Project 3: Blogging Server on NodeJS and MongoDB

Overview

The primary task of Project 3 is to implement the website for our markdown-based blogging service that (1) lets anyone read blogs written by our users through public URLs and (2) lets our registered users create and update their own blogs after password authentication. Through this process, we will learn how to develop a back-end service using NodeJS, Express and MongoDB.

Note that your implementation of Project 3 will be used for your Project 4 as well. Since Project 4 is dependent on Project 3, it is important that you **follow instructions on this spec exactly** to avoid any potential issues later in Project 4.

Development Environment

All development for Projects 3 (and 4) will be done on a Docker container based on the "junghoo/mean" image:

```
$ docker run -it -p3000:3000 -p4200:4200 -v {host_shared_dir}:/home/cs144/shared -
```

Make sure to replace `{host_shared_dir}` with the name of the shared directory on your host. The above command creates a docker container named `mean` with appropriate port forwarding and directory sharing. Once created, you can start the container simply by issuing the following command in a terminal window:

```
$ docker start -i mean
```

This container has MongoDB (v4.4.4), NodeJS (v14.16.0), Express application generator (v4.16.1), and Angular CLI (v11.2.6) pre-installed. Make sure that they run fine through the following commands:

```
$ mongo --version
$ node --version
$ express --version
$ ng --version
```

Project Requirements

Our back-end blogging service should be accessible at the following URLs:

#	URL	method	functionality
1	`/blog/:username/:postid`	GET	Return an HTML-formatted page that shows the blog post with `postid` written by `username`.
2	`/blog/:username`	GET	Return an HTML page that contains first 5 blog posts by `username`.
3	`/login`	GET, POST	Authenticate the user through username and password.
4	`/api/posts`	GET, POST, DELETE	This is the REST API used to insert, retieve, update, and delete blog posts

Note:

1. The URL patterns 1-3 should be publicly accessible by anyone. No prior user authentication should be required to access these URLs. More detailed requirements for the URL patterns 1-2 and 3 will be given in Parts B and C, respectively.
2. The URL pattern 4 should be protected behind authentication. More detailed specifications on this API will be given later in Part D.
3. The implemented server should listen on ***port 3000*** for HTTP requests.

All blog posts and the users' authentication credentials should be stored in the MongoDB server. The MongoDB server should have at least the following two collections, "Posts" and "Users", in the database "BlogServer".

1. **Collection: Posts**

```
[{ "postid": 1, "username": "cs144", "created": 1518669344517, "modified": 151
{ "postid": 2, "username": "cs144", "created": 1518669658420, "modified": 1518
{ "postid": 1, "username": "user2", "created": 1518669758320, "modified": 1518
{ "postid": 2, "username": "user2", "created": 1518669758330, "modified": 1518
{ "postid": 3, "username": "user2", "created": 1518669758350, "modified": 1518
{ "postid": 4, "username": "user2", "created": 1518669758360, "modified": 1518
{ "postid": 5, "username": "user2", "created": 1518669758370, "modified": 1518
{ "postid": 6, "username": "user2", "created": 1518669758380, "modified": 1518
```

The first collection "Posts" stores all blog posts created and saved by our users and must have eight initial documents shown above. As users write more blog posts, more documents should be inserted into this collection. Note that "created" and "modified" fields of the two documents are all integers, whose values are milliseconds since the the Unix epoch (Jan 1, 1970 UTC).

2. **Collection: Users**

```
[{ "username": "cs144", "password": "$2a$10$2DGJ96C77f/WwIwClPwSNuQRqjoSnDFj9G
{ "username": "user2", "password": "$2a$10$kTaFlLbfY1nnHnjb3ZUP3OhfsfzduLwl2k/
```

The second collection "Users" stores the users' authentication credentials and the current "maximum postid" of the user. The collection must have two initial documents shown above. The "password" field will be used for authenticating any user to our server through the URL pattern 3. The "maxid" field is used to easily calculate the postid to be assigned to a new posting. Note that **users' passwords must NEVER be stored in plaintext**. Instead, we have to store them only after we apply a cryptographic one-way hash function. This ensures that even if a hacker breaks into our system and gets a hold of our database, they won't be able to obtain the users' passwords easily since it is time-consuming to recover the plaintext passwords from the hash values. The downside of this approach is that when a user tries to login, we will have to apply the same cryptographic hash function to the user-provided password and then match the equivalence of this hash value to what is stored in our database. This can potentially increase the computational overhead of authenticating a user, but given the potential security risk of saving plaintext passwords, it is the cost that we are willing to pay. In our case, we applied bcrypt hash function to each user's password ("password" for "cs144" and "blogserver" for "user2", respectively) using the bcryptjs module of node.js.

Part A: Create Initial MongoDB Data

In Project 3, all blog posts must be managed by MongoDB. Unlike MySQL, MongoDB doesn't have the concept of schema. All types of data are saved as *documents* in a *collection*. Since MongoDB document is essentially a JSON object, it is often a preferred back-end data storage engine for JavaScript-based development.

When you start the Docker container, it starts MongoDB server in the background. So you can start the "MongoDB command-line shell" simply by:

```
$ mongo
```

Once you are inside the shell, you can issue most MongoDB commands interactively. Go over class notes on MongoDB to review the basic MongoDB commands. If needed, review online tutorials on MongoDB, such as this one.

Now *write a script named `db.sh` that includes the sequence of mongodb shell commands that load the following documents into the two collections, "Posts" and "Users", in the "BlogServer" database*:

1. **Collection: Posts**

```
[{ "postid": 1, "username": "cs144", "created": 1518669344517, "modified": 151
{ "postid": 2, "username": "cs144", "created": 1518669658420, "modified": 1518
{ "postid": 1, "username": "user2", "created": 1518669758320, "modified": 1518
{ "postid": 2, "username": "user2", "created": 1518669758330, "modified": 1518
{ "postid": 3, "username": "user2", "created": 1518669758350, "modified": 1518
{ "postid": 4, "username": "user2", "created": 1518669758360, "modified": 1518
{ "postid": 5, "username": "user2", "created": 1518669758370, "modified": 1518
{ "postid": 6, "username": "user2", "created": 1518669758380, "modified": 1518
```

2. **Collection: Users**

```
[{ "username": "cs144", "password": "$2a$10$2DGJ96C77f/WwIwClPwSNuQRqjoSnDFj9G
{ "username": "user2", "password": "$2a$10$kTaFlLbfY1nnHnjb3ZUP3OhfsfzduLwl2k/
```

We also provide two JSON files that contain the above documents, posts.json and users.json, in case they are helpful. Note that in our provided data, the user "cs144"'s password is "password" and "user2"'s password is "blogserver".

Note that “created” and “modified” fields of the post documents are all integers, whose values represent the milliseconds since the the Unix epoch (Jan 1, 1970 UTC). You must store all date fields in this format. Also recall that the above password values are obtained by applying the bcrypt cryptographic one-way hash function.

Note that your provided script `db.sh` will be executed through the following command

```
$ mongo < db.sh
```

before we grade your submission to initialize the MongoDB database for the server.

Note::

Your db.sh script must strictly follow the instructions above. Please make sure that the database names, the collection names, and their contents are ***exactly the same as this instruction including their case***. Since MongoDB is ***CASE SENSITIVE***, your submission will fail our test script even for a minor case mismatch and lead to a very low grade.

Notes on CR/LF issue: If your host OS is Windows, you need to pay attention to how each line ends in your script file. Windows uses a pair of CR (carriage return) and LF (line feed) characters to terminate lines, but Unix uses only a LF character. Therefore, problems may arise when you feed a text file generated from a Windows program to a Unix tool such as `mongo`. If you encounter any wired error when you run your script, you may want to run the `dos2unix` command in the container on your script file to fix line end characters.

Part B: Implement Public HTML Blog Web Pages

As the second task of Project 3, we now implement the public URLs by which any user can view blog posts published on our website. In particular, we will implement a server that returns an HTML page to an HTTP request to the following URL patterns:

#	URL	method	functionality
1	`/blog/:username/:postid`	GET	Return an HTML-formatted page that shows the blog post with `postid` written by `username`.
2	`/blog/:username`	GET	Return an HTML page that contains first 5 blog posts (by `postid`) from `username`. If the user has more than 5 posts, the page should contain a “next” button (or hyperlink) that link to the next 5 posts (according to

			the postid) by the user.

Return status code `404` for the above routes if either the `:username` or `:postid` does not exist in the database.

We use node.js JavaScript runtime engine and its express module to implement our back-end server. Node.js is built on Chrome's V8 JavaScript engine, which uses an event-driven, non-blocking I/O model to make it lightweight and efficient. Express is a module that provides an easy-to-use routing mechanism, HTML template integration, and third-party middleware integration. You may want to go over the class lecture note on node.js to brush up on node.js and express. If you need more detailed instruction on how to use them, online tutorials like this can be helpful. Express web site has more detailed documentation on routing and using a template engine.

Generate Server Skeleton Code

To generate the skeleton code for our web server, we will use the Express application generator. Initialize the project under a project directory (e.g. `blog-server`):

```
$ express --view=ejs blog-server
```

Here `--view=ejs` option makes the generated code use the "EJS" template engine for generating HTML pages from JSON. *You are welcome to use other template engine for your development*, but our project spec gives instructions for "EJS". When the above command is executed, you see the following folder and file structure within the `blog-server` directory:

```
blog-server
 +- bin
 +- public
 +- routes
 +- views
 +- package.json
 +- app.js
```

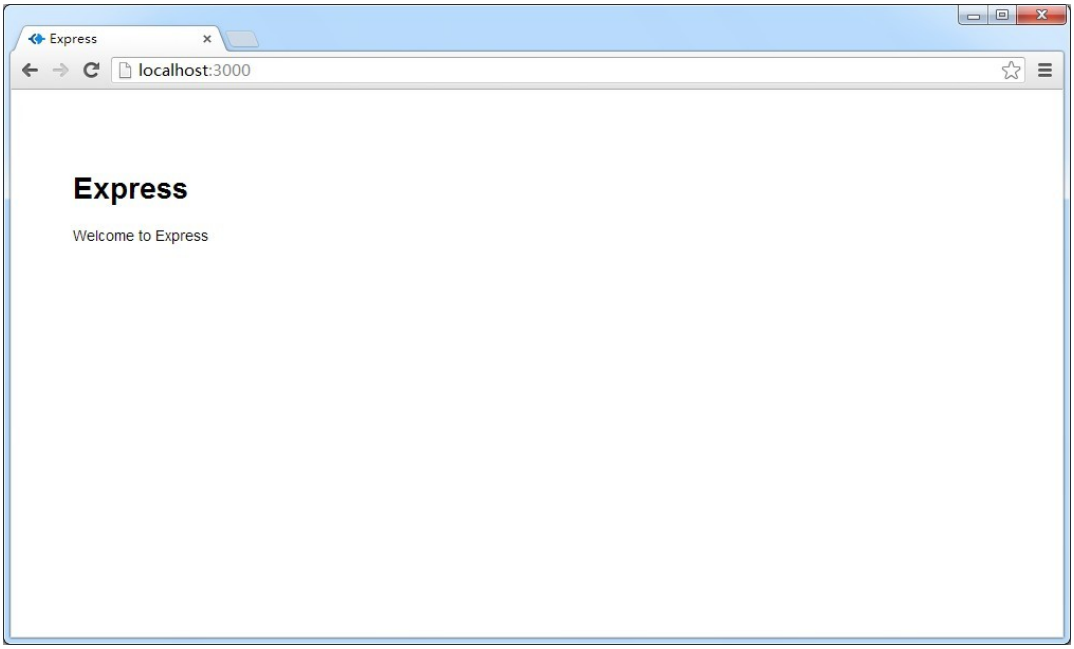
Here the `app.js` file serves as the entrance of the whole project. If you define other .js files, you need to reach them starting from app.js. The `package.json` file contains some meta information on the project including its "package dependencies". When you add new dependencies to this file and run `npm install`, `npm` will install all dependent modules into the subdirectory

`node_modules`. The `public` directory contains static resources that are made available on the developed web site, like HTML, CSS, JavaScript, and image files. The `views` directory contains HTML template files. The `routes` directory contains "middleware" that processes the requests. The `bin` directory contains the executable file. As you develop your code, you are welcome to add or modify any directory or file as needed.

Make sure that the generated code works properly on our container by executing the following commands:

```
$ npm install
$ npm start
```

The `npm install` command installs all dependent node packages specified in the `package.json` file. The `npm start` command executes "start script" specified in `package.json` (which is `node ./bin/www` in the auto-generated `package.json` file). Open a web browser on your host machine and make sure that you see a page similar to the following image at http://localhost:3000/:



Install mongodb and commonmark Modules

To generate public HTML blog pages using node.js, express, and MongoDB, we need a MongoDB client library for node.js and a markdown-to-HTML rendering library. We will use the official MongoDB driver for node.js and the commonmark.js library for this purpose. Install these two packages using `npm`

```
$ npm install mongodb
$ npm install commonmark
```

Note that commonmark.js's API is almost identical to the Java version, so you will find it easy to use.

In case you are not familiar with the MongoDB native client API, go over Using MongoDB with Node Tutorial and read the MongoDB tutorial on CRUD operations to learn the basics.

Note: Due to a strange interaction between node, Docker container, and shared folder, you may sometimes get an error similar to the following when you run `node` or `npm`:

```
path.js:1177
           cwd = process.cwd();
                         ^
Error: ENOENT: no such file or directory, uv_cwd
```

If you get the above error, you can "fix it" by `cd ../`, cd back, and try again.

Learn the Template Syntax

The responses to the two URL patterns shown above should be all in HTML. For generating an HTML response, you can use a *template engine*. When you use a template engine, you just need to write a static "template file", and "render" the final HTML response by combining the template with data.

EJS (Embedded JavaScript) uses a template syntax very similar to Java ServePages (JSP). Like JSP, you can use the standard HTML tags in the template, and sprinkle your JavaScript code inside `<% ... %>`, `<%= ... %>`, or `<%- ... %>` tags. `<%= ... %>` or `<%- ... %>` can include any expression and is replaced with the output string of the expression. The difference between the two is that `<%= ... %>` escapes HTML tags in the output, so that the HTML tags are displayed as strings, while `<%- ... %>` does not escape HTML tags, so that the browser can interpret them as HTML tags. `<% ... %>` can include any arbitrary JavaScript code, not just expressions.

Here is an example of a valid EJS template:

```
<ul>
    <% for(let i = 0; i < books.length; i++) { %>
        <li><%- books[i].isbn %></li>
    <% } %>
</ul>
```

For more EJS examples, look at the generated template files in the `views` directory. You may also want to go over an online tutorial like this to learn more on EJS.

Implement `/blog/:username/:postid` and `/blog/:username`

Now that you know the basics to implement Part B, go ahead and implement it. The responses from **these two URL patterns must meet the following requirements:**

1. All blog posts returned from the two URL patterns should be rendered in HTML from markdown using the `commonmark.js` module, both title and body.
2. The second URL pattern `/blog/:username` must return the first 5 posts (by `postid`) from `username`. When there are more posts from the user the returned page must contain a "next" link, which points to a page with the next 5 posts according to the postid by the user. ***Make sure that the "next" link is implemented as an HTML `<a>` element with `id="next_link"` and `href` pointing to the URL of the "next page".***
3. If `:username` is missing in the request or `:username` or `:postid` does not exist in the database, return status code `404`.
4. The second URL pattern `/blog/:username` must take an *optional query string* `start=:postid`, like

   ```
   /blog/cs144?start=3
   ```

   When this optional query string exists, the response must include the next 5 posts by `cs144` whose `postid` is 3 or above.

Note:

1. In implementing this part, remember that a request can be "routed" to a callback function through `app.METHOD(URL, callback)`, like `app.get('/blog/:username', callback)`, or using `app.use(path-prefix, callback)`. Inside the callback function, you can reference the HTTP request through the first req parameter, and you can generate the response

through the second res parameter.

2. In our project, all dates are stored as a number in MongoDB, which represents milliseconds since the the Unix epoch (Jan 1, 1970 UTC). You can convert a JavaScript `Date` object to this number using its `getTime()` method. Conversely, you can convert this number to a `Date` object by passing it as the constructor parameter of `Date` or by calling a date object's `setTime()` method.
3. Please remember that all MongoDB commands must be executed *asynchronously* either using callback functions or using a `Promise` object (potentially with `await` keyword).
4. To minimize the overhead from creating a connection to the database server, we strongly recommend that you create a connection to the MongoDB server when your application starts up and reuse the created connection for all MongoDB commands. This has been explained in Using MongoDB with Node Tutorial.
5. If you get an error like "listen EADDRINUSE: address already in use 3000" when you run your code, it is likely because you are calling `app.listen(3000)` multiple times in your code. Note that in the Express generated code, the `app.listen()` call is already made in the "bin/www" file. You either should not call `app.listen()` yourself or delete the existing `app.listen()` call in `/bin/www`.

Part C: Implement User Login Page

In this part, we will implement the user login page available at the following URL:

#	URL	method	functionality
1	`/login`	GET, POST	Authenticate the user through username and password.

Here are more detailed descriptions of the above API:

1. **GET /login**: The request may optionally include `redirect=:redirect` query string. Given the request, the server should return an HTML page that contains an HTML `<form>` with at least two `<input>` elements with names `username` and `password`. When the user inputs their username and password in these input elements, and presses the submit button, the page should issue a request `POST /login` with `username=:username&password=:password&redirect=:redirect` in the body, where the `redirect=:redirect` is added only if it was included as part of the `GET /login` request.

Note: Remember that when a browser generates an HTTP request from `<form>` fields with `POST` method, user inputs are included in the *body* of the generated request with `Content-Type: application/x-www-form-urlencoded`.

2. **POST /login**: The request body must contain `username=:username&password=:password`, optionally with `redirect=:redirect`, with `Content-Type: application/x-www-form-urlencoded`. Depending on the passed parameters, you need to take the following action: A. If the request does not have `username` or `password` parameter, the server must return the status code "401 (Unauthorized)". B. If the request has `username` and `password` parameters: i. If the provided `username` and `password` do not match our record, the server must return the status code "401 (Unauthorized)". The response body must include an HTML `<form>` with username and password `<input>` elements, so that the user can reattempt authentication. ii. If the provided `username` and `password` match our record, the server must set an authentication session cookie in JSON Web Token (JWT) (more on this later) and a. If the request does not contain `redirect` parameter, the server must return status code "200 (OK)" with the body saying that the authentication was successful. b. If the request contains `redirect` parameter, the server must redirect the browser to `:redirect` by returning the status code "302 (Found)" together with the HTTP header `Location:` pointing to the `:redirect` value from the request. Note that the MDN page on HTTP redirections explains different mechanisms for redirecting the browser to a different page automatically and the res.redirect() function can be used to generate `3XX` redirect response in Express.

Recall that our MongoDB server stores all users' credentials in the "Users" collection of the "BlogServer" database. In addition, recall that the stored user passwords are all hash values obtained by applying the bcrypt hash function. Therefore, to match a user's stored password against what she provides during authentication, you will need to use node.js bcryptjs module. Install the module through the following command:

```
$ npm install bcryptjs
```

Read the bcryptjs package page to learn how you can use it to compare a user's password against a hash value.

JSON Web Token (JWT)

Once the user's authenticity is established through the user-provided password, our server must establish a "authenticated session", so that it can recognize that any future request coming from the same browser comes from the authenticated user. There are a number of ways to implement this. In this project, you must use JSON Web Token (JWT) for this purpose. If you are not familiar with JWT, go over the JWT introduction page to learn what it is and how it can be used.

In your implementation, once the user is authenticated, you must set a *transient session cookie* (a cookie that has no expiration date, so that it is forgotten once the browser is closed) whose name is `jwt` and whose value is the following JWT:

1. Its header must have at least two claims, 'alg' (algorithm) and 'typ' (type), with the following values:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

2. Its payload must have at least two claims, 'exp' (expiration time) and 'usr' (user)

```
{
  "exp": expiration,
  "usr": "username"
}
```

   where `expiration` should be two hours from now (in ***seconds*** since Unix epoch, Jan 1, 1970) and `username` should be the authenticated username. Note that the unit of time here is seconds not milliseconds. According to JWT standard, this is how the expiration time should be represented.

3. The signature must be generated using the HS256 algorithm (HMAC-SHA256 hash function) with the following secret key

```
C-UFRaksvPKhx1txJYFcut3QGxsafPmwCY6SCly3G6c
```

Once this JWT cookie is set, our server will be able to recognize that any future request from the browser comes from the authenticated user `username`.

You can use the jsonwebtoken module to construct a JWT. Install it with the following command

```
$ npm install jsonwebtoken
```

and learn how to use it by going over examples in the jsonwebtoken module page.

Now implement the login page. Remember that after a successful authentication by the user (i.e., the user's username and password match), the server should generate an appropriate JWT, set it as the value of the cookie `jwt` (lowercase) and *redirect the request to the `redirect` parameter in the request if it exists.* If the authentication fails, (username or password is missing or they don't match with our record), the server must return a page with the username and password input box, so that they user can try again.

Remember that the user "cs144"'s password is "password" and "user2"'s password is "blogserver" in testing your authentication page.

Part D: Implement Blog-Management REST API

In this part, you will have to implement the REST API that will be used by an Angular-based editor (that will be implemented in Project 4) to save, retrieve, update, and delete the blog posts from the server.

#	URL	method	functionality
1	`/api/posts`	GET, POST, DELETE	This is the REST API used to insert, retrieve, update, and delete blog posts

Here are more detailed descriptions of the above API:

1. This REST API must be protected behind authentication. That is, if any request to `/api/posts` does not contain a valid jwt cookie with matching username (i.e., if the jwt cookie is not included in the HTTP header, if the included jwt has expired, or if the username in jwt does not match the username in the path or the body), the server should reply with "401 (Unauthorized)" status code.
2. **GET /api/posts?username=:username**: The server should return all blog posts by `:username`. The returned posts should be included in the body of the response as a JSON array with `Content-Type: application/json`. Each post in the array should have at least

five fields, `postid`, `title`, `body`, `created`, and `modified` (case sensitive). If the user does not exist or have no post, return an empty JSON array. If successful, the server should respond with the status code "200 (OK)".

3. **GET /api/posts?username=:username&postid=:postid**: The server should return the blog post with `:postid` by `:username`. If such a post exists, the response status code should be "200 (OK)", and the post should be included in a JSON body with at least five fields, `postid`, `title`, `body`, `created`, and `modified` (case sensitive). If not, the response status code should be "404 (Not found)".
4. **DELETE /api/posts?username=:username&postid=:postid**: The server should delete the existing blog post with `:postid` by `:username` from the database. If successful, the server should reply with "204 (No content)" status code. If there is no such post, the server should reply with "404 (Not found)" status code.
5. **POST /api/posts**: The request body should contain a JSON object (`Content-Type: application/json`) with four fields, `username`, `postid`, `title`, and `body`.
    1. **When `postid=0`**: The server should insert a *new* blog post whose `username`, `title`, and `body` values come from the request. The inserted `postid` must be the user's `maxid` (i.e., the `maxid` field of `username` in the `Users` collection) + 1, which should never be smaller than 1. The inserted `created` and `modified` fields must be the current time. The server must also increase the user's `maxid` field by 1 in the `Users` collection. If successful, the server should reply with "201 (Created)" status code with a JSON body with `postid`, `created`, and `modified` fields that reflect the inserted values.
    2. **When `postid>0`**: The server should update the *existing blog post* with `postid` by `username` with the `title` and `body` values from the request. The `modified` field should be updated to the current time as well. If the update is successful, the server should reply with "200 (OK)" status code with a JSON body that has a single `modified` field with the new modified value. If there is no blog post with `postid` by `username`, the server should reply with "404 (Not found)" status code.
6. All dates transmitted and stored in MongoDB should be in **milliseconds since the the Unix epoch (Jan 1, 1970 UTC) in number type**.
7. If a request does not meet our requirements (such as not including required fields, their values are in an incorrect format, etc.), the server must reply with "400 (Bad request)" status code.

Now add the appropriate routing instruction to your app and implement the above API. Make sure that the implementation works as intended before you proceed. You can test your implementation using the curl command (available within our Docker container) or Postman, which makes it easy to send an HTTP request to a server.

Once you made sure that everything works fine, protect this REST API behind authentication. That is, before you process any request to this API, first make sure that the request contains a valid jwt cookie with the matching username.

Note: Please make sure that you ***return the correct status code according to our spec. If your status code is diffrent from our spec you will get zero point for the part***, even if the response body may contain reasonable content.

What to Submit

Before you create the submission zip file for Project 3, please make sure that your server can be executed simply by running the following sequence of commands at the project root directory:

```
$ mongo < db.sh
$ npm install
$ npm start
```

Since this is how our autograder will run and test your code, it is essential that your code runs fine without any problem through the above commands. You may assume that the MongoDB server is running with no documents inside the “BlogServer” database when your your server is executed in the autograder.

After you have checked there is no issue with your project, package your work by running our p3_package script in your project root directory.

```
blog-server
 +- bin
 +- public
 +- routes
 +- views
 +- ...
 +- package.json
 +- app.js
 +- db.sh
```

When you execute the packaging script like `./p3_package` in the project root directory, it will check whether a few mandatory files are there, and package everything within the directory (except the files in `node_modules/`) and create a file named `project3.zip`. It will also perform basic sanity check to ensure that your code will run under our autograder. If everything goes smoothly, you will see something like the following at the end:

```
[SUCCESS] Created '/home/cs144/shared/blog-server/project3.zip'. Ready for submiss
```

Please only submit this script-created project3.zip to GradeScope. Do not use any other ways to package or submit your work!