

WEB322 Assignment 5

Submission Deadline:

Friday, November 18th, 2022 @ 11:59 PM

Assessment Weight:

9% of your final course Grade

Objective:

Work with a Postgres data source on the server and practice refactoring an application. You can view a sample solution online here: <https://web322-a5-sample.herokuapp.com>

Specification:

NOTE: If you are unable to start this assignment because Assignment 4 was incomplete - email your professor for a clean version of the Assignment 4 files to start from (effectively removing any custom CSS or text added to your solution).

Getting Started:

Before we get started, we must add a new Postgres instance on our web-322 app in Cyclic. Follow the instructions for setting up a new Postgres instance on elephantsql through here: <https://web322.ca/notes/week07>. You can verify that connection to your new database works by following the PgAdmin steps directly under the steps above.

Getting Started - Cleaning the solution

- To begin: open your Assignment 4 folder in Visual Studio Code
- In this assignment, we will no longer be reading the files from the "data" folder, so remove this folder from the solution
- Inside your **blog-service.js** module, **delete** any code that is **not** a **module.exports** function (ie: global variables, & "require" statements)
- Inside **every single module.exports** function (ie: module.exports.initialize(), module.exports.getAllPosts, module.exports.getPostsByCategory, etc.), remove all of the code and replace it with a return call to an "empty" promise that invokes reject() - (Note: we will be updating these later), ie:

```
return new Promise((resolve, reject) => {  
  reject();  
});
```

Installing "sequelize"

- Open the "integrated terminal" in Visual Studio Code and enter the commands to install the following modules:
 - sequelize

- pg
- pg-hstore
- At the top of your **blog-service.js** module, add the lines:
 - **const** Sequelize = require('sequelize');
 - **var** sequelize = **new** Sequelize('database', 'user', 'password', {
 host: 'host',
 dialect: 'postgres',
 port: 5432,
 dialectOptions: {
 ssl: { rejectUnauthorized: **false** }
 },
 query: { raw: true }
 });
 - **NOTE:** for the above code to work, replace *'database'*, *'user'*, *'password'* and *'host'* with the credentials that you saved when creating your new elephantsql Postgres Database (above)

Another Helper: formatDate

Part of the update to assignment 4 includes using real date values instead of strings. To help us keep our formatting consistent in the views from earlier assignments, you can use the following "formatDate" express-handlebars helper:

```
formatDate: function(dateObj){
  let year = dateObj.getFullYear();
  let month = (dateObj.getMonth() + 1).toString();
  let day = dateObj.getDate().toString();
  return `${year}-${month.padStart(2, '0')}-${day.padStart(2, '0')}`;
}
```

You can use it in the following way within your views:

Instead of writing something like {{postDate}}, you can instead write **{{#formatDate postDate}}{}/formatDate}}**

Creating Data Models

- Inside your **blog-service.js** module (before your module.exports functions), define the following 2 data models and their relationship (**HINT:** See "Models (Tables) Introduction" in the [Week 7 Notes](#) for examples)
- Post

Column Name	Sequelize DataType
body	Sequelize.TEXT
title	Sequelize.STRING
postDate	Sequelize.DATE
featureImage	Sequelize.STRING

published	Sequelize.BOOLEAN
-----------	-------------------

- Category

Column Name	Sequelize DataType
category	Sequelize.STRING

- **belongsTo** Relationship

Since a post belongs to a specific category, we must define a relationship between Posts and Categories, specifically:

Post.belongsTo(Category, {foreignKey: 'category'});

This will ensure that our Post model gets a "category" column that will act as a foreign key to the Category model. When a Category is deleted, any associated Posts will have a "null" value set to their "category" foreign key.

Update Existing blog-service.js functions

Now that we have Sequelize set up properly, and our "Post" and "Category" models defined, we can use all of the Sequelize operations, discussed in the [Week 7 Notes](#) to update our blog-service.js to work with the database:

initialize()

- This function will invoke the [sequelize.sync\(\)](#) function, which will ensure that we can connect to the DB and that our Post and Category models are represented in the database as tables.
- If the **sync()** operation resolved **successfully**, invoke the **resolve** method for the promise to communicate back to server.js that the operation was a success.
- If there was an error at any time during this process, invoke the **reject** method for the promise and pass an appropriate message, ie: `reject("unable to sync the database")`.

getAllPosts()

- This function will invoke the [Post.findAll\(\)](#) function
- If the **Post.findAll()** operation resolved **successfully**, invoke the **resolve** method for the promise (with the data) to communicate back to server.js that the operation was a success and to provide the data.
- If there was an error at any time during this process, invoke the **reject** method and pass a meaningful message, ie: "no results returned".

getPostsByCategory()

- This function will invoke the [Post.findAll\(\)](#) function and filter the results by "category" (using the value passed to the function - ie: 1 or 2 or 3 ... etc)
- If the **Post.findAll()** operation resolved **successfully**, invoke the **resolve** method for the promise (with the data) to communicate back to server.js that the operation was a success and to provide the data.
- If there was an error at any time during this process, invoke the **reject** method and pass a meaningful message, ie: "no results returned".

getPostsByMinDate()

- This function will invoke the [Post.findAll\(\)](#) function and filter the results to only include posts with the postDate value greater than or equal to the minDateStr (using the value passed to the function - ie: "2020-10-1" ... etc)
 - **NOTE:** This can be accomplished using one of the many operators (see: "Operators" in: <https://sequelize.org/v5/manual/querying.html>), ie:

```
const { gte } = Sequelize.Op;

Post.findAll({
  where: {
    postDate: {
      [gte]: new Date(minDateStr)
    }
  }
})
```

- If the **Post.findAll()** operation resolved **successfully**, invoke the **resolve** method for the promise (with the data) to communicate back to server.js that the operation was a success and to provide the data.
- If there was an error at any time during this process, invoke the **reject** method and pass a meaningful message, ie: "no results returned".

getPostById()

- This function will invoke the [Post.findAll\(\)](#) function and filter the results by "id" (using the value passed to the function - ie: 1 or 2 or 3 ... etc)
- If the **Post.findAll()** operation resolved **successfully**, invoke the **resolve** method for the promise (with the data[0], ie: only provide the first object) to communicate back to server.js that the operation was a success and to provide the data.
- If there was an error at any time during this process, invoke the **reject** method and pass a meaningful message, ie: "no results returned".

addPost()

- Before we can work with **postData** correctly, we must once again make sure the published property is set properly. Recall: to ensure that this value is set correctly, before you start working with the postData object, add the line:

- `postData.published = (postData.published) ? true : false;`
- Additionally, we must ensure that any blank values ("") for properties are set to null. For example, if the user didn't enter a Title (causing `postData.title` to be ""), this needs to be set to null (ie: `postData.title = null`). You can iterate over every property in an object (to check for empty values and replace them with null) using a [for...in loop](#).
- Finally, we must assign a value for `postDate`. This will simply be the current date, ie `"new Date()"`
- Now that the `published` property is explicitly set (true or false), all of the remaining "" are replaced with null, and the `"postDate"` value is set we can invoke the [Post.create\(\)](#) function
- If the **Post.create()** operation resolved **successfully**, invoke the **resolve** method for the promise to communicate back to `server.js` that the operation was a success.
- If there was an error at any time during this process, invoke the **reject** method and pass a meaningful message, ie: `"unable to create post"`.

getPublishedPosts()

- This function will invoke the [Post.findAll\(\)](#) function and filter the results by `"published"` (using the value `true`)
- If the **Post.findAll()** operation resolved **successfully**, invoke the **resolve** method for the promise (with the data) to communicate back to `server.js` that the operation was a success and to provide the data.
- If there was an error at any time during this process, invoke the **reject** method and pass a meaningful message, ie: `"no results returned"`.

getPublishedPostsByCategory()

- This function will invoke the [Post.findAll\(\)](#) function and filter the results by `"published"` and `"category"` (using the value `true` for `"published"` and the value passed to the function - ie: 1 or 2 or 3 ... etc for `"category"`)
- If the **Post.findAll()** operation resolved **successfully**, invoke the **resolve** method for the promise (with the data) to communicate back to `server.js` that the operation was a success and to provide the data.
- If there was an error at any time during this process, invoke the **reject** method and pass a meaningful message, ie: `"no results returned"`.

getCategories()

- This function will invoke the [Category.findAll\(\)](#) function
- If the **Category.findAll()** operation resolved **successfully**, invoke the **resolve** method for the promise (with the data) to communicate back to `server.js` that the operation was a success and to provide the data.
- If there was an error at any time during this process, invoke the **reject** method and pass a meaningful message, ie: `"no results returned"`.

Updating the Navbar & Existing views (.hbs)

If we test the server now and simply navigate between the pages, we will see that everything still works, except we no longer have any posts in our "Posts" or "Blog" views, and no categories within our "Categories" view. This is to be expected (since there is nothing in the database), however we are not seeing any error messages (just empty tables). To solve this, we must update our server.js file:

- /posts route
 - Where we would normally render the "posts" view with data
 - ie: `res.render("posts", {posts:data});`
 - we must place a condition there first so that it will only render "posts" if `data.length > 0`. Otherwise, render the page with an error message,
 - ie: `res.render("posts",{ message: "no results" });`
 - If we test the server now, we should see our "no results" message in the /posts route
 - **NOTE:** We must still show messages if the promise(s) are rejected, as before
- /categories route
 - Using the same logic as above (for the /posts route) update the /categories route as well
 - If we test the server now, we should see our "no results" message in the /categories route
 - **NOTE:** We must still show an error message if the promise is rejected, as before

For this assignment, we will also be moving the "add Post" link and inserting it into the "posts" view, as well as writing code to handle adding a new Category

- "add Post"
 - Remove the link (`{{#navLink}}` ... `{{/navLink}}`) from the "navbar-nav" element inside the main.hbs file
 - Inside the "posts.hbs" view (Inside the `<h2>Posts</h2>` element), add the below code to create a "button" that links to the `"/posts/add"` route:
 - `Add Post`
- "add Category"
 - You will notice that currently, we have no way of adding a new category. However, while we're adding our "add" buttons, it makes sense to create an "add Category" button as well (we'll code the route and blog service function later in this assignment).
 - Inside the "categories.hbs" view (Inside the `<h2>Categories</h2>` element), add the below code to create a "button" that links to the `"/categories/add"` route:
 - `Add Category`

Adding new blog-service.js functions

So far, all our blog-service functions have focused primarily on fetching Post / Category data as well as adding new Posts. If we want to allow our users to add Categories as well, we must add some additional logic to our blog-service.

Additionally, we will also let users **delete** Posts and Categories. To achieve this, the following (promise-based) functions must be added to blog-service.js:

addCategory(categoryData)

- Like addPost(postData), we must ensure that any blank values in **categoryData** are set to null (follow the same procedure)
- Now that all of the "" are replaced with null, we can invoke the [Category.create\(\)](#) function
- If the **Category.create()** operation resolved **successfully**, invoke the **resolve** method for the promise to communicate back to server.js that the operation was a success.
- If there was an error at any time during this process, invoke the **reject** method and pass a meaningful message, ie: "unable to create category"

deleteCategoryById(id)

- The purpose of this method is simply to **"delete"** categories using [Category.destroy\(\)](#) for a specific category by "id". Ensure that this function returns a **promise** and only "resolves" if the Category was deleted ("destroyed"). "Reject" the promise if the "destroy" method encountered an error (was rejected).

deletePostById(id)

- This method is nearly identical to the "deleteCategoryById(id)" function above, only instead of invoking "Category.destroy()" for a specific id, it will instead use [Post.destroy\(\)](#)

Updating Routes (server.js) to Add / Remove Categories & Posts

Now that we have our blog-service up to date to deal with adding / removing post and category data, we need to update our server.js file to expose a few new routes that provide a form for the user to enter data (GET) and for the server to receive data (POST) as well as let the user delete posts / categories by Id.

Additionally, since categories does not require users to upload an image, we should also include the regular `express.urlencoded()` middleware:

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

Once this is complete, add the following routes:

/categories/add

- This **GET** route is very similar to your current "/posts/add" route - only instead of "rendering" the "addPost" view, we will instead set up the route to "render" an "addCategory" view (added later)

/categories/add

- This **POST** route is very similar to the logic inside the "processPost()" function within your current "/post/add" POST route - only instead of calling the addPost() blog-service function, you will instead call your newly created

addCategory() function with the POST data in req.body (**NOTE:** there's also no "featureImage" property that needs to be set)

- Instead of redirecting to /posts when the promise has resolved (using .then()), we will instead redirect to /categories

/categories/delete/:id

- This **GET** route will invoke your newly created **deleteCategoryById(id)** blog-service method. If the function resolved successfully, redirect the user to the **"/categories"** view. If the operation encountered an error, return a **status code of 500** and the plain text: **"Unable to Remove Category / Category not found)"**

/posts/delete/:id

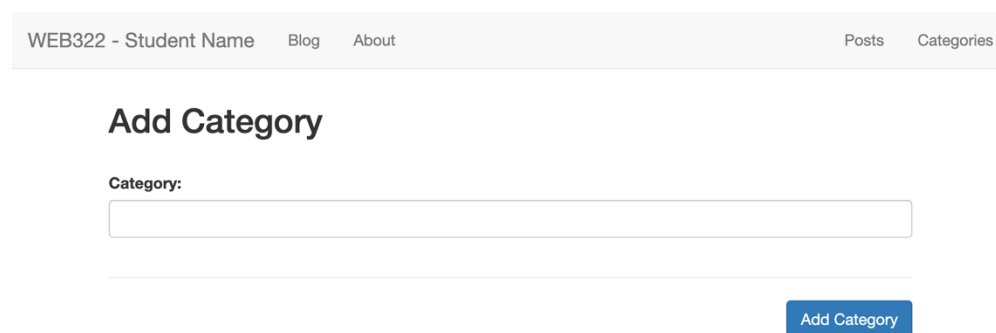
- This **GET** route functions almost exactly the same as the route above, only instead of invoking deleteCategoryById(id), it will instead invoke **deletePostById(id)** and return an appropriate error message if the operation encountered an error

Updating Views to Add & Delete Categories / Posts

In order to provide user interfaces to all of our new functionality, we need to add / modify some views within the "views" directory of our app:

addCategory.hbs

- Fundamentally, this view is nearly identical to the **addPost.hbs** view, however there are a few key changes:
 - The header (<h2>...</h2>) must read "Add Category"
 - The form must submit to "/categories/add" and the "enctype" property can be removed (multipart/form-data no longer required)
 - There must be only one input field (type: "text", name: "category", label: "Category:")
 - **NOTE:** You may wish to add the **autofocus** attribute here, since it is the only form control available to the user
 - The submit button must read "Add Category"
- When complete, your view should appear as:



The screenshot shows a web application header with the text "WEB322 - Student Name" and navigation links "Blog" and "About". On the right side of the header are links "Posts" and "Categories". Below the header is a section titled "Add Category". Under this title, there is a label "Category:" followed by a text input field. At the bottom right of the form is a blue button labeled "Add Category".

categories.hbs

- To enable users to access also delete categories, we need to make one important change to our current categories.hbs file:

- **Add** a "remove" link for every category within in a new column of the table (at the end) - Note: The header for the column should not contain any text (ie: <th></th>). The links in every row should be styled as a button (ie: class="btn btn-danger") with the text "remove" and simply link to the newly created GET route "categories/delete/*categoryId*" where *categoryId* is the category id of the category in the current row.

Once this button is clicked, the category will be deleted and the user will be redirected back to the "/categories" list. (Hint: See the sample solution if you need help with the table formatting)

Updating the "Categories" List when Adding a new Post

Now that users can add new Categories, it makes sense that all of the Categories are available when adding a new Post, should consist of all the current categories in the database (instead of just 1...5). To support this new functionality, we must make a few key changes to the corresponding route & view:

"/posts/add" route

- Since the "addPost" view will now be working with actual Categories, we need to update the route to make a call to our blog-service module to "getCategories".
- Once the **getCategories()** operation has resolved, we **then** "render" the " addPost view (as before), however this time we will and pass in the data from the promise, as "categories", ie: **res.render("addPost", {categories: data});**
- If the getCategories() promise is rejected (using **.catch**), "render" the "addPost" view anyway (as before), however instead of sending the data from the promise, send an empty array for "categories, ie: **res.render("addPost", {categories: []});**

"addPost.hbs" view

- Update the: `<select class="form-control" name="category" id="category">...</select>` element to use the new handlebars code:

```

{{#if categories}}
  <select class="form-control" name="category" id="category">
    <option value="">Select Category</option>
    {{#each categories}}
      <option value="{{id}}">{{category}}</option>
    {{/each}}
  </select>
{{else}}
  <div>No Categories</div>
{{/if}}
```

- Now, if we have any categories in the system, they will show up in our view - otherwise we will show a div element that states "No Categories"

Updating server.js, blog-service.js & posts.hbs to Delete Posts

To make the user-interface more usable, we should allow users to also remove (delete) posts that they no longer wish to be in the system. This will involve:

- Creating a new function (ie: **deletePostById(id)**) in blog-service.js to "**delete**" posts using [Post.destroy\(\)](#) for a specific post. Ensure that this function returns a **promise** and only "resolves" if the Post was deleted ("destroyed"). "Reject" the promise if the "destroy" method encountered an error (was rejected).
- Create a new GET route (ie: **"/posts/delete/:id"**) that will invoke your newly created **deletePostById(id)** blog-service method. If the function resolved successfully, redirect the user to the **"/posts"** view. If the operation encountered an error, return a **status code of 500** and the plain text: **"Unable to Remove Post / Post not found"**
- Lastly, update the **posts.hbs** view to include a "remove" link for every post within in a new column of the table (at the end) - Note: The header for the column should not contain any text. The links in every row should be styled as a button (ie: **class="btn btn-danger"**) with the text **"remove"** and link to the newly created GET route **"post/delete/id"** where **id** is the id of the post in the current row. Once this button is clicked, the post will be deleted and the user will be redirected back to the **"/posts"** list.

Sample Solution

To see a completed version of this app running, visit: <https://web322-a5-sample.herokuapp.com>

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 Academic Policy. No part of this
* assignment has been copied manually or electronically from any other source (including web sites) or
* distributed to other students.
*
* Name: _____ Student ID: _____ Date: _____
*
* Online (Cyclic) Link: _____
*
*****/

```

- Publish your application on Cyclic & test to ensure correctness
- Compress your web322-app folder and Submit your 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)**.
- After the end (11:59PM) of the due date, the assignment submission link on My.Seneca will no longer be available.
- 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.