

Task Manager API

With this project we will learn to setup and connect to the cloud database. So effectively we will learn how to persist our data to the cloud and will perform CRUD operations.

CRUD – Create, Read, Update and Delete

Instead of a regular to-do app which stores data in local storage, in this project the data is stored in the cloud database and persist data to the cloud.

Assume a frontend were,

First API Call – on load – get all the request and display on the screen. (GET Request)

Second API Call – Adding a new task into the list and getting all the tasks so that UI can display all the tasks. (POST Request)

Third API Call – Editing the present task, getting single task with id, and displaying all the details. (GET Request with ID).

Forth API Call – Saving the details of the edited task in the cloud. (PUT request with ID)

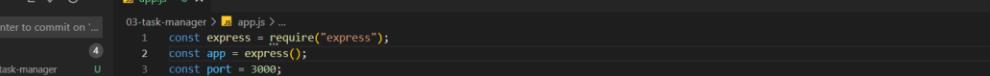
Fifth API Call – Deleting a task from the list. (DELETE with Request ID)

Version naming is important because whenever a new development comes and when accommodate that development in our existing API and route, we tend to create a new route with different version name.

The basic naming structure for an API would be

http://Domain-Name/api/version-name/items/:id

Routes in the Task Manager API Project



```
File Edit Selection View Go Run Terminal Help app.js - NodeJS-ExpressJS - Visual Studio Code SOURCE CONTROL E ✓ ⌂ ... app.js U x 30-task-manager > app.js >... 1 const express = require("express"); 2 const app = express(); 3 const port = 3000; 4 5 app.listen(port, () => { 6 | console.log(`Server is listening on Port ${port}....`); 7 }); 8 9 // Routes 10 11 // app.get("/api/v1/tasks") - get all the tasks 12 // app.post("/api/v1/tasks") - create a new task 13 // app.get("/api/v1/tasks/:id") - get single task 14 // app.patch("/api/v1/tasks/:id") - update task 15 // app.delete("/api/v1/:id") - delete task 16
```

To maintain the app.js lean and clean, we need to use the MVC pattern and hence we follow that pattern throughout the project.

As of now, we are creating two folders namely controllers and routes.

Let us start with creating a route

tasks.js file in routes folder

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure under "NODEJS-EXPRESS". The "routes" folder is expanded, showing "tasks.js". Other files like "app.js", "index.js", and "package.json" are also visible.
- Code Editor:** The "tasks.js" file is open, containing the following code:

```
const express = require("express");
const router = express.Router();
const getAllTasks = require("../controllers/tasks");

router.get("/", getAllTasks);

module.exports = router;
```
- Terminal:** Shows the message "Server is listening on Port 3000..."
- Status Bar:** Displays "Ln 15, Col 1" and other settings like "Spaces: 4", "UTF-8", "CRLF", "JavaScript", and "Prettier".

tasks.js in controllers folder

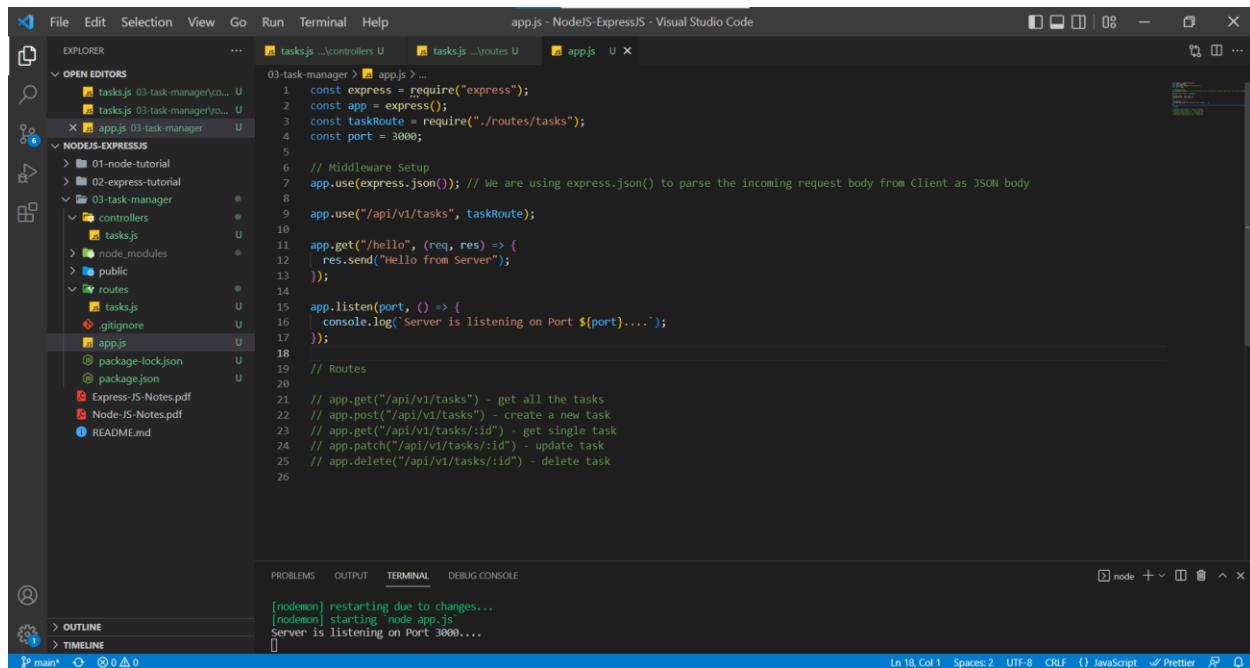
The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure under "NODEJS-EXPRESS". The "controllers" folder is expanded, showing "tasks.js". Other files like "app.js", "index.js", and "package.json" are also visible.
- Code Editor:** The "tasks.js" file is open, containing the following code:

```
const getAllTasks = (req, res) => {
  res.send("All Items from the controller");
};

module.exports = { getAllTasks };
```
- Terminal:** Shows the message "Server is listening on Port 3000..."
- Status Bar:** Displays "Ln 8, Col 1" and other settings like "Spaces: 4", "UTF-8", "CRLF", "JavaScript", and "Prettier".

app.js



```
File Edit Selection View Go Run Terminal Help app.js - NodeJS-ExpressJS - Visual Studio Code

OPEN EDITORS tasks.js ...controllers U tasks.js ...routes U app.js U
NODEJS-EXPRESSJS 01-node-tutorial 02-express-tutorial 03-task-manager
  controllers tasks.js
  node_modules
  public
  routes tasks.js .gitignore
  app.js package-lock.json package.json
  Express-JS-Notes.pdf Node-JS-Notes.pdf README.md

const express = require("express");
const app = express();
const taskRoute = require("./routes/tasks");
const port = 3000;

// Middleware Setup
app.use(express.json()); // We are using express.json() to parse the incoming request body from client as JSON body

app.use("/api/v1/tasks", taskRoute);

app.get("/hello", (req, res) => {
  res.send("Hello from Server");
});

app.listen(port, () => {
  console.log(`Server is listening on Port ${port}....`);
});

// Routes

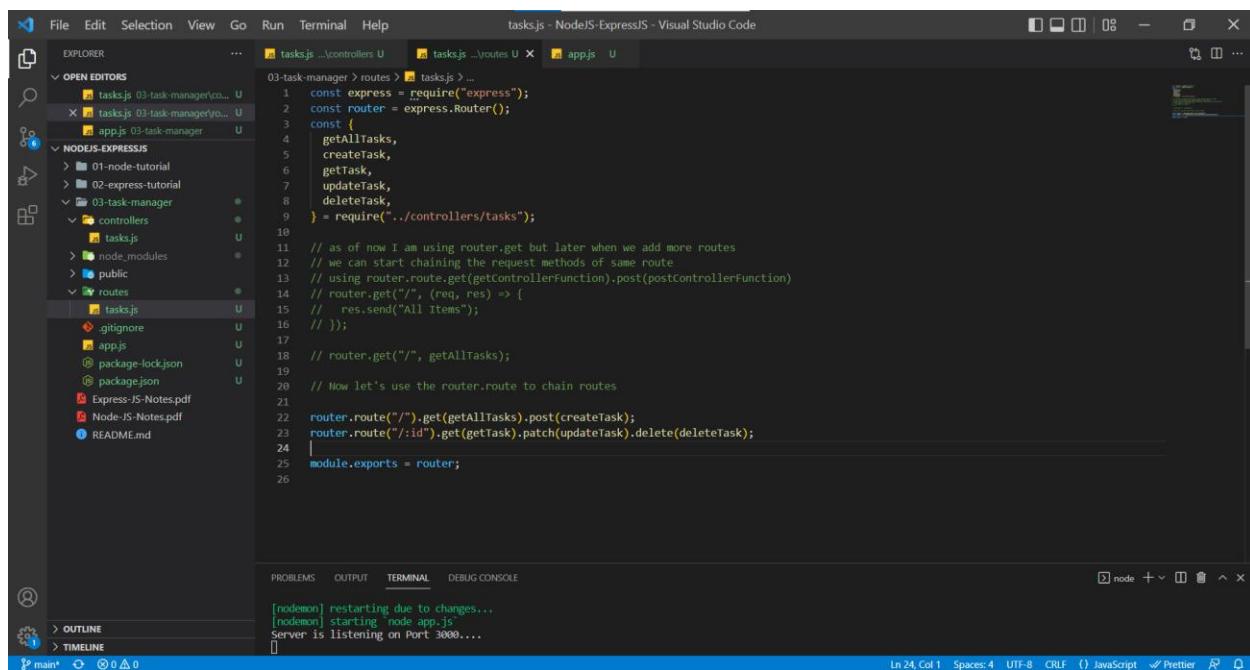
// app.get("/api/v1/tasks") - get all the tasks
// app.post("/api/v1/tasks") - create a new task
// app.get("/api/v1/tasks/:id") - get single task
// app.patch("/api/v1/tasks/:id") - update task
// app.delete("/api/v1/tasks/:id") - delete task

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
[nodemon] restarting due to changes...
[nodemon] starting 'node app.js'
Server is listening on Port 3000...
Ln 18, Col 1 Spaces: 2 UTF-8 CRLF ( JavaScript ⚡ Prettier ⚡ 
```

Now let's create controller functions and set up routes for other API routes.

There won't be any change in the app.js

tasks.js in routes folder



```
File Edit Selection View Go Run Terminal Help tasks.js - NodeJS-ExpressJS - Visual Studio Code

OPEN EDITORS tasks.js ...controllers U tasks.js ...routes U app.js U
NODEJS-EXPRESSJS 01-node-tutorial 02-express-tutorial 03-task-manager
  controllers tasks.js
  node_modules
  public
  routes tasks.js .gitignore
  app.js package-lock.json package.json
  Express-JS-Notes.pdf Node-JS-Notes.pdf README.md

const express = require("express");
const router = express.Router();
const {
  getAllTasks,
  createTask,
  getTask,
  updateTask,
  deleteTask
} = require("../controllers/tasks");

// as of now I am using router.get but later when we add more routes
// we can start chaining the request methods of same route
// using router.route.get(getControllerFunction).post(postControllerFunction)
// router.get("/", (req, res) => {
//   res.send("All Items");
// });

// router.get("/", getAllTasks);

// Now let's use the router.route to chain routes
router.route("/").get(getAllTasks).post(createTask);
router.route("/:id").get(getTask).patch(updateTask).delete(deleteTask);

module.exports = router;

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
[nodemon] restarting due to changes...
[nodemon] starting 'node app.js'
Server is listening on Port 3000...
Ln 24, Col 1 Spaces: 4 UTF-8 CRLF ( JavaScript ⚡ Prettier ⚡ 
```

tasks.js in controller folder

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure under "NODEJS-EXPRESS". The "controllers" folder contains "tasks.js". Other files like "app.js", "routes", and "node_modules" are also visible.
- Code Editor:** The active file is "tasks.js" which contains the following code:

```
const getAllTasks = (req, res) => {
  res.send("All Items from the Controller");
};

const createTask = (req, res) => {
  res.send("Create Task");
};

const getTask = (req, res) => {
  res.send("Get Single Task");
};

const updateTask = (req, res) => {
  res.send("Update Task");
};

const deleteTask = (req, res) => {
  res.send("Delete Task");
};

module.exports = [
  getAllTasks,
  createTask,
  getTask,
  updateTask,
  deleteTask,
];
```

- Terminal:** Shows the output of nodemon starting the application on port 3000.
- Status Bar:** Shows the current file is "tasks.js - NodeJS-ExpressJS - Visual Studio Code", line 25, column 14, and other settings like "Spaces: 4", "UTF-8", "CR LF".

Now let us test our routes in Postman

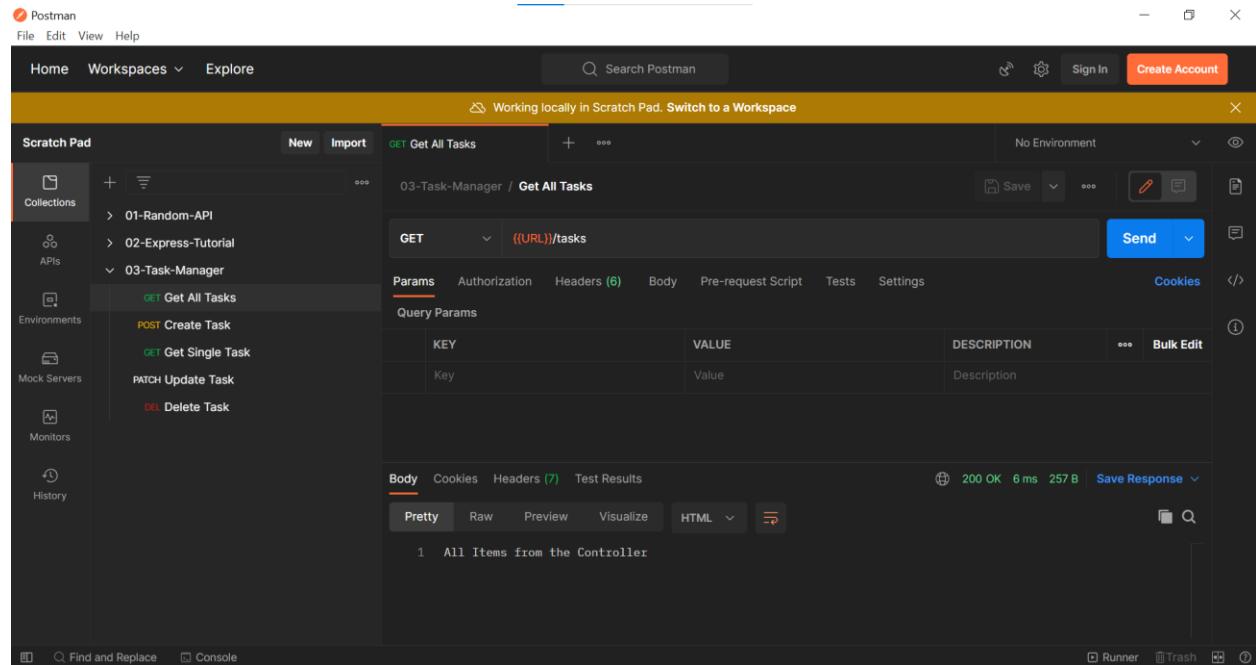
Let us create a collection in Postman because we will be setting up multiple routes and they will reference the same application which is going to easier as we create more and more applications.

Created a collection of Task Manager in Postman as well set up a Global Variable which can be accessed throughout the postman. (Instead of always writing localhost:3000 for every request, we can setup that link as a global variable)

The screenshot shows the Postman interface with the following details:

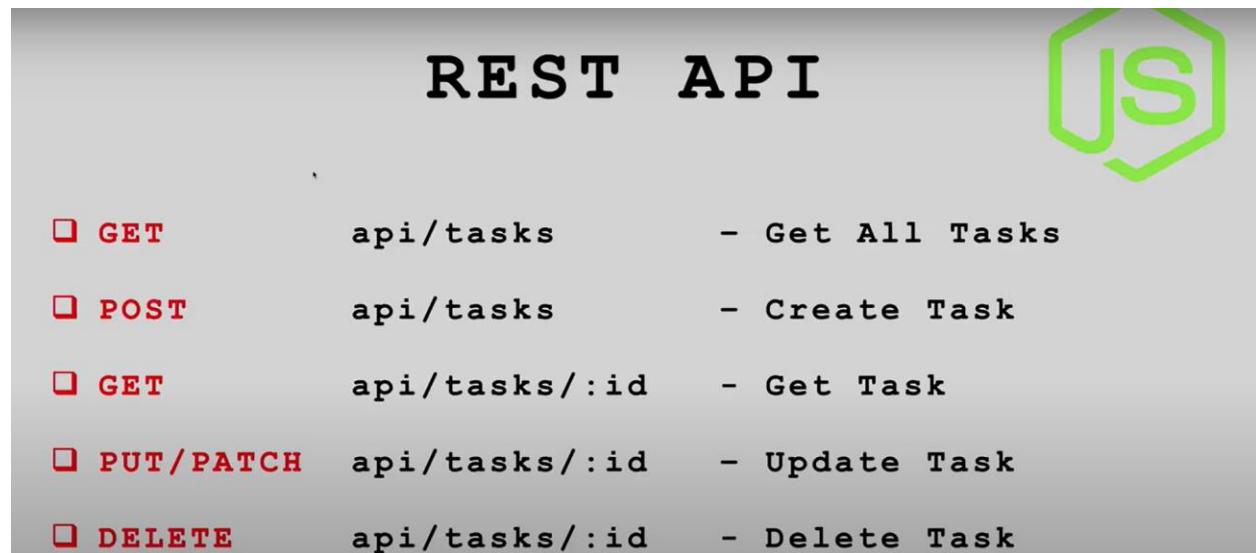
- Left Sidebar:** Shows collections like "01-Random-API", "02-Express-Tutorial", and "03-Task-Manager".
- Scratch Pad:** Shows the "03-Task-Manager" collection with four API endpoints:
 - GET** Get All Tasks
 - POST** Create Task
 - GET** Get Single Task
 - PATCH** Update Task
 - DELETE** Delete Task
- Environment:** Shows "No active Environment". It explains what an environment is: "An environment is a set of variables that allow you to switch the context of your requests".
- Globals:** Shows a table with one entry: URL (http://localhost:3000/api/v1).
- Bottom Status Bar:** Shows "Find and Replace", "Console", "Runner", "Trash", and other icons.

Get All Tasks Request in Postman



The screenshot shows the Postman application interface. In the top navigation bar, there are links for File, Edit, View, Help, Home, Workspaces, Explore, and a search bar. On the right side of the header, there are icons for Sign In and Create Account. Below the header, a yellow banner says "Working locally in Scratch Pad. Switch to a Workspace". The main area is titled "Scratch Pad" and shows a collection named "03-Task-Manager" with a sub-request titled "Get All Tasks". The request method is set to "GET" and the URL is "((URL))/tasks". The "Params" tab is selected, showing a single query parameter "Key" with a value "Value". Other tabs include Authorization, Headers (6), Body, Pre-request Script, Tests, and Settings. To the right of the request details, there are buttons for Save, Send, and Cookies. Below the request details, the response status is shown as 200 OK with a response time of 6 ms and a size of 257 B. The response body is displayed in a table with one row: "1 All Items from the Controller". At the bottom of the interface, there are buttons for Pretty, Raw, Preview, Visualize, and HTML, along with a "Save Response" button.

API Routes that have been setup



The screenshot shows a REST API documentation page. At the top, it says "REST API" and has a green hexagonal logo with the letters "JS" inside. Below the title, there is a table of API routes:

Method	Path	Description
GET	api/tasks	- Get All Tasks
POST	api/tasks	- Create Task
GET	api/tasks/:id	- Get Task
PUT/PATCH	api/tasks/:id	- Update Task
DELETE	api/tasks/:id	- Delete Task

We are using the above structure because we are building the REST (Representational State Transfer) API. We want to create an HTTP interface where other apps mostly frontend can interact with our data.

REST is arguably the most popular API design pattern and essentially it is a pattern that combines HTTP verbs, route paths, and our resources aka data. So effectively REST determines how our API looks like.

Since JSON is a common format for receiving and sending data in REST API, we will use that approach as well. Right now we are using the send() method to send the request but eventually we will use json() method to send the response body.

Mongo DB

- No SQL, Non-Relational Data base
- Stores JSON
- Easy to get Started
- Free Cloud Hosting – Atlas

Unlike traditional database where we store data in rows and columns, in No SQL we store data as JSON, and it basically doesn't care how data relates to each other. Instead of tables we have collections which represent group of items and instead of rows we have documents which represent single item.

A document is set of key value pairs and as far as data types we can use String, Arrays, Numbers, Array objects and more.

In this project we are using MongoDB Atlas, which is an official option, basically it is created by the people who created MongoDB. We are using the free tier for which we need to create an account.

Let's setup and configure MongoDB atlas so we can host and manage our data in the cloud.

Heroku and Digital Ocean are the most popular platforms to host Node JS applications. When Hosting with Heroku, we need to set our IP address as anyone access (**Allow from anywhere option**) so that our application won't break, or we don't get any error. In digital ocean, we need to change from our Local IP address to Production IP address.

We have created a Cluster in MongoDB and added few accesses like user and IP address restriction. We copied connection string from MongoDB and added it **db** folder of our application.

Let us create a database in MongoDB with dummy data.

The screenshot shows the MongoDB Atlas interface. The top navigation bar includes tabs for Node.js, Create, Get Started, Create an..., Verify Your..., Database, heroku, digitalocean, and a plus sign. Below the navigation is a horizontal bar with various industry icons: UMKC, US Banking, Investments, IND Banking, Entertainment, Insurance, Health, Food, Travel, Education, Taxes, Social Media, Resources, and TCS. The main header displays "SAI KRISHNA REDDY'S ORG - 2022-07-08 > PROJECT 0". The left sidebar has sections for Deployment (selected), Database (selected), Preview (highlighted), Data Services, Security, and Advanced. The main content area is titled "Database Deployments" and shows a deployment for "NodeExpressProjects". It displays metrics: R 0, W 0 (Last 21 minutes), Connections 2.0 (Last 21 minutes), In 20.8 B/s, Out 120.0 B/s (Last 21 minutes), Data Size 0.0 B (Last 21 minutes), and a total size of 512.0 MB. There are buttons for "Connect", "View Monitoring", "Browse Collections", and "...". A green "Create" button is located at the top right. A "FREE" badge is visible. A call-to-action banner on the right says "Enhance Your Experience" and "For production throughput and richer metrics, upgrade to a dedicated cluster now!" with a "Upgrade" button and a speech bubble icon.

Click on Browse Collections and below screen will appear

The screenshot shows the MongoDB Atlas interface. On the left, there's a sidebar with sections like Deployment, Database (selected), Data Services, Security, and others. The main area has tabs for Atlas, App Services, and Charts. A prominent callout box says "Add My First Dataset" with instructions to create a database and collection. Below it is another box for "Data Modeling Examples". At the top, there are links for Find, Indexes, Aggregation, and Search.

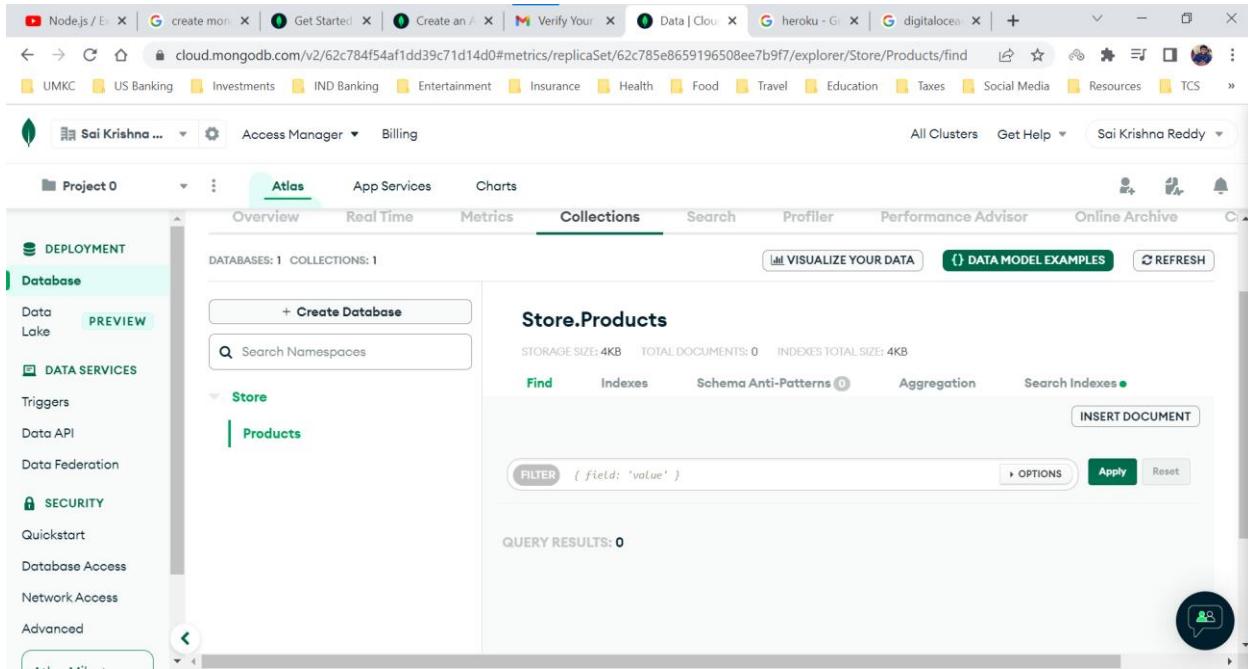
Click on Add My Own Data and below screen will appear.

We are creating a database named Store which contains a collection called Products (In SQL equivalence, Store is the database and Products is the table) and click on Create.

This screenshot shows the "Create Database" dialog box. It asks for the "Database name" (set to "Store") and "Collection name" (set to "Products"). There are checkboxes for "Capped Collection" and "Time Series Collection", neither of which is checked. At the bottom are "Cancel" and "Create" buttons. The background shows the same "Add data to your Collections" interface as the previous screenshot.

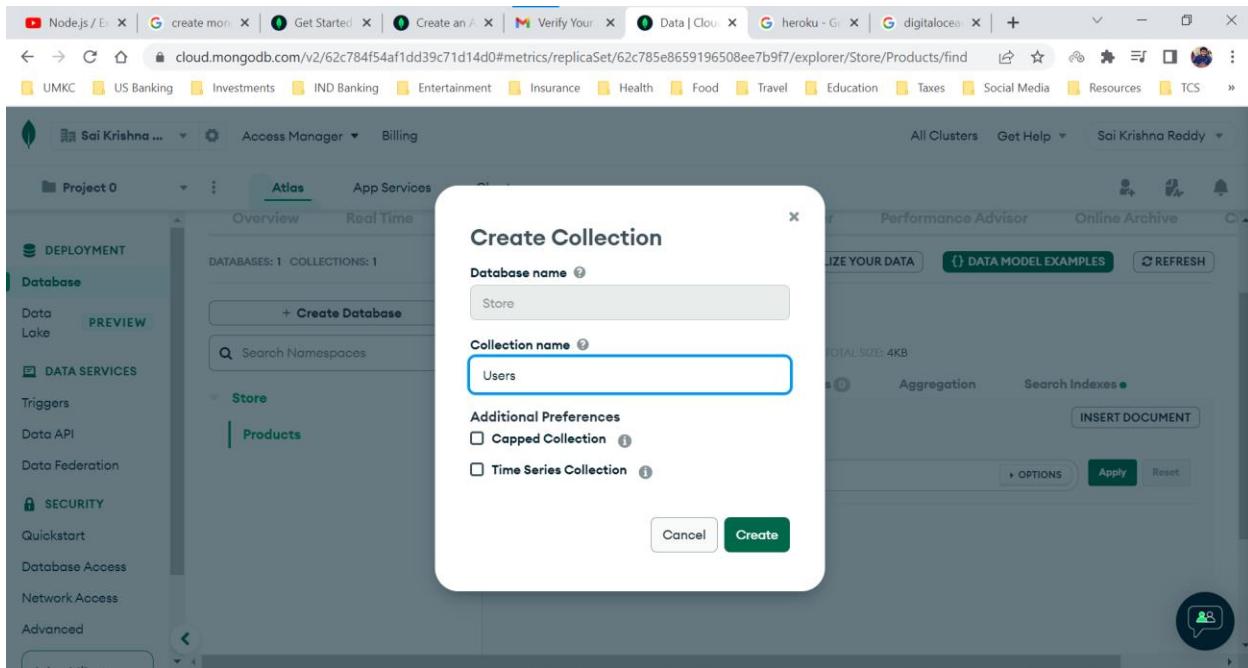
We can have as many collections (tables) as we want. In general SQL database we will be having many tables under one database.

Click on the + (Plus) icon beside the database Store to create another collection.



The screenshot shows the MongoDB Atlas interface. On the left sidebar, under 'Database' > 'PREVIEW', there's a list of services: Data Lake, Triggers, Data API, Data Federation, SECURITY, Quickstart, Database Access, Network Access, and Advanced. The main area shows 'Project 0' with 'Atlas' selected. Under 'Collections', it says 'DATABASES: 1 COLLECTIONS: 1'. Below this, there's a search bar for 'Search Namespaces' and a 'Store' section containing 'Products'. A modal window titled 'Create Collection' is open in the foreground, prompting for 'Database name' (set to 'Store') and 'Collection name' (set to 'Users'). Other options like 'Capped Collection' and 'Time Series Collection' are listed below. At the bottom of the modal are 'Cancel' and 'Create' buttons.

We are creating a collection (table in SQL equivalent) in database named Store.



The screenshot shows the same MongoDB Atlas interface as the previous one, but now the 'Create Collection' dialog is the central focus. It has fields for 'Database name' (set to 'Store') and 'Collection name' (set to 'Users'). Under 'Additional Preferences', there are checkboxes for 'Capped Collection' and 'Time Series Collection', neither of which is checked. At the bottom of the dialog are 'Cancel' and 'Create' buttons. The background shows the 'Store' database with the 'Products' collection selected.

After Creation of collections (Products and Users) in table Store.

The screenshot shows the MongoDB Atlas interface. On the left sidebar, under 'Database' (PREVIEW), there is a 'Collections' section with 'Store' expanded, showing 'Products' and 'Users'. The main panel displays the 'Store.Products' collection with the following details:

- STORAGE SIZE: 4KB
- TOTAL DOCUMENTS: 0
- INDEXES TOTAL SIZE: 4KB

Below these details are tabs for 'Find', 'Indexes', 'Schema Anti-Patterns', 'Aggregation', and 'Search Indexes'. At the bottom of the panel are 'FILTER' and 'OPTIONS' buttons, and a large 'INSERT DOCUMENT' button.

After creating collection, we need to add documents to the collection. In MongoDB a document effectively refers to a single item. Document is a set of key value pairs.

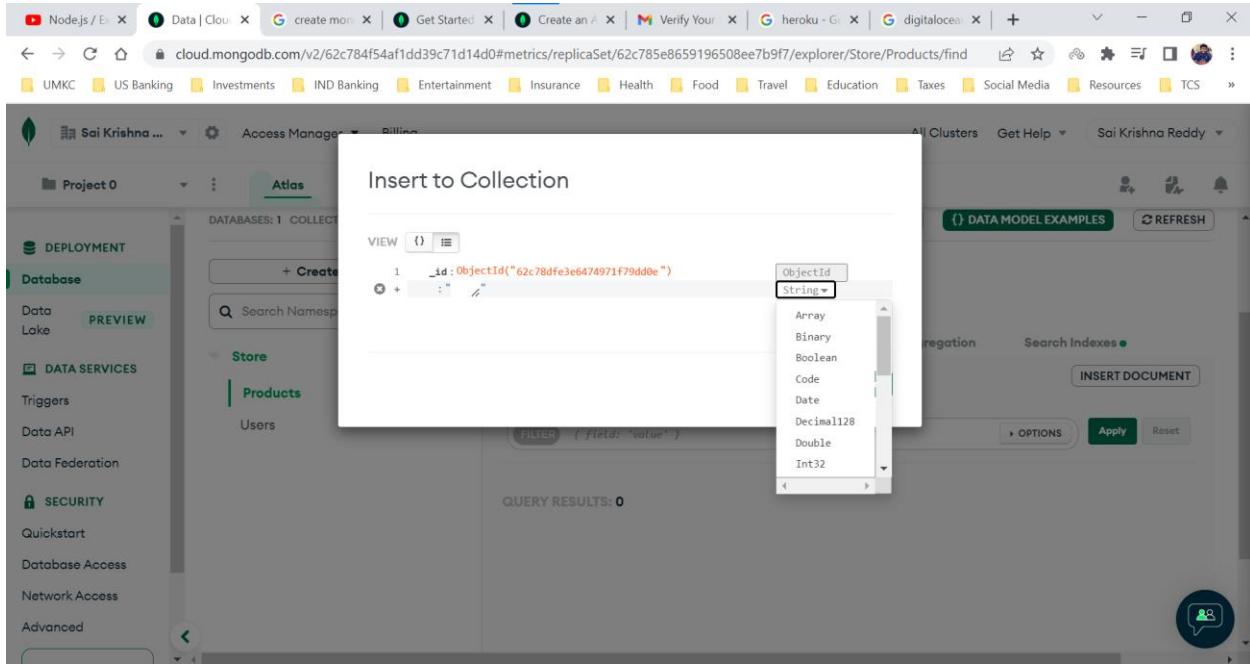
Now we are using the Manual Process to get a gist about MongoDB but once we start developing from server, we do all the operations using a tool called **mongoose**.

Let us insert our first document (a single item) into the Products collection.

Click on Insert Document

The screenshot shows the same MongoDB Atlas interface as before, but now the 'Store.Products' collection has 'TOTAL DOCUMENTS: 0'. Below the collection details, the 'QUERY RESULTS: 0' section is visible.

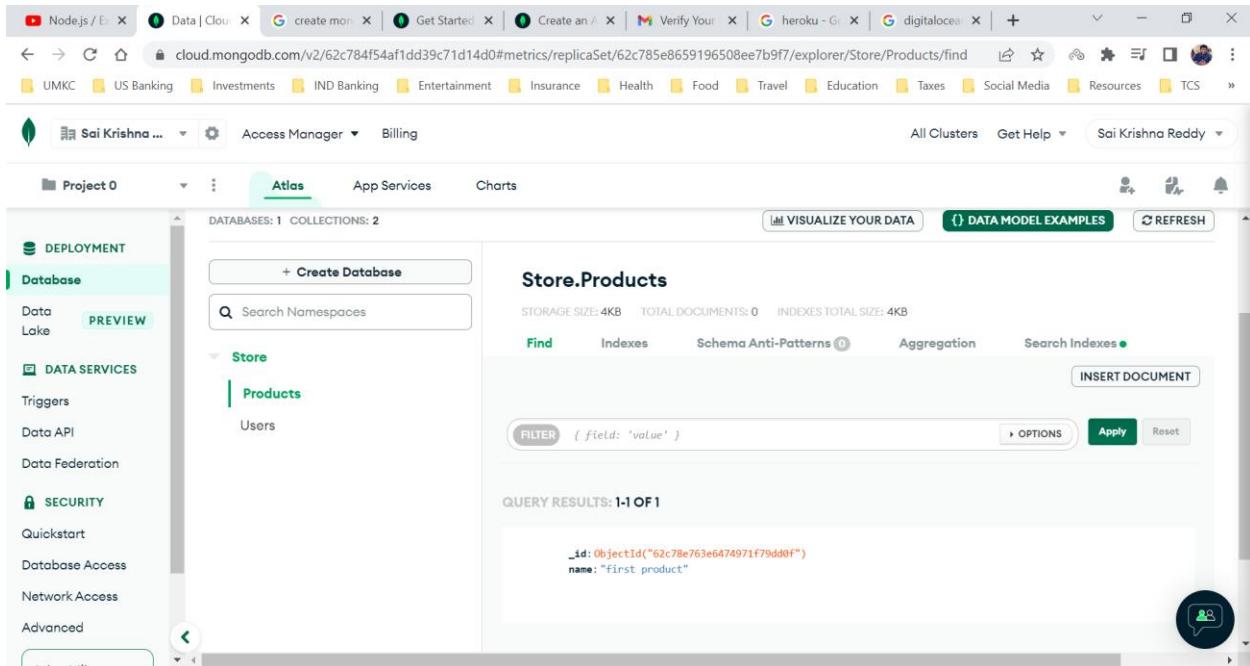
Here each document contains two details (key-value pairs), the first one being the id which is given by MongoDB by default.



The screenshot shows the MongoDB Atlas interface. On the left, there's a sidebar with 'Project 0' selected. Under 'Database', 'Data Lake' is chosen. In the center, under 'Store', 'Products' is selected. A modal window titled 'Insert to Collection' is open. Inside, there's a JSON-like entry: '_id : ObjectId("62c784f54af1dd39c71d14d0")'. To the right of this entry is a dropdown menu with 'String' selected, and other options like 'ObjectId', 'Array', 'Binary', etc., are listed. At the bottom of the modal, there's an 'INSERT DOCUMENT' button.

Now the second the property can be of any data type.

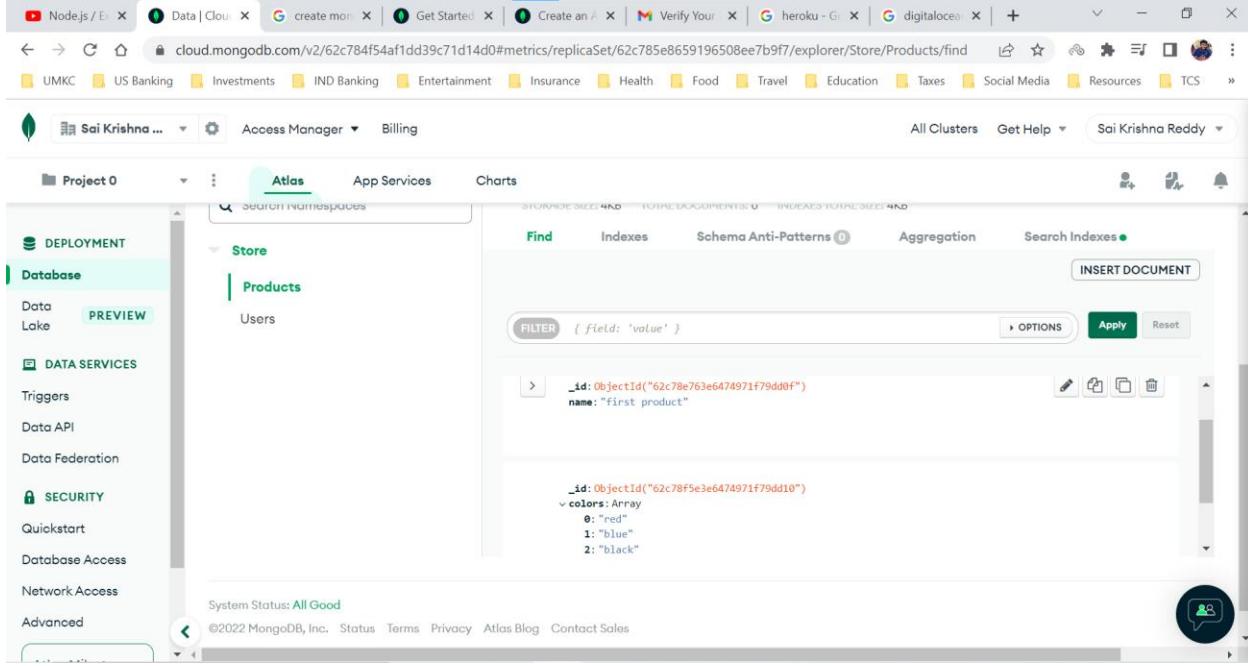
We have a created a document (a single item) with a key named name and value being first product.



The screenshot shows the MongoDB Atlas interface. The left sidebar is identical to the previous one. In the center, under 'Store', 'Products' is selected. The main area shows a table with one row. The row has two columns: '_id: ObjectId("62c78e763e6474971f79dd0f")' and 'name: "First product"'. Below the table, it says 'QUERY RESULTS: 1-1 OF 1'. At the bottom of the screen, there's a footer with various icons.

In MongoDB, documents have a dynamic schema, which means that documents in same collection doesn't need to have same set of fields or the structure.

Since MongoDB has a dynamic schema, we are able to create two documents with different schema for each document (equivalent to row in SQL).



The screenshot shows the MongoDB Atlas interface. On the left, there's a sidebar with sections like Deployment, Database (selected), Data Services, and Security. The main area is titled 'Store' and shows two documents under 'Products': 'Users' and 'Products'. The 'Products' document is expanded, showing its schema. A filter bar at the top says 'FILTER { field: 'value' }'. Below it, the first document is shown with fields: '_id: ObjectId("62c78e763e6474971f79dd0f")' and 'name: "First product"'. The second document is partially visible with fields: '_id: ObjectId("62c78f5e3e6474971f79dd10")', 'colors: Array', '0: "red"', '1: "blue"', and '2: "black"'. At the bottom, there's a footer with 'System Status: All Good' and links to Status, Terms, Privacy, Atlas Blog, and Contact Sales.

Just because MongoDB is allowing us to do that doesn't mean that we should create different schema for each document in a collection.

We will be using additional library named Mongoose which will setup that structure for us.

CRUD Operations using our Manual Setup

CRUD stands for Create, Read, Update, Delete

Create

Inserting a Document

The screenshot shows the MongoDB Atlas interface. On the left, there's a sidebar with 'Project 0' selected. Under 'DEPLOYMENT', 'Database' is highlighted. In the center, under 'Store', 'Products' is selected. A modal window titled 'Insert to Collection' is open, showing the following JSON document:

```
1 _id : ObjectId("62c794023e6474971f79dd11")
2 name : "second products"
```

Below the modal, the 'QUERY RESULTS' section shows two documents:

```
_id: ObjectId("62c78e763e6474971f79dd0f")
name: "first product"
```

```
_id: ObjectId("62c78e763e6474971f79dd0f")
name: "first product"
```

Read

Reading all the documents in the Products collection

The screenshot shows the MongoDB Atlas interface. The left sidebar has 'Project 0' selected. Under 'DEPLOYMENT', 'Database' is highlighted. In the center, under 'Store', 'Products' is selected. The top bar shows 'Databases: 1 COLLECTIONS: 2'. Below the bar, it says 'Store.Products' with 'STORAGE SIZE: 4KB TOTAL DOCUMENTS: 0 INDEXES TOTAL SIZE: 4KB'. There are tabs for 'Find', 'Indexes', 'Schema Anti-Patterns', 'Aggregation', and 'Search Indexes'. A 'FILTER' field contains the query `{ field: 'value' }`. Below the filter, three documents are listed:

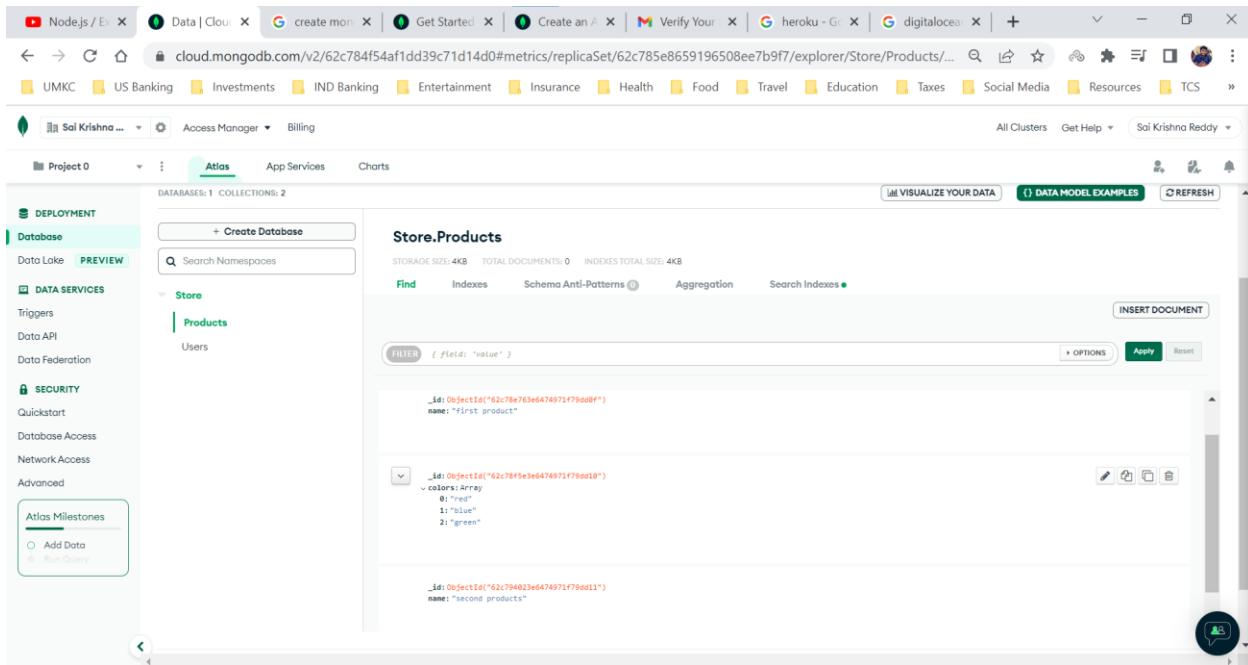
```
_id: ObjectId("62c78e763e6474971f79dd0f")
name: "first product"
```

```
_id: ObjectId("62c78e763e6474971f79dd10")
colors: Array
0: "red"
1: "blue"
2: "black"
```

```
_id: ObjectId("62c794023e6474971f79dd11")
name: "second products"
```

Update

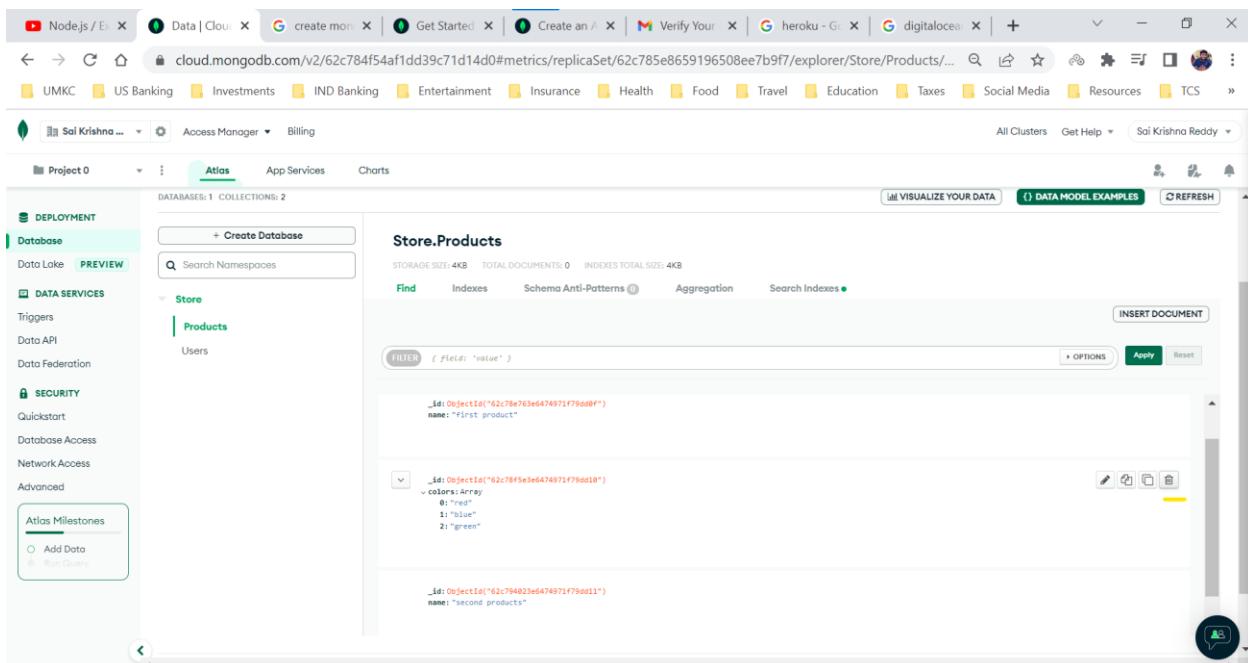
Updating a document in collection. (Updated a item in the color array from black to green)



The screenshot shows the MongoDB Atlas interface. On the left sidebar, under 'Project 0' > 'Store' > 'Products', there is a list of documents. One document is selected, showing its details. The document has an '_id' field of 'ObjectId("62c78e76e6474971f79dd0f")' and a 'name' field of 'first product'. It also has a 'colors' field which is an array containing three items: 'red', 'blue', and 'black'. In the bottom right corner of the document preview, there is a small delete icon.

Delete

Delete a document from the collection (click on the delete icon to delete the document)



This screenshot is similar to the previous one, showing the MongoDB Atlas interface. The 'Store.Products' collection is displayed. A document is selected, showing '_id: ObjectId("62c78e76e6474971f79dd0f")', 'name: "first product"', and 'colors: [red, blue, black]'. The delete icon in the bottom right corner of the document preview is highlighted with a yellow box.

The screenshot shows the MongoDB Atlas interface. On the left, the sidebar includes 'Project 0', 'Database' (selected), 'Data Services', 'Security', and 'Atlas Milestones'. The main area displays the 'Store' database and 'Products' collection. A search bar at the top right contains the query 'name: "First product"'. Below it, two documents are listed:

```
_id: ObjectID("62c78f54af1dd39c71d14d0")
+ colors: Array
  0: "red"
  1: "blue"
  2: "green"

_name: "First product"

_id: ObjectID("62c794023e6474971f798d11")
+ name: "second products"
```

A red box highlights the second document, with a message 'Document Flagged For Deletion.' above it. A 'DELETE' button is visible in the bottom right corner of the document card.

After deleting the documents

The screenshot shows the same MongoDB Atlas interface after the deletion. The list now only contains the first document:

```
_id: ObjectID("62c78f54af1dd39c71d14d0")
+ colors: Array
  0: "red"
  1: "blue"
  2: "green"

_name: "First product"
```

The above CRUD operations are done manually which is not the preferable way to perform the CRUD operations when working frontend and Node JS applications.

CRUD Operations using Mongoose

Once our database is ready, we need to connect to the database from our server. We can use the native MongoDB driver package name MongoDB, but a very popular alternative is to use by the package name **Mongoose**.

Mongoose is object data modeling library and is popular because it comes with bunch of goodies which makes our development faster. We can see the use of goodies in our Node JS projects.

Yes, we can also setup our applications just with native MongoDB package, but the reason mongoose is extremely popular because it has extremely straight forward API and basically it does all the heavy lifting for us.

Command to install Mongoose

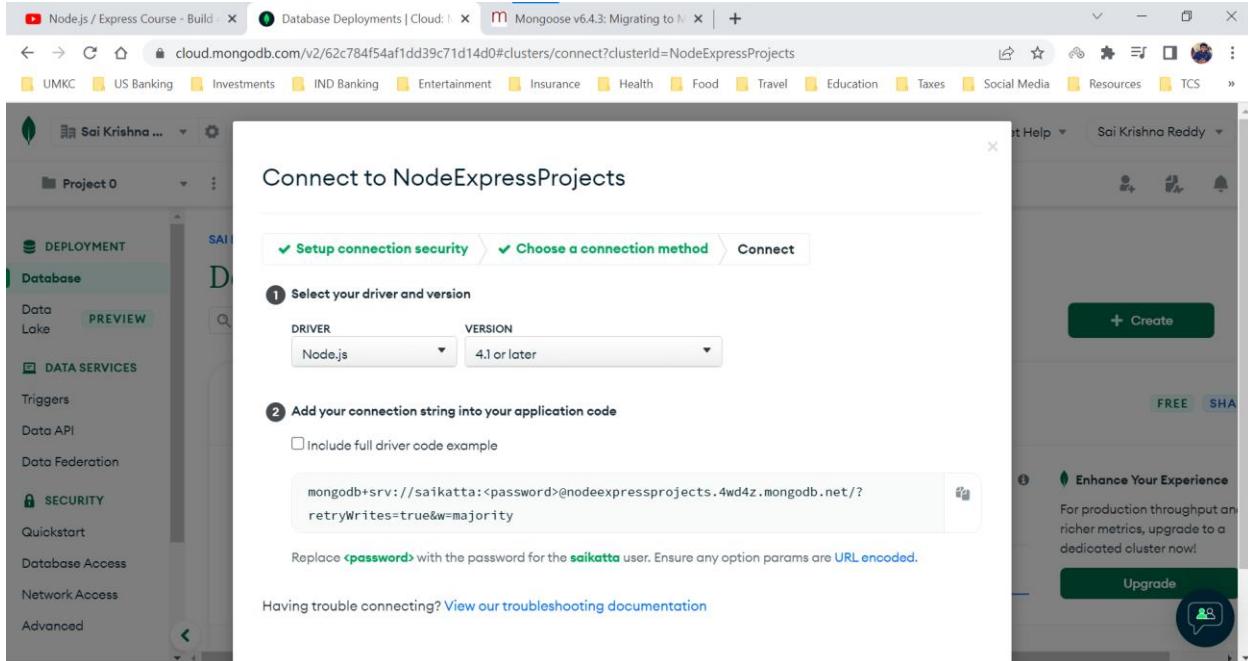
```
npm install mongoose
```

Using Database on Server

First thing we need to do is to setup a connection. To setup the connection with database we are using mongoose library.

Mongoose library is already installed in our code but since here we are using version 6 which has few changes when compared to version 5 (which was used by the instructor in the video).

To setup the connection we need to get the connection string from the MongoDB. We need to use it as a connection string in our code.



connection.js file in db folder

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure under "NODEJS-EXPRESS". The "db" folder is expanded, showing "connection.js" and "app.js".
- Editor:** Displays the content of "connection.js".

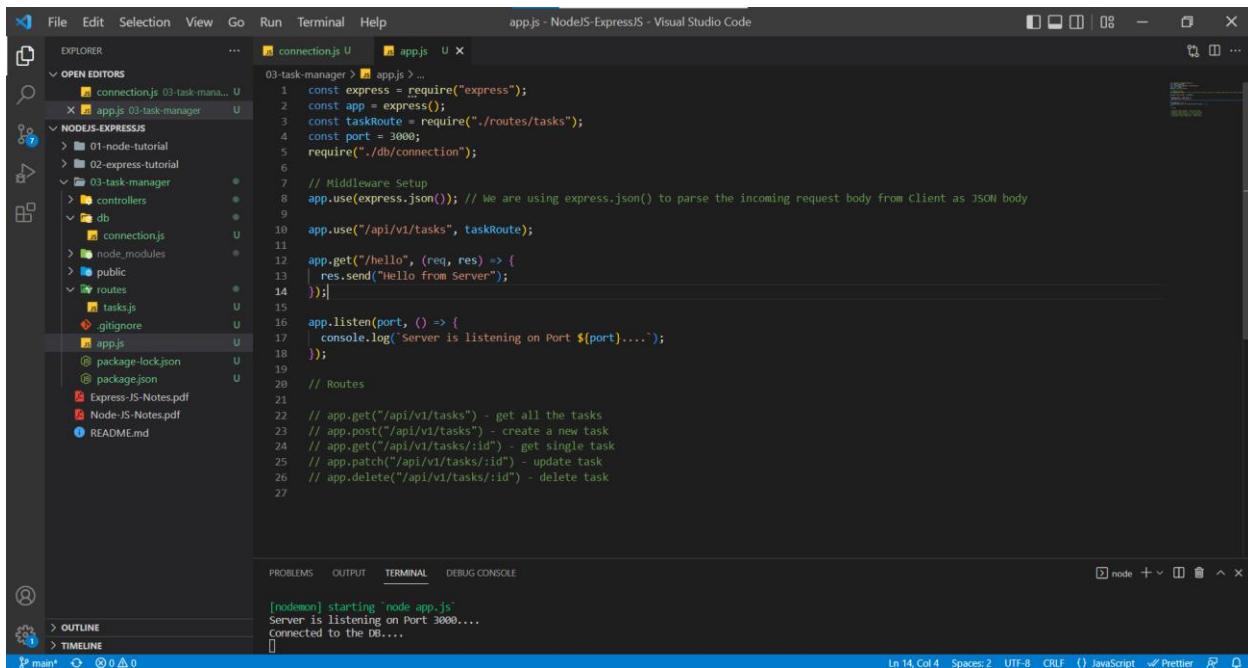
```
const mongoose = require("mongoose");
const connectionString =
  "mongodb+srv://saikatta:HoneyWell@nodeexpressprojects.4wd4z.mongodb.net/03-TASK-MANAGER?retryWrites=true&w=majority";
mongoose
  .connect(connectionString)
  .then(() => console.log("Connected to the DB..."))
  .catch((err) => console.log(err));
```
- Terminal:** Shows the output of the Node.js application.

```
ok: 0,
code: 8000,
codeName: 'AtlasError',
[Symbol(errorLabels)]: Set(1) { 'HandshakeError' }
}
[nodemon] restarting due to changes...
[nodemon] starting 'node app.js'
Server is listening on Port 3000...
Connected to the DB...
```

We need to add our database name in between the / and ?. I have added the database name as 03-TASK-MANAGER. `connect()` method in mongoose library returns a promise hence we try to catch it using the `then()` and `catch()` methods.

As we have learned in the basics of Node JS, when we import some files into `app.js` we are going to those imported files. We want to start the database connection when we want to start our server. Hence, we are importing the `connection.js` file and executing it in the `app.js` file.

app.js file



```
File Edit Selection View Go Run Terminal Help app.js - NodeJS-ExpressJS - Visual Studio Code

OPEN EDITORS
connection.js U app.js U ...
03-task-manager > app.js > ...
1 const express = require("express");
2 const app = express();
3 const taskRoute = require("./routes/tasks");
4 const port = 3000;
5 require("./db/connection");
6
7 // Middleware Setup
8 app.use(express.json()); // We are using express.json() to parse the incoming request body from client as JSON body
9
10 app.use("/api/v1/tasks", taskRoute);
11
12 app.get("/hello", (req, res) => {
13   res.send("Hello from Server");
14 });
15
16 app.listen(port, () => {
17   console.log(`Server is listening on Port ${port}....`);
18 });
19
20 // Routes
21
22 // app.get("/api/v1/tasks") - get all the tasks
23 // app.post("/api/v1/tasks") - create a new task
24 // app.get("/api/v1/tasks/:id") - get single task
25 // app.patch("/api/v1/tasks/:id") - update task
26 // app.delete("/api/v1/tasks/:id") - delete task
27

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
[nodemon] starting "node app.js"
Server is listening on Port 3000...
Connected to the DB...
Ln 14, Col 4 Spaces: 2 UTF-8 CR/LF (JavaScript) Prettier
node + v 🗑 ^ x
outline timeline
main* ⚡ 0 △ 0
```

We have completed the database connection setup but here we do have two problems

1. Exposing our database credentials to others when we try to push our code to GitHub.
2. We are starting the server and then checking the Database connection but ideally, we should check the database connection and then start our server.

The first problem can be solved by using a file containing our important details about the project. To do that we use a library named **dotenv**.

Dotenv is a zero-dependency module that loads environment variables from a .env file into process.env. Storing configuration in the environment separate from code is based on The Twelve-Factor App methodology.

Let us create a file named .env for the whole project, so that we can those secret codes throughout the project. We created the file and added the required variable.

.env file

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer (Left):** Shows the project structure under "NODEJS-EXPRESS". The ".env" file is listed in the "03-task-manager" folder.
- Editor (Top Right):** The ".env" file is open, containing the following code:

```
MONGO_URI = mongodb+srv://saikatta:HoneyWell@nodeexpressprojects.4wd4z.mongodb.net/03-TASK-MANAGER?retryWrites=true&w=majority
```
- Terminal (Bottom):** Shows the output of nodemon starting the app.js file, indicating the server is listening on port 3000 and connected to the database.

Don't forget to add .env extension in .gitignore file

Now we need to get access to the .env file in our app.js file

To get access, first we need to import the dotenv library into app.js and then configure it using **config()** and then we can use the global variable **process** to get hold of the **env** and variable where we stored the value.

app.js

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer (Left):** Shows the project structure under "NODEJS-EXPRESS". The "app.js" file is listed in the "03-task-manager" folder.
- Editor (Top Right):** The "app.js" file is open, showing the following code:

```
const express = require('express');
const app = express();
const taskRoute = require("./routes/tasks");
const port = 3000;
require("./db/connection");
require("dotenv").config();

console.log(process.env.MONGO_URI);

// Middleware Setup
app.use(express.json()); // We are using express.json() to parse the incoming request body from client as JSON body

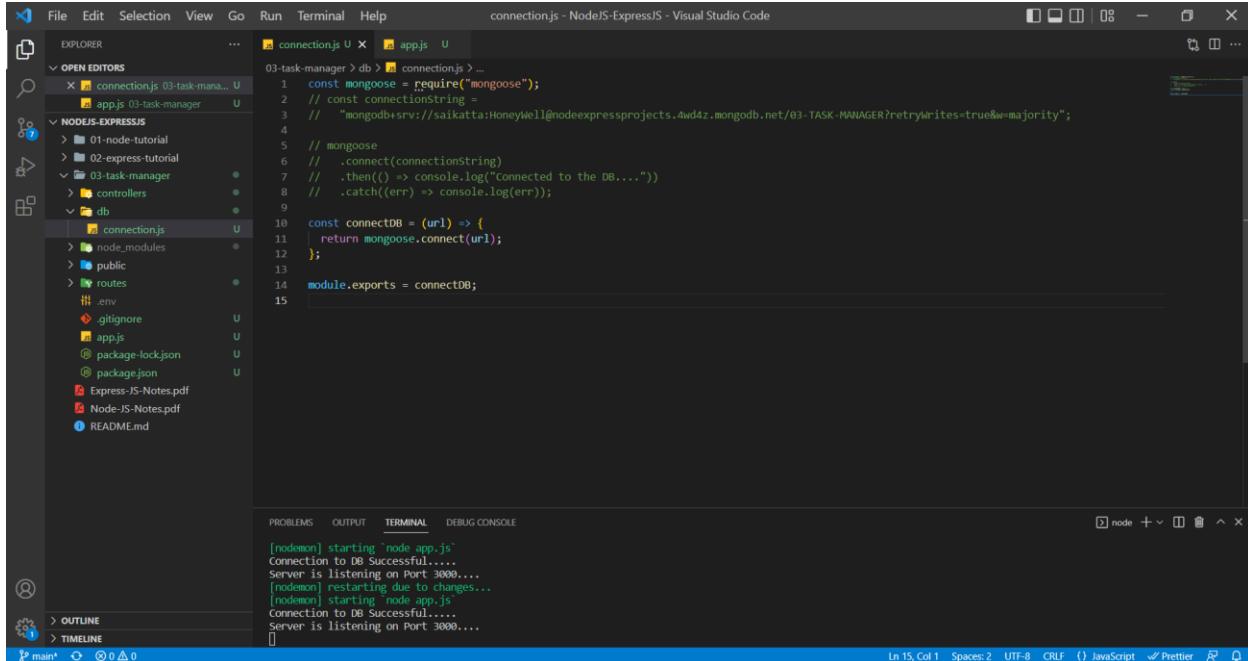
app.use("/api/v1/tasks", taskRoute);

app.get("/hello", (req, res) => {
  res.send("Hello from Server");
});

app.listen(port, () => {
  console.log(`Server is listening on Port ${port}....`);
});
```
- Terminal (Bottom):** Shows the output of running the app.js file, indicating the server is listening on port 3000 and connected to the database.

The second problem can be fixed by writing a logic in the app.js where we need to check the database connection and then start the server also, we will make the code a bit cleaner.

connection.js

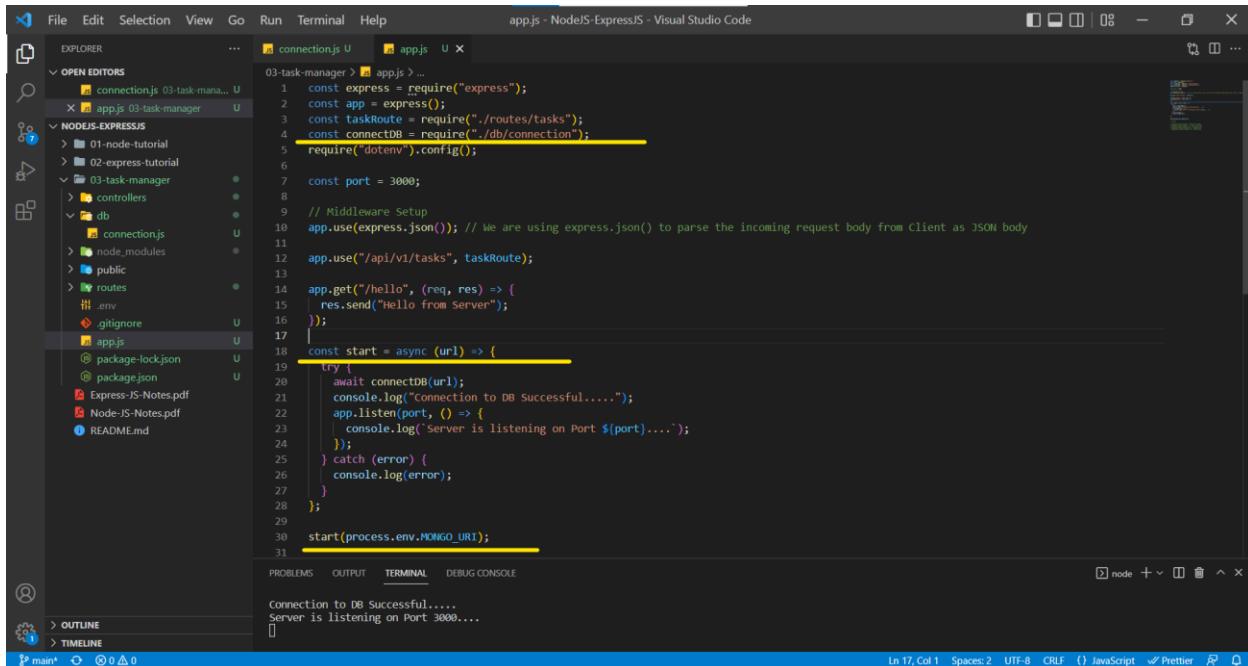


The screenshot shows the Visual Studio Code interface with the connection.js file open in the editor. The code connects to a MongoDB database using mongoose. It defines a connectDB function that takes a URL and returns a promise. The module exports this function. The terminal below shows the output of running nodemon, indicating successful database connection and server listening on port 3000.

```
const mongoose = require("mongoose");
// const connectionString =
// "mongodb+srv://saikatta@HoneyWell/nodeexpressprojects.4wd4z.mongodb.net/03-TASK-MANAGER?retryWrites=true&w=majority";
// mongoose
// .connect(connectionString)
// .then(() => console.log("Connected to the DB...."))
// .catch((err) => console.log(err));
const connectDB = (url) => {
  return mongoose.connect(url);
};
module.exports = connectDB;
```

```
[nodemon] starting "node app.js"
Connection to DB Successful.....
Server is listening on Port 3000....
[nodemon] restarting due to changes...
[nodemon] starting "node app.js"
Connection to DB Successful.....
Server is listening on Port 3000....
```

app.js



The screenshot shows the Visual Studio Code interface with the app.js file open in the editor. The code sets up an Express application, includes routes for tasks, and starts the server. It uses dotenv for environment variables and mongoose for the database. The start function attempts to connect to the database and then starts the server on port 3000. The terminal below shows the output of running nodemon, indicating successful database connection and server listening on port 3000.

```
const express = require("express");
const app = express();
const taskRoute = require("./routes/tasks");
const connectDB = require("./db/connection");
require("dotenv").config();
const port = 3000;
// Middleware Setup
app.use(express.json()); // We are using express.json() to parse the incoming request body from client as JSON body
app.use("/api/v1/tasks", taskRoute);
app.get("/hello", (req, res) => {
  res.send("Hello from server");
});
const start = async (url) => {
  try {
    await connectDB(url);
    console.log("Connection to DB Successful.....");
    app.listen(port, () => {
      console.log(`Server is listening on Port ${port}.....`);
    });
  } catch (error) {
    console.log(error);
  }
};
start(process.env.MONGO_URI);
```

```
Connection to DB Successful.....
Server is listening on Port 3000.....
```

Now, let us set a structure for our future documents and assign them to collections. We are going to do that using schema and model from mongoose.

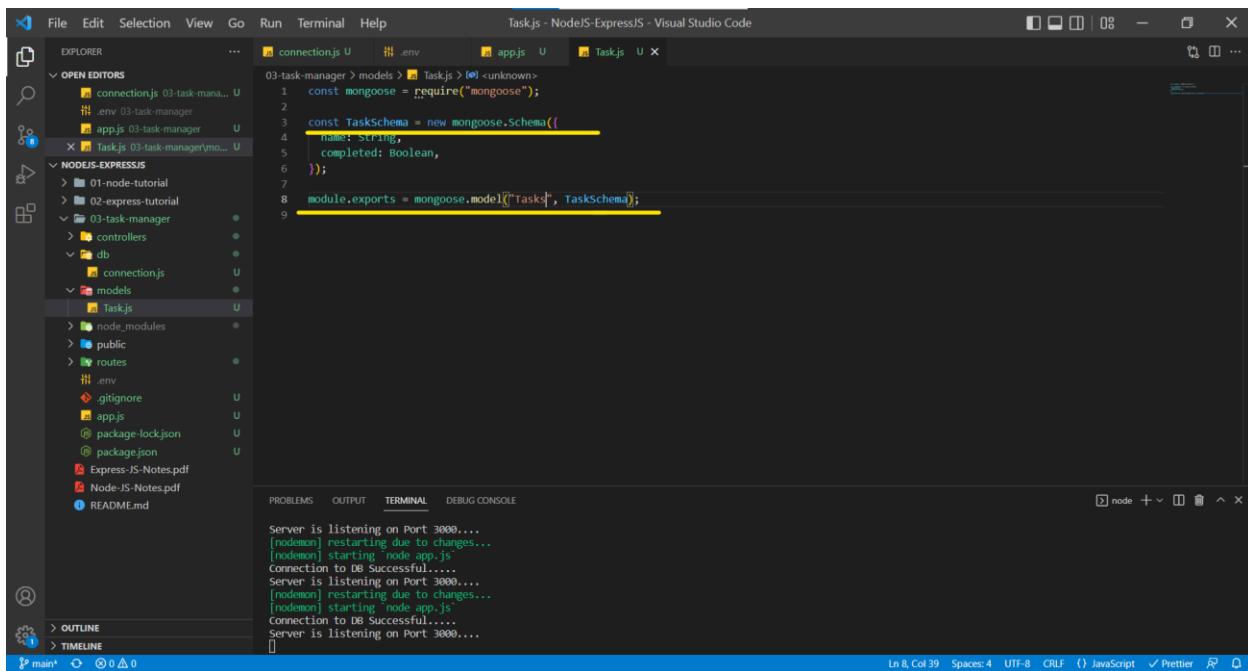
We will create a folder named models and inside of models we create a file named Task.js. With the help of mongoose schema, we are going to set up the structure of the documents in a collection.

In schema we define the name of the key and Schema Type the key is going to store. Once we have the schema/structure for the data now we want to set up the model.

Model is representation for collection. In Mongoose a model is a wrapper for the schema so if the schema defines the structure for the document, a mongoose model provides an interface to the database. Using the model, we will be able to CREATE, READ, UPDATE and DELETE our documents with great ease.

While creating a model, the first argument we use is the singular name of the collection your model is for. Mongoose automatically looks for the plural, lowercase version of your model name.

Task.js in models folder



```
File Edit Selection View Go Run Terminal Help Task.js - NodeJS-ExpressJS - Visual Studio Code

OPEN EDITORS
connection.js U .env app.js U Task.js U
03-task-manager > models > Task.js > <unknown>
1 const mongoose = require("mongoose");
2
3 const TaskSchema = new mongoose.Schema({
4   name: String,
5   completed: Boolean,
6 });
7
8 module.exports = mongoose.model("Tasks", TaskSchema);
```

The screenshot shows the Visual Studio Code interface with the Task.js file open in the editor. The file contains the code for defining a mongoose schema and a model. The schema has a field 'name' of type String and a field 'completed' of type Boolean. The model is named 'Tasks'. The code is syntax-highlighted, and the terminal tab shows logs of the application starting and connecting to a database.

Now we have created a model and we just need to use those models in our controllers to perform the CRUD Operations.

An Instance of a model is called document. Creating them and saving them to database is so easy.

Task.js form models folder.

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure under "NODEJS-EXPRESSJS". The "models" folder contains "Task.js".
- Code Editor:** Displays the content of "Task.js":

```
1 const mongoose = require("mongoose");
2
3 const TaskSchema = new mongoose.Schema({
4   name: String,
5   completed: Boolean,
6 });
7
8 module.exports = mongoose.model("Tasks", TaskSchema);
```

- Terminal:** Shows logs from nodemon:

```
Server is listening on Port 3000...
[nodemon] restarting due to changes...
[nodemon] starting "node app.js"
Connection to DB Successful.....
Server is listening on Port 3000...
[nodemon] restarting due to changes...
[nodemon] starting "node app.js"
Connection to DB Successful.....
Server is listening on Port 3000....
```

A Schema also acts like a validator which ignores other properties that are being sent in the request and will only add the properties which are mentioned in the schema to the database.

app.js

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure under "NODEJS-EXPRESSJS". The "controllers" folder contains "tasks.js".
- Code Editor:** Displays the content of "tasks.js":

```
1 const express = require("express");
2 const Task = require("../models/Task");
3
4 const getAllTasks = (req, res) => {
5   res.send("All items from the controller");
6 };
7
8 const createTask = async (req, res) => {
9   try {
10     const task = await Task.create(req.body);
11     res.status(201).json(task);
12   } catch (error) {
13     res.status(400).json(error);
14   }
15 };
16
17 const getTask = (req, res) => {
18   res.json({ id: req.params.id });
19 };
20
21 const updateTask = (req, res) => {
22   res.send("Update Task");
23 };
24
25 const deleteTask = (req, res) => {
26   res.send("Delete Task");
27 };
28
```

- Terminal:** Shows logs from nodemon:

```
Server is listening on Port 3000...
[nodemon] restarting due to changes...
[nodemon] starting "node app.js"
Connection to DB Successful.....
Server is listening on Port 3000...
[nodemon] restarting due to changes...
[nodemon] starting "node app.js"
Connection to DB Successful.....
Server is listening on Port 3000....
```

As we know that each instance of model is a document, we are using the `create()` method available in mongoose library to create a document and save it in the database. `create()` method in background calls the `save()` method to save the document in the database.

create() method returns a promise hence we are **async** and **await** to handle the promises. We are using the status 201 because it indicates the success using a POST method.

We are using the try catch blocks to handle if promise is rejected due to any database connectivity issue or any other issues.

POST Request from Postman

The screenshot shows the Postman application interface. The top navigation bar includes 'File', 'Edit', 'View', 'Help', 'Home', 'Workspaces', 'Explore', a search bar, and 'Sign In / Create Account' buttons. A yellow banner at the top center reads 'Working locally in Scratch Pad. Switch to a Workspace'. The left sidebar, titled 'Scratch Pad', contains sections for 'Collections' (with '01-Random-API', '02-Express-Tutorial', and '03-Task-Manager' expanded), 'APIs', 'Environments', 'Mock Servers', 'Monitors', and 'History'. The main workspace shows a 'Create Task' POST request for '03-Task-Manager / Create Task'. The 'Body' tab is selected, showing the following JSON payload:

```
1 ... "name": "first task",
2 ... "completed": true
```

The response section shows a 201 Created status with a response body identical to the payload.

We received the status 201 as expected and a JSON object having data of the document which we inserted with two parameters (`_id` and `_v`). Those two parameters are added by MongoDB by default.

Before refreshing the screen in MongoDB

The screenshot shows the MongoDB Cloud interface. On the left, the sidebar has sections for DEPLOYMENT, Database (selected), DATA SERVICES, and SECURITY. Under Database, there's a PREVIEW section with Data Lake. The main area shows the 'Store' database with the 'Products' collection selected. The collection details show a storage size of 36KB, 2 documents, and an index size of 36KB. A document is displayed with the following fields:

```
_id: ObjectId("62c794023e6474971f79dd11")
name: "Second products"
```

After refreshing the screen, we can see the database **03-TASK-MANAGER** being created and having collection **tasks** with one document.

The screenshot shows the MongoDB Cloud interface after refreshing. The sidebar now shows the '03-TASK-MANAGER' database with its 'tasks' collection selected. The collection details show a storage size of 20KB, 1 document, and an index size of 20KB. A document is displayed with the following fields:

```
_id: ObjectId("62c8db52ef8f7dc52367e41f")
name: "First task"
completed: true
__v: 0
```

When we try to add additional parameters into the request body, Mongoose will strictly ignore the additional parameters and will only add the parameters mentioned in the Schema into the database. Mongoose performs validation using the Schema.

Additional Parameters in the Request Body in Postman

The screenshot shows the Postman application interface. The top navigation bar includes 'File', 'Edit', 'View', and 'Help' menus, along with 'Sign In' and 'Create Account' buttons. Below the menu is a search bar with the placeholder 'Search Postman'. The main header features 'Home', 'Workspaces', and 'Explore' buttons. A yellow banner at the top center says 'Working locally in Scratch Pad. Switch to a Workspace'.

The left sidebar, titled 'Scratch Pad', contains sections for 'Collections' (with items '01-Random-API', '02-Express-Tutorial', and '03-Task-Manager'), 'APIs' (with 'Get All Tasks' selected), 'Environments', 'Mock Servers', 'Monitors', and 'History'. The '03-Task-Manager' section has sub-options: 'POST Create Task' (selected), 'GET Get Single Task', 'PATCH Update Task', and 'DELETE Delete Task'.

The central workspace shows a 'POST Create Task' request in the '03-Task-Manager / Create Task' collection. The request method is 'POST' with the URL '({URL})/tasks'. The 'Body' tab is active, showing a JSON payload:

```
1 {  
2   "name": "test-task",  
3   "completed": false,  
4   "random": "abcdefghijkl",  
5   "checked": false  
6 }
```

The 'Body' tab also includes tabs for 'Pretty', 'Raw', 'Preview', 'Visualize', and 'JSON'. At the bottom of the workspace, status indicators show '201 Created', '481 ms', '319 B', and 'Save Response'.

Document in the database after ignoring the additional parameters

The screenshot shows the MongoDB Cloud interface with the following details:

- Top Bar:** Node.js / Express Course - Build, dotenv - npm, Mongo v6.4.4, Data | Cloud: MongoDB Cloud.
- Side Navigation:** DEPLOYMENT (selected), Database, DATA SERVICES, Triggers, Data API, Data Federation, SECURITY, Quickstart, Database Access, Network Access, Advanced.
- Header:** Project 0, Access Manager, Billing, All Clusters, Get Help, Sai Krishna Reddy.
- Atlas Tab:** + Create Database, Search Namespaces.
- Collection List:** 03-TASK-MANAGER (selected), tasks.
- Collection Details:** 03-TASK-MANAGER.tasks, STORAGE SIZE: 36KB, TOTAL DOCUMENTS: 2, INDEXES TOTAL SIZE: 36KB.
- Actions:** Find, Indexes, Schema Anti-Patterns, Aggregation, Search Indexes, INSERT DOCUMENT.
- Document Preview:** One document is shown:

```
name: "first task"
completed: true
__v: 0
```
- Bottom:** FILTER { field: 'value' }, OPTIONS, Apply, Reset.

Even though Mongoose acts like a validator for ignoring additional parameters but it doesn't handle when we send an empty object or if we don't send the parameters mentioned in the schema.

To handle the validations, we provide validators for each key in the schema.

Let us see when we don't send parameters present in the Schema

The screenshot shows the Postman interface. In the left sidebar, under 'Collections', there is a tree view with '01-Random-API', '02-Express-Tutorial', and '03-Task-Manager'. Under '03-Task-Manager', there are five items: 'Get All Tasks', 'Create Task' (selected), 'Get Single Task', 'Update Task', and 'Delete Task'. The main workspace shows a 'POST Create Task' request. The 'Body' tab is selected, showing a JSON payload with only the 'name' field:

```
1 "name": "testing another task"
```

. Below the body, the response is shown in pretty JSON format:

```
1 { 2   "name": "testing another task", 3   "_id": "62c8e225ef8f7dc52367e423", 4   "__v": 0}
```

. The status bar at the bottom indicates a 201 Created response.

Document is created in the collection only with name property.

The screenshot shows the MongoDB Cloud Atlas interface. On the left, the navigation pane includes 'Project 0', 'DEPLOYMENT', 'Database' (selected), 'DATA SERVICES', 'SECURITY', and 'Atlas Milestones'. The 'Database' section shows '03-TASK-MANAGER' and 'tasks'. The main area displays the 'tasks' collection with two documents:

```
_id: ObjectId("62c8e225ef8f7dc52367e421")
name: "test task"
completed: false
__v: 0
```

 and

```
_id: ObjectId("62c8e225ef8f7dc52367e423")
name: "testing another task"
completed: false
__v: 0
```

. The bottom of the screen shows system status and footer links.

Let us see what happens when we send an empty value for key **name**.

The screenshot shows the Postman interface. In the left sidebar, under 'Collections', there is a section for '03-Task-Manager' which includes 'Get All Tasks', 'POST Create Task', 'GET Get Single Task', 'PATCH Update Task', and 'DELETE Delete Task'. The main area shows a POST request to '((URL))/tasks'. The 'Body' tab is selected, showing the following JSON:

```
1 "name": ""
```

Below the body, the response status is shown as 201 Created with a response time of 63 ms and a size of 292 B. The response body is also displayed:

```
1 "name": "",  
2 "_id": "62c8e2b1ef8f7dc52367e425",  
3 "__v": 0
```

Document with name property is created but it contains an empty value

The screenshot shows the MongoDB Cloud Atlas interface. On the left, the navigation bar includes 'Node.js Express Course - Build', 'dotenv - npm', 'Mongoose v6.4.4', 'Data | Cloud: MongoDB Cloud', and a '+' button. The main area shows the '03-TASK-MANAGER' database with the 'tasks' collection. The 'Find' tab is selected, displaying the following document:

```
_id: ObjectId("62c8e225e98f7dc52367e423")  
name: "testing another task"  
__v: 0
```

At the bottom, the system status is listed as 'All Good'.

Let us see what happens when we send an empty object

The screenshot shows the Postman interface. In the left sidebar, under 'APIs', there is a section for '03-Task-Manager' which includes 'POST Create Task'. The main workspace shows a POST request to '03-Task-Manager / Create Task'. The 'Body' tab is selected, showing a JSON object with one field:

```
1
2   "_id": "62c8e316ef8f7dc52367e427",
3   "_v": 0
4 }
```

Below the body, the response status is '201 Created' with a response time of '50 ms' and a size of '282 B'. The response body is identical to the one shown above.

Document is created with no name and completed properties

The screenshot shows the MongoDB Cloud Atlas interface. On the left, the navigation bar includes 'Project 0', 'Atlas', 'App Services', and 'Charts'. Under 'DEPLOYMENT', there is a 'Database' section with '03-TASK-MANAGER' and a 'tasks' collection. The main area shows the 'Find' results for the 'tasks' collection. A search bar at the top has the query 'field: "value"'. Two documents are listed:

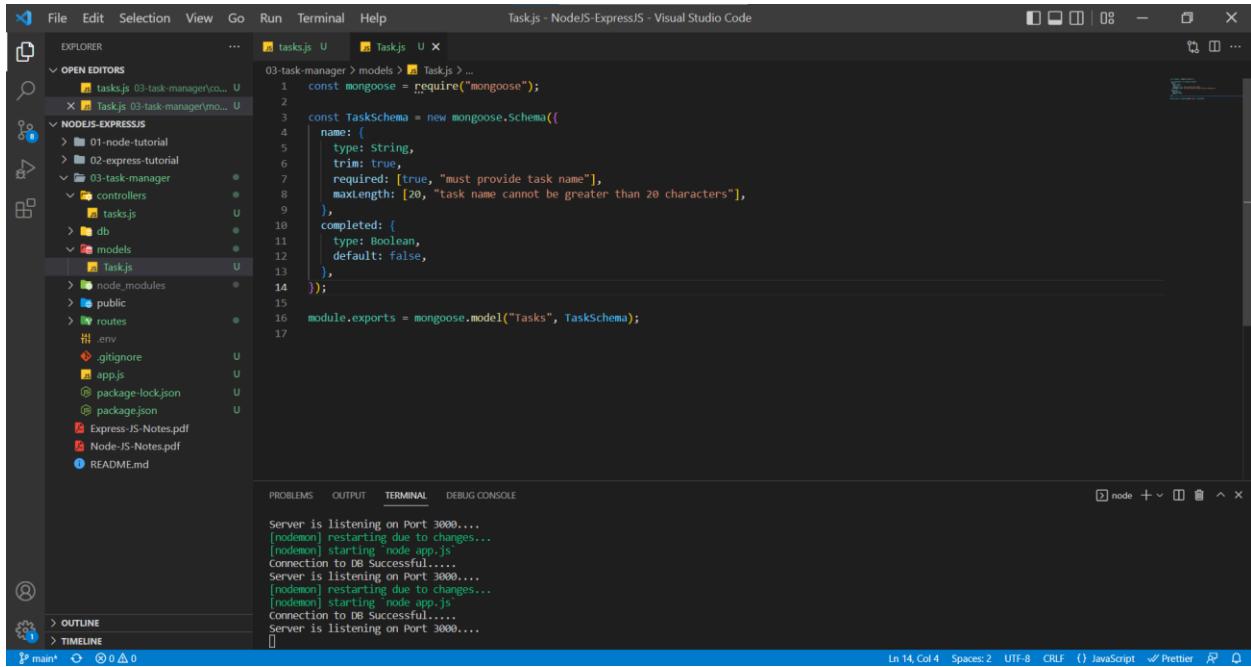
```
_id: ObjectId("62c8e2b1ef8f7dc52367e425")
name: ""
__v: 0

_id: ObjectId("62c8e316ef8f7dc52367e427")
__v: 0
```

At the bottom of the interface, it says 'System Status: All Good'.

To handle the above all issues we provide validators in the Schema

Task.js in models folder



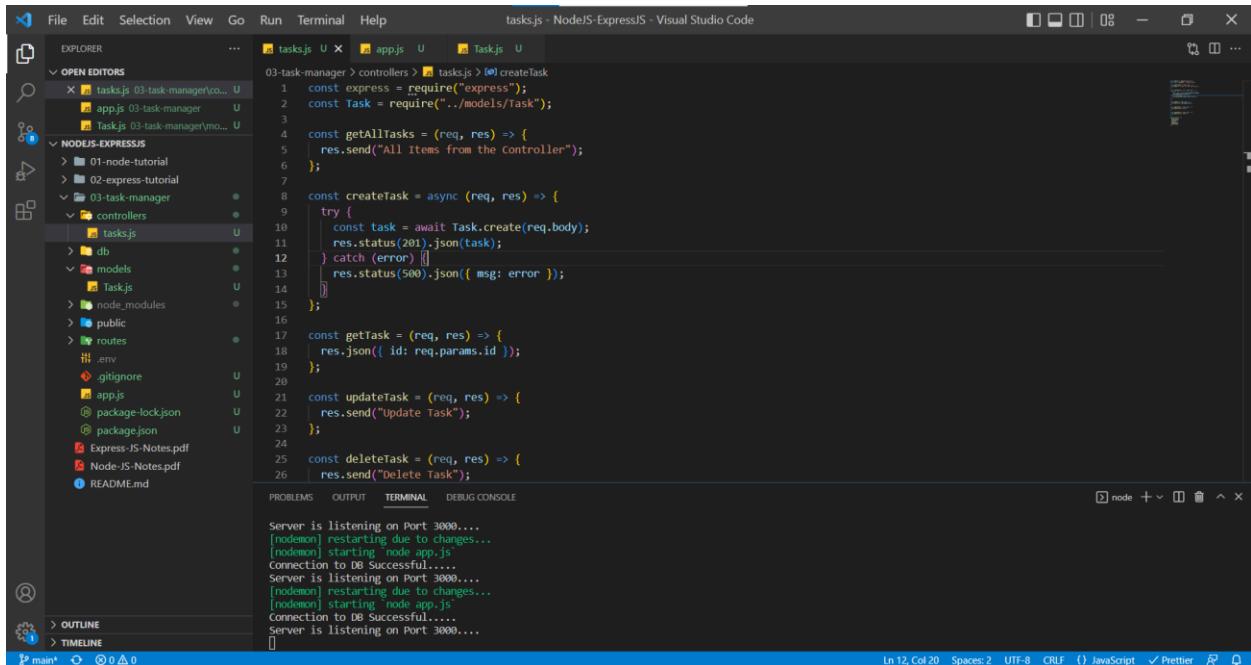
The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure with files like tasks.js, Task.js, and various node modules.
- Code Editor:** Displays the content of Task.js, which defines a mongoose schema for a task. It includes validation rules for the task name (required, string, max length 20) and completed status (boolean, default false).
- Terminal:** Shows logs from nodemon indicating the server is listening on port 3000 and restarting due to changes.
- Status Bar:** Shows the current file is Task.js, and other details like line 14, column 4, and file encoding.

Now let's see how the validators work

For any custom message to be sent as a response, we need to add array with first index being the datatype and the second index being the custom message. We can see in the above example for required and maxLength.

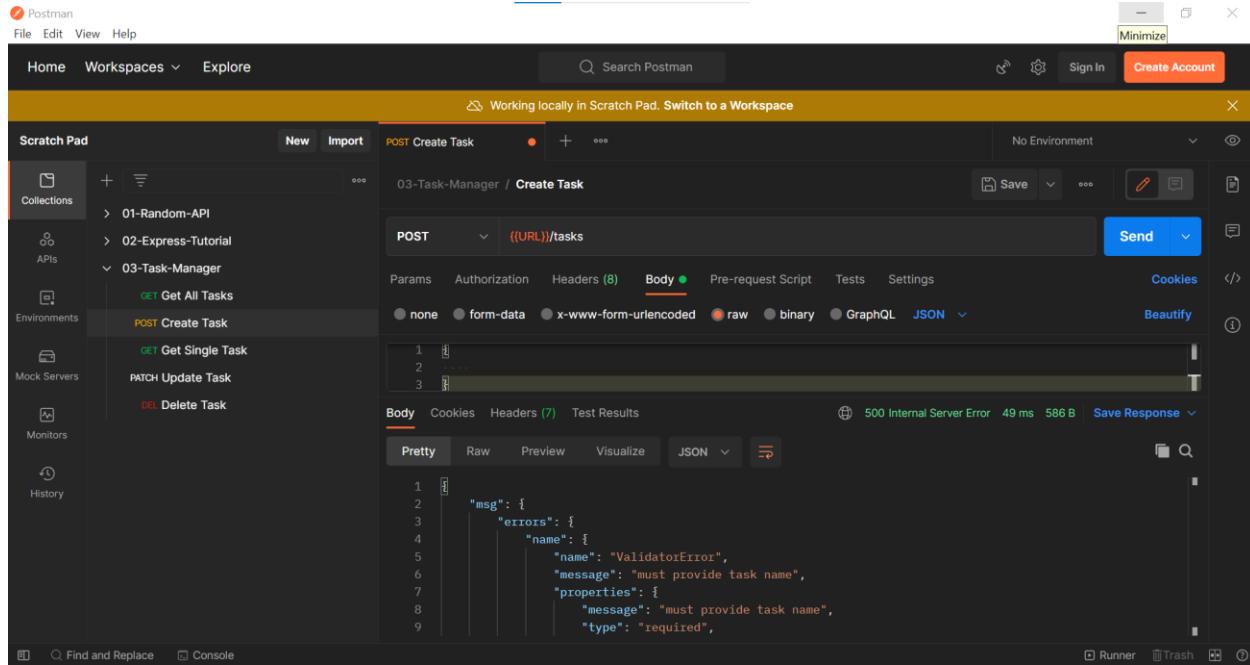
app.js



The screenshot shows the Visual Studio Code interface with the following details:

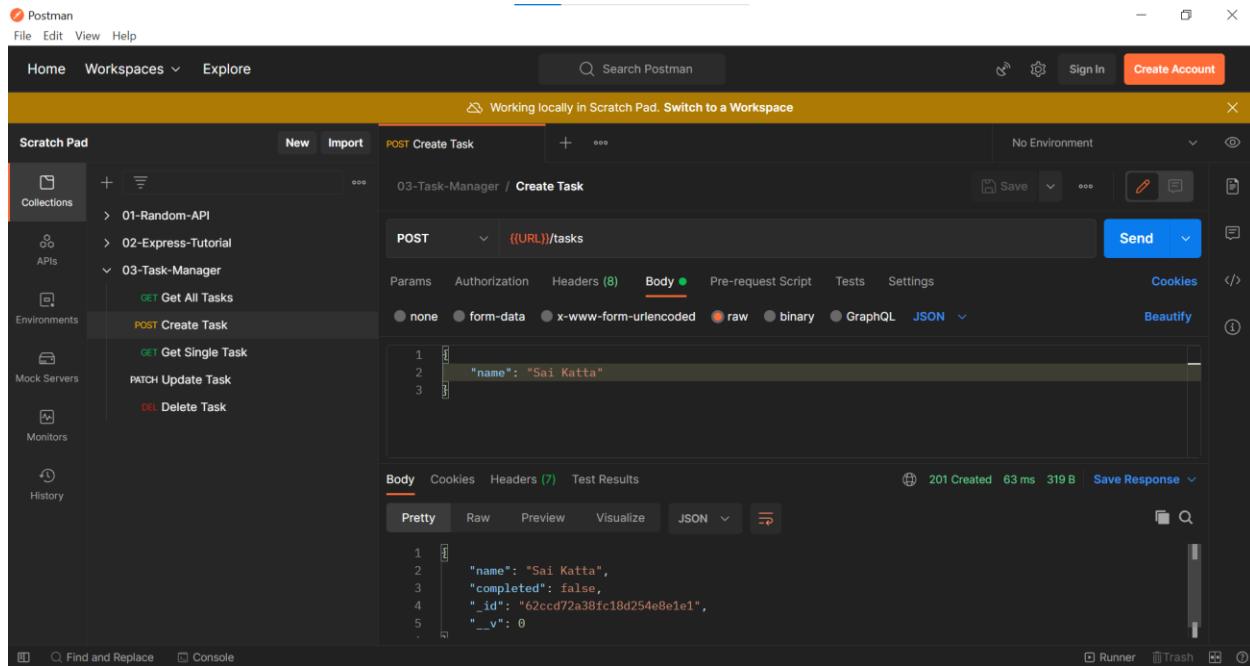
- File Explorer:** Shows the project structure with files like tasks.js, app.js, and various node modules.
- Code Editor:** Displays the content of app.js, which contains several middleware functions (express, task controller) and route handlers (createTask, getAllTasks, updateTask, deleteTask) using the task schema defined in Task.js.
- Terminal:** Shows logs from nodemon indicating the server is listening on port 3000 and restarting due to changes.
- Status Bar:** Shows the current file is app.js, and other details like line 12, column 20, and file encoding.

When we send an empty object



The screenshot shows the Postman interface with a POST request to '03-Task-Manager / Create Task'. The request body is an empty object. The response status is 500 Internal Server Error with a message: "msg": { "errors": { "name": { "name": "ValidatorError", "message": "must provide task name", "properties": { "message": "must provide task name", "type": "required" } } } }

When we send only name, it is created because the second parameter completed is added as false by default in the validators in Task.js file



The screenshot shows the Postman interface with a POST request to '03-Task-Manager / Create Task'. The request body is {"name": "Sai Katta"}. The response status is 201 Created with a message: "msg": { "name": "Sai Katta", "completed": false, "_id": "62cd72a38fc18d254e8e1e1", "__v": 0 }

When we send name as more than 20 characters

Note:

Generally, when we send a request from Client, we will send the data in the form of JSON object but in the server side when we want to do any operation we cannot work with JSON object, hence we want to convert it from JSON object to JavaScript Object. We can do the conversion by using the Express Middleware function known as express.json() method. After conversion from JSON object to JavaScript object we do perform some operations on the JavaScript object.

Now, we have performed some operations on the JavaScript object, and we want to send it back to the Client. If data must be passed through the client, we need to convert the JavaScript object into JSON object. We can convert the JavaScript object to JSON object by using the express method in response object named json() method.

Ex: res.status(200).json({name: "Sai"});

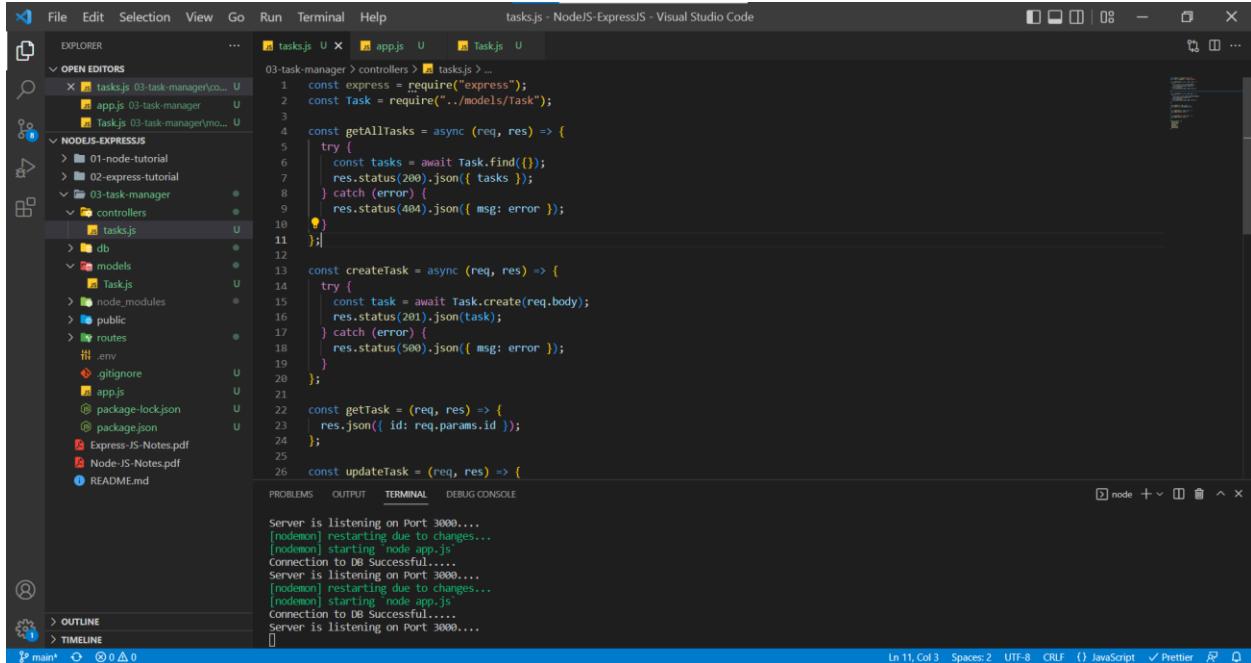
```
res.status(400).json({error});
```

Even Database sends a JavaScript object to the server. It's the server work to send an JSON object to the client and once the client receives the JSON object, the frontend engineer or developer converts the JSON object to JavaScript object and works on the data.

Till now we worked on creating a task, but now we will work to retrieve data from the database and send it to the client.

Mongoose models provide several static helper functions for CRUD operations. Each of these functions returns a mongoose Query object.

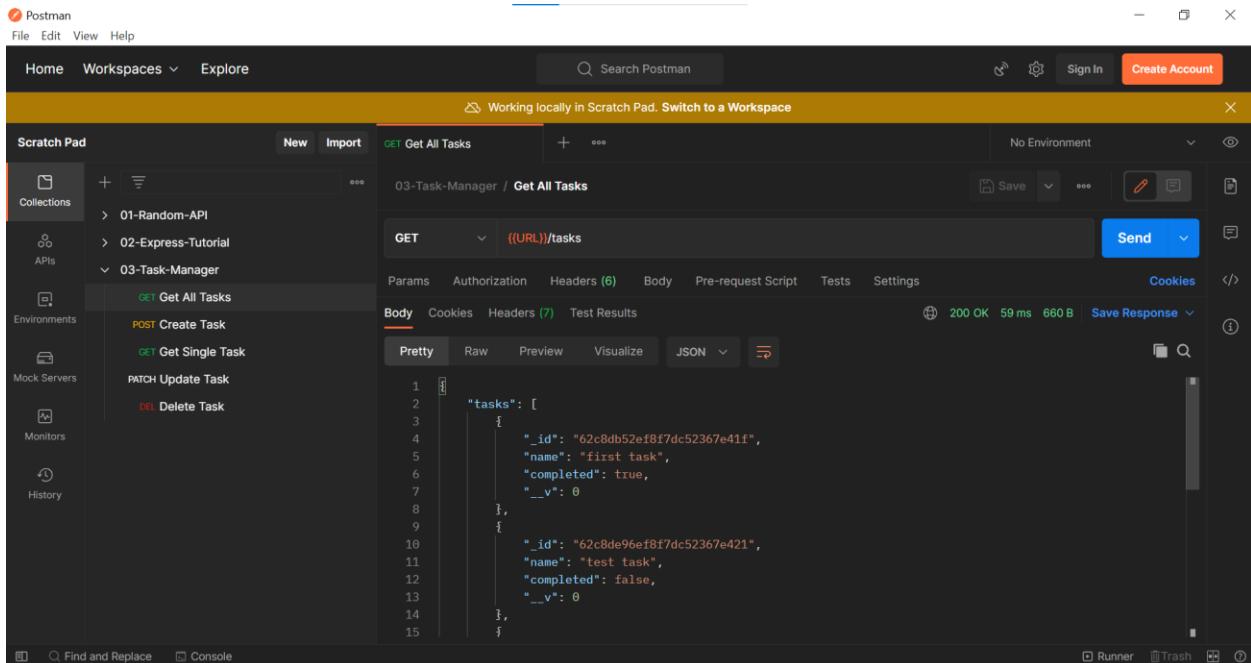
Get All Tasks



```
tasks.js - NodeJS-ExpressJS - Visual Studio Code

File Edit Selection View Go Run Terminal Help
EXPLORER OPEN EDITORS NODEJS-EXPRESSJS
tasks.js U x app.js U Task.js U
03-task-manager > controllers > tasks.js > ...
1 const express = require("express");
2 const Task = require("../models/Task");
3
4 const getAllTasks = async (req, res) => {
5   try {
6     const tasks = await Task.find({});
7     res.status(200).json({ tasks });
8   } catch (error) {
9     res.status(404).json({ msg: error });
10 }
11
12 const createTask = async (req, res) => {
13   try {
14     const task = await Task.create(req.body);
15     res.status(201).json(task);
16   } catch (error) {
17     res.status(500).json({ msg: error });
18   }
19 }
20
21 const getTask = (req, res) => {
22   res.json({ id: req.params.id });
23 };
24
25 const updateTask = (req, res) => {
26
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
Ln 11, Col 3 Spaces: 2 UTF-8 CR LF {} JavaScript ✓ Prettier
Server is listening on Port 3000...
[nodemon] restarting due to changes...
[nodemon] starting 'node app.js'
Connection to DB Successful.....
Server is listening on Port 3000...
[nodemon] restarting due to changes...
[nodemon] starting 'node app.js'
Connection to DB Successful.....
Server is listening on Port 3000...
[]
```

In Postman



Postman

File Edit View Help

Home Workspaces Explore

Working locally in Scratch Pad. Switch to a Workspace

Scratch Pad New Import GET Get All Tasks + ...

03-Task-Manager / Get All Tasks

GET {{URL}}/tasks

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Pretty Raw Preview Visualize JSON

```
1
2   "tasks": [
3     {
4       "_id": "62c8db52ef8f7dc52367e41f",
5       "name": "first task",
6       "completed": true,
7       "__v": 0
8     },
9     {
10       "_id": "62c8de96ef8f7dc52367e421",
11       "name": "test task",
12       "completed": false,
13       "__v": 0
14     }
15   ]
```

Save ... Send </> 200 OK 59 ms 660 B Save Response

Runner Trash

Get Single Task with Particular ID

Finding a Singular task in the database with ID from parameters.

tasks.js

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure under "NODEJS-EXPRESSJS". The "controllers" folder contains "tasks.js" and "Task.js".
- Editor:** The main editor window displays the content of "tasks.js". The code defines several functions: `getTask`, `updateTask`, and `deleteTask`. It uses promises and async/await syntax.
- Terminal:** The terminal at the bottom shows the output of the application's startup logs, indicating it is listening on port 3000 and connecting to a database.

```
File Edit Selection View Go Run Terminal Help tasks.js - NodeJS-ExpressJS - Visual Studio Code

EXPLORER OPEN EDITORS NODEJS-EXPRESSJS
task.js U app.js U Task.js U
03-task-manager > controllers > tasks.js > ...
17 } catch (error) {
18   res.status(500).json({ msg: error });
19 }
20 }

21 const getTask = async (req, res) => {
22   try {
23     const { id: taskID } = req.params;
24     const task = await Task.findOne({ _id: taskID });
25     if (!task) {
26       return res.status(404).json({ msg: `No task with ID: ${taskID}` });
27     }
28     res.status(200).json({ task });
29   } catch (error) {
30     res.status(500).json({ msg: error });
31   }
32 }
33 }

34
35 const updateTask = (req, res) => {
36   res.send("Update Task");
37 }

38
39 const deleteTask = (req, res) => {
40   res.send("Delete Task");
41 }
42

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

Server is listening on Port 3000.... [nodemon] restarting due to changes... [nodemon] starting 'node app.js' Connection to DB Successful.... Server is listening on Port 3000.... [nodemon] restarting due to changes... [nodemon] starting 'node app.js' Connection to DB Successful.... Server is listening on Port 3000....
```

In Postman, with task ID

The screenshot shows the Postman application interface. The top navigation bar includes 'File', 'Edit', 'View', 'Help', 'Home', 'Workspaces', 'Explore', a search bar 'Search Postman', and user options 'Sign In' and 'Create Account'. A yellow banner at the top center says 'Working locally in Scratch Pad. Switch to a Workspace'. The left sidebar has sections for 'Collections' (with '01-Random-API', '02-Express-Tutorial', and '03-Task-Manager'), 'APIs' (with 'Create Task'), 'Environments', 'Mock Servers', 'Monitors', and 'History'. The main workspace shows a 'Scratch Pad' session for '03-Task-Manager / Get Single Task'. The request URL is 'GET ((URL))/tasks/62c8db52ef8f7dc52367e41f'. The 'Params' tab shows a key 'Key' and value 'Value'. The 'Body' tab displays a JSON response:

```
1 [ { "task": { "id": "62c8db52ef8f7dc52367e41f", "name": "first task", "completed": true, "__v": 0 } } ]
```

In Postman, no task with ID containing same length of characters equal to ID characters in Database

Response 404 Status Code

The screenshot shows the Postman application interface. In the center, there's a request card for a GET operation on the URL `((URL))/tasks/62c8db52ef8f7dc52367e41a`. The 'Params' tab is selected, showing a single parameter 'Key' with the value 'Value'. Below the request card, the 'Test Results' section displays a JSON response with a single key-value pair: `"msg": "No task with ID: 62c8db52ef8f7dc52367e41a"`. The status bar at the bottom indicates a 404 Not Found error with 59 ms and 293 B.

In Postman, when ID characters length is less than the ID character length in Database, we receive an error named Cast Error.

Response 500 Status Code

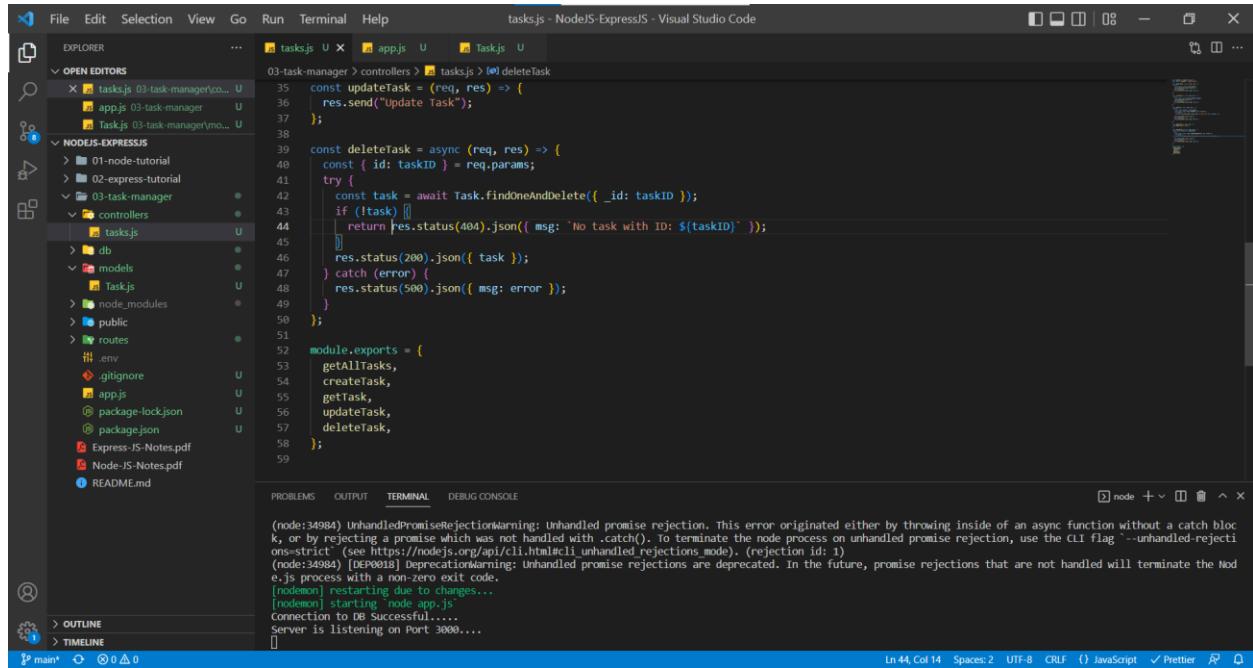
The screenshot shows the Postman application interface. In the center, there's a request card for a GET operation on the URL `((URL))/tasks/jhgjhgj`. The 'Params' tab is selected, showing a single parameter 'Key' with the value 'Value'. Below the request card, the 'Test Results' section displays a complex JSON response starting with `"stringValue": "\\"jhgjhgj\\\"", "valueType": "string", "kind": "ObjectId", "value": "jhgjhgj", "path": "_id", "reason": {}, "name": "CastError", "message": "Cast to ObjectId failed for value \"jhgjhgj\" (type string) at path \"_id\" for model Tasks"`. The status bar at the bottom indicates a 500 Internal Server Error with 12 ms and 501 B.

Delete Task

Deleting a particular task with ID

tasks.js

We shouldn't have two response in one try block, we have return only one so that is the reason we are returning the one inside the if block else we will return the response outside the if block.

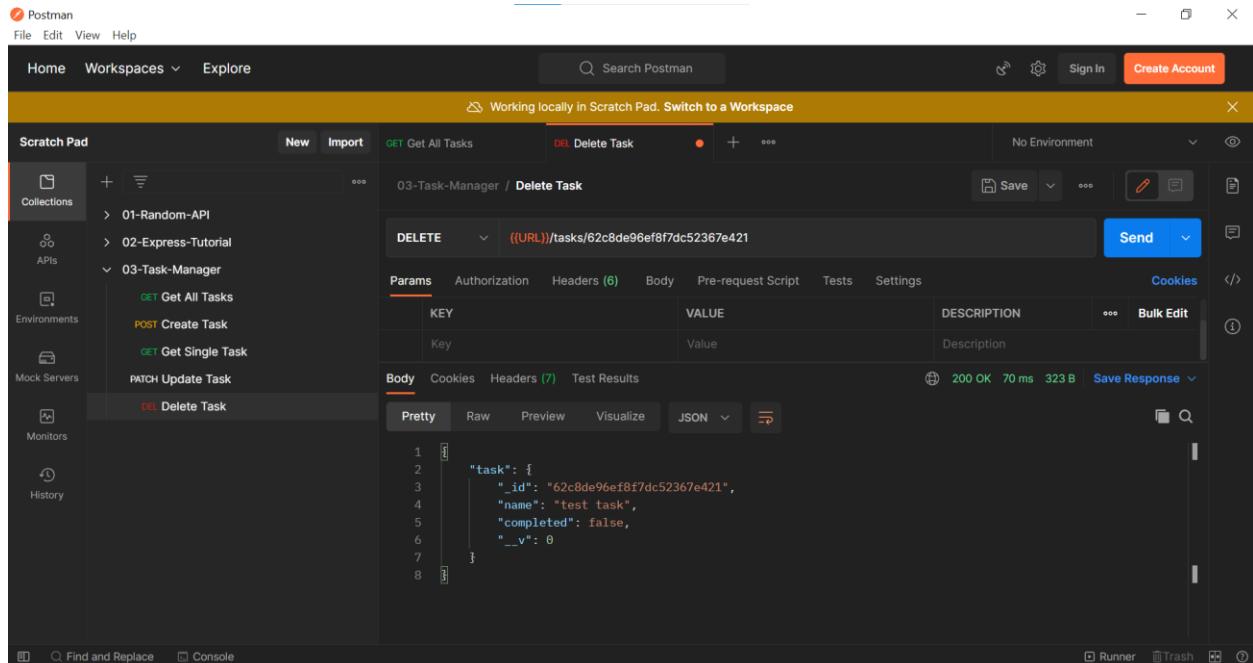


```
const updateTask = (req, res) => {
  res.send("Update Task");
};

const deleteTask = async (req, res) => {
  const { id: taskID } = req.params;
  try {
    const task = await Task.findOneAndDelete({ _id: taskID });
    if (!task) {
      return res.status(404).json({ msg: `No task with ID: ${taskID}` });
    }
    res.status(200).json({ task });
  } catch (error) {
    res.status(500).json({ msg: error });
  }
};

module.exports = {
  getAllTasks,
  createTask,
  getTask,
  updateTask,
  deleteTask,
};
```

In Postman, deleting a task with Particular ID available in the database



The screenshot shows the Postman interface with a Scratch Pad workspace. A DELETE request is being prepared to the URL `((URL))/tasks/62c8de96ef8f7dc52367e421`. The request body is set to a JSON object:

```
{"task": { "_id": "62c8de96ef8f7dc52367e421", "name": "test task", "completed": false, "__v": 0 }}
```

In Postman, deleting a task with ID does not present in database but has equal number of characters

Response Status: 404 – Not task found with ID

The screenshot shows the Postman application interface. The left sidebar is titled "Scratch Pad" and contains sections for Collections, APIs, Environments, Mock Servers, Monitors, and History. The main workspace shows a collection named "03-Task-Manager". Under this collection, there are four items: "01-Random-API", "02-Express-Tutorial", "03-Task-Manager", and "04-Task-Manager". The "03-Task-Manager" item is expanded, showing three sub-items: "GET Get All Tasks", "POST Create Task", and "GET Get Single Task". Below these, there is a "PATCH Update Task" entry and a "DELETE Delete Task" entry. The "DELETE Delete Task" entry is selected. The request details show a "DELETE" method with the URL `((URL))/tasks/62c8de96ef8f7dc52367e422`. The "Params" tab is active, showing a table with one row:

KEY	VALUE	DESCRIPTION	Bulk Edit
Key	Value	Description	

The "Body" tab is also visible. The response pane at the bottom shows a status of 404 Not Found with a timestamp of 65 ms and a size of 293 B. The response content is a JSON object:

```
1 "msg": "No task with ID: 62c8de96ef8f7dc52367e422"
```

In Postman, deleting a task with ID not available in database and doesn't have equal ID characters with database.

Response Status: 500 – Cast Error

The screenshot shows the Postman application interface. The left sidebar has sections for Scratch Pad, Collections, APIs, Environments, Mock Servers, Monitors, and History. The main area shows a 'DELETE' request to `((URL))/tasks/62c8de96ef8f7dc52367e422jjhgjhgjh`. The Headers tab shows a single header 'Content-Type: application/json'. The Body tab is selected and contains the following JSON:

```
3 |     "stringValue": "\"62c8de96ef8f7dc52367e422jjhgjhgjh\"",
4 |     "valueType": "string",
5 |     "kind": "ObjectId",
6 |     "value": "62c8de96ef8f7dc52367e422jjhgjhgjh",
7 |     "path": "_id",
8 |     "reason": {},
9 |     "name": "CastError",
10 |    "message": "Cast to ObjectId failed for value \"62c8de96ef8f7dc52367e422jjhgjhgjh\" (type
11 |      string) at path \"_id\" for model \"Tasks\""
12 | }
```

The response section shows a 500 Internal Server Error with 13 ms and 586 B. The response body is empty. The bottom right corner shows 'Runner' and 'Trash' buttons.

Update Task

Updating a task with certain data and ID

tasks.js in controllers folder

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure under "OPEN EDITORS". The current file is "tasks.js" located in the "controllers" folder of the "03-task-manager" project.
- Code Editor:** Displays the content of the "tasks.js" file. The code defines three asynchronous functions: `updateTask`, `deleteTask`, and `getTask`. It uses the `Task` model to interact with a database.
- Terminal:** At the bottom, the terminal shows the output of running the application: "Connection to DB Successful...." and "Server is listening on Port 3000....".

```
File Edit Selection View Go Run Terminal Help tasks.js - NodeJS-ExpressJS - Visual Studio Code

EXPLORER
OPEN EDITORS
X tasks.js 03-task-manager.js U
NODEJS-EXPRESS
> 01-node-tutorial
> 02-express-tutorial
> 03-task-manager
  controllers
    tasks.js U
  db
    connection.js U
  models U
  node_modules U
  public U
  routes
    tasks.js U
    .env U
    .gitignore U
    app.js U
    package-lock.json U
    package.json U
  Express-JS-Notes.pdf
  Node-JS-Notes.pdf
  README.md

tasks.js U x
03-task-manager > controllers > tasks.js > updateTask
26   if (!task) {
27     return res.status(404).json({ msg: 'No task with ID: ${taskID}' });
28   }
29   res.status(200).json({ task });
30 } catch (error) {
31   res.status(500).json({ msg: error });
32 }
33 }

const updateTask = async (req, res) => {
35   const { id: taskID } = req.params;
36   try {
37     const task = await Task.findByIdAndUpdate({ _id: taskID }, req.body);
38     if (!task) {
39       return res.status(404).json({ msg: 'No task with ID: ${taskID}' });
40     }
41     res.status(200).json({ task });
42   } catch (error) {
43     res.status(500).json({ msg: error });
44   }
45 }

const deleteTask = async (req, res) => {
46   const { id: taskID } = req.params;
47   try {
48     const task = await Task.findByIdAndDelete({ _id: taskID });
49     if (!task) {
50       return res.status(404).json({ msg: 'No task with ID: ${taskID}' });
51     }
52     res.status(200).json({ task });
53   } catch (error) {
54     res.status(500).json({ msg: error });
55   }
56 }

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
Connection to DB Successful.... Server is listening on Port 3000....
```

In Postman, we are going to edit the first task with ID 62c8db52ef8f7dc52367e41f

The screenshot shows the Postman application interface. The left sidebar contains navigation links: Home, Workspaces (with a dropdown arrow), Explore, Scratch Pad, Collections, APIs, Environments, Mock Servers, Monitors, and History. The main workspace is titled "Working locally in Scratch Pad. Switch to a Workspace". It displays a "GET Get All Tasks" request for the URL "((URL))/tasks". The "Params" tab shows a "Query Params" table with one entry: KEY "name" and VALUE "first task". The "Body" tab shows a JSON response:

```
1 {  
2   "tasks": [  
3     {  
4       "_id": "62c8db52ef8f7dc52367e41",  
5       "name": "first task",  
6       "completed": true,  
7       "__v": 0  
8     },  
9     {  
10       "completed": false,  
11       "name": "second task"  
12     }  
13   ]  
14 }
```

The status bar at the bottom includes icons for Find and Replace, Console, Runner, Trash, and a settings gear.

In Postman, updating the first task

Here we are still receiving the last item but not the latest item. Let's check that with the help of the Get All Tasks route.

The screenshot shows the Postman application interface. The top navigation bar includes 'Postman' logo, 'File', 'Edit', 'View', 'Help', 'Home', 'Workspaces', 'Explore', a search bar 'Search Postman', and 'Sign In / Create Account' buttons. A yellow banner at the top says 'Working locally in Scratch Pad. Switch to a Workspace'. The left sidebar has sections for 'Collections' (with '01-Random-API', '02-Express-Tutorial', and '03-Task-Manager'), 'APIs' (with 'Create Task', 'Get Single Task', 'Update Task', and 'Delete Task'), 'Environments', 'Mock Servers', 'Monitors', and 'History'. The main area is titled 'Scratch Pad' with 'New' and 'Import' buttons. It shows a 'GET Get All Tasks' request under '03-Task-Manager / Get All Tasks'. The request URL is 'GET {{(URL)}/tasks}'. The 'Params' tab is selected, showing 'Query Params' with 'KEY' and 'VALUE' columns. The 'Body' tab is selected, showing a JSON response:

```
1 {
2   "tasks": [
3     {
4       "_id": "62c8db52ef8f7dc52367e41f",
5       "name": "testing update func",
6       "completed": false,
7       "__v": 0
8     },
9     {
10       "completed": false,
11       "__v": 0
12     }
13   ]
14 }
```

The response status is '200 OK' with '53 ms' and '590 B' size. There are 'Save Response' and 'Bulk Edit' buttons. The bottom navigation bar includes 'Find and Replace', 'Console', 'Runner', and 'Trash' buttons.

In Postman, sending the empty data through UPDATE

The screenshot shows the Postman application interface. On the left, the sidebar lists collections like '01-Random-API' and '02-Express-Tutorial'. Under '03-Task-Manager', there are four items: 'Get All Tasks', 'Create Task', 'Get Single Task', 'PATCH Update Task', and 'Delete Task'. The 'PATCH Update Task' item is selected. In the main workspace, a PATCH request is configured with the URL `((URL))/tasks/62c8db52ef8f7dc52367e41f`. The 'Body' tab is selected, showing the following JSON payload:

```
1
2   ...
3   ...
4     "name": "",
5     "completed": "false"
```

Below the body, the response status is shown as 200 OK with 64 ms and 333 B. The response body is also displayed in JSON format:

```
1
2   {
3     "task": {
4       "_id": "62c8db52ef8f7dc52367e41f",
5       "name": "testing update func",
6       "completed": false,
7       "__v": 0
8     }
9   }
```

Still, we are receiving the last item instead of the latest item. Also, validators are not working.

In Postman, Get All Tasks route

The screenshot shows the Postman application interface. The sidebar lists the same collections and items as the previous screenshot. The 'Get All Tasks' item under '03-Task-Manager' is selected. In the main workspace, a GET request is configured with the URL `((URL))/tasks`. The 'Params' tab is selected, showing a table with one row:

KEY	VALUE	DESCRIPTION	Bulk Edit

Below the params, the response status is shown as 200 OK with 65 ms and 571 B. The response body is displayed in JSON format:

```
1
2   {
3     "tasks": [
4       {
5         "_id": "62c8db52ef8f7dc52367e41f",
6         "name": "",
7         "completed": false,
8         "__v": 0
9       },
10      {
11        "completed": false,
12        ...
13      }
14    ]
15  }
```

With the current setup, we are facing two issues,

1. Whenever we are updating, we are receiving the old item data instead of the updated item data.
 2. Validators are not working.

We can fix those issues by using the `options` object in the `findByIdAndUpdate` method.

tasks.js in controllers folder

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer (Left):** Shows the project structure for "03-task-manager". It includes files like connection.js, models, routes/tasks.js, and various JSON and PDF documents.
- Code Editor (Center):** Displays the content of the tasks.js file under the controllers folder. The code handles task creation, update, and deletion.
- Bottom Status Bar:** Shows the status bar with tabs for PROBLEMS, OUTPUT, TERMINAL, and DEBUG CONSOLE, along with the current file path (tasks.js - NodeJS-ExpressJS - Visual Studio Code).
- Bottom Taskbar:** Includes icons for powershell, terminal, file operations, and browser links.

In Postman, the latest item is getting returned as the response.

The screenshot shows the Postman application interface. The left sidebar contains collections, APIs, environments, mock servers, monitors, and history. The main area displays a Scratch Pad workspace titled "03-Task-Manager / Update Task". A PATCH request is being sent to the URL `((URL))/tasks/62c8db52ef8f7dc52367e41f`. The request body is set to "raw" and contains the following JSON:

```
1 {  
2   ... "name": "testing with patch",  
3   "completed": true  
4 }
```

The response status is 200 OK, with a duration of 163 ms and a size of 331 B. The response body is also displayed as JSON:

```
1 {  
2   "task": {  
3     "_id": "62c8db52ef8f7dc52367e41f",  
4     "name": "testing with patch",  
5     "completed": true,  
6     "__v": 0  
7   }  
8 }
```

In Postman, Validation errors are sent back when empty name field is sent in the request.

The screenshot shows the Postman interface. On the left, the 'Scratch Pad' sidebar lists collections like '01-Random-API', '02-Express-Tutorial', and '03-Task-Manager'. The '03-Task-Manager' collection is expanded, showing 'Get All Tasks', 'Create Task', 'Get Single Task', 'Update Task' (which is selected), and 'Delete Task'. The main workspace shows a 'PATCH' request to `((URL))/tasks/62c8db52ef8f7dc52367e41f`. The 'Body' tab is selected, showing the JSON payload:

```
1 ... "name": "",  
2 ... "completed": "false"
```

The 'Test Results' tab shows a 500 Internal Server Error response with the following JSON content:

```
5   "name": "ValidatorError",  
6   "message": "must provide task name",  
7   "properties": {  
8     "message": "must provide task name",  
9     "type": "required",  
10    "path": "name",  
11    "value": ""
```

Instead of using the frontend code provided by the instructor. I have developed the React App which performs the CRUD Operations using the Backend application that we prepared using Node and Express.

Until now, our core functionality is completed but now let's work on improving the backend code.

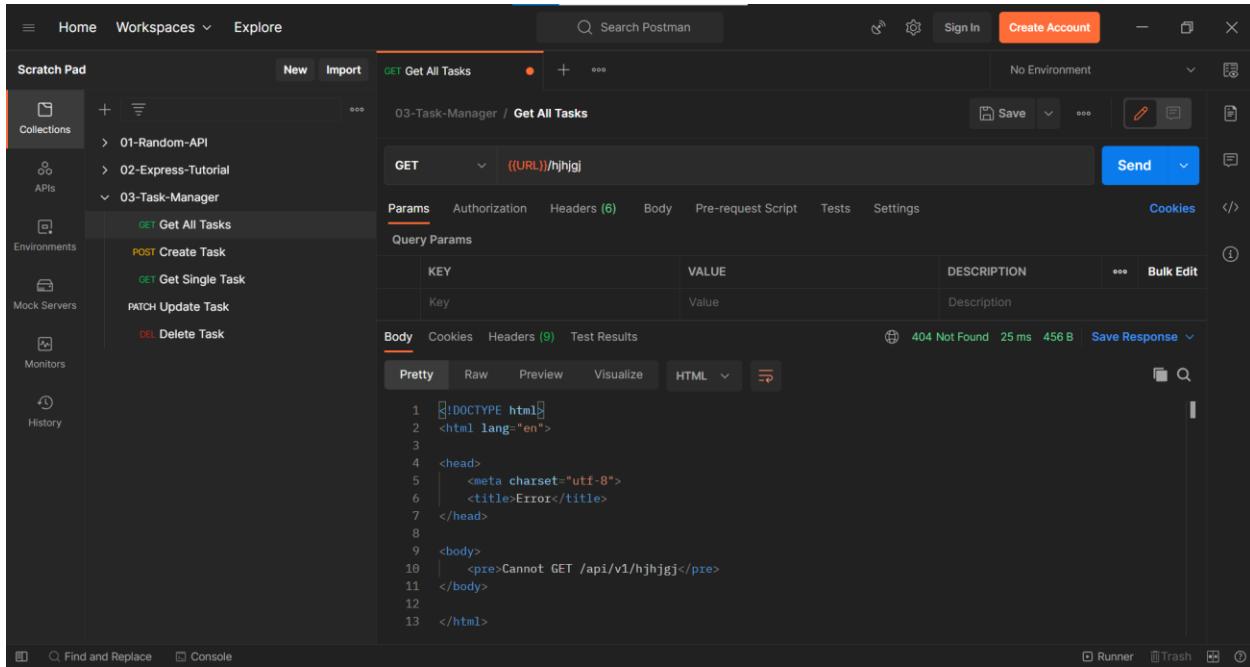
Now let's discuss about **PUT** method vs **PATCH** method

PUT and **PATCH** are both used for updating the resource, but when we are using **PUT** method, we are trying to replace the existing resource (sometimes we may forget to send parameters in request body and those parameters may be removed since we are using **PUT**) and **PATCH** (In PATCH Method, if we don't send the parameter in the request body it won't be replaced and will stay in the data) method is for partial update.

It depends on the requirement to choose between **PUT** and **PATCH**. For this Project we are using **PATCH** method.

Error Route

When a user enters a route that isn't present in the controllers then we receive an 404 Status Code with an HTML Body.

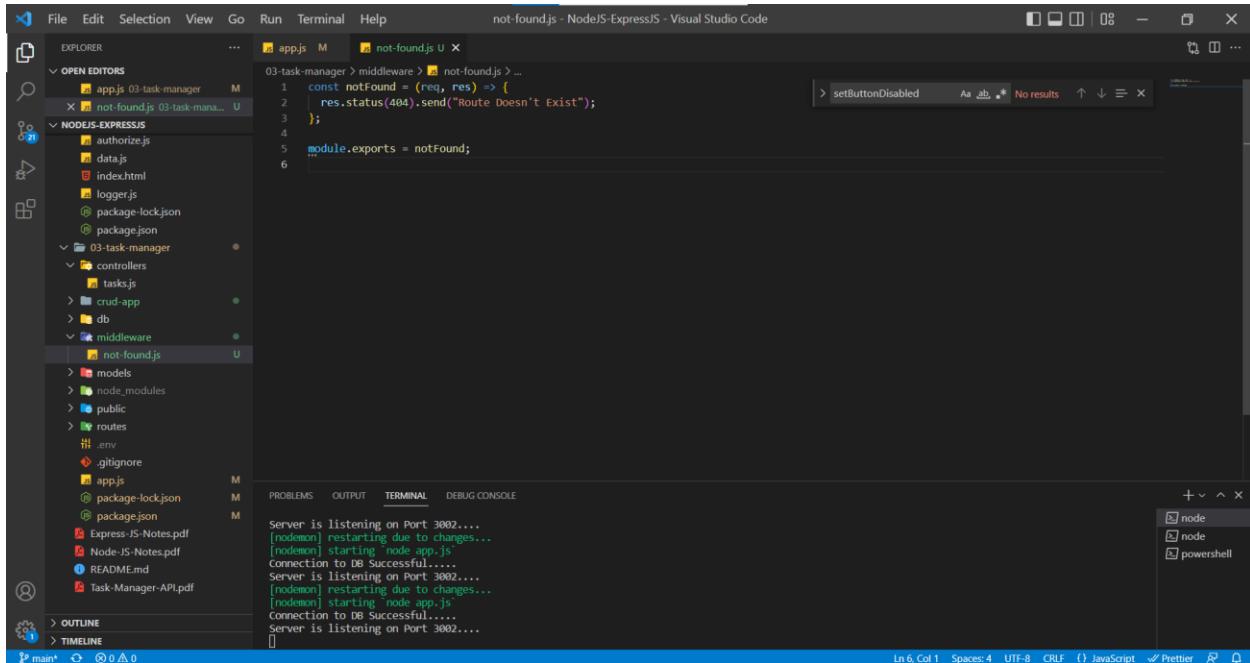


The screenshot shows the Postman interface. In the left sidebar, there's a 'Collections' section with '01-Random-API', '02-Express-Tutorial', and '03-Task-Manager'. Under '03-Task-Manager', there are four items: 'GET Create Task', 'GET Get Single Task', 'PATCH Update Task', and 'DELETE Delete Task'. The 'GET Create Task' item is selected. The main panel shows a GET request to '((URL))/hjhjgj'. The 'Params' tab is active, showing a single parameter 'Key' with value 'Value'. Below it, the 'Body' tab shows a JSON object with 'Description' set to 'Description'. The 'Test Results' tab shows a 404 Not Found response with a status of 25 ms and a size of 456 B. The response body is a simple HTML page:

```
1 <!DOCTYPE html>
2 <html lang="en">
3
4 <head>
5   <meta charset="utf-8">
6   <title>Error</title>
7 </head>
8
9 <body>
10  <pre>Cannot GET /api/v1/hjhjgj</pre>
11 </body>
12
13 </html>
```

We can solve the above problem by handling the unhandled API routes. We will create a middle function which we will used whenever a user enters with an unhandled route.

not-found.js in Middleware folder

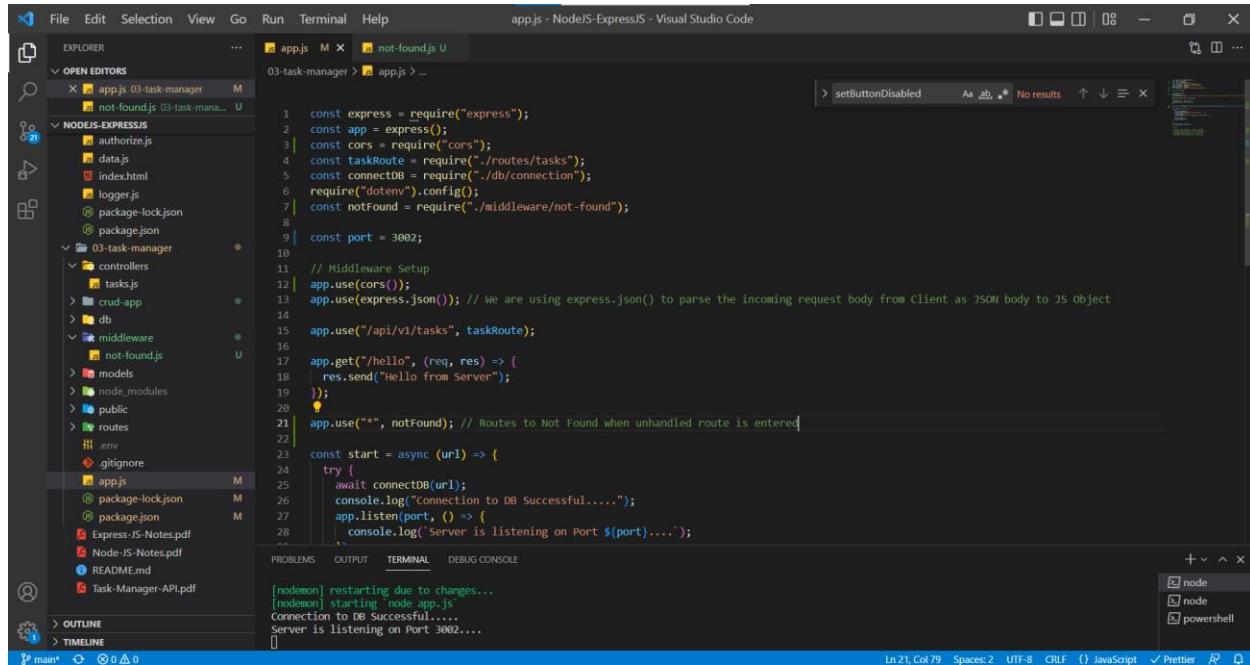


The screenshot shows the Visual Studio Code interface. The left sidebar shows a project structure with '03-task-manager' as the active workspace. Inside '03-task-manager', there are 'controllers', 'crud-app', 'db', 'models', 'node_modules', 'public', 'routes', '.env', '.gitignore', 'app.js', 'package-lock.json', and 'package.json'. A 'NODEJS-EXPRESS' folder contains 'authorize.js', 'data.js', 'index.html', 'logger.js', 'package-lock.json', and 'package.json'. The 'middleware' folder contains 'not-found.js'. The code editor shows the 'not-found.js' file with the following content:

```
1 const notFound = (req, res) => {
2   res.status(404).send("Route Doesn't Exist");
3 };
4
5 module.exports = notFound;
```

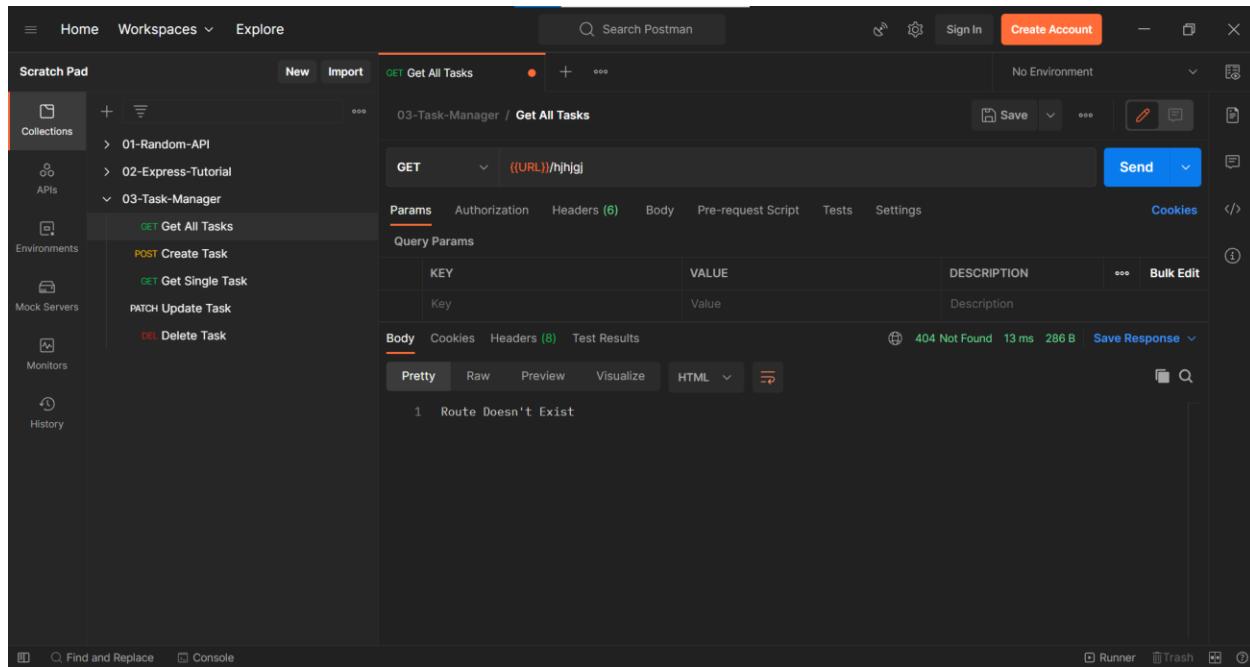
The terminal at the bottom shows the output of the application running on port 3002, indicating successful connection to the database and the server listening on port 3002.

app.js



```
const express = require("express");
const app = express();
const cors = require("cors");
const taskRoute = require("./routes/tasks");
const connectDB = require("./db/connection");
require("dotenv").config();
const notFound = require("./middleware/not-found");
const port = 3002;
// Middleware Setup
app.use(cors());
app.use(express.json()); // We are using express.json() to parse the incoming request body from client as JSON body to JS object
app.use("/api/v1/tasks", taskRoute);
app.get("/hello", (req, res) => {
  res.send("Hello from Server");
});
app.use("*", notFound); // Routes to Not Found when unhandled route is entered
const start = async (url) => {
  try {
    await connectDB(url);
    console.log("Connection to DB Successful....");
    app.listen(port, () => {
      console.log(`Server is listening on Port ${port}....`);
    });
  } catch (err) {
    console.log(err);
  }
};
start("mongodb://localhost:27017/taskmanager");
```

In Postman, when an unhandled route is entered we are sending the Response with Status Code 404



Scratch Pad

03-Task-Manager / Get All Tasks

GET ((URL))/hjhjg

Params

KEY	VALUE	DESCRIPTION	Bulk Edit
Key	Value	Description	

Body

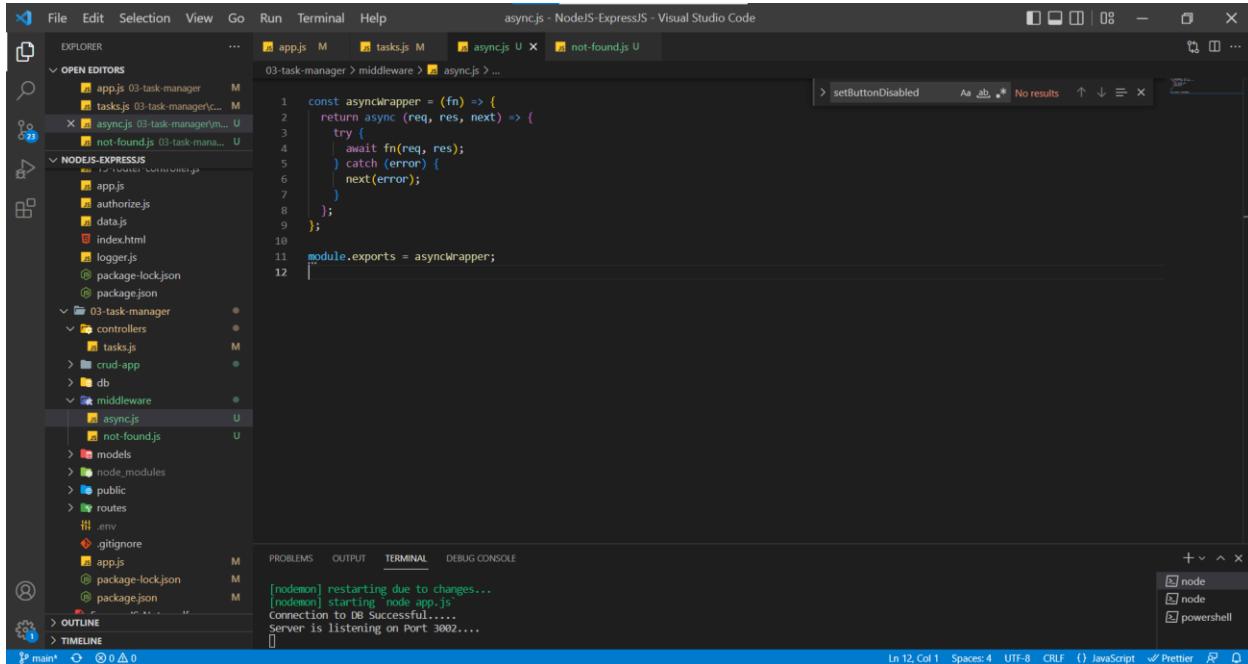
1 Route Doesn't Exist

Removing the Redundant Code

As we can see there is a lot of repetitive code in the Controllers folder where we are using try and catch blocks to handle the asynchronous functions. To remove the redundant code, we can create a middleware function which will take care of handling the asynchronous functions.

Let us create a middleware function named `asyncWrapper` which is used to wrap the asynchronous functions in our code.

`async.js` in Middleware folder

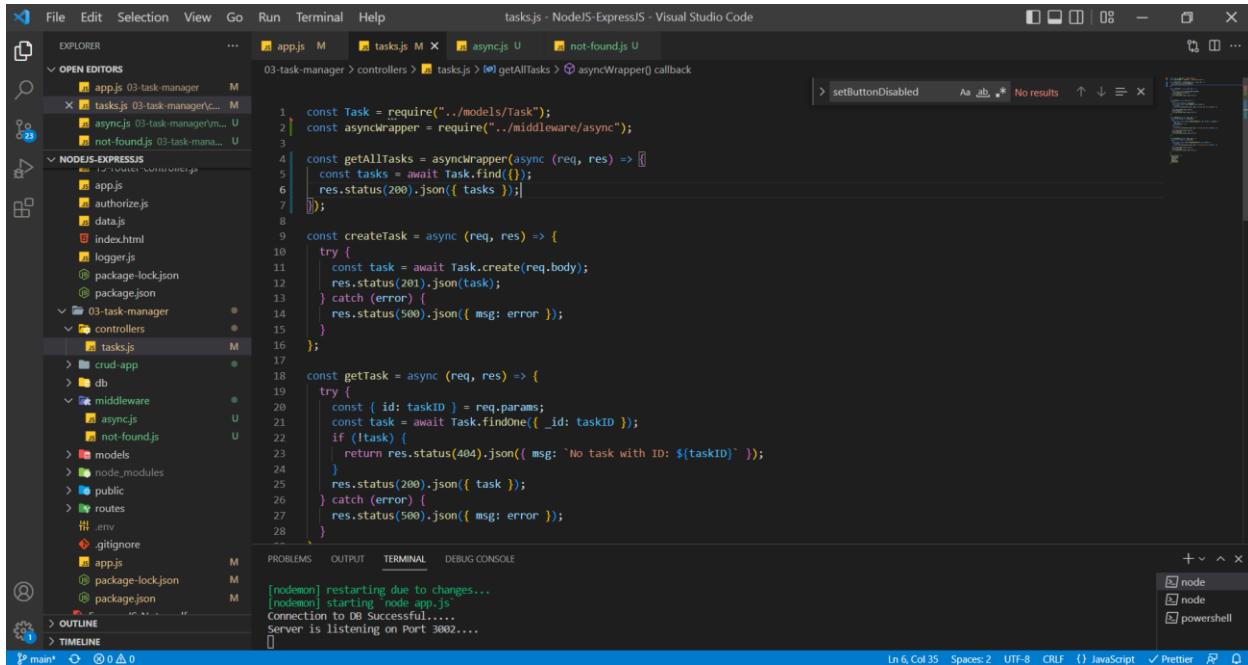


```
const asyncWrapper = (fn) => {
  return async (req, res, next) => {
    try {
      await fn(req, res);
    } catch (error) {
      next(error);
    }
  };
};

module.exports = asyncWrapper;
```

The screenshot shows the Visual Studio Code interface with the `async.js` file open in the editor. The file content is displayed above, showing the implementation of the `asyncWrapper` middleware function. The code uses a try-catch block to handle errors and calls `next(error)` if an error occurs. The `module.exports` statement exports the `asyncWrapper` function. The terminal at the bottom shows the application starting up with nodemon and connecting to the database.

Using AsyncWrapper in Tasks.js



The screenshot shows the Visual Studio Code interface with the tasks.js file open in the editor. The code uses the AsyncWrapper module to handle asynchronous operations like database queries.

```
const Task = require("../models/Task");
const asyncWrapper = require("../middleware/async");

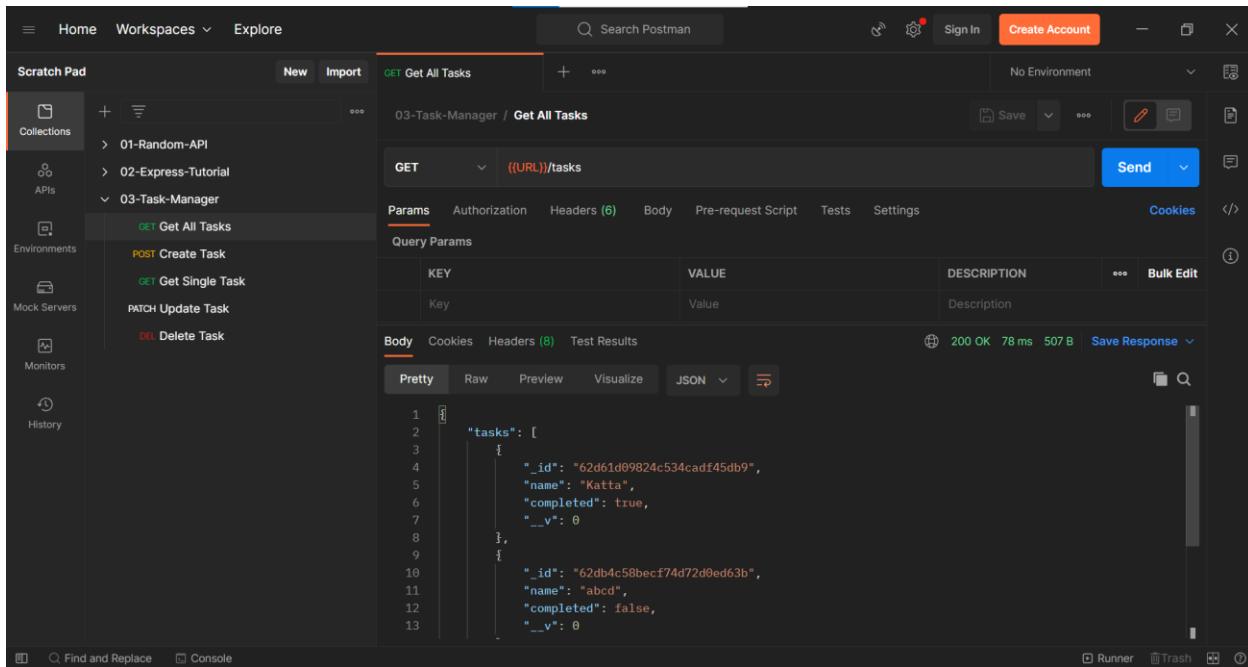
const getAllTasks = asyncWrapper(async (req, res) => {
  const tasks = await Task.find();
  res.status(200).json({ tasks });
});

const createTask = async (req, res) => {
  try {
    const task = await Task.create(req.body);
    res.status(201).json(task);
  } catch (error) {
    res.status(500).json({ msg: error });
  }
};

const getTask = async (req, res) => {
  try {
    const { id: taskID } = req.params;
    const task = await Task.findOne({ _id: taskID });
    if (!task) {
      return res.status(404).json({ msg: `No task with ID: ${taskID}` });
    }
    res.status(200).json(task);
  } catch (error) {
    res.status(500).json({ msg: error });
  }
};
```

VS Code status bar: Ln 6, Col 35 · Spaces: 2 · UTF-8 · CR LF · {} JavaScript · ✓ Prettier

Getting All tasks from Postman

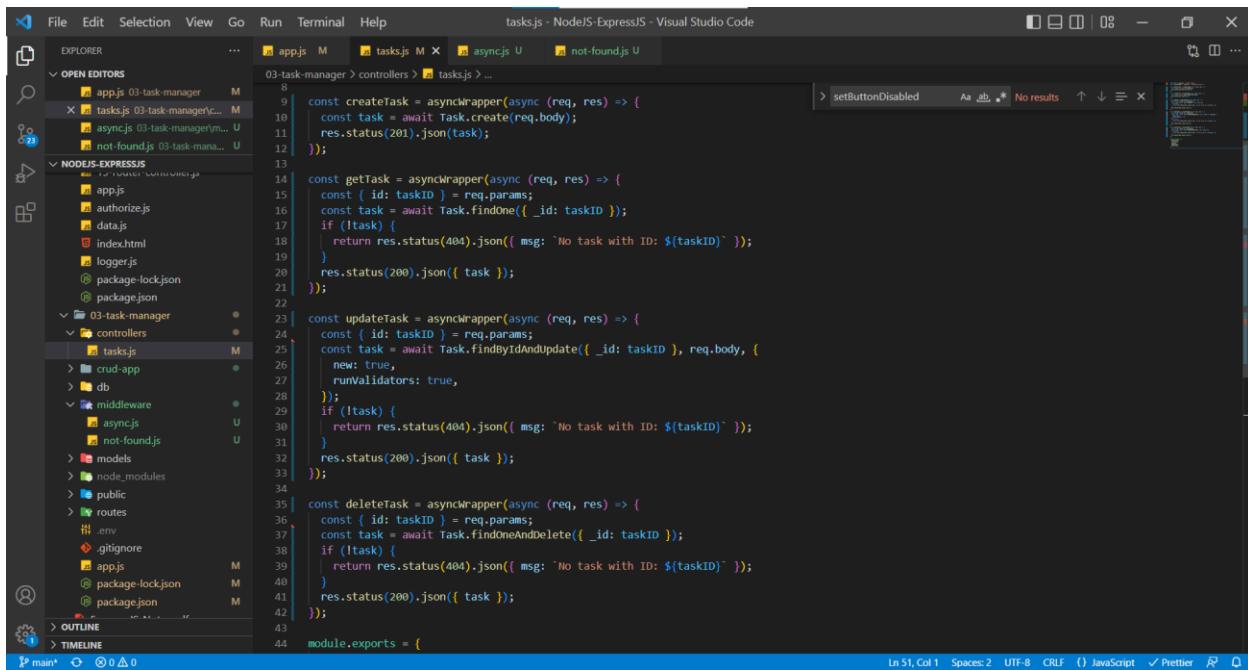


The screenshot shows the Postman application interface. A GET request is being prepared to the endpoint /tasks. The response body shows a JSON array of tasks.

KEY	VALUE	DESCRIPTION	Bulk Edit
Key	Value	Description	

```
1 "tasks": [
2   {
3     "_id": "62d61d09824c534cadff45db9",
4     "name": "Katta",
5     "completed": true,
6     "__v": 0
7   },
8   {
9     "_id": "62db4c58becf74d72d0ed63b",
10    "name": "abcd",
11    "completed": false,
12    "__v": 0
13 }
```

Now let's apply the AsyncWrapper to all the API Routes



```
const createTask = asyncWrapper(async (req, res) => {
  const task = await Task.create(req.body);
  res.status(201).json(task);
});

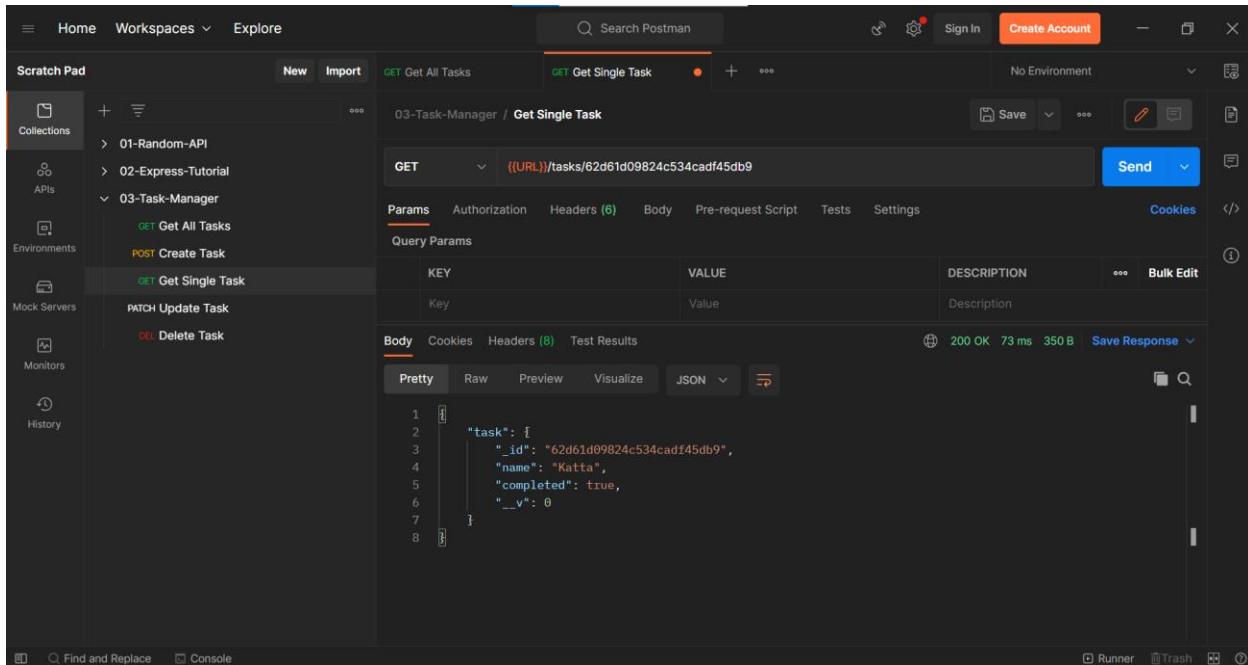
const getTask = asyncWrapper(async (req, res) => {
  const { id: taskID } = req.params;
  const task = await Task.findById({ _id: taskID });
  if (!task) {
    return res.status(404).json({ msg: `No task with ID: ${taskID}` });
  }
  res.status(200).json({ task });
});

const updateTask = asyncWrapper(async (req, res) => {
  const { id: taskID } = req.params;
  const task = await Task.findByIdAndUpdate({ _id: taskID }, req.body, {
    new: true,
    runValidators: true,
  });
  if (!task) {
    return res.status(404).json({ msg: `No task with ID: ${taskID}` });
  }
  res.status(200).json({ task });
});

const deleteTask = asyncWrapper(async (req, res) => {
  const { id: taskID } = req.params;
  const task = await Task.findByIdAndDelete({ _id: taskID });
  if (!task) {
    return res.status(404).json({ msg: `No task with ID: ${taskID}` });
  }
  res.status(200).json({ task });
});

module.exports = {
```

Get Single Task



The screenshot shows the Postman interface with the following details:

- Collection:** 03-Task-Manager
- Request Type:** GET
- URL:** {{URL}}/tasks/62d61d09824c534cadf45db9
- Params:** (None)
- Headers:** (6)
- Body:** (Raw, JSON)
{"task": {
 "_id": "62d61d09824c534cadf45db9",
 "name": "Katta",
 "completed": true,
 "__v": 0
}}
- Tests:** (None)
- Settings:** (None)

Create Task

The screenshot shows the Postman application interface. In the top navigation bar, 'Home' and 'Workspaces' are visible. The main workspace is titled 'Scratch Pad'. A collection named '03-Task-Manager' is selected. Inside the collection, there are four items: 'Get All Tasks' (GET), 'Create Task' (POST), 'Get Single Task' (GET), and 'Delete Task' (DELETE). The 'Create Task' item is currently selected. The request details panel shows a POST method with the URL `((URL))/tasks`. The 'Body' tab is selected, showing a JSON payload:

```
1 "name": "New Task",
2 "completed": true
```

The response panel shows a 201 Created status with 109 ms and 349 B. The response body is:

```
1 "name": "New Task",
2 "completed": true,
3 "_id": "62dd89208a08a222b84e9173",
4 "__v": 0
```

Updating Task

The screenshot shows the Postman application interface. The workspace and collection structure are identical to the previous screenshot. The 'Update Task' item under the 'Create Task' POST item is selected. The request details panel shows a PATCH method with the URL `((URL))/tasks/62d61d09824c534cadf45db9`. The 'Body' tab is selected, showing a JSON payload:

```
1 "name": "testing task",
2 "completed": true
```

The response panel shows a 200 OK status with 62 ms and 357 B. The response body is:

```
1 "task": {
2     "_id": "62d61d09824c534cadf45db9",
3     "name": "testing task",
4     "completed": true,
5     "__v": 0
6 }
```

Deleting a Task

The screenshot shows the Postman interface. In the left sidebar, under the 'Collections' section, there is a '03-Task-Manager' collection. Inside it, there is a 'Delete Task' item. The main workspace shows a DELETE request to `((URL))/tasks/62d61d09824c534cadf45db9`. The 'Body' tab is selected, showing the JSON body of the request:

```
1
2   "task": {
3     "_id": "62d61d09824c534cadf45db9",
4     "name": "testing task",
5     "completed": true,
6     "__v": 0
7 }
```

The response status is 200 OK, with a duration of 62 ms and a size of 357 B. The response body is empty, indicating the task was successfully deleted.

Everything works as expected but here we are not handling the errors.

In the below scenario, we are trying to create a task, but we are sending a value for name property hence we should get an error but here we are receiving an HTML since we are not handling the error.

The screenshot shows the Postman interface. In the left sidebar, under the 'Collections' section, there is a '03-Task-Manager' collection. Inside it, there is a 'Create Task' item. The main workspace shows a POST request to `((URL))/tasks`. The 'Body' tab is selected, showing the JSON body of the request:

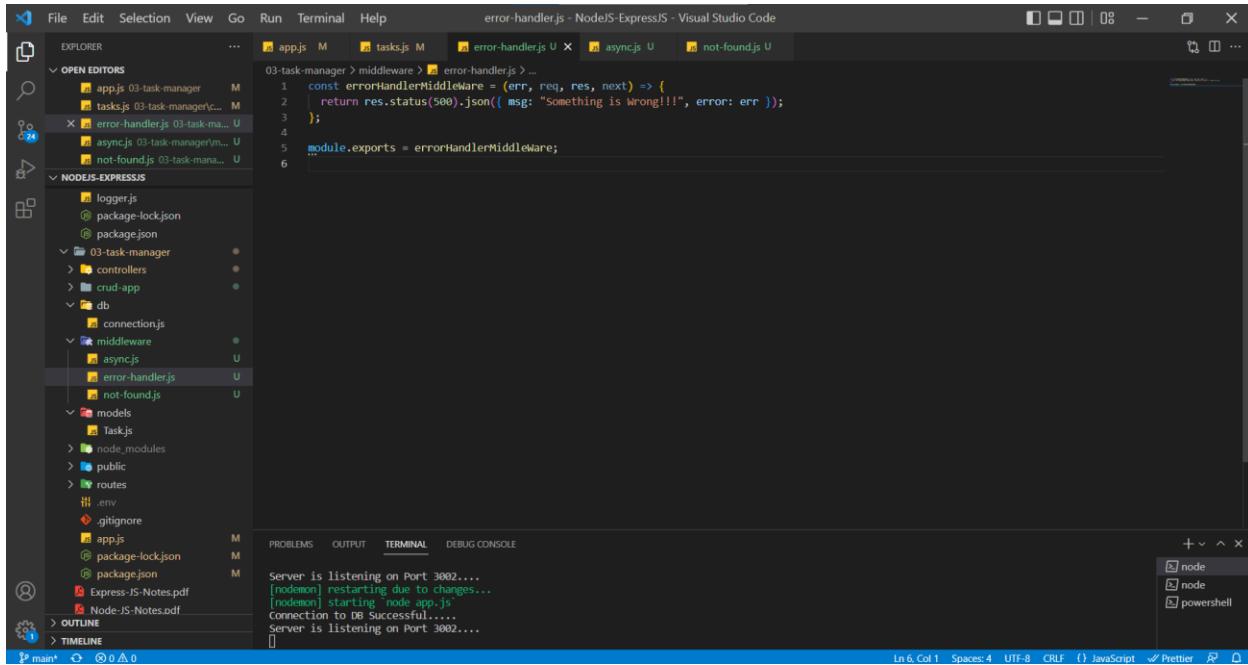
```
1
2   "name": "",
3   "completed": true
4 }
```

The response status is 500 Internal Server Error, with a duration of 14 ms and a size of 1000 B. The response body is an HTML error page:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>Error</title>
6 </head>
7 <body>
8
9 <pre>ValidationError: Tasks validation failed: name: must provide task name<br>  at model.Document.invalidate
10  (<file>:1:1)
```

We can handle the above scenario by writing an express default error handling middleware function.

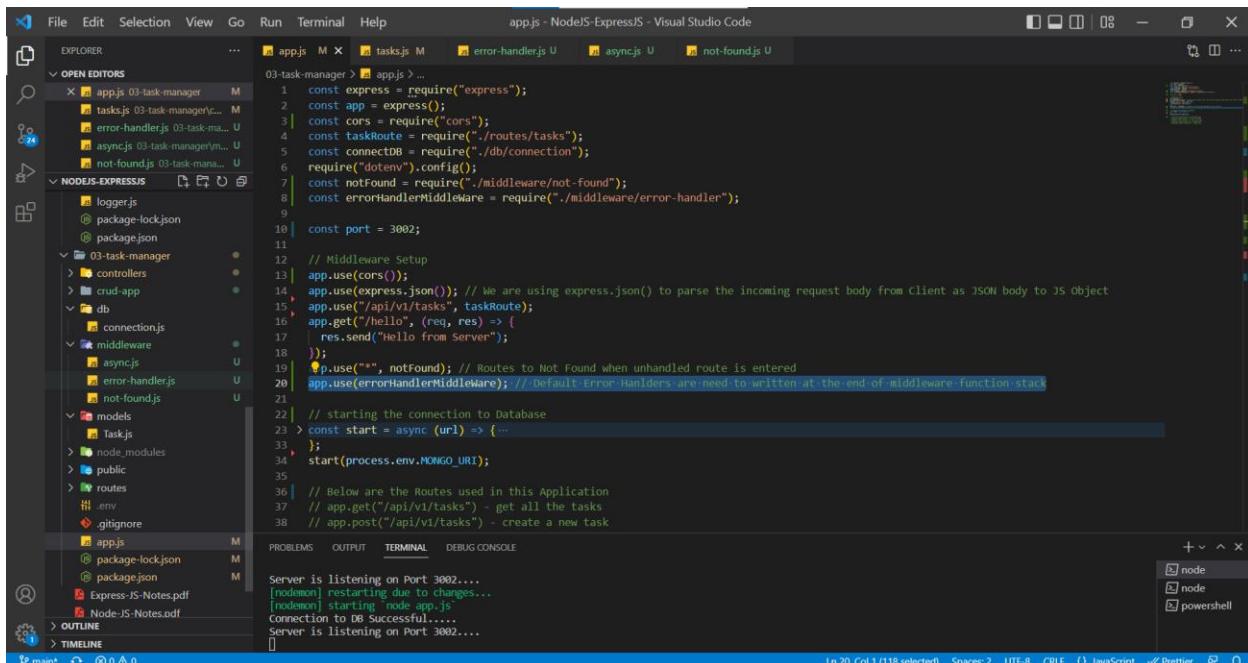
error-handler.js in Middleware folder



```
const errorHandlerMiddleware = (err, req, res, next) => {
  return res.status(500).json({msg: "Something is Wrong!!!", error: err});
};

module.exports = errorHandlerMiddleware;
```

Default Error Handler in express contains 4 parameters (err, req, res, next)



```
const express = require('express');
const app = express();
const cors = require('cors');
const taskroute = require('./routes/tasks');
const connectDB = require('./db/connection');
require('dotenv').config();
const notFound = require('../middleware/not-found');
const errorHandlerMiddleWare = require('../middleware/error-handler');

const port = 3002;

// Middleware Setup
app.use(cors());
app.use(express.json()); // We are using express.json() to parse the incoming request body from client as JSON body to JS object
app.use('/api/v1/tasks', taskroute);
app.get('/hello', (req, res) => {
  res.send("Hello from Server");
});
app.use('*', notFound); // Routes to Not Found when unhandled route is entered
app.use(errorHandlerMiddleWare); // Default Error Handlers are need to be written at the end of middleware function stack

// starting the connection to Database
const start = async (url) => {
  const MongoClient = require('mongodb').MongoClient;
  const client = new MongoClient(url, { useNewUrlParser: true, useUnifiedTopology: true });
  await client.connect();
  console.log(`Connected successfully to mongoDB at ${url}`);
};

// Below are the Routes used in this Application
app.get("/api/v1/tasks") - get all the tasks
app.post("/api/v1/tasks") - create a new task
```

In app.js we need to place error-handler middleware function at the bottom of the middleware functions stack.

In Postman, For Creating the Task

The screenshot shows the Postman application interface. On the left, there's a sidebar with 'Collections' (01-Random-API, 02-Express-Tutorial, 03-Task-Manager), 'APIs', 'Environments', 'Mock Servers', 'Monitors', and 'History'. The main area shows a 'Create Task' collection under '03-Task-Manager'. A POST request is being made to `((URL))/tasks`. The 'Body' tab is selected, showing the JSON payload:

```
1  ...
2  ...
3  ...
4  ...
5  ...
6  ...
7  ...
8  ...
9  ...
10 ...
11 ...
12 ...
```

The 'Body' tab also displays the raw JSON response received from the server:

```
1 "msg": "Something is Wrong!!!",
2 "error": {
3     "errors": [
4         {
5             "name": {
6                 "name": "ValidatorError",
7                 "message": "must provide task name",
8                 "properties": {
9                     "message": "must provide task name",
10                    "type": "required",
11                    "path": "name",
12                    "value": ""
13                }
14            }
15        }
16    ]
17 }
```

At the bottom, it shows a status of 500 Internal Server Error, 45 ms, 672 B, and a 'Save Response' button.

Since we handled the Default Errors in express, now it's the time to remove the redundant code used to handle Custom Errors in the project.

Creating a custom error file in errors folder

The screenshot shows the Visual Studio Code interface with the file 'custom-error.js' open. The code defines a new error class 'customError' that extends the built-in 'Error' class. It includes a constructor that takes a message and a statusCode, and sets the statusCode on the error object. A middleware function 'customErrorMiddleware' is defined to handle errors by creating a new 'customError' with the provided message and statusCode. Finally, the module exports both the middleware and the error class.

```
1 class customError extends Error {
2     constructor(message, statusCode) {
3         super(message);
4         this.statusCode = statusCode;
5     }
6 }
7
8 const customErrorMiddleware = (message, statusCode) => {
9     return new customError(message, statusCode);
10 };
11
12 module.exports = { customErrorMiddleware, customError };
```

Need to use the next() method to transfer the error the custom error file or object

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows files in the 03-task-manager directory, including app.js, custom-error.js, tasks.js, error-handler.js, and not-found.js.
- Code Editor:** Displays the content of async.js:

```
const asyncWrapper = (fn) => {
  return async (req, res, next) => {
    try {
      await fn(req, res, next);
    } catch (error) {
      next(error);
    }
  };
};

module.exports = asyncWrapper;
```
- Terminal:** Shows logs from nodemon: [nodemon] restarting due to changes... [nodemon] starting "node app.js" Connection to DB Successful.... Server is listening on Port 3002.... [nodemon] restarting due to changes... [nodemon] starting "node app.js" Connection to DB Successful.... Server is listening on Port 3002....
- Status Bar:** Shows Ln 9, Col 3 - Spaces: 4 - UTF-8 - CR/LF - JavaScript - Prettier.

Adding extra lines of code to handle the custom errors in the application

error-hanlder.js in middleware folder

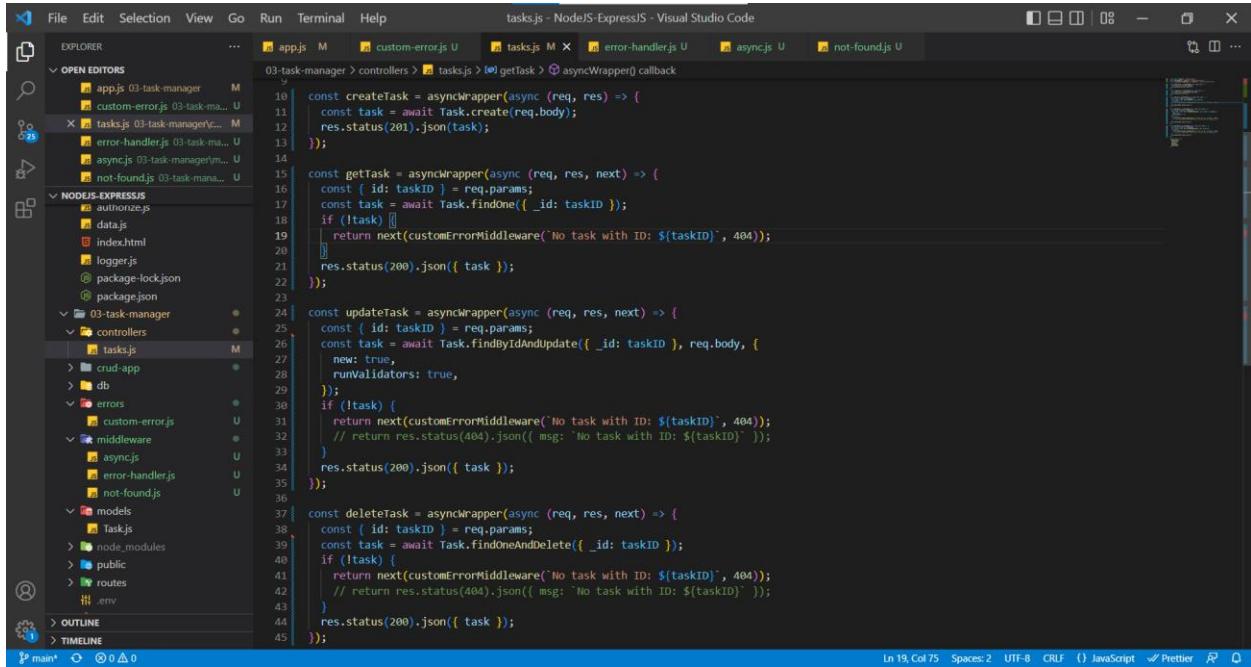
The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows files in the 03-task-manager directory, including app.js, custom-error.js, tasks.js, error-handler.js, and not-found.js.
- Code Editor:** Displays the content of error-handler.js:

```
const { customError } = require("../errors/custom-error");
const errorHandlerMiddleWare = (err, req, res, next) => {
  if (err instanceof customError) {
    return res.status(err.statusCode).json({ msg: err.message });
  }
  return res
    .status(500)
    .json({ msg: "Something went wrong, please try again!!!" });
};

module.exports = errorHandlerMiddleWare;
```
- Terminal:** Shows logs from nodemon: [nodemon] restarting due to changes... [nodemon] starting "node app.js" Connection to DB Successful.... Server is listening on Port 3002.... [nodemon] restarting due to changes... [nodemon] starting "node app.js" Connection to DB Successful.... Server is listening on Port 3002....
- Status Bar:** Shows Ln 11, Col 41 - Spaces: 4 - UTF-8 - CR/LF - JavaScript - Prettier.

Using the newly created custom error function in the app.js with help of next() method to instantiate the Error object and send it to the default handler function in error-handler.js file



```

File Edit Selection View Go Run Terminal Help
tasks.js - NodeJS-ExpressJS - Visual Studio Code
EXPLORER OPEN EDITORS NODEJS-EXPRESSJS
app.js M custom-error.js U tasks.js M error-handler.js U async.js U not-found.js U
03-task-manager controllers tasks.js M
  authorize.js M
  data.js M
  index.html M
  logger.js M
  package-lock.json M
  package.json M
  tasks.js M
    crud-app M
    db M
    errors M
      custom-error.js U
    middleware M
      async.js U
      error-handler.js U
      not-found.js U
    models M
      Task.js M
    node_modules M
    public M
    routes M
    .env M
OUTLINE TIMELINE
In 19, Col 75 Spaces: 2 UTF-8 CRLF () JavaScript ✓ Prettier R D
const createTask = asyncWrapper(async (req, res) => {
  const task = await Task.create(req.body);
  res.status(201).json(task);
});

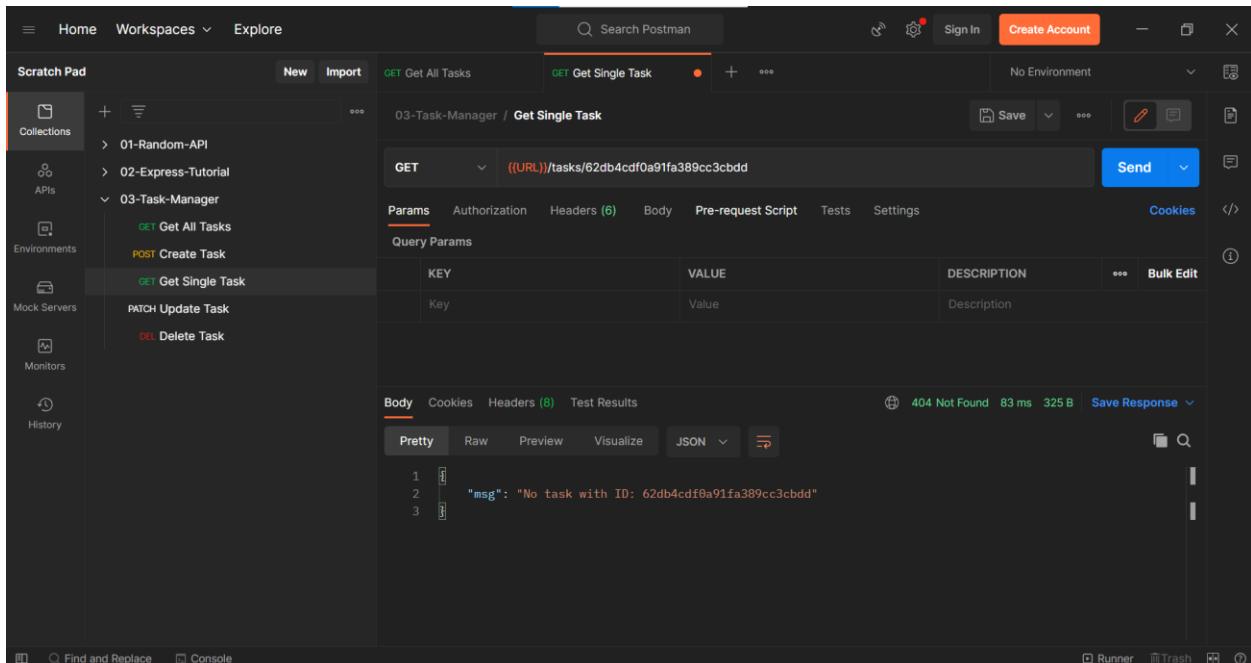
const getTask = asyncWrapper(async (req, res, next) => {
  const { id: taskID } = req.params;
  const task = await Task.findOne({ _id: taskID });
  if (!task) {
    return next(customErrorMiddleware(`No task with ID: ${taskID}`, 404));
  }
  res.status(200).json({ task });
});

const updateTask = asyncWrapper(async (req, res, next) => {
  const { id: taskID } = req.params;
  const task = await Task.findByIdAndUpdate({ _id: taskID }, req.body, {
    new: true,
    runValidators: true,
  });
  if (!task) {
    return next(customErrorMiddleware(`No task with ID: ${taskID}`, 404));
  }
  // return res.status(404).json({ msg: `No task with ID: ${taskID}` });
  res.status(200).json({ task });
});

const deleteTask = asyncWrapper(async (req, res, next) => {
  const { id: taskID } = req.params;
  const task = await Task.findByIdAndDelete({ _id: taskID });
  if (!task) {
    return next(customErrorMiddleware(`No task with ID: ${taskID}`, 404));
  }
  // return res.status(404).json({ msg: `No task with ID: ${taskID}` });
  res.status(200).json({ task });
});

```

In Postman, Get Single Task – when there is no task with ID



The screenshot shows the Postman interface with the following details:

- Scratch Pad:** A collection named "03-Task-Manager" is selected.
- Request:**
 - Method: GET
 - URL: `((URL))/tasks/62db4cdf0a91fa389cc3cbdd`
 - Params tab is active, showing "Query Params" with a single entry: "Key" (Value: "msg") and "Description" (Value: "Description").
 - Headers tab shows "(8)" entries.
 - Body tab is selected.
 - Response tab shows the following JSON output:
- Response Body:**

```

1   "msg": "No task with ID: 62db4cdf0a91fa389cc3cbdd"
2
3

```
- Response Headers:**
 - 404 Not Found
 - 83 ms
 - 325 B
 - Save Response

In Postman, Get Single Task – when Cast Error or any other generic error (Not a custom Error)

The screenshot shows the Postman interface. On the left, the sidebar has sections for Collections, APIs, Environments, Mock Servers, and Monitors. The main area shows a collection named '03-Task-Manager' with a 'Get Single Task' request. The request method is GET, and the URL is `((URL))/tasks/62db4cdf0a91fa389cc3cb`. The 'Params' tab is selected, showing a single query parameter 'Key' with value 'Value'. The 'Body' tab shows the response JSON:

```

1
2   "msg": "Something went Wrong, please try again!!!",
3   "err": {
4     "stringValue": "\\"62db4cdf0a91fa389cc3cb\\",
5     "valueType": "string",
6     "kind": "ObjectId",
7     "value": "62db4cdf0a91fa389cc3cb",
8     "path": "_id",
9     "reason": {},
10    "name": "CastError",
11    "message": "Cast to ObjectId failed for value \\\"62db4cdf0a91fa389cc3cb\\\" (type string) at path \\"_id\\" for model \\"Tasks\\""
12  }

```

The status bar at the bottom indicates a 500 Internal Server Error with 24 ms and 629 B.

Sometimes, applications require different Port Numbers, hence hardcoding the PORT number is not the right way. We need to follow the below format.

The screenshot shows the Visual Studio Code interface. The left sidebar shows a project structure with files like app.js, custom-error.js, tasks.js, error-handler.js, and not-found.js. The main editor window shows the app.js file with the following code:

```

1  const express = require("express");
2  const app = express();
3  const cors = require("cors");
4  const taskRoute = require("./routes/tasks");
5  const connectDB = require("./db/connection");
6  require("dotenv").config();
7  const notFound = require("./middleware/not-found");
8  const errorHandlerMiddleware = require("./middleware/error-handler");
9
10 const port = process.env.PORT || 3002; // PORT number depends upon the organization
11
12 // Middleware Setup
13 app.use(cors());
14 app.use(express.json()); // We are using express.json() to parse the incoming request body from client as JSON body to JS object
15 app.use("/api/v1/tasks", taskRoute);
16 app.get("/hello", (req, res) => {
17   res.send("Hello from server");
18 });
19 app.use("*", notFound); // Routes to Not Found when unhandled route is entered
20 app.use(errorHandlerMiddleware); // Default Error Handlers are need to written at the end of middleware function stack
21
22 // starting the connection to Database
23 const start = async (url) => {
24   await mongoose.connect(url);
25   console.log(`MongoDB Connected to ${url}`);
26 };
27
28 start(process.env.MONGO_URI);
29
30 // Below are the Routes used in this Application

```

The terminal at the bottom shows node.js logs:

```

[nodemon] restarting due to changes...
[nodemon] starting `node app.js`
Connection to DB Successful.....
Server is listening on Port 3002.....
[nodemon] restarting due to changes...
[nodemon] starting `node app.js`
Connection to DB Successful.....
Server is listening on Port 3002.....

```

Let's use a different PORT number to run our application