Assignment 4

Trivia Quiz Builder

Submit a single zip file called **assignment4.zip**. **Your submission MUST contain a package.json file that allows your assignment to be built using `npm install`. You should not include the node_modules folder or database files in your submission. You may assume that the TA will have the Mongo.db daemon running and that the TA will have run `database-initializer.js` before running your server.**
This assignment has 100 marks. Read the marking scheme posted on cuLearn for details.

Assignment Background

In this assignment, you will implement a trivia quiz creation server. Your assignment must use MongoDB for storage of all question and quiz data.

To get started, download the database-initializer.js file from cuLearn into the directory you will do your assignment. Initialize your project using `npm init`. Ensure the MongoDB Node.js driver is installed in your assignment directory by running `npm install mongodb`. Now run the database-initializer.js file using Node.js while the MongoDB daemon is also running on your computer, and it will initialize a database called 'a4' with a collection called 'questions' that contains the same 500 trivia questions that were used in Assignment #3. If you need to reset your questions collection at any point in time, you can re-run this file to return the 'questions' collection to its original state.

Several of the routes for the server you will develop must support query parameters. For this assignment, you may assume that the query parameter values your server receives will be of the correct type. For example, if the specification indicates the value will be an integer, you can assume that you will always get a value that can be converted to an integer.

Before starting your design for the assignment, you are encouraged to read through the entire specification. It is possible that later parts will inform your design decisions and, by preparing in advance, you can avoid having to significantly refactor your code as you progress through the assignment.

Assignment Requirements

Your server must support the following routes:

GET: /questions, /questions/:qID, /createquiz, /quizzes, /quiz/:quizID
POST: /quizzes

The details that follow explain the specific functionality that your application must support for each of these routes.

GET /questions: This route will be used to query the questions in the server's database. This route must be capable of returning either JSON or HTML, depending on the content type specified in the Accept header of the request. For each request, the response may contain up to 25 matching questions. Simply return the first 25 matching questions or all of the matching questions if there are less than 25. There is no need to implement a limit query parameter or any sort of pagination for HTML results. The route must support the following two query parameters:

1. `category`: a string that should be compared to the `category` field within the supplied question objects. Note that this is different from the `category_id` field that was used in Assignment #3. If the category parameter is not specified, you should match questions of any category.
2. `difficulty`: a string that should be compared to the `difficulty` field within the supplied question objects. Note that this is different from the `difficulty_id` field that was used in Assignment #3. If no difficulty parameter is specified, you should match questions of any difficulty.

If JSON data is requested, the response should be a JSON string containing the key "`questions`" with the associated value being an array of the matching question objects.

If HTML data is requested, you must use a templating engine to render an HTML page containing the results. This HTML page should contain the matching questions. For each question, there should be a link to that specific question resource on the server (e.g., /questions/*someQuestionId*) and the text of the link should be the question's question text.

GET /questions/:qID: This parameterized route will be used to retrieve a single question with the unique ID *qID* from the server. This route must be capable of returning either JSON or HTML, depending on the content type specified in the Accept header of the request. If JSON is requested, your server should return the JSON representation of the question object. If HTML is requested, your server must use a template engine to render an HTML page containing the question's data. At minimum, this HTML page must have the question text, category text, difficulty text, correct answer, and incorrect answers. If the supplied question ID does not exist on the server, you should return an HTTP status of 404 with an error message.

GET /createquiz: This route represents the page that will be used to perform quiz creation. This route should only return HTML. The HTML can be served from a static file or generated using a template engine. You will also need to write client-side Javascript to achieve the functionality requirements of this page. The requirements of this page are described below.

1. The page must have an input field to specify the quiz creator's name.
2. The page must have an input field to specify the quiz's tags. This will be a list of keywords to describe the quiz, separated by spaces. For example, the string

"science technology computers" would represent the three tags: science, technology, and computers.

3. The page must have a list of questions that are currently part of the quiz. This list will initially be blank. For each question currently in the quiz, the entry for that question in the list should be a link to that specific question resource on the server, which should open in a new window (use the link attribute `target="_blank"` to accomplish this), and the text of the link should be the question's question text. Additionally, there must be a way for the user to select questions that are currently part of the quiz and remove them. This could involve a remove button beside each question, a checkbox beside each question combined with a remove button (as in earlier to-do list tutorials), or some other method.
4. A "Save Quiz" button. When this button is clicked, the data on the page should first be validated to ensure the following: that there is a creator name (i.e., it is not blank), there is at least one tag, and there is at least one question in the quiz. If any of those conditions are not met, an alert should be displayed indicating what needs to be changed. Otherwise, the quiz data should be sent to the server using a POST request to the /quizzes route. If the new quiz is successfully created on the server, the client should be redirected to the URL representing the newly created quiz resource.
5. A 'Search/Add Questions' section that will allow the user to browse possible questions from the server and add them to the quiz. This section must have two drop-down lists to filter questions based on category and difficulty. These drop-down lists must contain all possible categories and difficulties based on the questions stored on the server. The values in these drop-down lists should not be hard-coded. Instead, they should be generated at runtime using the information from the server's database. Below the drop-down lists, there must be a list of questions that match the currently selected category and difficulty. If the selected item in either of the drop-down menus changes, the list of questions must be updated to reflect the change. You can use a request to your server's /questions route to retrieve matching questions. For each question that matches the currently selected difficulty/category, the entry for that question in the list should be a link to that specific question resource on the server, which should open in a new window (using the link attribute `target="_blank"`), and the text of the link should be the question's question text. Additionally, there must be a method for the user to select questions to add to the quiz. This could involve an add button beside each question, a checkbox beside each question combined with an add button (like the remove functionality in earlier to-do list tutorials), or some other method. It should not be possible for the user to add the same question to a quiz more than once. Note that some categories in the dataset have the character '&' within their text, which will cause issues if you try to add the category string directly to a query string for a request (e.g., if making an XMLHttpRequest to the /questions route). Use Javascript's encodeURIComponent(string) function to encode the category string in URI format before adding it to your query string.

POST /quizzes: This route will be used to add a new quiz into the database. This route will expect a JSON body containing the representation of a quiz (creator, tags, questions). Your server must verify the information to ensure that: there is a creator name (i.e., it is not blank), there is at least one tag, and that each question within the quiz is a valid question within the database. If the quiz data is valid, a new quiz document should be inserted into the 'quizzes' collection in the database. The server should respond in a way that will allow the client that made the POST request to navigate to the newly created quiz resource (e.g., by sending back the quiz's unique ID or by redirecting the client from the server-side). If the quiz data is invalid, the response should include an HTTP status code and message describing the error.

GET /quizzes: This route will be used to query the quizzes in the server's database. This route must be capable of returning either JSON or HTML, depending on the content type specified in the Accept header of the request. The route must support the following query parameters:

1. creator: a string that should be compared to the `creator` field of the quiz objects. If no creator parameter is specified, your server should match any quiz creator's name. If the parameter is specified, a quiz should match this query parameter if the value of the query parameter can be found in the name of the quiz creator. The search should also be done in a case-insensitive manner. So, for example, if a quiz's creator is "Dave", this would match all the following values for the creator query parameter: "da", "ave", "dave", "DAVE".
2. tag: a string that should be searched for within the set of tags for each quiz. If no tag parameter is specified, a quiz should match regardless of its tags. If the tag parameter is specified, a quiz should match this query parameter if any of the quiz's tags match the given tag. The matching should be done in a case-insensitive manner but should NOT perform partial matching like the creator parameter. So a tag query parameter value of "dog" would match the tags "dog" and "DOG", but not "dogs". Note that since a quiz can have many tags, you must check to see if the query parameter value is equal to any of the tags of that quiz.

If JSON data is requested, the response should be a JSON string containing the key "quizzes" with the associated value being an array of matching quiz objects.

If HTML data is requested, you must use a templating engine to render an HTML page containing the results. This HTML page should contain a list of the matching quizzes. For each quiz, there should be a link to that specific quiz resource on the server (e.g., /quizzes/*someQuizId*). The text of the link should contain the quiz creator's name and the set of tags associated with that quiz.

GET /quiz/:quizID: This parameterized route will be used to retrieve a single quiz with the unique ID *quizID* from the server. This route must be capable of returning either JSON or HTML, depending on the content type specified in the Accept header of the request. If JSON is requested, your server should return the JSON representation of the quiz object. This JSON

data must contain, at minimum, the creator name, array of tags, and an array containing all the question data for the quiz (i.e., not just the question IDs, but all of the data). If HTML is requested, your server must use a template engine to render an HTML page containing the quizzes data. At minimum, this HTML page must have the quiz creator's name, quiz tags, and a list of the quiz questions. Each entry in the list of quiz questions should be a link to that specific question resource on the server. The text of each link should contain the question's question text. If an invalid ID is specified, a 404 error should be sent in response.

Code Quality and Documentation

Your code should be well-written and easy to understand. This includes providing clear documentation explaining the purpose and function of pieces of your code. You should use good variable/function names that make your code easier to read. You should do your best to avoid unnecessary computation/communication and ensure that your code runs smoothly throughout operation. **You must also include a README.txt file that explains any design decisions that you made and precise instructions for how to run/use your system.**

Recap

Your zip file should contain all the resources required for your assignment to run. Your submission MUST contain a package.json file that allows your assignment to be built using `npm install`. You should not include the node_modules folder or database files in your submission. The TA must be able to run your server and use the system by following instructions in the README.txt file. You may assume that the TA will have the MongoDB daemon running and will have run `database-initializer.js` before running your server.

Submit your **assignment4.zip** file to cuLearn.

Make sure you download the zip after submitting, verify the file contents, and verify that your instructions are sufficient for installing/running your system.
