Databases and Persistence

Up to now we've been either getting data from hard coded data in a file or from another service via network calls. This is fine, but storing our own data can be helpful! This is where databases come in. There are a ton of options for storing data not only in types of databases (such as document stores versus relational databases), but also database services. For the sake of easiness (and something new), we're going to use MongoDB, a document store!

MongoDB

Mongo is a document based database that is extremely easy to use! We're going to download and work with a local instance, but if you are interested, it is possible to set up a cloud based mongo instance via their website! In order to install mongo, follow the installation guide in their documentation.

If you are running a windows machine the below steps may be necessary (Thank you Duc):

1. Once you install your mongodb, open Control Panel >
2. Make sure you in View By: Category >
3. Click on [System and Security] >
4. Click on [System] >
5. On left side, click on [Advanced system settings] >
6. On the bottom, click on [Environment Variables...] >
7. Look for PATH variable and if you don't have one, just create one >
8. EDIT > add this "C:\Program Files\MongoDB\Server\4.2\bin"
9. Open your favorite terminal and type "mongo --version" >
10. If something show up, then you are done)

Once you've installed mongo, go to your terminal and type `mongo --version`, you should see something along the lines of:

```
➜  ~ mongo --version
MongoDB shell version v4.2.5
git version: 2261279b51ea13df08ae708ff278f0679c59dc32
allocator: system
modules: none
build environment:
    distarch: x86_64
    target_arch: x86_64
```

If you type `mongo` you'll wind up opening a command line interface with mongo. Before we do that, however, let's download some data first.

Inserting Datasets

First, we'll need to download a dataset to our machine. We're going to be using a [pokemon dataset from github](). Navigate to a directory in your machine and type:

```
git clone https://github.com/ATL-WDI-Exercises/mongo-pokemon.git
cd mongo-pokemon
```

Once inside the mongo-pokemon directory, take a quick look at the `seed.json` file. This is the data we'll be loading into our database. It's nothing but JSON, and, unsurprisingly, that's how the data will be worked with (i.e. just like how we work with JSON objects).

To load our data, type:

```
mongoimport -d pokemon -c pokemons --jsonArray < seed.json
```

What we're doing above is importing the seed array (as a json array) into the collection `pokemons` inside of the database `pokemon`.

Viewing the Data

Inserting data is fine and dandy, but actually viewing it within your database is also nice! First, we'll need to open mongo via our command line, so go to your command line and type:

```
mongo
```

This will open a mongo session. When you type that, it ought to have an output that looks something along the lines of:

```
(base) ➜  mongo
MongoDB shell version v4.2.5
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("ff306c58-6817-4c8a-a39c-b5c6f2ad46d1") }
MongoDB server version: 4.2.5
>
```

You may also have some warnings about deprecations and possibly a message or two about getting a cloud instance. You can ignore those for now! One thing that we'll need to note is `connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb` What we really care about is knowing that initial URL, which we'll need later. For now, just remember, our base url where our database lives is:

```
mongodb://127.0.0.1:27017
```

In order to access our database, we'll need to choose it! But first, type `help` into your mongo repl:

```
> help
  db.help()                    help on db methods
  db.mycoll.help()             help on collection methods
  sh.help()                    sharding helpers
  rs.help()                    replica set helpers
  help admin                   administrative help
  help connect                 connecting to a db help
  help keys                    key shortcuts
  help misc                    misc things to know
  help mr                      mapreduce

  show dbs                     show database names
  show collections             show collections in current database
  show users                   show users in current database
  show profile                 show most recent system.profile entries with time >= 1ms
  show logs                    show the accessible logger names
  show log [name]              prints out the last segment of log in memory, 'global' is default
  use <db_name>                set current database
  db.foo.find()                list objects in collection foo
  db.foo.find( { a : 1 } )     list objects in foo where a == 1
  it                           result of the last line evaluated; use to further iterate
  DBQuery.shellBatchSize = x   set default number of items to display on shell
  exit                         quit the mongo shell
>

```

Here we get a list of options to work with! Now let's use some of these to find our database. In order to do that, type in `show dbs`. This will display all of the database names that we have in our instance:

```
> show dbs
admin    0.000GB
config   0.000GB
local    0.000GB
pokemon  0.000GB
>
```

We have our newly added pokemon database at the very bottom! Let's use that, and then look at the collections inside of it:

```
> use pokemon
switched to db pokemon
> show collections
pokemons
>
```

In order to access the data, we'll use the syntax `db.<collection_name>.<function>`. To retrieve all of the data, you pass nothing into find, so your query looks like:

```
db.pokemons.find()
```

This will grab everything, but luckily will ask us if we want to continue printing the data (databases can get very big).

To find a specific pokemon, we search on any key within the dataset! Let's search for Charizard:

```
> db.pokemons.find({name: "Charizard"})
{ "_id" : ObjectId("5e93c8708a0be3ec97f570fc"), "id" : "006", "name" :
"Charizard", "img" : "http://img.pokemondb.net/artwork/charizard.jpg", "type"
: [ "Fire", "Flying" ], "stats" : { "hp" : "78", "attack" : "84", "defense" :
"78", "spattack" : "109", "spdefense" : "85", "speed" : "100" }, "moves" : {
"level" : [ { "learnedat" : "", "name" : "dragon claw", "gen" : "V" }, {
"learnedat" : "", "name" : "shadow claw", "gen" : "V" }, { "learnedat" : "",
"name" : "air slash", "gen" : "V" }, { "learnedat" : "", "name" : "scratch",
"gen" : "V" }, { "learnedat" : "", "name" : "growl", "gen" : "V" }, {
"learnedat" : "", "name" : "ember", "gen" : "V" }, { "learnedat" : "", "name"
: "smokescreen", "gen" : "V" }, { "learnedat" : "7", "name" : "ember", "gen" :
"V" }, { "learnedat" : "10", "name" : "smokescreen", "gen" : "V" }, {
"learnedat" : "17", "name" : "dragon rage", "gen" : "V" }, { "learnedat" :
"21", "name" : "scary face", "gen" :
....
"method" : "Move Tutor  FRLG" }, { "name" : "mimic", "method" : "Move Tutor
FRLG" } ] }, "damages" : { "normal" : "1", "fire" : "0.5", "water" : "2",
"electric" : "2", "grass" : "0.25", "ice" : "1", "fight" : "0.5", "poison" :
"1", "ground" : "0", "flying" : "1", "psychic" : "1", "bug" : "0.25", "rock" :
"4", "ghost" : "1", "dragon" : "1", "dark" : "1", "steel" : "0.5" }, "misc" :
{ "sex" : { "male" : 87.5, "female" : "12.5" }, "abilities" : { "normal" : [
"Blaze" ], "hidden" : [ "Solar Power" ] }, "classification" : "flame pokemon",
"height" : "5’07″", "weight" : "199.5", "capturerate" : 45, "eggsteps" :
"5120", "expgrowth" : "1059860", "happiness" : "70", "evpoints" : [ "3 Sp.
Attack Point(s)" ], "fleeflag" : "94", "entreeforestlevel" : "36" } }

```

That's a lot of data! And it's incredibly difficult to read. In order to make it more readable for ourselves, we can tack on a `.pretty()` to our commands:

```
> db.pokemons.find({name: "Charizard"}).pretty()
{
  "_id" : ObjectId("5e93c8708a0be3ec97f570fc"),
  "id" : "006",
  "name" : "Charizard",
  "img" : "http://img.pokemondb.net/artwork/charizard.jpg",
  "type" : [
    "Fire",
    "Flying"
  ],
  ...
    "classification" : "flame pokemon",
    "height" : "5’07″",
    "weight" : "199.5",
    "capturerate" : 45,
    "eggsteps" : "5120",
    "expgrowth" : "1059860",
    "happiness" : "70",
    "evpoints" : [
      "3 Sp. Attack Point(s)"
    ],
    "fleeflag" : "94",
    "entreeforestlevel" : "36"
  }
}
```

For the sake of space, you've noticed that we've pared a lot of the data out. Take note that in our JSON that we looked at earlier, there was no `_id` key. That key is a unique identifier for mongo! You can create your own unique identifiers as well.

We could spend an entire week (or more) talking about the intricacies of mongo's system, but knowing the basics for how to view your data within your command line is good enough! If you have any other interest, please consult the documentation!

Working with Mongo in Node:

Having a working database is great, but it doesn't help us if we can't interact with it in our code! Let's take our code from the previous lecture (middleware) and add on to it!

First, we need to add a package for us to interact with our database. There are a number of available options, but we're going to use Mongoose! Mongoose is robust package that allows for us to quickly and simply interact with our data in our mongo database.

```
npm i mongoose
```

Now that we have mongoose added to our code, let's create a brand new route for characters from our favorite show: X-Files! We'll need to create a new directory: `xfiles` and a file in there `xfilesRoute.js`

```
.
├── index.js
├── lib
|   ├── middleware
|   |   ├── bodyParser.js
|   |   └── logger.js
|   └── swagger.js
├── package-lock.json
├── package.json
└── routes
```

```
    ├── office
    |   ├── office.js
    |   └── officeRoute.js
    ├── parksAndRec
    |   ├── parksAndRecRoute.js
    |   └── parksNRec.js
    └── xfiles
        └── xfilesRoute.js
```

Let's start with our xfilesRoute.js being nothing more than a quick "hello world" styled file, and then we can add the route in our index:

`xfilesRoute.js`

```
const express = require("express");
const bodyParser = require("../../lib/middleware/bodyParser");


const sayHello = (req, res) => {
  res.send("The Truth Is Out There")
}


const xfilesRouter = express.Router();

xfilesRouter
  .route("/")
  .get(getAllCharacter);

module.exports = xfilesRouter;
```

`index.js`

```
const express = require("express");
const officeRouter = require("./routes/office/officeRoute");
const parksAndRecRouter = require("./routes/parksAndRec/parksAndRecRoute");
const xfilesRouter = require("./routes/xfiles/xfilesRoute");

const logger = require("./lib/middleware/logger");
const app = express();
const swaggerUI = require("swagger-ui-express");
const swaggerDoc = require("./lib/swagger");

app.use(logger);
app.use("/api-docs", swaggerUI.serve, swaggerUI.setup(swaggerDoc));
app.use("/office", officeRouter);
app.use("/parksAndRec", parksAndRecRouter);
app.use("/xfiles", xfilesRouter);

const port = 3000;
app.listen(port);
console.log("Now listening on port " + port);
console.log(`Swagger docs at localhost:${port}/api-docs`);
```

So, now when we attempt to call our API's with a `GET` at `localhost:3000/xfiles`, we'll now receive:

```
"The Truth Is Out There"
```

Now that we have our endpoint set up, let's turn our server into a CRUD app! CRUD stands for `Create`, `Read`, `Update`, and `Delete`, which are all methods we'll want to add to our server so that we can add, read, update, and delete characters from the X-Files!

Creating a New Character

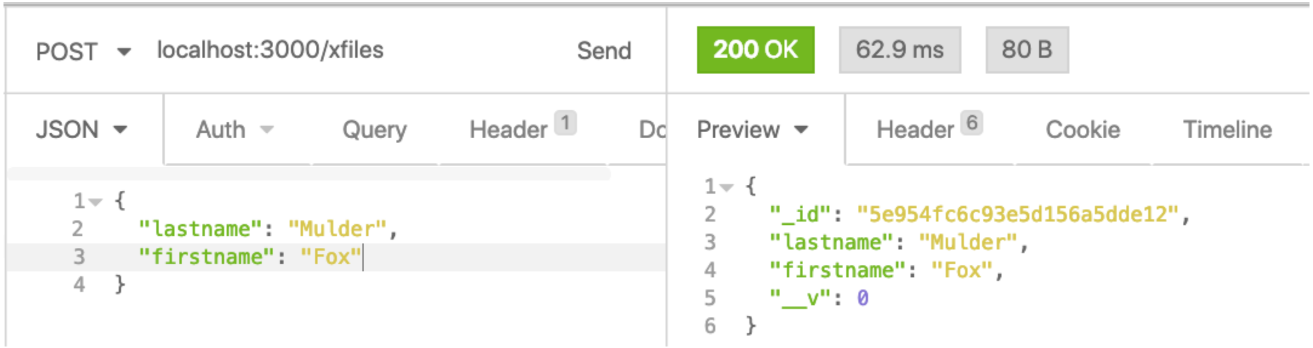
Before we can read, update, or delete a character, we'll need to create one first! And in order to create a character, we'll need to connect to the database! One thing to note before we go on, we could theoretically connect to our database once when we start our application, and just keep the connection open. However, opening connections and leaving them open is somewhat bad practice, so we should plan to open our database connection, make the calls, and then close them when we're done! Let's start with adding a character:

```
const express = require("express");
const bodyParser = require("../../lib/middleware/bodyParser");
const mongoose = require("mongoose");

const mongoURL = "mongodb://127.0.0.1:27017/xfiles";

const XfilesCharacter = mongoose.model("xfilescharater", {
  lastname: String,
  firstname: String,
});

const addXfilesCharacter = async (req, res) => {
  try {
    const xfilesCharacter = new XfilesCharacter(req.body);

16      mongoose.connect(mongoURL);
17      const result = await xfilesCharacter.save();
18      mongoose.disconnect();
19
20      res.send(result);
21    } catch (error) {
22      console.error(error);
23      res.status(500);
24      res.send(error);
25    }
26  };
27
28  const xfilesRouter = express.Router();
29
30  xfilesRouter
31    .route("/")
32    .post(bodyParser.json(), addXfilesCharacter)
33
34  module.exports = xfilesRouter;
35
```

We created the sayHello function just to make sure our route was working, but now we can get rid of it (or keep it if you want to know where the truth is)! In the above code, we've added a few things! First and foremost, we brought in `mongoose` at line 3. We can't really do much without that. Recall when we first called `mongo` in our command line, connecting to our mongo server? When we connected, the connection printed the URL to the server. That's the same URL here, but instead of adding any parameters, we specified a path `/xfiles`. In the event that you already have a database named `xfiles` then you'll connect to it, but if you don't, no worries! You've now created one!

At line 7, we've created a schema called `XFilesCharacter`. This schema will allow for us to interact with the database. The first parameter in our schema, `xfilescharacter`, is the collection that we would like to connect to in the defined database (`xfiles`). The object that we passed as the second parameter is the object we expect to interact with.

Starting at line 12, we actually have our code to add a character to our database. First, we add the character obtained from the req.body to our XfilesCharacter schema (in real life scenarios, some object validation would occur before we attempt to interact with our schema). Afterward we connect to our database (with our mongoURL), and then save. On save, we save the results in the variable `result` and ultimately send it back. Let's see what happens!



```
{
  "lastname": "Mulder",
  "firstname": "Fox"
}
```

```
{
  "_id": "5e954fc6c93e5d156a5dde12",
  "lastname": "Mulder",
  "firstname": "Fox",
  "__v": 0
}
```

On sending the post with a body similar to that of our schema, we receive an entirely new body with the keys we sent, but with an `_id` and a `__v`. The `_id` is the same as the `"_id" : ObjectId("5e93c8708a0be3ec97f570fc")` key value pairing we saw with Charizard above! That is just the unique key. The `__v` you can just ignore for now, as it's the `versionKey` property. Now that we've inserted a character, insert another one with the body:

```
{
  "lastname" : "Scully",
  "firstname" : "Dana"
}
```

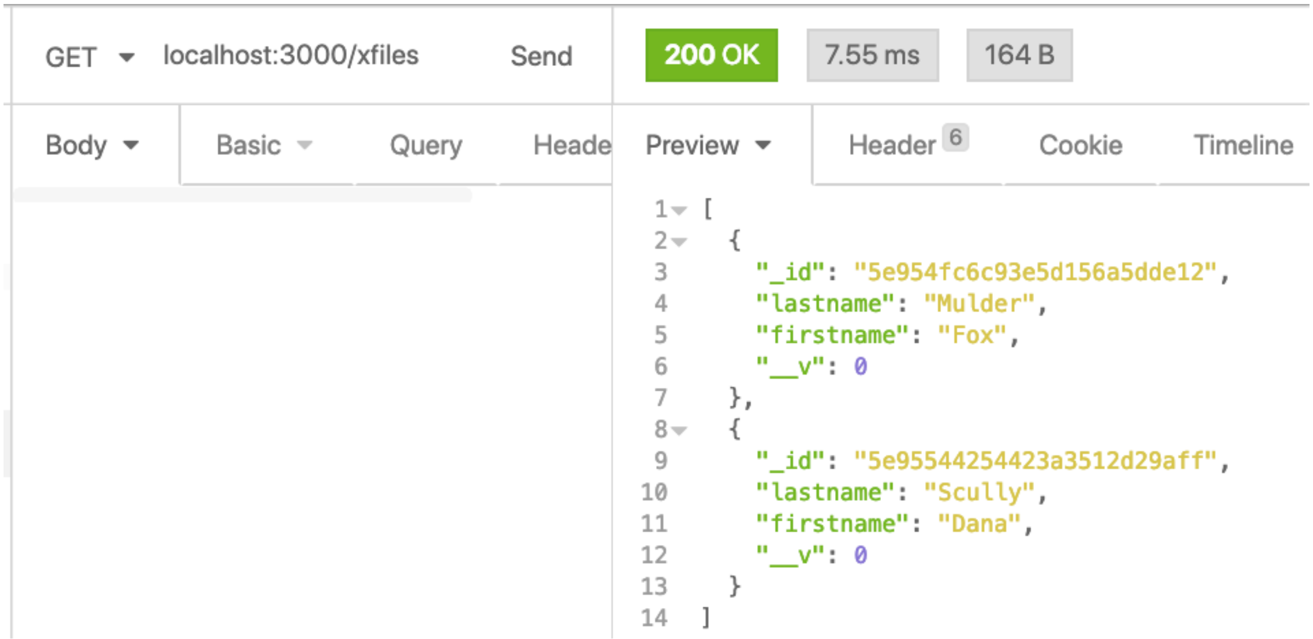
Reading Data

We've created a couple of characters, now let's see if we can retrieve them! Because it's the easier of the two options (if you remember from the pokemon example above), let's see if we can just get everything!

```
...

const getAllCharacters = async (req, res) => {
  try {
    mongoose.connect(mongoURL);
    const results = await XfilesCharacter.find().exec();
    mongoose.disconnect();

    res.send(results);
  } catch (error) {
    console.error(error);
    res.status(500);
    res.send(error);
  }
};

...
```

```


const xfilesRouter = express.Router();

xfilesRouter
  .route("/")
  .post(bodyParser.json(), addXfilesCharacter)
  .get(getAllCharacters);

module.exports = xfilesRouter;
```

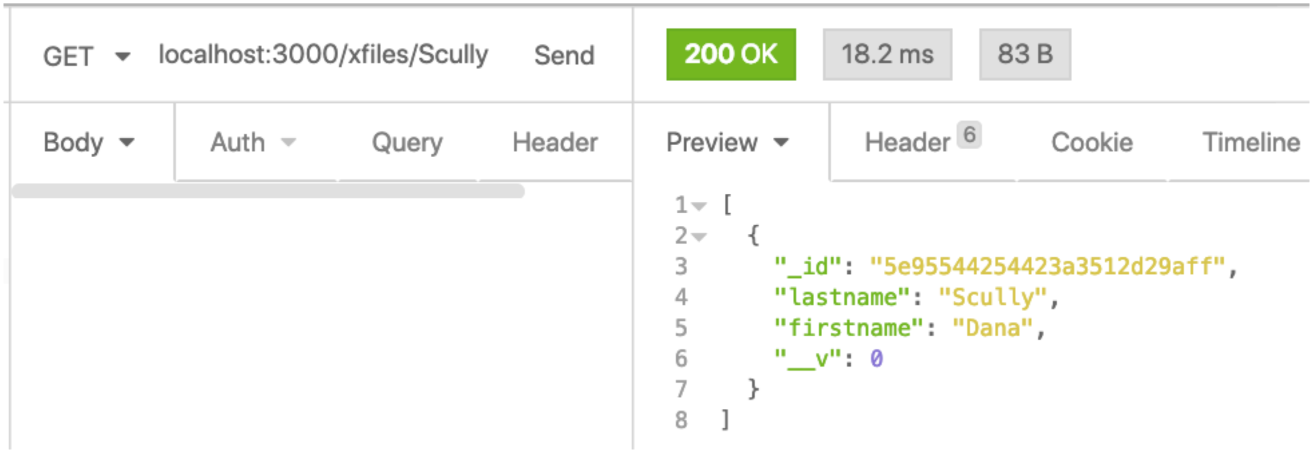
Just like how we used the find in the database repl, we just use the same general idea with our schema! If we use our schema, and call find with no arguments, we'll receive everything!



```
[
  {
    "_id": "5e954fc6c93e5d156a5dde12",
    "lastname": "Mulder",
    "firstname": "Fox",
    "__v": 0
  },
  {
    "_id": "5e95544254423a3512d29aff",
    "lastname": "Scully",
    "firstname": "Dana",
    "__v": 0
  }
]
```

For getting a specific character, however, we'll need to pass in a parameter. Let's grab those characters by their last names:

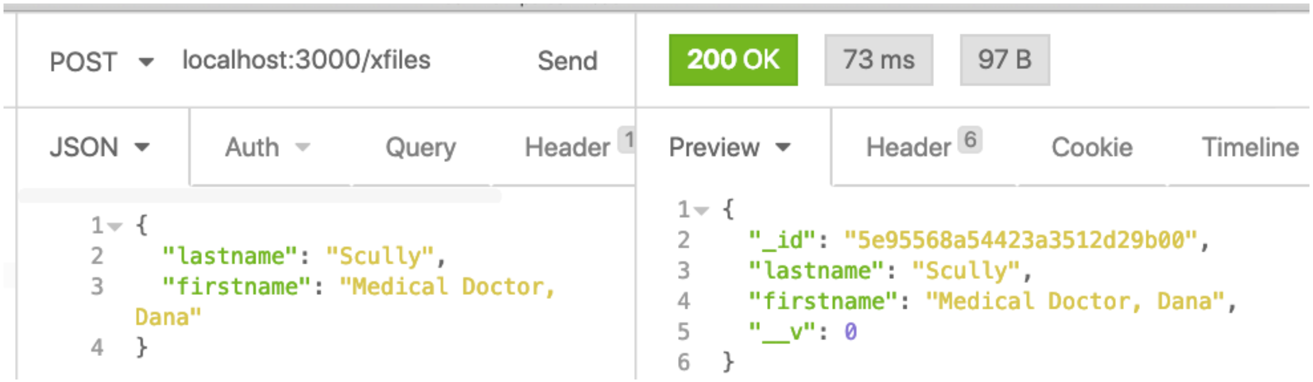
```
...

const getXfilesCharacter = async (req, res) => {
  try {
    mongoose.connect(mongoURL);
    const results = await XfilesCharacter.find({
      lastname: req.params.lastname,
    }).exec();
    mongoose.disconnect();

    res.send(results);
  } catch (error) {
    console.error("error", error);
    res.status(500);
    res.send(error);
  }
};

...

const xfilesRouter = express.Router();

xfilesRouter
  .route("/")
  .post(bodyParser.json(), addXfilesCharacter)
  .get(getAllCharacters);

xfilesRouter.route("/:lastname").get(getXfilesCharacter);

module.exports = xfilesRouter;

```



```
[
  {
    "_id": "5e95544254423a3512d29aff",
    "lastname": "Scully",
    "firstname": "Dana",
    "__v": 0
  }
]
```

You might be tempted to try to update data by just overwriting something, such as changing `Dana Scully` to `Medical Doctor, Dana Scully`, and you'd be right in wanting to do so (she is a medical doctor, after all), but if you were to attempt to overwrite based entirely off of values that are not the unique `_id`, then you'll only create a new record:

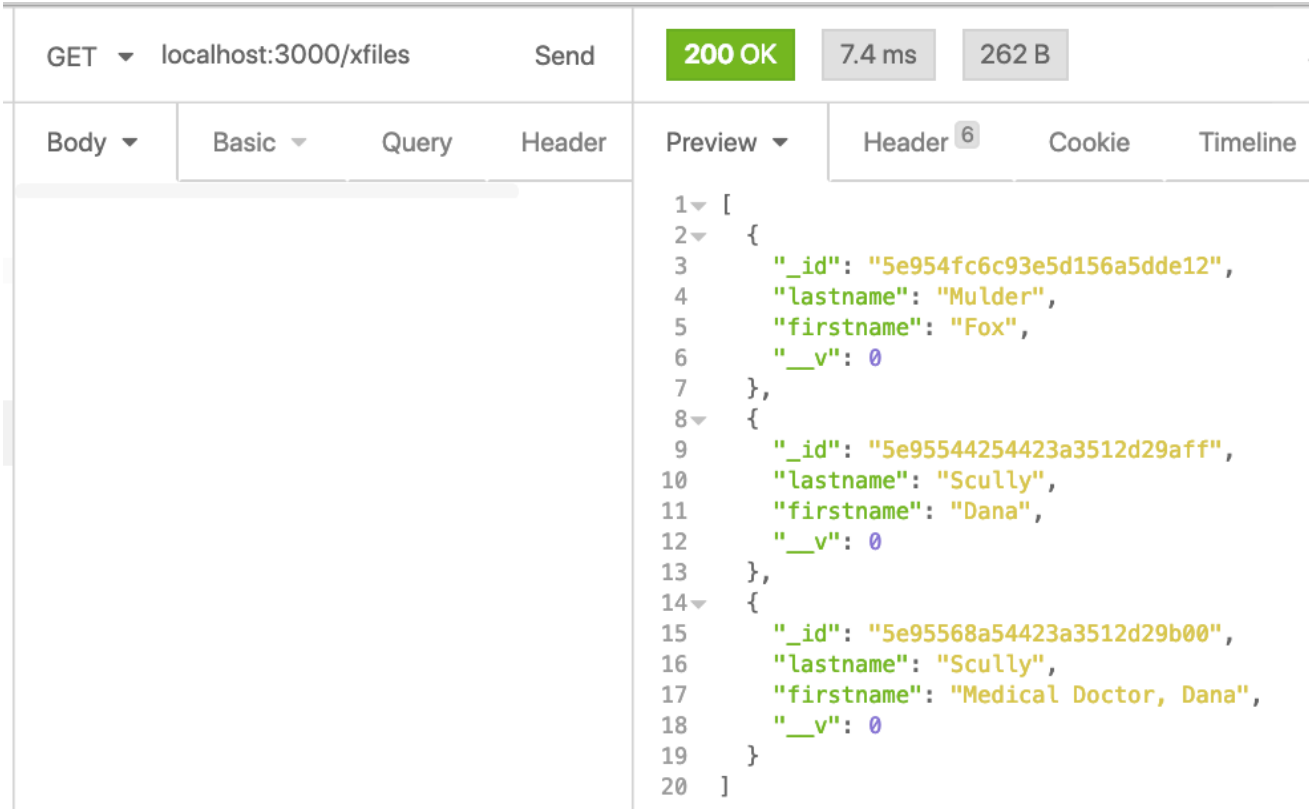
Inserting:



```
{
  "lastname": "Scully",
  "firstname": "Medical Doctor,
Dana"
}
```

```
{
  "_id": "5e95568a54423a3512d29b00",
  "lastname": "Scully",
  "firstname": "Medical Doctor, Dana",
  "__v": 0
}
```

Get all:



```
[
  {
    "_id": "5e954fc6c93e5d156a5dde12",
    "lastname": "Mulder",
    "firstname": "Fox",
    "__v": 0
  },
  {
    "_id": "5e95544254423a3512d29aff",
    "lastname": "Scully",
    "firstname": "Dana",
    "__v": 0
  },
  {
    "_id": "5e95568a54423a3512d29b00",
    "lastname": "Scully",
    "firstname": "Medical Doctor, Dana",
    "__v": 0
  }
]
```

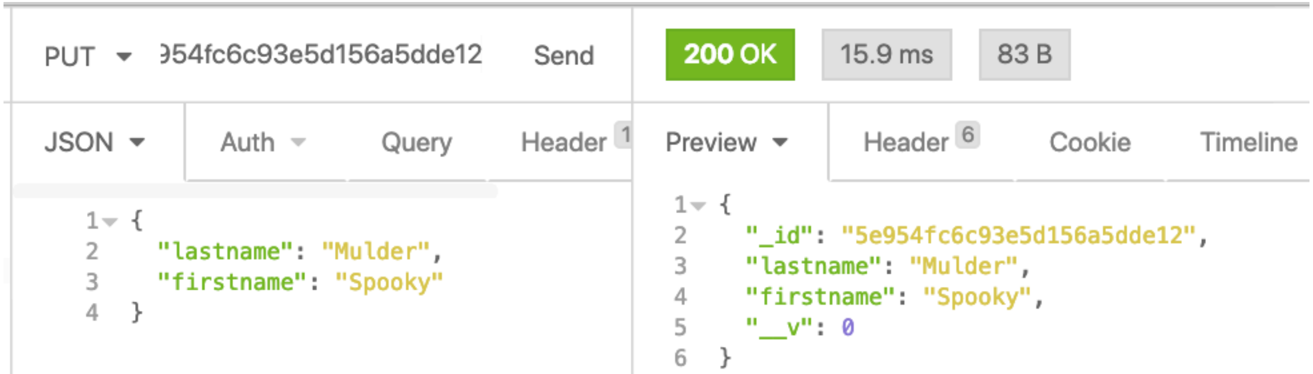
Updating Data

In order to update data, you'll need to find by a specific unique key! Let's write some code so that we can update `Fox Mulder` with his `_id`:

```
...

const updateCharacter = async (req, res) => {
  try {
    mongoose.connect(mongoURL);

    const id = req.params.id;
    const character = await XfilesCharacter.findById(id).exec();
    character.set(req.body);
    const result = await character.save();
    mongoose.disconnect();

    res.send(result);
  } catch (error) {
    console.error("error", error);
    res.status(500);
    res.send(error);
  }
};

...

const xfilesRouter = express.Router();

xfilesRouter
  .route("/")
  .post(bodyParser.json(), addXfilesCharacter)
  .get(getAllCharacter);

xfilesRouter
  .route("/:id")
  .put(bodyParser.json(), updateCharacter)

xfilesRouter.route("/:lastname").get(getXfilesCharacters);

module.exports = xfilesRouter;

```

There are multiple things happening in the above code! First, notice that we're grabbing an ID out of the params, and then using `XfilesCharacter.findById(id).exec()`. There, we're grabbing an object that specifically matches on only that ID (given that it's a unique identifier). Then, we're using the character we assigned it to, and using `character.set(req.body)`. This is where we're passing in our new and improved body to be saved over our previous data, which we then do with the following line of `character.save()`.

Additionally, take note of the router we're using. We're passing in a parameter, but more importantly, we're using `put`, a common http method to denote that we're updating data.



```
{
  "lastname": "Mulder",
  "firstname": "Spooky"
}
```

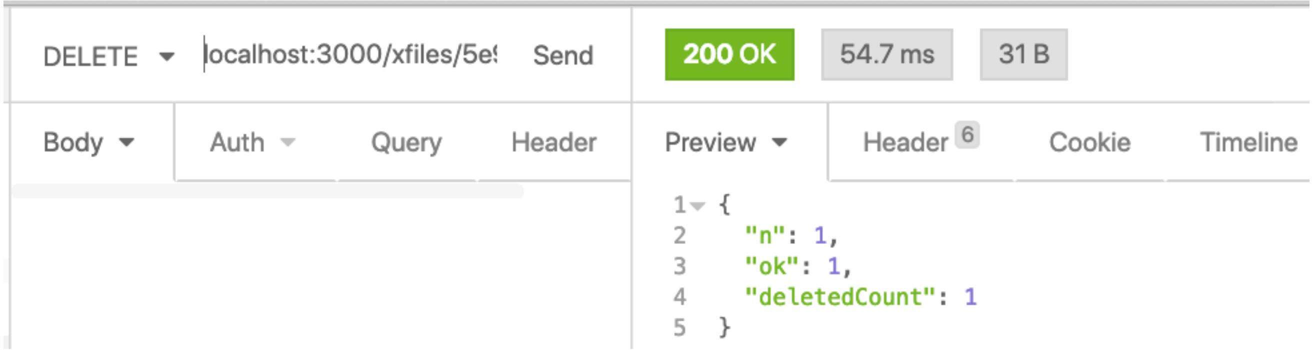
```
{
  "_id": "5e954fc6c93e5d156a5dde12",
  "lastname": "Mulder",
  "firstname": "Spooky",
  "__v": 0
}
```

Removing Data

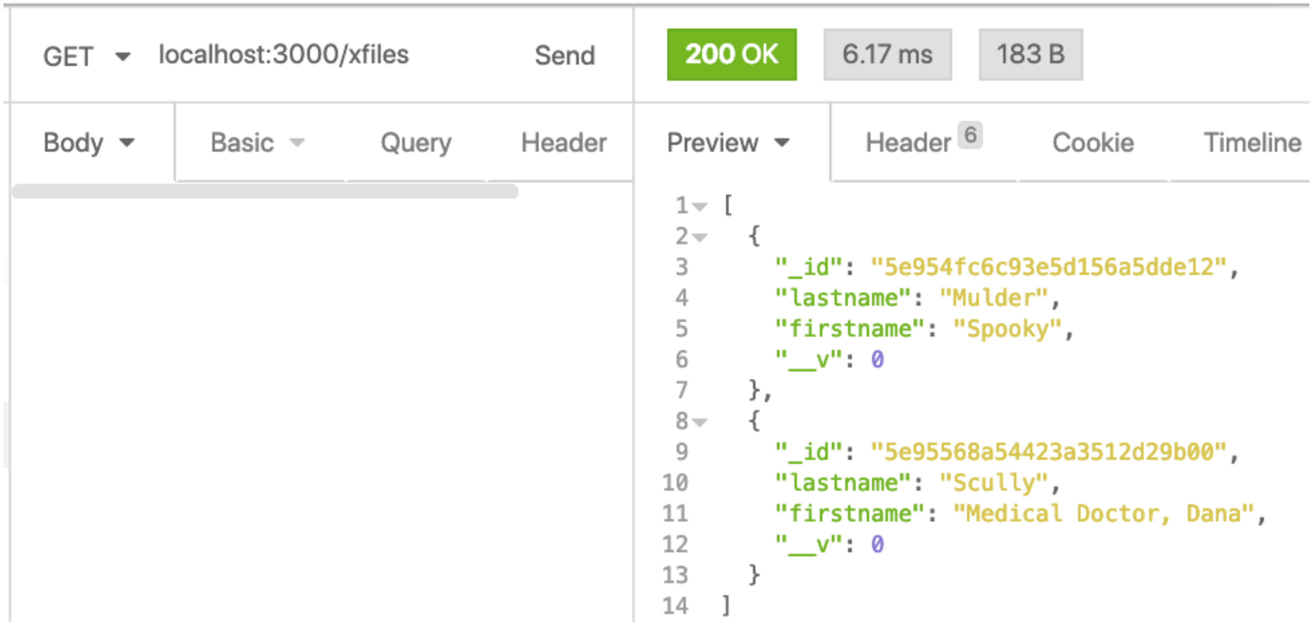
Often times you'll need to remove data from a given database, this can be for any number of reasons (maybe you accidentally added two of the same characters?). For us, we're going to remove our characters with the same route as the update:

```
...

const removeCharacter = async (req, res) => {
  try {
    const id = req.params.id;

    mongoose.connect(mongoURL);
    const result = await XfilesCharacter.deleteOne({ _id: id }).exec();
    mongoose.disconnect();

    res.send(result);
  } catch (error) {
    console.error("error", error);
    res.status(500);
    res.send(error);
  }
};

const xfilesRouter = express.Router();

xfilesRouter
  .route("/")
  .post(bodyParser.json(), addXfilesCharacter)
  .get(getAllCharacter);

xfilesRouter
  .route("/:id")
  .put(bodyParser.json(), updateCharacter)
  .delete(removeCharacter);

xfilesRouter.route("/:lastname").get(getXfilesCharacter);

module.exports = xfilesRouter;

```

Here, we take in the id, and then choose the `deleteOne` method, and pass in the `{ _id: id }`. Let's try removing our duplicate Scully!



```
{
  "n": 1,
  "ok": 1,
  "deletedCount": 1
}
```



```
[
  {
    "_id": "5e954fc6c93e5d156a5dde12",
    "lastname": "Mulder",
    "firstname": "Spooky",
    "__v": 0
  },
  {
    "_id": "5e95568a54423a3512d29b00",
    "lastname": "Scully",
    "firstname": "Medical Doctor, Dana",
    "__v": 0
  }
]
```

There are a ton of ways to expand further on Mongo, but for the time being (and because this isn't a databases class), this is more than enough to know for how to store and retrieve information!