

# IS 5700/6700

Fall 2025

## Final Exam

### **Instructions – READ CAREFULLY!!!!**

- 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 and will be reported to the university.
- 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.
- The **appropriate** use of generative AI (e.g., ChatGPT, GitHub Copilot, etc.) is allowed and encouraged. However, AI should be used to enhance your learning, not substitute for it.
  - AI may be used to:
    - Explain a concept or technique and provide examples.
    - Generate website/database content (see **text in green** throughout the exam for places where AI may be used to generate content).
    - Troubleshoot and debug code written by you.
  - AI **may NOT** be used to:
    - Generate wholesale answers to exam questions.
- To complete the exam, download and extract the accompanying zip file. Inside this extracted folder you will find files and subdirectories that correspond to one or more exam questions. Write the necessary code in the provided files as instructed in the questions.
- 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.
- 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.
- Turn in your exam by zipping your exam directory and submitting it to Canvas by the due date. To save space, **please do NOT include the node\_modules folder in your submission** (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.
- HAPPY CODING!!!

**Note:** For this exam, you will continue developing the *Mentor* web app that you started in the midterm exam. You are encouraged to start with the provided starting version of this application; however, you may also use your own version if you successfully completed all parts of the midterm app.


1. **Migrating to MongoDB and Mongoose.** For this question, make the necessary code changes to migrate the midterm exam version of the Mentor app (which uses MySQL and Sequelize) to use MongoDB and Mongoose. Specifically:
  - a. Import the Mongo and Mongoose packages into the application. Modify the app.js file with the necessary code to connect to a new MongoDB database that the app will use.
  - b. **Create new versions of the *contact*, *course*, *event*, *testimonial*, and *trainer* schemas/models using Mongoose.** Note the following:
    - i. The schemas do not need an id field specified, as this will be added automatically (as `_id`) by MongoDB.
    - ii. In the course model, the *registrants* field should be converted from an integer value to an array of Mongoose IDs that reference a user model (to be created in the next question). In addition, add a *schedule* field that shows the day/time that the course is offered (this can be a date/time or string field according to your preference).
    - iii. Any slug fields should be automatically set using either a custom set function or a Mongoose pre-save hook.
    - iv. Be sure to preserve validation on any model field (e.g., email, images, etc.) that has Sequelize validation.
  - c. Modify all controller functions in the app to use the new mongoose models (modify the code to use the appropriate method calls for executing queries).
  - d. Modify any necessary EJS view code to account for any changes in model property names (e.g., `id` → `_id`). The views should all function as they did before.
  - e. Populate the database on your Atlas cluster with data for courses, events, testimonials, and trainers. This can be done by entering the data manually on Atlas or through a custom initialization script that adds the data. (You can provide the models to AI and ask it to generate sample JSON data.)

After making these changes, the app should run as it did before.

2. **User Authentication.** Add the following components/functionality to the *Mentor* application:
  - a. 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, valid email format)
    - iv. `password` (string, required, minimum of 6 characters. Should be encrypted before storing in database.)
    - v. `roles` (array of strings identifying user roles – e.g., “user”, “admin”)
    - vi. `courses` (array of course IDs that reference the course model)

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.)

- b. In the appropriate folders, create a *user-controller.js* and *user-routes.js* modules that will contain all routing and handler functions related to application users.

- c. Install the express-session package and add the necessary code in app.js to configure this package for use in the application. The session data should be stored in your MongoDB database.
  - d. Implement functionality to allow the user to login or sign up for an account using the login and signup views. Upon successful login, the user should see a flash welcome message “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 Loginlink in the main site menu should be changed to Logout. Clicking this link should log the user out.
  - e. Create at least one admin user in your MongoDB database, by manually adding “admin” to their roles array using the MongoDB Atlas interface. **IMPORTANT:** For testing purposes, display the credentials (email and password) for this account directly on your login.ejs view.
3. **Course Registration.** Add functionality to allow a logged-in user to register for a course. Specifically:
  - a. Add a “register” button for each course on **both** the courses and course detail views. This button should be visible to everyone, but should redirect to the login view if the user is not logged in. If the user is logged in, it should redirect them to a **registration view, which you should create** (see next step).
  - b. The registration view should present a form that has pre-populated, readonly fields that show the logged-in user’s first name, last name, and email. There should then be a drop-down list field that contains all of the courses, with the course that the user clicked on being the default selected option (**Hint:** display course titles in each dropdown option and store course IDs as the value for each option.) A “Register” button should appear at the bottom.
  - c. Upon clicking “Register”, the user’s registration for the selected course should be recorded in the database (the course ID should be added to the “courses” array in the user document, and the user ID should be added to the “registrants” array in the course document.). The user should receive a confirmation message upon successful registration, or be redirected to an error page if something goes wrong.
  - d. On the courses.ejs view, the user icon for each course tile (  28 ) should reflect the number of registrants for the course. On the course-detail.ejs view, the “Available Seats” part of the page should show how many seats are available (Hint: capacity minus length of registrants array). If the course has reached capacity, the registration button on both the courses and course-detail views should be disabled and display “Course Full”.
  - e. On both the courses and course-detail views, if the currently logged-in user is already registered for a course, the “Register” button should be changed to “Unregister”. Clicking this button should remove the user from the course document and the course from the user document.
4. **Course Management.** Implement functionality that will allow an administrative user to insert, update, and delete courses. Specifically:
  - a. Add an “Admin” dropdown list to the main site menu (use the dropdown menu template code provided in layout.ejs). In the dropdown menu, add a “Create course” link that directs the user to a **new create-course.ejs view** that contains a form to create a new course, including uploading an image to the server file system. The form should include `<input type=“text”>` or `<textarea>` fields for the course title, summary, description, price, and capacity; a *dropdown* (`<select>`) field for the trainer, and a `<input type=“file”>` field for the

- image upload. (**Hint:** Be sure to add the `enctype="multipart/form-data"` attribute to your form so that it correctly handles file uploads.) When the form is submitted, the course information (including the image file name) should be inserted into the database. In addition, the image itself should be saved in the `public/img` directory. To do this, use a package such as [multer](#) or [express-fileupload](#) (see basic code example [here](#)). Upon successful upload, the user should be redirected to the `courses.ejs` view, where the new course should be visible in the list.
- b. On the `courses.ejs` and `course-details.ejs` views, add an “Edit/Delete” button or link to each course. Clicking on this link should render a new `edit-course.ejs` view that displays a form similar to the `create-course.ejs` view, with all editable form fields for the selected course populated in the form. Submitting this form should allow the user to change any editable course field, including the course image.
  - c. On the `edit-course.ejs` view, add a “Delete Course” button. Clicking this button should delete the course and should remove its ID from the `courses` array of any user registered for the course.
  - d. Ensure that the links to access course management functions are visible only to administrative users. In addition, make sure to protect all routes to this functionality with middleware that redirects the user to the login page if they are not an admin.
5. **Contact Response.** Implement functionality that allows administrative users to respond to contact requests. Specifically:
- a. Add a “Respond to Contacts” link under the Admin dropdown in the main site navigation menu. Clicking this link should redirect the user to a new `contact-list.ejs` view, where they can see a list of contact requests that have not received a response. Upon selecting a contact request, the user should be presented with a form that allows them to enter a response. Submitting this form should save the response to the database and update the `responseDate` field. The user should then be redirected back to the `contact-list.ejs` view.
  - b. Ensure that the link to access the contact respond function is visible only to administrative users. In addition, make sure to protect all routes to this functionality with middleware that redirects the user to the login page if they are not an admin.
6. **Courses API.** Suppose you would like to create a REST API that allows other applications to retrieve a list of courses. This list is sent as a JSON array of course objects when a GET request is sent to `/api/courses`. Each course object in the array should show all course fields EXCEPT for registrants and should provide a link to the course image (e.g., <http://localhost:3000/img/course-1.jpg>). Do the following:
- a. Add a new `api-controller.js` module with a function called `getCourses` that retrieves the courses from the database, modifies the `imageUrl` of each course to the full hyperlink url, and returns the array of courses in JSON format. Add a route for `/api/courses` that calls this function. Test the functionality by using the browser or Postman to send a GET request to this route.
  - b. Restrict access to the courses 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/courses?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 `api-controller`.

- ii. Create a new function called `getToken` in the `api-controller`. 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 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`. (Hint: use `req.query` to retrieve this value) If the token is successfully verified, call `next()`; otherwise, send an error message in JSON format. Modify the GET route for `/api/courses` to first call the `verifyToken` function before calling `getCourses`. You can test this route by using the browser or Postman to send a GET request to <http://localhost:3000/api/courses?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 *Mentor* application, but could be something you'd be interested in using for another 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 *Mentor* application that calls this API (use [Axios](#) or a similar npm module) and renders the data returned in a new view called `external-api.ejs`. This view should be rendered when a GET request is sent to `/externalapi`. In the `Q7-api-description.md` file, 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.

8. **Error Handling.** Modify the app to use express error-handling techniques. Specifically:
  - a. Create an `error-controller.js` file that will hold all error-handling middleware. Move the 404 handler function to this file and point the app to it to handle all 404 errors.
  - b. Create an express error-handling middleware function that will handle all server-level (500) errors. This function should log the error message to the console and redirect the user to a custom 500 error page.
  - c. Modify all controller functions in the app to include the following:
    - i. Any risky operation (an operation that could throw an error) should be enclosed in `try/catch` or `.then().catch()`.
    - ii. The handler functions should either (a) handle errors by manually checking for an error condition and redirecting or re-rendering if an error occurs, or (b) passing the error to the custom error page.

In short, the user should never land on a page that shows a raw, unhandled error.

9. **Create!** For this step, come up with your own additional functionality that you can add to the *Mentor* application. This functionality should be different from that implemented for the midterm exam, and 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 event.

Alternatively, you might use a package such as [nodemailer](#) to create a middleware function that sends an email to a user when their contact request receives a response (see Chapter 16 for guidance on sending emails). For this question, implement your new functionality and then describe what you did in the Q9-description.md file provided in the application.

10. **Deploy.** When your application is complete, deploy it on [render.com](#), [back4app.com](#) or a similar hosting service provider. Make sure to store database credentials, port numbers, API keys, etc. in environment variables rather than in the code itself. When you have successfully deployed your app, provide the URL to your live application in the provided Q10-deployed-url.md file.

Deep Thought:

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

-Jack Handey