Server Technical Documentation

Routing and Middleware

When requests first come in to the NodeJS server, they first go through a set of middleware. These actions are defined in /server/app/config/express.js and include things like parsing message bodies or inserting compression. It is also at this step that the request is logged (using the morgan module) to the file access.log under the home directory. Request information is output in standard Apache format and is disabled when the environment is in 'test' mode. These steps are executed for every request.

From there, the request gets routed according to the paths in /server/app/config/routes.js. If the request path matches one of the defined paths, the request gets routed to the specified controller. If none of the routes match, a 404 error is returned (declared at the bottom of the routes file). The other major component here is that each route has authentication associated with it (see the 'jwtAuth' variable). For routes that should only be accessible when logged in, the jwtAuth middleware should be added to that route. For routes like 'login', you don't want any sort of authentication middleware. jwtAuth first runs the request through the passport middleware, which is responsible for authenticating a user using a predefined set of 'strategies' (in our case it is either JWT or facebook-token). These strategies are defined in /server/app/config/passport.js. It is very important that authentication is added to any login-sensitive routes.

The one other significant piece of middleware to be aware of is the `multer` middleware. It can be seen on routes for making posts or registering a profile. The multer middleware manages the upload of files (in our case, images) through form-data posts. It saves these uploaded files to a specified location (the /public/ folder) and puts the filepath in the request for later use.

Authentication

The server currently supports one centralized form of authentication: JWT (JSON Web Tokens). These are returned to the user upon logging in and are essentially a session key. The one major difference is that the JWT itself contains the user's id, so that doesn't need to be fetched from the database at a later point. The validity of the JWT is ensured by using a 'secret' to sign the JWT, which is then verified later to ensure that the user id was not forged or tampered with. The JWT also handles expiration times internally, so both the client and server can check if the token is expired by just looking at the encoded JWT data.

To accommodate for alternative forms of login, such as facebook login, we can simply have the user first access a route that requires the user to send an appropriate facebook login, which is then verified through a passport strategy. Once the user is verified to have a facebook account (and a new account is potentially created), the user is sent a JWT token. The user can then use this JWT token to access the main portions of the app in the same way that they would if they were logged in with a local account.

Models

A relational diagram of the 4 models can be seen in the database-schema-diagram.png file.

All of the models are defined in /server/app/models/

- User: represents the account of each user
  - username: unique string created during local registration OR unique string copied from facebook email
  - password: (hashed) string created during local registration
  - imagePath: filepath of locally stored profile image
  - posts: array of Listing Object IDs
  - likes: array of Listing Object IDs
  - dislikes: array of Listing Object IDs
  - facebookID: optional field for user's unique Facebook account ID
  - facebookImagePath: url of user's facebook profile image (via Facebook Graph API)
- Listing: represents a post
  - creator: User Object ID of account that posted this listing
  - description: long-form text describing this posting
  - price: cost number to purchase
  - type: required category string
  - imagePath: filepath of locally stored listing image
  - numLikes: number of people that current like this post (non-relational)
  - coordinates: longitude, latitude pair of coordinates
  - locality: City/Town name associated with coordinates above
- Conversation
  - buyer: User Object ID of potential buyer in this conversation
  - seller: User Object ID of seller (item creator) in this conversation
  - item: Listing Object ID associated with this conversation
  - messages: array of Message Object ID's associated with this conversation
  - createdAt: timestamp of when this object was created
  - updatedAt: timestamp of last time this object was modified
- Message
  - sender: User Object ID for who sent the message
  - text: string of the message text
  - createdAt: timestamp of when this object was created
  - updatedAt: timestamp of last time this object was modified

Bugs

- When a user attempts to link their profile to a brand new Facebook account, the accounts do not merge correctly and there ends up being two separate accounts (one through local login, one through Facebook login).
- Other than basic required vs. optional field designations and username uniqueness, most of the

database fields do not have any validations. This includes things such as password strength requirements or acceptable language in post descriptions.