Using a database with your Node app

Now that you've got a handle on creating a Node app with basic templates, let's try adding a database. We'll be taking a look at MongoDB (a JSON-based DB). MongoDB is the "M" in a MEAN or MERN stack and it is commonly used with JS-based frameworks and servers because it's fast and JSON plays very nicely with JavaScript. (I'll add a little bit about how to use MySQL at the end as this is a database you're probably more familiar with.)

MongoDB

MongoDB is a document-based database. It doesn't use SQL like a traditional database. Instead, it uses JSON to store data. Instead of tables, there are collections and instead of rows, there are documents. In other words:

- Collections = tables
- Documents = rows

This means that a collection is made up of a set of documents. Each document contains data for one item in the collection. The data is formatted in JSON.

Installation

We'll test things out locally.

1. Download and install the latest MongoDB Community Server (https://www.mongodb.com/try/download/community).
    a. **For Windows:** Download the package.
    b. **For Mac:** Install using Homebrew using these instructions.
2. Install using the defaults (you can include MongoDB Compass, which is the GUI tool).

> *Windows installation*
>
> For Windows, you'll need to add MongoDB to your computer's Path environment variable so that you'll be able to use the MongoDB shell.
>
> 1. Click on the Windows **Start** menu.
> 2. Type "env" and hit enter to edit system environment variables.
> 3. In the dialog box, click on the **Environment Variables…** button.
> 4. In the "System variables" pane, look for a variable named "Path".
> 5. Select "Path" and click **Edit**.
> 6. Click on **Browse** and navigate to your MongoDB installation folder. Expand the folder and click on the **bin** folder.
> 7. Click **OK**. You should now see the path to the ***bin*** folder in your Path variable. (For me, the path was "`C:\Program Files\MongoDB\Server\7.0\bin`".)

3. Download and install the MongoDB shell—mongosh (https://docs.mongodb.com/mongodb-shell/install/#std-label-mdb-shell-install).
4. Test the installation:

a. Open your CLI and enter "mongosh" at the prompt. If everything goes well, you should enter the Mongo shell.
b. To create and select a new database type at the prompt:
    - `use <your_db_name>`
c. Try adding some data by creating a collection then inserting data to the collection:
    - `db.createCollection("menuLinks")`
    - `db.menuLinks.insertOne({ "weight": 0, "path": "/", "name": "Home" })`
d. You can try checking all the data/documents in the collection:
    - `db.menuLinks.find({})`

Notice that the `find()` query has empty curly brackets. Empty curly brackets (`{}`) in a query are the equivalent of * in SQL (so it's a SELECT *).

Also notice that, in the listing, there's an _id field. This is added by MongoDB and serves as the index (primary key). It is of type BSON, which is a binary representation of JSON.

Useful commands are:

- `find()`
- `findOne()`
- `insertOne()`
- `insertMany()`
- `updateOne()`
- `deleteOne()`

We'll be looking at the native Node.js driver's version of these commands since we'll be using them with Node. You can check the MongoDB documentation for MongoDB Shell to see how to run the shell commands (https://docs.mongodb.com/mongodb-shell/crud/#std-label-mdb-shell-crud).

For the Node.js native MongoDB driver, the documentation is found here (first link is the official documentation):

- https://mongodb.github.io/node-mongodb-native/6.3/
- https://www.mongodb.com/docs/drivers/node/current/ (this is MongoDB's documentation page for the Node.js driver—once you get how to connect to MongoDB, you may find it easier to use the *Quick Reference*)

Of interest are the Collection module (see the ***Exports*** list on the right). The Collection module contains the CRUD (**C**reate **R**ead **U**pdate **D**elete) operations.

Set up Node app to use MongoDB

For this example, we'll be using the previous Node/Express/Pug example and add a MongoDB database. Don't forget to run the Nodemon and Browser-sync scripts.

1. Create a MongoDB database, add a collection and add some data. I named my DB "testdb" and created a collection named "menuLinks". You can use the data added in the `insert()` command above.
2. Open your CLI to the root folder of your app and install the mongo:

a. `npm i mongodb`

3. In your Node app (e.g. ***index.js***), add the following where you import your modules/libraries:

```
const { MongoClient } = require("mongodb");

const dbUrl = "mongodb://localhost:27017/testdb"; //default port is 27017
const client = new MongoClient(dbUrl);
```

4. Try making a connection. To do this, we're going to create some async functions: one to make a connection and return the database and one to retrieve the menu link data from the menuLinks collection we created using MongoDB Shell earlier. You can place these functions at the bottom of the page and out of the way of the routing/other server code.

```
async function connection() {
  db = client.db("testdb");
  return db;
}

async function getLinks() {
  db = await connection();
  var results = db.collection("menuLinks").find({});
  res = await results.toArray();
  return res;
}
```

> ***async/await***
>
> When you need your code to wait for a bit of code which takes longer to execute before continuing with code execution, it is better to write asynchronous requests. Previously, you've used callback functions to execute code upon a successful asynchronous request. When there are *multiple* asynchronous requests, the code starts to become messy and hard to follow because you'll need multiple nested callback functions (it's not elegant code). This is why **async/await** was created.
>
> When you want to make an asynchronous request (e.g. MongoDB functions or API requests), use **async/await**. To use async/await, you define any function which *involves* asynchronous requests as an **async** function. Within the function, you use **await** to retrieve any results of an asynchronous request.
>
> In the above step, for the connection() function, the MongoDB .find() method is asynchronous (it returns a Promise), but in order to do something with the data returned from the database, we have to wait for a successful retrieval of the data. To do this, use **await** to wait for the result of an asynchronous request before continuing to execute the rest of the function code.

5. Now convert the code to use the menu links from the *menuLinks* collection to populate the site menu. Modify the home page routing code to use links result from the `getLinks()` asynchronous function (which returns an array of your menu link data from the menuLinks collection). New code has been indicated in red.
    a. First make the callback function an asynchronous function by adding async before the (request, response).
    b. Add code to load a local variable named links with the results from `async getLinks()`.

```
app.get("/", async (request, response) => {
  let links = await getLinks();
  response.render("index", { title: "Home", menu: links });
});
```

> *Notes*
>
> - In the above code, we previously had a global `links` variable so since we're initializing a local `links` variable with the result from `getLinks()`, we don't need to change the render line because it will just use the locally scoped `links` variable instead. If you refresh the home page, you should now see only the *Home* link in the site menu because it's now using the results from the **menuLinks** collection.
>   - The same property names (**path** and **name**) were used when adding a document to the collection so that you don't have to modify the template code.
>   - Recall the menu in the template (*layout.pug*):
>
> ```
> nav#main-nav
>   ul
>     each link in menu
>       li
>         a(href=link.href) #{link.name}
> ```
>
> - We've stuck with the callback structure for the routing code but you could put the callback code in a function and execute that instead to keep the routing code a little neater.

6. Modify the `app.get("/about"…)` code to use the links from the `getLinks()` function.

You should now see the links in the main menu as taken from your MongoDB database on all pages.

Add code to list all menu links

Just to demonstrate a few CRUD functions, we'll next create a central admin page for managing menu links stored in the DB (without a login for simplicity's sake). This admin page will list all links and be the starting point for adding, editing and deleting a link from the menuLinks collection. We will later create other pages for updating and deleting a link.

1. Add a new route to ***index.js*** pointing to "/menu/admin". This will be the path for our admin page for menu links. The menu links listing is being

```
app.get("/admin/menu", async (request, response) => {
  let links = await getLinks();
  response.render("menu-list", { title: "Menu links admin", menu: links });
});
```

2. Create a new template named ***menu-list.pug*** (matching the render name). Add the following code to the template:

```
extends layout

block layout-content
  div.content
    h1.page-title Administer menu links
    a(href="/admin/menu/add") + Add link
    table
      thead
        tr
          th Link
          th Operations
      tbody
        each link in menu
          tr
            td #{link.name}
            td
```

> *Understanding the code*
>
> The above template creates a table listing all of the links in `menuLinks` (remember that the `menu` variable was populated with the links from the DB.
>
> For each link, there will an update/delete link (to be added later in the currently-empty TD). The update/delete functionalities will be triggered using forms.
>
> There is a link to an "add link" page (to be added) above the table. We will next add the code/functionality for this page.

Add code to create a new link

First, we'll create a simple form to add a new link

1. Create a new path which will render an "add link" form. Add to ***index.js*** (cont'd next page):

```
app.get("/admin/menu/add", async (request, response) => {
  let links = await getLinks();
  response.render("menu-add", { title: "Add menu link", menu: links });
});
```

2. Create a ***menu-add.pug*** file and add the following to the file:

```
extends layout

block layout-content
  div.content
    h1.page-title Add menu link
    form(method="post", action="/admin/menu/add/submit")
      div
        label(for="wgt") Weight:
        input(type="number", min="0", id="wgt", name="weight")
      div
        label(for="href") URI:
        input(type="text", id="href", name="path", placeholder="e.g. /my-
page")
      div
        label(for="link-name") Name:
        input(type="text", id="link-name", name="name", placeholder="e.g.
About")
      button(type="submit", value="add-link") Add link
```

The above code adds a form which submits to ***/admin/menu/add/submit*** (we'll need to add the code for processing the form submission and adding the link here). Of importance are the `name` values for the inputs. These will be used in the code for */admin/menu/add/submit*.

3. Before we can even retrieve form values, we need to tell Express how to parse the form data. Add the following lines of code to ***index.js***—I would put them with the other .use() statements. This allows us to use JSON-like methods for parsing data.

```
app.use(express.urlencoded({ extended: true }));
app.use(express.json());
```

4. Now we need to add code to handle the actual form submission. Since the form (in the template) is a POST form, in ***index.js*** add:

```
app.post("/admin/menu/add/submit", async (request, response) => {

  //get form data
  let weight = request.body.weight; //get the value for field with
name="weight"
  let href = request.body.href; //request.body is form POST data
  let name = request.body.name;
  var newLink = { "weight": weight, "path": href, "name": name };

  await addLink(newLink);
  response.redirect("/admin/menu"); //redirect back to admin page

});
```

(The `addLink()` function code is on the next page.)

```
async function addLink(link) {
  db = await connection();
  var status = await db.collection("menuLinks").insertOne(link);
  console.log("link added");
}
```

- https://www.mongodb.com/docs/drivers/node/current/usage-examples/insertOne/#std-label-node-usage-insert
- You can test the code by going to ***/admin/menu/add*** to view the form and submitting. After submitting, you should see the new link added.

Add code to delete a link

We'll add the delete link code first as it's a little simpler.

1. First we need to add a path to a delete page. In the empty TD on the admin listing page (***menu-list.pug***), add:

   …

   ```
   td
     form(action="/admin/menu/delete")
       input(type="hidden", name="linkId", value=link._id)
             button(type="submit") Delete
   ```

   Clicking on "Delete" will go to the ***/admin/menu/delete?linkId***=<*link _id value*> path. Note that since the form has no method, it's using the default GET method. GET form values get sent via the URL, which is why you should see the `linkId` field (by name) in the URL upon clicking on the button.

   > *Query strings*
   >
   > A query string is the "?..." stuff which comes after your URL in the address bar. For example, the query string in the following URL is highlighted.
   >
   > `http://localhost:3000/admin/menu/delete?linkId=61ee32ef1334a536eec4a443`

2. Since the code is for a GET form, we'll need the following:

   ```
   app.get("/admin/menu/delete", async (request, response) => {
     //get linkId value
     let id = request.query.linkId;

     await deleteLink(id);
     response.redirect("/admin/menu");
   });
   ```

```
async function deleteLink(id) {
  db = await connection();
  const deleteId = { _id: new ObjectId(id) };
  const result = await db.collection("menuLinks").deleteOne(deleteId);
  if (result.deletedCount == 1)
    console.log("delete successful");
}
```

- https://www.mongodb.com/docs/drivers/node/current/usage-examples/deleteOne/

> *Understanding the code*
>
> To delete just one document from the ***menuLinks*** collection, you'll need to use the `deleteOne(<filter>, <callback>)` method.
>
> The filter passes in what you're querying to match. In step 1 we were querying to match a specific `_id` value in ***menuLinks***.
>
> In the `deleteLinks()` function, we need to use the `ObjectId` constructor to convert the link _id value (from the query string) to an object of type `ObjectId`.

3. If you try out the above, you'll get an error because the ObjectId class has not been imported. Modify the line where you import MongoClient to include ObjectId.

```
const { MongoClient, ObjectId } = require("mongodb");
```

Add code to update a link

This functionality will require a template for the edit form as well as a route for the actual updating (processing the form data).

First add a path to the edit form page and then create the edit form page.

1. Add the following form to the same TD as the one for the delete form in the ***menu-list.pug*** template.

```
…

form(action="/admin/menu/edit")
  input(type="hidden", name="linkId", value=link._id)
  button(type="submit") Edit
```

2. Add an async function to retrieve a single document from ***menuLinks*** by _id.

```
async function getSingleLink(id) {
  db = await connection();
```

```
  const editId = { _id: new ObjectId(id) };
  const result = await db.collection("menuLinks").findOne(editId);
  return result;
}
```

- https://www.mongodb.com/docs/drivers/node/current/usage-examples/findOne/

3. Add the GET route for "/admin/menu/edit" that will be used for displaying the edit form page by adding the following to ***index.js***:

```
app.get("/admin/menu/edit", async (request, response) => {
  if (request.query.linkId) {
    let linkToEdit = await getSingleLink(request.query.linkId);
    let links = await getLinks();
    response.render("menu-edit", { title: "Edit menu link", menu: links,
editLink: linkToEdit });
  } else {
    response.redirect("/admin/menu");
  }
});
```

Understanding the code

In the /menu/admin page, there is a GET form for the **Edit** button. The form's only data is to submit the `_id` for the link you want to edit (`name="linkId"`).

In the above code, since the form (holding the initial **Edit** button) is a GET form, the form data is passed in the URL rather than through the request/form submission body. This means that to retrieve the `linkId` GET parameter, we need to retrieve it from the query string via `request.query.linkId`.

As this code is meant to render the edit form, at this point, there is no user submission to be handled. Instead, we want to pre-populate the form with the selected link's values, so we need to pass along the link's data.

To do this, we need to query the collection for the link data we want. To ensure we select the correct one, we'll query using the `_id` (the primary key) because this is unique. `findOne()` will only retrieve one result from the `menuLinks` collection as opposed to a list.

```
db.collection("menuLinks").findOne({ _id: id })
```

The rendering code will render the template named `menu-edit(.pug)`. Notice that a variable named `editLink` holds the value from the `result`. This is passed to the template so that we can use the values in the form display.

4. Now create a ***menu-edit.pug*** file and add the contents below. Notice how the `editLink.<property_name>` has been used in the input attributes. Remember that if using in **attributes** (in the parentheses), don't use quotes **and** don't use the `#{…}` syntax. The `#{…}` syntax is used in element contents (outside parentheses).

```
extends layout

block layout-content
  div.content
    h1.page-title Edit menu link
    form(method="post", action="/admin/menu/edit/submit")
      input(type="hidden", name="linkId", value=editLink._id)

      div
        -//add label and input for the weight with the value equaling the
editLink's weight
      div
        -//add label/input for the path
      div
        -//add label/input for the link name
      button(type="submit") Update link
```

5. Fill out the rest of the template, following the instructions in the comments.
6. Now we need to add the code to handle the actual form processing (i.e. retrieve form submission data and update the link in the DB. The following lines of code are skeleton code to guide you.

```
app.post("<path-of-edit-form-submission>", async (request, response) => {
  //get the _id and set it as a JSON object to be used for the filter
  let idFilter = { _id: new ObjectId(<replace_with_linkId_value>) };
  //get weight/path/name form values and build a JSON object containing
these (updated) values
  let link = {
    weight: <replace>,
    path: <replace>,
    name: <replace>
  };
  //run editLink(idFilter, link) and await the result

  response.redirect("/admin/menu");
});

async function editLink(filter, link) {
  db = await connection();
  //create the update set { $set: <JSON document> }
  //execute an updateOne() to update the link as selected via the filter
}
```

- https://www.mongodb.com/docs/drivers/node/current/usage-examples/updateOne/

Understanding the code

The method we'll need to update **one** document in a MongoDB collection is `updateOne(<filter>, <update>)`.

The filter is structured just like we've done before. Again, we're querying to match a specific _id.

The actual update portion is a little different:

```
{
  $set: {
    weight: <weight_value>,
    path: <path_value>,
    name: <name_value>
  }
}
```

The above uses the `$set` operator to specify the data changes. The text to the **left** of the colons are the field names from the MongoDB collection. The text to the **right** of the colons are the names of the *variables* in which the data are stored (retrieved from the form submission data in the lines above).

You can add `{ new: true }` after the `$set` (i.e. `{ $set: { }, <here> }`) will return the modified MongoDB document. You'd need this if you want to use the result of the method. You actually don't need this (unless desired), so you can omit this if you want.

Resources

The following resources are for more info (two links are the same ones provided somewhere above but were included here again):

- Useful examples for MongoDB Node.js driver: https://www.mongodb.com/docs/drivers/node/current/usage-examples/
- Adding sort to your query: https://www.mongodbtutorial.org/mongodb-crud/mongodb-sort/
    - You can add sort to your find() query options in the example with (from the Node.js MongoDB native driver documentation):

```
db.collection("menuLinks").find({}, { sort: { weight: 1 })
```

- Mongoose (alternative to the Node.js native MongoDB driver): https://mongoosejs.com/docs/index.html
    - Mongoose vs Node.js native driver: https://www.mongodb.com/developer/article/mongoose-versus-nodejs-driver/
- Info from MongoDB documentation about documents: https://docs.mongodb.com/manual/core/document/
- Handling file uploads (not a MongoDB thing but useful to know): https://www.twilio.com/blog/handle-file-uploads-node-express
- Using MongoDB Compass: https://docs.mongodb.com/compass/current/instance/

In the example in this document, we haven't secured our MongoDB database with an account. This was not done because the focus was on the actual database stuff and it would make this document very long. Take a look at this page for info on securing your database (we'll do this next week):

- https://blog.sqreen.com/top-10-security-best-practices-for-mongodb/
- https://docs.mongodb.com/manual/core/authentication/

If you've secured your database, the connection string will need to include the user account and password. To view how a connection string is formatted, see:

https://docs.mongodb.com/manual/reference/connection-string/

Node.js and MySQL

In this document, I've covered MongoDB because it's a popular choice for Node.js apps since it's JS-friendly (it's built using JSON-based structures). To see how you can use a MySQL database, check this resource:

https://www.mysqltutorial.org/mysql-nodejs/

It uses the ***mysql*** module/driver instead.