Week 4: REST API Continued, and MongoDB

This week, we will setup our database schema and modify our REST calls to use it. Then, we will start a new part of the project that involves the front-end.

Now, we will continue from last week onto:

0: Try running the app first

1. run the project.
2. if you see:

```
    app listening on port 3000
```

everything is good to go! 😃

if not, please download the code from github, open it and run `npm install`. If this doesn't work, please message me on slack and we can figure it out.

1: Defining our first Mongoose Schema

MongoDB, is a system that is optimized for storing data.
Each data is called a document (**JSON**-like object) and a document can have different fields, just like properties in **JSON**.

One biggest advantage is that MongoDB, is *schema-less*, which means that fields are not necessarily defined, neither are their types. This helps greatly in starting a new project and adding new features/fields as you further develop your project.

But, since we already know what fields we need for our project (the ones we defined last week), we can go off of those, to define the schema.

And then, we will define a model, that we will be able to use in our project from that.

Think of it as:

- **schema**: definition/blueprint of the document (how it should look like) for MongoDB
- **model**: actual object based off of the schema that we as programmers can use.

First of, let's create a `/model` directory, and inside it, we can add `items.js` file. That is where we will define everything.

1. Import `mongoose` from the `/db` directory.

```
const mongoose = require('./../db/mongoose')
```

2. Let's define types that an item can be. (it is convention, that variables with constant values have their names capitalized)

```
const TYPES = ['Movies', 'TV Shows', 'Video Games', 'Books']
```

Then, we can start defining the schema which can be done by providing a JavaScript Object, to a mongoose function. Like so:

```
const ItemSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true,
        trim: true
    },
    type: {
        type: String,
        required: true,
        enum: TYPES
    },
    completed: {
        type: Boolean,
        required: true
    },
    addedAt: {
        type: Date,
        required: true
    }
})
```

Then we can define the following fields in the schema:

- *outer-most field*: MongoDB field
- **type**: type of the object (String, Number, Boolean, Date, ObjectId, Array, Buffer, and some other ones)
  Test
- **required**: if the field is required in creation of the document
- **enum**: an array of values that are allowed for that field
- **trim**: removes white spaces before, and after the word (e.g. " test " becomes "test")

There are other ones that you can find at:
https://mongoosejs.com/docs/guide.html

Aside:
The following are equivalent in JavaScript (ES6/ES2015+)

```
const title = "some title"
const name = "some name"
const a = { title: title, name: name }
const a = { title, name }
```

Then, we can make a model, and export the model, and the types to the outside files.

```
const Item = mongoose.model('Item', ItemSchema)
module.exports = { Item, TYPES }
```

2: Import our model to the routes, removing the local array, and creating FILTERS

in `/routes/items/index.js` at the very top we can import it like so:

```
const { Item, TYPES } = require('./../../models/item')
```

Then, delete `itemsArr` and `id` near the top.

And, finally, define FILTERS like so:

```
const FILTERS = ['current', 'completed', 'suggested']
```

3: Transitioning our GET Routes to Mongoose

Now we can start using our database in our routes.

So, we have 2 **GET** routes, they will look like this:

Let's re-define the first, easy one `GET /`, and see how the item model works.

`find()` is the method that the model has available.
You can use it like `find()` or with `find({ name: 'movie name' })`, where `{ name: 'movie name' }` is a filter.
What it does, is it sends an HTTP request (which work much like the ones we defined, which is why mongoose is very helpful) to the database system (MongoDB), and retrieves results that MongoDB finds by trying to find items that have matching values (in the first case, no filters; in the second case: look for a field `name` that has a matching value of `'movie name'`)
Note: filter returns a Query (very similar to a Promise object in this regard), but you don't know what the value returned will be. This is why Promises are needed.

recall this line in the db configuration:

```
mongoose.Promise = global.Promise
```

Aside:
What is a Promise?

much like in real-life, a Promise can be broken (rejected) or kept (resolved). Thus, it makes sense that a Promise is a name for a type of object for which you don't know the value of immediately. In fact, you can only know the type if you know that the promise was held, or not.

Why is this useful?

On the web, you are making requests around the world. This could take a lot time, and be potentially very difficult to manage. Imagine every time someone makes a request, the system/server suspends, and just waits for that request. What if that request never finishes? Should it wait forever, maybe like a certain amount of time, or something along these lines? Then if you have many users, this will create a giant queue of requests that your users make. So, one idea that is used commonly today, is notion of running the requests in the background, and when the Promise is ready, only then you can use its value. [Topic of Concurrency]

How to use Promises

```
const reject = function(value) { /* this is also a boring function*/ }

// create a new Promise
const p = new Promise(function(resolve, reject) {
    setTimeout(function() { // to create a sense of time
        if (/* something that expected happens */) {
            resolve(/* some value(s)*/)
        } else {
            // something did not go as expected
            reject(/* some value(s)*/)
        }
    }, 10000) // 10 seconds in milliseconds
})

// wait for the Promise
p.then(function(/*some values(s)*/) { // success case
    /* this is a boring function*/
})
p.catch(function(/*some values(s)*/) { // fail case
    /* this is a boring function*/
})

// can combine the two, like this:
p.then(function(/*some values(s)*/) { // success case
    /* this is a boring function*/
})
.catch(function(/*some values(s)*/) { // fail case
    /* this is a boring function*/
})
```

So, we get our route like this:

```
items.get('/', function (req, res) {
    Item.find().then(function(items) {
        res.send(items)
    })
    .catch(function() {
        res.status(500).send({ // 500: Internal Server Error
            message: 'something went wrong'
        })
    })
})
```

The next **GET** request is getting an item by 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({ // 404: Not Found
                    message: 'no item found'
                })
            }
        })
    } else {
        res.status(400).send({ // 400: Bad Request
            message: 'please provide `id` as a param'
        })
    }
})
```

Note:
The following are exactly equivalent:

```
findById(id)
find({ id: id })
find({ id })
```

Let's also add a new **GET** request that filters based on type and filter.

First, we can make a function that returns an appropriate filter based on the `type` and `filter` provided by the user;

```
const createQuery = 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
        }
    }
}
```

Next, we can define the route like so:

```
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 = createQuery(type, filter)

            Item.find(query).then(function (items) {
                res.send(items)
            })
        } else {
            res.send({
                message: 'please provide a valid `type` and `filter`'
            })
        }
    } else {
        res.send({
            message: 'please provide `type` and `filter` as params'
        })
    }
})
```

Aside:

Let's also send `TYPES` and `FILTERS` as a request, to drive our front-end later on.

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

4: Transitioning our POST Route to Mongoose

in a **POST** request, we create a new Object and send it to the database, so let's do that using the `save()` function that our `Item` model provides.

Much like the `find()` function, except this one returns an actual Promise (not like `find()`, where that returned a Query). So, we know how to handle it.

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

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

        item.save().then(function() {
            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 `title` and `type`'
        })
    }
})
```

5: Transitioning our DELETE Route to Mongoose

Next, we need to transition our DELETE Route.

our `Item` model has a method that deals with this too.

It is called, `findByIdAndDelete()` or `findOneAndDelete()`. This is very similar to `findById` vs `find()`.

There are also functions like `deleteMany` which is more similar to `find()`.

Please lookup the documentation here if you have a different use-case: https://mongoosejs.com/docs/api.html

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

    if (id != null) {
        Item.findByIdAndDelete(id).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'
        })
    }
})
```

6: Transitioning our PATCH Route to Mongoose

Last week it was a challenge to implement a **PATCH** route with the array, if you have completed that. Good job, this part now should be very intuitive for you, if not, don't worry, the complete and final implementation is here.

This time, the function that the `Item` model provides for us, is called `findByIdAndUpdate()` it is similar to `findOneAndUpdate()`, but similarly to DELETE, there are: `updateMany` and `updateOne`

First, let's create a simple function that handles an update query easily.

```
const createUpdateQuery = function(name, type, completed) {
    const updateQuery = {
        completed: completed
    }
    if (name != null) {
        updateQuery.name = name
    }
    if (type != null) {
        updateQuery.type = type
    }
    return updateQuery
}
```

Now, we can make our **PATCH** route

```
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'
        })
    }
})
```

Aside:
`$set` is an operator in mongoose that is used to replace fields that are provided in the object. More concretely, set the document that MongoDB finds to the values that $set provides.

`new: true` is an option that tells MongoDB to please return an updated object, rather than the old version of it. Default is `false`.
I think it makes more sense to return the new object in our case.

This is the end of Back-end for our project

Now, our back end server is fully defined and everything is ready to be used by the front-end part of the project.

But, just to leave this off nicely, please refactor the `createQuery` and `createUpdateQuery` functions into `/utils/routes/items.js` and export them.

Then in the `/routes/items/index.js` please import those two functions.

We do this, since it doesn't make too much sense to have these two functions in the routes, as they do not define any route, and it makes sense to have helper-functions together.

7. Front-End: HTML and CSS basics

Which parts make up front-end in web development?

1. **HTML: Hyper Text Markup Language**

HTML is a markup language that is meant to in a sense describe what to display on a website

2. **CSS: Cascading Style Sheets**

CSS is a stylesheet language that is meant to be used to describe how the wevsite should look like.

3. **JavaScript**

We are already familiar with what it can do on the back-end. However, on the front-end, it is meant to be used to provide user interaction with the web site that you make.

Examples

HTML

```
<html>
    <head>
        <title>Title of our Document</title>
    </head>
    <body>
        <h1>H1 Title element</h1>
        <h2>H2 Title element</h2>
        <h2>H3 Title element</h2>
        <div>Some text in a div element</div>
        <p>Some text in a paragraph element</p>
    </body>
</html>
```

CSS

```
    p {
        background-color: red
    }
    #some-id {
        background-color: blue;
    }
    .a-class {
        background-color: yellow;
    }
```

Aside:
What is the difference between `id` and `class`.

- id's are unique per element, whereas a classes are not. So we can chain classes together

8. Why just HTML and CSS is not good enough, and what is Vue.js

Every webpage is made up using CSS, and HTML. That is not something that can be avoided, nor it should be as the combination of the two is very expressive. However, it would be nice to add some interactability with the html elements, and allow interaction with servers too in a nice manner. Now, if the webpage is simply enough, you can get by using older libraries like jQuery. However, more recently the webpages became more compliated, and are just able to accomplish a lot without the use of a server.

This is where Vue.js comes in.

What is Vue.js

- It is a front-end framework for making user interfaces for the web that works on JavaScript. More precisely on Node.js.

9. Setting up and running Vue.js project using CLI (Command-line interface)

Let's get started with Vue, by first installing it. We can do it by typing in:

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

The command above installs the Command-line Interface that allows us to easily create new projects, and configure them on the fly.

So let's open a new directory, in a separate project, I will call mine `/front-end` and once opened, in the command line type:

```
> vue create front-end
```

And, pick all default settings.

Then, run:

```
> npm run serve
```

to start your front-end application, and you can then navigate to `locahost:8080` to see how your application looks like at default.

10. Installing a CSS library: Bulma

Now, we were introduced to CSS, but this is not a CSS course, and most people use third-party CSS libraries for their projects anyway. So this works out well.

The most commonly used one is: Bootstrap
However, Bulma is a nicer one to use for this course.

we install it by:

`> npm install --save bulma`

and in `main.js` add:

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

Now, let's delete all css parts in:

1. `App.vue`
2. `HelloWorld.vue`

We do this, since we are using our Bulma.

You can see how nice Bulma is if you comment out:
`// import 'bulma' from 'bulma/css/bulma.min/css'`

Then, our page looks very old. So, we uncomment it to bring it back to modern age.

11. Creating our first Vue Component: Hero with Navbar

0. in `App.vue`, please add the following:

```
<style>
  .content-section {
    margin-top: -3.25rem;
    padding-top: 3.25rem;
  }
</style>
```

1. Let's create one of the main components called: `Home.vue` in `/components` folder.

```
<template>
    <div class="content-section">
        <section class="hero is-white is-fullheight-with-navbar">
            <div class="hero-body">
                <div class="container">
                    <h1 class="title
                    has-text-centered
                    is-unselectable">Backlogger</h1>
                    <h2 class="subtitle
                    has-text-centered
                    is-unselectable">keep your backlog organized</h2>
                </div>
            </div>
        </section>
    </div>
</template>
```

In Vue.js, all your HTML code must be in the element called: `<template>`

all your CSS code in the element called:
`<style>`

and, JavaScript in `<script>`.

Actually, the last two are not as suprising as that's where you would usually keep your CSS and JavaScript in normal HTML code.

2. Let's add a header to our page. Create a new file in the `/components` folder, called `Navbar.vue`

```
<template>
    <nav class="navbar is-transparent">
        <div class="navbar-brand">
            <div class="navbar-burger burger">
                <span></span>
                <span></span>
                <span></span>
            </div>
        </div>
        <div class="navbar-menu">
            <div class="navbar-start">
                <a class="navbar-item">Home</a>
                <a class="navbar-item">Dashboard</a>
                <a class="navbar-item">Add an Item</a>
                <a class="navbar-item is-active">Friends</a>
            </div>
        </div>
    </nav>
</template>
```

But, now let's add some interactability.
Start by adding, the second part:

```
<script>
    export default {
        data: function() {
            return {

            }
        }
    }
</script>
```

Aside:
`data` would make sense to be a JavaScript Object conceptually, however, due to another conceptual restriction is why it should be a function that returns a JavaScript Object. The reason for that is to avoid having multiple instances of this component sharing the same data object.

e.g.

```
    <Navbar/>
    <Navbar/>
```

This would have a same `data` object which would have changes reflect in both instances, even if another class should not do that change.
Thus, the use-function solution makes all our instances of a same component exist independently of each other.

```
<template>
    <nav class="navbar is-transparent">
        <div class="navbar-brand">
            <div class="navbar-burger burger"
                :class="{'is-active':activeNavbar}"
                @click="toggleStatus">
                <span></span>
                <span></span>
                <span></span>
            </div>
        </div>
        <div class="navbar-menu"
            :class="{'is-active':activeNavbar}"
            @click="toggleStatus">
            <div class="navbar-start has-dropdown">
                <a class="navbar-item is-unselectable"
                   :class="{'is-active': itemIndex == 0}"
                    @click="toggleActiveItem(0)">Home</a>
                <a class="navbar-item is-unselectable"
                   :class="{'is-active': itemIndex == 1}"
                    @click="toggleActiveItem(1)">Dashboard</a>
                <a class="navbar-item is-unselectable"
                   :class="{'is-active': itemIndex == 2}"
                    @click="toggleActiveItem(2)">Add an Item</a>
                <a class="navbar-item is-unselectable"
                   :class="{'is-active': itemIndex == 3}"
                    @click="toggleActiveItem(3)">Friends</a>
            </div>
        </div>
    </nav>
</template>
<script>
export default {
    data: function() {
        return {
            activeNavbar: false,
            itemIndex: 0
        }
    },
    methods: {
        toggleStatus: function() {
            this.activeNavbar = !this.activeNavbar
        },
        toggleActiveItem: function(index) {
            this.itemIndex = index
        }
    }
}
</script>
```

Lastly, the action of giving indices seems very repetitive. What if we add a for-loop?

We do so like so:

```
<template>
    <nav class="navbar is-fixed-top is-transparent">
        <div class="navbar-brand">
            <div class="navbar-burger burger"
                :class="{'is-active':activeNavbar}"
                @click="toggleStatus">
                <span></span>
                <span></span>
                <span></span>
            </div>
        </div>
        <div class="navbar-menu" :class="{'is-active':activeNavbar}"
            @click="toggleStatus">
            <div class="navbar-start has-dropdown">
                <a v-for="(option, index) in OPTIONS" :key="index"
                    class="navbar-item is-unselectable"
                    :class="{'is-active': optionIndex == index}"
                    @click="toggleActiveOption(index)"
                >{{ option }}</a>
            </div>
        </div>
    </nav>
</template>
<script>
export default {
    data: function() {
        return {
            OPTIONS: ['Home', 'Dashboard', 'Add an Item', 'Friends'],
            activeNavbar: false,
            optionIndex: 0
        }
    },
    methods: {
        toggleStatus: function() {
            this.activeNavbar = !this.activeNavbar
        },
        toggleActiveOption: function(index) {
            this.optionIndex = index
        }
    }
}
</script>
```