Week 6: Integrating Vue.js with back-end

0: JavaScript Classes

Object-Oriented programming was a very popular paradigm in computer science. Object-oriented programming conceptually is: 'contained state management'. Which means objects keep track of their own properties.

Now, in fact we did something like that using JavaScript objects earlier.

e.g.

```
const obj = {
    property: 'value',
    someFunc: function () {
        console.log('function called')
        console.log(this.property)
    }
}
```

in the example above we have a variable named `obj` which is an object, and it has the following properties: `property`, and `someFunc` that store or interact with some form of the state of the object

Aside:
By state, it means the value of the object as a whole. e.g.

```
// initialization of the object : state 0
const example = {
    num: 1
}

// change the state of the object by modifying its property : state 1
example.num = 2 // now `example.num` has a value of 2


// adding a new property : state 2
example.newProp = function () {
    this.num = this.num + 1
}

// calling the `newProp` function property : state 3
// this is a state change since example.num will have
// the value of 3 after this line of code
example.newProp()
```

In majority of existing program languages, there is a concept of classes which is fairly equivalent to the idea of a blueprint of an object. (That is if you make a class, it is not defined as a variable, so it is not an object)

- This section is very important for later, when we will be making a `driver` for our back-end.

in Javascript 2015+ (before 2015 the code below would not work), we can define a class like so:

```
// class version
class MyClass { // convention: class names must start with capital letter
    myProperty = 1

    test() {
        console.log(this.property)
    }
}
// now since class itself is not an object, but a blueprint
// you need to create (more correct term : `instantiate`) the object like so
const myClass = new MyClass()
```

now here is a little secret about Javascript. Consider the code:

```
const myClass = {
    myPropert: 1,
    test: function () {
        console.log(this.property)
    }
}
```

This creates an object equivalent in terms of functionality to the one we made using class, primarily because `class` is more of syntatic-sugar, which means it just creates an object like the one above behind-the-scenes. The `class` keyword brings no new functionality of the new existing one, it just makes object-oriented programming more intuitive for people coming from different languages like C++ or Java.

Now, the reason why classes are nicer for us to use, is because they also bring us the notion of constructors. Which work like this:

```
class MyClass {
    constructor (firstVar, secondVar) {
        console.log('the constructor was called with:')
        console.log(`firstVar=${firstVar} and secondVar=${secondVar}`)
    }
}

// now you instantiate the object
const obj = new MyClass(1, 2)
```

Now, as you might guess, the code above will print the following

```
the constructor was called with:
firstVar=1 and secondVar=2
```

because, behind the scene, `new MyClass(1, 2)` is a function. That is called a constructor because it in essence *constructs* the object from class.
So `constructor` *= function that creates the object*

in fact in pure old-school (pre-class) Javascript, the equivalent constructor would be

```
function MyClass(firstVar, secondVar) {
    this.firstVar = firstVar
    this.secondVar = secondVar
}

// if you previously defined a class named: MyClass
// the above definition would fail, as they are equivalent

// then to construct the object you would do:
// note the `new` keyword, this part is required
const obj = new MyClass(firstVar, secondVar)
```

Aside: Why is the `new` keyword required?

As we talked about: the constructor, behind the scenes, is just a function.

So, if you do something like this:

```
function MyClass(firstVar, secondVar) {
    this.firstVar = firstVar
    this.secondVar = secondVar
}
const obj = MyClass(1, 2)
```

this will execute `MyClass()` as a function, and nothing will happen. `obj` will be `undefined` since the constructor did not return anything, so there was no assignment to the `obj` variable

however, if you use `new` keyword, you are telling JavaScript, that this will create an object after the code in the function gets executed.

```
function MyClass(firstVar, secondVar) {
    this.firstVar = firstVar
    this.secondVar = secondVar
}
const obj = new MyClass(1, 2)
```

Now `obj` has a value of an object with the properties: `firstVar`, and `secondVar` as expected.

Now, this should be enough to get started with basic object-oriented concepts that will be presented in the remainder of this lesson-plan.

1. Adding functionality to our modals

Before we get onto integration of our back-end, I would like firstly, finish up the work we did last week with front-end. By that I mean, give our modals (DeleteModal, ItemModal) proper functionality, such as closing and opening. If you downloaded the code from last week, you might see that popups (modals) do not work since their interactability is not implemented.

Let's start with **DeleteModal**, as it's a bit easier:

1. Note that you need to click on the little **x** on the Item Component.
2. our Item Component doesn't `emit` any events, so we need to define such emitters inside the Item component. The events are: `open`, and `delete`. For DeleteModal, we will only define `delete` event emitter
3. The way emitters work:

When you define an emit of an event, what happens is basically the current component that will emit an event will notify its parent component that a certain event was emitted.

A good example:

consider **a hypothetical component: Parent.vue**

```
<template>
    <p @click="doSomething()">Some text</p>
</template>
```

in this case the child component is the default HTML tag: p. And the event that it emits in this case is the **click** event. (a user clicks on the `p` element, and then `doSomething()` gets ran)

Now, say you have an actual Vue component as a child.

a hypothetical component: Parent.vue

```
<template>
    <ChildComponent @click="doSomething()"></ChildComponent>
</template>
```

a hypothetical component: ChildComponent.vue

```
<template>
    <p>Some text</p>
</template>
```

By default, you won't be able to listen to `click` event, as ChildComponent doesn't emit any eventsm so we need to define our own emitter. We can do it like so:

a hypothetical component: ChildComponent.vue

```
<template>
    <p @click="$emit('myevent')">Some text</p>
</template>
```

Usually, emmiting works in the following manner:

```
this.$emit('eventname', somevalue)
```

where `eventname` is a name of an event that you name, and want to be referenced by parent components. And `somevalue` is a value that can be passed to an event.

Now the following will work in: ParentComponent.vue

```
<template>
    <ChildComponent @myevent="doSomething()"></ChildComponent>
</template>
```

Now, we will have to do something similar with DeleteModal and Item

Item.vue
This file should have the following code:

```
<template>
    <div class="notification is-dark">
        <button class="delete" @click="$emit('delete')"></button>
        <p class="has-text-centered">{{ name }}</p>
    </div>
</template>
<script>
export default {
    props: {
        name: {
            type: String,
            required: true
        }
    }
}
</script>
```

Then, in **DeleteModal.vue** you should add `$emit('close')` for closing

the file should look like this:

```
<template>
    <section class="section" v-if="item != null && active">
        <div class="modal is-active">
            <div class="modal-background"></div>
            <div class="modal-content">
                <article class="message">
                    <div class="message-header">
                        <p class="is-unselectable">Delete this item?</p>
                        <button
                            class="delete"
                            aria-label="delete" @click="$emit('close')"></button>
                    </div>
                    <div class="message-body has-text-centered">
                        <p>{{ item && item.name }}</p>
                        <br/>
                        <div class="buttons is-centered">
                            <a
                                class="button is-danger is-unselectable">
                                Delete</a>
                            <a
                                class="button is-dark is-unselectable"
                                @click="$emit('close')">Cancel</a>
                        </div>
                    </div>
                </article>
            </div>
        </div>
    </section>
</template>
<script>
export default {
    props: {
        item: Object,
        active: {
            type: Boolean,
            required: true
        }
    },
}
</script>
```

Now, lastly in **Dashboard.vue** you need to listen for **'close'** and **'delete'** events

Where you use DeleteModal:

```
<DeleteModal
    @close="closeDeleteModal()"
    :active="activeDeleteModal"
    :item="selectedItem" />
```

Where you use Item

```
<Item v-for="(item, index) in items"
    :name="item.name"
    :key="index"
    @delete="openDeleteModal(item)" />
```

in methods part of the Vue object (near the bottom)"

```
openDeleteModal (item) {
    this.activeDeleteModal = true
    this.selectedItem = item
},
closeDeleteModal () {
    this.activeDeleteModal = false
    this.selectedItem = { } // reset the selected item
},
```

Now, after all this work, we are able to open and close the DeleteModal flawlessly.

Now, onto **ItemModal**

Item.vue

This file should have the following code:

```
<template>
    <div class="notification is-dark">
        <button class="delete" @click="$emit('delete')"></button>
        <p class="has-text-centered" @click="$emit('open')">{{ name }}</p>
    </div>
</template>
<script>
export default {
    props: {
        name: {
            type: String,
            required: true
        }
    }
}
</script>
```

Now, **ItemModal.vue**

```
<template>
    <section class="section" v-if="item != null && active">
        <div class="modal is-active">
            <div class="modal-background"></div>
            <div class="modal-content">
                <article class="message">
                    <div class="message-header">
                        <p>Edit Item</p>
                        <button class="delete" aria-label="delete"
                         @click="$emit('close')"></button>
                    </div>
                    <div class="message-body has-text-centered">
                        <ItemForm
                            @cancel="$emit('close')"
                            :item="item"
                            :isModal="true"
                        />
                    </div>
                </article>
            </div>
        </div>
    </section>
</template>
<script>
import ItemForm from './ItemForm'
export default {
    props: {
        item: Object,
        active: {
            type: Boolean,
            required: true
        }
    },
    components: {
        ItemForm
    }
}
</script>
```

Aside: ItemForm component has a listener for `cancel` event too.
Here is how ItemForm.vue should look like near Submit and Cancel buttons:

```
<div class="field">
    <div class="control buttons is-centered">
        <button type="submit" class="button is-success">Submit</button>
        <button class="button is-dark"
            @click.prevent="$emit('cancel')"
            >Cancel</button>
    </div>
</div>
```

Note:
I wrote `@click.prevent`, which is a recommended type of event, as sometimes, the buttons have default behaviour. For example, if a button is a submit button (type="submit"), it will reload the page when pressed. In Vue.js, we don't want that behaviour as it is considered very impractical, and needless. Thus, to avoid that type of behaviour on `button` elements, you can write: `@click.prevent`.

If you know pure JavaScript document in HTML, then this is calls `event.preventDefault()` behind the scenes, which is why this function works.

Some bug fixes that came from last week:

Bugfix 1: please correct a snippet in `AddItem.vue` to the following:

```
<ItemForm
    title="Add a new item"
    :isNew="true"
    @cancel="$router.go(-1)" />
```

Bugfix 2: please add a function: created (which gets ran after creation of the component like so)

```
export default {
    created: function () {
        if (!this.isNew) {
            const { name, type, completed } = this.item

            this.name.value = name
            this.type.value = type
            this.completed = completed
        }
    },
    data: function () {
        return {
            name: {
                value: '',
                pristine: true,
                valid: true
            },
            type: {
                value: '',
                pristine: true,
                valid: true
            },
            completed: false,
            types: types
        }
    },
    props: {
        item: Object,
        title: String,
        isNew: {
            type: Boolean,
            default: false
        },
        isModal: {
            type: Boolean,
            default: false
        }
    },
    methods: {
        checkValidness(field) {
            const value = this[field].value
            const ruleset = ItemRuleSet({ [field]: value })

            this[field] = {
                valid: ruleset[field].valid,
                pristine: false,
```

```
                value: value
            }
        }
    }
}
```

Bugfix 3: please update the ./router/index.js to the following:

```
import Vue from 'vue'
import VueRouter from 'vue-router'
import Home from '../components/Home.vue'
import Dashboard from '../components/Dashboard.vue'
import AddItem from '../components/AddItem.vue'
import Error from '../components/Error'

Vue.use(VueRouter)

export default new VueRouter({
    mode: 'history',
    routes: [
        { path: '/dashboard', component: Dashboard },
        { path: '/add-item', component: AddItem },
        { path: '/', component: Home },
        { path: '*', component: Error}
    ],
    linkActiveClass: 'is-active'  // needed because of Bulma
})
```

and in Navbar.vue please add `exact` keyword to **router-link** element
so it will look like this:

```
<router-link v-for="(option, index) in options" :key="index" exact
    class="navbar-item is-unselectable"
    :to="urls[index]"
    @click="toggleActiveItem(index)">{{ option }}</router-link>
```

*******END OF BUGFIXES*******

Now, we put it all together in **Dashboard.vue**:

we not listen to Item component's event: open

```
<Item v-for="(item, index) in items"
    :name="item.name"
    :key="index"
    @open="openItemModal(item)"
    @delete="openDeleteModal(item)" />
```

we, use ItemModal like so:

```
<ItemModal
    @close="closeItemModal()"
    :active="activeItemModal"
    :item="selectedItem" />
```

and here are the function definitions that we are now using:

```
openItemModal (item) {
    this.activeItemModal = true
    this.selectedItem = item
},
closeItemModal () {
    this.activeItemModal = false
    this.selectedItem = { }
}
```

Now our project is ready for easy integration.

2. Adding Vuex (State Management) to Vue

We talked about the state before. What Vuex brings, is a global space that contains shareable variables, functions across all the components.

Now, there are two schools of thought here: some would think it is an overkill (too powerful) for an app of our size (which is pretty small, relatively speaking), and another: if it makes things easier, why not?

For this course, I think it makes sense to use it, for 2 main reasons:

1. learning new things is always good, and
2. if you would want to develop the app further (adding new features), the library and its setup is already here, and writing further code will be easier (no need to think about

restructuring code)

What is Vuex? (from website: https://vuex.vuejs.org/):

```
Vuex is a state management pattern + library for Vue.js applications.
It 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.
```

Here is how you can add it:

1. in /src folder, please make a /store folder with `index.js` file. That's where we will keep our **store**
2. in that folder, please make the following files:

`actions.js`, `state.js`, `mutations.js`

(We'll talk about what each of them now)

These are the parts that make up a store:

- actions
- state
- mutations
- getters (not covered here, but used to compute values, and store them for later use, if the state changes)

Aside: theoretically, the contents of these files can be in one file, but it is considered bad practice as the files individually could potentially get pretty lengthy. So a good practice is to refactor these parts that make up a store from the start

1. Copy the following into `state.js`:

```
export const defaultState = function () {
    return {
        items: [],
        filters: [],
        types: [],
        selectedType: 0,
        selectedFilter: 0,
    }
}
```

As you might have guessed, this is our default global state, which contains no information (all the properties have empty-like values).

No information is a good default state as it just indicates that no information has been loaded from the server. In fact, if you would like you can use the emptiness of `types` and `filters` arrays to check if you have loaded any data. (e.g. if `types.length === 0`, show a loading spinner, or render a component like the `Error.vue` one)

Note that default state is still a function, just like the `data` property in a Vue Component.

4. Copy the following into `mutations.js`

```
import { defaultState } from './state'

const resetState = function(state) {
    Object.assign(state, defaultState())
}
const setItems = function (state, payload) {
    state.items = payload
}
const setFilters = function (state, payload) {
    state.filters = payload
}
const setFilter = function (state, payload) {
    if (payload > -1 && payload < state.filters.length) {
        state.selectedFilter = payload
    }
}
const setTypes = function (state, payload) {
    state.types = payload
}
const setType = (state, payload) => {
    if (payload >= 0 && payload < state.types.length) {
        state.selectedType = payload
    }
}

export const mutations = {
    resetState,
    setItems,
    setFilters,
    setFilter,
    setTypes,
    setType
}
```

As you can see, this file handles mutations to our state, such as setting the `types`, and `filters` properties and setting `items`. Note that these types of changes must be synchonous. In other words, you cannot use await Promises with a `then`, or `catch` functions (such as making API requests), or other asynchonous actions. For such actions, the next part of the store is used.

5. Copy the following into the `actions.js`

```
import server from './../driver'

const getMetaData = function (context) {
    const { commit } = context

    return server.meta()
    .then(function (data) {
        commit('setFilters', data.filters)
        commit('setTypes', data.types)
    })
    .catch(function(error) {
        return error
    })
}
const getItems = function (context) {
    const { commit, state } = context

    if (state.filters.length == 0 || state.types.length == 0) {
      throw new Error('no items to get')
    }
    const filter = state.filters[state.selectedFilter]
    const type = state.types[state.selectedType]

    return server.getAllItems(type, filter)
    .then(function(items) {
        commit('setItems', items)
    })
    .catch(function(error) {
        return error
    })
}
const getItem = function (context, payload) {
    return server.getItemById(payload)
    .then(function (data) {
        return data
    })
    .error(function (error) {
        return error
    })
}
const deleteItem = function (context, id) {
    //note we just pass `id` as the payload
    const { dispatch } = context

    return server.deleteItem(id)
    .then(function() {
```

```
        return dispatch('getItems')
    })
    .catch(function(error) {
        return error
    })
}
const addItem = function (context, payload) {
    const { dispatch } = context
    const { name, type, completed } = payload

    return server.addItem(name, type, completed)
    .then(function() {
        dispatch('getItems')
    })
    .catch(function(error) {
        return error
    })
}
const patchItem = function (context, payload) {
    const { dispatch } = context
    const { id, name, type, completed } = payload

    return server.patchItem(id, name, type, completed)
    .then(function() {
        return dispatch('getItems')
    })
    .catch(function(error) {
        return error
    })
}
const init = function (context) {
    const { commit, dispatch } = context

    return dispatch('getMetaData')
    .then(function () {
        return dispatch('getItems')
    })
    .catch(function() {
        commit('resetState')
    })
}
export const actions = {
    getMetaData,
    getItems,
    getItem,
    deleteItem,
```

```
    addItem,
    patchItem,
    init
}
```

So, first of all you might see at the very top that we are trying to import a `server` variable from a non-existant file. Think of it as a black-box for now, we'll get to the implementation of it in the next section.

the key part to understand is that an action is a function that gets passed two parameters: `context` and `payload`

context contains the context of when the action gets called. Things like *state*, *commit*, *dispatch* (this gets passed automatically, i.e. you have no control over this variable)

payload data that you can pass a parameter yourself (i.e. you have full control over what this is)

Aside: parts of context

state: an object that contains the most up-to-date state of the application
commits: a function that triggers a mutation in the string. (must be synchronous)
dispatch: a function that triggers an action in the string. (can be asynchronous)

6. Now, let's actually install `vuex`. In your terminal please execute the following command

```
npm install --save vuex
```

7. Now finally, we can copy the following into `index.js` in the /store

```
import Vuex from 'vuex'
import Vue from 'vue'
import { mutations } from './mutations'
import { actions } from './actions'
import { defaultState } from './state'

Vue.use(Vuex) // needed to add the module globally to Vue

const store = new Vuex.Store({
  state: defaultState(),
  mutations,
  actions
})

export default store
```

So, now our store is ready, but we need to add it to our application.

To add it to the application, please navigate to `main.js` and import `store` and add it to the Vue object like the router.

your `main.js` file should look like this:

```
import Vue from 'vue'
import 'bulma/css/bulma.min.css'

import App from './App.vue'
import router from './router'
import store from './store'

new Vue({
  render: h => h(App),
  router,
  store
}).$mount('#app')
```

now as we add store to the Vue instance, we can have access to `$store` variable in every component of our application.

To access the `$store`'s states, you must use `$store.state`, and there you can use the variable

3: Server API class

In the previous section we defined a black-box class called Server that handles every API request. This is considered a good practice to define a class with methods that encapsulate REST API requests. Main reason is if something changes in the request, you only have one place to change, rather than find all the relevant requests throughout the whole codebase, and update them to match the new changes.

Such a class is sometimes called a Driver class, and we will implement it as a class.

in the **/src** folder, please create a folder called: **/driver**

and then there I made a JSON file that contains all our end-points that we made on the back-end.

1. Please, create a file named: `api.json`, and paste the following:

```
{
    "api" : {
        "ping": {
            "method": "get",
            "url": "/ping"
        },
        "items": {
            "meta": {
                "method": "get",
                "url": "/items/meta"
            },
            "getItemById": {
                "method": "get",
                "url": "/items/:id"
            },
            "getAllItems":{
                "method": "get",
                "url": "/items/:type/:filter",
                "queryParams": {
                    "numRecords": {
                        "type": "number"
                    },
                    "page": {
                        "type": "number"
                    }
                }
            },
            "addItem": {
                "method": "post",
                "url": "/items"
            },
            "deleteItem": {
                "method": "delete",
                "url": "/items/:id"
            },
            "patchItem": {
                "method": "patch",
                "url": "/items/:id"
            }
        }
    }
}
```

Note: obviously this is just a suggestion, you may not even need a JSON file, but I think it's a good idea as it shows which API you have defined for yourself, and others (if you choose to make

your project open-source)

2. We will be using a 3rd-party library that can handle making requests for us, called: **`axios`**

To install it, please execute the following in your terminal:

```
npm install --save axios
```

Now, we should be able to run **`axios`** in our app to handle all our requests

3. Now please copy the following code into `index.js` in the /**driver** folder:

```
import { create } from 'axios' // create is the object that will make the requests
const endpoints = require('./api.json')

class ServerAPI {
    constructor () { // class constructor
        const port = process.env.PORT || 3000
        const domain = process.env.URL || `http://localhost:${port}`
        this.url = `${domain}/api`
        this.request = create({
            baseURL: this.url, // defines which url make the request to
            validateStatus: function(status) {
                return status > 0
            }
            // validateStatus is important so it doesn't throw
            // an error for non 200 statuses (default axios behavior)
        })
    }

    /**
     * GET /api/items/meta
     */
    meta () {
        const { method, url } = endpoints['api']['items']['meta']
        return this.makeRequest(method, url)
    }

    /**
     * GET /api/items/getItemById
     */
    getItemById (id) {
        const { method, url } = endpoints['api']['items']['getItemById']
        const urlWithParams = this.paramify(url, { ':id': id })
        return this.makeRequest(method, urlWithParams)
    }

    /**
     * GET /api/items/getAllItems
     */
    getAllItems (type, filter) {
        const { method, url } = endpoints['api']['items']['getAllItems']
        const urlWithParams = this.paramify(
            url, { ':type': type, ':filter': filter }
        )
        return this.makeRequest(method, urlWithParams)
    }
```

```
    /**
     * POST /api/items/addItem
     */
    addItem (name, type, completed) {
        const data = { name, type, completed }
        const { method, url } = endpoints['api']['items']['addItem']
        return this.makeRequest(method, url, data)
    }

    /**
     * DELETE /api/items/deleteItem
     */
    deleteItem (id) {
        const { method, url } = endpoints['api']['items']['deleteItem']
        const urlWithParams = this.paramify(url, { ':id': id })
        return this.makeRequest(method, urlWithParams)
    }

    /**
     * PATCH /api/items/patchItem
     */
    patchItem (id, name, type, completed) {
        const data = {
            name,
            type,
            completed
        }
        const { method, url } = endpoints['api']['items']['patchItem']
        const urlWithParams = this.paramify(url, { ':id': id })
        return this.makeRequest(method, urlWithParams, data)
    }

    /* helper functions */

    // puts values as params into the url like so: url.com/type/filter/id1234
    /**
    * @param url: String - url
    * @param params: Object - contains key-value pairs of params
    */
    // returns an updated url
    paramify (url, params) {
        const keys = Object.keys(params)
        for (let key of keys) {
            url = url.replace(key, params[key])
        }
        return url
```

```
    }

    // logs errors, and returns the message
    /**
    * @param error: Object - error object
    */
    // returns a string part of the error
    handleError (error) {
        /*eslint-disable*/
        console.log(error)
        return error.message
    }

    /**
    * @param method: String - HTTP Method
    * @param url: String - url
    * @param body: Object - contains the body
    */
    /* returns: data and headers from server */
    makeRequest (method, url, body) {
        return this.request[method](url, body)
        .then(function(response) {
            return Promise.resolve(response.data)
        })
        .catch(function (e) {
            return Promise.reject(e)
        })
    }
}

export default new ServerAPI() // export the instantiated class
```

Key notes:

- Please have a look at how different methods work. And note how all of them use the method: `makeRequest`
- we defined **`this.request`** as a property of the class that gets defined in the constructor, and it is an axios object. This object does a very implicit thing that JavaScript allows.

Consider the following example:
If we are to do a more explicit request, it would look like this:

```
// say your REST method is: `get`,
// then you would make the following call
this.request.get(url, body)

// similarly if your REST method is: `post`,
this.request.post(url, body)
```

And similar thing happens for other REST methods.
We were able to simplify (do without if-statements), because if you recall, `this.request` is an object, and the `post`, `get`, etc, are simply properties of that object, hence we can just use the indexing syntax in javascript to access those methods

- Also note that if the request succeeds then the value gets passed to `then` part of the Promise, and if it fails it goes to `catch`
- Lastly, note that we return an instantiation of the class Server API (actual object, rather than just a class)

Now everything in the store works, and we should be able to make our requests to the backend (assuming the back-end is running [npm run start])

4. Integrating our GET requests

We started making our first requests for GET methods, so it makes sense to follow that pattern. Luckily, we finished adding modals' functionality, so these upcomingsections should be easier.

We have the following GET requests that we need to use:

```
- GET /items/meta   # used for getting types and filters
- GET /items/:type/:filter   #used for filtering items
- GET /items/:id   # used for getting individual item details
```

Note, we won't need to make a request for GET /items/:id, since we will query all the filtered items, and just pass them individually.

So, we can start by going to `App.vue` and there we want to init our variables for our store, as `App.vue` is the outer-most component, and is always created for obvious reasons.

Thanks to all the machinary that we have defined in previous section, retriving data from our

server will just be a matter of calling actions/mutations from our store with appropriate values.

So we are only adding `created` function in App.vue, like so:

App.vue

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

<script>
import Navbar from './components/Navbar'

export default {
  name: 'App',
  created: function () {
    this.$store.dispatch('init')
  },
  components: {
    Navbar
  }
}
</script>
<style>
  .content-section {
    margin-top: -3.25rem;
    padding-top: 3.25rem;
  }
</style>
```

After this change, we should have access to `types`, `filters` and `items` in our state's store, so let's remove dependancies on our fake/non-server data in other components

Dashboard will have the most changes.
So it will look like this after the changes:

Dashboard.vue

```
<template>
    <div class="content-section">
        <section>
            <div class="hero is-dark is-medium">
                <div class="hero-body">
                    <div class="container" >
                        <h1 class="title has-text-centered is-unselectable">
                        Welcome!
                        </h1>
                        <h1 class="subtitle has-text-centered is-unselectable">
                        Your Dashboard
                        </h1>
                    </div>
                </div>
            </div>
            <div class="tabs is-centered">
                <ul v-if="$store.state.types && $store.state.types.length > 0">
                    <li v-for="(type, index) in $store.state.types"
                        :key="index"
                        :class="{ 'is-active' : $store.state.selectedType === index }"
                        @click="pickType(index)">
                        <a>{{ type }}</a>
                    </li>
                </ul>
            </div>
            <div class="tabs is-small is-toggle is-centered has-text-centered">
                <ul
                    v-if="$store.state.filters && $store.state.filters.length > 0">
                    <li v-for="(filter, index) in $store.state.filters"
                        :key="index"
                        :class="{ 'is-active' : $store.state.selectedFilter === index }"
                        @click="pickFilter(index)">
                        <a>{{ filter }}</a>
                    </li>
                </ul>
            </div>
        </section>
        <section class="section"
            v-if="$store.state.items && $store.state.items.length > 0">
            <Item v-for="(item, index) in $store.state.items"
                :name="item.name"
                :key="index"
                @open="openItemModal(item)"
                @delete="openDeleteModal(item)" />
        </section>
        <section class="section" v-else>
```

```
            <div class="notification is-white">
                <p class="has-text-centered is-unselectable">
                No Items to Show
                </p>
            </div>
        </section>
        <DeleteModal
            @close="closeDeleteModal()"
            :active="activeDeleteModal"
            :item="selectedItem" />
        <ItemModal
            @close="closeItemModal()"
            :active="activeItemModal"
            :item="selectedItem" />
    </div>
</template>
<script>
import ItemModal from './ItemModal'
import DeleteModal from './DeleteModal'
import Item from './Item'

export default {
    data: function () {
        return {
            selectedItem: { },
            activeItemModal: false,
            activeDeleteModal: false
        }
    },
    methods: {
        pickType (index) {
            this.$store.commit('setType', index)
            this.$store.dispatch('getItems')
        },
        pickFilter (index) {
            this.$store.commit('setFilter', index)
            this.$store.dispatch('getItems')
        },
        openDeleteModal (item) {
            this.activeDeleteModal = true
            this.selectedItem = item
        },
        closeDeleteModal () {
            this.activeDeleteModal = false
            this.selectedItem = { }
        },
```

```
        openItemModal (item) {
            this.activeItemModal = true
            this.selectedItem = item
        },
        closeItemModal () {
            this.activeItemModal = false
            this.selectedItem = { }
        }
    },
    components: {
        DeleteModal,
        ItemModal,
        Item
    }
}
</script>
```

Note the changes in the following two functions:

```
pickType (index) {
    this.$store.commit('setType', index) //sets the index of the new type
    this.$store.dispatch('getItems') // refresh the items with new type set
},
pickFilter (index) {
    this.$store.commit('setFilter', index) //sets the index of the new filter
    this.$store.dispatch('getItems') // refresh the items with new filter set
},
```

Next, in **ItemForm.vue** we want to use `types` from the `$store`, rather than the predefined once. So, you can replace it like so:

```
<div class="select">
    <select
        v-model="type.value"
        @change="checkValidness('type')">
        <option value="" disabled selected>
        Select your option</option>
        <option
            v-for="(type, index) in $store.state.types"
            :key="index">{{ type }}</option>
    </select>
</div>
```

Also make sure that you remove all the refences to `types`, in the import

```
import { ItemRuleSet } from './../rulesets/ItemRuleset'
```

and `data` object

```
data: function () {
    return {
        name: {
            value: '',
            pristine: true,
            valid: true
        },
        type: {
            value: '',
            pristine: true,
            valid: true
        },
        completed: false
    }
},
```

ItemRuleSet.js

in this file we remove `types` and `filters`

```
export const ItemRuleSet = function (object) {
    const { name, type } = object
    const ruleset = {
        'name': {
            valid: name != null && name.trim().length > 0
        },
        'type': {
            valid: type != null && type.trim().length
        },
    }
    return ruleset
}
export const ItemRequiredFields = ['name', 'type']
```

Now, with all our changes, GET requests should work

5. Integrating our POST request

Now onto the POST request.
We only have one request for this method

```
- POST /items # used for creating a new item
```

For creating a new item, we only need to modify one file: **ItemForm.vue**

1. make a new method: `submit()`, which will ran when submit button is pressed.
2. modify the Submit and Cancel template to:

```
<div class="field">
    <div class="control buttons is-centered">
        <button class="button is-success" @click.prevent="submit()">Submit<button>
        <button class="button is-dark" @click.prevent="$emit('cancel')">Cancel</button>
    </div>
</div>
```

3. in `submit()` method, please write the following code:

```
submit() {
    // need to make sure the values are valid
    if (this.name.valid && this.type.valid) {
        const item = {
            name: this.name.value,
            type: this.type.value,
            completed: this.completed
        }
        if (this.isNew) {
            this.$store.dispatch('addItem', item)
        }
    }
}
```

6. Integrating our PATCH request

Now onto the PATCH request.
We only have one request for this method

```
- PATCH /items/:id # used for updating a particular item
```

in **ItemForm.vue** we defined a POST request for an item in **submit()** function.

now to do a patch, we simply need to add a bit of code to only that function. Here is the final result:

```
submit() {
    // need to make sure the values are valid
    if (this.name.valid && this.type.valid) {
        const item = {
            name: this.name.value,
            type: this.type.value,
            completed: this.completed
        }
        if (this.isNew) {
            this.$store.dispatch('addItem', item)
        } else {
            item.id = this.item._id // must update by existing id
            this.$store.dispatch('patchItem', item)
        }
    }
}
```

7. Integrating our DELETE request

Now onto the DELETE request.
We only have one request for this method

```
- DELETE /items/:id # used for deleting a particular item
```

In **DeleteModal.vue** is where the delete popup is defined.
If you look at the template, you can see the 'Submit' html.

So we should do similar to what we did in PATCH route.

please a methods property and there a new method called: `deleteItem`:

```
deleteItem() {
    if (this.item._id != null) {
        this.$store.dispatch('deleteItem', this.item._id)
        this.$emit('close')
    }
}
```

and for submit in html:

```
<div class="buttons is-centered">
    <a class="button is-danger is-unselectable" @click.prevent="deleteItem()">Delete</a>
    <a class="button is-dark is-unselectable" @click.prevent="$emit('close')">Cancel</a>
</div>
```

Aside: we can't name our function `delete` as that's a reserved keyword by JavaScript

Now all our requests work, and front-end is ready

8: Bugfixes in our server that made it in from previous lessons.

There is a bug-fix that is required in `/routes/items/index.js` in `/server` part of our code.

1. Name of the Filter,
2. missing function
3. res/response and req/request inconsitancy

Here is the updated file:

`/routes/items.js`

```
const express = require('express')
const items = express.Router()

const { Item, TYPES } = require('./../../models/item')
const FILTERS = ['Current', 'Completed', 'Suggested' ]

items.get('/meta', function (_, res) {
    res.send({
        types: TYPES,
        filters: FILTERS
    })
})

// GET /items
items.get('/', function(_, res) {
    Item.find().then(function(items) {
        res.send(items)
    })
})


// GET /items/:id
items.get('/:id', function(req, res) {
    const id = req.params.id

    if (id != null) {
        Item.findById(id).then(function(item) {
            if (item != null) {
                res.send(item)
            } else {
                res.status(404).send({
                    message: 'item not found'
                })
            }
        }).catch(function(e) {
            console.log(e)
            res.status(500).send({
                message: 'something went wrong'
            })
        })
    } else {
        res.status(400).send({
            message: 'please provide `id`'
        })
    }
})
```

```
const getTypeFilterQuery = function (type, filter) {
    if (FILTERS[0] == filter) { // current
        return {
            completed: false,
            type: type
        }
    } else if (FILTERS[1] == filter) { //completed
        return {
            completed: true,
            type: type
        }
    } else {
        return { // default
            type: type
        }
    }
}

items.get('/:type/:filter', function(req, res) {
    const type = req.params.type
    const filter = req.params.filter

    if (type != null && filter != null) {
        if (TYPES.includes(type) && FILTERS.includes(filter)) {
            const query = getTypeFilterQuery(type, filter)
            Item.find(query).then(function(items) {
                res.send(items)
            }).catch(function(e) {
                console.log(e)
                res.status(500).send({
                    message: 'something went wrong'
                })
            })
        } else {
            res.status(400).send({
                message: 'please provide a valid `type` and `filter`'
            })
        }
    } else {
        res.status(400).send({
            message: 'please provide `type` and `filter`'
        })
    }
})
```

```
// then: happy paths
// catch: unhappy paths

items.post('/', function(req, res) {
    const { name, type } = req.body

    if (name != null && type != null && TYPES.includes(type)){
        const item = new Item({
            name,
            type,
            completed: false,
            createdAt: Date.now()
        })

        item.save().then(function(item) {
            res.send(item)
        }).catch(function(e) {
            console.log(e)
            res.status(500).send({
                message: 'something went wrong'
            })
        })
    } else {
        res.status(400).send({
            message: 'please provide `type` and `name`'
        })
    }
})


items.delete('/:id', function(req, res) {
    const { id } = req.params

    if (id != null) {
        Item.findByIdAndDelete(id).then(function(item) {
            if (item != null) {
                res.send(item)
            } else {
                res.status(404).send({
                    message: 'item not found'
                })
            }
        }).catch(function(e) {
            console.log(e)
            res.status(500).send({
                message: 'something went wrong'
```

```
            })
        })
    } else {
        es.status(400).send({
            message: 'please provide `id`'
        })
    }
})

const createUpdateQuery = function (name, type, completed) {
        const updateQuery = {
        completed
    }

    if (name) updateQuery.name = name
    if (type) updateQuery.type = type

        return updateQuery
}

items.patch('/:id', function (req, res) {
    const { id } = req.params
    const { name, type, completed } = req.body

    if (id != null) {
        const updateQuery = createUpdateQuery(name, type, completed)
        Item.findByIdAndUpdate(
            id,
            { $set: updateQuery },
            { new: true }
        )
        .then(function(item) {
            if (item != null) {
                res.send(item)
            } else {
                res.status(404).send({
                    message: 'no item found'
                })
            }
        })
        .catch(function(e) {
            console.log(e)
            res.status(500).send({
                message: 'something went wrong'
            })
        })
```

```
    } else {
        res.status(400).send({
            message: 'please provide `id` as a param'
        })
    }
})

module.exports = items
```

9: Re-structuring our project

If you have your project in a **/webdev** folder
You most likely have **/webdev/server/front-end** structure, which is okay in some instances if you want the server to render the front-end. However, this could be problematic. So a better structure would be to have two different directories in **/webdev**: **/server** and **/front-end**, i.e. move **/front-end** out of **/server**

Now if you have done everything above, you could run two terminals, and navigate to `/webdev/server` and run:

```
npm start
```

and in the other terminal: navigate to `/webdev/front-end` and run:

```
npm run serve
```

and now you can test your hard work. Hopefully everything will work.

This concludes this course

Thank you so much for coming out to the lessons, and taking interest in this course! The code is available online