Lab 1

Due Jan 31 by 11:59pm **Points** 100 **Submitting** a file upload **File Types** zip
Available after Jan 24 at 12am

CS-554 Lab 1

Reviewing API Development

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

Verb	Route	Description
GET	/blog	Shows a list of blog posts in the system. By default, it will show the first 20 blog posts in the collection. If a querystring (http://expressjs.com/en/api.html#req.query) variable `?skip=n` is provided, you will skip the first `n` blog posts. If a querystring variable `?take=y` is provided, it will show `y` number of blog posts. By default, the route will show up to `20` blog posts; at most, it will show `100` blog posts.
GET	/blog/:id	Shows the blog post with the supplied ID
POST	/blog	Creates a blog post with the supplied detail and returns created object; fails request if not all details supplied. The user **MUST** be logged in to post a blog. In the request body, you will only be sending the `title` and the `body` fields of the blog post. The `userThatPosted` 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 as an empty array in your DB create function as there cannot be any comments on a blog post, before the blog post has been created.
PUT	/blog/:id	Updates the blog post with the supplied ID and returns the updated blog post object; **Note**: PUT calls must provide all details of the new state of the object! **Note:** you cannot manipulate comments in this route! A user has to be logged in to update a blog post **AND** they must be the same user who originally posted the blog post. So if user A posts a blog post, user B should NOT be able to update that blog post. In the request body, you will only be sending the `title` and the `body` fields of the blog post.

Verb	Route	Description
PATCH	/blog/:id	Updates the blog post with the supplied ID and returns the updated blog post object; **Note**: PATCH calls only provide deltas of the value to update! **Note:** you cannot manipulate comments in this route! A user has to be logged in to update a blog post **AND** they must be the same user who originally posted the blog post. So if user A posts a blog post, user B should NOT be able to update that blog post. In the request body, you will only be sending the `title` and the `body` fields of the blog post.
POST	/blog/:id/comments	Adds a new comment to the blog post; ids must be generated by the server, and not supplied, a user needs to be logged in to post a comment
DELETE	/blog/:blogId/:commentId	Deletes the comment with an id of `commentId` on the blog post with an id of `blogId` 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.
POST	/blog/signup	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.
POST	/blog/login	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 someway to identify the user in the session. You will store their `username` and their `_id` which will be read when they try to create a blog post, try to update a blog post (making sure they can only update a post they originally posted), post a comment or delete a comment (making sure they can only delete a comment they posted)
GET	/blog/logout	This route will expire/delete the `cookie/session` and inform the user that they have been logged out.

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 middlewares:

1. You will apply a middleware that will be applied to the POST, PUT and PATCH routes for the /blog 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.

For PUT and PATCH routes you also need to make sure the currently logged in user is the user who posted the blog post that is being updated.

2. You will apply a middleware that will be applied to the POST and DELETE routes for the /blog/:id/comments and /blog/:blogId/: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. For the DELETE route, you also need to make sure the currently logged in user is the user who posted the comment that is being deleted.

Database

You will use a module to abstract out the database calls.

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/code)** , **Lecture 5 (https://github.com/stevens-cs546-cs554/CS-546/tree/master/lecture_05/code)** , **Lecture 6 (https://github.com/stevens-cs546-cs554/CS-546/tree/master/lecture_06/code)** , **Lecture 10 (https://github.com/stevens-cs546-cs554/CS-546/tree/master/lecture_10/code)**

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!

The blog document

```
{
  _id: new ObjectID(),
  "title": string, "body": string,
  "userThatPosted": {_id: ObjectID, username: string},
  "comments": [objects]
}
```

The comment object (stored as a sub-document in the blog 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 blog post:

```
{
  "_id": "61294dadd90ffc066cd03bed",
  "title": "My experience Teaching JavaScript",
  "body": "This is the blog post body.. here is the actually blog post content.. blah blah blah.....",
  "userThatPosted": {_id: "61294dadd90ffc066cd03bee", username: "graffixnyc"},
  "comments": [
      {
        "_id": "61294dadd90ffc066cd03bef",
        "userThatPostedComment": {_id:"61294dadd90ffc066cd03bee", username: "graffixnyc"},
        "comment": "Thank you for all your wonderful comments on my blog post!"
      },
      {
        "_id": "61296014082fe5073f9ba4f2",
        "userThatPostedComment": {_id:"6129617a082fe5073f9ba4f5", username: "progman716"}
        "comment": "Very informative post!"
      }
  ]
}
```

Example User Document

```
{
  _id: "61294dadd90ffc066cd03bee",
"name": "Patrick Hill", "username":"graffixnyc",
"password": "$2a$16$7JKSiEmoP3GNDSalogqgPu0sUbwder7CAN/5wnvCWe6xCKAKwlTD."
}
```

Error Checking

1. **You must error check all routes checking correct data types, making sure all the input is there, in the correct range etc..**
2. **You must error check all DB functions checking correct data types, making sure all the input is there, in the correct range etc..**
3. **You must fail with proper and valid HTTP status codes depending on the failure type**
4. **Do not forget to check for proper datatypes in the query string parameters for skip and take (they should be positive numbers, if they are not positive numbers, you should throw an error)**

Notes

1. Remember to submit your `package.json` file but **not** your `node_modules` folder.