WEB322 Assignment 5

Submission Deadline:

Friday, November 17th, 2023 @ 11:59 PM

Assessment Weight:

9% of your final course Grade

Objective:

Build upon Assignment 4 by refactoring our code to use a Postgres database to manage our Lego Data, as well as enable the creation, modification and deletion of Lego Sets in the collection.

If you require a *clean version* of Assignment 4 to begin this assignment, please email your professor.

NOTE: Please reference the sample: https://wptf-a5-sample.cyclic.app when creating your solution. Once again, the UI does not have to match exactly, but this will help you determine which elements / syntax should be on each page. You may copy any HTML / CSS code from here if it helps with your solution.

Additionally, since this sample is shared with all students in the class, please **do not add** any content to the Lego collection that may be considered harmful or disrespectful to other students.

Part 1: Connecting to the DB & Adding Existing Sets

Since the major focus of this assignment will be refactoring our code to use a Postgres database, let's begin with this. Follow the course notes PostgreSQL (Postgres) to set up a database on https://neon.tech and record the following information.

- PGHOST
- PGDATABASE
- PGUSER
- PGPASSWORD

Now, with your assignment folder open, add the file ".env" in the root of your solution and add the text:

DB_USER="**PGUSER**"
DB_DATABASE="**PGDATABASE**"
DB_PASSWORD="**PGPASSWORD**"
DB_HOST="**PGHOST**"

Where: **PGUSER**, **PGDATABASE**, **PGPASSWORD** and **PGHOST** are the values that you recorded from Neon.tech (above)

To actually connect to the database and use the .env file, we must install the Sequelize,pg / pg-hstore and dotenv modules from NPM:

npm install sequelize pg pg-hstore dotenv

Finally, before we write our Sequelize code, open the file: /modules/legoSets.js and add the "dotenv" module at the top using the code:

require('dotenv').config();

This will allow us to access the DB_USER, DB_DATABASE, etc. values from the ".env" file using the "process.env" syntax, ie: **process.env.DB_USER**, **process.env.DB_DATABASE** etc.

Beneath this line, add the code to include the "sequelize" module:

const Sequelize = require('sequelize');

and create the "sequelize" object only using **let sequelize = new Sequelize(…);** - see: "Getting Started" in the Relational Database (Postgres) Notes. Be sure to include all of the correct information using **process.env** for "database", "user", "password" and "host".

With our newly created "sequelize" object, we can create the two "models" required for our Assignment according to the below specification (Column Name / Sequelize Data Type):

NOTE: We also wish to disable the **createdAt** and **updatedAt** fields – see: Models (Tables) Introduction

- **Theme**: const Theme = sequelize.define('Theme', { ... });

Column Name	Sequelize DataType
id	Sequelize.INTEGER primaryKey (true) autoincrement (true)
name	Sequelize.STRING

- **Set:** const Set = sequelize.define('Set', { ... });

Column Name	Sequelize DataType
set_num	Sequelize.STRING primaryKey (true)
name	Sequelize.STRING
year	Sequelize.INTEGER
num_parts	Sequelize.INTEGER
theme_id	Sequelize.INTEGER
img_url	Sequelize.STRING

Now that the models are defined, we must create an association between the two:

Set.belongsTo(Theme, {foreignKey: 'theme_id'})

Adding Existing Lego Sets using "BulkCreate"

With our models correctly defined, we have everything that we need to start working with the database. To ensure that our existing data is inserted into our new "Themes" and "Sets" tables, copy the code from here:

https://pat-crawford-sdds.netlify.app/shared/fall-2023/WEB322/A5/bulkInsert.txt

and insert it at the bottom of the **/modules/legoSets.js** file (beneath all module.exports)

(**NOTE:** this code snippet assumes that you have the below code from Assignment 3 still in place):

```
const setData = require("../data/setData");
const themeData = require("../data/themeData");
```

With the code snippet from the above URL in place, open the integrated terminal and execute the command to run it:

node modules/legoSets.js

This should show a big wall of text in the console, followed by "data inserted successfully"!

Part 2: Refactoring Existing Code to use Sequelize

Now that all of our sets exist on the database, we can refactor our existing code in the **legoSets.js** module to retrieve them. This can be done by following the below steps:

1. **Delete** the above code snippet to "bulkInsert" (we no longer need it, now that our data is available in the database)
2. **Delete** the code to read the JSON files / initialize an empty "sets" array:

   ```
   const setData = require("../data/setData");
   const themeData = require("../data/themeData");
   let sets = [];
   ```

3. **Change** your code in "initialize" to instead invoke **sequelize.sync()**. If sequelize.sync() resolves successfully, then we can resolve the returned Promise, otherwise reject the returned Promise with the error.
4. **Change** your code in the "getAllSets()" function to instead use the "Set" model (defined above) to resolve the returned Promise with all returned sets (see: Operations (CRUD) Reference).

   NOTE: Do not forget the option **include: [Theme]** to include Theme data when invoking "findAll".

5. **Change** your code in the "getSetByNum(setNum)" function to instead use the "Set" model (defined above) to resolve the returned Promise with a single set whose set_num value matches the "setNum" parameter. As before, if no set was found, reject the Promise with an error, ie: "Unable to find requested set"

   NOTE: Do not forget the option **include: [Theme]** to include Theme data when invoking "findAll". Also, remember to resolve with the **first** element of the returned array ([0]) to return a single object, as "findAll" always returns an array

6. **Change** your code in the "getSetsByTheme(theme)" function to instead use the "Set" model (defined above) to resolve the returned Promise with all the returned sets whose "Theme.name" property contains the string in

the "theme" parameter. This will involve a more complicated "where" clause, ie:

```
Set.findAll({include: [Theme], where: {
  '$Theme.name$': {
    [Sequelize.Op.iLike]: `%${theme}%`
  }
}});
```

As before, if no sets were found, reject the Promise with an error, ie: "Unable to find requested sets"

NOTE: We have once again included the option **include: [Theme]** to include Theme Data.

7. Look through your .ejs files for "set.theme" and instead replace it with: "set.Theme.name". This is because when "including" the Theme model in our above functions, we have added a full "Theme" object to the result objects (set / sets).

8. Test your code by running the usual **node server.js** – it should work exactly as before!

Part 3: Adding New Sets

Since we are now using a database to manage our data, instead of JSON file(s), the next logical step is to enable users to Create / Update and Delete set data. To begin, we will first create the logic / UI for creating sets and will focus on editing and deleting in the following steps.

Creating the Form

To begin, we should create a simple UI with a form according to the following specification

- This should be in a new file under "views", ie "/views/addSet.ejs"
- It should have some kind of title / header / hero, etc. text to match the other views, ie: "Add Set"
- It should render the "navbar" partial with a "page" value of "/lego/addset", ie:
  <%- include('partials/navbar', {page: '/lego/addSet' }) %>
- It must have the following form controls and submit using "POST" to "/lego/addSet" (to be created later)
  NOTE: The HTML in the sample code may be used here to help render the form (if you wish)
  - **name**
    - input type="text"
    - required
  - **year**
    - input type="number"
    - required
  - **num_parts**
    - input type="number"
    - required

- **img_url**
  - input type="url"
  - required
- **theme_id**
  - select
  - required
  - each option must be a "theme" (added to the view later), ie:

```
<% themes.forEach(theme=>{ %>
  <option value="<%= theme.id %>">
    <%= theme.name %>
  </option>
<% }) %>
```

- **set_num**
  - input type="text
  - required
- **Submit Button**

NOTE: Do not forget to run the command **npm run tw:build** after creating the form, as new CSS was likely used.

Updating the Navbar Partial

As you have noticed from the above steps, a small update is required to our navbar to support linking to the view & highlighting the navbar item. To achieve this, add the following navbar item where appropriate (ie: in the regular & responsive navbar HTML elements)

```
<li><a class="<%= (page == "/lego/addSet") ? 'active' : '' %>" href="/lego/addSet">Add to Collection</a></li>
```

Adding a New View: "500.ejs"

Since it's possible that we may encounter database errors, we should have some kind of "500" error message to show the user instead of rendering a regular view. To get started, make a copy of your "404.ejs" file and update it to show the text "500" as well as any other cosmetic updates you would like to use.

Creating the routes in server.js

To correctly serve the "/lego/addSet" view and process the form, two routes are required in your server.js code (below).

Additionally, since our application will be using urlencoded form data, the "**express.urlencoded({extended:true})**" middleware should be added

- GET /lego/addSet

  This route must make a request to a Promise-based "getAllThemes()" function (to be added later in the legoSets.js module)

Once the Promise has resolved with the themes, the "addSet" view must be rendered with them, ie: **res.render("addSet", { themes: themeData })**;

- POST /lego/addSet

  This route must make a request to a Promise-based "addSet(setData)" function (to be added later in the legoSets.js module), and provide the data in **req.body** as the "setData" parameter.

  Once the Promise has resolved successfully, redirect the user to the "/lego/sets" route.

  If an error was encountered, instead render the new "500" view with an appropriate message, ie:

  res.render("500", { message: `I'm sorry, but we have encountered the following error: ${err}` });

  NOTE: we do not explicitly set the "500" status code here, as it causes unexpected behaviour on Cyclic.

Adding new functionality to legoSets.js module

In our new routes, we made some assumptions about upcoming functionality to be added in the "legoSets.js" module, specifically: "addSet(setData)" and "getAllThemes()". To complete the functionality to add new sets, let's create these now:

NOTE: do not forget to use "module.exports" to make these functions available to server.js

- addSet(setData)

  This function must return a Promise that resolves once a set has been created, or rejects if there was an error.

  It uses the "Set" model to create a new Set with the data from the "setData" parameter. Once this function has resolved successfully, resolve the Promise returned by the addSet(setData) function without any data.

  However, if the function did not resolve successfully, reject the Promise returned by the addSet(setData) function with the message from the *first* error, ie: **err.errors[0].message** (this will provide a more human-readable error message)

- getAllThemes()

  This function must return a Promise that resolves with all the themes in the database. This can be accomplished using the "Theme" model to return all of the themes in the database

Test the Server

We should now be able to add new sets to our collection. Additionally, if we accidentally create a set with a duplicate set_num, our users will see the "500" status code.

Part 4: Editing Existing Sets

In addition to allowing users to **add** sets, we should also let them **edit** existing sets. Let's try to follow the same development methodology used when creating the functionality for adding new sets. This means starting with the view:

- This should be in a new file under "views", ie "/views/editSet.ejs"
- It should have some kind of title / header / hero, etc. text to match the other views, ie: "Edit Set" followed by the **set name**, ie **<%= set.name %>**
- It should render the "navbar" partial with an empty "page" value "" (since this page will not be represented in the navbar), ie: **<%- include('partials/navbar', {page: '' }) %>**
- It must have the exact form controls as the "addSet" view, however the values must be populated with data from a "set" object, passed to the view. For each of the controls, this will simply mean setting a "value" attribute, ie:

  value="<%= set.name %>" or **value="<%= set.year %>"**

  However, things get more complicated when we wish to correctly set the "selected" attribute of the "theme_id" select control, ie:

  ```
  <% themes.forEach(theme=>{ %>
    <option <%= (set.theme_id == theme.id) ? "selected" : "" %> value="<%= theme.id %>">
      <%= theme.name %>
    </option>
  <% }) %>
  ```

  (once again this assumes that a "themes" collection will be added to the view along with a "set" object later)
- Ensure that the "set_num" text field is set to **readonly** (you may wish to optionally give it the "cursor-not-allowed" class (see: https://tailwindcss.com/docs/cursor)
- Finally, we should change the "action" attribute on the <form> control to submit the form to **"/lego/editSet"** as well as change the submit button text to something like "Update Set"

Creating the routes in server.js

To correctly serve the "/lego/editSet" view and process the form, two routes are required in your server.js code (below).

- GET "/lego/editSet/:num"

  This route must make a request to the Promise-based "getSetByNum(setNum)" function with the value from the **num** route parameter as "setNum" in order to retrieve the correct set.

  It also must make a request to the Promise-based "getAllThemes() function in order to retrieve an array of "theme" data

  Once the Promises have resolved with the theme data and the set data the "edit" view must be rendered with them, ie:

res.render("edit", { themes: themeData, set: setData });

However, if there was a problem obtaining the set or collection of teams (ie: the Promises were rejected), instead render the "404" view with an appropriate message, ie:

res.status(404).render("404", { message: err });

- POST "/lego/editSet"

  This route must make a request to a Promise-based "editSet(set_num, setData)" function (to be added later in the legoSets.js module), and provide the data in **req.body.set_num** as the "set_num" parameter and **req.body** as the "setData" parameter

  Once the Promise has resolved successfully, redirect the user to the "/lego/sets" route.

  If an error was encountered, instead render the "500" view with an appropriate message, ie:

  res.render("500", { message: `I'm sorry, but we have encountered the following error: ${err}` });

  NOTE: we do not explicitly set the "500" status code here, as it causes unexpected behaviour on Cyclic.

Adding new functionality to legoSets.js module

In our new routes, we made some assumptions about upcoming functionality to be added in the "legoSets.js" module, specifically: "editSet(set_num, setData)". To complete the functionality to edit sets, let's create this now:

NOTE: do not forget to use "module.exports" to make the function available to server.js

- editSet(set_num, setData)

  This function must return a Promise that resolves once a set has been updated, or rejects if there was an error.

  It uses the "Set" model to update an existing set that has a "set_num" property that matches the "set_num" parameter to the function, with the data from the "setData" parameter. Once this function has resolved successfully, resolve the Promise returned by the editSet(set_num, setData) function without any data.

  However, if the function did not resolve successfully, reject the Promise returned by the editSet(set_num,setData) function with the message from the *first* error, ie: **err.errors[0].message** (this will provide a more human-readable error message)

Adding an Edit button

At the moment, we should be able to edit any of our lego sets by going directly to the route in the browser, ie: "/lego/editSet/001-1" However, it makes more sense from a usability perspective to allow users the ability to navigate to this route using the UI.

To achieve this, add an "edit" link (rendered using the tailwind button classes, ie: "btn btn-success", etc) in the **set.ejs** template that links to: "/lego/editSet/**set_num**" where **set_num** is the "set_num" value of the current set, ie: **<%= set.set_num %>**

Test the Server

We should now be able to edit sets to our collection!

Part 5: Deleting Existing Sets

The final piece of logic that we will implement in this assignment is to enable users to remove (delete) an existing set from the database. To achieve this, we must add an additional function on our **legoSets.js** module:

NOTE: do not forget to use "module.exports" to make the function available to server.js

- deleteSet(set_num)

  This function must return a Promise that resolves once a set has been deleted, or rejects if there was an error.

  It uses the "Set" model to delete an existing set that has a "set_num" property that matches the "set_num" parameter to the function. Once this function has resolved successfully, resolve the Promise returned by the deleteSet(set_num) function without any data.

  However, if the function did not resolve successfully, reject the Promise returned by the editSet(set_num,setData) function with the message from the *first* error, ie: **err.errors[0].message** (this will provide a more human-readable error message)

With this function in place, we can now write a new route in server.js :

- GET "/lego/deleteSet/:num"

  This route must make a request to the Promise-based "deleteSet(setNum)" function with the value from the **num** route parameter as "setNum" in order to delete the correct set.

  Once the Promise has resolved successfully, redirect the user to the "/lego/sets" route.

  If an error was encountered, instead render the "500" view with an appropriate message, ie:

  res.render("500", { message: `I'm sorry, but we have encountered the following error: ${err}` });

  NOTE: we do not explicitly set the "500" status code here, as it causes unexpected behaviour on Cyclic.

Finally, let's add a button in our UI to enable this functionality by linking to the above route for the correct set. One place where it makes sense is in our "**editSet.ejs**" view. Here, we give the user the choice to either update the set or delete it.

To achieve this, add a "Delete Set" link (rendered using the tailwind button classes, ie: "btn btn-error", etc) that links to: "/lego/deleteSet/**set_num**" where **set_num** is the "set_num" value of the current set, ie: **<%= set.set_num %>**

Test the Server

We should now be able to remove sets from our collection!

Part 6: Updating your Deployment

Finally, once you have tested your site locally and are happy with it, you're ready to update your deployed site by pushing your latest changes to GitHub. However, before you do that, you should add **.env** to your **.gitignore** file to prevent your environment variables from being included:

File: .gitignore

```
node_modules
.env
```

Additionally, you should add those environment variable values to your app on Cyclic by logging in and proceeding to the dashboard for your app and navigating to the "Variables" tab. From there you can add or remove environment variables

For more information on setting up environment variables on Cyclic, see: https://docs.cyclic.sh/concepts/env_vars

Once this is complete, "push" your code to GitHub to update your deployment

Assignment Submission:

- Add the following declaration at the top of your **server.js** file:

```
/*********************************************************************************
* WEB322 – Assignment 05
*
* I declare that this assignment is my own work in accordance with Seneca's
* Academic Integrity Policy:
*
* https://www.senecacollege.ca/about/policies/academic-integrity-policy.html
*
* Name: _____ Student ID: _____ Date: _____
*
* Published URL: _____
*
*********************************************************************************/
```

- Compress (.zip) your assignment folder and submit the .zip file to My.Seneca under **Assignments** -> **Assignment 5**

Important Note:

- **NO LATE SUBMISSIONS** for assignments. Late assignment submissions will not be accepted and will receive a **grade of zero (0)**.
- Submitted assignments must run locally, ie: start up errors causing the assignment/app to fail on startup will result in a **grade of zero (0)** for the assignment.