

Lab 1

Due Feb 3 by 11:59pm

Points 100

Submitting a file upload

File Types zip

CS-554 Lab 1

Reviewing API Development

In this 546 Review lab, we will build a recipe API.

For this lab, you will submit a web server with the supplied routes and middlewares.

Verb	Route	Description
GET	/recipes	Shows a paginated list of recipes in the system. By default, it will show the first <code>50</code> recipes in the collection. If a querystring (http://expressjs.com/en/api.html#req.query) variable <code>?page=n</code> is provided, you will show the the next 50 recipes for that page <code>n</code> . So <code>page=2</code> will show recipes 51-100, <code>page=3</code> will show recipes 101-150, <code>page=4</code> will show recipes 151-200 and so on.. <code>page=1</code> would show the initial recipes of 1-50 that you show by default on this route. If there are no recipes for a page number (meaning there are no more recipes in the DB, then you will return a 404 status code and message stating there are no more recipes) Hint: You can use the skip and limit cursors in MongoDB that we learned about in 546 to make this work.
GET	/recipes/:id	Shows the recipe with the supplied ID. if a recipe cannot be found for that ID, you will return a 404 Status code along with an error message.
POST	/recipes	Creates a recipe with the supplied detail and returns created object; fails request if not all details supplied. The user MUST be logged in to post a recipe. In the request body, you will not be sending the <code>userThatPosted</code> , it will be populated from the currently logged in user (when they login, you will save a representation of the user in the session). You will initialize comments and the likes as empty arrays in your DB create function as there cannot be any comments or likes on a recipe, before the recipe has been created. Please see example schema below. The fields in the request body will have the following constraints:

Verb	Route	Description
		<p>title : Must be a valid string. No strings with empty spaces allowed. The data must make sense for the data you are storing.</p> <p>ingredients : Each element must be a valid string, no strings with empty spaces and there should be at least 3 valid string elements in the array. The data must make sense for the data you are storing. The minimum characters for each ingredient should be 3 characters and the max 50 characters.</p> <p>steps : Each element must be a valid string, no strings with empty spaces and there should be at least 5 valid string elements in the array. The data must make sense for the data you are storing. The minimum number of characters should be 20. No max character constraint.</p> <p>cookingSkillRequired you will ONLY allow the following values, if a value not on this list is supplied, you will display an invalid cooking skill required error: "Novice", "Intermediate", "Advanced"</p> <p>This route will return the newly created recipe once it's inserted into the database.</p>
PATCH	/recipes/:id	<p>Updates the recipe with the supplied ID and returns the updated recipe object; Note: PATCH calls can have one or more fields in the request body!</p> <p>Note: you cannot manipulate comments, likes or modify the userThatPosted object in this route! A user has to be logged in to update a recipe AND they must be the same user who originally posted the recipe. So if user A posts a recipe, user B should NOT be able to update that recipe.</p> <p>The constraints for the fields are the same as they are in the POST route. At least one field needs to be supplied in the request body and the value must be different than what is currently stored in the DB, but more than one field can also be present.</p> <p>This route will return the newly updated recipe.</p>
POST	/recipes/:id/comments	<p>Adds a new comment to the recipe; ids must be generated by the server, and not supplied. a user needs to be logged in to post a comment.</p> <p>This route will return the entire recipe object, showing the new comment in the data.</p>

Verb	Route	Description
DELETE	/recipes/:recipeId/:commentId	<p>Deletes the comment with an id of <code>commentId</code> on the recipe with an id of <code>recipeId</code>. A user has to be logged in to delete a comment AND they must be the same user who originally posted the comment. So if user A posts a comment, user B should NOT be able to delete that comment.</p> <p>This route will return the entire recipe object, showing the deleted comment is gone from the data.</p>
POST	/recipes/:id/likes	<p>Allows a user to like a recipe. A user needs to be logged in to like a recipe. If they have not already liked it, you will add the user's ID to the likes array in the recipe document. If they have already liked the recipe and hit this route again, it should remove their ID from the likes array in the recipe document.</p> <p>This route will return the entire recipe object, showing the new like in the data (if they liked it, or showing their id is removed if they deleted it).</p>
POST	/signup	<p>Creates a new user in the system with the supplied detail and returns the created user document (sans password); fails request if not all details supplied. The username must be alphanumeric and at least 3 characters long. The password should be 6 characters minimum, with at least one lowercase letter, one uppercase letter, one number and one special character contained in it.</p>
POST	/login	<p>Logs in a user with the supplied username and password. Returns the logged in user document (sans password). You will set the session so once they successfully log in, they will remain logged in until the session expires or they logout. You will store some way to identify the user in the session. You will store their <code>username</code> and their <code>_id</code> which will be read when they try to create a recipe, try to update a recipe (making sure they can only update a recipe they originally posted), post a comment or delete a comment (making sure they can only delete a reply they posted), and adding/removing a recipe like.</p>
GET	/logout	<p>This route will expire/delete the <code>cookie/session</code> and inform the user that they have been logged out.</p>

All PUT, POST, and PATCH routes expect their content to be in JSON format, supplied in the body.

All routes will return JSON.




Middleware

You will write and apply the following middleware functions:

1. You will apply a middleware that will be applied to the POST, PUT and PATCH routes for the /recipes endpoint that will check if there is a logged in user, if there is not a user logged in, you will respond with the proper status code and display an error message. (A non-logged in user SHOULD be able to access the GET /recipes route)
2. You will apply a middleware that will be applied to POST and DELETE for the /recipes/:id/comments and /recipes/:recipeId/:commentId endpoints respectively that will check if there is a logged in user. If there is not a user logged in, you will respond with the proper status code and display an error message.
3. The third middleware will apply to the entire application and will log all request bodies if there is a request body (GET routes can/will just log an empty object for the request body). Do not log passwords from the request body if the request body contains a password field. You will also log the url path they are requesting, and the HTTP verb they are using to make the request.
4. The fourth will apply to the entire application and will keep track of how many times a particular URL has been requested, updating and logging with each request.

Database

You will use a module to abstract out the database calls and will create the data access functions. You may name these however you like. You MUST do input validation and error handling in ALL of your DB functions using the same constraints as stated in the routes above.

You may find it helpful to reference the following 546 lecture code: [Lecture 4](https://github.com/stevens-cs546-cs554/CS-546/tree/master/lecture_04) , [Lecture 5](https://github.com/stevens-cs546-cs554/CS-546/tree/master/lecture_05) , [Lecture 6](https://github.com/stevens-cs546-cs554/CS-546/tree/master/lecture_06) , [Lecture 10](https://github.com/stevens-cs546-cs554/CS-546/tree/master/lecture_10) .

You will store all data in a database named as such: `LastName-FirstName-CS554-Lab1`.

You may name the collection however you would like.

All ids must be generated by the server and be sufficiently random!

YOU MUST ONLY USE TWO COLLECTIONS! ONE FOR THE RECIPES AND ONE FOR USERS. COMMENTS ARE STORED AS A SUB-DOCUMENT IN THE RECIPE COLLECTION.

The Recipe document

```
{
  _id: new ObjectId(),
  title: string,
  ingredients: [strings],
  cookingSkillRequired: string: (only valid values are "novice", "intermediate", "advanced"),
  steps: [strings],
  userThatPosted: { _id: ObjectId, username: string },
  comments: [objects],
}
```

```
likes: [of user ids that have liked the recipe]
}
```

The comment object (stored as a sub-document in the Recipe document)

```
{
  _id: new ObjectID(),
  userThatPostedComment: {_id:ObjectID, username: string},
  comment: string
}
```

The user document: You will use bcrypt to hash the password to store in the DB for signup and for login you will use the compare method to validate the correct password

```
{
  _id: new ObjectID(),
  name: string,
  username: string,
  password: hashedPW
}
```

Example Recipe:

```
{
  _id: "61294dadd90ffc066cd03bed",
  title: "Fried Chicken",
  ingredients: ["One whole chicken", "2 cups of flour", "2 eggs", "salt", "pepper", "1 cup cooking oil"],
  cookingSkillRequired: "Novice",
  steps: ["First take the two eggs and mix them with the flour, the salt and the pepper", "Next, dip the chicken into the mix", "take 1 cup of oil and put in frier", "Fry the chicken on medium heat for 1 hour"],
  userThatPosted: {_id: ObjectID, username: "graffixnyc"},
  comments: [
    {
      _id: "61294dadd90ffc066cd03bef",
      userThatPostedComment: {_id:"51294dadd90ffc066cd03bff", username: "ZeroCool"},
      comment: "Nice Recipe!"
    },
    {
      _id: "61296014082fe5073f9ba4f2",
      userThatPostedComment: {_id:"6129617a082fe5073f9ba4f5", username: "progman716"},
      comment: "This recipe was bad!"
    }
  ],
  likes: ["51294dadd90ffc066cd03bff", "61296014082fe5073f9ba4f2"]
}
```

Example User Document

```
{
  _id: "61294dadd90ffc066cd03bee",
  name: "Patrick Hill",
  username: "graffixnyc",
  password: "$2a$16$7JKSiEmoP3GNDSalogqgPu0sUbwder7CAN/5wnvCWe6xCKAKw1TD."
}
```

Error Checking

1. **You must error check all routes checking correct data types, making sure all the input is there, in the correct range and MAKES SENSE for the data you're capturing etc..**
2. **You must error check all DB functions checking correct data types, making sure all the input is there, in the correct range and MAKES SENSE for the data you're capturing etc..**
3. **You must fail with proper and valid HTTP status codes depending on the failure type. 400 bad request, 404 not found, 403 etc...**
4. **Do not forget to check for proper datatypes in the query string parameters for page (it should be a positive number, if is not a positive number, you should throw/respond with an error and status code)**

Notes

1. Remember to submit your `package.json` file but **not** your `node_modules` folder.
2. Remember to add the start command!
3. Submit your zip file as: LastName_FirstName_Section_Lab1.zip ie: Hill_Patrick_A_Lab1.zip