A</option>
    <option>B</option>
    <option>C</option>
  </select>
  <br />
  <span>Selected: {{ selected }}</span>
</template>

<script>
export default {
  name: "App",
  data() {
    return {
      selected: "",
    };
  },
};
</script>
```



```
C
Selected: C
```

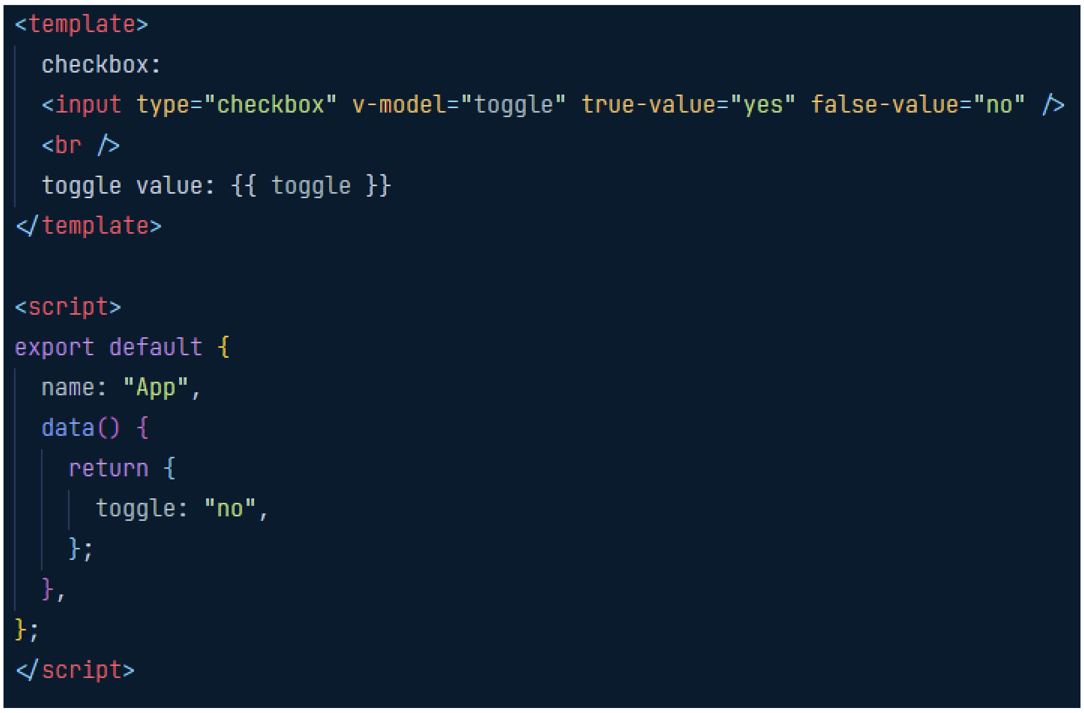
Of course, you can render select options dynamically:



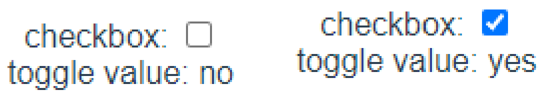
```
<template>
  <select v-model="selected">
    <option
      v-for="(option, index) in options"
      :value="option.value"
      :key="index"
    >
      {{ option.text }}
    </option>
  </select>
  <span>Selected: {{ selected }}</span>
</template>

<script>
export default {
  name: "App",
  data() {
    return {
      selected: "A",
      options: [
        { text: "One", value: "A" },
        { text: "Two", value: "B" },
        { text: "Three", value: "C" },
      ],
    };
  },
};
</script>
```

For radio, checkbox and select options, the v-model binding values are usually static strings (or booleans for checkbox), but sometimes we may want to bind the value to a dynamic property on the current active instance. We can use v-bind to achieve that. In addition, using v-bind allows us to bind the input value to non-string values.

Checkbox example:

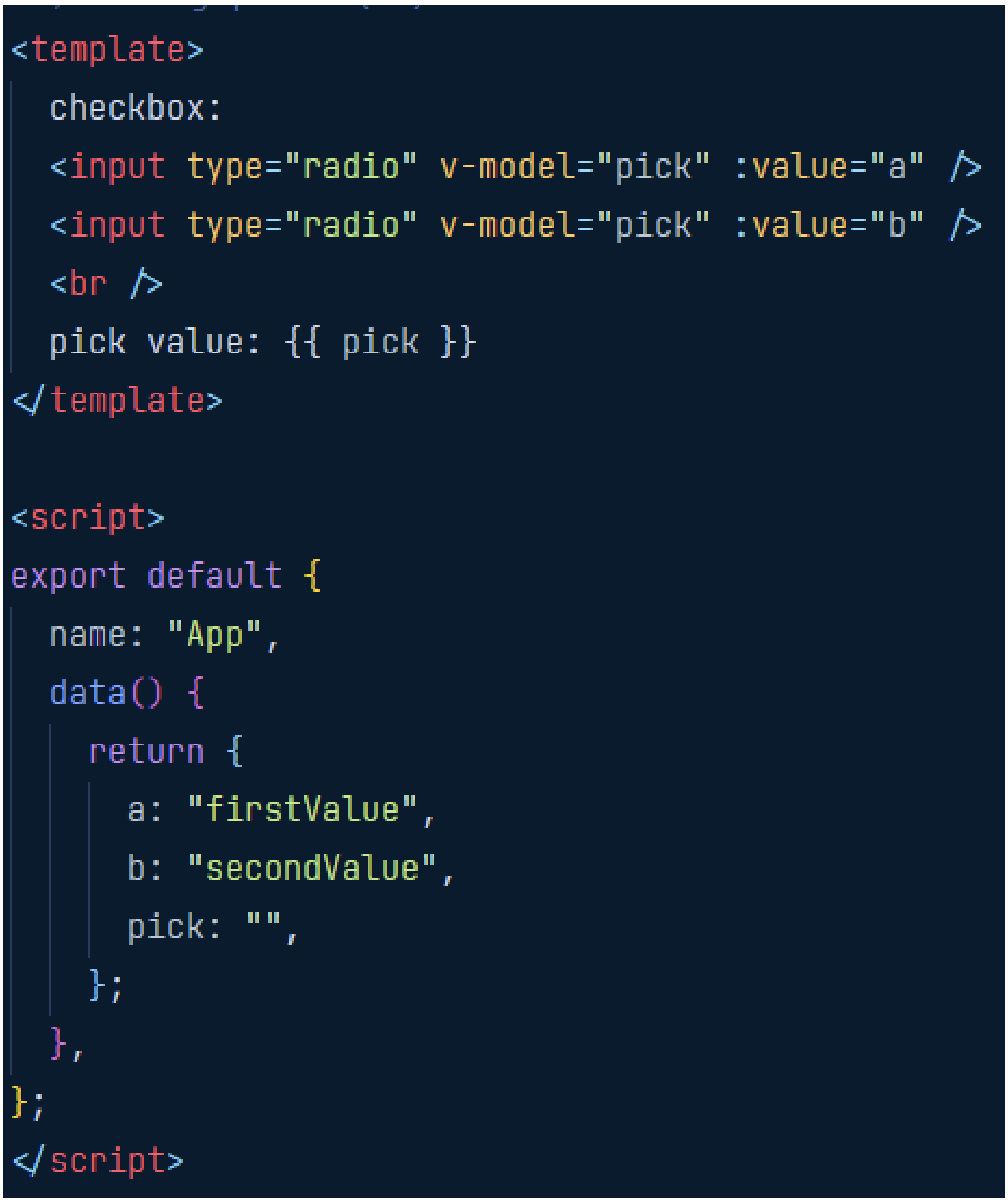


```
<template>
  checkbox:
  <input type="checkbox" v-model="toggle" true-value="yes" false-value="no" />
  <br />
  toggle value: {{ toggle }}
</template>

<script>
export default {
  name: "App",
  data() {
    return {
      toggle: "no",
    };
  },
};
</script>
```

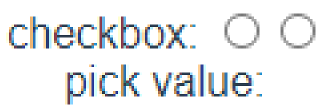


```
checkbox: ☐          checkbox: ☑
toggle value: no     toggle value: yes
```

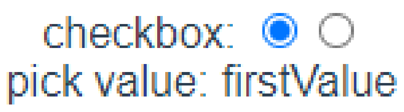
Radio example:



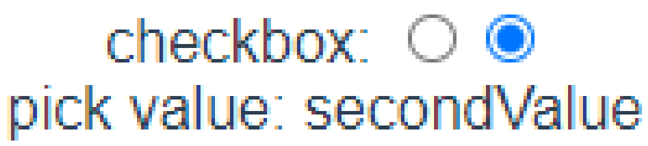
```
<template>
  checkbox:
  <input type="radio" v-model="pick" :value="a" />
  <input type="radio" v-model="pick" :value="b" />
  <br />
  pick value: {{ pick }}
</template>

<script>
export default {
  name: "App",
  data() {
    return {
      a: "firstValue",
      b: "secondValue",
      pick: "",
    };
  },
};
</script>
```



checkbox: ○ ○
pick value:

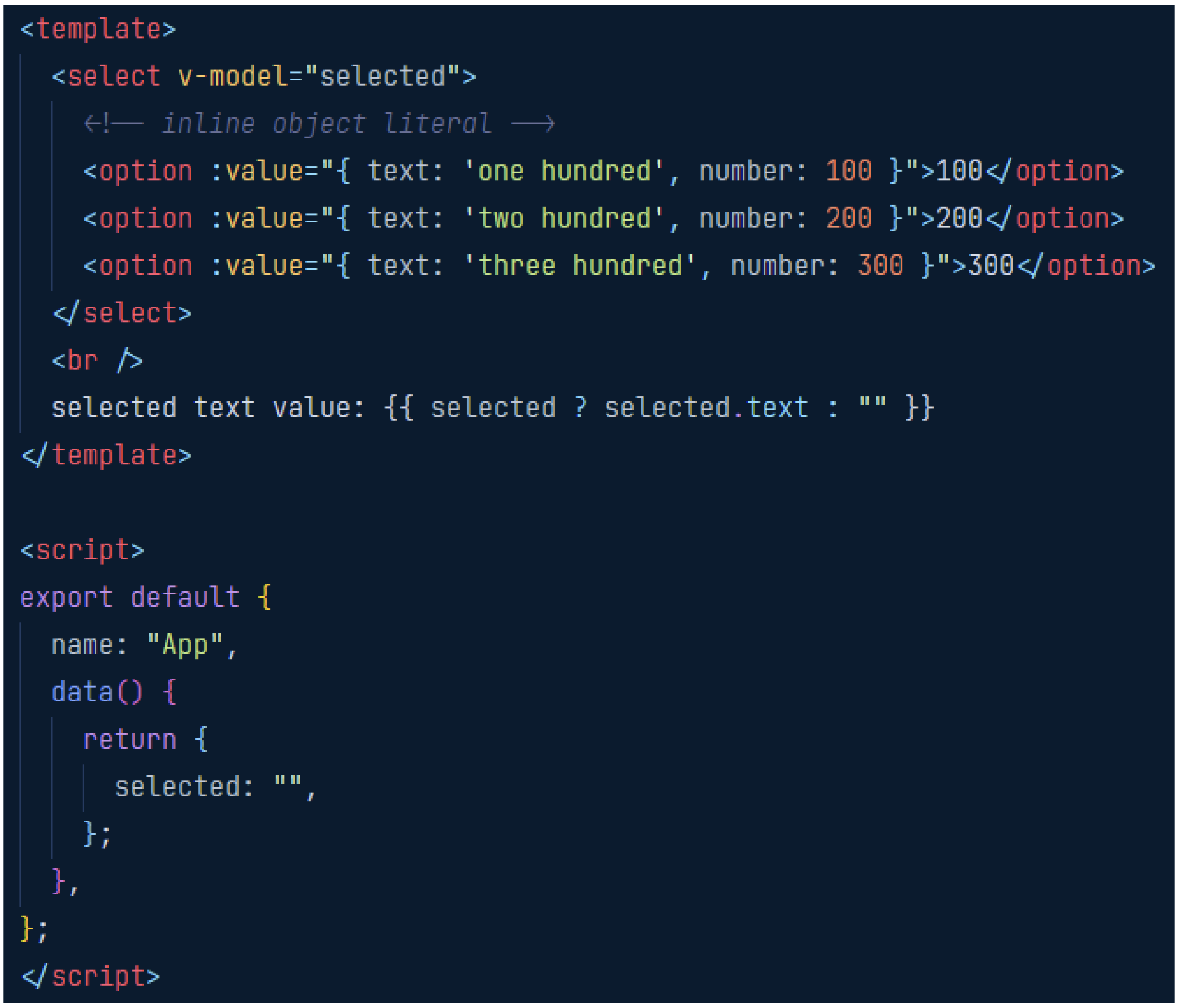


checkbox: ● ○
pick value: firstValue

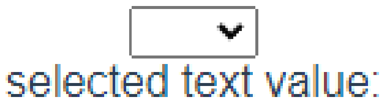


checkbox: ○ ●
pick value: secondValue

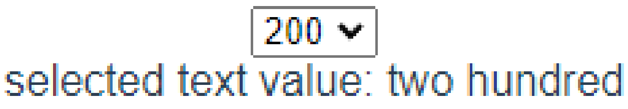
Select example:



```
<template>
  <select v-model="selected">
    <!-- inline object literal -->
    <option :value="{ text: 'one hundred', number: 100 }">100</option>
    <option :value="{ text: 'two hundred', number: 200 }">200</option>
    <option :value="{ text: 'three hundred', number: 300 }">300</option>
  </select>
  <br />
  selected text value: {{ selected ? selected.text : "" }}
</template>

<script>
export default {
  name: "App",
  data() {
    return {
      selected: "",
    };
  },
};
</script>
```



```
selected text value:
```

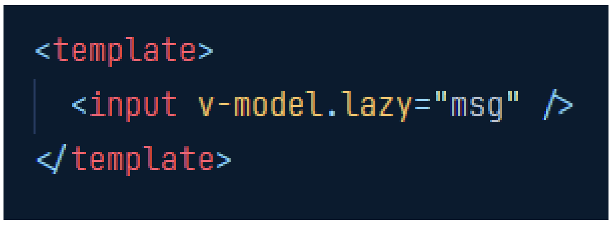


```
200
selected text value: two hundred
```

Modifiers

.lazy

By default, v-model syncs the input with the data after each input event. You can add the lazy modifier to instead sync after change events:



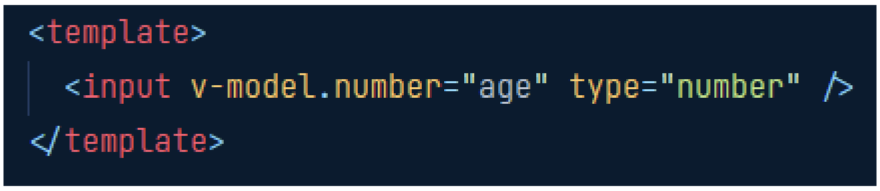
```
<template>
  <input v-model.lazy="msg" />
</template>
```

Depending on the kind of element being changed and the way the user interacts with the element, the change event fires at a different moment:

- When the element is :checked (by clicking or using the keyboard) for <input type="radio"> and <input type="checkbox">;
- When the user commits the change explicitly (e.g., by selecting a value from a <select>'s dropdown with a mouse click, by selecting a date from a date picker for <input type="date">, by selecting a file in the file picker for <input type="file">, etc.);
- When the element loses focus after its value was changed, but not committed (e.g., after editing the value of <textarea> or <input type="text">).

.number

If you want user input to be automatically typecast as a number, you can add the number modifier to your v-model managed inputs:

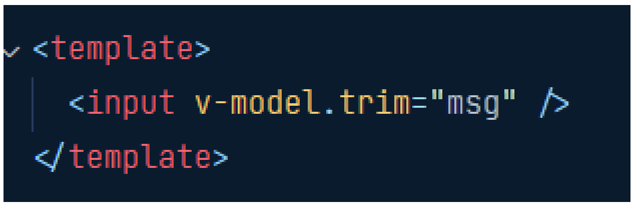


```
<template>
  <input v-model.number="age" type="number" />
</template>
```

This is often useful, because even with type="number", the value of HTML input elements always returns a string. If the value cannot be parsed with parseFloat(), then the original value is returned.

.trim

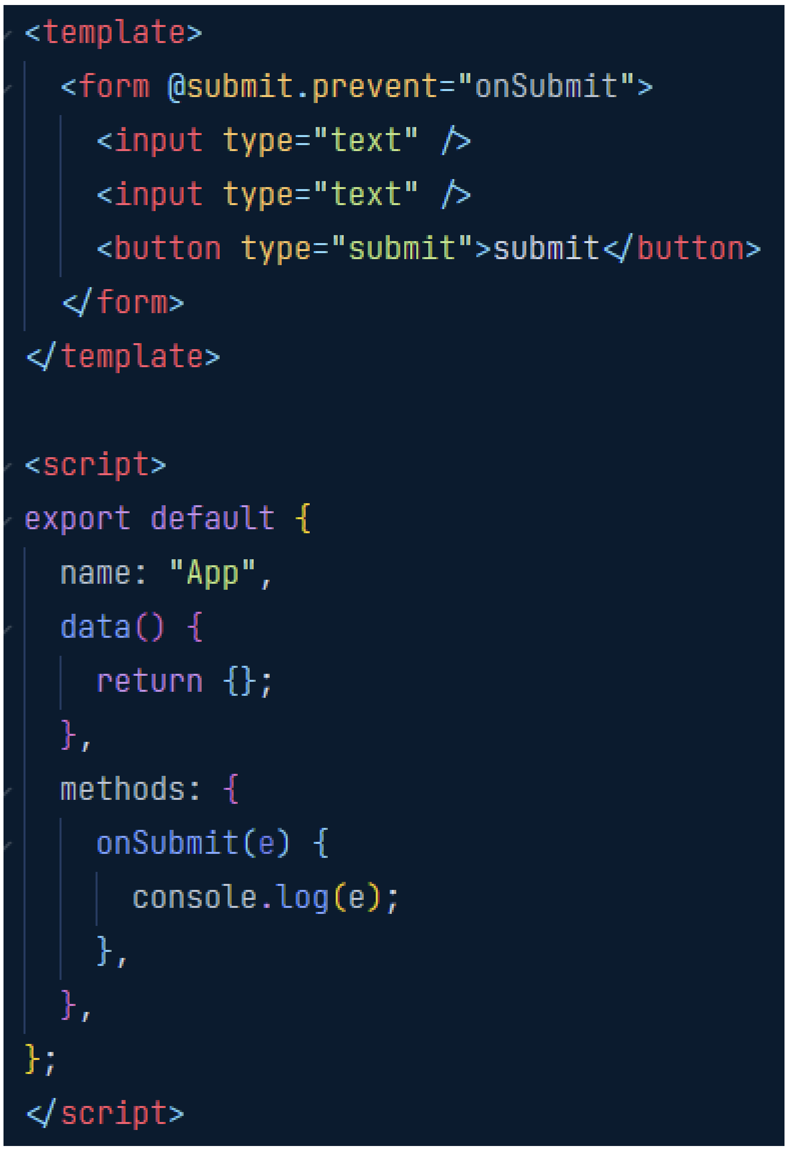
If you want whitespace from user input to be trimmed automatically, you can add the trim modifier to your v-model-managed inputs:



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

Form submission with event handler

To handle form submissions, we can make use of Vue’s submit event handler. We can also plug in the .prevent modifier to prevent the default action, which in this case would be the page refreshing when the form is submitted:



```
<template>
  <form @submit.prevent="onSubmit">
    <input type="text" />
    <input type="text" />
    <button type="submit">submit</button>
  </form>
</template>

<script>
export default {
  name: "App",
  data() {
    return {};
  },
  methods: {
    onSubmit(e) {
      console.log(e);
    },
  },
};
</script>
```

Validation

Form validation is natively supported by the browser, but sometimes different browsers will handle things in a manner which makes relying on it a bit tricky. Even when validation is supported perfectly, there may be times when custom validations are needed and a more manual, Vue-based solution may be more appropriate. Let’s begin with a simple example. Given a form of three fields, make two required. Let’s look at the code first:

```
<template>
  <form @submit.prevent="checkForm">
    <div v-if="errors.length">
      <b>Please correct the following error(s):</b>
      <ul>
        <li v-for="(error, index) in errors" :key="index">{{ error
}}</li>
      </ul>
    </div>

    <p>
      <label for="name">Name</label>
      <input id="name" v-model="name" type="text" name="name" />
    </p>

    <p>
      <label for="age">Age</label>
      <input id="age" v-model="age" type="number" name="age"
min="0" />
    </p>

    <p>
      <label for="movie">Favorite Movie</label>
      <select id="movie" v-model="movie" name="movie">
        <option>Star Wars</option>
        <option>Vanilla Sky</option>
        <option>Atomic Blonde</option>
      </select>
    </p>

    <p>
      <input type="submit" value="Submit" />
    </p>
  </form>
</template>
```

```
<script>
export default {
  data() {
    return {
      errors: [],
      name: "",
      age: "",
      movie: "",
    };
  },
  methods: {
    checkForm() {
      if (this.name && this.age) {
        return true;
      }

      this.errors = [];

      if (!this.name) {
        this.errors.push("Name required.");
      }
      if (!this.age) {
        this.errors.push("Age required.");
      }

      return false;
    },
  },
};
</script>
```

Let’s cover it from the top. We start with a form tag with a submit event handler preventing default behaviour. Beneath that there is a paragraph that shows or hides itself based on an error state. This will render a simple list of errors on top of the form.

The final thing to note is that each of the three fields has a corresponding v-model to connect them to values we will work with in the JavaScript. Now let’s look at that. We define an array to hold errors and set empty string values for the three form fields. The checkForm logic checks for name and age only as movie is optional. If they are empty we check each and set a specific error for each.

Exercise 1.1. Write a game character creator form. There should be 7 fields:

- character name (input type=”text”)
- character age (input type=”number”)
- character class (select with 4 options: ‘mage’, ‘warrior’, ‘rogue’, ‘druid’)
- character statistics: (each input type=”number”)
    - Strength
    - Intelligence
    - Wisdom
    - Charism

You have 10 points to distribute between all statistics. Default value for each one is 0. At the bottom of the form there is information about how many points are left to distribute. It should change dynamically after each statistic change. If a user used more points than 10, inform the user about it and disable the submit button. After form submission, validate all fields. All fields except for age are required. If something is not ok, eg. age field is not a number or any required attribute is empty, add new error to errors array and display them. After each form submission, clear the errors array. If the form is valid, hide the form and display info that a new character was created with all its properties (name, class and so on). If the age was not given, don’t display it. At the bottom there should be a button with ‘create new character’ text in it. After click, it should run a method displaying the form again with cleared values.

Section 2. Vue Router

The aim of the exercises in this section is to practice vue router.

One of the most powerful features of modern single-page web applications (SPA) is routing. Modern single-page apps such as a Vue application can transition from page to page on the client-side (without requesting the server). Vue Router is the official library for page navigation in Vue applications.

We will cover:

- Vue Router Fundamentals
- Dynamic routing
- How to pass router params
- Lazy loading

Why do we need a Vue Router in VueJS?

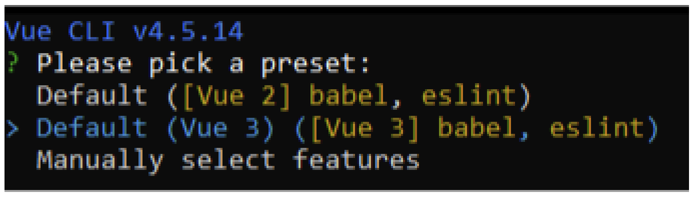
We need a router when you need to sync URLs to views in your app. It's a very common need, and all the major modern frameworks now allow you to manage routing. The Vue Router library is the way to go for Vue.js applications.

One of the biggest advantages of VueJS is the ability to build great Single Page Applications (SPAs). Single Page Applications are great because they don't require page loads every time the route changes. This means that once everything is loaded, we can switch the view really quickly and provide a great user experience. If you want to build a SPA in Vue, you're going to need Vue Router.

Routing is often categorized into two main buckets:

- Server-side routing: the client (i.e. the browser) makes a request to the server on every URL change.
- Client-side routing: the client only makes a request to the server upon initial page load. Any changes to the application UI based on URL routes are then handled by the client.

Let's create a new project and get right into coding. We'll create a new project with [vue-cli](vue-cli) using the default Vue 3 config by running command 'vue create name_of_app'

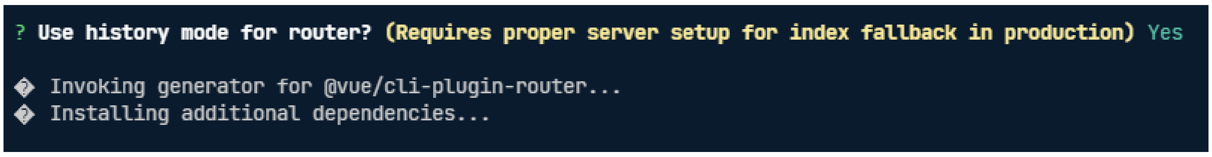


```
Vue CLI v4.5.14
? Please pick a preset:
  Default ([Vue 2] babel, eslint)
> Default (Vue 3) ([Vue 3] babel, eslint)
  Manually select features
```

Unfortunately, this config does not contain vue-router. To add it to our project, run the following command in the project directory.

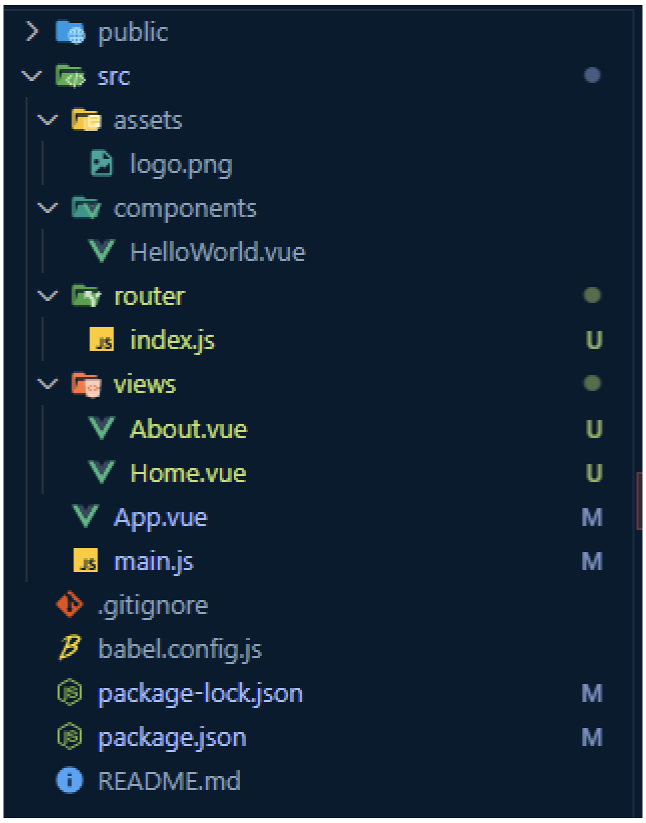
vue add router

The cli will ask if we want to use history mode. We'll be using history mode, so we'll select yes.



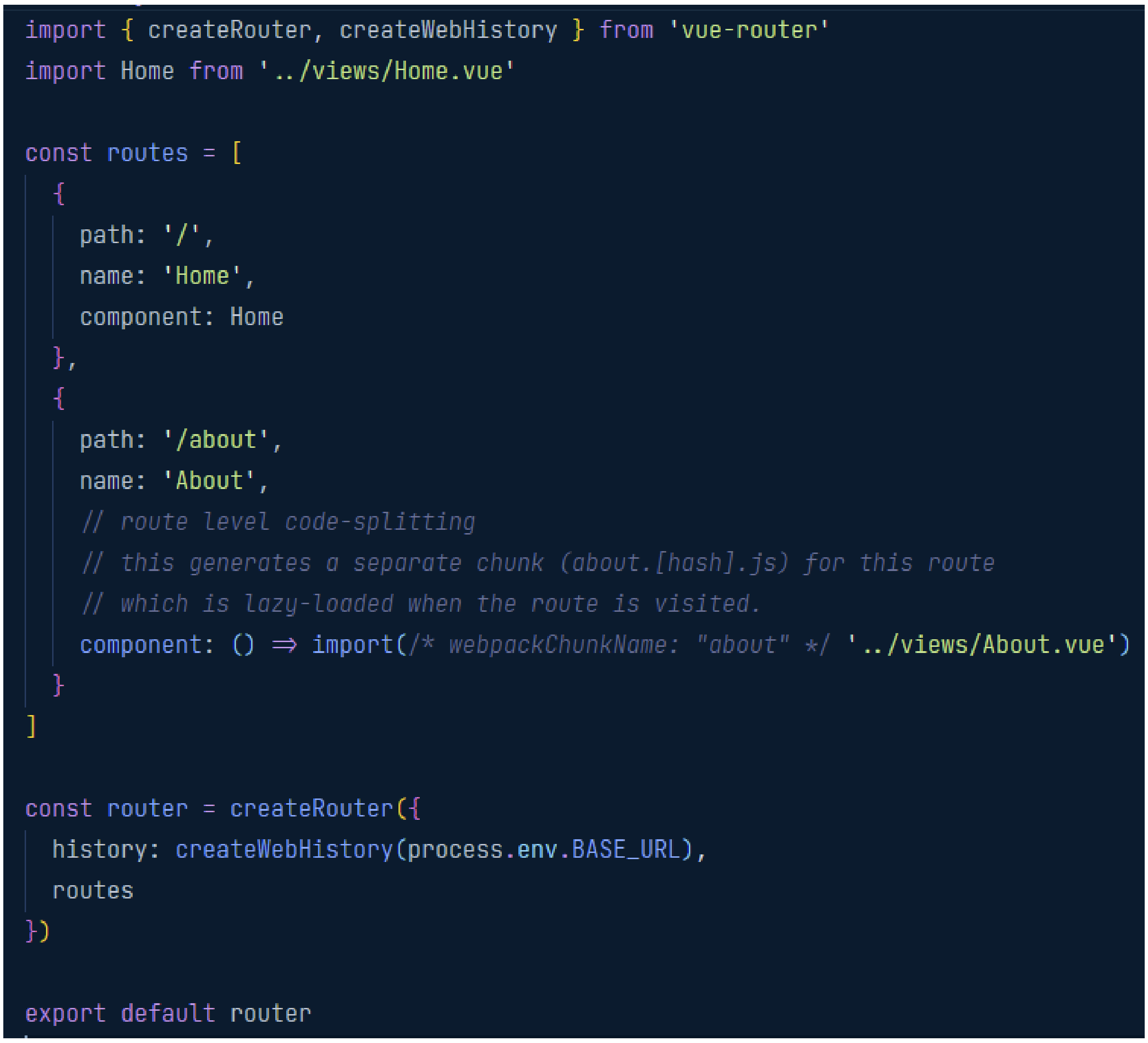
```
? Use history mode for router? (Requires proper server setup for index fallback in production) Yes

� Invoking generator for @vue/cli-plugin-router...
� Installing additional dependencies...
```

Let's look at the changes that command made for us.



```
> public
v src                    ●
  v assets
      logo.png
  v components
      HelloWorld.vue
  v router               ●
      index.js           U
  v views                ●
      About.vue          U
      Home.vue           U
    App.vue              M
    main.js              M
  .gitignore
  babel.config.js
  package-lock.json      M
  package.json           M
  README.md
```

A folder named views is created for us along with two files About.vue and Home.vue. These files represent our pages. A file named router/index.js file is also generated. This file contains all the router configurations. Let's open up the router/index.js file.

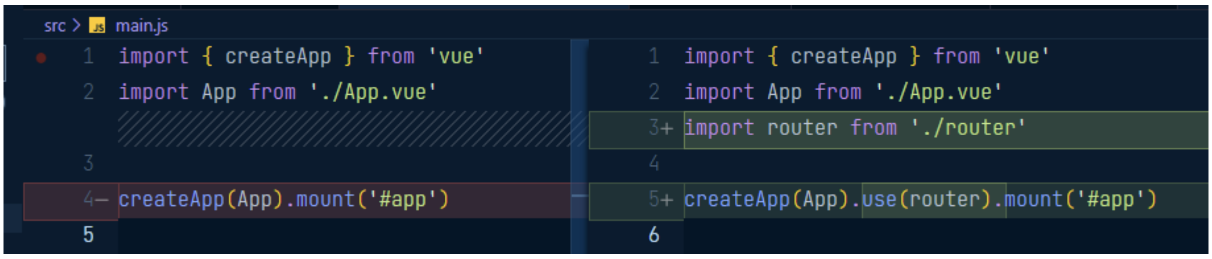


```javascript
import { createRouter, createWebHistory } from 'vue-router'
import Home from '../views/Home.vue'

const routes = [
  {
    path: '/',
    name: 'Home',
    component: Home
  },
  {
    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(/* webpackChunkName: "about" */ '../views/About.vue')
  }
]

const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes
})

export default router
```

We are importing createRouter and createWebHistory from the vue-router library. Next, we import the Home component from views/Home.vue. On line 4 we are declaring an array of objects named routes. This array represents our routes in the application. These array items are called **route objects**. The first route object has a path '/' which means this is going to be our base URL. The component property represents what component will be rendered when the user visits this path. We will render the Home page on this path. Finally, the name property represents the name of the route.

We have pretty much the same logic for the /about the path. However, instead of directly importing the component we are importing it through Webpack's code splitting feature. More on this later.

In index.js we import and use Vue Router:

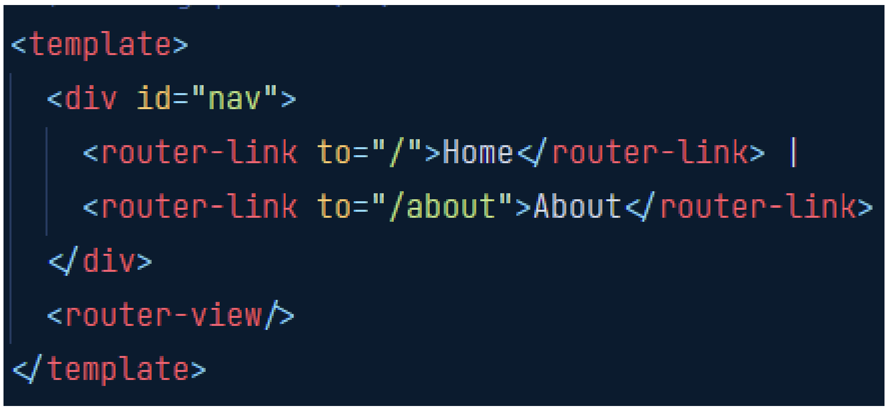


src > main.js

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

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

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

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

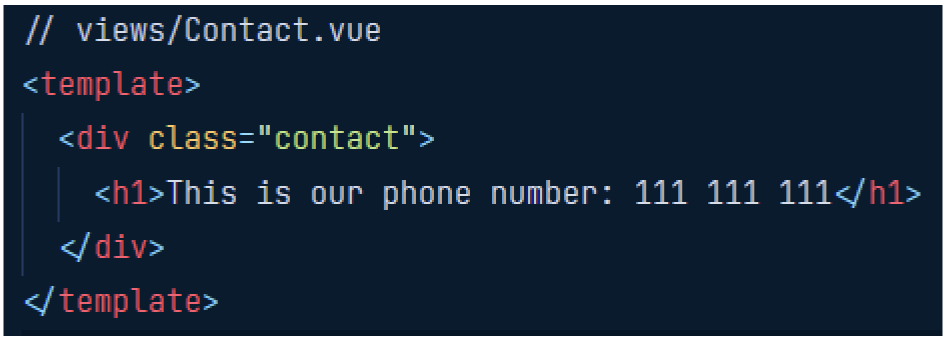
Now let's jump into App.vue file.



```
<template>
  <div id="nav">
    <router-link to="/">Home</router-link> |
    <router-link to="/about">About</router-link>
  </div>
  <router-view/>
</template>
```

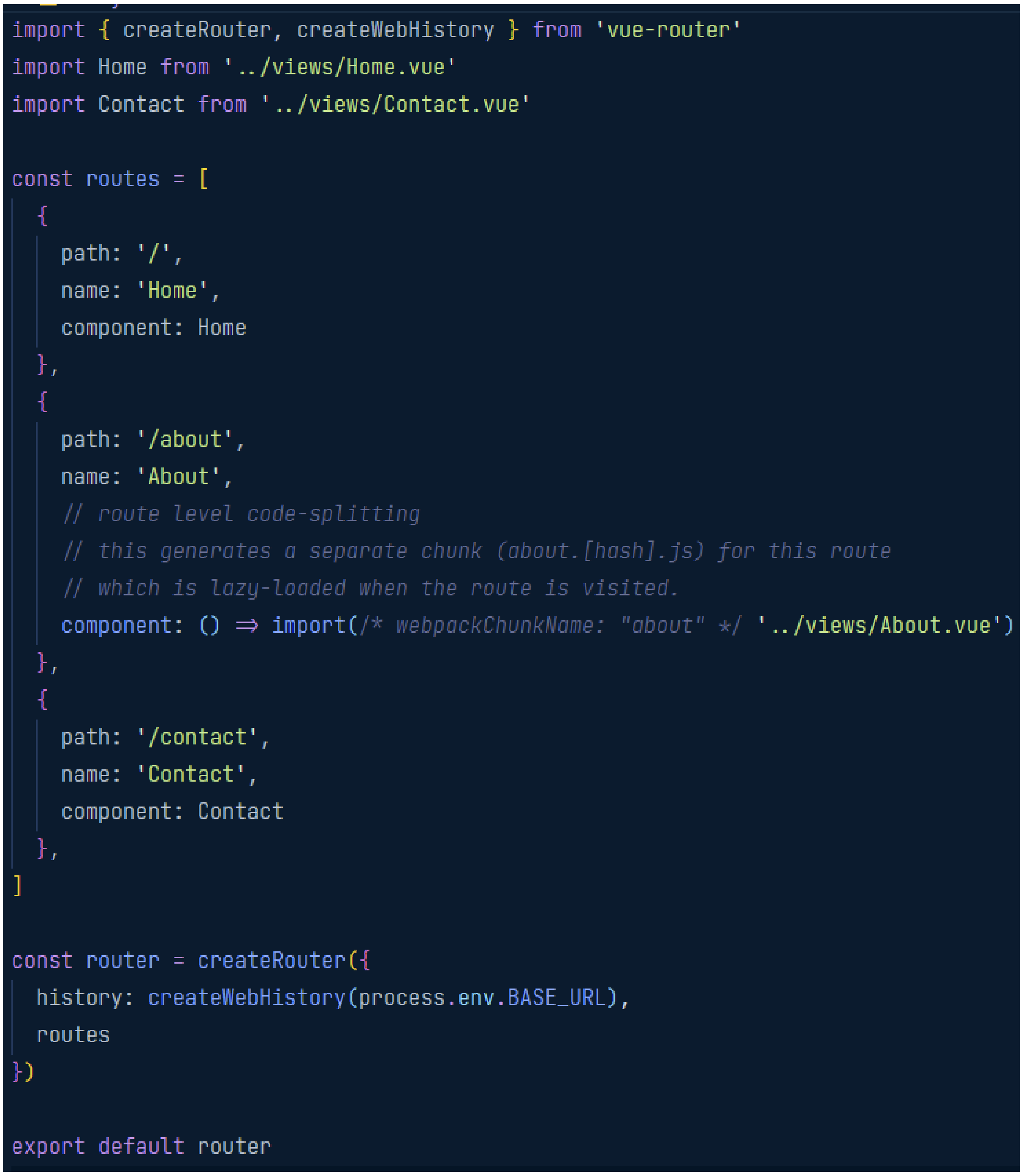
Observe the router-link tags. These tags are just fancy anchor links. However unlike an anchor link (<a href=""> tag) the <router-link> will not reload the whole page. Remember Vue is a single-page application. The data for the app is already downloaded from the server. When we route to another view the application just hides some information and displays the requested information. Router-link tags have a 'to' attribute which refers to which page to visit. The <router-view/> tag is what renders the right component when navigation links are triggered.

To add new view, we need to create a new component in the views directory, let's name it 'Contact.vue'.



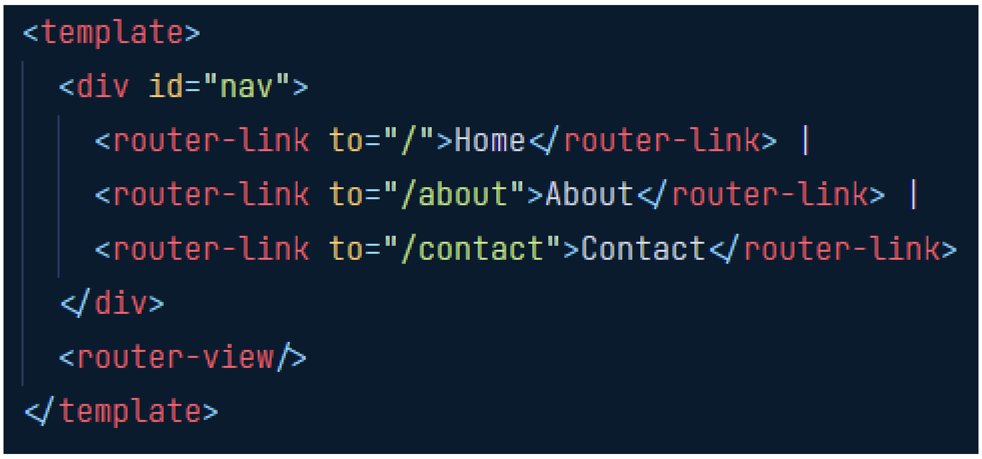
```
// views/Contact.vue
<template>
  <div class="contact">
    <h1>This is our phone number: 111 111 111</h1>
  </div>
</template>
```

Now, in our routes array we have to import our new view and specify the route path.



```javascript
import { createRouter, createWebHistory } from 'vue-router'
import Home from '../views/Home.vue'
import Contact from '../views/Contact.vue'

const routes = [
  {
    path: '/',
    name: 'Home',
    component: Home
  },
  {
    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(/* webpackChunkName: "about" */ '../views/About.vue')
  },
  {
    path: '/contact',
    name: 'Contact',
    component: Contact
  },
]

const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes
})

export default router
```

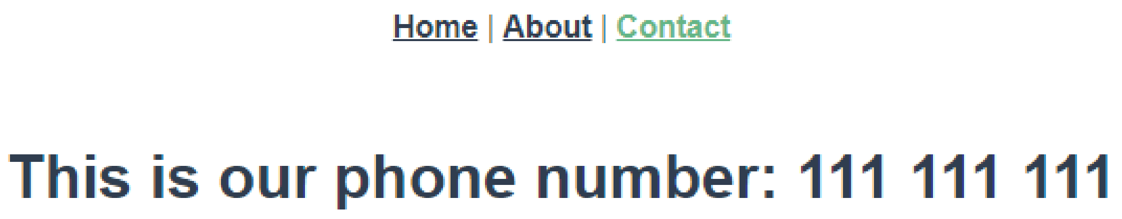
Finally, we will add new router link:



```
<template>
  <div id="nav">
    <router-link to="/">Home</router-link> |
    <router-link to="/about">About</router-link> |
    <router-link to="/contact">Contact</router-link>
  </div>
  <router-view/>
</template>
```

That’s it, we have our own route added to the application.

If you click on ‘Contact’ link you will get:



Home | About | Contact

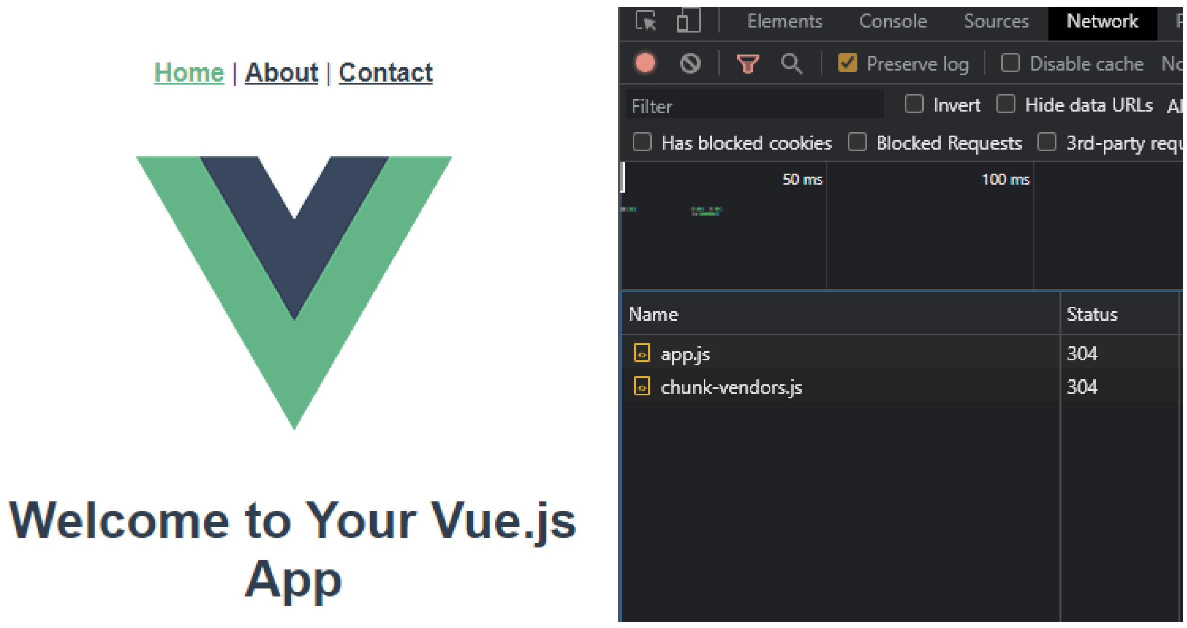
This is our phone number: 111 111 111

and the url will be http://localhost:8080/contact

Exercise 2.1. Recreate all steps and add your own separate page.

Lazy loading

Let’s talk about lazy loading. This is one of the advanced features that Vue Router provides out of the box. Let’s open up our devtools and observe the js tab.

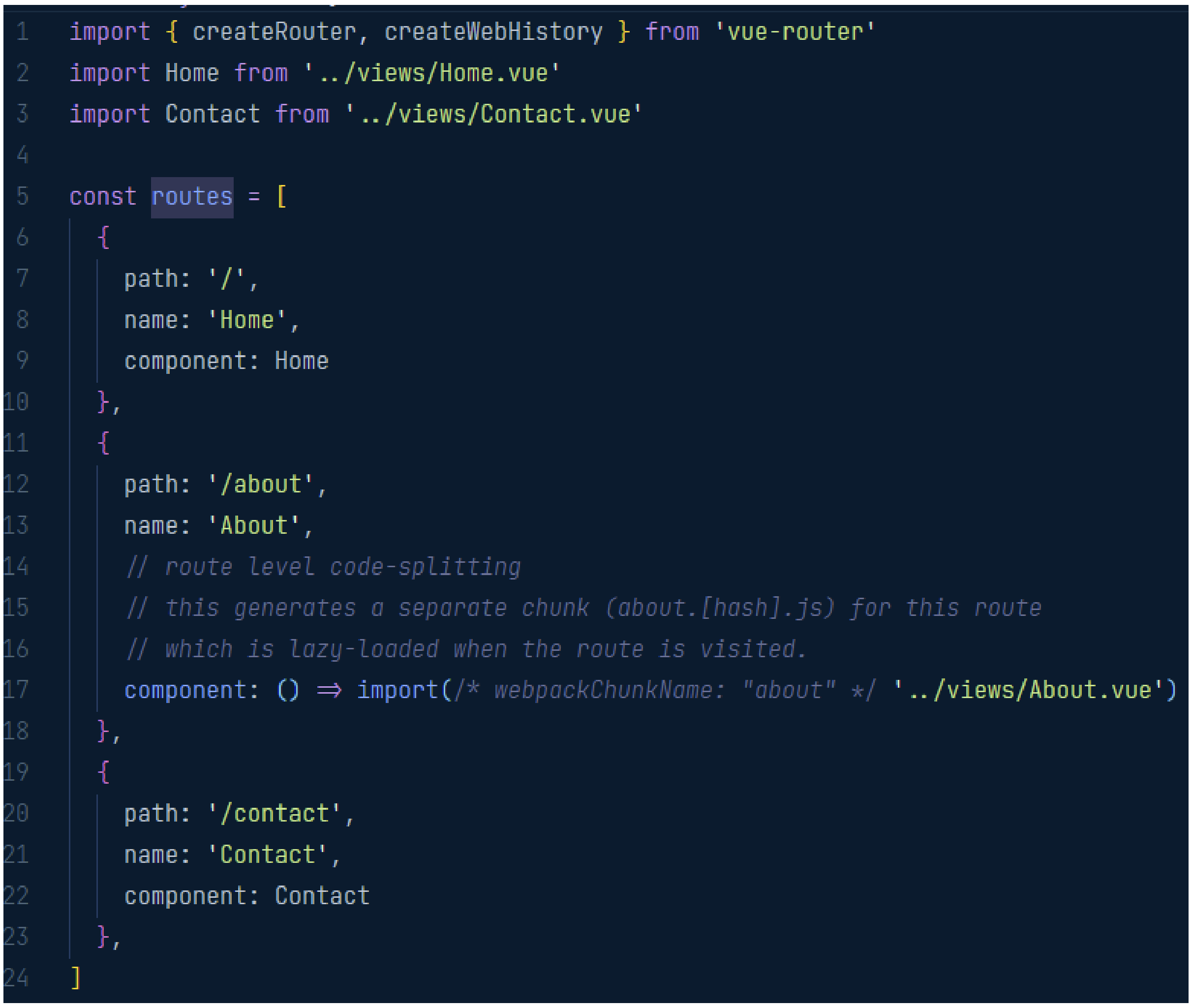


[✓ Preserve log] Disable cache

Name	Status
app.js	304
chunk-vendors.js	304

You can see that when we load the application for the first time it loads an app.js script. When you navigate to ‘/’ route or /contact route you can see no new javascript is being loaded. This is because all the JavaScript code for those components was already loaded with app.js. In a smaller application, this is not a problem. However, what if we have a large application. In that case, loading all the JS code in one go can make the page load time significantly longer. Lazy loading is used to avoid this.

When we lazy load, we load only the parts we need, when we need them.



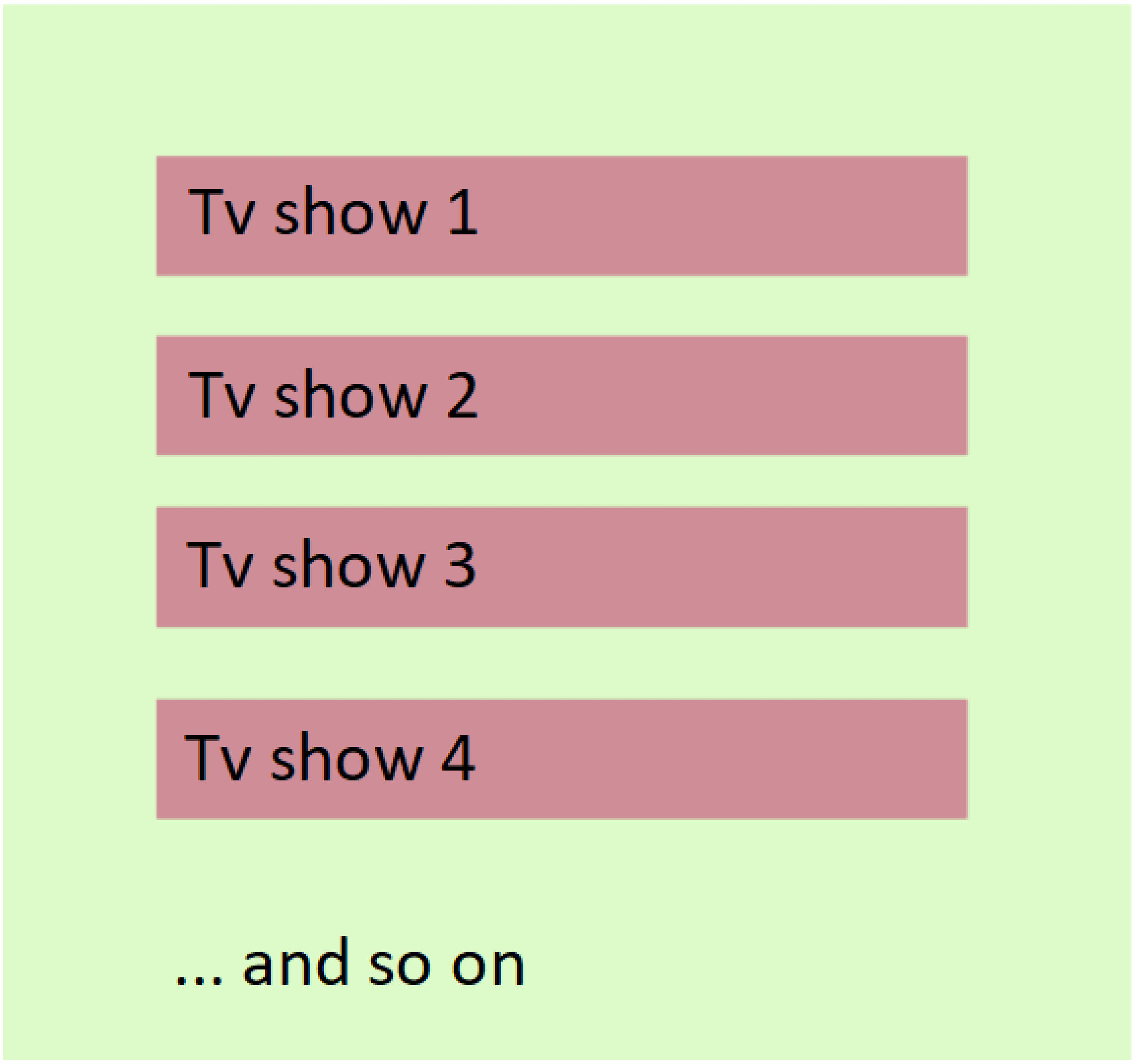
```
 1  import { createRouter, createWebHistory } from 'vue-router'
 2  import Home from '../views/Home.vue'
 3  import Contact from '../views/Contact.vue'
 4
 5  const routes = [
 6    {
 7      path: '/',
 8      name: 'Home',
 9      component: Home
10    },
11    {
12      path: '/about',
13      name: 'About',
14      // route level code-splitting
15      // this generates a separate chunk (about.[hash].js) for this route
16      // which is lazy-loaded when the route is visited.
17      component: () => import(/* webpackChunkName: "about" */ '../views/About.vue')
18    },
19    {
20      path: '/contact',
21      name: 'Contact',
22      component: Contact
23    },
24  ]
```

Take a look at line 17 of router/index.js. Do you recall this from an earlier example? The About component here is being imported with an arrow function. This is an example of lazy loading. Unlike Home and Contact component, the About component is not loaded with the app.js. The About component will only get loaded when a user visits the /about route. This is all we have to do to implement lazy load. This lazy loading is being done through Webpack's code splitting feature. Under the hood Vue uses Webpack as a bundler.

Exercise 2.2. Make all routes lazy-loaded. Check if it works in the browser.

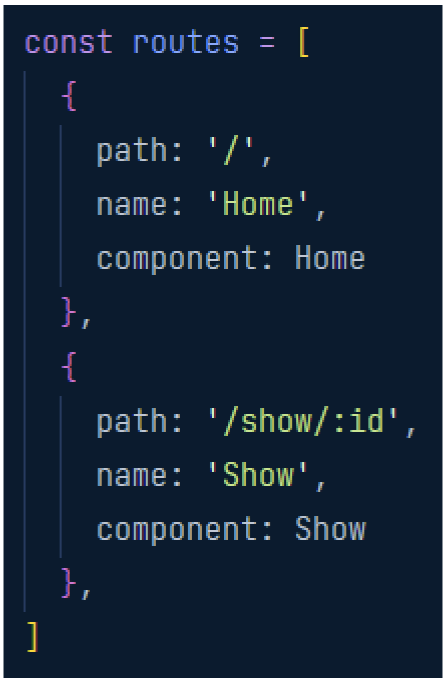
How to use router params with Vue Router

First of all, let’s take a look at what router parameters (router params in short) are and why they are important. Let’s say we have a page in our application where we show users a list of tv shows.



Let’s say in our application this path is /shows. When a user clicks on any of these shows the user is taken to that tv show page. For each show the path will be different (/show/1, /show/2, etc.) based on their id. How do we achieve this? We can add a new route for every show we have, but then we would have to change our code every time a new tv show is published. Therefore, we will use a dynamic route. This is where router params comes in handy.

In our application, we can specify a generic route like the /show and then we can specify optional route params to the same route like this /show/:id. The :id picks up any dynamic string as a parameter. This means we can now route to paths like /show/1 , etc. and the Vue router in the application will know which path we are trying to go to.



```
const routes = [
  {
    path: '/',
    name: 'Home',
    component: Home
  },
  {
    path: '/show/:id',
    name: 'Show',
    component: Show
  },
]
```

In our router-link tag we can use the :to property to pass in an object with view name and params data. In the code snippet below we are passing in an object with an id parameter.

```
<router-link :to="{ name: 'Show', params: { id: 1 } }">
  Go to show view
</router-link>
```

How to access route params from the view

When we add a Vue router to our Vue application we get access to a special $route object inside the global scope of our application. This object holds all the router information of our application. We can access the $route object by simply calling this.$route. We can access the params object through the $route object. Take a look at the code snippet below.

```
<template>
  <span>
    {{ this.$route.params.id }}
  </span>
</template>
```

The second method of achieving that is shown below. We have to be sure that the name of the route parameter declared in route path is the same as the prop name in your view. Additionally, you have to add

props: true,

to your route object (in index.js) for this to work.

```
<template>
  <div>{{ id }}</div>
</template>
<script>
export default {
  props: {
    id: String,
  },
};
</script>
```

Exercise 2.3. Create app listing many different Tv Shows containing two views:

- **all shows view**
- **single show view**

In all shows view each Tv Show should have its name and image displayed. After click on the show name we should be redirected to single show view, where we should see:

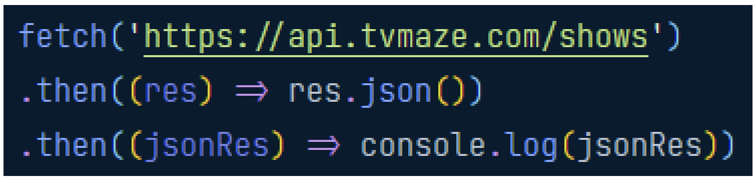
- **show name**
- **show image**
- **language**
- **genres**
- **averageRuntime**
- **router link to all shows view**

In both views use a lifecycle hook to fetch shows / show info.

API endpoints:

List of tv shows:

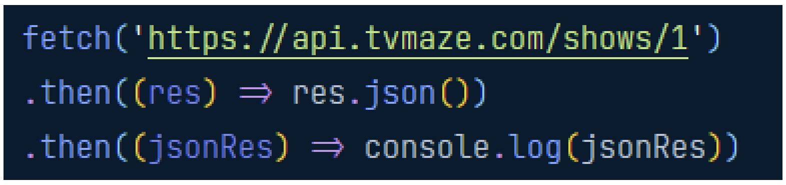
https://api.tvmaze.com/shows



```
fetch('https://api.tvmaze.com/shows')
.then((res) => res.json())
.then((jsonRes) => console.log(jsonRes))
```

Single show:

https://api.tvmaze.com/shows/1



```
fetch('https://api.tvmaze.com/shows/1')
.then((res) => res.json())
.then((jsonRes) => console.log(jsonRes))
```

Programmatic Navigation

In our applications, there will be scenarios when we would want to programmatically navigate to another page. Inside our application, we have access to the router instance as $router. We can access it by calling this.$router. The router instance has a function called push. This function can be used to navigate to different routes. Here are some examples of how it works:

```
// route by route path
this.$router.push("home");


// object
this.$router.push({ path: "home" });


// named route navigation with parameters
this.$router.push({ name: "user", params: { userId: "123" } });


// with query params, resulting in /profile?info=some-info
this.$router.push({ path: "profile", query: { info: "some-info" }
});
```

404 page

After you have all of your other routes defined, you can create a default route. Here, the default route will act as a 404 - Not Found page - a fallback in the case no route is found.

```
import { createWebHistory, createRouter } from "vue-router"
import Home from "@/views/Home.vue"
import PageNotFound from '@/views/PageNotFound.vue'

const routes = [
  {
    path: "/",
    name: "Home",
    component: Home,
  },
  {
    path: '/:catchAll(.*)*',
    name: "PageNotFound",
    component: PageNotFound,
  },
]
```

Vue Router for Vue 3 uses a custom RegEx. The value of path contains RegEx, which is telling Vue to render PageNotFound.vue for every route, unless the route is already defined. The catchAll in this route refers to a dynamic segment within Vue Router, and (.*) is a regular expression that captures any string.

Exercise 2.4. Add a 404 page to your tv shows app. Check if it's working.

Homework

Create app displaying products using fakestoreapi containing two views:

- all products view
- single product view

In both views use a lifecycle hook to fetch shows / show info.

In all products view each product should have its name, image and price displayed. After click on the product image we should be redirected to single product view, where we should see:

- product title
- product image
- product description
- category
- price
- rating (rate & count)
- router link to go back to all products view
- review form and already added reviews

Review form should contain:

- name input (required)
- review textarea (required, review should be longer than 50 characters but less than 500 characters)
- rating (required, select with five values from 1 to 5)
- purchase date (input type=”date”) (optional)

After form submission, validate all fields. All fields except for purchase date are required. If something is not ok, eg. review is too short or any required attribute is empty, add new error to errors array and display them. After each form submission, clear the errors array. If the form is valid, add review to reviews array (just locally, don’t send any request) and clear form fields. We should be able to add as many reviews as we want. Of course, after page reload or going to products list and coming back to the product view our reviews will be gone, it’s ok. We will learn how to share state between views in our next lesson.

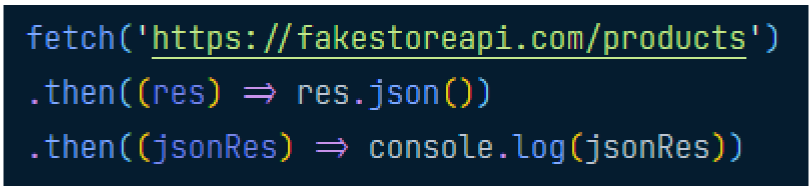
Upload a .zip with your homework without node_modules.

Warning: Sometimes this API can be very slow. Display a ‘loading’ message until you receive a response.

API endpoints:

List of products:

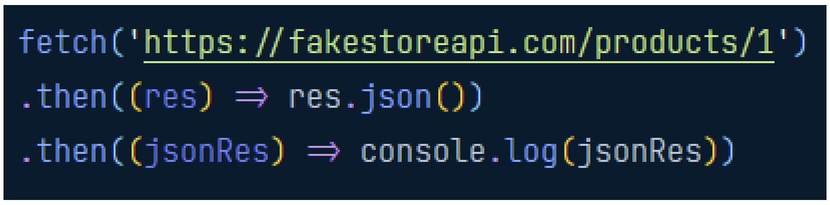
https://fakestoreapi.com/products



```javascript
fetch('https://fakestoreapi.com/products')
.then((res) => res.json())
.then((jsonRes) => console.log(jsonRes))
```

Single product:

https://fakestoreapi.com/products/1



```javascript
fetch('https://fakestoreapi.com/products/1')
.then((res) => res.json())
.then((jsonRes) => console.log(jsonRes))
```