|  |
| --- |
| FACULTY OF APPLIED SCIENCES AND TECHNOLOGY |
| **NODE/EXPRESS WEB API** |
| ITE5315 - Project |
|  |
| Mohaiminul Islam n01512929;  **Submission Date** |
| **4/10/2023** |

|  |
| --- |
| This document explains how to build Node/Express Web API …………………………. |

**Table of Contents**

[Question 1: 2](#_heading=h.gjdgxs)

[Question 2: 3](#_heading=h.30j0zll)

[Question 3: 4](#_heading=h.1fob9te)

[Question 4: 5](#_heading=h.3znysh7)

[Question 5: 6](#_heading=h.2et92p0)

[Question 6: 7](#_heading=h.tyjcwt)

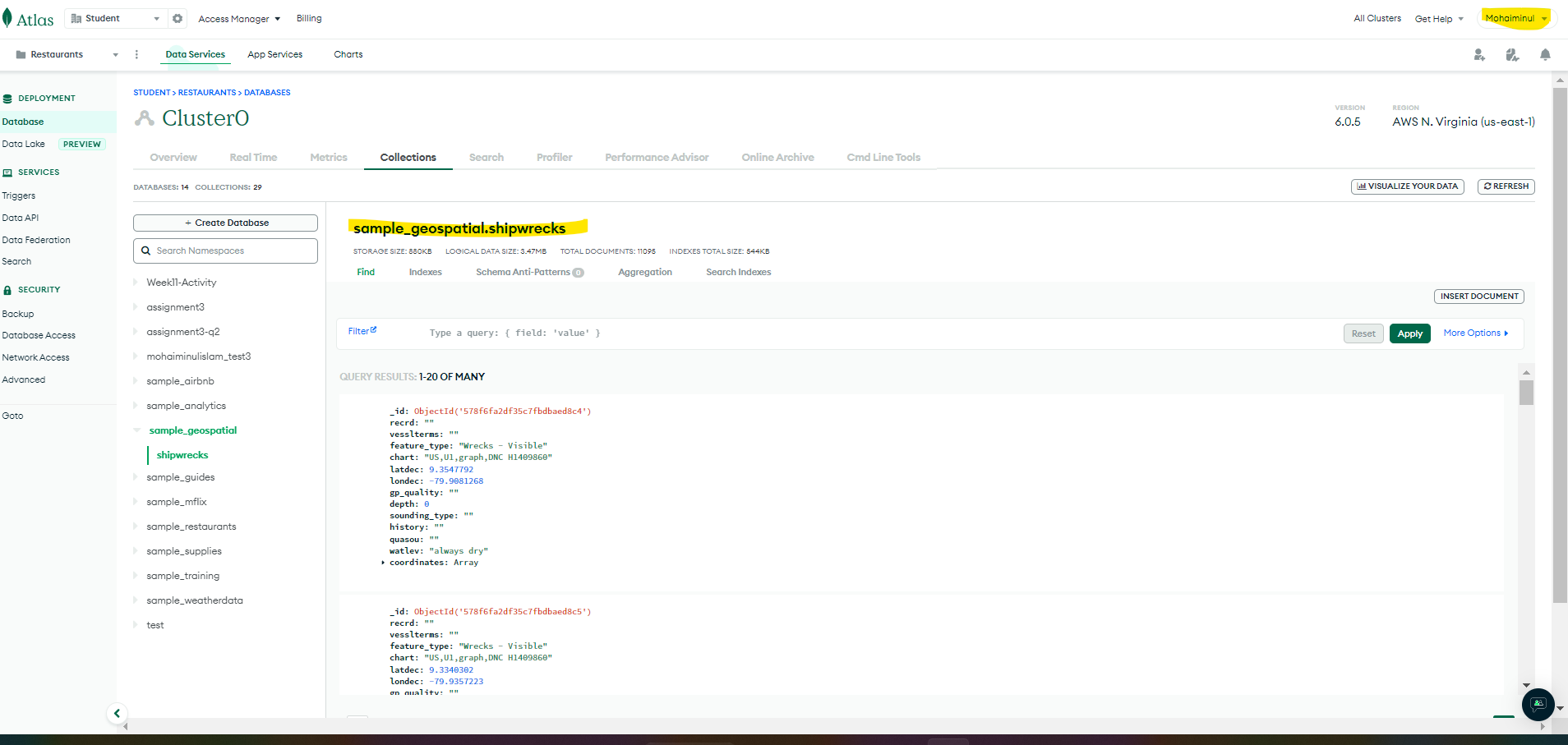
[Summary 8](#_heading=h.3dy6vkm)

# Question 1:

(Describe the major steps for implementing the MongoDB database in Atlas )

Major steps for implementing the MongoDB database in Atlas are as follows:

1. Sign up for an Atlas account: First we have signed up for a MongoDB Atlas account.
2. Create new project: After signing up we need to create a new project which will organize our MongoDB Atlas resources, including clusters, databases and users.
3. Create new MongoDB cluster: Now we need to create a cluster. It is a group of servers that store our data and provide access to it.
4. Configure network access: We need to configure network access to allow incoming connections to our MongoDB cluster.
5. Create database user: To access MongoDB cluster, we need to create a database user.
6. Connect to cluster: After creating user we can connect to our MongoDB cluster using a MongoDB client. Here we are using Node.js to connect to our cluster.
7. Create Database and Collections: Once we have connected to cluster we can create new Database and Collections. For this project we are using the database sample\_geospatial and collection shipwrecks. This is from sample data that we loaded to our Atlas.



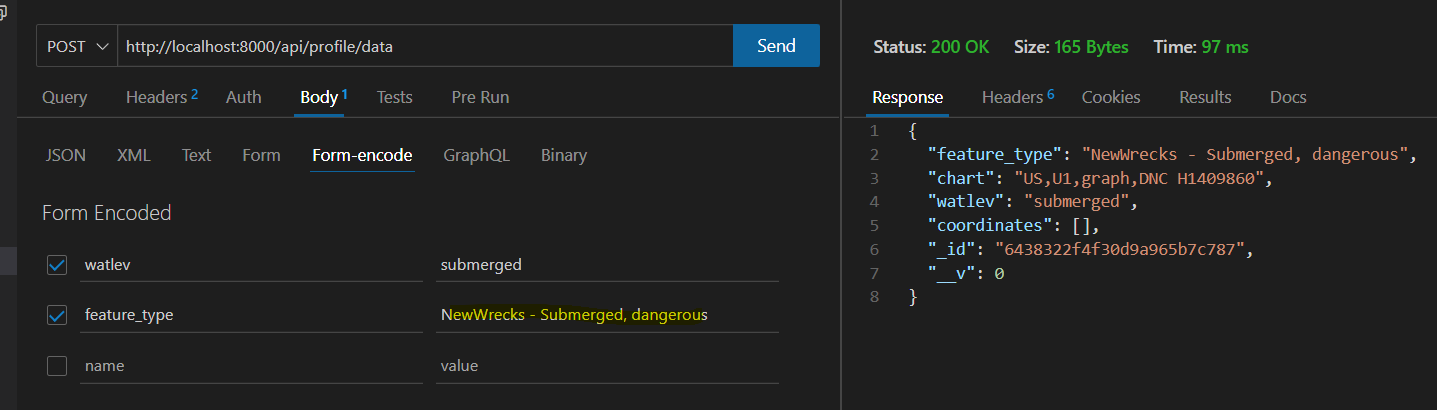
# Question 2:

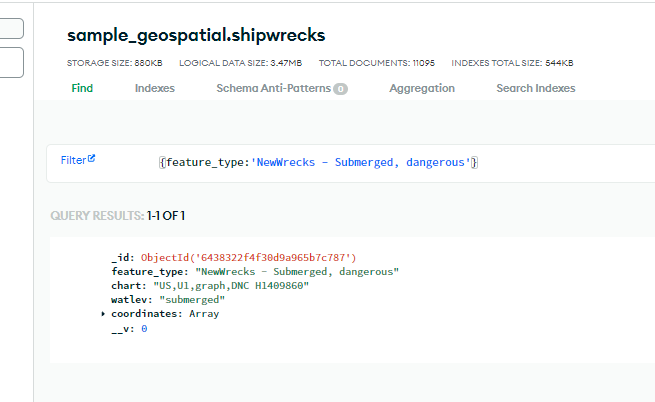
(Describe the major steps for implementing Routes in the API, how you test this program, add some screenshots of the output)

1. **POST /api/data:** First we are implementing a POST route for creating a new “Shipwreck” document in sample\_geospatial database. For this we took following steps:

* **Define the route:** Here the route is defined using the POST method and the route path is ‘/data’.
* **Define the callback function:** The callback function is the logic that gets executed when a POST request is made to the ‘/data’ endpoint. Here, the callback function receives the request object (req) and the response object (res).
* **Create a new Shipwreck object:** Next we create new Shipwreck object using the data from request body. Here watlev and feature\_type properties of the new Shipwreck object are set to the values received in the request body.
* **Save new Shipwreck object:** The new Shipwreck object is now saved to the MongoDB sample\_geospatial database using save() method. This method returns a promise which resolves with the saved Shipwreck object.
* **Handle the promise:** Once the promise is resolved, the saved Shipwreck object is sent back to the client as the response. If there’s an error during the save() operation, the catch block logs the error to the console.

Text

Description automatically generated

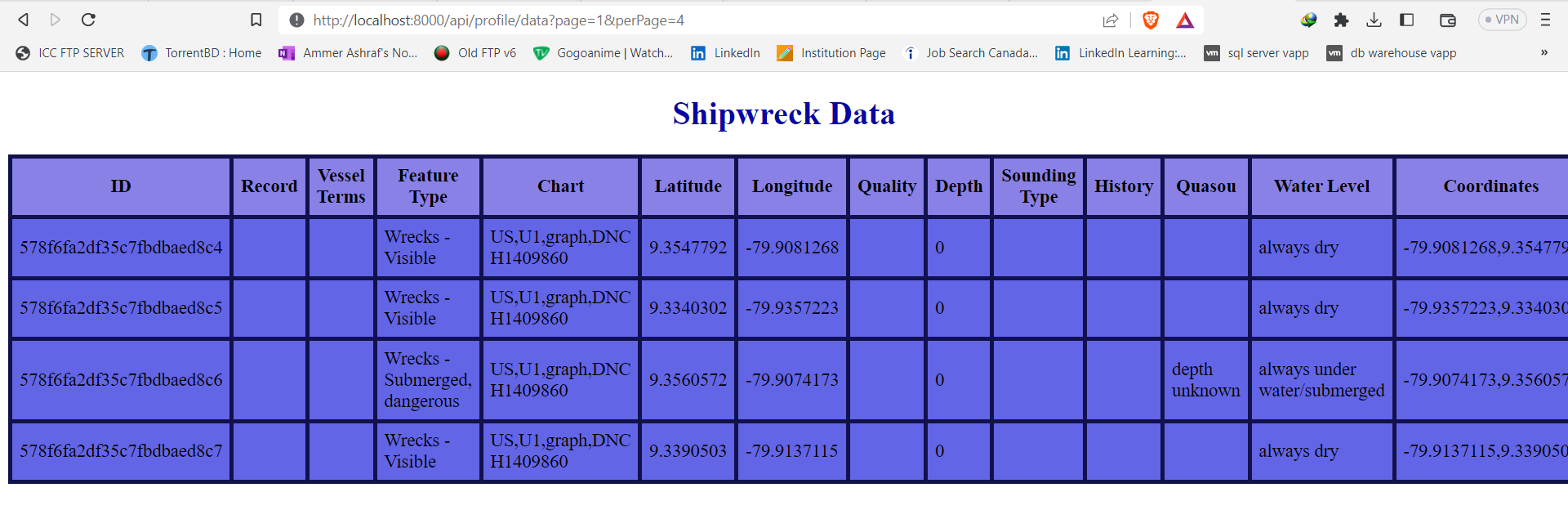


1. **GET /api/data?page=1&perPage=5&depth=4:** Here we are implementing GET route for retrieving a list of Shipwreck documents from our database with pagination and optional depth filter. Steps followed are:

* **Define the route**: Here the router is defined with GET method and the route path is ‘/data’.
* **Parse query parameters**: The code then parses the query parameters passed in the request url. Specifically, it parses the ‘page’ and ‘perPage’ parameters to determine the pagination of the results. If these parameters are not specified, default values of 1 and 10 are used respectively. It also parses the depth parameter to filter the results by depth if it is specified.
* **Construct the query**: The next step is to construct the MongoDB query based on the parsed parameters. If the depth parameter is specified, a query object is constructed with a depth field set to the specified value. Otherwise, an empty query object is used to retrieve all shipwreck documents.
* **Query the Database**: Now the Shipwreck model is used to query the database using the constructed query. The find() method is used with a query object to retrieve the Shipwreck documents matching the specified criteria. The skip() and limit() methods are used to paginate the results based on the ‘page’ and ‘perPage’ parameters.
* **Handle the promise**: Once the find() method is executed, it returns a promise which resolves with an array of Shipwreck documents. If the query was successful, the response body is set to the array of Shipwreck documents.

Text

Description automatically generated

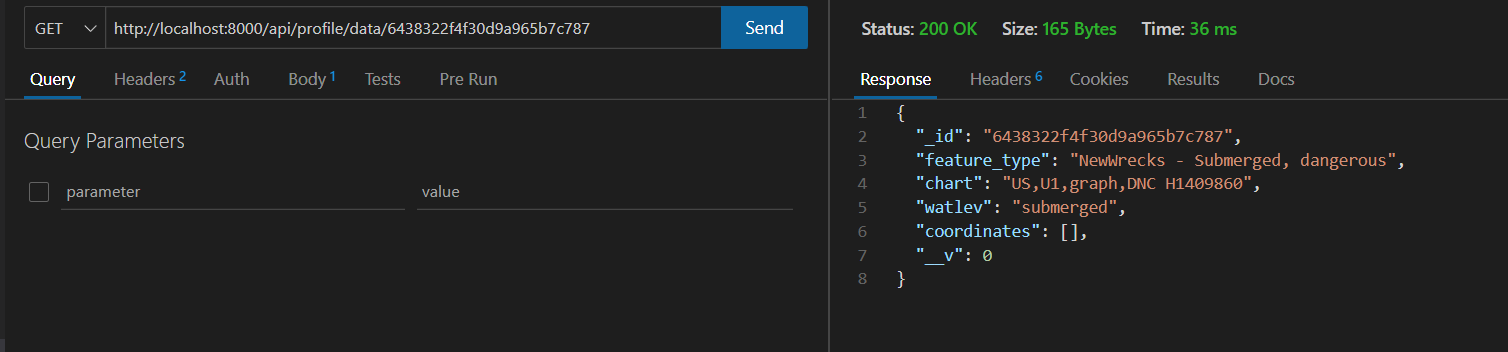


1. **GET /api/data/:\_id:** Now we are implementing a GET route for retrieving a single Shipwreck document from our database. The major steps taken are:

* **Define the route:** Like before here the router is defined with GET method and the route path is ‘/data/:\_id’.
* **Define callback function:** Here also we are defining callback function as in first route we did.
* **Retrieve the Shipwreck document:** The next step is to use findOne() method of the Shipwreck model to retrieve the Shipwreck document with the specified ‘\_id’. The value of \_id is obtained from the url parameter using req.param.\_id.
* **Handle the promise:** Once the findOne() method is executed, it returns a promise which resolves with the retrieved Shipwreck document. If the document is found , it is sent back to the client as the response. If the document is not found the response will be empty.

Text

Description automatically generated



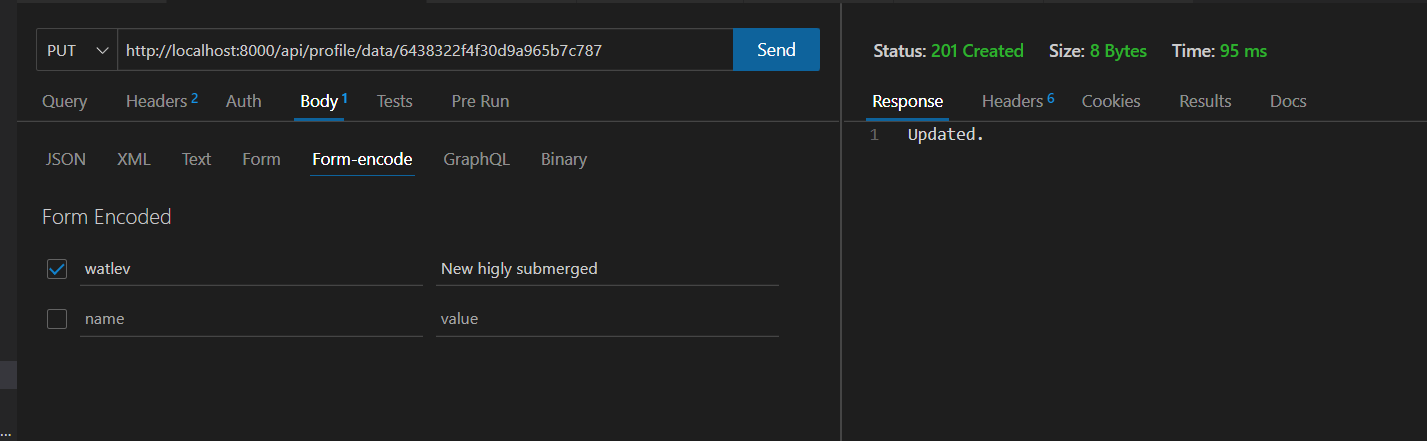
Here we fetched the new data we created in step 2 using its id value.

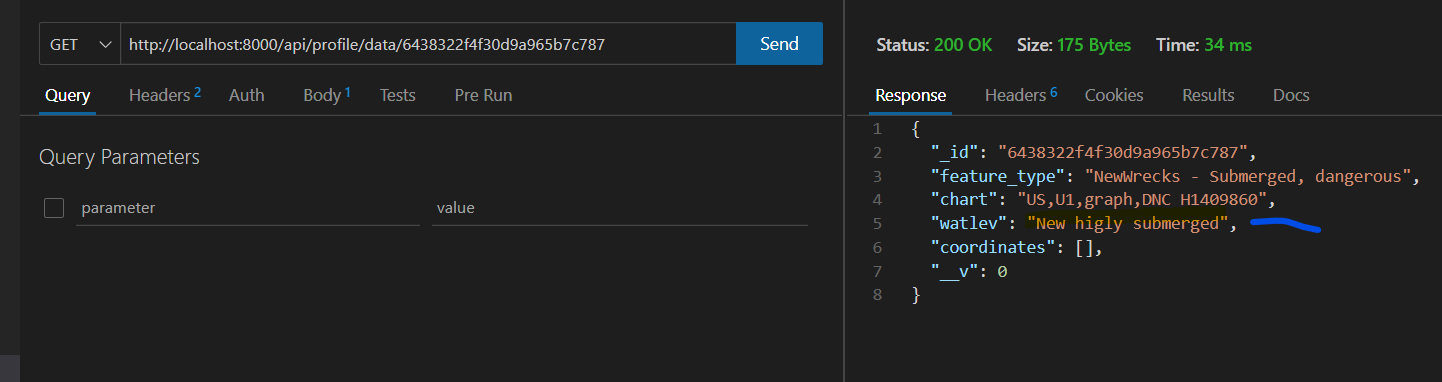
1. **PUT /api/data/:\_id:** Next we are implementing a PUT route for updating an existing Shipwreck document in our database.

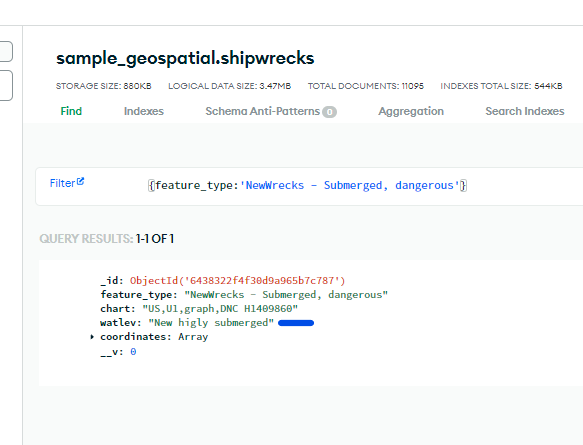
* **Define the router:** The router is defined with PUT method and the route path is ‘/data/:\_id’.
* **Define callback:** Like above we defined the callback.
* **Update the Shipwreck document:** We used the updateMany() method of the Shipwreck model to update the Shipwreck document with the specified \_id.
* **Handle the promise:** Once the updateMany() method is executed, it returns a promise which resolves if the update was successful. If update is successful, the response status code is set to 201(Created) and the response body is set to ‘Updated’

A screenshot of a computer

Description automatically generated with medium confidence

****





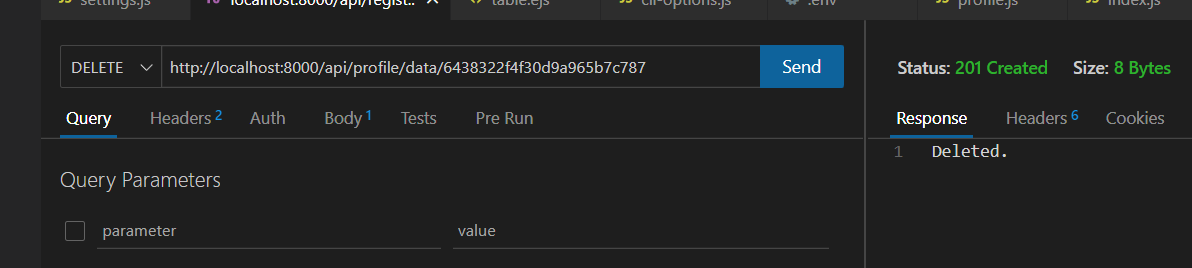
We updated the watlev of the newly added data and shoed that update was applied.

1. **DELETE /api/data/:\_id:** Finally we are implementing a DELETE route for deleting an existing Shipwreck document in our database. We took the following steps for this:

* **Define the router:** First and foremost like above we defined a router with DELETE method. The route path is ‘/api/data/:\_id’.
* **Define the callback function:** next we defined the callback function.
* **Delete Shipwreck document:** Now the deleteOne() method of the Shipwreck model is used to delete the Shipwreck document with the specified \_id. The \_id value is obtained from the URL parameter using req.param.\_id.
* **Handle the promise:** Once the deleteOne() method is executed, it returns a promise which resolves if the delete was successful.

Text

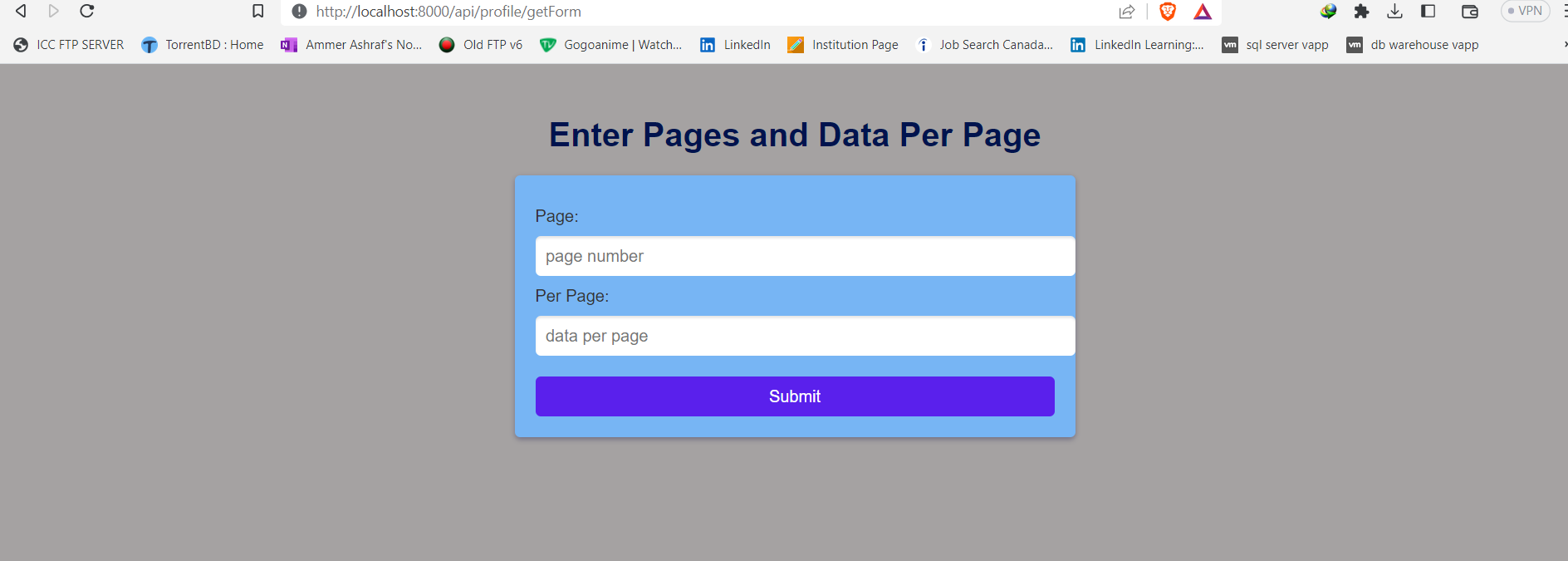
Description automatically generated

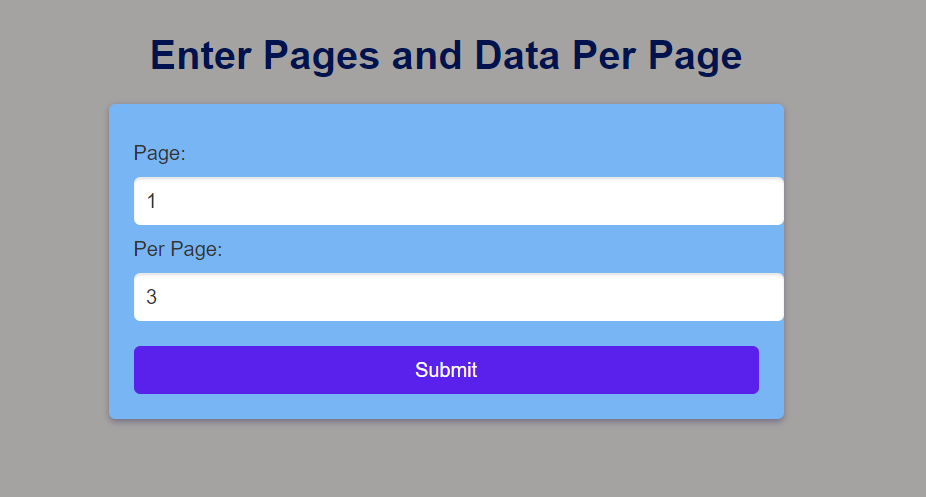


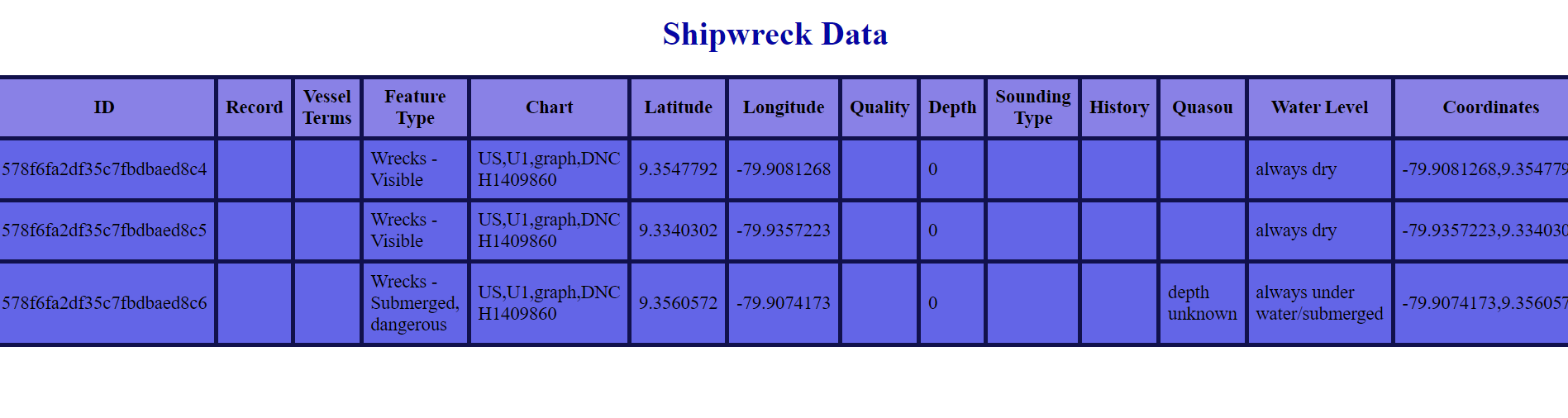
We deleted the new document we created at the beginning here.

# Question 3:

(Describe the major steps for designing the FORM/UI, how you test this program, add some screenshots of the output)





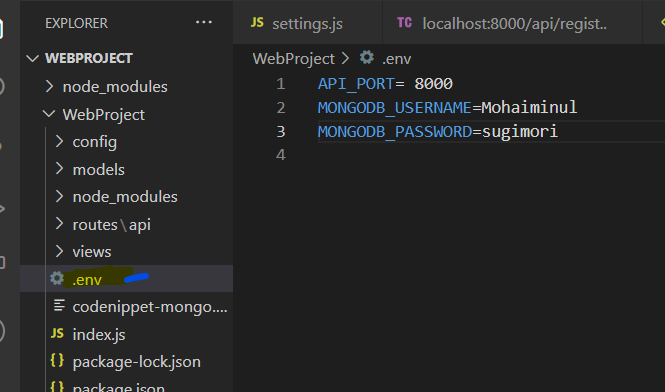


# Question 4:

(Describe the major steps for setting up environment variables.)

Solution:

1. Create a **.env** file.
2. Create a new file in the root of your Node.js project and name it **.env**. This file stores all environment variables.
3. Add variables to the **.env** file.
4. Open the **.env** file and add variables in the format VARIABLE\_NAME=variable\_value. We can add as many variables as we like and use different types of values ​​(strings, numbers, booleans, etc.).



1. Install the **dotenv** package. The dotenv package allows it to load environment variables from a .env file into her Node.js application. You can install it with npm by running the following command in your terminal:

**→ npm install dotenv**

1. Load environment variables.Node.js applications should use the dotenv package to load environment variables. Add the following lines of code to the top of your app.js or server.js file.

**→ require('dotenv').config();**

1. This line loads environment variables from the .env file and makes them available to your application. Use environment variables in your code.

Once the variables are loaded, you can use them in your code by accessing them through the process.env object. For example, if you have a variable named PORT in your .env file, you can access its value in code like this:

**→ const port = process.env.PORT || 3000;**

1. Secure your environment variables: Since environment variables can contain sensitive information like passwords or API keys, it's important to protect them. Make sure not to commit your .env file to your code repository, and avoid logging or printing environment variables in plain text. You can also use encryption or other security measures to protect your variables.

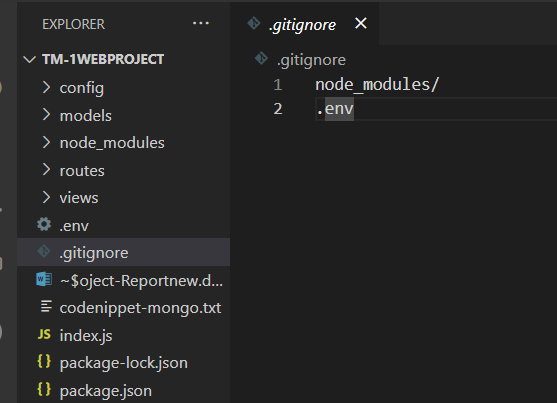
# Question 5:

(Describe the major steps for implementing security features, how you test this program, add some screenshots of the output.)

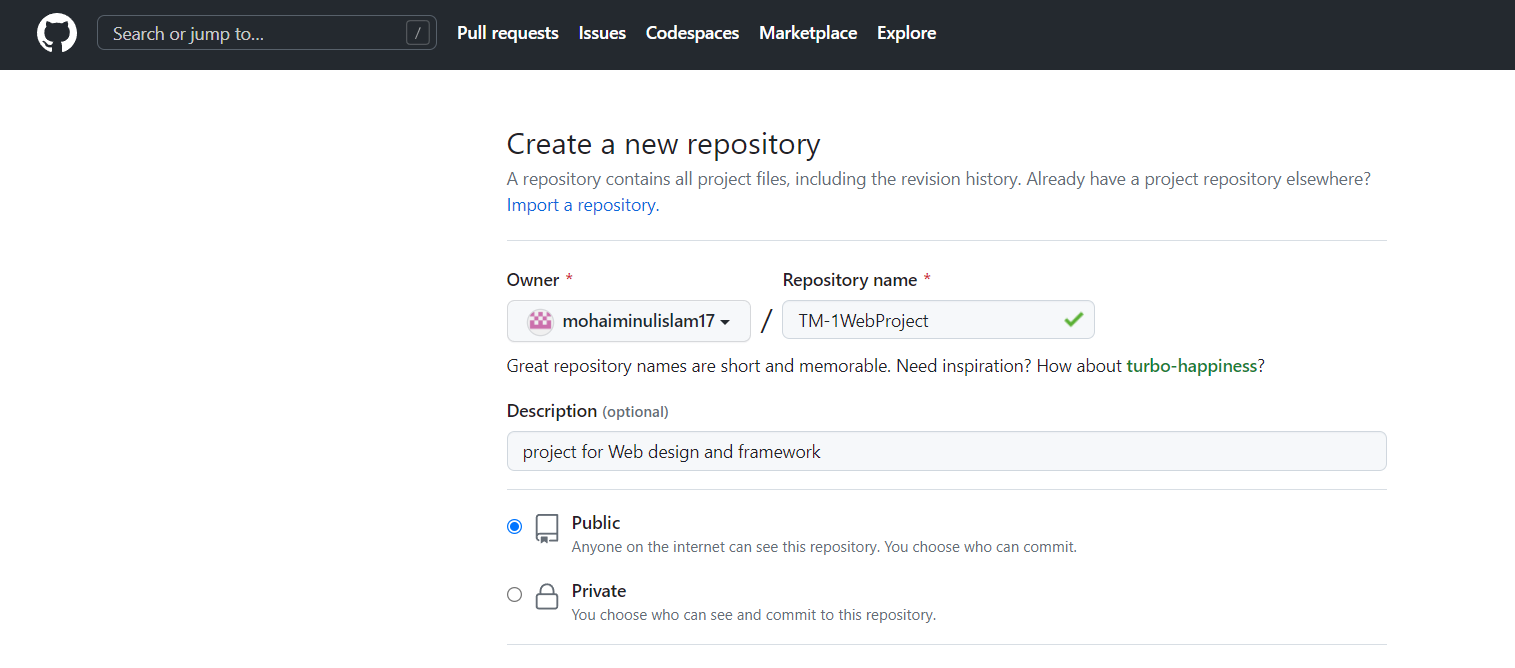
# Question 6:

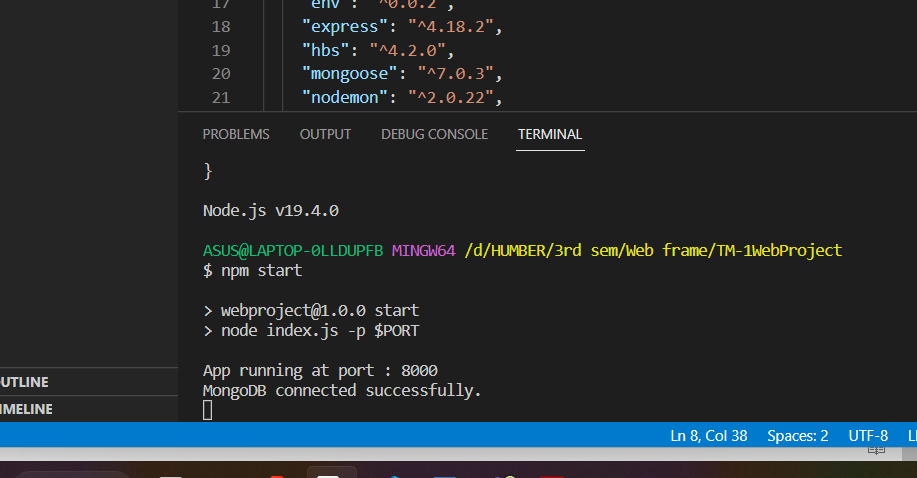
(Describe the major steps for deployment)

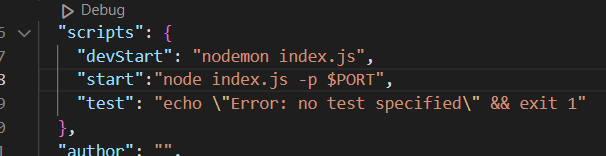
* + - 1. We need to create file .gitignore

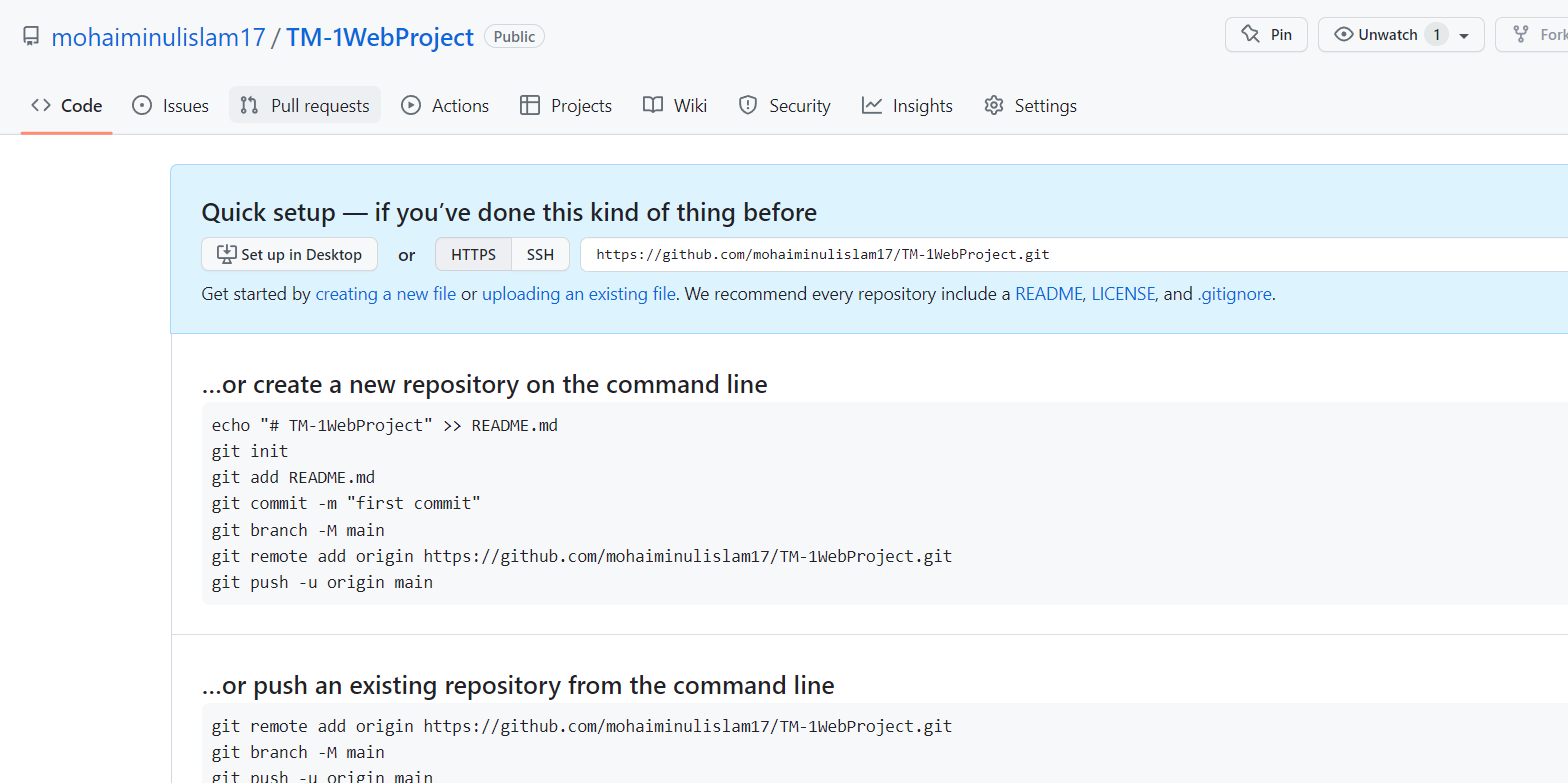


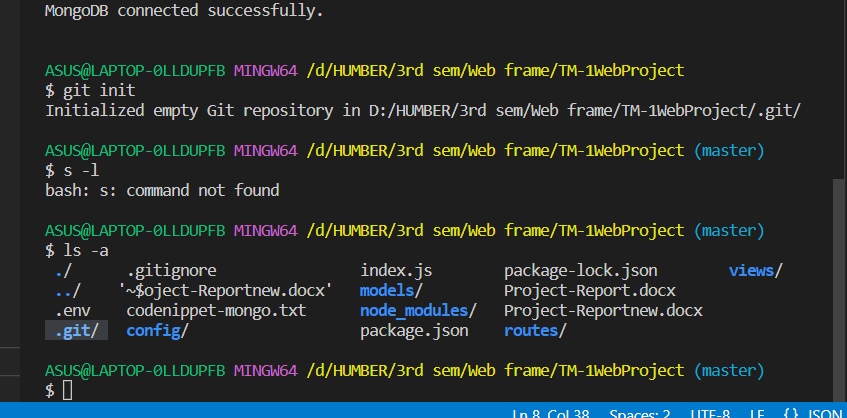
* + - 1. First, we need to create new repository in GitHub

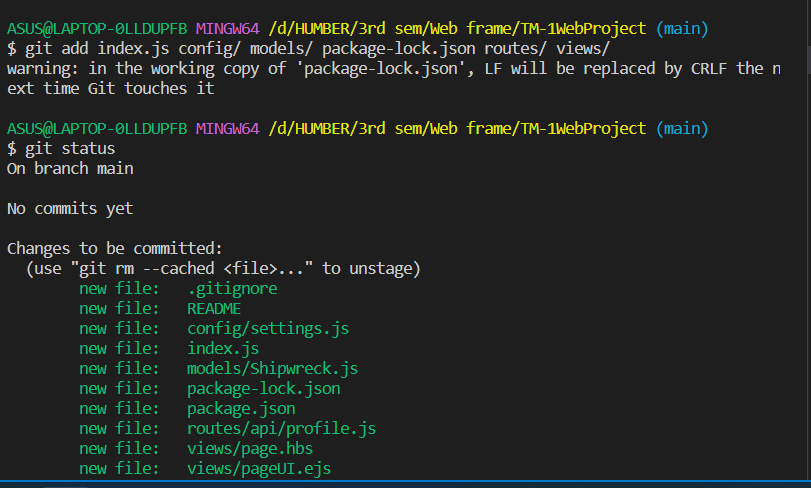


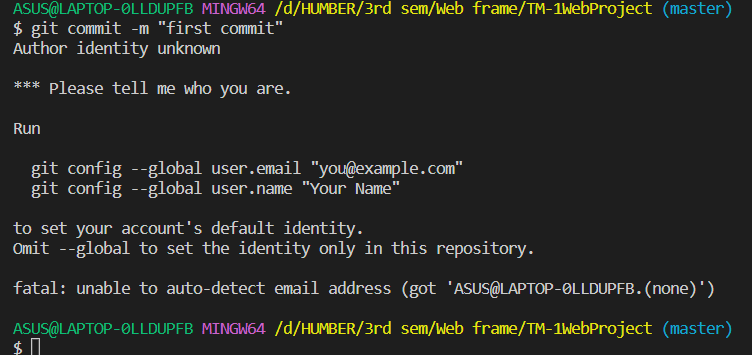


****

****

****

****

****

# Summary

(Describe how did you divide the work, share your feedback about this project like new points that you learn, challenges, …)

Completing this project within such a very short span of time was very challenging. It was especially challenging because we had back-to-back assignments and tests in the days leading to the project due date. However, being able to work in groups eased the task a bit since we were able to divide the tasks and use multiple approaches to solving the problem.

Although we divided the work, we did not completely work on the assigned questions individually. We were doing everything together virtually, bouncing ideas off of each other from the beginning till the end of the project.

**Challenges**

**Learnings**

We cannot deny the fact that the project was pretty challenging. However, at the same time we learned a few new things as well.