MIS 5050/6050
Fall 2020
Final Exam

Instructions – READ CAREFULLY!!!!

1. This is a take-home exam and is an **individual effort**; you may NOT collaborate in any way with other people. This includes not viewing/sharing answers in any form with **current** or **former** students. Any violation of academic integrity will result in a failing grade.
2. You may refer to the textbook, class assignments, online resources, or **the instructor** for help in answering the questions. However, **your answers must be in your own words**, particularly for questions that ask for descriptions or explanations. Copying text from the textbook, the web, or any other source, even with minor modifications, is **plagiarism**, and will result in a failing grade.
3. This exam requires a significant time investment (estimated 1-hour per question assuming you understand the material). Use your time wisely and focus on the easier questions first.
4. BE SURE TO SAVE YOUR FILES FREQUENTLY AND BACK THEM UP!! Lost data is not a valid excuse for failing to submit your work on time. Avoiding data loss is solely your responsibility.
5. For this exam, you will continue making modifications to the *Mustacchio* application developed in the midterm exam. To begin, download and extract the accompanying MIS5050-6050-final-exam.zip file. Inside this extracted folder you will find a starting version of the Mustacchio application containing the functionality from the midterm exam. Before starting on the questions for the final exam, be sure to:
    a. Run npm install to install all required packages
    b. Replace the localhost MongoDB connection string with your Atlas connection string in main.js
    c. Ensure that the collection names specified in the Mongoose models (third argument of the Mongoose.model() function call in each model module) match the collection names in your own mustacchio database.
6. Turn in your exam by zipping your exam directory and submitting it to Canvas by the due date. To save space, you may delete the node_modules folder (make sure the package.json file contains a list of all application dependencies). Make sure you have run all code to verify that it works.
7. HAPPY CODING!!!

1. ***New Style with File Upload.*** Add functionality to the *Mustacchio* application that allows a user to upload a new mustache style to the application database, including uploading an image to the server file system. Specifically:
    - On gallery.ejs, add a "New style" hyperlink at the top of the page (above the gallery images). Clicking on this link should render a view called new-style.ejs that contains a form for uploading a new style. This form should include text input fields for title and description, and a file input field (`<input type="file">`) for the style image.
    - When the form is submitted, the title, description, and image name should be inserted into the database. In addition, the image itself should be saved in the public/images directory. To do this, use a package such as express-fileupload (see basic code example here). Upon successful upload, the user should be redirected to the gallery.ejs view, where the new style should be visible in the list.

2. ***User Model and Controller.*** Add the following components/functionality to the *Mustacchio* application:
    - In the appropriate folder, create a Mongoose schema for an application user that contains the following fields:
        i. firstName (string, required)
        ii. lastName (string, required)
        iii. email (string, required)
        iv. isAdmin (Boolean, default: false)
        v. favoriteStyles (array of style IDs that reference the styles collection)

      Add a plugin for this schema to work with the passport-local-mongoose package, using email as the username field. Create a model based on this schema and make this model available to other files in your application. (**Hint**: you can supply a third string argument to the Mongoose.model() function to specify the exact collection name that the schema should be connected to.)
    - In the appropriate folder, create a usersController.js module that will contain all route handler functions related to application users.

3. ***User Authentication.*** Add the necessary code to enable user account creation and authentication using the Passport.js and passport-local-mongoose modules (i.e., Passport.js should use a local authentication strategy that uses your user schema/model). Specifically:
    - Install the express-session, passport, passport-local-mongoose packages and add the necessary code in main.js to configure these packages for use in the application.
    - Create a LOGIN link in the main site menu that directs the user to a *login.ejs* view. This view should have fields for the user to enter their email address and password and submit these for Passport.js authentication via a function defined in usersController.js. There should also be a link on this view to enable new users to create an account. Clicking this link should redirect to a register.ejs view with a form for the user to enter their first name, last name, email address, and password. Upon submitting this form, a new user account should be created using Passport.js via a function defined in usersController.js.
    - Upon successful login, the user should see a "Welcome <<firstName>>" message that appears somewhere near the top of the page (Hint: place this in the layout.ejs file). This message should only appear if the user has successfully authenticated. In addition, the LOGIN link in the main site menu should be changed to LOGOUT. Clicking this link should log the user out using Passport.js via a function defined in usersController.js.

4. ***Favorite Styles.*** Add functionality to allow a logged-in user to save a mustache style as a favorite style and then view a list of his/her favorite styles. Specifically:
    - Modify the style detail view (gallery-single-post.ejs) to add a "Save as favorite" button/hyperlink that appears only when a user is logged in to the application. When the user clicks this button/link, the style ID should be saved to the user's list of favorite styles in the database (favoriteStyles in user schema). This should be done using a route handler function in the usersController.js module.
    - Create a new view called favorite-styles.ejs that shows a user's favorite styles. A link to this view should appear in the main site menu only when a user is logged in. The view should look identical to the gallery.ejs view, but should only show the styles marked as favorites by the currently logged-in user. As on gallery.ejs, each image tile should be a link to the style detail view.

5. ***Admin Functionality*.** Implement the following functionality for administrative users (isAdmin=true):
    - The views *contact-list.ejs* (shows all contact requests that have not received a response) and *contact-respond.ejs* (allows for responding to a contact request) should be visible only to administrative users. Implement the necessary code to (a) show the link to the *contact-list.ejs* view only to administrative users and (b) restrict access to both these views to only administrative users. (Other users should be redirected to a view that gives an unauthorized access message.)
    - Create a new view that allows an administrative user to view, grant, and revoke admin privileges for each user. The view should show a list of all application users in an html form. For each user, display the full name, email address, and a checkbox that is either checked if the user is an administrative user or unchecked otherwise. The admin user viewing this form should be able to check/uncheck the box for each user to grant/revoke administrative privileges. The form should have a submit button that posts to a route that calls a handler function in the usersController.js module. This function should loop through the array of users submitted in the form and change the admin privileges for each one. (**Hints**: You want to set up the form to return an array of objects representing each user. See this page for setting up the form using array syntax: https://mattstauffer.com/blog/a-little-trick-for-grouping-fields-in-an-html-form/ and be sure that you have the {extended: true} option set in the line app.use(express.urlencoded({ extended: true })); in main.js. Use the Mongoose findByIdAndUpdate function to loop through the array of users and update admin privileges for each one.)

6. ***Styles API.*** Suppose you would like to create a REST API that allows other applications to retrieve a list of mustache styles. This list is sent as a JSON array of style objects when a GET request is sent to `/api/styles`. Each style object in the array should show the style ID, title, and description, and should provide a link to the style image (e.g., http://localhost:3000/images/mustache1.jpg). Do the following:
    - Add a new apiController.js module with a function called getStyles that retrieves the styles from the database, modifies the imageUrl of each style to the full hyperlink url, and returns the array of styles in JSON format. Add a route for `/api/styles` that calls this function. Test the functionality by using the browser or Postman to sent a get request to this route.
    - Restrict access to the styles API by requiring the user to present a valid JSON Web Token before retrieving the data. For simplicity, you will provide a JSON Web Token to anyone

who requests one (a user account for the application is not required) and receive the token in a token querystring parameter (http://localhost:3000/api/styles?token=<<token>>) rather than in the Authorization header. Specifically, do the following:

i. Install the jsonwebtoken npm package in the application and require it in the apiController.
ii. Create a new function called getToken in the apiController. This function should generate (sign) a new JSON web token that expires 24 hours in the future. (Since you are not requiring a valid account for the application, you do not need to authenticate using passport or retrieve a user id; only the expiration date is required in the token's body.) Return a JSON object containing this token in the response. Create a GET route for `/api/token` that calls this function. Test this function by using the browser or Postman to send a GET request to http://localhost:3000/api/token.
iii. Create a new function called verifyToken in the apiController. This function should verify a token that is passed in a querystring parameter called token. If the token is successfully verified, call next(); otherwise, send an error message in JSON format. Modify the GET route for `/api/styles` to first call the verifyToken function before calling getStyles. You can test this route by using the browser or Postman to send a GET request to http://localhost:3000/api/styles?token=<<token>>, substituting a valid token from getToken for the <<token>> placeholder.

7. ***External API.*** Find a third-party REST API that returns some useful or interesting data. (The API does not need to be related to the *Mustacchio* application, but could be something you'd like to use for your project.) You can find free APIs on many websites, including:
   - https://github.com/public-apis/public-apis
   - https://any-api.com/
   - https://rapidapi.com/collection/list-of-free-apis

   Create a new route in the *Mustacchio* application that calls this API (use Axios or a similar npm module) and renders the data returned in a view called external-api.ejs. Include a short paragraph on this view that provides a link to the API website and a short description of what the API call does. You can also place a form on this view to pass data to the API if necessary. Be sure to supply any required API keys in the API request.

Deep Thought:

"I hope that someday we will be able to put away our fears and prejudices and just laugh at people."

-Jack Handey