Workshop 7 - Heroku and MongoDB

This week, we will run through how to deploy the demo app to Heroku, and use mongoose to connect the app to a MongoDB database (hosted on MongoDB Atlas).
If you aren't familiar with these topics, you can watch the MongoDB and Mongoose lectures and the lecture on deploying to the cloud.

You can follow this tutorial sheet using the app which you made in tutorial 6, and/or directly apply the concepts to your project app.

If you want to use the app from tutorial 6, then you can either use your existing code, or download a fresh version:

- Create a new folder called tutorial07.
- Download the code called 'tutorial07-start'
- Run `npm install`.
- Run `node app.js` and use your browser to check that the app works.

Deploying to Heroku

Setting up a Heroku account

1. Open the Heroku signup page.
2. Sign up for a free account.
3. Fill in the form, choosing Node.js as the primary development language.
4. You should receive an email with a link to complete registration.

Installing Heroku Command Line Interface (CLI)

Heroku CLI is the command line interface tool for creating and managing Heroku apps.
We will be using it to deploy our app.

1. Heroku CLI requires git.
    - check to see that you have git installed: `git --version`
    - If you don't, install git
2. Install Heroku for your operating system:
    - Linux: `sudo snap install --classic heroku`.
      If snap is not installed, use `npm -g install heroku` instead.
    - Mac: `brew tap heroku/brew && brew install heroku`
    - Windows: use the installer found here
3. Verify installation: `heroku --version`

Making the app suitable for Heroku deployment

Initialising a git repo

Heroku expects your app directory to be located in a git repository.
Are you already managing this app in Git? If not:

1. Browse to the root of the app (where `package.json` is located).
2. Initialise the directory as a git repository with the command: `git init`
3. Create a `.gitignore` file at the root of the repository.
   Add to this all the files that should be ignored from version control. These may include:

```
node_modules/
.env
.idea/
.vscode/
```

4. Add all files: `git add -A`
5. Commit your files to the local repository: `git commit -m "initial commit"`

Making minor code changes

1. Make sure that a start script is defined in the `package.json` file:

```
"scripts": {
    "start": "node app.js"
},
```

- The "scripts" property in a `package.json` file is a dictionary, where the keys are names and the values are commands which are executed when you make the corresponding `npm` call.
- Now, when you run `npm start` (which is an alias for `npm run start`), `node app.js` is executed.

This is necessary because Heroku (by default) uses the command `npm start` to start your application.

> You may want to set up a code formatter like `prettier` to have a consistent code style.
>
> 1. `npm install -D prettier` to add prettier as a dependency.
> 2. Add `prettier` under "scripts" in `package.json`, e.g.
>   `"prettier": "prettier --single-quote --no-semi --tab-width 4 --write **/*.js *.js"`
> 3. Now you can invoke `npm run prettier` to get `prettier` to format the JS code.

2. Heroku assigns a dynamic port number (exposed to the app in the `$PORT` environment varaible) and expects apps to bind to the allocated port. Therefore, there will be a problem if your app is listening to a hardcoded port number.

To resolve this issue, change `app.listen` in `app.js` to:

```
app.listen(process.env.PORT || 3000, () => {
    console.log('The library app is running!')
})
```

This will allows the app to listen to either: the port number provided by Heroku when running on Heroku, or your hard-coded port number when running locally.

> What does the || operator do?
> Given `expr1` || `expr2`:
> If `expr1` if falsy (e.g. undefined, false, null), `expr2` will be returned. Otherwise, `expr1` will be returned.

3. Navigate to your app's directory in the terminal.
4. Make sure that the app can run locally without problem.
    - `npm start`
    - Open http://localhost:3000/ in your browser.
5. Add and commit these new changes:
    - `git add .`
    - `git commit -m "add start script, dynamic port number for app.js`

Deploying to Heroku

1. Login to Heroku using the command: `heroku login`
    - Your command prompt will ask you to press any key to complete the login through your web browser.
    - If, after logging in, you do not get back to the command prompt, you should open a new instance of the command prompt.
2. Complete the login to Heroku account and close the browser. You should see a message of successful login in the terminal.
3. Create a Heroku app for your library app, using the command:
    - `heroku create <app-name>`
    - Note that `app-name` needs to be unique across all of Heroku. Ideas for possible unique names include: `mydemo-<yourname>` or `INFO30005-t07-<your name>`.
    - If you don't provide an app name, Heroku will generate a random name for the app.
    - After the Heroku app is successfully created, you will see a link to the online app in the terminal, as well as the link to the git repository on Heroku.

    > If you have already created an app on Heroku, here are instructions for associating an existing Heroku app to a local repository.

4. Deploy your app to Heroku using the command line:
    - `git push heroku`
    - After that is successful, run: `heroku open`. Your browser should open, and you should shortly be looking at your app working on Heroku. Check everything works: for example, try adding `/people` to the URL.
    - Make sure you understand the workflow for updating and deploying your code. For example, change some text on the home page of your app, and then push it up to Heroku. (The version on Heroku is visible to the public, by the way.)
    - For more details visit here.

> Optionally, you can link your Heroku app to a GitHub repository. This will be useful for your actual project, so that your hosted app tracks changes to your code base.
> You can then specify a branch to deploy automatically to Heroku when changes are pushed to GitHub.
> On the other hand, you may prefer to keep your pushes to GitHub and Heroku independent of each other.

Setting up MongoDB Atlas

1. Follow these instructions to create an account on Atlas.
2. Create a new project:
    - Click the `New Project` button at the top right of the screen.

      > If you have created a project before, open the organisation name dropdown (top left) and select the organisation to visit the projects page.

    - Name this what you want (eg demo).
3. Build a database cluster:
    - Click `Build a Database`.
    - Use the free shared option (this will be more than sufficient for this subject).
    - Create a username and password pair which will be used by the app to authenticate with the database.
      Make sure you **store this username and password** since we will need to use it later.

      > Note: it is a good idea to use the password autogeneration tool, since it will generate a secure password for you.

    - Add 0.0.0.0/0 to the IP access lists. This will allow all IP addresses to connect.
      Click `Finish and Close` to return to your list of databases.
4. Connect to your cluster
    - Click the `Connect` button.
    - Choose the `Connect your application` option.
    - Choose the `Node.js` driver option with version `4.0 or later`.
    - **Copy the connection string.** We will need to use it later.
    - Close the connect dialog.
5. Make a collection:
    - Click the `Browse Collections` button, and click on the Collections tab.
    - Click the `Add My Own Data` button.
    - Enter a useful name for your database (e.g. `demo` for our demo app).
    - For collection name, use the data model you plan to store there (e.g. `authors` for our demo app).
    - Add some demo data to your collection by clicking the `Insert Document` button.
        - For the demo app, let's insert the following document:

```
_id: <leave as is>,
first_name: "John",
last_name: "Smith"
```

Connecting the app to MongoDB using Mongoose

- Mongoose is a MongoDB object-document modelling (ODM) tool.
  It abstracts over MongoDB, and offers a "schema-based solution to model application data".

1. Install mongoose. `npm install mongoose`
2. Install dotenv. `npm install -D dotenv`
    - Environment variables allow us to manage configuration data and keep this data separate from the codebase.
    - This is useful because this type of data is often private (e.g. api keys, connection passwords).
3. Create a file named `.env` inside your app folder and create an environment variable for connecting to the database on Atlas:

```
MONGO_URL="<replace-this>"
```

Replace `<replace-this>` with the URL and username/password pair saved above.
Change `myFirstDatabase` to `demo` (or the database name for your project app).

> e.g. `mongodb+srv://app:<password>@cluster0.fbzu2.mongodb.net/myFirstDatabase?retryWrites=true&w=majority` with username `app` and password `QNYT6pp57kzxswW0` becomes:
> `MONGO_URL="mongodb+srv://app:QNYT6pp57kzxswW0@cluster0.fbzu2.mongodb.net/demo?retryWrites=true&w=majority"`

4. This .env file won't be committed and therefore won't be on the Heroku server.
   To set the environment variable on Heroku we will run:
   `heroku config:set MONGO_URL="<replace-this>"`

Add code that makes our backend able to talk with our database

1. Create a new file `models/index.js`.

```
// Load envioronment variables
if (process.env.NODE_ENV !== 'production') {
    require('dotenv').config()
}
const mongoose = require('mongoose')

// Connect to your mongo database using the MONGO_URL environment
variable.
// Locally, MONGO_URL will be loaded by dotenv from .env.
// We've also used Heroku CLI to set MONGO_URL for our Heroku app before.
mongoose.connect(process.env.MONGO_URL || 'mongodb://localhost', {
    useNewUrlParser: true,
    useUnifiedTopology: true,
    dbName: 'demo'
```

```
})

// Exit on error
const db = mongoose.connection.on('error', err => {
    console.error(err);
    process.exit(1)
})

// Log to console once the database is open
db.once('open', async () => {
    console.log(`Mongo connection started on ${db.host}:${db.port}`)
})

require('./author')
```

2. We also need to modify app.js by adding `require('./models')`.

Setting up a mongoose schema and model

Now that we have all the prerequisites set up and installed, it's finally time to start using a MongoDB database instead of the static array which was previously defined in `models/peopleModel.js`.

1. We need to declare a schema for the list of authors.
2. Add a `models/author.js` file, with following contents:

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

const schema = new mongoose.Schema({
    first_name: String,
    last_name: String
})

const Author = mongoose.model('Author', schema)
module.exports = Author
```

We need to define our Mongoose **schema** and then compile to a Mongoose **model** before we can start operating on **documents** (which are instances of the **model**).
Here, we defined a `Author` schema with 2 String fields, `first_name` and `last_name`.

Take a look at the other types and constraints you can place on schemas. For example, you may want **first_name** values to be non-empty rather than null or undefined. This can be achieved with the schema definition: `first_name: { type: String, required: true }`

> Note that we have removed the `id` field because `mongoose` has the `_id` property (a unique id) for every document.

Update the controller to use the database

Having defined the Mongoose model with which we can run database queries, we can now update the controller functions to interact with the database.

1. We will need to make the following changes to `controllers/peopleController.js` to interact with the model:

```
const Author = require('../models/author')

const getAllPeopleData = async (req, res, next) => {
    try {
        const authors = await Author.find().lean()
        return res.render('allData', { data: authors })
    } catch (err) {
        return next(err)
    }
}

const getDataById = async(req, res, next) => {
    try {
        const author = await Author.findById(req.params.author_id).lean()
        if (!author) {
            // no author found in database
            return res.sendStatus(404)
        }
        // found person
        return res.render('oneData', { oneItem: author })
    } catch (err) {
        return next(err)
    }
}

module.exports = {
    getAllPeopleData,
    getDataById,
}
```

2. If you're using your own solution from previous weeks and have a complete implementation of the add author challenge, comment out the part where you're pushing to the static array for now.
3. Edit your `peopleRouter.js` file so that the GET route for individual authors refers to ":author_id" instead of ":id".
4. Edit `allData.hbs` and `oneData.hbs` to use `_id` instead of `id` and remove the `id` field from `allData.hbs`.
   At this point, your app should work locally on your dev machine.

Redeploy the app

1. Update your git repository with these changes (add, commit).
2. Now deploy the updated app on Heroku using the following command: `git push heroku`
3. Open the new app in the browser and test it:
   - `heroku open`
4. If your app is working then you should be able to visit `/people` and see John Smith which we added earlier.

Implementing the other controller functionality

1. Implement or fix the add author functionality.
2. Add functionality (form, route and controller function) to update author first name/last name.

This is left for you as a challenge!

Resources

- mongoose Schemas
- mongoose Model