Week 3: REST API Continued, and MongoDB

This week, we will need to finish up parts from last week. If you already went over last week notes online, you can skip the next 3 sections.

Recall, last week we initialized the back-end of our application, we learned about REST, made our first HTTP routes, and tested that they work as required.

Now, we will continue from last week onto:

1: Implementing POST Item route

Now, let's add some values to our `items` array.

Recall that we can use the `push` function to add an element to the array. So let's define what the item object would look like first:

```
const item = {
    title: 'Some Movie Title',
    completed: true, // or false?
    type: 'Movie',
    createdAt: Date.now(), // gets the time of this instant
    id: 1 // some unique way to identify an object
}
```

and then we can just use `items.push(item)` , thus adding the item to our list.
But, we are currently hard-coding the title, and type , so let's give the ability to specify the properties: 'title', and 'type' to the user (we will handle 'id' and 'createdAt' ourselves)

To do that we need to be able to understand, and use the data that the user sends us.

Previous route we made was a **GET** route. Now, by design, a **GET** request cannot accept packaged data from the user, which is usually referred to as a *payload*.

That is in fact, the main difference between a **POST** request and a **GET** methods. This should make sense, since if you recall - **POST** is meant to create a new resource, and the way it will do

that is by using user's input.

We will force the client to send us **JSON** and just not understand non-JSON data, first, to gain some consistency.

To do that, at the top of the file (after definition of `app`), please add:

`app.use(express.json())`

This is called **middleware**, which is named like that because it is a function that gets executed after express receives a request from the client, then **middleware** gets executed and only then the callback function of your route will get executed.

What `express.json()` does is it parses the body of the user's request to json, which will be stored in `req.body`) as a JavaScript object and easily usable by us.

Note: `req.body` will have the data that we will send to the server from Postman.

So, we can now define the route as follows:

```
let currId = 1

app.post('/items', function(req, res) {
    const item = {
        title: req.body.title,
        type: req.body.type,
        completed: false, //since you why would you add something to watch
                          // that you already watched
        createdAt: new Date(),
        id: currId
    }
    items.push(item)
    res.json(item)
    currId = currId + 1 // update our id to keep it unique
})
```

We can stop our server by pressing CTRL C in **Terminal**, and restarting the server with `node index.js`.

Now we can **POST** new items, and **GET** our existing items.

2: Implementing GET of certain item

So, now a task we would like to be able to do is to have the ability get a certain item, rather than the entire list of items. To do this, we will make a route that queries for an id of an item, and returns a valid item if it finds it.

But, how do we pass the id to the server, since there is no body in a GET request?

We can pass the id in the URL, e.g. `items/314` where 314 is the id of the item, then we use that *parameter* (`req.params` is where you can access all the parameters passed in the request) to query the items.

Note: in express we need to specify that 314 is a parameter, otherwise it will just think it's a normal url. To do that we make specify id as: `:id` , to get the full URL as: `items/:id` . Then this will make a variable `req.params.id` be available in the callback function.

We then can implement our function like so:

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

    // search for the id
    for (let i = 0; i < items.length; i++) {
        const item = items[i]
        if (item.id == id) {
            res.json(items[i])
            return // important to use return so that the code under
                   // doesn't get executed
        }
    }
    res.status(404).json({}) // return nothing with 404 status
})
```

What the function does:
It looks at every item that we currently have in `items` , and tries to find one with the id we have provided as a parameter. If we do find one, we want to send the function, and terminate our search and the function since the id is unique, so you won't find a copy.

Note: we can specify that status of our response as 404, if we don't find anything.

3: Implementing DELETE Item route

Now we would like to delete an item that is in the list, because we don't want it to be there for some reason.

Aside: To remove an element in the array in JavaScript: `array.splice(indexOfElement, 1)`

So, similarly, to **GET** item route, we would like to specify the id of the item we would like to **DELETE**.

[DELETE method does have a request body, but we will be following best practices, as it doesn't make too much sense to send a body for a DELETE request. Although there might be use-cases where you would need to send a payload, as opposed to a parameter, so nobody really stops you.]

Here is the code that does what we would like to accomplish:

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

    for (let i = 0; i < items.length; i++) {
        const item = items[i]
        if (item.id == id) {
            res.json(items.splice(i, 1))
            return
        }
    }
    res.status(404).json({})
})
```

4: Implementing PATCH Item route

Patching an item requires you to modify an existing item. For this, you will be implementing your own route.

Hint, this is very similar to a POST request, except you are not creating a new item, but modifying an existing one. So, try using the things we learned while implementing the other routes, as it will require you to use a mixture of code from previous routes.

Note:

Your patch route should be able to:

- find an item by id
- be able to modify the following properties: 'type', 'completed', and 'name' of the existing item
- should return an updated item, if you find one with the right id.
  if you can't, return an empty object with status **404**.

We will go over the solution next week, as this is not as crucial yet this week.

5. Databases

As we have seen that our data is stored in the array, and thus every time we restart the server, it expectedly so clears the array.

This is very obviously can be problematic for an application that is expected to live for longer than a few minutes.

So to circumvent this issue, we will require a database to store our information in a safe, and a reliable manner.

The database we will be using is MongoDB with Mongoose library which will help us model our database, by adding field constraints, data types, and provide us very intuitive APIs to use.
If you are interested in some technicalities, this is considered a NoSQL database, which is just opposite to the usual SQL database.

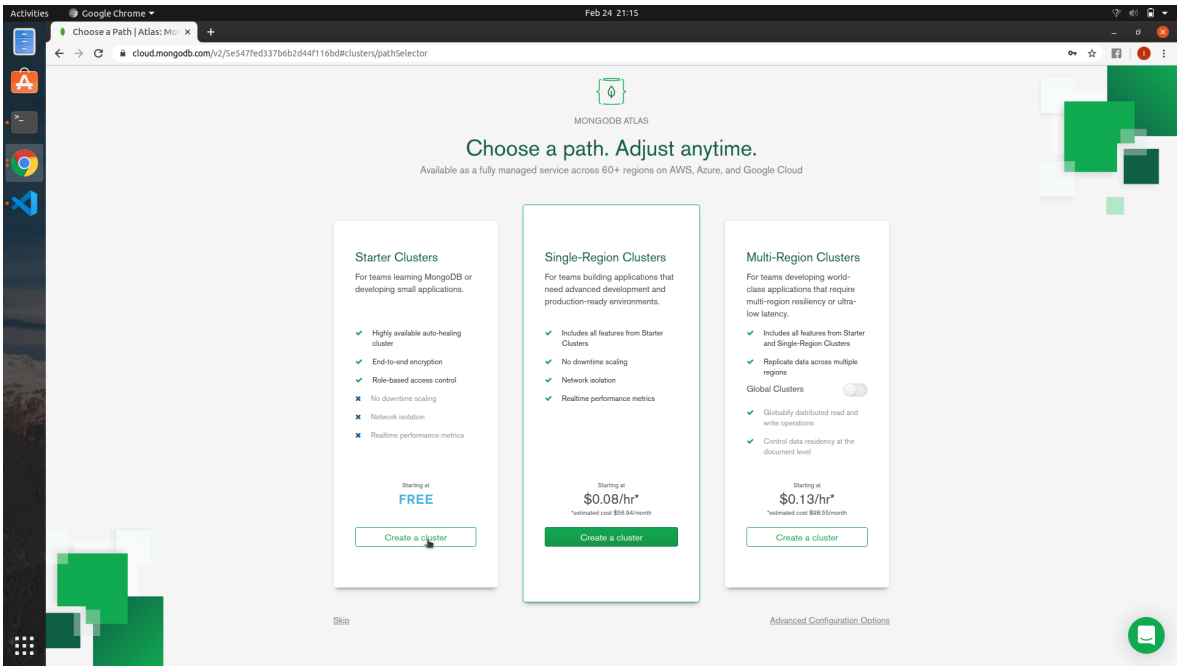
The key difference is just how the data is stored, but for our purposes, it doesn't matter too much which to use. MongoDB is just more beginner friendly.

6. Setting up MongoDB and organizing our project

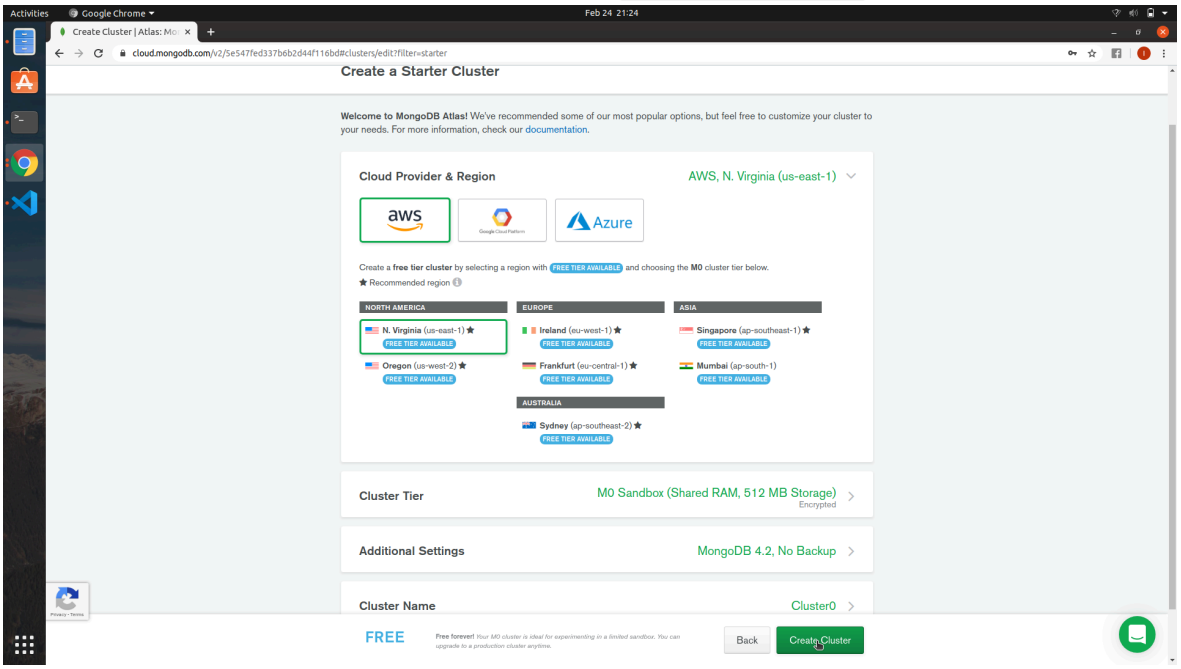
Before, we can transition from using `items` array to an actual database we need to set up one. We will setup one on the cloud using MongoDB Atlas.

Aside: MongoDB Atlas is a cloud database service that is made by MongoDB, and it makes database management much easier.

1. Please navigate to: `https://www.mongodb.com/cloud/atlas` and Sign Up
2. Then, create a free cluster:



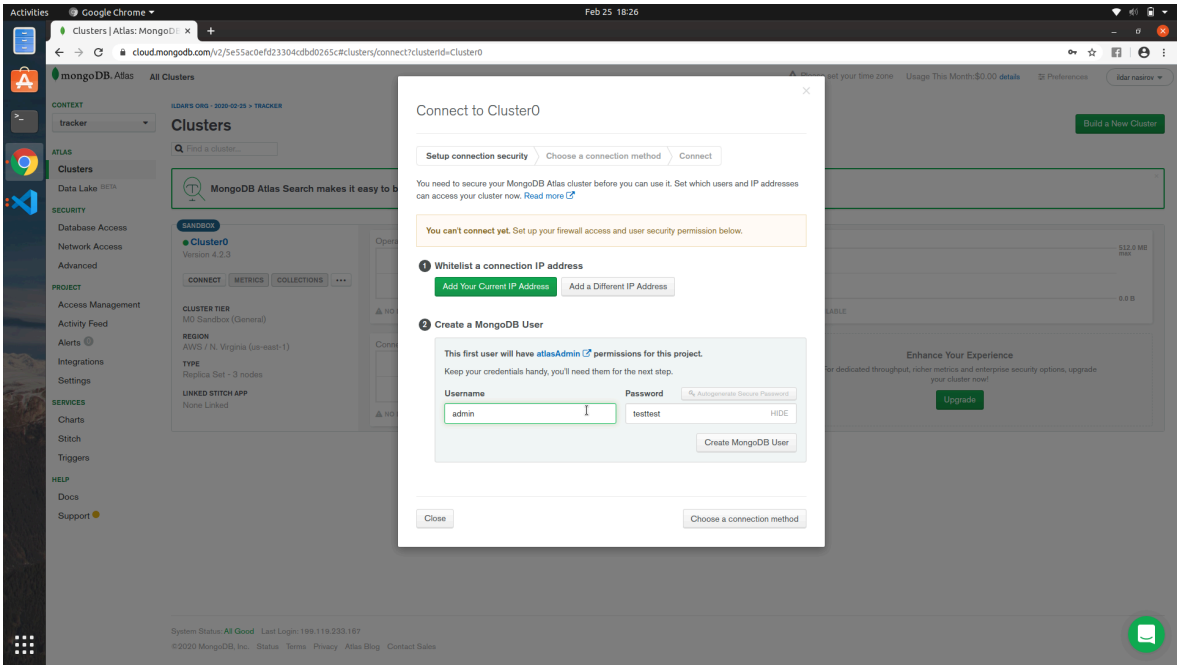
3. Let's user the default settings, so just press: `Create Cluster`.



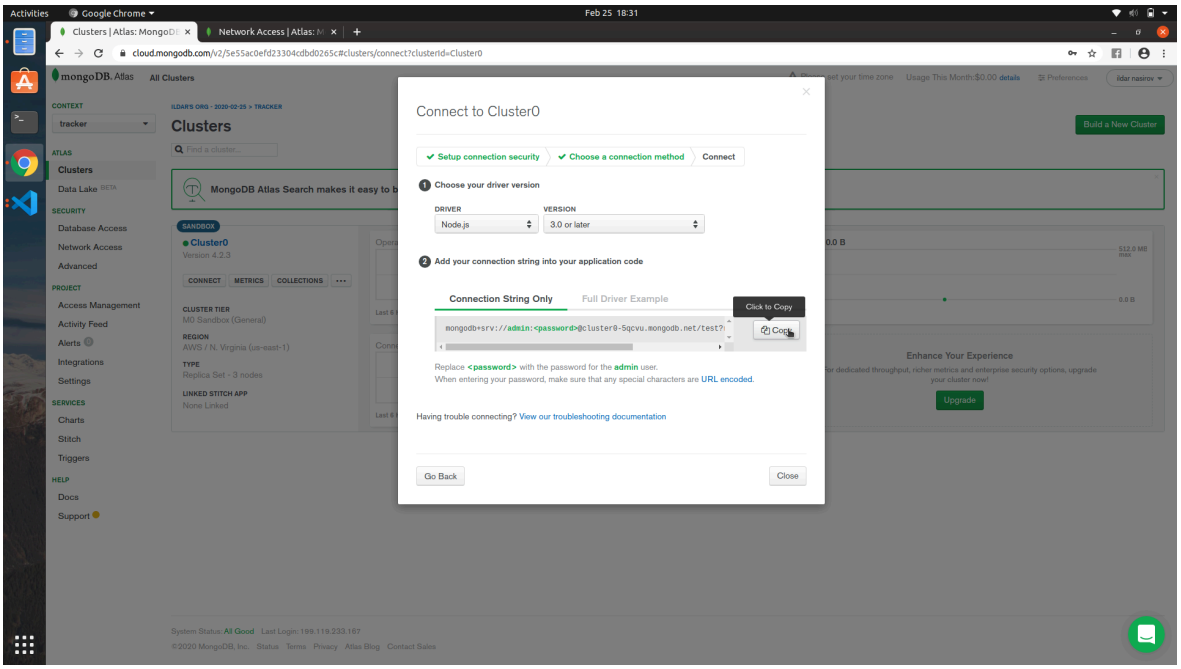
4. Now, the server with our database is being setup, and configured by itself, on Atlas. Wait, until it's ready.
   (If it's taking too long, refresh the page)
5. Press Connect, and there under step 1, please press Whitelist your Current IP

Address. (If you would like connection from anywhere in the world to you database, there will show up a link that will let you do that.)

6. We need to set up a database user: we do it by clicking at the same place as where the screenshot is pointing to. You can make up any user, just remember your credential details as they are needed to communicate with your database. I'll use name: `admin`, and password `testtest`. Please do not use your password that you use at other places as you will be writing out your password in plain text in the later steps.



7. Press on 'Choose a connection method' -> 'Connect your application' and then copy the text string that you see.



8. In the project, create a file called `config.json`
   fill it like so:

config.json

```
{
    "MONGO_DB" : "your string"
}
```

and change `<password>` to the password you used in creation of the user.

Aside: if you are using Github and want to publish your code, please create a file called: `config.json.template` with empty value for `MONGO_DB` and add `config.json` to `.gitignore`.

9. Now we need to setup our local environment to be able to communicate with our database in the cloud. We do so by writing `npm install --save mongoose` in the terminal in the working directory.
10. Let's refactor our code a bit first. Create a directory called `db` and in it a file called `mongoose.js` and copy and paste the following code into it:

```
const mongoose = require('mongoose')

mongoose.Promise = global.Promise
mongoose.set('useUnifiedTopology', true)
mongoose.set('useCreateIndex', true)
mongoose.set('useFindAndModify', false)
mongoose.connect(process.env.MONGO_DB, {
    useNewUrlParser: true
})

module.exports = mongoose
```

What this does, is just setups up mongoose to use:
1) Promises (we'll talk about it later)

2) Some more recent requirements for communication which are set with `set`, and

3) Establishes an actual connection to the database url which is defined in `process.env.MONGO_DB`

Now, our MongoDB server should be ready to use.
So let's first organize our project so that it would be a bit easier to use.

1. Then create a directory `config` where we can put our `config.json` and `index.js` file in there.
2. Now, let's make a directory `/routes`, and another directory in `/routes` called `/items`. There (in `/items`) we can make a file called `index.js` and there please put:

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

and then paste your item routes into here, replacing every instance of `app` with `items`, and add the following to the end:

```
    module.exports = items
```

next please make a file called api in `/routes` and paste the following:

```
const express = require('express')
const { items } = require('./items')

const api = express.Router()
api.use('/items', items)

api.get('/ping', function(req, res) {
    res.json({
        message: 'pong'
    })
})

module.exports = api
```

3. Add the following code to `/config/index.js`

```
const express = require('express')

// config the express server
const app = express()
const port = process.env.PORT || 3000

if (!process.env.NODE_ENV) {
    const config = require('./config.json')
    process.env['MONGO_DB'] = config.MONGO_DB
}

app.use(express.json())

const api = require('../routes/api')

app.use('/api', api)
app.listen(port, function() {
    console.log(`app listening on port ${port}`)
})
```

4. Finally we just create `index.js` in the work repository, with the following code:

```
require('./config')
```

5. Now, running `node index.js` should work, but our code is much more organized, and well refactored.

7: Defining our first Mongoose Schema

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

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

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
    }
})

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

8: Transitioning our GET Routes to Mongoose

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

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

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: 'no item found'
                })
            }
        })
    } else {
        res.send({
            message: 'please provide `id` as a param'
        })
    }
})
```

Note the new `.then` function. This is function that is used on Promises objects (Mongoose returns a Promise, when you make a Query), which are asynchronous objects. This simply means that at the value of the object being called, is not known.

9: Transitioning our POST Route to Mongoose

```
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: name,
            type: type,
            completed: false,
            addedAt: Date.now()
        })

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

10: Transitioning our DELETE Route to Mongoose

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

11: New Get Route for filtering

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

12: Small Note

In JavaScript, these are equivalent:

```
const id = req.params.id
const { id } = req.params
```

So I will replace my code to the second option, as it's just easier.