Lab 6: Structuring applications in VueJS

Over the past two weeks, we have looked at using Vue to create applications. This week's lab will look at how to structure Vue applications to manage scalability and readability.

In this lab, we will look at re-creating the chat application from lab 5. Only this week we will write the application in a more structured and accepted fashion, making it scalable and ready for the future.

1 Exercise 1: Initial setup (same as lab 5)

We are going to build a front end application for the chat app API that we wrote in lab 3. Start by running the API that you created for lab 3. If you don't have access to your code, ask a friend for a copy, or the lab tutor for their solution.

Note: If your API is hosted at a different location to your client application (i.e. a resource has to make a cross-origin HTTP request to a different domain, protocol, or port) then you will need to add CORS support. This allows your API to accept requests from different domains. One method to add CORS support is to add the below code into your express config file. More information about CORS can be found at: https://enable-cors.org/index.html.

```
app.use(function(req, res, next) {
        res.header("Access-Control-Allow-Origin", "*");
        res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With,
Content-Type, Accept");
       res.header("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE");
       next();
});
```

2 Structuring large applications with Vue

In this lab, we will rewrite the chat application built in last weeks lab. The difference being that this week, we will look at structuring our application in a way that is scalable and usable.

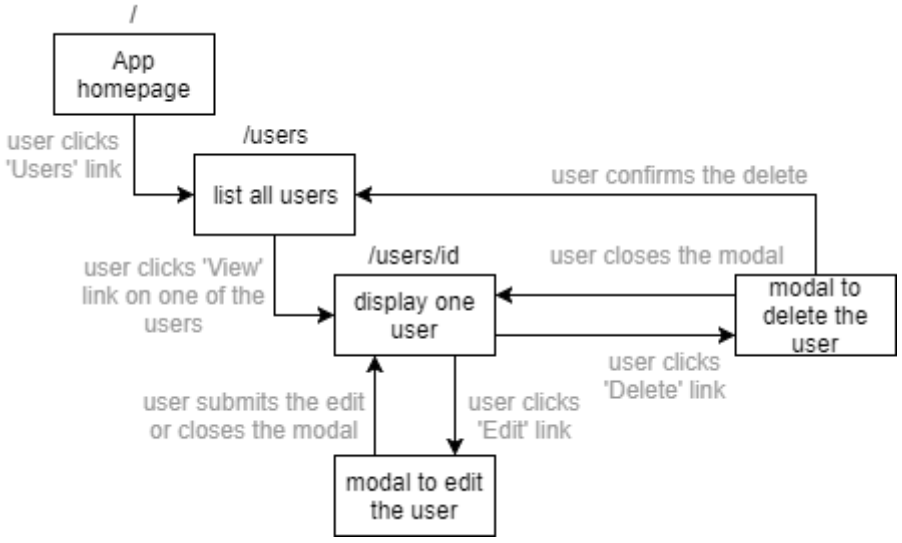
2.1 Single Page Applications

Taken from: https://msdn.microsoft.com/en-gb/magazine/dn463786.aspx
Single-Page Applications (SPAs) are Web apps that load a single HTML page and dynamically update that page as the user interacts with the app.

SPAs use AJAX and HTML5 to create fluid and responsive Web apps, without constant page reloads. However, this means much of the work happens on the client side, in JavaScript.

3 Exercise 2: Planning

This exercise is going to run through how to implement the ‘User’ functionality. The first thing you want to do before developing an application is to plan how it will look and how the user will interact with the functionality (and the routes that will hold each page). For the User functionality, let’s look at how the user will interact with the application:



Here, we start with at the applications homepage. The user clicks the ‘Users’ link to go navigate to /users. The /users page includes a list of all users, each with a link to view that individual user (located at /users/id). When viewing an individual user, there is the option to ‘Edit’ or ‘Delete’ that user. Editing the user opens up a modal with a form to edit the user. If the delete button is clicked, another modal is displayed asking the user to confirm that they want to delete. If the user closes the delete modal, they are taken back to /users/id. If they confirm the delete, then the user is deleted and the user is sent to /users.

Don’t know what a modal is? **Read more**:
https://v4-alpha.getbootstrap.com/components/modal/

1. Plan out the rest of the application, this can just be a rough sketch for now and may change when you come to implement. However, sketching the applications structure out before coding will allow you to get an idea of the ‘bigger picture’ in your head before starting the implementation.

4 Exercise 3: Vue-cli

Vue-cli is a tool that helps you to scaffold out a Vue project easily. Vue-cli will set up a project structure and a workflow using Webpack (https://webpack.js.org/concepts/). Among other useful features, Webpack allows us to run our code on a development server using one command.

1. Install vue-cli using npm as a global package (npm install -g vue-cli).

Note: This may have to be done as an Administrator ('sudo' on Mac and Linux). In the labs, you can install vue-cli without the -g parameter inside the lab 6 directory (npm install vue-cli). Then when running 'vue init' (step three), instead run 'node node_modules/vue-cli/bin/vue init webpack-simple chat-app'.

2. Create a directory called 'Lab 6.' Navigate to this directory in the command line
3. Run 'vue init webpack-simple chat-app' (Webpack-simple defines the vue-cli template that we want to use).
4. Vue-cli will ask you a series of questions:
    a. Project name: hit enter
    b. Project description: give a description
    c. Author: give your name
    d. Use Sass? N
5. In your Lab 6 directory, a new 'chat-app' directory will have been created. This contains our project. Have a look around and get familiar with the new structure. Don't worry about anything that doesn't make sense for now, the main things to notice are:
    a. that inside 'chat-app' we have our index.html file
    b. the rest of our files are inside the src directory
    c. App.js is now called main.js
    d. There's a .vue file. These are Vue components, we will get onto these in a little while
6. Navigate into the chat-app directory and run 'npm install.' This will install the dependency packages defined in package.json.
7. Now run 'npm run dev' to run the development server. A new tab will open in your browser with the below page:

Essential Links

Core Docs Forum Gitter Chat Twitter

Ecosystem

vue-router vuex vue-loader awesome-vue

5 Exercise 4: Routing with vue-router

Now that we have our project set up, we need to set up a client-side router so that we can create our desired structure. A client side router works similarly to the server-side express router we have used in earlier labs (and assignment 1). However, client-side routers work using fragment identifiers. In depth knowledge of how they work isn't required for you to

implement them, but here's a good video for if you'd like to learn more: https://www.youtube.com/watch?v=qvHecQOiu8g

To implement routing in Vue, we will use the official vue-router (http://router.vuejs.org/en/essentials/getting-started.html). This exercise will show you how to implement the routing for the chat application.

1. In the command line, navigate to your chat-app directory, created in exercise 3.
2. Run 'npm install --save vue-router.' **Note**: the --save will add the module to our package.json
3. In 'main.js,' import the router and add 'Vue.use(VueRouter)' to link the router to our Vue object.

```
import Vue from 'vue';
import App from './App.vue';

import VueRouter from 'vue-router';
Vue.use(VueRouter);

new Vue({
  el: '#app',
  render: h => h(App)
});
```

4. Now we have our router imported, we need to declare some routes. After our Vue.use() line, lets create a constant called 'routes' that will store a list of the routes that our application will use.

```
const routes = [];
```

5. Inside our list, add a JSON object for each route. To implement our users functionality, we initially need two routes; Home and Users. **Note**: Each route needs a path and a component. The path is the route that we want to define (e.g. /users). The component is the .vue file that we want to include (we will create these next).

```
const routes = [
        {
                path: "/",
                component: Home
        },
        {
                path: "/users",
                component: Users
        }
];
```

6. Now at the top of ‘main.js,’ import the components.

```
import Vue from 'vue';
import App from './App.vue';
import Home from './Home.vue';
import Users from './Users.vue';

import VueRouter from 'vue-router';
Vue.use(VueRouter);

const routes = [
        {
                path: "/",
                component: Home
        },
        {
                path: "/users",
                component: Users
        }
];

new Vue({
  el: '#app',
  render: h => h(App)
});
```

7. We also need to create the Home.vue and Users.vue files to the ‘src’ directory.
8. To get these newly created routes into our application, we add the following code under our routes const:

```
const router = new VueRouter({
        routes: routes,
        mode: 'history'
});
```

9. Next, add the router to our root Vue instance:

```
new Vue({
  el: '#app',
  router: router,
  render: h => h(App)
});
```

10. Now in App.vue, add the <router-view> element. This is where the routed templates will be displayed on the page. Your ‘App.vue’ page should look like below (you can get rid of everything else for now):

```
<template>
        <div id="app">
                <router-view></router-view>
        </div>
</template>
```

11. Inside your Home.vue and Users.vue, add the templates and some simple text for testing.

```
<template>
	<div>
		Home Page
	</div>
</template>
```

12. Now run your development server (npm run dev). Test that your two routes work in your browser.

6 Exercise 5: Adding the functionality

6.1 Exercise 5.1: Listing all users

1. In your command line install vue-resource, saving it also to your package.json with --save
2. Add the following to your main.js file:

```
import VueResource from 'vue-resource';
Vue.use(VueResource);

Vue.http.options.emulateJSON = true;
```

3. In your Users.vue, underneath your template element, add the following script element. This is the same code as from Lab 5, except that here, the data object is a function that returns the data as an object. Also, the code is encased in a ‘default’ object that we export.

```
<script>
export default {
	data (){
		return{
			error: "",
			errorFlag: false,
			users: []
		}
	},
	mounted: function() {
		this.getUsers();
	},
	methods: {
		getUsers: function() {
			this.$http.get('http://localhost/api/users')
			.then(function(response) {
				this.users = response.data;
			}, function(error) {
				this.error = error;
				this.errorFlag = true;
			});
		}
```

```
        }
}
</script>
```

4. Now in the template element, let's add the following. **Note**: we've added an error div that will be displayed if there is a problem getting the users. Also, we have left the 'View' link to implement later.

```
<template>
        <div>
                <div v-if="errorFlag" style="color: red;">
                        {{ error }}
                </div>

                <div id="users">
                        <table>
                                <tr v-for="user in users">
                                        <td>{{ user.username }}</td>
                                        <td><!--- view link here --></td>
                                </tr>
                        </table>
                </div>
        </div>
</template>
```

5. Run your development server and test that the users are displayed in the table. You can test the error div works by stopping your API server temporarily.

6.2 Exercise 5.2: Viewing one user

1. In your main.js file, edit the 'users' route to add a 'name' with value 'users.' Also add an new route for '/users/:userId.' This will use the same component but have a name of 'user'

```
const routes = [
        {
                path: "/",
                component: Home
        },
        {
                path: "/users/:userId",
                name: "user",
                component: Users
        },
        {
                path: '/users',
                name: "users",
                component: Users
        }
];
```

2. In Users.vue, add a 'View' link to the rows of the table for each user. Instead of using the traditional <a> tag, we will use <router-link>, passing in the routes name and parameter.

```
<table>
  <tr v-for="user in users">
      <td>{{ user.username }}</td>
      <td><router-link :to="{ name: 'user', params: { userId: user.user_id
}}">View</router-link></td>
  </tr>
</table>
```

3. Encompass the div with an id of ‘users’ in another div with the directive v-else. Before this, add a div with a v-if directive that will be displayed if $route.params.userId has a value (i.e. there is an ID given in the route).

```
<template>
  <div>
    <div v-if="errorFlag" style="color: red;">
      {{ error }}
    </div>

    <div v-if="$route.params.userId">

    </div>

    <div v-else>
      <div id="users">
        <table>
          <tr v-for="user in users">
            <td>{{ user.username }}</td>
            <td><router-link :to="{ name: 'user', params: { userId: user.user_id
}}">View</router-link></td>
          </tr>
        </table>
      </div>
    </div>
  </div>
</template>
```

4. Now inside this div, we will create our layout for displaying a single user. First start with another wrapper div with an id of ‘user’ and a link to send the user back to /users

```
<div v-if="$route.params.userId">
  <div id="user">
    <router-link :to="{ name: 'users'}">Back to Users</router-link>
  </div>
</div>
```

5. Add a method called ‘getSingleUser.’ This method will take in an ID and return the user with that ID.

```
getSingleUser: function(id){
       for(var i = 0; i < this.users.length; i++){
              if(this.users[i].user_id == id){
                     return this.users[i];
              }
       }
}
```

6. Now use this method to present the users details in a table.

```
<div v-if="$route.params.userId">
	<div id="user">
		<router-link :to="{ name: 'users'}">Back to Users</router-link>

		<br /><br />

		<table>
			<tr>
				<td>User ID</td>
				<td>Username</td>
			</tr>
			<tr>
				<td>{{ $route.params.userId }}</td>
				<td>{{ getSingleUser($route.params.userId).username }}</td>
			</tr>
		</table>
	</div>
</div>
```

7. Run your development server and test that everything is working.

6.3 Exercise 5.3: Deleting a user

1. To create a modal, we need to import bootstrap into our project. In your index.html, add links for the bootstrap css and js from the bootstrap CDN. Also add a link for jQuery from the google CDN.

Note: The CDN links that we provide in the sample code below may have expired. If you have trouble running bootstrap, get the latest CDN links from the bootstrap website: https://getbootstrap.com/docs/3.3/getting-started/

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">

    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
crossorigin="anonymous">

    <title>chat-app</title>
  </head>
  <body>
    <div id="app"></div>
    <script src="/dist/build.js"></script>

    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"
integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa"
crossorigin="anonymous"></script>
  </body>
</html>
```

2. Now we can create our modal. In the Users.vue file, underneath the table that displays a single user, add a delete button. This button will open our modal.

```
<table>
	<tr>
		<td>User ID</td>
		<td>Username</td>
	</tr>
	<tr>
		<td>{{ $route.params.userId }}</td>
		<td>{{ getSingleUser($route.params.userId).username }}</td>
	</tr>
</table>

<button type="button" class="btn btn-primary" data-toggle="modal"
data-target="#deleteUserModal">
	Delete
</button>
```

3. Now, underneath the user div, add the modal

```
<div class="modal fade" id="deleteUserModal" tabindex="-1" role="dialog"
aria-labelledby="deleteUserModalLabel" aria-hidden="true">
  <div class="modal-dialog" role="document">
    <div class="modal-content">
      <div class="modal-header">
        <h5 class="modal-title" id="deleteUserModalLabel">Delete User</h5>
        <button type="button" class="close" data-dismiss="modal" aria-label="Close">
          <span aria-hidden="true">&times;</span>
        </button>
      </div>
      <div class="modal-body">
        Are you sure that you want to delete this user?
      </div>
      <div class="modal-footer">
        <button type="button" class="btn btn-secondary" data-dismiss="modal">
          Close
        </button>
      </div>
    </div>
  </div>
</div>
```

4. Run your application and check that the modal is working. At this stage, you can only open and close the modal, no user will be deleted.

5. Add a delete method to the methods list. **Note**: router.push redirects the user back to /users.

```
deleteUser: function(user_id) {

  this.$http.delete('http://localhost/api/users/' + user_id)
    .then(function(response){

      for(var i = 0; i <= this.users.length; i++){
        if(user_id == this.users[i].user_id){
            this.users.splice(i, 1);
        }
      }

      this.$router.push('/users');

  }, function(error){
    this.error = error;
    this.errorFlag = true;
  });
}
```

6. Next, add a button to the modal that will delete the user. Note that we still have to close the modal (‘data-dismiss’). If we don’t close the modal, we are left with a page that is grayed out and unusable.

```
<button type="button" class="btn btn-primary" data-dismiss="modal"
v-on:click="deleteUser($route.params.userId)">
  Delete User
</button>
```

7. Run your application and test that it works

6.4 Exercise 5.4: Editing a user

You now have all the tools you need to implement the edit functionality. Here are the basic steps that you need to take.

1. Create an empty modal
2. Create the method and any required data values
3. Add a form to the modal, implement the edit form in the same way as Lab 7
4. Ensure all redirects are working
5. Test

6.5 Exercise 5.5: Creating a new user

Omitted from the previous exercises in this lab are plans for creating a new user. Implement this functionality on the /users page. This may be a modal or just a form on the page.

7 Exercise 6: Implement the rest of the application - optional

This last exercise is optional. Carry on working until you are comfortable with the concepts covered.

Using the details and API specification given in Lab 2, create the rest of the chat application. Use the previous exercises in this lab to help you structure and implement the application. Add styling to your application to make it responsive (using the HTML/CSS pre-lab for help).