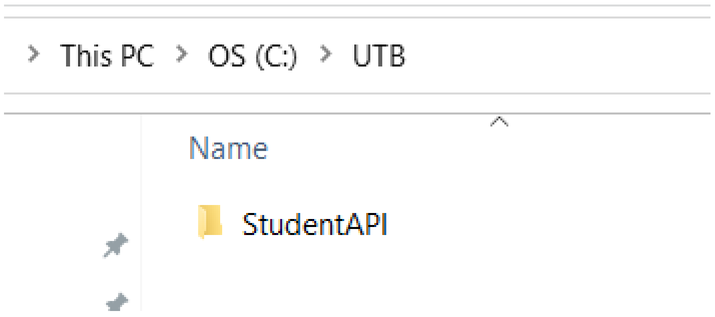
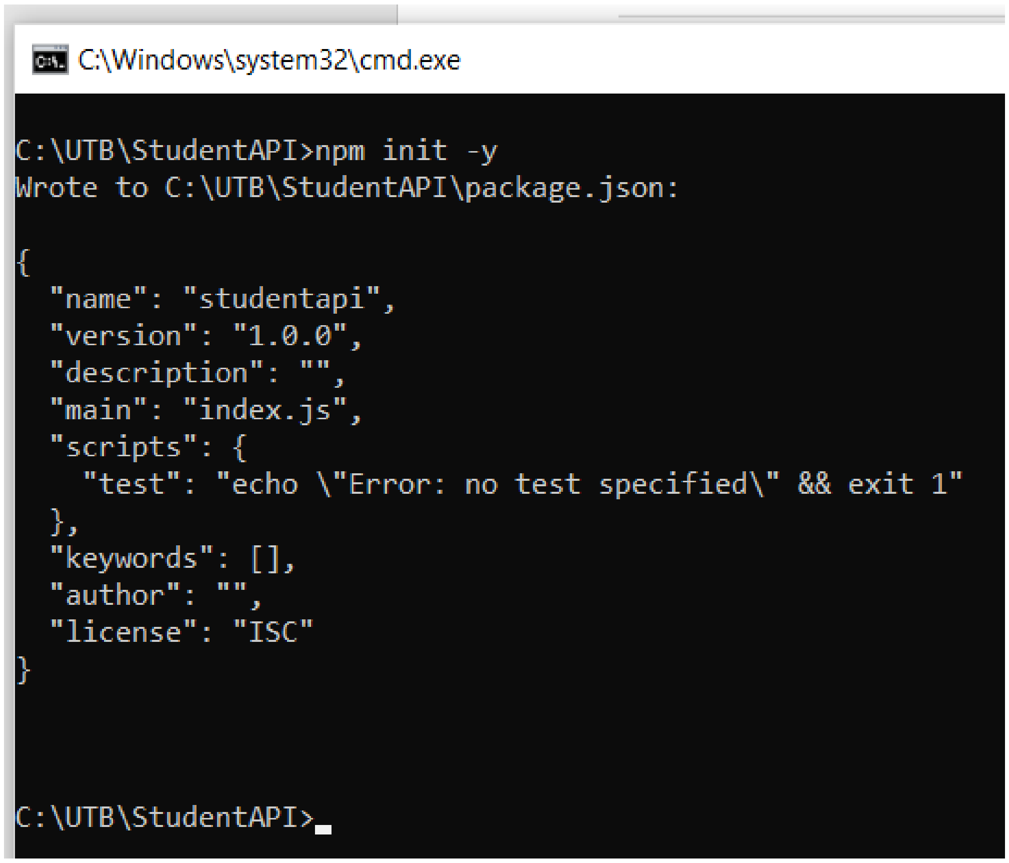
Task 1. Set up the project

Create a folder **StudentAPI**



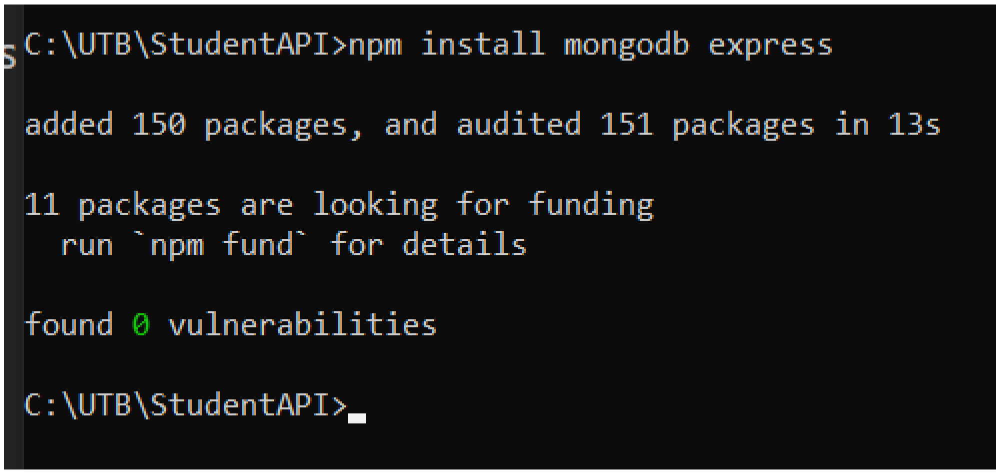
Open a Terminal (Command Prompt, Terminal or from Visual Studio), set the location to the folder you just created and run the following command in order to **start a Node.js project:**

`npm init -y`



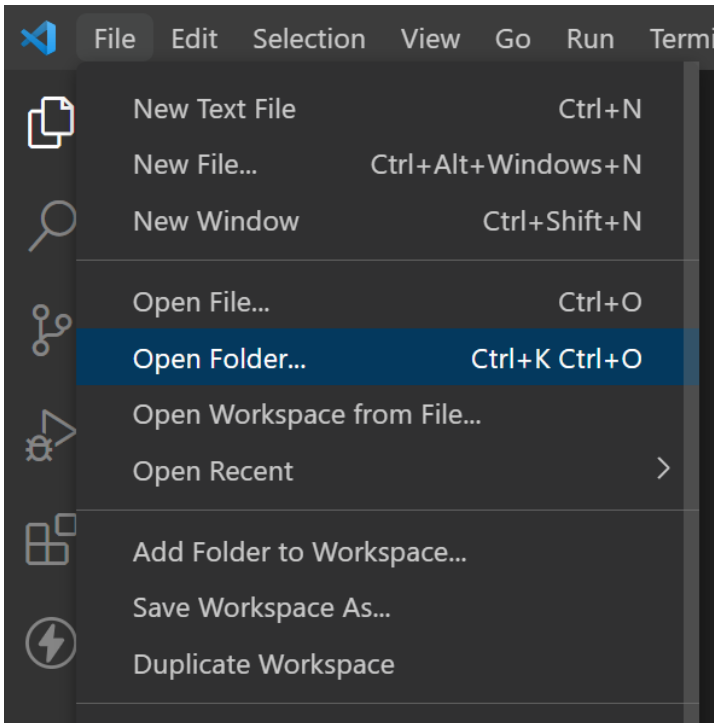
```
C:\UTB\StudentAPI>npm init -y
Wrote to C:\UTB\StudentAPI\package.json:

{
  "name": "studentapi",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}



C:\UTB\StudentAPI>
```

Now install the **`mongodb`** and **`express`** packages with the command **`npm install`**

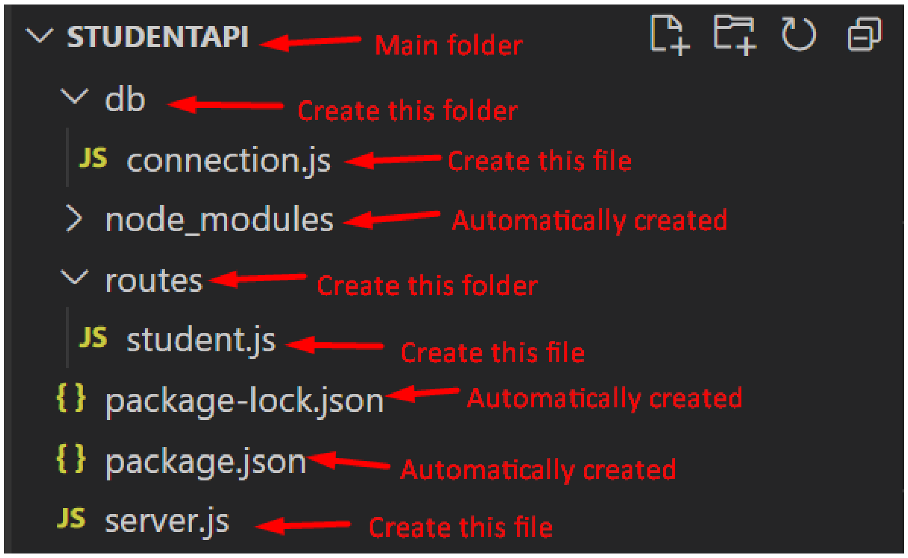


```
C:\UTB\StudentAPI>npm install mongodb express

added 150 packages, and audited 151 packages in 13s

11 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

C:\UTB\StudentAPI>_
```

Use an editor (such as Visual Studio Code) and **access that folder** (File > Open Folder)



Create the following structure inside your project (folder and files)



StudentAPI

- db (folder)
  - connection.js (file)
- routes (folder)
  - student.js (file)
- server.js (file)

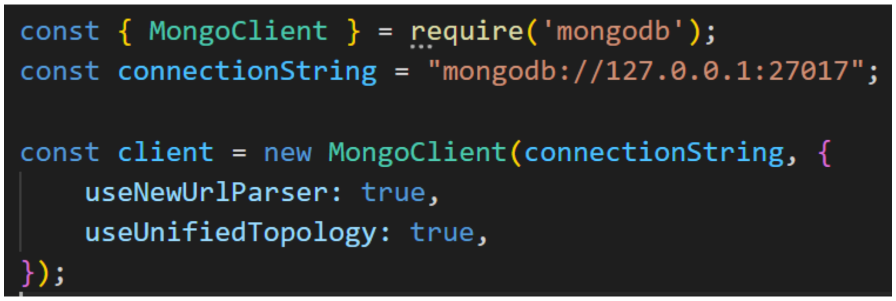
The **StudentAPI** directory hosts the **Express.js** server application and all of its dependencies. It includes the following files:

- **db/connection.js**: Exposes a global connection to the MongoDB database by exporting a MongoDB client that any other module can use.
- **routes/student.js**: Exposes the REST API endpoints and performs their business logic against the database.
- **server.js**: The main entry point for the Express server and configuration initialization.

Task 2. Implement the database connection

Let’s add the code for **db/connection.js:**

First, set up a **MongoClient** object; it is created from the **mongodb** package that was added earlier; a connection string is required and. In this case, the default values for a local MongoDB server (IP and port) are used. Finally, a new instance is created with specific options which are required by the MongoDB driver.

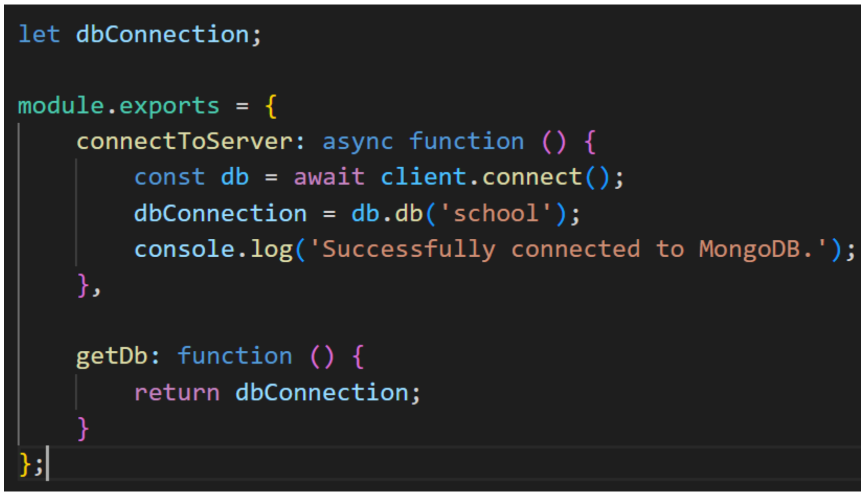


```
const { MongoClient } = require('mongodb');
const connectionString = "mongodb://127.0.0.1:27017";

const client = new MongoClient(connectionString, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
});
```

The main object this module exports out is the **dbConnection** variable, which will hold the **school** database-level object. Via this object, we will be able to access any collection within that database. Two methods are defined:

- **connectToServer**: Establishes a connection to a specific database (**school**) using the client and **connect** method.
- **getDB**: returns the connection object that interacts directly with the database

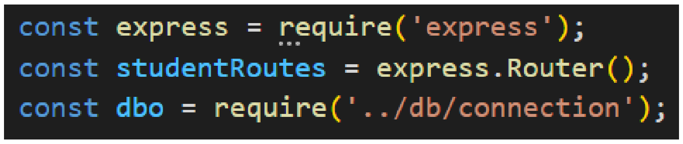
The code goes as follows:



```
let dbConnection;

module.exports = {
    connectToServer: async function () {
        const db = await client.connect();
        dbConnection = db.db('school');
        console.log('Successfully connected to MongoDB.');
    },

    getDb: function () {
        return dbConnection;
    }
};
```

Task 3. Implement the REST endpoints

The goal of this tutorial is to expose REST API routes to perform CRUD (**Create, Read, Update, and Delete**) operations as part of a backend application. The file that will host the routes is **routes/student.js**. It uses the Express Router feature and a reference of **connection.js** file is required:

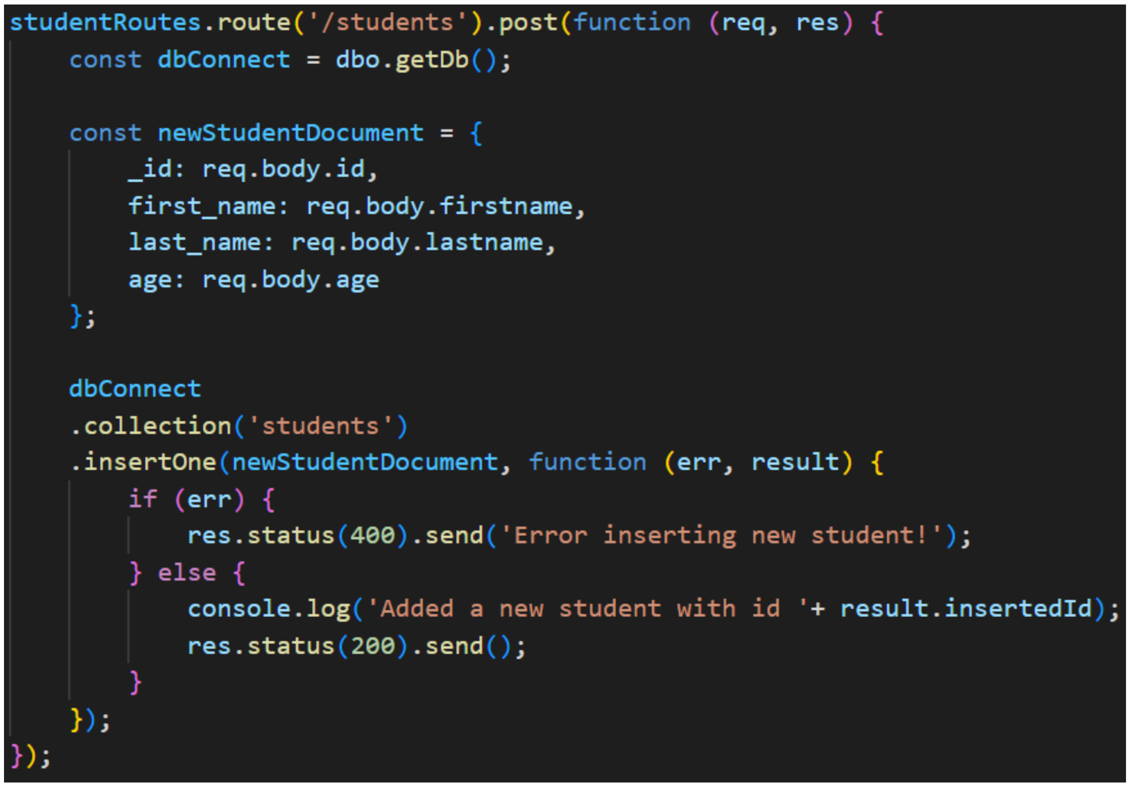


```
const express = require('express');
const studentRoutes = express.Router();
const dbo = require('../db/connection');
```

studentRoutes is an instance of the express router. We use it to define our routes. The router will be added as a middleware and will take control of requests starting with path **/students**.

Create Route

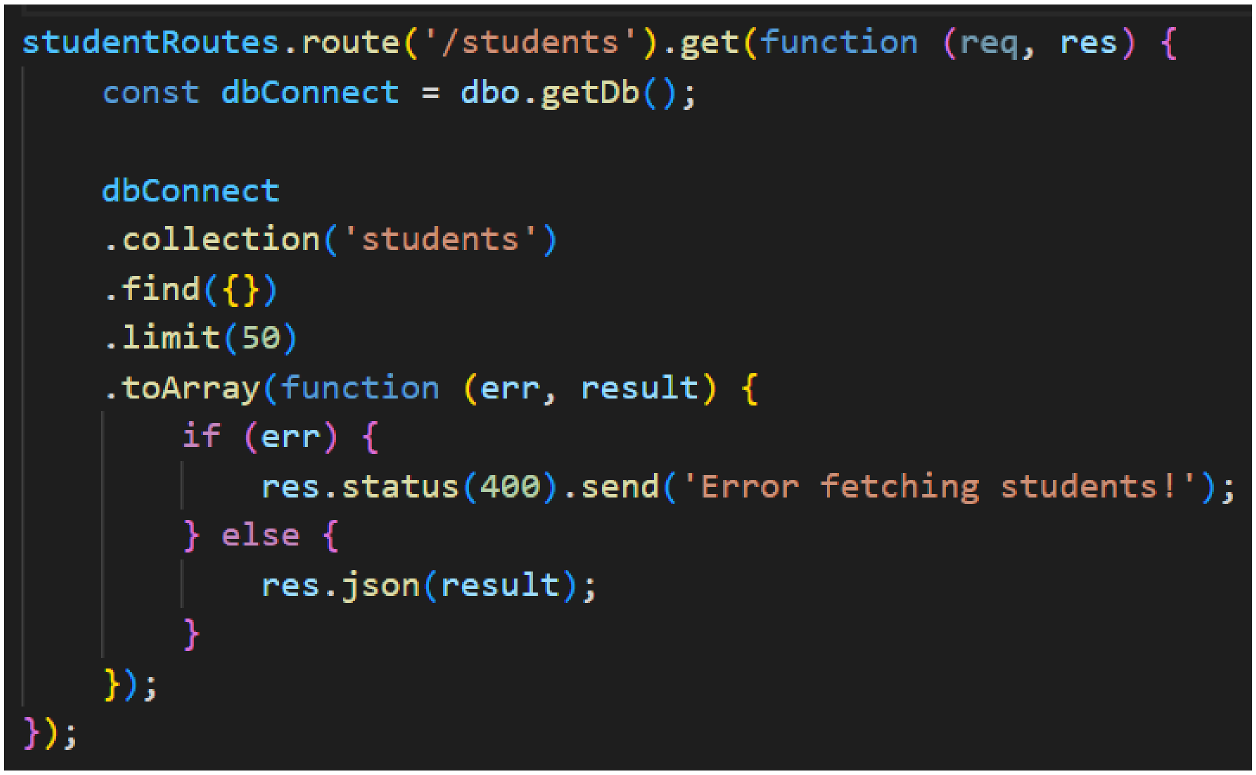
The Create route will add a new student in the **students** collection. The body of this **POST** method will present data to create a **student** document. Finally, adding data is done via **insertOne()** method with the prebuilt **newStudentDocument**. If an error occurs, a 400 Bad Request response is sent back. If the requesr is successful, a 200 OK response is sent to the client. The code is:



```
studentRoutes.route('/students').post(function (req, res) {
    const dbConnect = dbo.getDb();

    const newStudentDocument = {
        _id: req.body.id,
        first_name: req.body.firstname,
        last_name: req.body.lastname,
        age: req.body.age
    };

    dbConnect
    .collection('students')
    .insertOne(newStudentDocument, function (err, result) {
        if (err) {
            res.status(400).send('Error inserting new student!');
        } else {
            console.log('Added a new student with id '+ result.insertedId);
            res.status(200).send();
        }
    });
});
```

Read Route

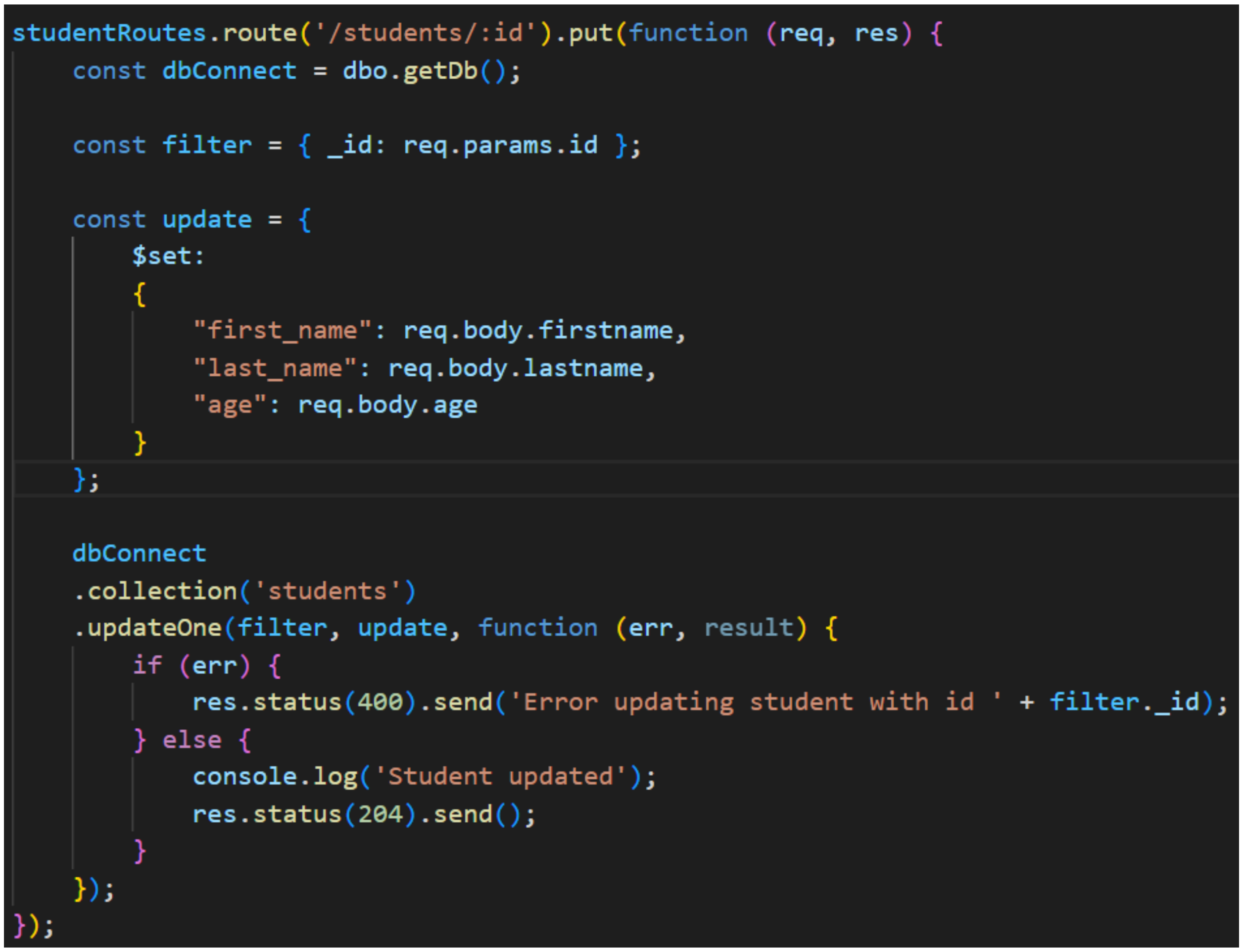
The Read route will be used when the **/students** path on a **GET** method is called. It will use a **find()** method to query our **students** collection for the first 50 available documents. The code sends back the result set as the API response. If an error occurs, a 400 Bad Request response is sent back. The code is:



```
studentRoutes.route('/students').get(function (req, res) {
    const dbConnect = dbo.getDb();

    dbConnect
    .collection('students')
    .find({})
    .limit(50)
    .toArray(function (err, result) {
        if (err) {
            res.status(400).send('Error fetching students!');
        } else {
            res.json(result);
        }
    });
});
```

Update Route

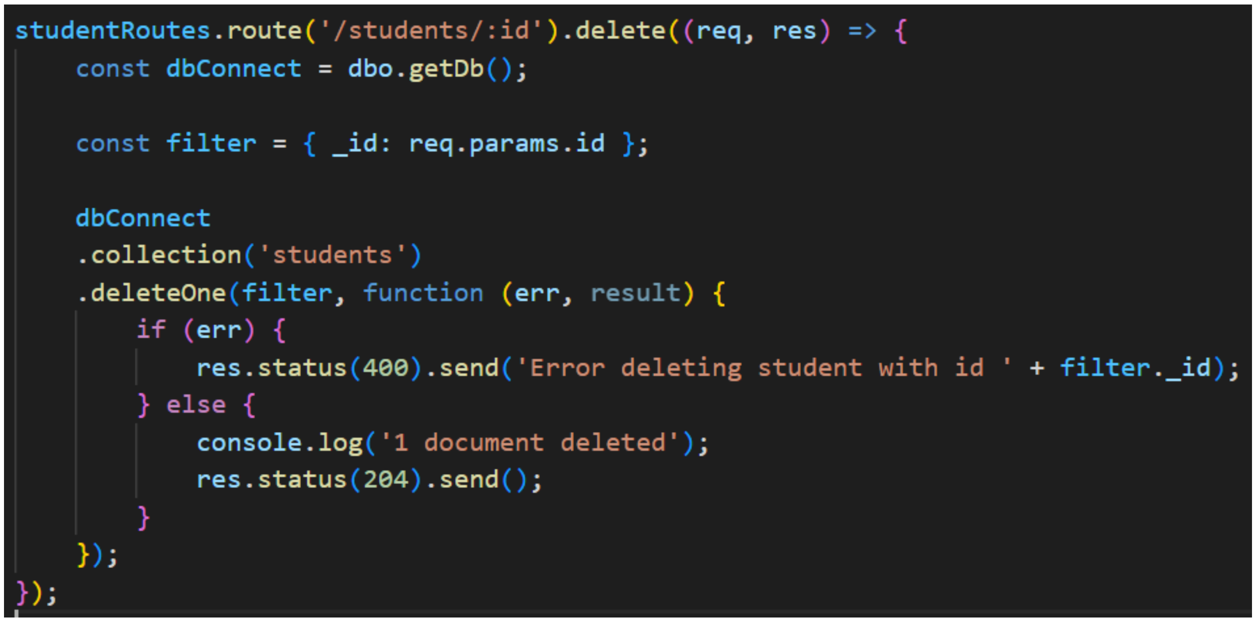
The Update route looks for the **student** document specified by the :**id** URL parameter and updates its information according to the data included in the body request via a **PUT** method. The **updateOne()** method is used to perform the operation in the database. If an error occurs, a 400 Bad Request response is sent back. The code is:



```
studentRoutes.route('/students/:id').put(function (req, res) {
    const dbConnect = dbo.getDb();

    const filter = { _id: req.params.id };

    const update = {
        $set:
        {
            "first_name": req.body.firstname,
            "last_name": req.body.lastname,
            "age": req.body.age
        }
    };

    dbConnect
    .collection('students')
    .updateOne(filter, update, function (err, result) {
        if (err) {
            res.status(400).send('Error updating student with id ' + filter._id);
        } else {
            console.log('Student updated');
            res.status(204).send();
        }
    });
});
```

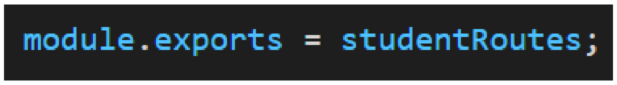
Delete Route

We can remove information from the database via the Delete route, which includes the **:id** parameter that is used to find a specific student that will be deleted via **deleteOne()** method and **DELETE** request.



```
studentRoutes.route('/students/:id').delete((req, res) => {
    const dbConnect = dbo.getDb();

    const filter = { _id: req.params.id };

    dbConnect
    .collection('students')
    .deleteOne(filter, function (err, result) {
        if (err) {
            res.status(400).send('Error deleting student with id ' + filter._id);
        } else {
            console.log('1 document deleted');
            res.status(204).send();
        }
    });
});
```

The routes are exported so they can be used in other files:



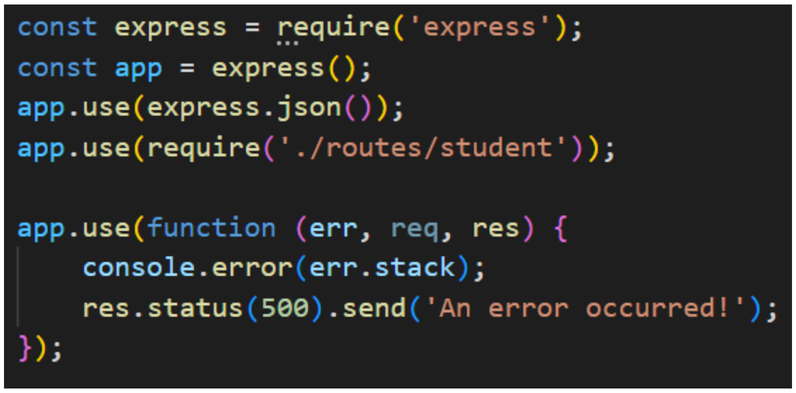
```
module.exports = studentRoutes;
```

Now that we have everything in place, we can prepare and launch the server.

Task 4. Prepare and launch the server

The following code:

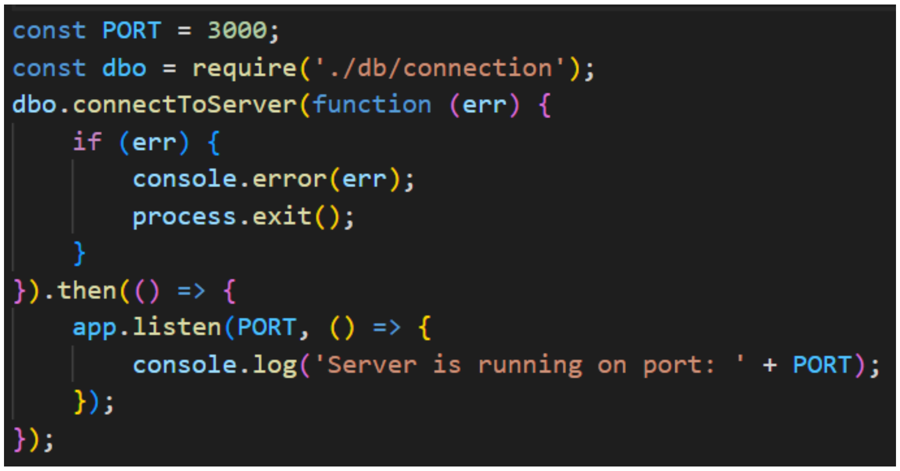
- Initializes the Express app with the required package
- Uses **`express.json()`** middleware because data will be sent and returned in JSON format
- Gets a reference to the student routes
- Defines a global error handling function in case something unexpected happens.



```
const express = require('express');
const app = express();
app.use(express.json());
app.use(require('./routes/student'));

app.use(function (err, req, res) {
    console.error(err.stack);
    res.status(500).send('An error occurred!');
});
```

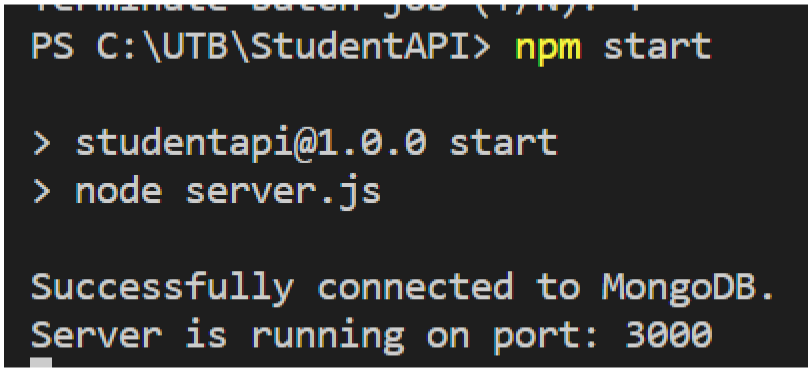
Now it is time to run the application, so:

- get the MongoDB driver connection (**dbo** object),
- perform a database connection (by invoking the **connectToServer** method),
- if the connection is successful, use the **`listen()`** method on **`app`** to start the Express server. You can set the app to run on any available port (in this case, the port 3000 is being used)



```
const PORT = 3000;
const dbo = require('./db/connection');
dbo.connectToServer(function (err) {
    if (err) {
        console.error(err);
        process.exit();
    }
}).then(() => {
    app.listen(PORT, () => {
        console.log('Server is running on port: ' + PORT);
    });
});
```

Launch the server by using the **`npm start`** command. (Open a Terminal and run the command in the same path were your server.js file is located):



```
PS C:\UTB\StudentAPI> npm start

> studentapi@1.0.0 start
> node server.js

Successfully connected to MongoDB.
Server is running on port: 3000
```

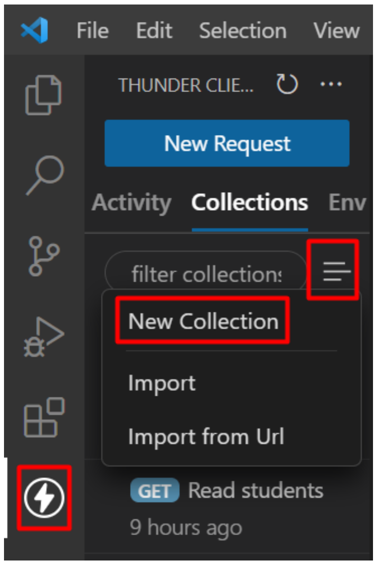
You can observe that two successful messages are received:

- One for the database connection
- Another one for a running server that listens for requests on port 3000

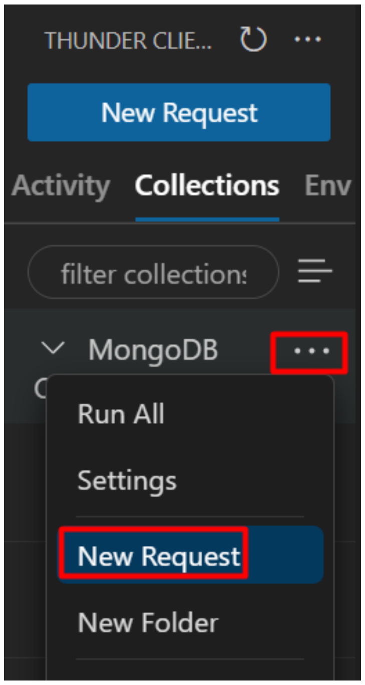
Task 5. Test the routes

We can test our endpoint. Use any tool you have installed (cURL, Postman, Thunder Client extension, etc).

For Thunder Client, you can create a collection of requests by clicking on the following icons:

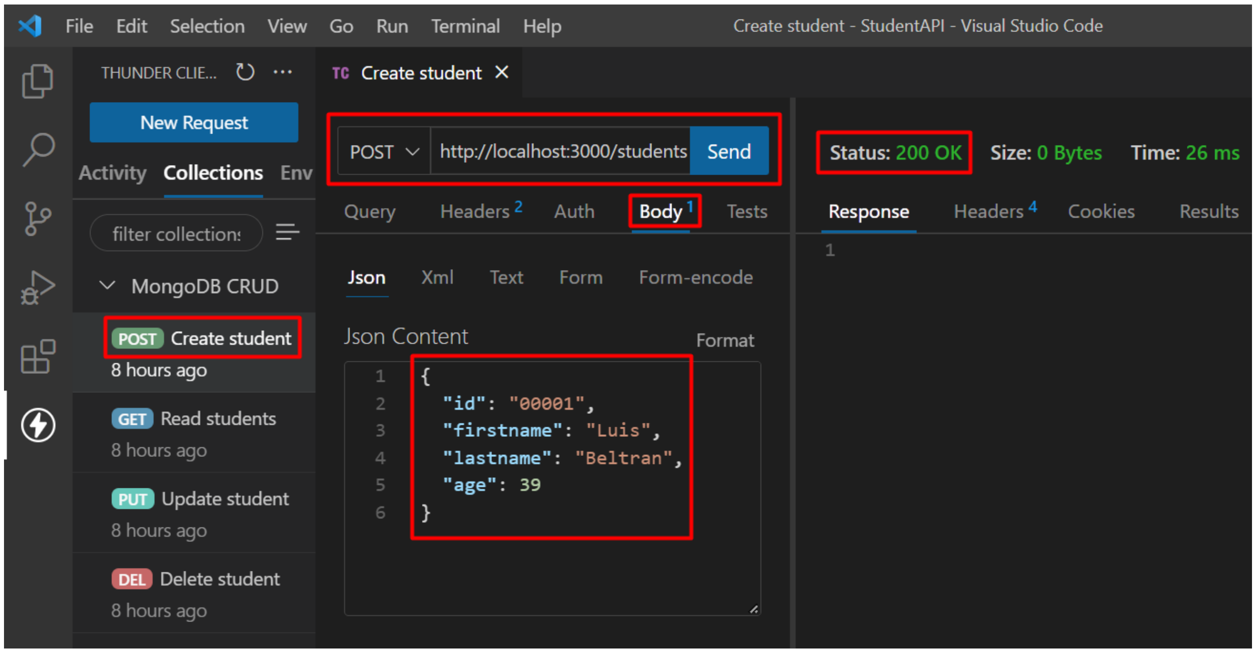


Then, add a new Request:



Here is an example for each route:

Add a student (request on the left side, response on the right side):

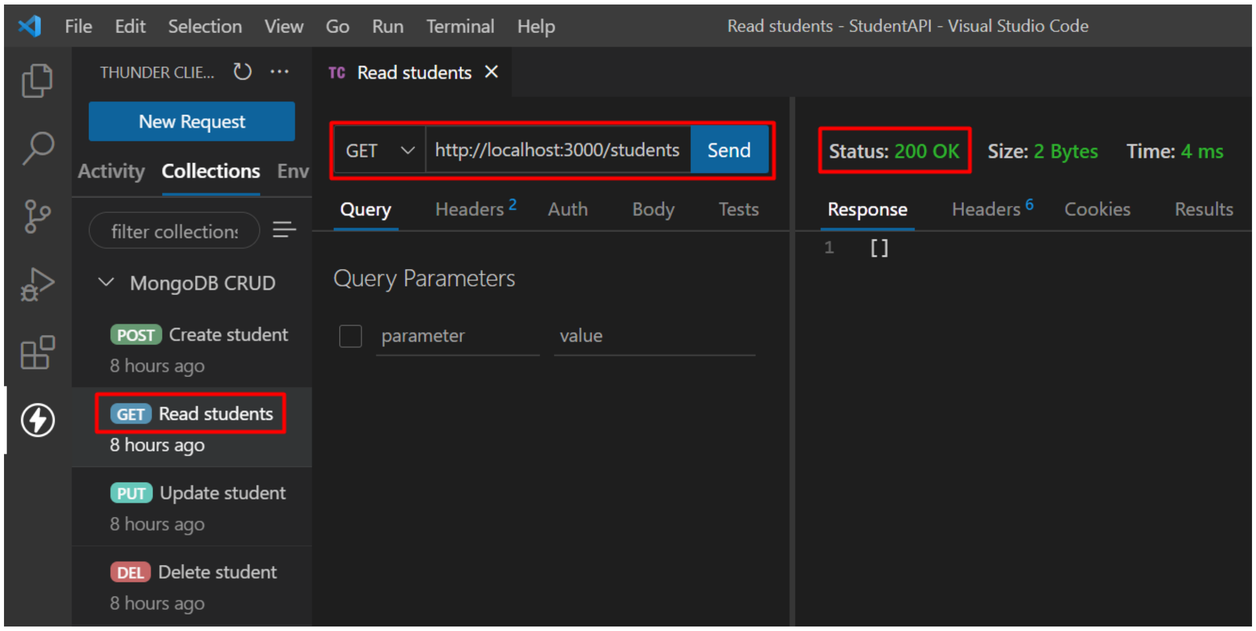


http://localhost:3000/students

```
{
  "id": "00001",
  "firstname": "Luis",
  "lastname": "Beltran",
  "age": 39
}
```

Status: 200 OK Size: 0 Bytes Time: 26 ms

Get students

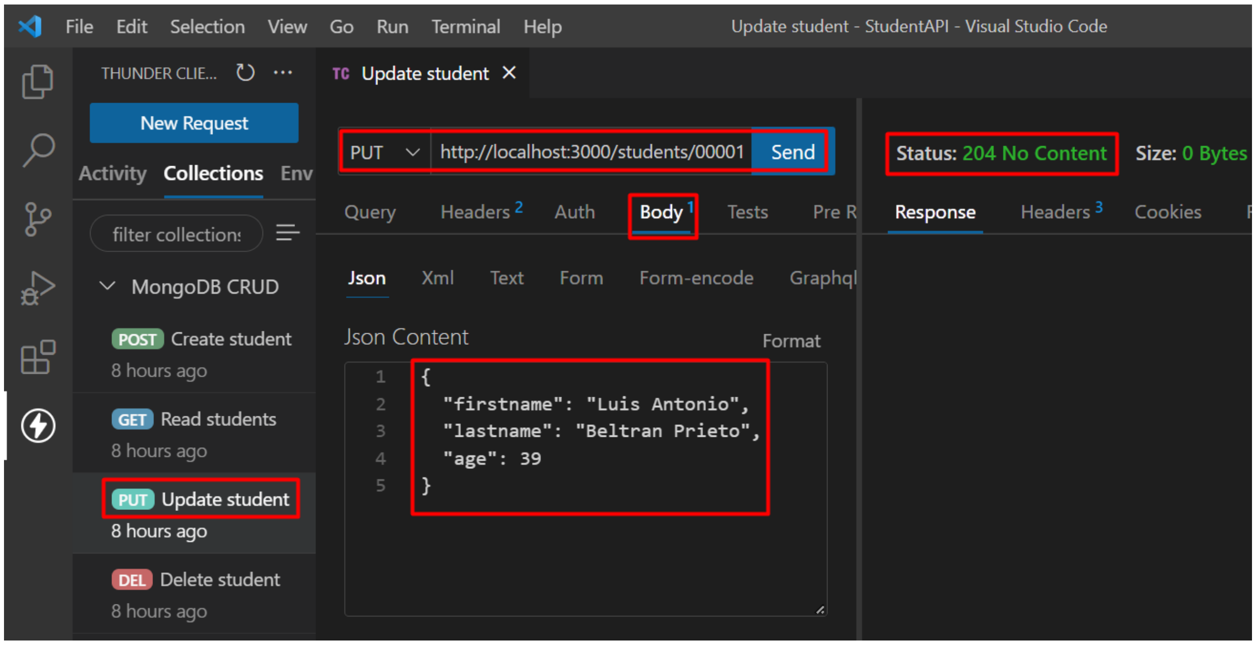


http://localhost:3000/students

Status: 200 OK Size: 2 Bytes Time: 4 ms

```
[]
```

Edit a student

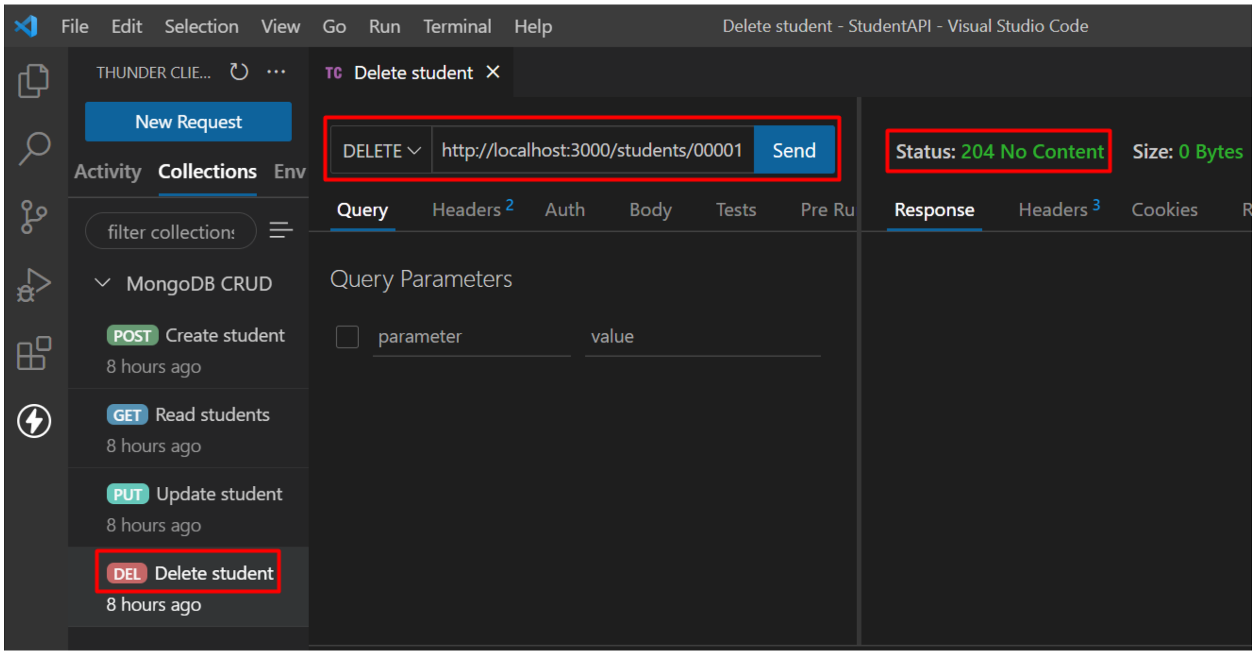


PUT http://localhost:3000/students/00001 Send

Status: 204 No Content Size: 0 Bytes

```
{
  "firstname": "Luis Antonio",
  "lastname": "Beltran Prieto",
  "age": 39
}
```

Remove a student



DELETE http://localhost:3000/students/00001 Send

Status: 204 No Content Size: 0 Bytes