Mongo/Mongoose

Pathway

- DBs in general
- SQL v NoSQL
- MongoDB
- Mongo into the cloud
- Mongoose, and connecting to the cloud
- Mongoose models
- Validations
- Seeding
- HTTP Client

DBs

In the last lesson we started setting up some basic structures in our back end that we used like data stores. This was handy for the time being, but we are starting to see their limitations. Among those limitations are that they can only be read into our back-end as a whole structure, and that any changes don't persist if we restart our server (unless we were to write them to a file, or ammend that file). It's all a bit clumsy, and we can improve on this.

That improvement will be a database. In short, a database is just a place to store data, an *organised collection* of data. They are set up to be able to make rapid queries on large amounts of data.

Relational v NoSQL

There are two main forms of DB. The more 'classical' style of DB is the relational DB. This is the type of DB that consists of tables. A table will have a number of columns and rows, and will resemble a spreadsheet. You can relate one table to another, or to several others, and build up your DB using these tables.

SQL (Structured Query Language) is the language used to communicate with the DB in this instance. There are several types of SQL DB, including Oracle, POSTGres, and so on, each that uses SQL, usually with much the same commands, and some extended commands that are unique to this or that system.

The other type of DB that is popular currently is a NoSQL DB, which, as you can imagine, does not use SQL as a way to communicate with the DB. NoSQL databases are also known as document databases, and they have a very different way of storing data. There are a variety of NoSQL databases, but the common element is that they do not store their data in tables.

Rather than looking at all the possibilities, let's talk about MongoDB.

MongoDB

MongoDB is a NoSQL document database that stores its data in structures that resemble JSON objects. We have seen JSON objects several times now. This is great, as we now work in JS on the front end, Node (JS), on the backend, and have a DB that resembles JS objects too. The whole ecosystem that we are working in features JS like structures.

This structure means that you have some more options in how you structure your data than with the tabular data of relational databases. You have the option to nest data, much like you might have an object that has an array within it. There is also no limit to the nesting that you can do - it will depend on how you want to represent the data, and what data will need to be retrieved quickly.

There are two main structures within a Mongo database.

- Collections
- Documents

Collections are designed to hold homogenous data, a bit like an array. If you like, you can think of collections as arrays of documents. *Documents* are very much like the individual objects within an array.

A strucuture like the following (in JS):

```
const albums = [
    {
        artist: "Horace Silver",
        title: "Song for My Father",
        year: 1964,
        genre: "Jazz"
    },
    {
        artist: "Charli XCX",
        title: "How I'm Feeling Now",
        year: 2020,
        genre: "Pop"
    },
    {
        artist: "Charli XCX",
        title: "Charli",
        year: 2019,
        genre: "Pop"
    },
    {
        artist: "Morrissey",
        title: "Vauxhall and I",
        year: 1994,
        genre: "Rock"
    }
]
```

would in MongoDB usually be represented as a collection of album documents. You do *not* have to store this structure in MongoDB in this way, but this would be an obvious way to do this, and is an example to show the similarities.

You will see collections more as we get set up, so let's do that first.

Mongo Cloud Setup

Follow the instructions in this guide to get Mongo setup in the cloud.
https://medium.com/@colinbaird_51123/getting-started-with-mongodb-atlas-2b996d5be099

By signing up to the cloud, we are outsourcing some of the pain. We don't need to get the OS to play nice, and we are also going to skip the step of the MongoDB queries, and instead use Mongoose to connect to our DB, and to run the queries. Mongoose is a lovely wrapper for MongoDB native queries.

The important steps there are:

- **Sign up to Mongo Atlas**. This is a MongoDB cloud service. It's very much worth noting here that you can run MongoDB locally, and I was setting up to do that. However I made a late change, because I have found in the installation can be brutal in a big class with several OSs. If you would like to have a go though, it's worthwhile having a play with. You can also have a go at the MongoDB queries. For now we are going to be in the cloud.
- You **chose the free tier**, and a **provider**, and create a **cluster**. A cluster is just the term for several MongoDB servers working together. The choice of location should be closest to where you are, but it won't be a big deal.
- You are welcome to follow the rest of the guide, add the sample dataset, and also get into the data visualisations. We are going to concentrate on other things.

The next step will be to make a connection to the DB:

- Hit the connect button in your cluster.
- You need to whitelist your IP (or all IPs)
- You need to create a database user, including a password.
- Choose connect your application.
- Choose your driver and version (Node > 3.6)
- Copy and paste your connection string somewhere to be used shortly, eg, mine looks like this: `mongodb+srv://matt:<password>@cluster0.pxhul.gcp.mongodb.net/<dbname>?retryWrites=true&w=majority`
- Replace `<password>` with the password for the matt user. Replace `<dbname>` with the name of the database that connections will use by default. My `<dbname>` will probably end up being 'record-store'.

Now we are set up with MongoDB in the cloud, and we have a connection string to get access. Next we need a way to make queries on that database. We will be using Mongoose for this.

Mongoose

Mongoose is what is known in the dev world as a model. I have found the idea of a model difficult for people in the past, but essentially you can think of a model as a tool to talk to the database in the language that you are happy working in. In this case we are working in JS, and we would love to be able to make requests of our Mongo database, and do it in JS. Mongoose is a package that wraps around the Mongo shell, connects to the database, lets us performs queries on the database, and sets up the structure of our data.

Let's first make sure we can connect to the database, and then have a look at schemas. Then we can start adding some data to the database.

Connecting Mongoose to the DB

First `> yarn add mongoose`, and then right towards the top of your `server.js` (the Express one):

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

mongoose.connect('mongodb+srv://<username>:
<password>@cluster0.pxhul.gcp.mongodb.net/<dbname>?
retryWrites=true&w=majority', { useNewUrlParser: true, useUnifiedTopology:
true },
    function(err, database) {
        if (err) {
          throw err;
        }
        console.log("Connection made to database.")
    }
)
```

We are requiring in the package, and then using Mongoose to connect to our DB. *Make sure you put your username and password (the DB username) into those fields, and also give a name to your database (<dbname>)*. We aren't going to worry about the config object sent to the connect function for now.

If everything goes to plan, you should see the server running happily, and that success string being logged out ("Connection made to database.").

Now we want to have a look at the shape of the data (documents) that we are going to place in the collections.

Why have models

Our schemas are the shape of the documents that are being loaded into the database. Let's say that our DB is going to contain albums, so that a user can upload a list of their favourite albums. An album is has some features that we would like to capture. Most obviously an album has a title. We can imagine that this title is a string. So let's start creating this album, and think through how it should look.

We can do this in JS, and start like this:

```
album1 = {
    title: "Back in Black",
    artist: "AC/DC",
    year: 1980,
    genre: "rock"
}
```

This is all well and good as a JS object, but we are going to import Mongoose to help us to formalise this as the shape of our data that will be going into the database.

This is the code we need to start making a schema for our albums, and our model (all the functionality that is required to talk to the database) will be created along with this structure:

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

```
const AlbumSchema = new mongoose.Schema({
    title: {
        type: String,
        required: true
    },
    artist: {
        type: String,
        required: true
    },
    year: {
        type: Number
    },
    genre: {
        type: String
    }
})

const Album = mongoose.model("Album", AlbumSchema);
module.exports = Album;
```

We are going to make several of these over the course of the project, and so we are going to create a new directory to house them `./models`. Also note what is taking place in this file:

- We are requiring in the package
- We are getting the Schema part of the Mongoose package
- We are defining an album in detail, and specifying its parts
- We are creating the model from this schema
- We are exporting it out

Validations

Validations are the fine tuning of the models. This is the way that we define the specifics of how we need the data to look. For example, we can say whether a particular field is required (in our case an album is not really complete until it at least has a title and artist), or even something more specific like the format of an email.

For now we are keeping it simple, and you can see that we are specifying that only the 'title' and 'artist' fields of the schema are required. There are many many ways that you can validate the data, providing some amount of checking on the fields as they are entering into the database. We can't cover all of those here, and we will discover more as they come up.

Schema Challenges

1. Create three models based on some data that you are interested in, or that you might end up using in your project. If you are stuck for ideas you can use albums, artists, and users (the users who will use your app).
2. Perform three different types of validations in your schemas. Try to think through realistic validations that might actually be useful.

Adding some data to the DB

So we are now set to fire it all up. We can now start making queries to our DB. We have made a connection to the database, and we have set up a model for our Albums (meaning we have a schema for the data, and we passed that to Mongoose to make the model - the thing that will talk to the DB for us). We are going to do this a slightly idiocyncratic way to use our routes and reinforce the concepts that we have been learning already.

Let's get to the code:

```
// Import our model
const Album = require('./models/Album.js')

// ...

// Add a route to put this one hard coded album into the DB.
// This won't be typical, but to demonstrate for now and use a route, we
are hardcoding in an album
app.get('/add-one-album', function(req, res) {
    // Using the model to set up an Album Mongoose object ready to go
    albumToAdd = new Album({
        title: "Highway to Hell",
        artist: "ACDC",
        year: 1980,
        genre: "rock"
    })
    // Using Mongoose to save the album to the DB.
    albumToAdd.save()
        // The success path, and the album object will be presented to the
callback
        .then(function(album) {
            console.log("ITEM SAVED!")
            console.log(album)
            // Sending back the album to the front end (which we might do
in our app too)
            res.send(album)
        })
        // The fail path
        .catch(function(err) {
            console.log(err)
            res.send(err)
        })
})
```

Here a lot of what we have done is coming together. We have created a route and are going to use that to trigger a hit to the DB. It's a little unconventional, but helps to double down on what we have been doing with routes, and also will load an album into the DB.

Go to the browser and hit the route `http://localhost:3000/add-one-album`. Check that you get all the appropriate feedback in the server logs, and that the album is sent back to the front end. If all of that goes well you can have a look in your collection in Mongo Atlas. You should also see the album, now a document sitting in a collection (albums), within a database (recordStore) in our cluster on Atlas. Celebrate a little that you made it all work!

Seeding

I'm going to keep seeding (priming the DB with some data) the database using the same method, and passing the array of albums that we saw earlier (which comes in through `data.albums`):

```
// A route to load several albums
app.get('/add-several-albums', function(req, res) {
    // Mongoose's bulk insert option
    // We are passing the array of albums through to the database.
    Album.insertMany(data.albums)
        .then(function(albums) {
            console.log(albums)
            res.send(albums)
        })
        .catch(function(err) {
            console.log(err)
            res.send(err)
        });
})
```

Seeding can be a somewhat tricky thing, and we can cover some of those details later, but for now we want to have a database with some data in it, and we also got to practice making routes and hitting those routes. If for some reason we need to start again we can just seed the database with the data in that file fairly simply.

We won't need these routes anymore (or not for a while), so we can comment them out for now (or just ignore them). What we will need are some of the classic CRUD routes that we need for our API to reach its full potential. We will start with the R of the CRUD set.

This is the route:

```
app.get('/albums', function(req, res) {
    // The argument to find is an empty object, telling the model to not
restrict the search at all.
    // We can refine this search using this parameter.
    Album.find({})
        .then(function(albums) {
            console.log(albums)
            res.send(albums)
        })
        .catch(function(err) {
            console.log(err)
            res.send(err)
        })
});
```

Here we have set up the classic route to retrieve all the albums from the albums collections in the recordStore DB.

Suggested challenges

1. Use the other models you have created, and seed the DB with three documents of each type.
2. Create a route similar to the above to pull all of the data in those collections from the database.

HTTP REST Client

Install the HTTP Rest Client from the extensions in VSCode. At the root of your express project, make a file called `albums.http`. This is a fantastic extension that will enable us to test our API does what we would like it to do, and keep a record of our progress. We are going to ignore our progress on the front end for now, and just have our API performing all the CRUD operations using RESTful routes. What that means should become apparent as we get started, and the challenge will be to finish this.

Add the following to the `albums.http` file:

```
GET http://localhost:3000/albums HTTP/1.1

###

POST http://localhost:3000/albums HTTP/1.1
content-type: application/json

{
    "title": "Fear Innoculum",
    "artist": "Tool",
    "year": 2020,
    "genre": "Rock"
}

###
```

This quite magical file can send HTTP requests, and so we can use it to test our routes. Here we are just testing two routes, the one that reads all the data from the collection, and one to add an album to the collection. You will see the 'Send Request' appear above requests that are in the correct format. The `###` is required to separate the requests. Also note the mandatory blank line between the metadata and the body in the POST request. If you keep hitting this route you will keep adding this album to the database. Check what's happening by also trying out the GET request after adding the album.

To keep up with what we are doing, we need to modify the route to add an album in the following way:

```
app.post('/albums', function(req, res) {
    console.log("ALBUM POST route hit")
    console.log(req.body)

    albumObject = {
        title: req.body.title,
        artist: req.body.artist,
        year: req.body.year,
        genre: req.body.genre
```

```
    }

    albumToAdd = new Album(albumObject)

    albumToAdd.save()
        .then(function(album) {
            console.log("ITEM SAVED!")
            console.log(album)
            res.send(album)
        })
        .catch(function(err) {
            console.log(err)
        })
})
```

Now our route can take in a JSON object that is sent through the body of the request, and save that to the database.

API Challenge

1. Start building out your API to be able to be able to find one record, and edit or delete a record. To do this you are going to use the id (`_id`) of the record. You will need to check through the docs relating to Mongoose to work this out, and create a few more routes. It can be done in various ways, and we will have a chat about these in the future. Have a go first, and see what you can discover. You will also need to ammend the `.http` file to run those requests.
2. If you get through those, you could build out your api to also start being able to cope with the full range of routes for your other models.

Express params

Now we are really starting to build out the routing for our API, and when we do this we are going to attempt to conform to the idea of RESTful routes. This means that we are going to follow convention with our routing so that we expect the HTTP method incoming to be matched by the appropriate route on the back end.

This article explains it all much better than I ever could:

`https://medium.com/@atingenkay/restful-routes-what-are-they-8fe221521bb`

What we are going to do is that, rather than you all learning and memorising that table of options around this convention, we are going to build things out in a restful manner, and learn by doing. Then we can revisit the theory and convention over time with a better and more real understanding.

One of the conventions that you can see there is to have a route that looks something like the following:

```
app.delete('/albums/:id', function(req, res) {
    console.log("DELETE ROUTE hit")
    console.log(req.params.id)
    Album.findByIdAndDelete(req.params.id)
        .then(function(x) {
            console.log(x)
```

```
            res.send(x)
        })
        .catch(function(err) {
            console.log(err)
            res.send(err)
        })
})
```

This is our delete route and will work. Let's dig in to why it works, and then talk about how we will test this route.

Really, the main thing that you can see here that is new is in the path of the route: `'/albums/:id'`. What is this `:id` business?

What's happening here is that we are creating a variable as part of our path here. The colon signifies to Express that we want to set the part of the path in that position to be equal to `id`. We are setting it to `id` because it's going to be the idea of the album (resource) that we want to delete. We could have written this: `'/albums/:turnip'`, and then whatever came through the path at that point will be the value of `turnip`.

Here is an example of part of the output when this route is hit with an appropriate id:

```
DELETE ROUTE hit
5f8f85efe1e990f36ceefce3
```

You can see there that we are logging out information that the delete route was hit, and also the value of `req.params.id`. Express is helping us out here, and setting an atribute on the `req.params` object, with the key `id`, and the value being set to whatever it is that comes in that place in the string.

This might make more sense when we hit that route, so here is how we are going that in our HTTP client:

```
DELETE http://localhost:3000/albums/5f8f85efe1e990f36ceefce3 HTTP/1.1

###
```

You can see here that we are making a DELETE request to our API. We are sending the path as `/albums/5f8f85efe1e990f36ceefce3`, and because it's a delete request and the path is right for the circumstances, it is matched on our backend by our `app.delete('/albums/:id', ...` route. Because we have set up this route with the '*wildcard*' `:id`, Express will accept whatever string comes after the `/albums/` and store it in the params object, with the key of `id`. This is why `req.params.id` logs out `5f8f85efe1e990f36ceefce3` when we hit this route. It's a bit like making a variable part of the string, and that Express knows how to cope with that.

This all takes a bit of getting used to, and again, it's one of those things that is much better off tinkered with and used, and then the explanation will make more sense. It is likely to be very confusing at first.

You might have many questions, such as how would you send this through using Axios when the time comes. We will get to how to integrate this more fully shortly, but let's assume for now you have in a variable the id of

the album that you want to delete. Your Axios delete request will look something like this:

```
axios.delete(`/albums/${id}`)
    .then(...
    // ...
```