CS 628: Full-Stack Development – Web App

City University of Seattle
School of Technology & Computing
Professor Sam Chung

HOS08: Node, Express, and Express Router

Samantha Hipple
August 20, 2023

PLEASE NOTE

Screenshots in this guide may differ from your environment (e.g., directory paths, version numbers, etc.). When choosing between a stable or most recent release, we advise you install the stable release rather than the best-testing version. Additionally, there may be subtle discrepancies along the steps, please use your best judgment to complete the tutorial. If you are unfamiliar with terminal, command line, and bash scripts, we recommend watching this video prior to moving forward with this guide. Not all steps are fully explained. Lastly, we advise that you avoid copy-pasting code directly from the guide or GitHub repositories. Instead, type out the code yourself to improve familiarity.

More information on this guide can be found under the related module in this repository.

SECTION CONTENTS

SECTION 1. ACCESSING GITHUB CODESPACES

GitHub Codespaces is an online cloud-based development environment that allows users to easily write, run and debug code. Codespaces is fully integrated with your GitHub repository and provides a seamless experience for developers. In order to access Codespaces, users only need a GitHub account and an active internet connection.

After downloading the current HOS assignment, in the top-right corner of the repo, click on the **<> Code** drop-down menu and select **Create codespace on main** as shown in the following image. The free and pro GitHub subscriptions include free use of GitHub Codespaces *up to a fixed amount of usage each month*. In order to avoid unexpected charges, please review the billing information.

SECTION 2. PROJECT INITIALIZATION

Node.js is an open-source, server-side, runtime environment that enables JavaScript (JS) code execution on the server. Node is designed to assist developers in building scalable and efficient network applications. In this guide, we will be creating the backend for an application. Let us first check the node version installed in your development environment.

1. Use the command, **`node -v`** , in your terminal to see the current version in use:

```
\cs628-summer-hos08-Hipples> node -v
v18.17.1
```

2. Create a backend folder under the root directory using the following terminal commands:

```
>> mkdir backend
>> cd backend
```

3. Install the required dependencies with the following terminal commands:

```
>> npm install mongodb express cors dotenv
```

The installation of **mongodb** includes the MongoDB database driver, enabling your node.js applications to establish connections with the database service and manage your data effectively. Installing **express** adds the node.js web framework to your environment. Express is a rapid, versatile, and streamlined platform that equips developers with a set of tools and features to easily create web and API applications. The **cors** package enables cross-origin resource sharing, providing a secure way to share resources across different origins. Lastly, installing **dotenv** adds a module that automatically loads environment variables from a **.env** file into the **process.env** object. This practice of separating configuration information from the code fosters a cleaner and more organized development approach.

4. View the installed dependencies by checking your **package.json** file:

app > backend > {} package.json > ...

```json
{
  "dependencies": {
    "cors": "^2.8.5",
    "dotenv": "^16.3.1",
    "express": "^4.18.2",
    "mongodb": "^5.7.0"
  }
}
```

5. Create a file called **server.mjs** and add the following code:

```
import express from 'express';
import cors from 'cors';

import './load-environment.mjs';
import records from './routes/records.mjs';
```

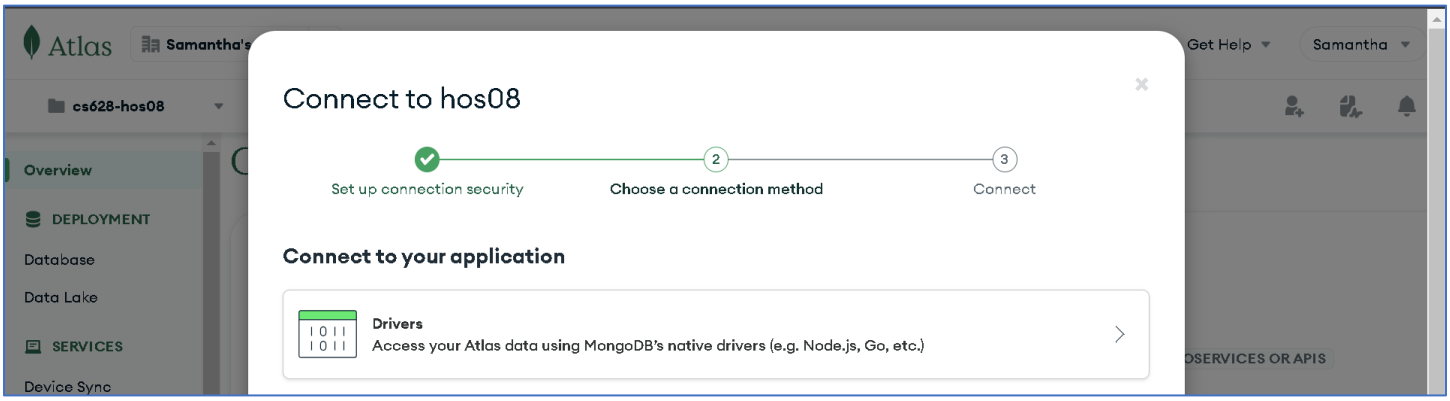
```
const PORT = process.env.PORT || 5050;
const app = express();

app.use(cors());
app.use(express.json());
app.use("/record", records);

// start the express server
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

For those curious about the **.mjs** extension, it's essential to recognize that both **.js** and **.mjs** extensions are used in Node.js to execute JavaScript code. However, they handle modules differently. The **.js** files adopt the CommonJS module system in Node.js, using **require()** for importing modules and **module.exports** for exporting them. On the other hand, **.mjs** files employ the ES Modules (ESM) system, leveraging the **import** and **export** statements for module operations. In the code provided above, we are importing **express** and **cors**. The statement **const port = process.env.port** retrieves the **PORT** variable from **config.env**.

SECTION 3. CONNECTING TO MONGODB ATLAS

In order to connect to MongoDB Atlas, we first need a connection string.

1. Log in to your MongoDB Atlas account.
2. Head to the overview section and click on **CONNECT**, then **Drivers**.



3. Copy and paste the connection string, then update the password.
4. Create a file called **config.env** and store the updated connection string as shown:

```
app > backend > config.env
1   ATLAS_URI=mongodb+srv://hos08-sami:puVQwFy35yQrfDIZ@hos08.0au8czo.mongodb.net/?retryWrites=true&w=majority
```

5. Next, create a file called **load-environment.mjs** and add the following code:

```
import dotenv from 'dotenv';

dotenv.config({ path: "./config.env" });
```

6. Within **backend**, create a new directory called **db**.

7. Within **db**, create a new filed called **conn.mjs** and add the following code:

```
import { MongoClient } from 'mongodb';

const connectionString = process.env.ATLAS_URI || "";

const client = new MongoClient(connectionString);

const connectToDB = async () => {
  try {
    const conn = await client.connect();
    return conn.db("hos08");
  } catch (error) {
    console.error(error);
    throw error;
  }
}

const db = connectToDB();

export default db;
```

SECTION 4. DEFINING SERVER API ENDPOINTS

1. Within **backend**, create a new directory called **routes**.
2. Within **routes**, create a new file called **records.mjs** and add the following code:

```
import express from 'express';
import { ObjectId } from 'mongodb';

import db from '../db/conn.mjs';

const router = express.Router();

// Get a list of all records
router.get("/", async (req, res) => {
  const collection = await db.collection("records");
  const result = await collection.find({}).toArray();
  res.status(200).send(result);  // code: OK
});

// Get a single record by id
router.get("/:id", async (req, res) => {
  const collection = await db.collection("records");
  const query = { _id: new ObjectId(req.params.id) }
  const result = await collection.findOne(query);

  if (!result) res.status(404).send("Not found");  // code: Not Found
  else res.status(200).send(result);
});
```

```
// Add a new record
router.post("/", async (req, res) => {
  const newDocument = {
    name: req.body.name,
    position: req.body.position,
    level: req.body.level
  }
  const collection = await db.collection("records");
  const result = await collection.insertOne(newDocument);
  res.status(201).send(result);  // code: Created
});

// Update a record by id
router.patch("/:id", async (req, res) => {
  const query = { _id: new ObjectId(req.params.id) }
  const updates = {
    $set: {
      name: req.body.name,
      position: req.body.position,
      level: req.body.level
    }
  }
  const collection = await db.collection("records");
  const result = await collection.updateOne(query, updates);
  res.status(200).send(result);
});
```

```
// Delete a record by id
router.delete("/:id", async (req, res) => {
  const query = { _id: new ObjectId(req.params.id) }
  const collection = await db.collection("records");
  const result = await collection.deleteOne(query);
  res.status(204).send(result);  // code: No Content
});

export default router;
```

This code defines a **router** that allows for CRUD (Create, Read, Update, Delete) operations on a collection named **records**, leveraging the Express Router feature from the Express web application framework in Node.js.

The **router** is composed of several routes: the first route retrieves all records from the **records** collection in response to a GET request at the root path (**"/"**). The second route handles a GET request with a parameterized route (**"/:id"**) and fetches a single record based on the provided ID. The third route accommodates a POST request, facilitating the addition of a new record to the **records** collection. Finally, the fourth route responds to a PATCH request (**"/:id"**), updating a specific record identified by the given ID with the data provided in the request body.

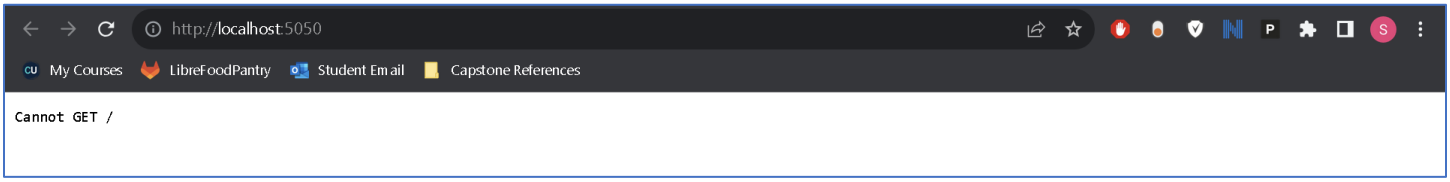
3. Run the server using the following terminal command:

```
>> node server.mjs
```

4. If everything is working properly, this message will appear in the console:

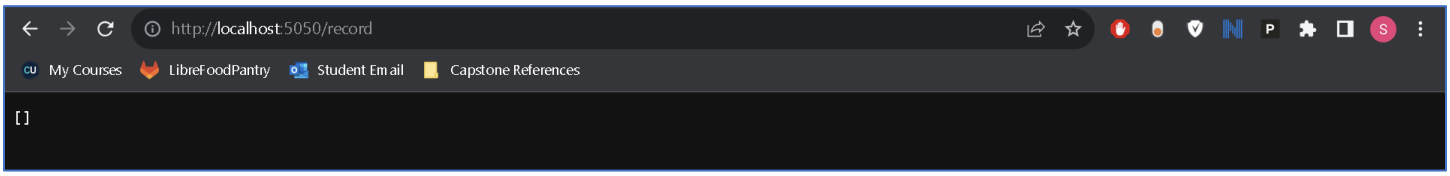
```
PS C:\Users\Saman\Desktop\CS 628\HOS\cs628-summer-hos08-Hipples\app\backend> node server.mjs
Server is running on http://localhost:5050
```

5. Click on the **Ports** tab next to the terminal and open the local URL in your browser.
6. The following message should be displayed:



In our **server.mjs** file we are using the slug **/record** to mount the records router to a specific route path within the Express application. This means that whenever a request is made to the **/record** route, the functionality defined in the records router will be executed.

7. Append **/record** to the localhost port and refresh the browser:



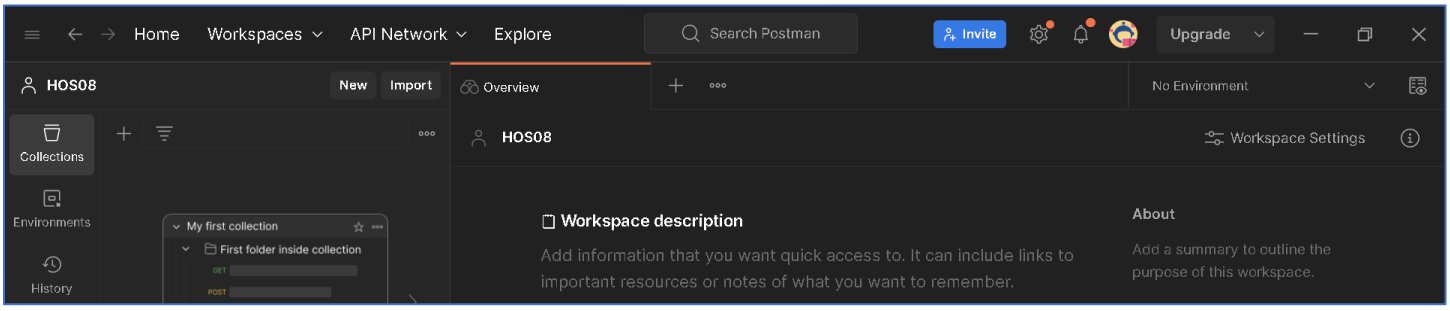
The error should go away, and an empty array will be returned. This is expected as we have not inserted any records into our database.

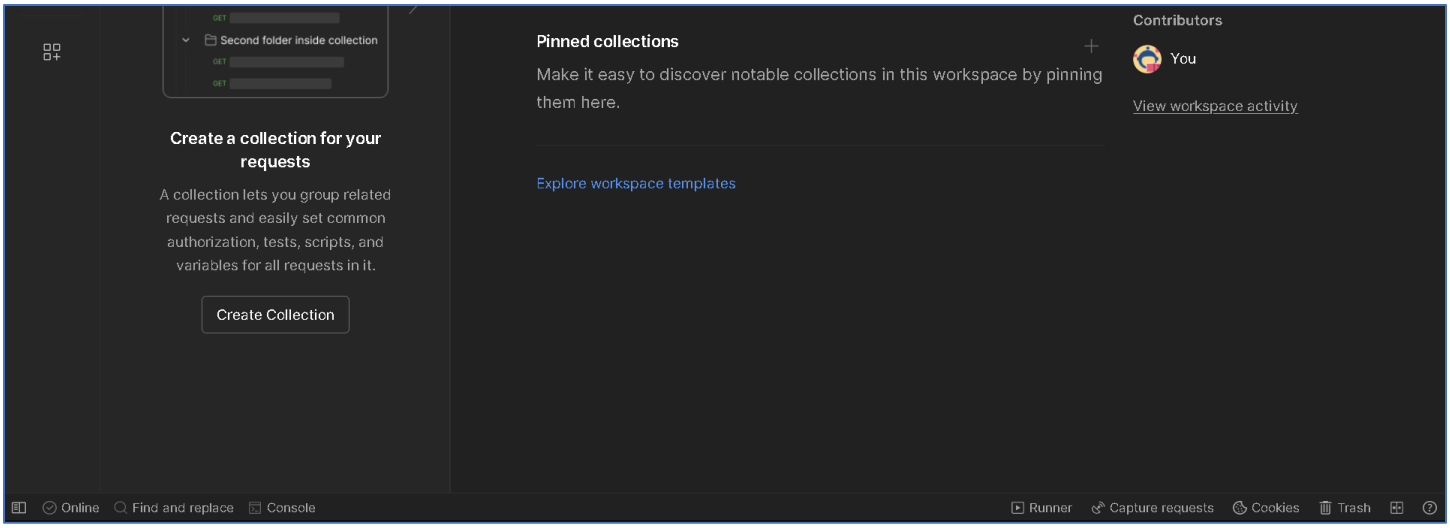
SECTION 5. TESTING API ENDPOINTS WITH POSTMAN

Postman is a widely-used tool for testing, developing, and documenting APIs. It streamlines the process of working with APIs by offering a user-friendly interface that enables developers to perform HTTP requests, view responses, and test various API functionalities. To test your API, you'll need to download and install the Postman app and make the port public to allow Postman to access it. After signing into the Postman app, you can create a collection and add requests for different operations like creating, fetching, deleting, and updating records.

We will now test our API endpoints using the Postman tool. If you do not already have Postman installed, you can download the application at https://www.postman.com/. Once Postman is installed, open the application and create an account or sign in via Google.

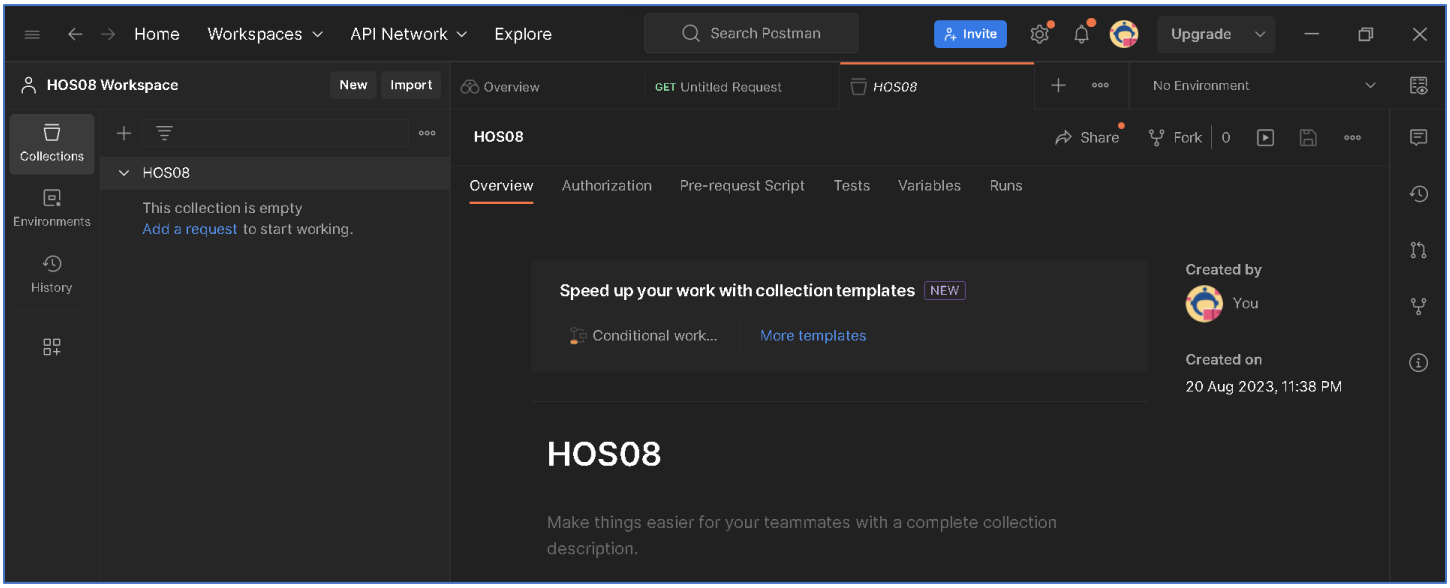
1. Navigate to a new Workspace in Postman:



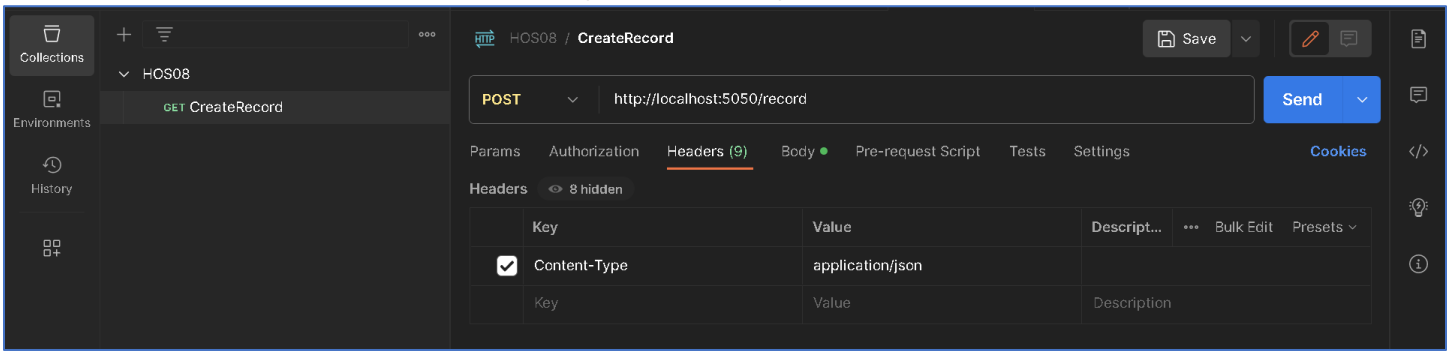


NOTE: *If you are using GitHub codespace, you will need to go to the **PORTS** tab next to the **TERMINAL** tab and right-click to access its context menu and set the port number 5050 as public so that Postman can access the API.*

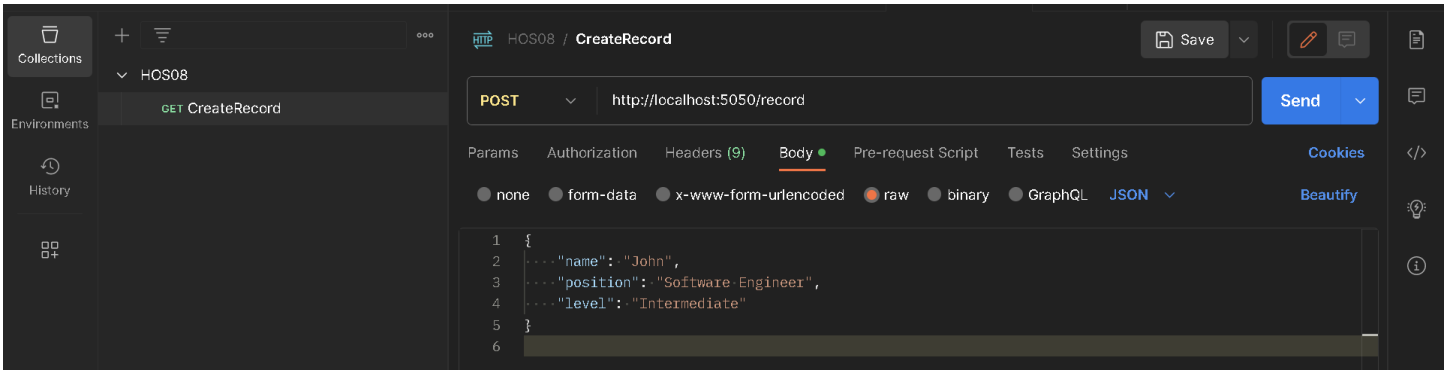
2. Create a new collection in the workspace and name it **HOS08**:



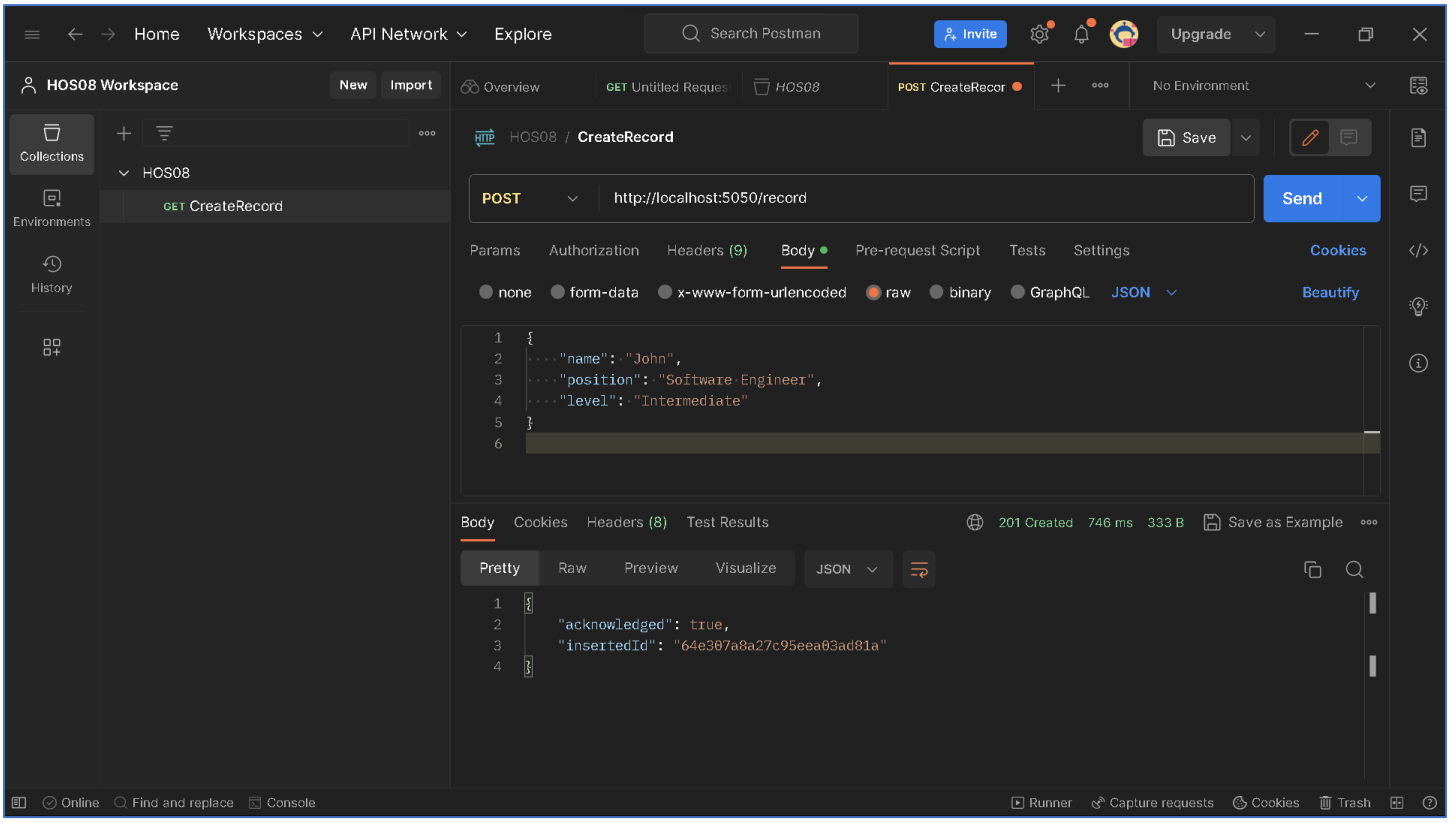
3. Click on **Add a request** to create your first request named **CreateRecord**.
4. Set the method to **POST** and enter the URL used earlier to access the API records.
5. Under **Headers**, enter **Content-Type** as the **Key** and **application/json** as the **Value**:



6. Under **Body**, select **Raw** and using **JSON**, enter the request body as shown below:

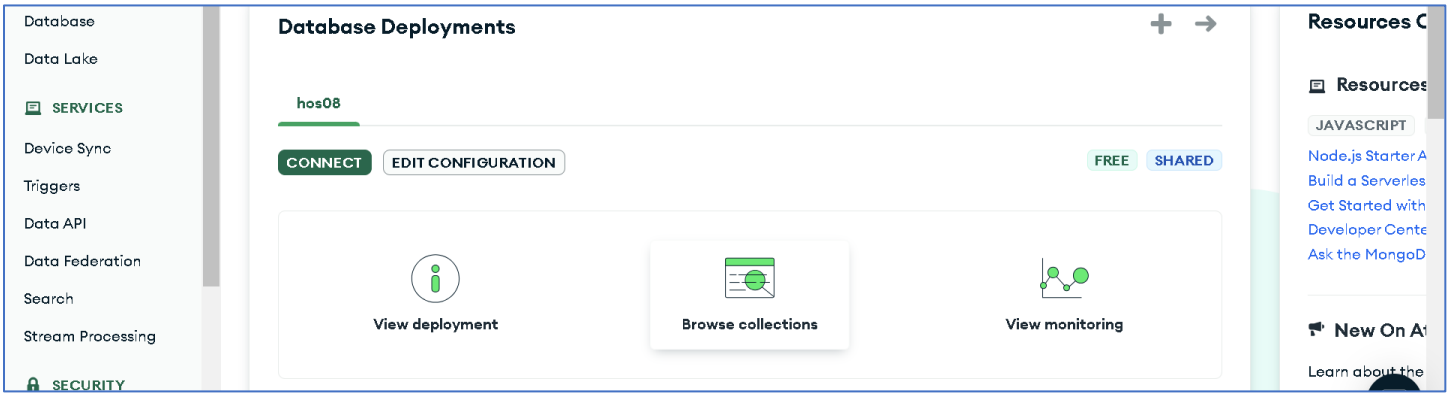


7. Click on **Send** to insert our first record and see the response body below:

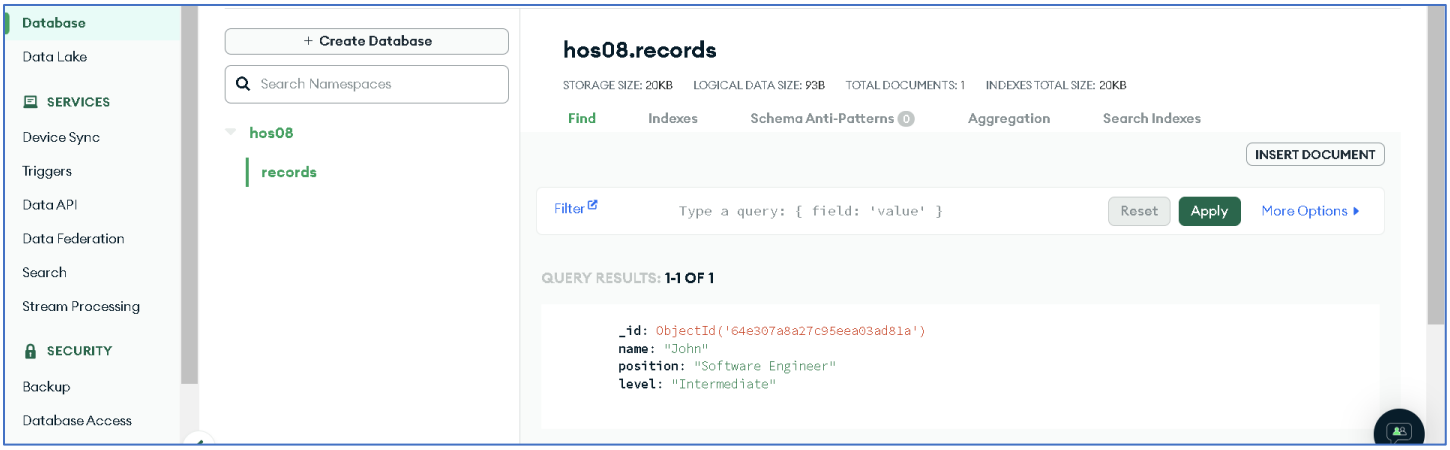


We can now head to our MongoDB Atlas account to verify that our document was successfully inserted into our records collection in our HOS08 cluster.

8. Once you have navigated to the appropriate cluster, click on **Browse collections**:



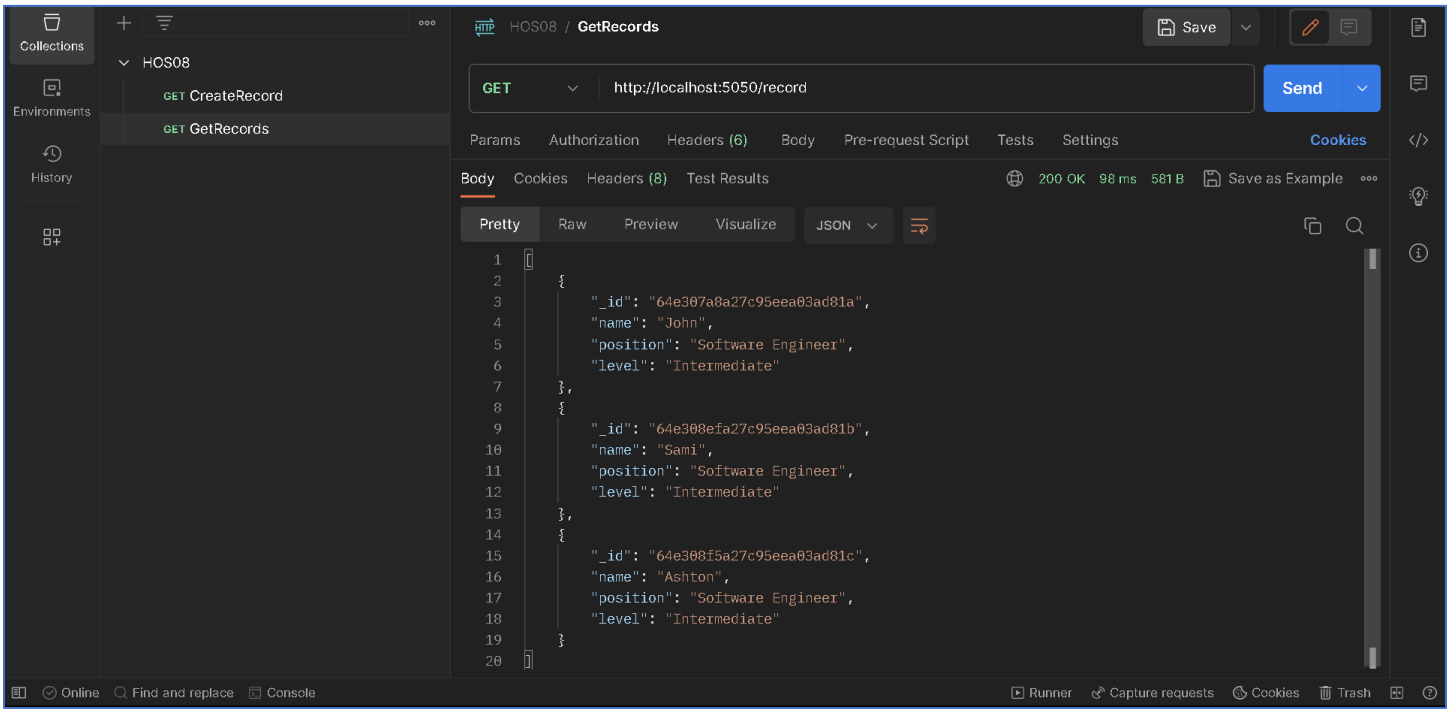
9. Display the newly inserted record:



10. Insert at least two more records in the same manner.

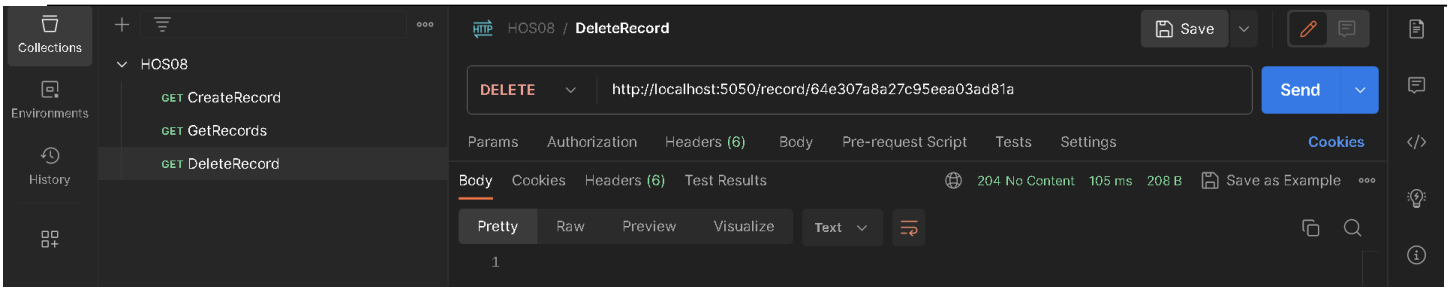
NOTE: *We need to complete step 10 so that there are extra documents in the collection to manipulate as we work through the following steps.*

11. Create another request called **GetRecords**.
12. Select the **GET** method and enter the same URL used in the previous request.
13. Press **Send** and observe the records returned in response:

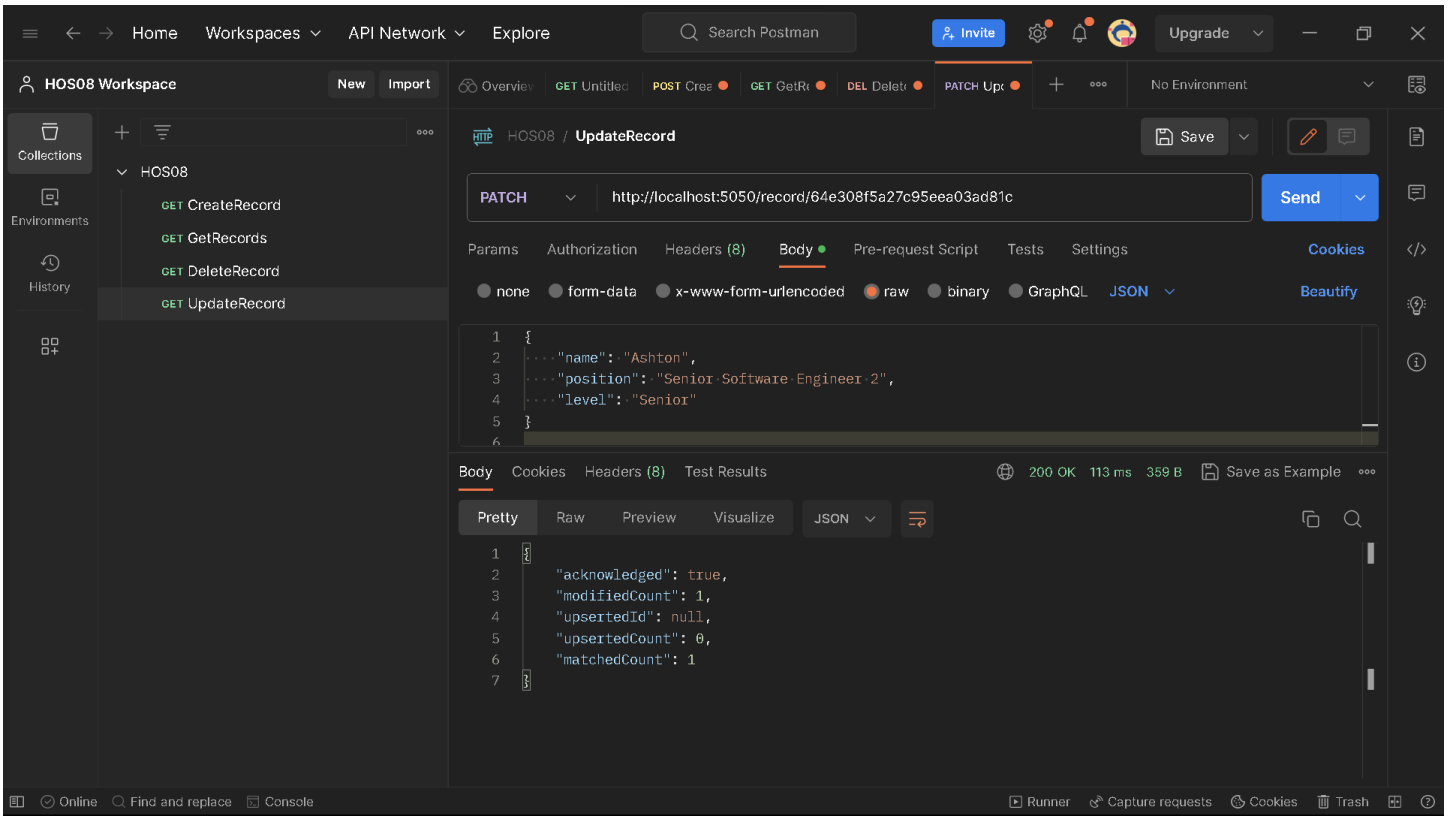


14. Create a third request called **DeleteRecord**.
15. Select the **DELETE** method for the request.
16. Copy an ID from one of the records in the response body of the **GetRecords** request.
17. Append the record ID after **/record** using the same URL from the other requests.

18. Press **Send** and display the returned response HTTP status code:



19. Create a fourth request called **UpdateRecord**.
20. Set the HTTP method to **PATCH**.
21. Copy another (different) ID from the **GetRecords** response body.
22. Enter the same URL with the new ID in place of the previous one we just deleted.
23. Update the data as shown below in the request body and press **Send** to see the response:



PUSH YOUR WORK TO GITHUB TO SUBMIT