MIS 5050/6050
Fall 2020
Midterm 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. To complete the exam, download and extract the accompanying zip file MIS5050-6050-midterm-exam. Inside this extracted folder you will find subdirectories that correspond to one or more exam questions with the necessary files inside (Q1-2, Q3, etc.)
    a. For conceptual questions (1-2): Write your answers in the Word document provided. Be sure to answer all parts of each question and label each question number clearly.
    b. For coding questions (3-10): Write or change the necessary code in the files provided for these questions.
4. 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.
5. 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.
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. Node.js has been defined as follows:

> "Node.js is a JavaScript runtime build on Chrome's V8 Javascript Engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js package ecosystem, npm, is the largest ecosystem of open-source libraries in the world."

In your own words, briefly describe the following (one short paragraph for each):

a. What is meant by a "JavaScript runtime" and how Node.js interfaces with the Chrome V8 JavaScript engine.
b. What the terms "event-driven" and "non-blocking" mean and how they are implemented in Node.js (your answer should include a description of the event loop and its function).

c. What an npm package is and how to (i) install a package both globally and as an application dependency and (ii) import a package into a JavaScript file so that it can be used. (Provide examples of commands.)

2. The Model-View-Controller (MVC) architecture is a popular approach for designing many different types of software applications on many different platforms.
    a. **In your own words**, briefly describe the function and purpose of each MVC component (model, view, and controller) and describe how each component can be implemented in a Node.js web application.
    b. Name at least two advantages of using the MVC architecture and, **in your own words**, briefly describe how/why MVC provides these advantages.

3. Becoming a proficient Node.js developer requires familiarity with two common JavaScript data structures: arrays and objects. In the Q3/Q3.js file in the midterm exam application, do the following:
    a. Declare a variable that contains an array of objects representing at least three of your friends. Each friend object should have the following properties: firstName (string), lastName (string), birthDate (Date), and favoriteFoods (array of strings).
    b. Write a function that accepts the array of friend objects as a parameter and logs the following to the console **for each friend** (use any appropriate array looping technique):
        <<firstName>> <<lastName>> is <<age>> years old and likes the following foods: <<favoriteFoods>>.
        You may choose to calculate age and build the output string in the function or add suitable methods to each friend object. Call the function to verify that it works correctly.
    c. Write a function that accepts a Date and the array of friends objects as parameters and returns a *new array* of friends whose birthdays are before (older than) that date. Call the function and log the resulting array to the console.

4. The callback pattern is a central feature of asynchronous JavaScript programming. In the Q4/Q4.js file in the midterm exam application, use *asynchronous* functions of the Node *fs* module to do the following:
    a. Create a new file called *working .txt* (saved to the same folder) with the contents "Creating file…"
    b. Append one or more new lines of text to the file (you choose the content)
    c. Read the contents of the file and print them to the console
    d. Rename the file to *complete.txt*

    For each operation, use the callback pattern. Structure your code so that the operations are guaranteed to happen in the prescribed order. For each operation, log any errors that occur to the console.

For the remaining questions, imagine the following scenario: You are a descendant of the great Italian general Giuseppe Garibaldi. You have been blessed with your progenitor's hirsute genes and the ability to grow a manly mustache that even Tom Selleck would envy. You have come to realize that all your success in life is due to your luscious lip foliage, and you would like to create a web application where other Lipholstery enthusiasts can share ideas and interact. You know a little HTML/CSS and have developed a basic website template called *Mustacchio* (Italian for mustache). With your newfound

Node.js expertise, you would now like to develop Mustacchio into a dynamic, data-driven web application.

In the midterm exam folder, you will find a directory called *Q5-10_mustacchio*, to which you will make changes in the following questions. These questions are cumulative in nature, so complete them in order!

5. ***App Initialization***. In this step, make the necessary changes to convert the Mustacchio html website template to a Node.js web application that follows the architecture outlined in the book. When you are finished with this step, the following should be complete:
    a. The *Q5-10_mustacchio* root directory should include public, models, views, and controllers directories
    b. The css, fonts, images, and js folders should be moved to the proper directory
    c. The .html files should be renamed as .ejs files and moved to the proper directory
    d. A layout.ejs file should be created that contains content the for the site header and footer (everything before and after and this div itself but NOT anything inside this div). All other pages should be modified to use this layout. (**Hint**: add a leading / to any relative path values for href and src attributes to ensure that these resources are loaded correctly with later routes.)
    e. In the footer, the social media links appear in a <ul> element. This element should be moved to a partial called _socialmedia.ejs so that it can potentially be used elsewhere in the application. This partial should then be included in the footer in its present location.
    f. A homeController should be created with functions to render the appropriate views for the index and about links in the site navigation menu.
    g. A main.js file should be created in the *Q5-10_mustacchio* directory that requires Express, EJS, and Express-EJS-layouts, launches an Express web server (port 3000), and provides GET routes that call the homeController functions to render the index and about views. (Be sure to update the href attributes of all links to point to the appropriate routes).

6. ***Style gallery and style detail***. The gallery.ejs view contains tiled images of nine different mustache styles, but this page is currently static. In this step, convert this page into a dynamic view that is populated with information from a MongoDB database called *mustacchio* hosted on your MongoDB Atlas cluster. Specifically, do the following:
    a. Add the necessary code to main.js to use Mongoose to connect to a *mustacchio* database on your Atlas cluster
    b. In the appropriate location within your app, create a Mongoose schema for a mustache style that contains the following three required fields:
        i. title (string, required, 30 characters max)
        ii. imageUrl (string, required, must end with .jpg or .png)[1]
        iii. description (string, required)

        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.)
    c. Using either the Atlas web interface or your own code written in a separate file, insert into a *styles* collection in the database nine documents that represent each of the nine styles on

[1] You can check this with the match Mongoose schema validator and a regular expression, or you can write a custom validation function using path.extname()

this page. Come up with your own titles and descriptions (be creative, but short is fine). For the imageUrl, store only the image file name (e.g., mustache1.jpg) that corresponds with each style (see images folder).

d. Make the necessary changes so that the content of gallery.ejs is pulled from the database (using the Mongoose model) when a GET request is sent to `/styles`. Create a stylesController.js module in the appropriate location and add a request listener method that queries the database for all mustache styles and renders the gallery.ejs view with the appropriate data. Each image in the gallery should be a link to `/styles/:id`, where :id is the id of the style. Then, create the necessary route in main.js and update the link on the site navigation menu in index.ejs. The final gallery.ejs view should look just like the .html version (different ordering of images is ok).
e. When the user clicks on an image in the gallery, a GET request is sent to `styles/:id`. Implement the necessary code to show a more detailed view of a style by making the necessary changes in main.js, stylesController.js, and gallery-single-post.ejs. The view should look exactly like gallery-single-post.html in the template, and should show the image, title, and description.

7. ***Blog and blog posts***. Implement functionality similar to the previous question for the blog.ejs and blog-single-post.ejs views. Specifically:
    a. Create a Mongoose schema/model for blog posts. Each post should have:
        i. title (string, required, 30 characters max)
        ii. summary (string, required, 250 characters max)
        iii. content (string, required)
        iv. imageUrl (string, required, must end with .jpg or .png)
        v. datePosted (Date, required)
    b. Populate an appropriately-named collection in your *mustacchio* database with three blog posts (feel free to use the images provided on the blog.html template page)
    c. Create a blogPostsController that will provide all handler functions for route requests related to blog posts.
    d. Write the necessary code to render blog posts dynamically on the blog.ejs view, which should be served when a GET request is sent to `/blogposts`. The page should look exactly like blog.html, and should show the image, title, and summary of each post. Posts should be displayed in reverse chronological order by datePosted. Clicking the image or "READ THIS" on a post should send a GET request to `/blogposts/:id`
    e. When the user clicks on a blog post on the blog.ejs view, a GET request is sent to `blogposts/:id`. Implement the necessary code to show a more detailed view of a style by making the necessary changes in main.js, blogPostsController.js, and blog-single-post.ejs. The view should look exactly like blog-single-post.html in the template, and should show the image (use the same image), title, and full content of the blog post.

8. ***Contact requests***. In this step, implement functionality to (a) allow users to submit a new contact request (contact.ejs), (b) allow admins[2] to see a list of contact requests that have not received a response (contact-list.ejs), and (c) allow admins to respond to contact requests (contact-respond.ejs). Specifically, do the following:
    a. Create a Mongoose schema/model for contact requests. Each contact request should have:

[2] Although this functionality is intended for admins only, it will be available to any user for the purposes of this application.

i. name (string, required, 30 characters max)
ii. address (string)
iii. email (string, required, should be in valid email format)[3]
iv. phone (string)
v. message (string, required)
vi. datePosted (Date, required)
vii. response (string)
viii. dateResponded (Date)
ix. shortMessage (**virtual**, returns the first 10 words of the message followed by …)

b. Create a contactsController that will provide all handler functions for route requests related to contacts.
c. Add the necessary code in main.js and contactsController.js to serve up the contact.ejs view when a GET request is sent to `contacts/new`.
d. Write the necessary code in main.js, contactsController.js, and contact.ejs to insert a new contact request into an appropriately named collection in the mustacchio database when the user submits the form on contact.ejs. The form should be submitted to `contacts/create` using an http POST request. Create a new view called Thanks.ejs and redirect the user to this view upon successful submission of the form.
e. Write the necessary code in main.js, contactsController.js, and contact-list.ejs to show all contact requests that have not yet received a response (dateResponded is null). The contact-list.ejs view should be served when a GET request is sent to `/contacts`. The table in this view should show the datePosted[4], name, and shortMessage of the contact request. The "View" link for each message should send a GET request to `contacts/:id/edit`, where :id is the id of the contact request.
f. Write the necessary code in main.js and contactsController.js to render the contact-respond.ejs view when a GET request is sent to `contacts/:id/edit`. The handler method should query the database for the contact request to be edited and populate the grayed-out (disabled) form fields with the values of this contact request.
g. When the user submits the form on contact-respond.ejs, a POST request should be sent to `/contacts/:id/update`. Write the necessary code in main.js and contactsController.ejs to update the contact request document with the response and the responseDate. (**Hint**: Note the hidden field on this form for storing the id of the contact). The user should then be redirected to `/contacts`. Clicking 'Cancel' should redirect to `/contacts` without performing the update.

9. ***Middleware***. Create the following middleware for your application:
   a. In the homeController, add a function to log all requests to a file called requestLog.txt, which should be stored in a log directory within the mustacchio root directory. For each request made to the server, a new line should be added to this file as follows:

      Sat Jun 09 2020 17:46:21 - GET request sent to /contacts

[3] You can check this with the match Mongoose schema validator and a regular expression, or you can write a custom validation function using a package such as validator that can validate various types of inputs.
[4] Check out the dateformat package to easily display dates in various formats. You can pass a reference to this (or any) required package to the .ejs view within res.render() and then use it's functions in the embedded JavaScript within the view.

The log function should execute first and then call the next function in the request handler sequence. Modify the main.js file to use this middleware function for all requests.

b. Create an errorController with an error-handling middleware function that handles any errors that occur in processing the request/response cycle. This function should render an error view (which you will need to create) that states that an error occurred. The error message itself should be logged to the console. In the other controllers, modify all route handler functions that involve asynchronous operations (e.g., accessing database, reading/writing files, etc.) to catch any errors that occur and pass them to this error-handling function using next().

10. ***Create!*** For this final step, come up with your own additional functionality that you can add to the Mustacchio application. This function should either (a) require the creation of an additional view that interfaces with the database, (b) require the implementation of a new npm package, or (c) both. For example, you could create a view (with associated routing functions) that allows a user to create a new blog post. Alternatively, you might use a package such as nodemailer to create a middleware function that sends an email to an administrator whenever a new contact request is sent. For this question, implement your new function and then describe what you did in the document inside the Q10 folder. Make your great-great-great grandfather proud!

Deep Thought:

"If you're in a war, instead of throwing a hand grenade at the enemy, throw one of those small pumpkins. Maybe it'll make everyone think how stupid war is, and while they are thinking, you can throw a real grenade at them."

-Jack Handey