Final Project

Kendra Swafford
CS 496 - Fall 2016

http://ec2-54-70-44-156.us-west-2.compute.amazonaws.com:3000/login

I. Restful API:

For this project, I built off of my previous MEAN applications. Following the standard model-view-controller architectural pattern, I'm using an Angular front end to dynamically create views; routing is handled by Express, and Node acts as the controller to make changes to the model; MongoDB, the model, is a non-relational database. The URIs are as follows:

GET to http://ec2-54-70-44-156.us-west-2.compute.amazonaws.com:3000/login:
Returns 'login' view
(note: if user was logged in, this will log the user out by deleting the session token)

GET to http://ec2-54-70-44-156.us-west-2.compute.amazonaws.com:3000/register:
Returns 'register' view

POST to
http://ec2-54-70-44-156.us-west-2.compute.amazonaws.com:3000/register:
Calls the userService create(userParam) method:
  Checks if username is already in use
  Adds hashed password to user object
  Inserts user object into database
Flashes status message (either for success or for error)
Returns 'login' view

POST to http://ec2-54-70-44-156.us-west-2.compute.amazonaws.com:3000/login:
Routes to users/authenticate, which calls the userService's authenticate(username, password) method:
  Hashes the provided password and compares with the hashed password in database
  If authentication is successful, sets the jwt token to user id
If unsuccessful, displays error message and redirects to login
If successful, saves JSON Web Token to session and makes available to angular app
Returns 'home' view

GET to
http://ec2-54-70-44-156.us-west-2.compute.amazonaws.com:3000/app/#/account:
Routes to users/getCurrentUser, which calls the userService's getById(_id) method:
  Queries the database using the logged-in user's ID
  If found, returns the user (without the hashed password)

Returns the ‘home’ view

PUT to
http://ec2-54-70-44-156.us-west-2.compute.amazonaws.com:3000/api/users/_id:
Routes to users/updateUser:
- Checks that ID of user making request matches ID of user to edit
- Calls the userService’s update(_id, userParam) method:
  - If a new username is provided, checks that this name isn’t taken
  - Updates each field in user object (first name, last name, and username)
  - If new password was provided, hashes password and updates

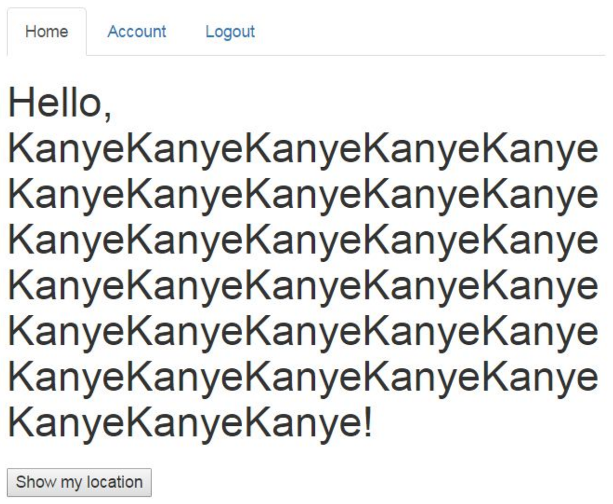
If successful, flashes success and shows updated fields to ‘account’ view

DELETE to
http://ec2-54-70-44-156.us-west-2.compute.amazonaws.com:3000/api/users/_id:
Routes to users/deleteUser:
- Checks that ID of user making request matches ID of user to delete
- Calls the userService’s delete(_id) method:
  - Removes user from database

Returns ‘login’ view

II. Mobile Content

I created a responsive web app that is neatly formatted to mobile devices. I kept a fairly streamlined layout, so the key here was mostly in adding “<meta name="viewport" content="width=device-width, initial-scale=1.0" />” to the views. Setting the viewport width to ‘device-width’ indicates that we want to match the screen’s width in *device-independent pixels*. This way, the content can match a variety of screen sizes. I tested this on a few different devices, including a Chromebook, Samsung Galaxy S7, iPhone 6, and iPad, and each view scales appropriately (login fills the screen, results wrap when necessary, etc). Here’s an example of a user with an extraordinarily long first name, which is nonetheless rendered without falling off the edge of the screen:



Hello, KanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanyeKanye!

For the mobile feature, since I went with the responsive web app route, I wasn't sure what would be sufficient, so I enabled the app to get the user's current location and display it to the view. As noted below, this should ultimately be saved to the database in the locations table.

III. Account System

My account system was created by hand. I went through a *lot* of tutorials, and my final implementation most closely resembles the very helpful Jason Watmore's (at http://jasonwatmore.com/post/2015/12/09/mean-stack-user-registration-and-login-example-tutorial). The authentication is username and password based, but it utilizes JSON Web Token (JWT), which is then hashed using bsync before being saved in the database.

The API is secured using ExpressJWT, a middleware that validations JSON Web Tokens. The server sets the app's secret to a string specified in the config file, so that when a request goes to the database, it's verified that it's coming from the expected server. Then, when the angular app starts, it sets the JWT token as the default authentication header (the 'bearer' attribute is set to the window's JWT token).

When the application attempts to authenticate a user, the userService's authenticate(username, password) method is called, which hashes the provided password and compares it with the hashed password in database. If the user is not successfully authenticated, the application redirects to the login screen *without* loading the rest of the app. If the user *is* successfully authenticated, then the jwt is saved to the session, which makes it handy to check for authentication with every operation. (For example, if a user attempts to modify a record which is not their own, the app will reject it; the ID of the user-to-update must match the ID of the currently-logged-in-user.

IV. Notes

Depending on what I can get done tonight, I might resubmit this code tomorrow with a functioning second entity; with the user table, I’ve got one working. I have all the pieces for a one-to-many users-to-locations relationship in the database, but when I try to render the view with a ‘Location’ tab, I’m getting some really weird error messages I haven’t yet figured out how to resolve (I think something is janky with my locationService), so I rolled back to an earlier version to at least turn in the functional account system with a responsive, geolocating web app. If I *can* get the second entity functional, then the routes will be similar to those outlined above, where the user can create a new location visited by getting their current position = POST /locations, view locations visited = GET /locations, edit locations visited = PUT /locations/_id, and delete location = DELETE /locations/_id.