CS142 Project 6: Appserver and Database

Due: Thursday, May 23, 2019 at 11:59 PM

In this project you will start up a database system and convert your Photo Sharing App you built in Project #5 to fetch the views' models from it. We provide you a new `webServer.js` supporting the same interface as Project #5's web server but it also establishes a connection to a database. This allows you to make your app into a legitimate *full stack* application.

Setup

You should have MongoDB and Node.js installed on your system. If not, follow the installation instructions now.

Project #6 setup is different from the previous projects. You start by making a copy of your `project5` directory files into a directory named `project6`. Into the `project6` directory extract the contents of this zip file. This zip file will overwrite the files `package.json`, `webServer.js`, `.jshintrc,` and `index.html` and add several new files and directories. In the unlikely event you had made necessary changes in any of these files in your `project5` directory you will need to reapply the changes after doing the unzip.

Once you have the Project #6 files, fetch the dependent software using the command:

```
npm install
```

For this and the rest of the assignments in the course we will be running all three tiers of the web application (browser, web server, database) on your local machine.

Start and initialize the MongoDB database

Once you have installed MongoDB and created the directory for the database as described in the installation instructions, you can start MongoDB by running the command:

```
mongod
```

Since this command doesn't return until the database is shutdown you will want to either run it in a separate window or as a background process (e.g. `mongod &` on Linux/MacOS).

Once the MongoDB server is started you can load the photo app data set by running the command:

```
node loadDatabase.js
```

This program loads the fake model data from previous projects (i.e. `modelData/photoApp.js`) into the database. Since our app currently doesn't have any support for adding or updating things you should only need to run `loadDatabase.js` once. The program erases whatever is in the database before loading the data set so it is safe to run multiple times.

We use the MongooseJS Object Definition Language (ODL) to define a schema to store the photo app data in MongoDB. The schema definition files are in the directory `schema`:

- `schema/user.js` - Defines the User collection containing the objects describing each user.
- `schema/photo.js` - Defines the Photos collection containing the objects describing each photo. It also defines the objects we use to store the comments made on the photo.
- `schema/schemaInfo.js` - Defines the SchemaInfo collection containing the object describing the schema version.

These files are loaded both into the `loadDatabase.js` program where they are used to create the database and the `webServer.js` where they are used to access the database. Note: The object schema stored in the database is similar to but necessarily different from the `cs142models` JavaScript objects used in the previous assignment. Familiarize yourself with these schema definitions.

Start the Node.js web server

Once you have the database up and running you will need to start the web server. This can be done with the same command as the previous assignments (e.g. `node webServer.js`). Start your web server with the command from your `project6` directory:

```
node webServer.js
```

If you use the above command, remember to **restart the web server after each change you make to the server code.** You can also use `nodemon`, which will watch for any changes to the server code and automatically restart the web server:

```
nodemon webServer.js
```

After updating your Photo Share App with the new files from Project #6 and starting the database and web server make sure the app is still working before continuing on to the assignment.

Problem 1: Convert the web server to use the database (40 points)

The `webServer.js` we give you in this project is like the Project #5 `webServer.js` in that the app's model fetching routes use the magic `cs142models` rather than a database. Your job is to convert all the routes to use the MongoDB database. There should be no accesses to `cs142models` in your code and your app should work without the line:

```
var cs142models = require('./modelData/photoApp.js').cs142models;
```

in `webServer.js`. Note that any `console.log` statements in `webServer.js` will print to the terminal rather than the browser.

Web Server API

As in Project #5 the web server will return JSON encoded model data in response to HTTP GET requests to specific URLs. We provide the following specification of what URLs need to be support and what they should return. Your web server should support the following model fetching API:

- `/test` - Return the schema info (`/test/info`) and object counts (`/test/counts`) of the database. This interface is for testing and as an example for you we provide an implementation that fetches the information from the database. You will not have to change this one.
- `/user/list` - Return the list of users model appropriate for the navigation sidebar list. Since we anticipate large numbers of users, this API should only return an array of user properties needed by the navigation side bar (`_id, first_name, last_name`). It replaces the `cs142models.userListModel()` call in the provided code.
- `/user/:id` - Return the detail information of the user with ID of id. This should return the information we have on the user for the detail view (`_id, first_name, last_name, location, description, occupation`) and replaces the `cs142models.userModel()` call. If some thing other than the id of a User is provided the response should be an HTTP status of 400 and an informative message.
- `/photosOfUser/:id` - Return the photos of the user with _id of id. This call generates all the model data needed for the photos view including all the photos of the user as well as the comments on the photos. The photos properties should be (`_id, user_id, comments, file_name, date_time`) and the comments array elements should have (`comment, date_time, _id, user`) and only the minimum `user` object information (`_id, first_name, last_name`). This replaces the `cs142models.photoOfUserModel()` call. If some thing other than the id of a User is provided the response should be an HTTP status of 400 and an informative message. Note this API will take some assembling from multiple different objects in the database. The assignment's `package.json` file fetches the async module to make the assembling the multiple photos easier.

To help you make sure your web server conforms to the proper API we provide a test suite in the sub-directory `test`. **Please make sure that all of the tests in the suite pass before submitting.** See the Testing section below for details.

Your GET requests do not return exactly the same thing that the `cs142models` functions return but they do need to return the information needed by your app so that the model data of each view can be displayed with a single `FetchModel` call. You will need to do subsetting and/or augmentation of the objects coming from the database to build your response to meet the needs of the UI. For this assignment you are not allow to alter the database schema in anyway.

Implementing these Express request handlers requires interacting with two different "model" data objects. The Mongoose system returns models from the objects stored in MongoDB while the request itself should return the data models needed by the Photo App views. Unfortunately since the Mongoose models are set by the database schema and front end models are set by the needs of the UI views they don't align perfectly. Handling these requests will require processing to assemble the model needed by the front end from the Mongoose models returned from the database.

Care needs to be taken when doing this processing since the models returned by Mongoose are JavaScript objects but have special processing done on them so that any modifications that do not match the declared schema are tossed. This means that simply updating a Mongoose model to have the properties expected by the front end doesn't work as expected. One way to work around this is to create a copy of the Mongoose model object. A simple way of doing the copy is to translate the model into JSON and back to an JavaScript objects. The following code fragment does this object cloning:

```
JSON.parse(JSON.stringify(modelObject));
```

by taking `modelObject` converting into a JSON string and then converting it back to a JavaScript object, this time without the methods and special handling done on Mongoose models.

Problem 2: Convert your app to use `axios` (10 points)

In preparation for the next assignment where we will use more of the REST API convert your photo app to use `axios` rather than your own `FetchModel` routine to fetch the models from the web server. Axios is one of the many npm packages available for fetching data. It is faster and has more functionality than our `FetchModel` function, and it is what we will use to make our requests going forward. Your photo app should work with `FetchModel` function definition deleted after finishing this problem. The functionality of your app should be exactly the same -- make sure you do not break anything in the process of making this switch.

In particular, any time you call `FetchModel` in `photoShare.jsx` or your other components, you should replace it with a call to `axios.get`. See the `axios` documentation for more details.

Here are some hints for making the switch to `axios`:

- In any components fetching data using `axios`, be sure to import it as follows:

  ```
  import axios from 'axios';
  ```

- Notice that `axios.get` returns a Promise, much like `FetchModel`. We can attach success and failure handlers using `.then` and `.catch` respectively. You shouldn't have to change the logic of your handlers from `FetchModel`, very much, or at all, when attaching them to `axios.get`.

Testing

Testing a full web application is challenging. In the directory `test` we provide a test of just the backend portion of your application. The test uses Mocha, a popular framework for writing Node.js tests. To setup the test environment from inside the `test` subdirectory do an `npm install` to fetch Mocha and all the related dependencies. Once you have done this you can run the test running the command: `npm test`.

The `npm test` command runs the file `test/serverApiTest.js` which is a program written in the Mocha language (e.g. `describe()` and `it()`) testing the three Photo App backend URLs (`/user/list, /user/ID, /photosOfUser/ID`). **In order to be reasonably sure that the functionality of the backend routes conforms to spec, please check that all our provided tests pass before submitting. A portion of your project grade will be based on how many of these tests you pass.**

Extra Credit (10 points)

Your Photo App's marketing department has come up with a "small" tweak to the app to make it more social network friendly. The change is:

- The side navigation bar containing the list of users shall include two count bubbles next to each user name. The first count bubble (colored green) should be the count of the number of photos the user has in the system. The second bubble (colored red) should be a count of the number of comments that the user has authored.
- Clicking on the comment count bubble of a user should go to a new view component that shows all the comments of the user. For each of the user's comments the view should show a small thumbnail of the photo on which the comment was made and the text of the comment. Clicking on the comment or photo should switch the view to photo's detail view containing that photo and all its comments. The exact view will depend on if you implemented the stepper extra credit in project 5 or not.
- This change should only be visible if the advanced feature flag of project 5 is enabled. If you didn't do the extra credit of Project #5 you don't need to do the stepper but you will still need to implement the advanced feature flag control so that the above extra credit functionality can be toggled on or off.

In implementing this you are welcome to add new server API calls or enhance existing calls. If you do so you need to update the Mocha test (`test/serverApiTest.js`) to test your new functionality. If you add new APIs include them in a new `describe()` block following the pattern used by the other tests. Make sure that the provided tests still pass before submitting. You should not add new properties to the Mongoose Schema but you are welcome to add any indexes you need to make this work on larger data sets.

For grading the course staff will enable the Advanced Features setting on your app (if present) and look for the count bubbles UI to determine whether or not they should grade you on the extra credit portion.

Style Points (5 points)

These points will be awarded if your problem solutions have proper MVC decomposition. In addition, your code and templates must be clean and readable, and your app must be at least "reasonably nice" in appearance and convenience.

In addition, your code and templates must be clean and readable. Remember to run `npm run lint` before submitting. The linter should raise no errors.

Deliverables

Use the standard class submission mechanism to submit the entire `project6` directory. Make sure your code is free of any lint warnings and passes the provided test suite.