Lab 3: Appserver and Database

In this lab, you will start up a database system and develop a backend application to connect to our database and provide APIs to the frontend application developed in the Lab 2. We provide you with a skeleton backend app supporting the interfaces and also establishes a connection to a database.

Setup

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

Create a directory named lab3. Into the lab3 directory extract the contents of the attached zip file. Once you have the lab3 files, fetch the dependent software using the command:

npm install

Start and initialize the MongoDB database

Make sure you have set up a MongoDB Atlas account and database.

Once the MongoDB database is ready you can load the photo app data set by running the command:

node ./db/dbLoad.js

This program loads the fake model data from previous projects (i.e. modelData/models.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](http://mongoosejs.com/) Object Definition Language (ODL) to define a [schema](http://mongoosejs.com/docs/guide.html) to store the photo app data in MongoDB. The schema definition files are in the directory db:

* db/userModel.js - Defines the User collection containing the objects describing each user.
* db/photoModel.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.
* db/schemaInfo.js - Defines the SchemaInfo collection containing the object describing the schema version.

Problem 1: Build a backend app to use the database (40 points)

Backend API

We provide the following specification of what URLs need to be supported and what they should return. Your backend should support the following model fetching API:

* /user/list - Return the list of users' models appropriate for the navigation sidebar list. Since we anticipate a large number of users, this API should only return an array of the user properties needed by the navigation sidebar (\_id, first\_name, last\_name).
* /user/:id - Return the detailed information of the user with \_id of id. This should return the information we have on the user for the detailed view (\_id, first\_name, last\_name, location, description, occupation). If something 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). If something 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 need some assembling from multiple different objects in the database. The asynchronous interface to the database provided by Mongoose means these multiple object fetches can be done concurrently.

Your GET requests do not return exactly the same thing that the models 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 allowed to alter the database schema in any way.

**\*NOTE\***

Implementing these Express request handlers requires interacting with two different "model" data objects. The Mongoose system returns [models](http://mongoosejs.com/docs/models.html) 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 issue is to create a new JavaScript object and populate it with the fields sourced from the Mongoose model. The exact JavaScript data structure specified by the API can be built this way and returned in the HTTP response. For REST endpoints in which the API model closely matches the Mongoose model, another approach is to use the Mongoose projection operator ([select](https://mongoosejs.com/docs/5.x/docs/api.html#query_Query-select)) to have the query return only the API-specified model fields. Section will cover these techniques in more detail.

Even if the Mongoose and API models are identical, returning the Mongoose model without performing a projection is a poor programming practice. Future projects will extend the database schema (e.g., add a password to the User model), but we won't want these properties included in the API models.

Problem 2: Fetch model data from backend app (10 points)

In preparation for the next assignment where we will use more of the REST API, convert your frontend photo app to fetch model data from the backend app as would typically done in a real application.

To convert your frontend app to fetch models from backend app, you should implement the fetchModel function in lib/fetchModelData.js.

After successfully implementing the fetchModel function, you should modify the code in

/components/UserDetail/index.jsx

/components/UserList/index.jsx

/components/UserPhotos/index.jsx

to use the fetchModel function to request data from the backend app.

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](https://www.google.com/search?q=count+bubbles+UI&rlz=1C5CHFA_enUS503US503&espv=2&biw=1440&bih=782&source=lnms&tbm=isch&sa=X&ved=0ahUKEwjS7v2Ig4LLAhVM8GMKHTAlC8IQ_AUIBigB) 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 the 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 lab 2 or not.