Milestone 2: Heroku Deployment and Database Setup

Overview

This milestone will require you to configure your application for deployment to Heroku, adding support for database connections, database migrations, and a testing route. In this process, you will install and configure multiple node packages, and begin to work with your express application.

Specification

Create Your Application

We completed this step in class; refer to slide deck 4, express.js.

Nodemon

Set up nodemon to provide a better development experience:

```
npm i --save-dev nodemon
```

I typically add to the scripts section of `package.json` to allow me a quick way to start the application in development mode (note that the start script is required by Heroku, and should not be changed):

```
"start:dev": “NODE_ENV=development DEBUG=APP_NAME:* nodemon ./bin/www"
```

Note that students on Windows machines may need to set their environment variables differently - students from previous semesters have reported that this works:

```
"start:dev": "SET \"NODE_ENV=development\" SET DEBUG=myapp:* & nodemon ./bin/www”
```

Your app can now be started with `npm run start:dev` at the command line. The `NODE_ENV` environment variable is used in express, and later in `dotenv` (next section), to do things like print debug information and choose which environment variables to load.

Dotenv
Set up dotenv to enable user specific environment variables in your development environment. Dotenv makes use of a `.env` file that stores key/value pairs of environment variables. This file must not be checked into the repository, as we do not want to expose potentially sensitive password strings in our repository - we accomplish this by adding the `.env` file to `.gitignore`:

```
npm i --save dotenv
echo ".env" >> .gitignore
touch .env
```

We will make use of the `.env` file later on in our setup.

To make use the environment variables defined in the `.env` file in our development environment, we add the following to `app.js` (towards the top of the file, before the app is instantiated):

```
if(process.env.NODE_ENV === 'development') {
  require("dotenv").config();
}
```

This takes care of loading the environment variables defined in `.env` whenever the `NODE_ENV` is `development` (Heroku sets `NODE_ENV` to `production` in the deploy environment).

Database Setup (pg-promise)
Start by creating your database (I usually name it after my app):

```
createdb DATABASE_NAME
```

Install the pg-promise library:

```
npm i --save pg-promise
```

I encourage you to read about best practices for setting up your database connection in your application. A straightforward way to do so is to add `db/index.js` to your application, with the following content:

```
const pgp = require('pg-promise')();
const connection = pgp(process.env.DATABASE_URL);

module.exports = connection;
```

Understand what this code does by reading the documentation! We will make use of this connection in our test route, later.

Now that we have created the database, we can add the `DATABASE_URL` environment variable to our `.env` file:

```
echo DATABASE_URL=postgres://`whoami`@localhost:5432/DATABASE_NAME >> .env
```

Migration Setup (sequelize-cli)
We want to be able to handle versioning of our database schema without a lot of work. Migration frameworks are built to allow us to write code to handle schema changes. For this setup, we will be using the Sequelize CLI (which also serves as an ORM, but we will not explore these features for this milestone).

Install the sequelize and sequelize CLI packages, and then initialize:

```
npm i --save sequelize
npm install --save sequelize-cli
npx sequelize init
```

The initialization step prepares our project to leverage the sequelize CLI to manage database migrations (note that binaries are sometimes installed with npm packages, and they can be found in the `node_modules/.bin` folder - the scripts in `package.json` will look in this folder for executables, so you need not provide the complete path in `package.json` scripts).

One of the files created in the initialization process is config/config.json. We will need to change this to config/config.js in order to be able to use our environment variables. After renaming, replace the contents with the following:

```
require('dotenv').config();

module.exports = {
  "development": {
    "use_env_variable": "DATABASE_URL",
    "dialect": "postgres"
  },
  "test": {
    "use_env_variable": "DATABASE_URL",
    "dialect": "postgres"
  },
  "production": {
    "use_env_variable": "DATABASE_URL",
    "dialect": "postgres"
  }
}
```

This will load our environment variables prior to any of the sequelize CLI commands running, and tells the CLI to look for database connection information in the `DATABASE_URL` environment variable.
I add the following scripts to package.json so that I do not need to remember the command line instructions and parameters for sequelize (though you should familiarize yourself with these through the documentation):

```
"db:create:migration": "npx sequelize migration:generate --name ",
"db:migrate": "npx sequelize db:migrate",
"db:rollback": "npx sequelize db:migrate:undo”
```

The first script allows me to generate new migrations with the following command:

```
npm run db:create:migration MIGRATION_NAME
```

The second command is used to apply any migrations that have not yet been applied to our database:

```
npm run db:migrate
```

The third command is used to revert the most recent migration (there are other options to revert, please see the documentation for more details):

```
npm run db:rollback
```

We can now create our first migration:

```
npm run db:create:migration first-migration
```

This creates a new migration file in the migrations/ folder. Add the following contents (and read the documentation so that you understand what this is doing!!):

```
'use strict';

module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.createTable(
      'test_table',
      {
        id: {
          type: Sequelize.INTEGER,
          primaryKey: true,
          autoIncrement: true
        },
        createdAt: {
          type: Sequelize.DATE,
          defaultValue: Sequelize.literal('NOW()'),
          allowNull: false
        },
        testString: {
          type:Sequelize.STRING,
          allowNull: false
        }
      }
    );
  },

  down: (queryInterface, Sequelize) => {
    return queryInterface.dropTable('test_table');
  }
};
```

This tells the migration framework that, when this migration is run, it should create a table named `test_table`, with the fields `id`, `createdAt`, and `testString`. Read the documentation for additional types and options for table creation.

Now, run the migration:
`npm run db:migrate`

I encourage you to open the psql shell and explore the definition of the table that has been created!

Adding a tests route
You will now add a route to your application, that will be accessed at `/tests`, that will allow us to verify that our database connection is working. We have discussed how to add a route in class, and you can look at the contents of `app.js` for examples.

The content of the route file will be the following:

```
const express = require("express");
const router = express.Router();
const db = require('../db');

router.get("/", (request, response) => {
  db.any(`INSERT INTO test_table ("testString") VALUES ('Hello at ${Date.now()}')`)
    .then( _ => db.any(`SELECT * FROM test_table`) )
    .then( results => response.json( results ) )
    .catch( error => {
      console.log( error )
      response.json({ error })
    })
});

module.exports = router;
```

Note that we are using the database connection module that we created in a previous step to provide database access to this route. Read and understand this code!

Start your application and browse to `localhost:3000/tests`. Refresh the page a few times.

Deploying to Heroku

You will need a heroku account, and to have installed the heroku CLI. Once completed, you can run heroku create in your applications root directory in order to create an application. Only one person on the team needs to do this!

Currently, the heroku environment does not have a database for your application. To set one up:

1. Go to your heroku dashboard, and select your application
2. Select the resources tab
3. Add the Heroku Postgres add-on (the free version should be fine)
4. Go to the settings tab and select reveal config variables - you should see that a DATABASE_URL has been added to your production environment.

We now need to tell heroku to run our migrations every time we deploy. We can add a `postinstall` script that will be executed by heroku after every deployment:

```
“postinstall”: “npx sequelize db:migrate”
```

Now, deploy your application:

```
git push heroku master
```

Browse to your application (or use the heroku open command from your shell), and verify that your application is connecting to the database.