WEB322 Assignment 5

Submission Deadline:

Friday, March 27th, 2020 @ 11:59 PM

Assessment Weight:

9% of your final course Grade

NOTE:

As I was making this application, there had been a change to handlebars which created issues for working with Postgres's data that sequelize retrieves. I will be providing a copy of the full server.js from my application in the assignment's documents. Please use that. There may be a compatibility with your app as some methods or variables may be different.

Objective:

Work with a Postgres data source on the server and practice refactoring an application. You can view a sample solution online here: https://dry-plateau-13317.herokuapp.com/

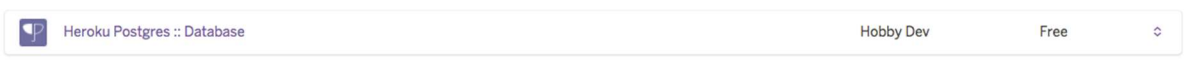
Specification:

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:

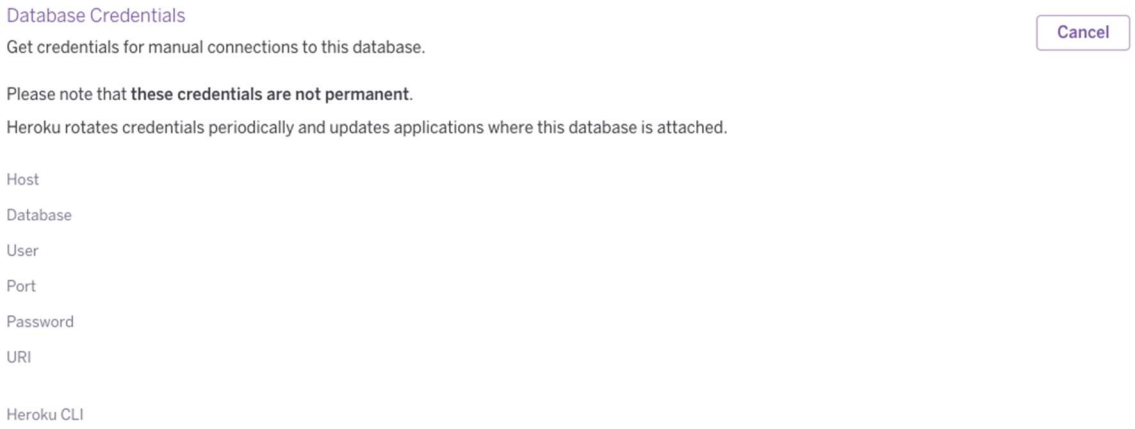
Before we get started, we must add a new Postgres instance on our web-322 app in Heroku:

- Create a new application on heroku
- Navigate to the application from your Heroku dashboard
- Click the "Resources" header and type "Postgres" in the bottom text box labeled: "Add-ons"
- Click the "Heroku Postgres" link that comes up
- This should cause a modal window to appear. Keep the settings the way they are "Hobby Dev --Free" and click "Provision"
- Click on the new Add-on "Heroku Postgres :: Database" link



- In the new page, click the "Settings" link. Here, you will see an "Administration" section. Click the "View Credentials…" button

- Record all of the credentials (ie: Host, Database, User, Port, etc.) - we will be using them to connect to the database:



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 **data-server.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.getAllPeople, module.exports.getPeopleByVin, 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(function (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 **data-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: true
    }
});
```

- **NOTE:** for the above code to work, replace *'database'*, *'user'*, *'password'* and *'host'* with the credentials that you saved when creating your new Heroku Postgres Database (above)

Creating Data Models

- Inside your **data-service.js** module (before your module.exports functions), define the following 3 data models and their relationship (**HINT**: See "Models (Tables) Introduction" in the Week 7 Notes for examples)
- People

Column Name	Sequelize DataType
first_name	Sequelize.STRING
last_name	Sequelize.STRING
phone	Sequelize.STRING
address	Sequelize.STRING
city	Sequelize.STRING

- Car

Column Name	Sequelize DataType
vin	type: Sequelize.STRING unique: true
make	Sequelize.STRING
model	Sequelize.STRING
year	Sequelize.STRING

<- Here we need to add this attribute for this column.
The definition will look like this:
vin : { type: Sequilize.STRING, primaryKey: true, unique : true }

- Store

Column Name	Sequelize DataType
retailer	Sequelize.STRING
phone	Sequelize.STRING
address	Sequelize.STRING
city	Sequelize.STRING

- **hasMany** Relationship

Since a Car can be owned by many people, we must define a relationship between People and Car, specifically:

Car.hasMany(People, {foreignKey: 'vin'});

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

Update Existing data-service.js functions

Now that we have Sequelize set up properly, and our "People", "Car" and "Store" models defined, we can use all of the Sequelize operations, discussed in the Week 7 Notes to update our data-service.js to work with the database:

initialize()

- This function will invoke the sequelize.sync() function, which will ensure that we can connected to the DB and that our People, Car and Store 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").

getAllPeope()

- This function will invoke the People.findAll() function
- If the **Peope.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".

getPeopleByVin(***vin***)

- This function will invoke the People.findAll() function and filter the results by "vin" (using the value passed to the function - ie: the vin number
- If the **People.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".

getPeopleById(id)

- This function will invoke the People.findAll() function and filter the results by "id" (using the value passed to the function - ie: 1 or 2 or 3 … etc
- If the **People.findAll()** operation resolved **successfully**, invoke the **resolve** method for the promise (with an array of only one 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".

getCars()

- This function will invoke the Car.findAll() function
- If the **Car.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 (or no results were returned), invoke the **reject** method and pass a meaningful message, ie: "no results returned".

getCarByVin(***vin***)

- This function will invoke the Car.findAll() function and filter the results by "vin" (using the value passed to the function - ie: the vin number
- If the **Car.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".

getCarByMake(make)

- This function will invoke the Car.findAll() function and filter the results by "make" (using the value passed to the function - ie: the Car Make
- If the **Car.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".

getCarByYear(year)

- This function will invoke the Car.findAll() function and filter the results by "year" (using the value passed to the function - ie: the Car Year
- If the **Car.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".

getStoreByRetailer(retailer)

- This function will invoke the Store.findAll() function and filter the results by "retailer" (using the value passed to the function - ie: the Retailer name
- If the **Store.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".

addPeople(***peopleData***)

- Before we can work with ***peopleData*** correctly, we must ensure that any blank values ("") for properties are set to null. For example, if the user didn't enter an address (causing peopleData.address to be ""), this needs to be set instead to null (ie: peopleData.address = 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.
- Now that we've checked the properties and set them to null if they were "", we can invoke the People.create() function
- If the **People.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 the person".

updatePeople(***peopleData***)

- Like addPeople(peopleData) we must ensure that all of the "" are replaced with null, we can invoke the People.update() function and filter the operation by "id" (ie peopleData.id)
- If the **People.update()** 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 update person".

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 people in our "People" view, and no cars within our "Cars" view. This is to be expected (since there is nothing in the database), however we are not seeing an any messages (just empty tables). To solve this, we must update our server.js file:

- /people route
    - Where we would normally render the "people" view with data
        - ie: res.render("people", {people: data});

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

- Using the same logic as above (for the /people route) update the /stores route as well
- If we test the server now, we should see our "no results" message in the /stores route
- **NOTE**: We must still show an error message if the promise is rejected, as before

For this assignment, we will be moving the "add People" and "add Pirctures" links into their respective pages (ie: "add People" will be moved out of the Navbar and into the "people" view and "add Pictures" will be moved out of the Navbar and into the "pictures" view)

- "add People"
  - Remove this link ({{#navLink}} … {{/navLink}}) from the "navbar-nav" element inside the main.hbs file
  - Inside the "people.hbs" view (Inside the <h2>People</h2> element), add the below code to create a "button" that links to the "/people/add" route:
    - Add New Person
- "add Pictures"
  - Remove this link ({{#navLink}} … {{/navLink}}) from the "navbar-nav" element inside the main.hbs file
  - Inside the "pictures.hbs" view (Inside the <h2>Pictures</h2> element), add the below code to create a "button" that links to the "/pictures/add" route:
    - Add New Picture
- "add Cars"
  - You will notice that currently, we have no way of adding a new car. However, while we're adding our "add" buttons, it makes sense to create an "add Car" button as well (we'll code the route and data service later in this assignment).
  - Inside the "cars.hbs" view (Inside the <h2>Cars</h2> element), add the below code to create a "button" that links to the "/cars/add" route:
    - Add New Car
- "add Stores"
  - You will notice that currently, we have no way of adding a new store. However, while we're adding our "add" buttons, it makes sense to create an "add Store" button as well (we'll code the route and data service later in this assignment).
  - Inside the "stores.hbs" view (Inside the <h2>Stores</h2> element), add the below code to create a "button" that links to the "/stores/add" route:
    - Add New Store

Adding new data-service.js functions for Cars

addCar(***carData***)

- Like addPeople(peopleData) function we must ensure that any blank values in ***carData*** are set to null (follow the same procedure)
- Now that all of the "" are replaced with null, we can invoke the Car.create() function
- If the **Car.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 car"

updateCar(***carData***)

- Like addCar(carData) function we must ensure that any blank values in ***carData*** are set to null (follow the same procedure)
- Now that all of the "" are replaced with null, we can invoke the Car.update() function and filter the operation by "vin" (ie carData.vin) **Note: We are filtering by vin as vin is now the primary key of Cars**
- If the **Car.update()** 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 update car".

getCarByVin(***vin***)

- Similar to the getPeopleById(***id***) function, this function will invoke the Car.findAll() function (instead of People.findAll()) and filter the results by "vin" (using the value passed to the function - ie: 3N1CN7AP6EL561139 etc.
- If the **Car.findAll()** operation resolved **successfully**, invoke the **resolve** method for the promise (with an array of only one 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".

deleteCarByVin(***vin***)

- The purpose of this method is simply to "**delete**" Cars using the Car.destroy() for a specific car by "vin". Ensure that this function returns a **promise** and only "resolves" if the Car was deleted ("destroyed"). "Reject" the promise if the "destroy" method encountered an error (was rejected).

Updating Views to Add / Update & Delete Cars

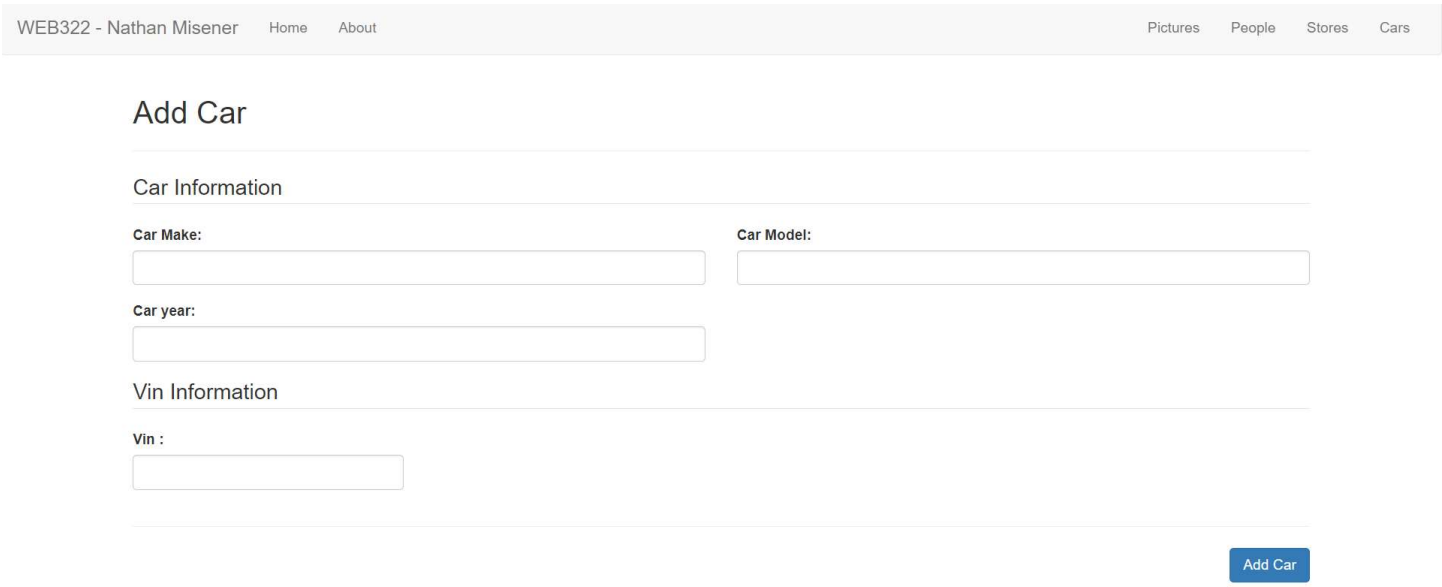
In order to provide user interfaces to all of our new "Car" functionality, we need to add / modify some views within the "views" directory of our app:

addCar.hbs

- Fundamentally, this view is nearly identical to the **addPeople.hbs** view, however there are a few key changes:
    - The header (<h1>…</h1>) must read "Add Car"

- The form must submit to "/car/add"
- There must be 4 input fields (type: "text", name: "vin", label: "Car Vin", **required** (Cannot be null on the form)), (type: "text", name: "make", label: "Car Make"), (type: "text", name: "model", label: "Car Model"),(type: "text", name: "year", label: "Car Year")
- The submit button must read "Add Car"

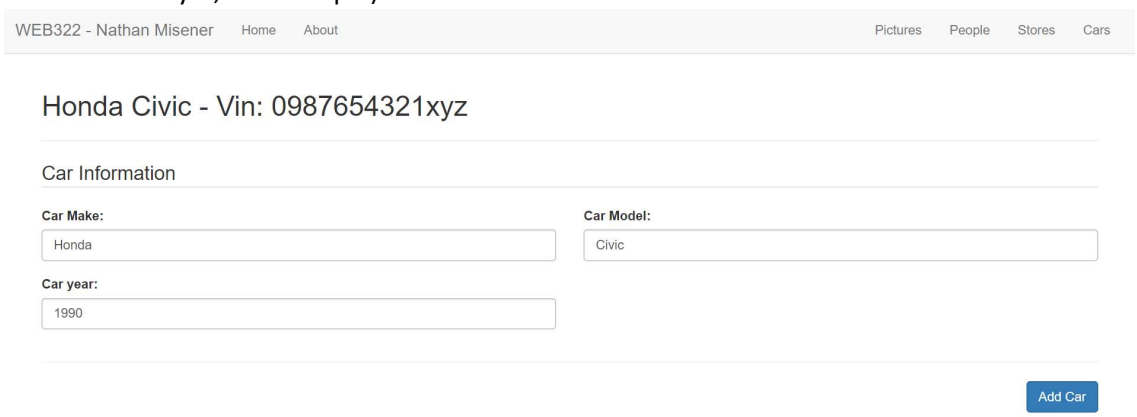
- When complete, your view should appear as:



car.hbs

- Like addCar.hbs, the car.hbs view is very similar to its people counterpart: people.hbs, only with a few key differences:
  - The header (<h1>…</h1>) must read "***make*** - make: ***vin***" where ***make*** and ***vin*** represent the make and vin of the current car.
  - The form must submit to "/car/update"
  - The hidden field must have the properties: name="vin" and value="{{data.vin}}" in order to keep track of exactly which car the user is editing
  - There must be 3 input fields (type: "text", name: "make", label: "Car Make", value="{{data.make}}"), (type: "text", name: "model", label: "Car Model", value="{{data.model}}"), (type: "text", name: "year", label: "Car Year", value="{{data.year}}")
  - The submit button must read "Update Car"

- When complete, your view should appear as the following (if we're currently looking at a newly created "0987654321xyz", for example):



cars.hbs

- Lastly, to enable users to access all of this new functionality, we need to make two important changes to our current cars.hbs file:
  - **Replace** the code:
    <td>{{vin}}</td>
    with
    <td>{{vin}}</td>
  - **Add** a "remove" link for every car within in a new column of the table (at the end) - Note: The header for the column should not contain any text (ie: <hr></hr>). 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 "cars/delete/*vin*" where *vin* is the vin of the car in the current row. Once this button is clicked, the car will be deleted and the user will be redirected back to the "/cars" list. (Hint: See the sample solution if you need help with the table formatting)

Now, users can click on the car's vin if they wish to edit an existing car, the car make if they wish to filter the stores list by make, or "delete" if they wish to remove the car!

Adding new data-service.js functions for stores

So far, all our data-service functions have focused primarily on fetching data and only adding/updating Stores data. If we want to allow our users to fully manipulate the data, we must add some additional promise-based data-service functions to add/update Car:

addStore(***storeData***)

- Like addPeople(peopleData) function we must ensure that any blank values in ***storeData*** are set to null (follow the same procedure)
- Now that all of the "" are replaced with null, we can invoke the Store.create() function

- If the **Store.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 store"

updateStore(***storeData***)

- Like addStore(storeData) function we must ensure that any blank values in ***storeData*** are set to null (follow the same procedure)
- Now that all of the "" are replaced with null, we can invoke the Store.update() function and filter the operation by "id" (ie storeData.id)
- If the **Store.update()** 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 update store".

getStoreById(***id***)

- Similar to the getPeopleById(***id***) function, this function will invoke the Store.findAll() function (instead of People.findAll()) and filter the results by "id" (using the value passed to the function - ie: 3, 5 etc.
- If the **Store.findAll()** operation resolved **successfully**, invoke the **resolve** method for the promise (with an array of only one 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".

deleteStoreByIs(id)

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

Updating Views to Add / Update & Delete Stores

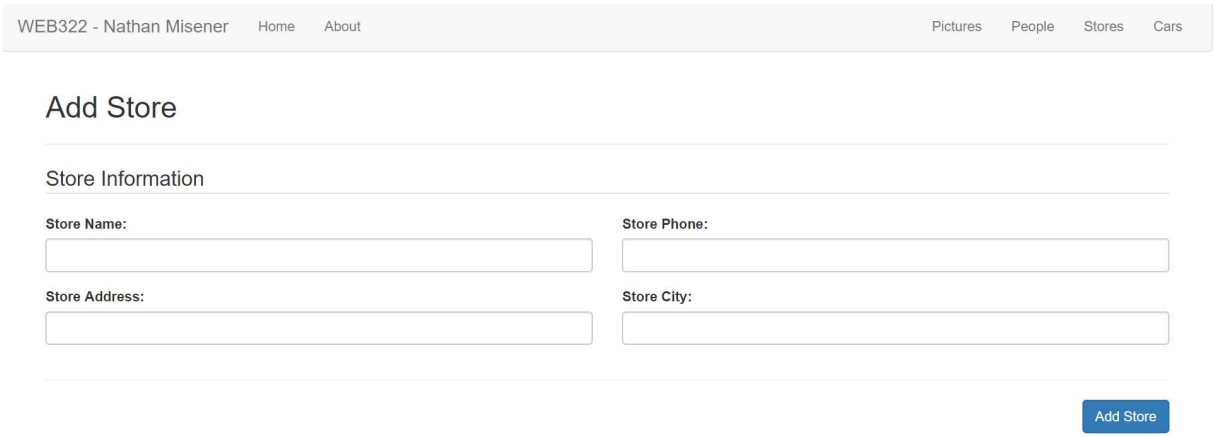
In order to provide user interfaces to all of our new "Car" functionality, we need to add / modify some views within the "views" directory of our app:

addStore.hbs

- Fundamentally, this view is nearly identical to the **addPeople.hbs** view, however there are a few key changes:
    - The header (<h1>…</h1>) must read "Add Store"
    - The form must submit to "/stores/add"
    - There must be 4 input fields (type: "text", name: "retailer", label: "Store Retailer") , (type: "text", name: "phone", label: "Store Phone"), (type: "text", name: "address", label: "Store Address"),(type: "text", name: "city", label: "Store City")

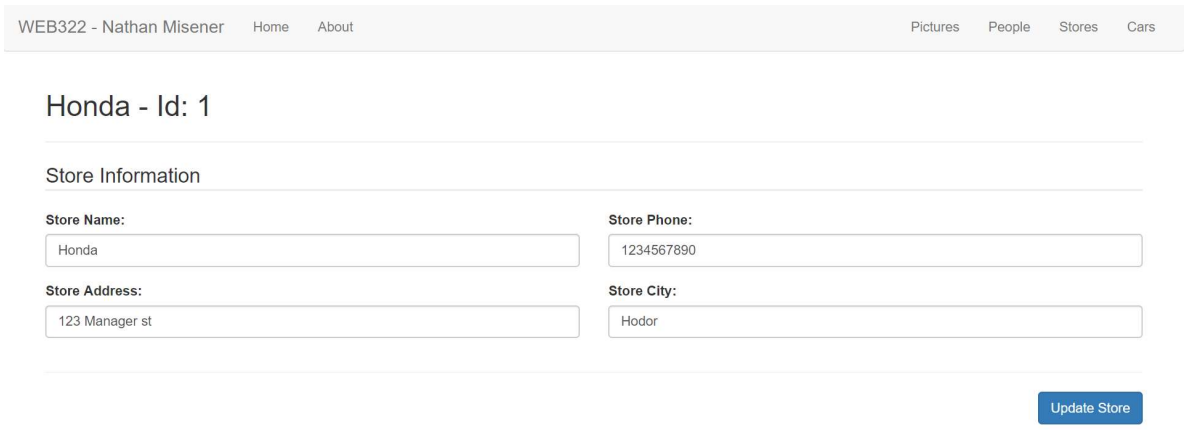
- The submit button must read "Add Store"

- When complete, your view should appear as:



store.hbs

- Like addStore.hbs, the store.hbs view is very similar to its people counterpart: people.hbs, only with a few key differences:
  - The header (<h1>…</h1>) must read "***Retailer*** - Id: ***Id***" where ***Retailer*** and ***id*** represent the Retail's name and id of the current store.
  - The form must submit to "/store/update"
  - The hidden field must have the properties: name="id" and value="{{data.id}}" in order to keep track of exactly which store the user is editing
  - There must be 4 input fields (type: "text", name: "retailer", label: "Store Retailer" value="{{data.retailer}}") , (type: "text", name: "phone", label: "Store Phone" value="{{data.phone}}"), (type: "text", name: "address", label: "Store Address" value="{{data.address}}"),(type: "text", name: "city", label: "Store City" value="{{data.city}}")
  - The submit button must read "Update Store"
  - When complete, your view should appear as the following (if we're currently looking at a newly created "Honda", for example):



stores.hbs

- Lastly, to enable users to access all of this new functionality, we need to make two important changes to our current stores.hbs file:
    - **Replace** the code:
      <td>{{id}}</td>
      with
      <td>{{id}}</td>
    - **Add** a "remove" link for every car within in a new column of the table (at the end) - Note: The header for the column should not contain any text (ie: <hr></hr>). 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 "stores/delete/*id*" where *id* is the id of the store in the current row. Once this button is clicked, the store will be deleted and the user will be redirected back to the "/stores" list. (Hint: See the sample solution if you need help with the table formatting)

Now, users can click on the store's id if they wish to edit an existing store, the store make if they wish to filter the stores list by make, or "delete" if they wish to remove the store!

Updating the "Vin" List in the Person Views

Now that users can add new Vins, it makes sense that all of the vins available to an person (either when adding a new person or editing an existing one), should consist of all the current vins in the cars database (instead of just typing it by hand). To support this new functionality, we must make a few key changes to our routes and views:

"/people/add" route

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

"addPeople.hbs" view

- Update the: **<input class="form-control" name="vin" id="vin" type="text"/>** element to use the new handlebars code:

```
{{#if viewData.cars}}
    <select class="form-control" name="vin" id="vin">
        <option value="">Select Vin</option>
        {{#each viewData.cars}}
            <option value="{{vin}}">{{ vin}}</option>
        {{/each}}
    </select>
{{else}}
```

```
    <div>No Cars</div>
{{/if}}
```

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

"/person/:id" route

- If we want to do the same thing for existing people, the task is more complicated: In addition to sending the **current Person** to the "people" view, we must also send all of the **Cars**. This requires two separate calls to our data-service module ("dataService" in the below code) that return data that needs to be sent to the view. Not only that, but we must ensure that the right vin is selected for the person and respond to any errors that might occur during the operations. To ensure that this all works correctly, use the following code for the route:

```
app.get("/person/:id", (req, res) => {

  // initialize an empty object to store the values
  let viewData = {};

  dataService.getPeopleById(req.params.id).then((data) => {
    if (data) {
      viewData.person = data; //store person data in the "viewData" object as "person"
    } else {
      viewData.person = null; // set person to null if none were returned
    }
  }).catch(() => {
    viewData.person = null; // set person to null if there was an error
  }).then(dataService.getCars)
  .then((data) => {
    viewData.cars = data; // store cars data in the "viewData" object as "cars"

    // loop through viewData.cars and once we have found the vin that matches
    // the person's "vin" value, add a "selected" property to the matching
    // viewData.cars object

    for (let i = 0; i < viewData.cars.length; i++) {
      if (viewData.cars[i].vin == viewData.person.vin) {
        viewData.cars[i].selected = true;
      }
    }

  }).catch(() => {
    viewData.cars = []; // set cars to empty if there was an error
  }).then(() => {
    if (viewData.person == null) { // if no person - return an error
      res.status(404).send("Person Not Found");
    } else {
      res.render("person", { viewData: viewData }); // render the "person" view
    }
  });
});
```

"person.hbs" view

- Now that we have all of the data for the person inside "viewData.person" (instead of just "person"), we must update every handlebars reference to data, from ***person.propertyName*** to ***viewData.person.propertyName***. For example: **{{person.first_name}}** would become: **{{viewData. person.first_name }}**

- Once this is complete, we need to update the **<select class="form-control" name="vin" id="vin">…</select>** element as well:

```
{{#if viewData.cars}}
   <select class="form-control" name="vin" id="vin">
      <option value="">Select Car</option>
{{#each viewData.cars}}
      <option value="{{vin}}" {{#if selected}} selected {{/if}} >{{vin}} </option>
   {{/each}}
   </select>
{{else}}
   <div>No Cars added yet</div>
{{/if}}
```

Updating server.js, data-service.js & people.hbs to Delete People

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

- Creating a new function (ie: **deletePeopleById(id)**) in data-service.js to "**delete**" people using the People.destroy() for a specific person. Ensure that this function returns a **promise** and only "resolves" if the Person was deleted ("destroyed"). "Reject" the promise if the "destroy" method encountered an error (was rejected).

- Create a new GET route (ie: "**/people/delete/:id**") that will invoke your newly created **deletePeopleById(id)** data-service method. If the function resolved successfully, redirect the user to the **"/people"** view. If the operation encountered an error, return a **status code of 500** and the plain text: **"Unable to Remove Person / Person not found)"**

- Lastly, update the **people.hbs** view to include a "remove" link for every person 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 "**/people/delete/*id***" where ***id*** is the Person's id of the person in the current row. Once this button is clicked, the person will be deleted and the user will be redirected back to the **"/people**" list.

Handling Rejected Promises in server.js

As a final step to make sure our server doesn't unexpectedly stall or crash on users if a promise is rejected, we ***must*** match every .then() callback function with a .catch() callback. . To resolve this, we must:

- Ensure that whenever we specify a .then() callback, we specify a matching .catch() callback.

- If the behavior isn't defined (ie: redirect or display a specific message), simply return a **status code of 500** and send a string describing the problem, within the .catch() callback function ie:

```
.catch((err)=>{
    res.status(500).send("Unable to Update Person");
});
```

NOTE: This is a catch-all that we can use for now, however a better solution would be to implement client-side validation with more sophisticated server-side validation and save this as a last resort.

Sample Solution

To see a completed version of this app running, visit: https://dry-plateau-13317.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 (Heroku) Link: _____
*
*********************************************************************************/
```

- Publish your application on Heroku & 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.