

# 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)

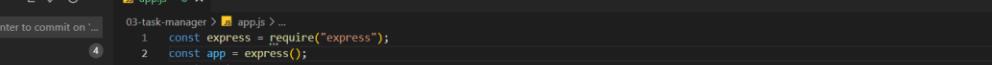
Fifth API Call – Deleting a task from the list. (DELETE with Request ID)

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



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

- File Explorer:** On the left, it shows a tree view of files and folders:
  - Message (Ctrl+Enter to commit on ...)
  - Changes (4)
  - .gitignore (03-task-manager)
  - app.js (03-task-manager) - The current file is open in the editor.
  - package-lock.json (03-task-manager)
  - package.json (03-task-manager)
- Editor:** The main area displays the content of app.js:

```
03-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
```
- Status Bar:** At the bottom right, it shows the status bar with icons for file operations and a message: "03-task-manager".

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:** Shows "Ln 15, Col 1" and other settings like "Spaces: 4", "UTF-8", "CRLF", "JavaScript", and "Prettier".

tasks.js in controllers folder

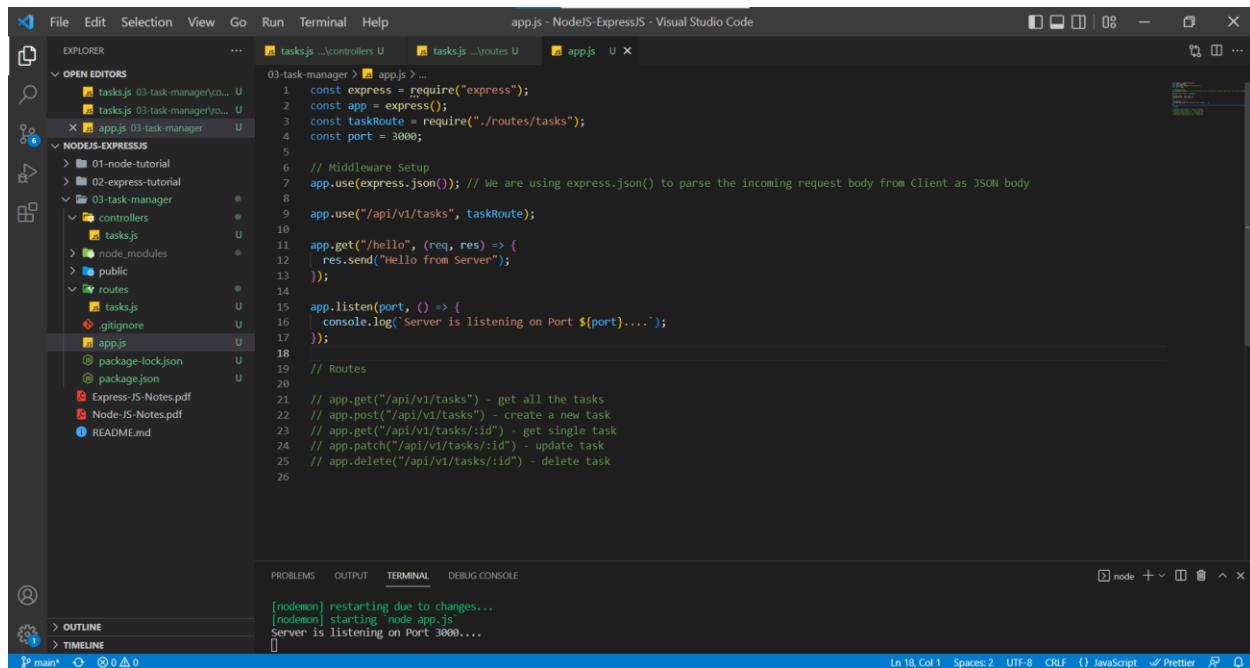
The screenshot shows the Visual Studio Code interface with the following details:

- 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:** Shows "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

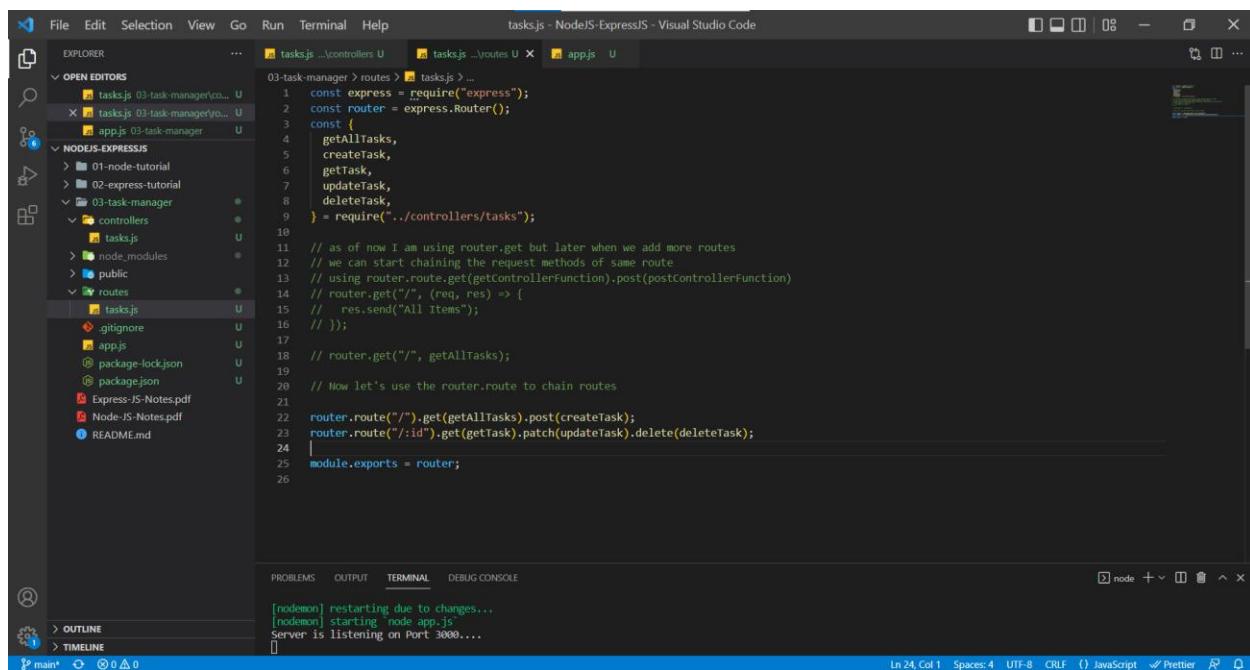
// app.get("/api/v1/tasks") - get all the tasks
// app.post("/api/v1/tasks") - create a new task
// app.get("/api/v1/tasks/:id") - get single task
// app.patch("/api/v1/tasks/:id") - update task
// app.delete("/api/v1/tasks/:id") - delete task

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 a "No active Environment" message and a table for "Globals".

VARIABLE	INITIAL VALUE	CURRENT VALUE
URL	http://localhost:3000/api/v1	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, there are buttons for Sign In and Create Account. The main workspace is titled "Working locally in Scratch Pad. Switch to a Workspace". The left sidebar has sections for Collections, APIs, Environments, Mock Servers, Monitors, and History. The "Scratch Pad" section is active, showing a collection named "03-Task-Manager" which contains a "Get All Tasks" endpoint. The endpoint details are as follows:

- Method: GET
- URL: {{URL}}/tasks
- Params tab (selected):
  - Query Params table:

KEY	VALUE	DESCRIPTION	Bulk Edit
Key	Value	Description	
- Authorization tab
- Headers tab (6 items)
- Body tab
- Pre-request Script tab
- Tests tab
- Settings tab

On the right side, there are tabs for Cookies, Body, Cookies, Headers (7), Test Results, and a status bar showing 200 OK, 6 ms, 257 B, and Save Response. Below the status bar, the response body is displayed as:

```
1 All Items from the Controller
```

## API Routes that have been setup

The screenshot shows a REST API documentation page. At the top, it says "REST API" and features 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.

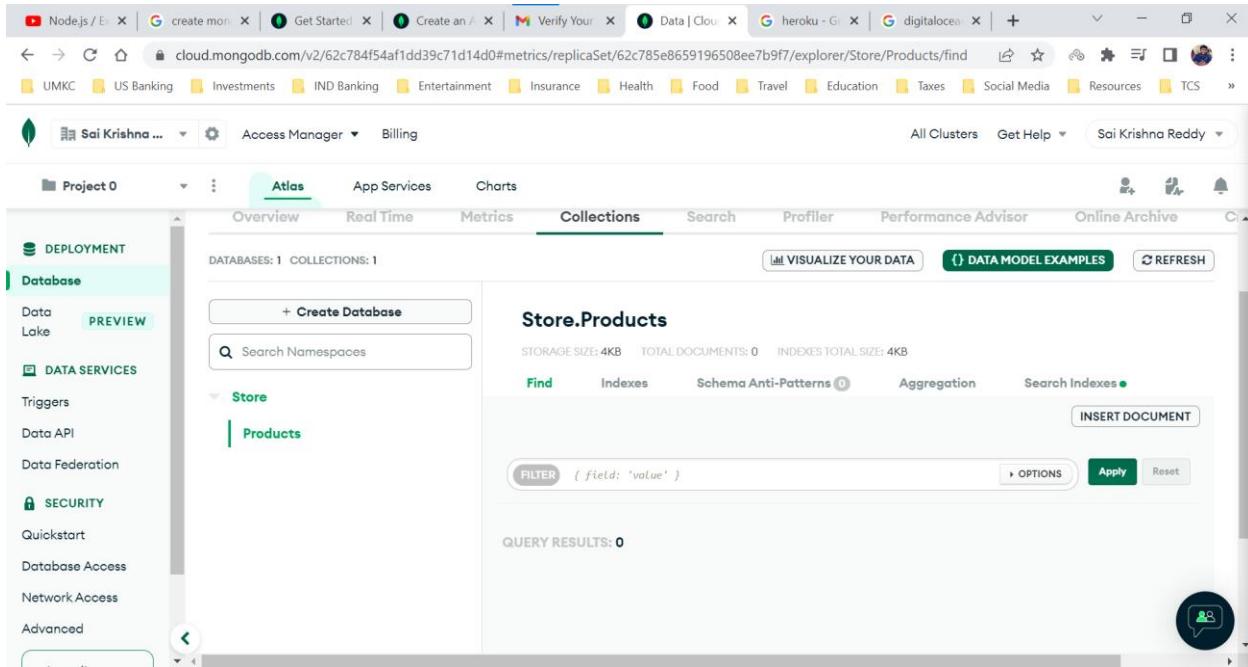
Click on Add My Own Data and below screen will appear.

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 interface as the previous screenshot, with the "Add My First Dataset" callout still visible.

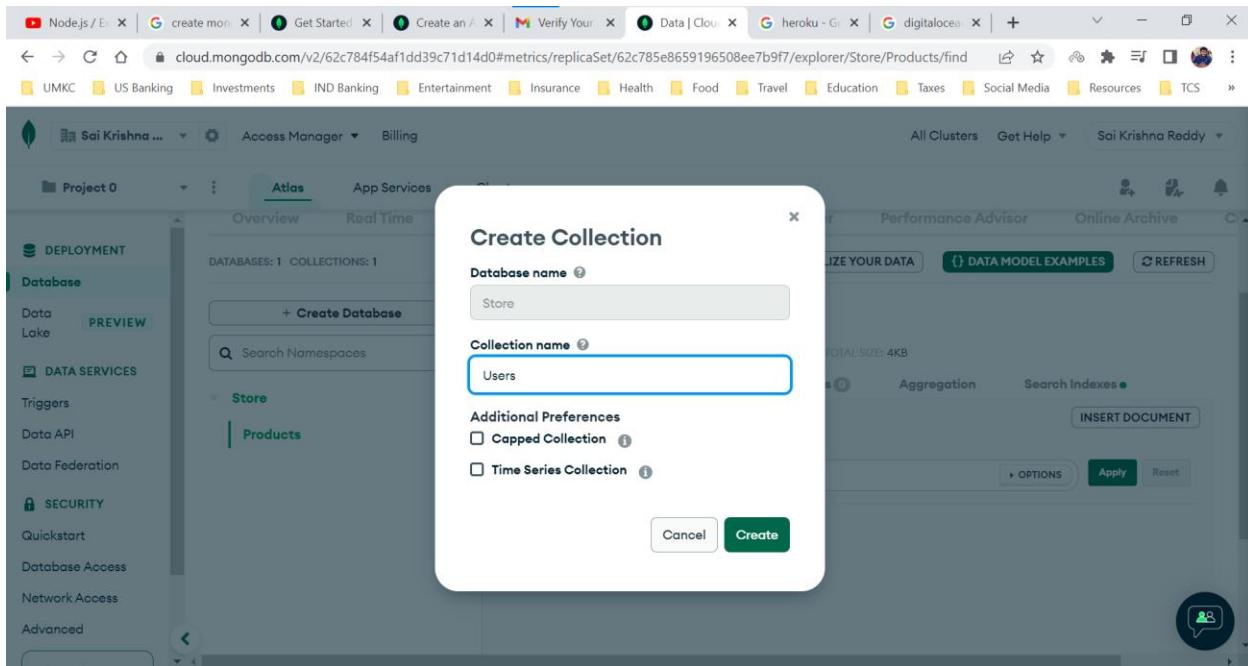
We can have as many collections (tables) as we want. In general SQL database we will be having many tables under one database.

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'. There's a 'Store' database with a 'Products' collection. A modal window titled 'Create Collection' is open, prompting for 'Database name' (set to 'Store') and 'Collection name' (set to 'Users'). Other options like 'Capped Collection' and 'Time Series Collection' are shown with checkboxes. At the bottom right 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 primary 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.

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.

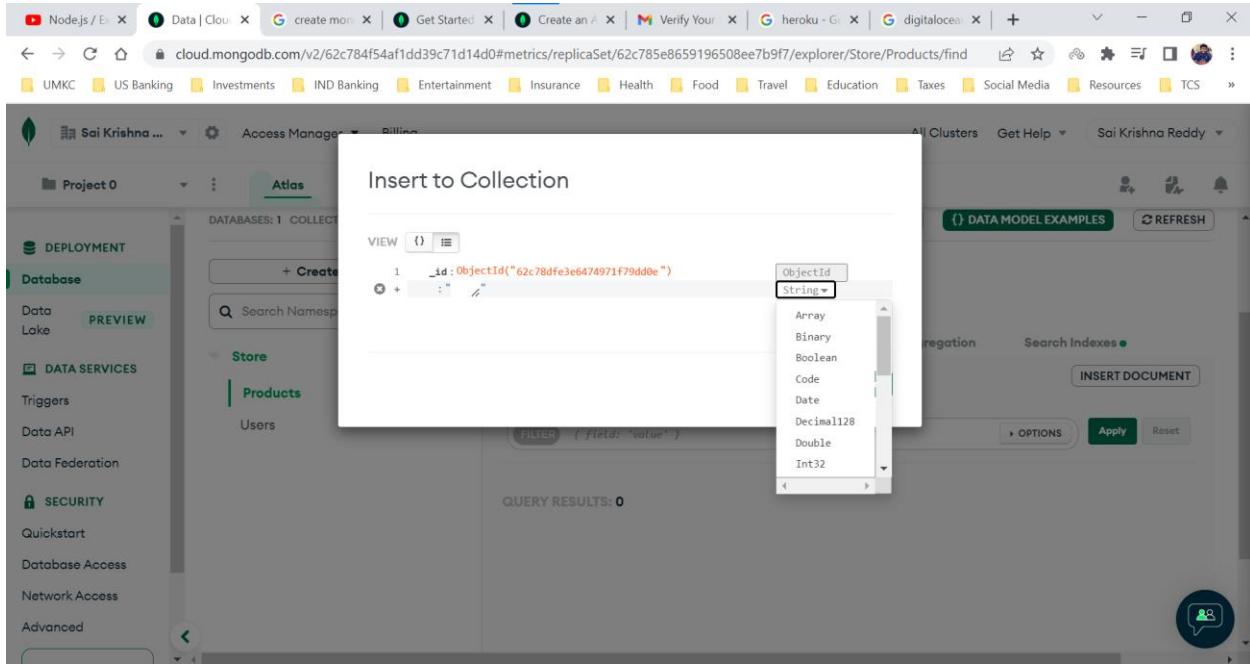
Now we are using the Manual Process to get a gist about MongoDB but once we start developing from server, we do all the operations using a tool called **mongoose**.

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.

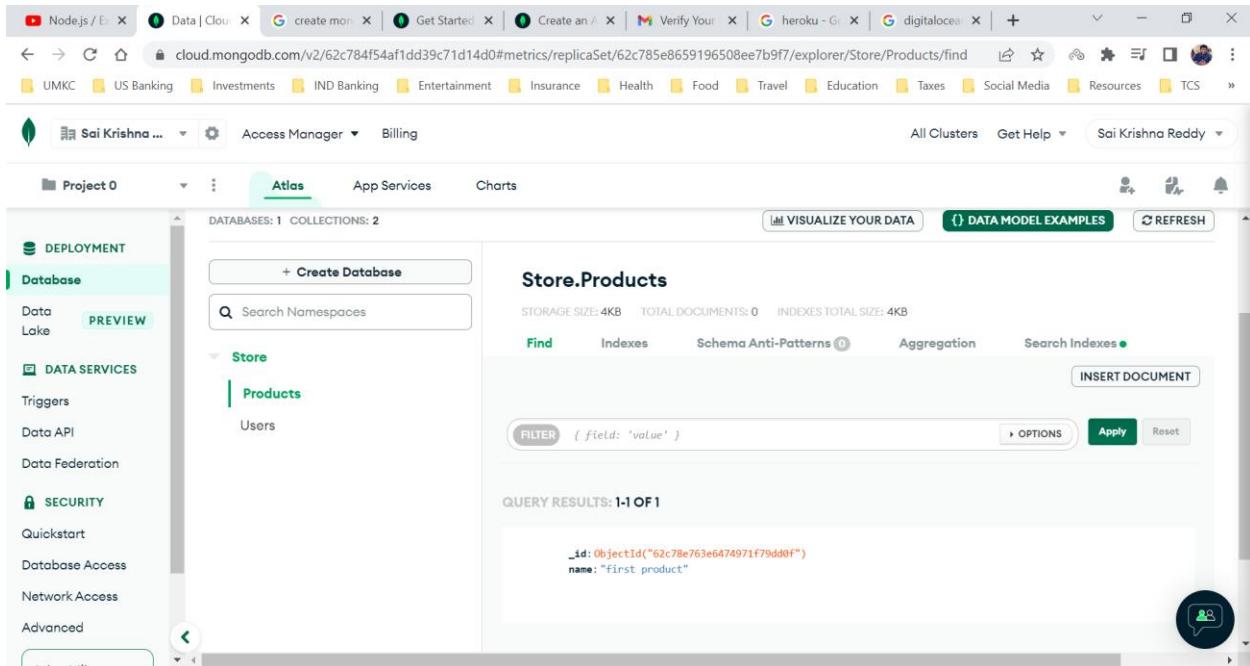
Here each document contains two details (key-value pairs), the first one being the id which is given by MongoDB by default.



The screenshot shows the MongoDB Atlas interface. On the left, there's a sidebar with 'Project 0' selected. Under 'Database', 'PREVIEW' is chosen. In the center, under 'Store', 'Products' is selected. A modal window titled 'Insert to Collection' is open, showing a document with an '\_id' field set to an ObjectId. A dropdown menu is open next to the value, showing options like ObjectId, String, Array, Binary, Boolean, Code, Date, Decimal128, Double, Int32, and Int64. At the bottom of the modal, there's an 'INSERT DOCUMENT' button.

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

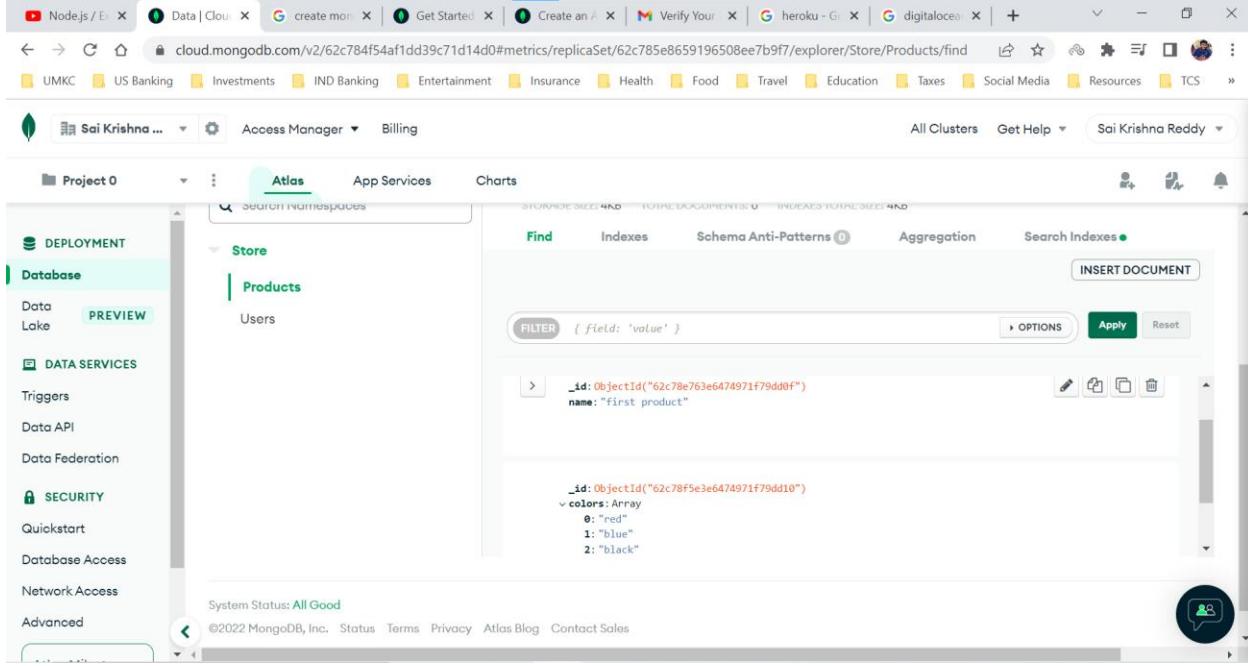
We have a created a document (a single item) with a key named name and value being first product.



The screenshot shows the MongoDB Atlas interface. On the left, there's a sidebar with 'Project 0' selected. Under 'Database', 'PREVIEW' is chosen. In the center, under 'Store', 'Products' is selected. The main area shows the 'Store.Products' collection with a single document listed. The document has an '\_id' field (ObjectId) and a 'name' field with the value 'First product'. The document is highlighted in red. At the bottom, it says 'QUERY RESULTS: 1-1 OF 1'.

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' }' with an 'Apply' button. Below the filter, 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, it says 'System Status: All Good'.

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 right shows 'All Clusters' and 'Sai Krishna Reddy'. Below the cluster selector, there are buttons for 'VISUALIZE YOUR DATA', 'DATA MODEL EXAMPLES', and 'REFRESH'. The main area displays the 'Store.Products' collection details: 'STORAGE SIZE: 4KB', 'TOTAL DOCUMENTS: 0', and '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:

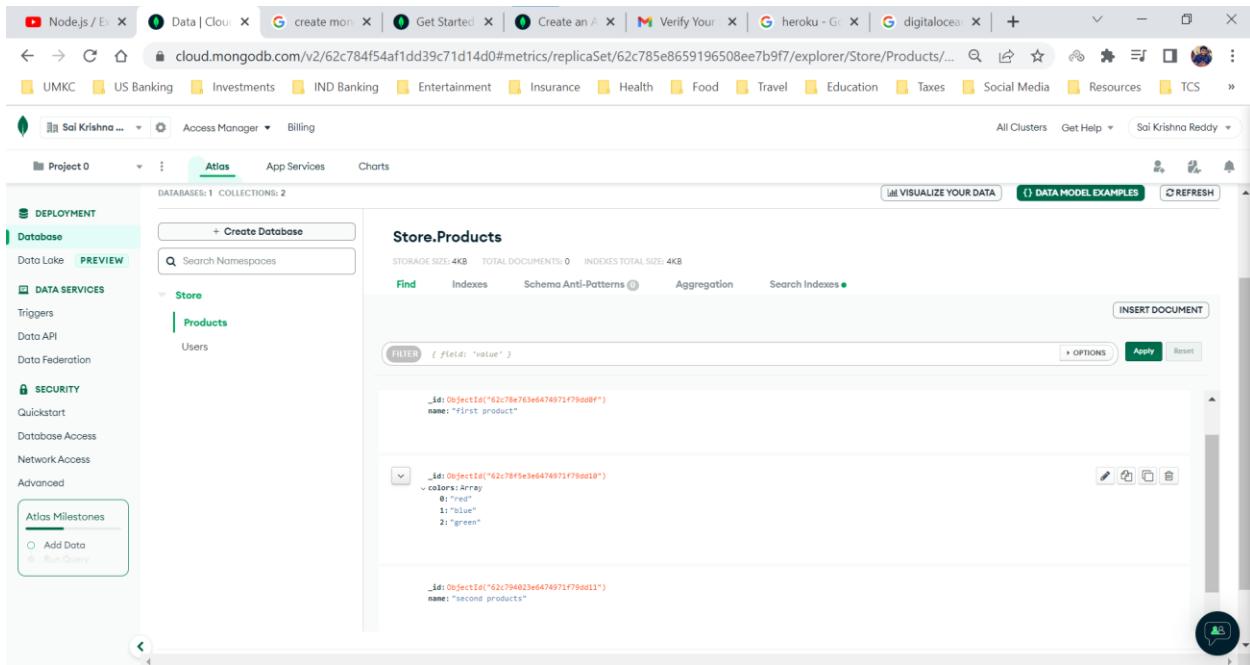
```
_id: ObjectId("62c78e763e6474971f79dd0f")
name: "first product"
```

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

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

## Update

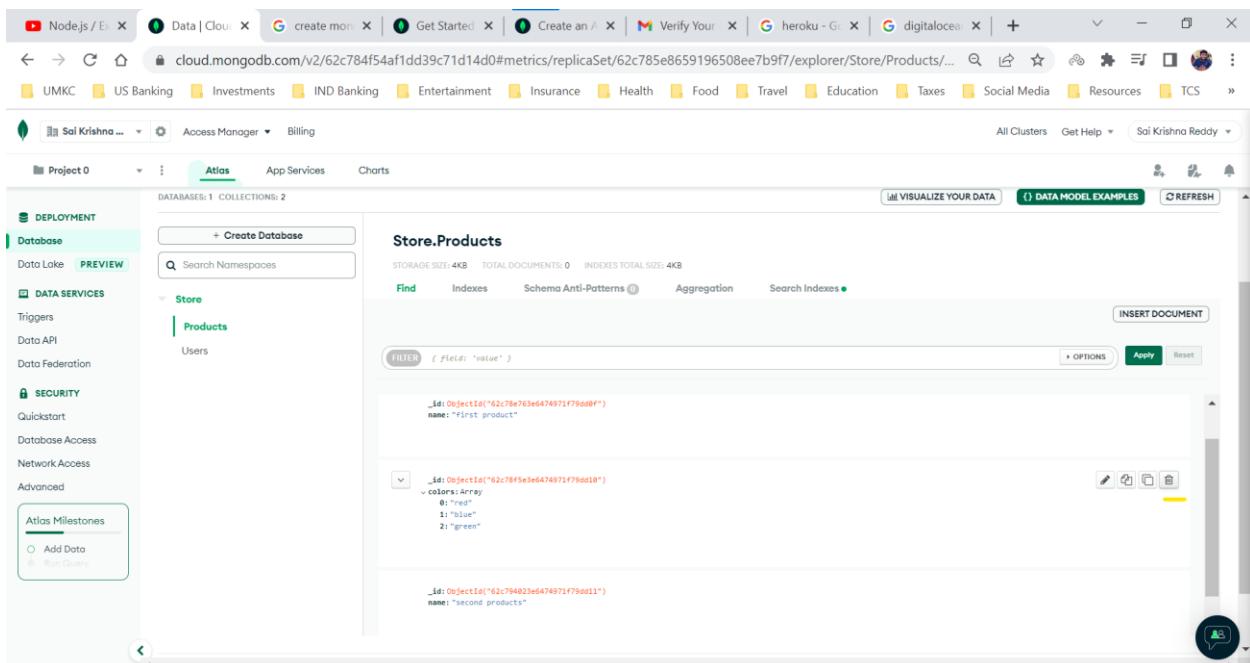
Updating a document in collection. (Updated a item in the color array from black to green)



The screenshot shows the MongoDB Atlas interface. On the left sidebar, under the 'Store' section, there is a 'Products' entry. In the main area, the 'Store.Products' collection is displayed. A document is selected for editing, showing its internal structure. The document includes fields like '\_id' (ObjectId), 'name' (e.g., "first product"), and a 'colors' array containing three items: 'red', 'blue', and 'green'. An edit icon is visible next to the 'colors' field, indicating it can be updated.

## Delete

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



This screenshot is similar to the previous one but shows a different state. The same 'Store.Products' collection is shown, but the second document (with '\_id' ObjectId("62c794023e6474971f79dd11") and name "second products") is now highlighted with a yellow background. This indicates it is selected for deletion. The edit icon is still present on the first document's colors array.

The screenshot shows the MongoDB Atlas interface. On the left sidebar, under 'DEPLOYMENT', 'Database' is selected, showing 'Data Lake' and 'PREVIEW'. Under 'DATA SERVICES', 'Store' is selected, showing 'Products' and 'Users'. In the main panel, the 'Store' section is active, displaying the 'Products' collection. The collection has 1 database and 2 documents. The first document, '\_id: ObjectId("62c78f54af1dd39c71d14d0")', has a 'name' field value of 'first product'. The second document, '\_id: ObjectId("62c794023e6474971f798d11")', has a 'name' field value of 'second products'. A red box highlights the 'DELETE' button for the first document, indicating it is flagged for deletion.

After deleting the documents

The screenshot shows the MongoDB Atlas interface after the deletion of the first document. The main panel now displays the 'Store' section with the 'Products' collection. It shows 1 database and 1 document. The single document, '\_id: ObjectId("62c794023e6474971f798d11")', has a 'name' field value of 'second products'. The 'DELETE' button is no longer highlighted.

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

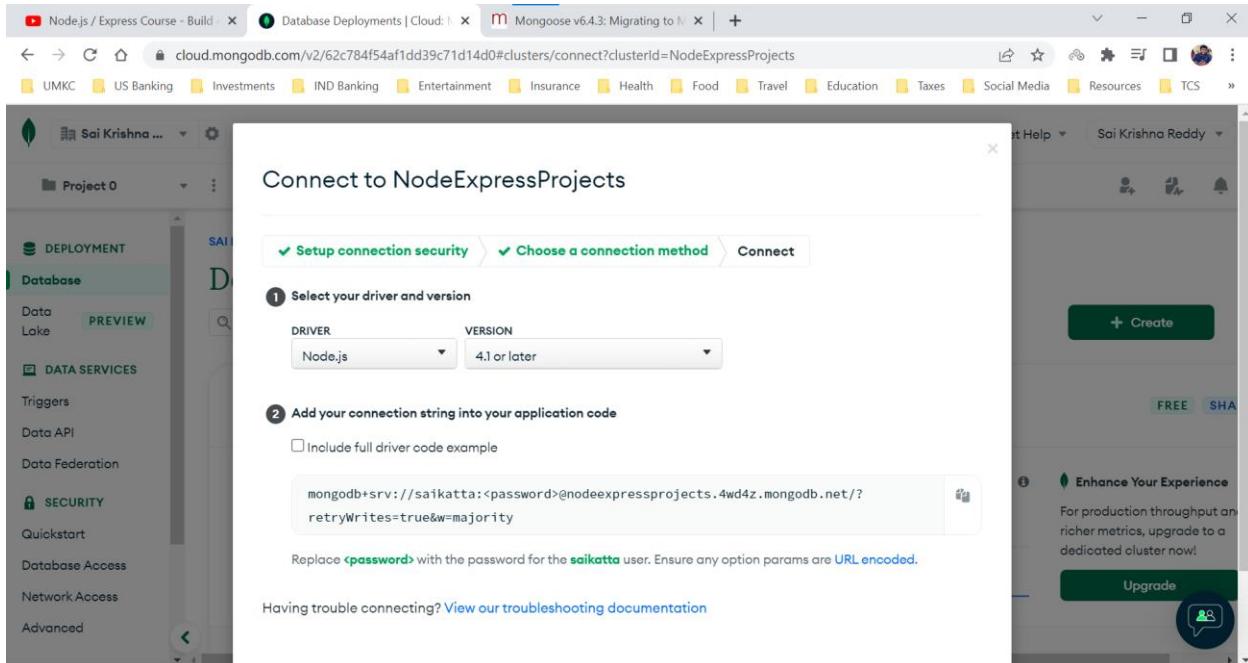
```
npm install mongoose
```

## 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:

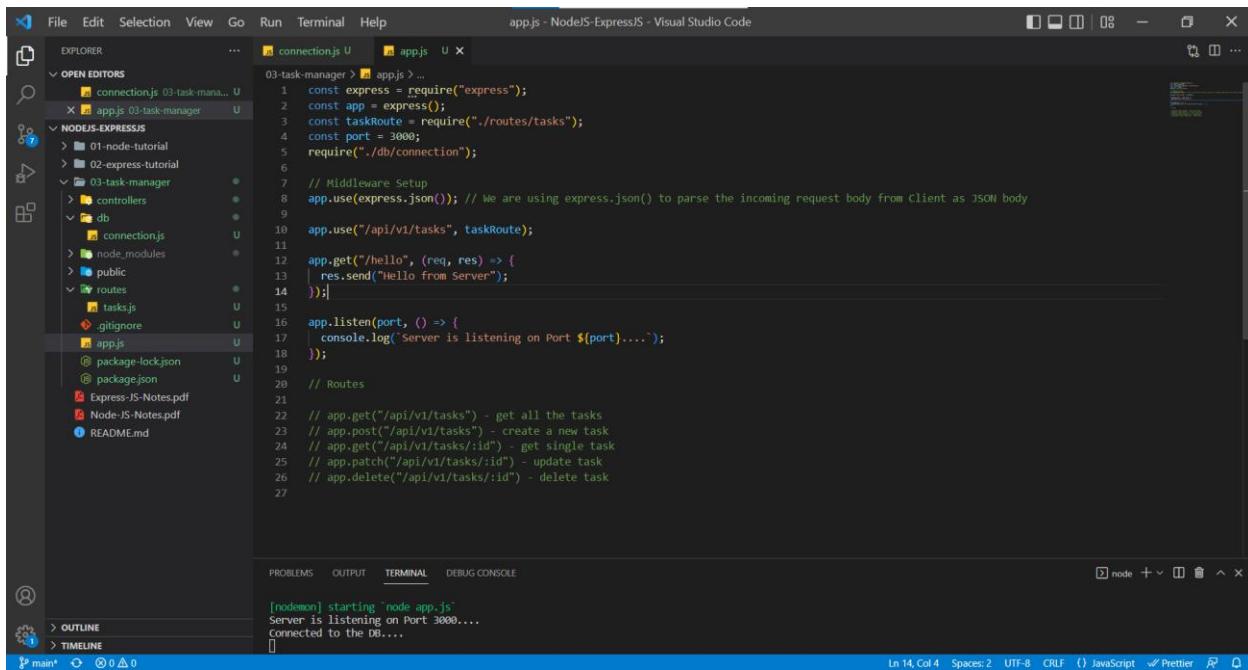
- File Explorer:** Shows the project structure under "NODEJS-EXPRESS". The "db" folder is expanded, showing "connection.js" and "app.js".
- Editor:** The "connection.js" file is open, displaying the following code:

```
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:** The terminal tab is active, showing the output of the application's execution. It includes logs from nodemon and the application itself, indicating a successful connection to the database.

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
```

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

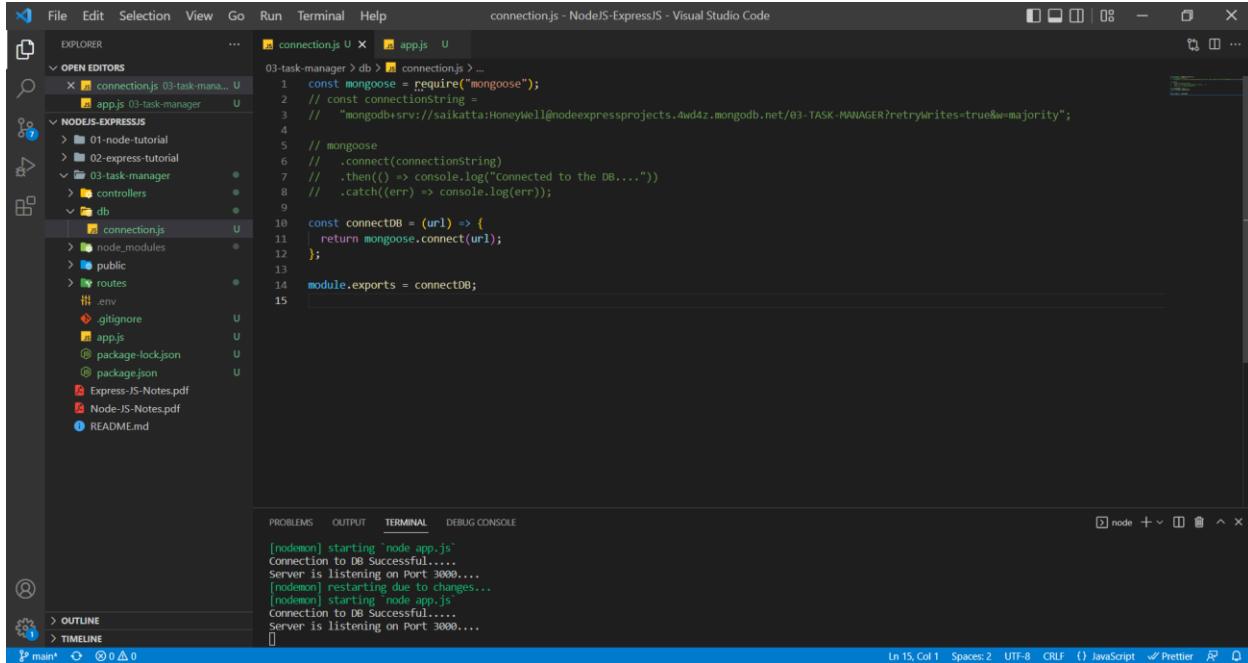
app.use("/api/v1/tasks", taskRoute);

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

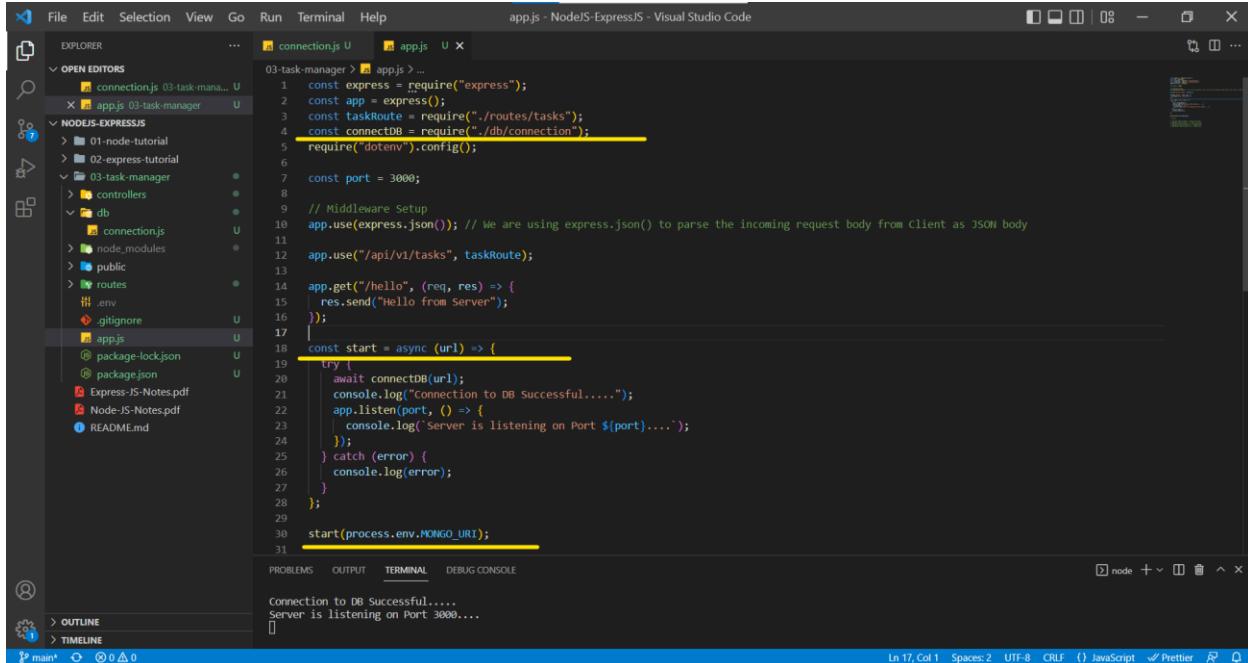


The screenshot shows the Visual Studio Code interface with the connection.js file open in the editor. The code connects to a MongoDB database using mongoose. It defines a connectDB function that takes a URL and returns a promise. The module exports this function. The terminal below shows the output of running nodemon, indicating successful database connection and server startup 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 middleware for parsing JSON bodies, and defines routes for hello and tasks. It then starts the server at port 3000 after connecting to the database. The terminal below shows the output of running nodemon, indicating successful database connection and server startup 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.

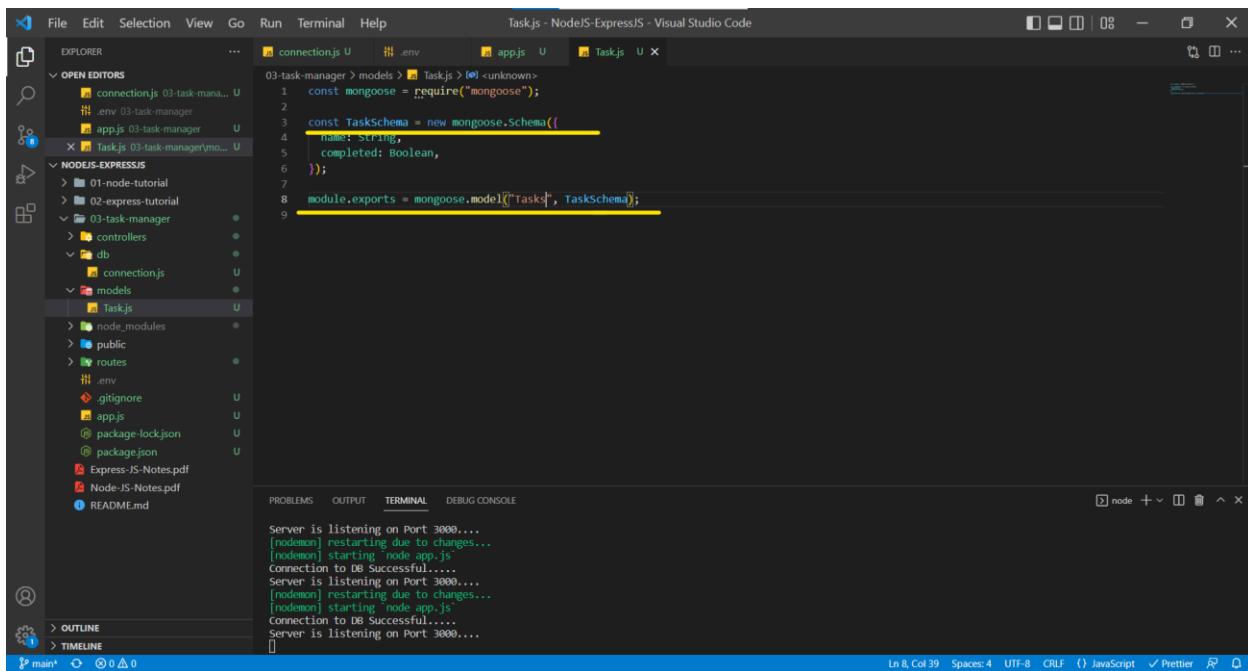
We will create a folder named models and inside of models we create a file named Task.js. With the help of mongoose schema, we are going to set up the structure of the documents in a collection.

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 at the bottom 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', and a search bar. A banner at the top right says 'Working locally in Scratch Pad. Switch to a Workspace'. The left sidebar is titled 'Scratch Pad' and 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 to '03-Task-Manager / Create Task'. The 'Body' tab is selected, showing a JSON payload:

```
1 {  
2   ... "name": "first task",  
3   ... "completed": true  
4 }
```

The response section shows a 201 Created status with a response body:

```
1 {  
2   "name": "first task",  
3   "completed": true,  
4   "_id": "62c8db52ef8f7dc52367e41f",  
5   "__v": 0  
}
```

At the bottom, there are 'Pretty', 'Raw', 'Preview', 'Visualize', and 'JSON' buttons, along with a 'Save Response' button.

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 under Database. The main area shows the '03-TASK-MANAGER' database with the '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 reads '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 is expanded, showing 'POST Create Task', 'GET Get Single Task', 'PATCH Update Task', and 'DELETE Delete Task'.

The central workspace displays a 'POST Create Task' request for the URL '((URL))/tasks'. The 'Body' tab is selected, showing a JSON payload:

```
1 ... "name": "test-task",
2 ... "completed": false,
3 ... "random": "abcdefghijklm",
4 ... "checked": false
```

The response pane shows a successful '201 Created' status with a response time of '481 ms' and a size of '319 B'. The response body is identical to the request body.

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.
- Middle Section:** Atlas (selected), App Services, Charts.
- Create Database:** + Create Database.
- Search Namespace:** Search Namespaces.
- Database List:** 03-TASK-MANAGER (selected), tasks.
- Collection List:** Store.
- Collection Details:** 03-TASK-MANAGER.tasks.
  - STORAGE SIZE: 36KB TOTAL DOCUMENTS: 2 INDEXES TOTAL SIZE: 36KB
  - Find, Indexes, Schema Anti-Patterns, Aggregation, Search Indexes.
  - INSERT DOCUMENT button.
  - FILTER button with query: { field: 'value' }.
  - Apply and Reset buttons.
  - Document Preview:
    - name: "first task"  
completed: true  
\_\_v: 0
    - \_\_v: 0
    - \_\_v: 0
    - \_\_v: 0

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

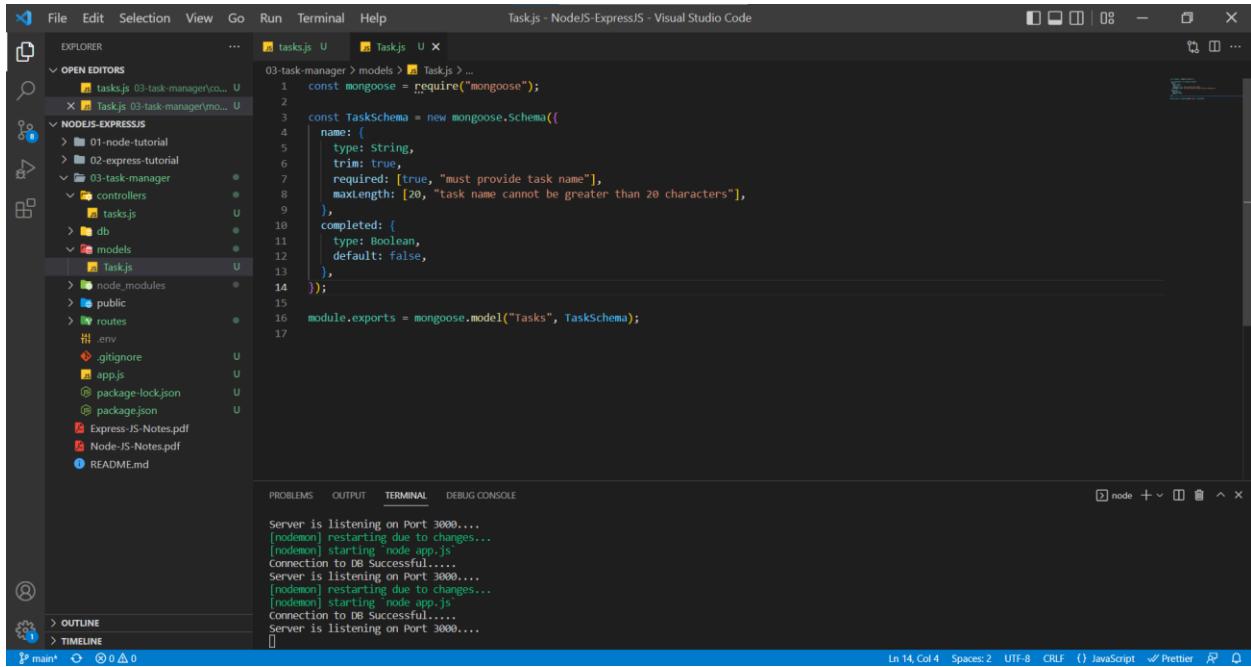
The screenshot shows the Postman interface. In the left sidebar, under 'APIs', there is a section for '03-Task-Manager' which includes 'POST Create Task'. The main workspace shows a POST request to '03-Task-Manager / Create Task'. The 'Body' tab is selected, showing a JSON object with three fields: '\_id', 'name', and '\_v'. The value for 'name' is empty ('"name": ""'). The response status is '201 Created'.

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 'tasks' listed. The main area shows the 'tasks' collection with a table header: '\_id', 'name', and '\_v'. A single document is visible in the list, showing '\_id: ObjectId("62c8e316ef8f7dc52367e425")', 'name: "",' and '\_v: 0'. The document has edit, delete, and copy icons next to it.

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

Task.js in models folder



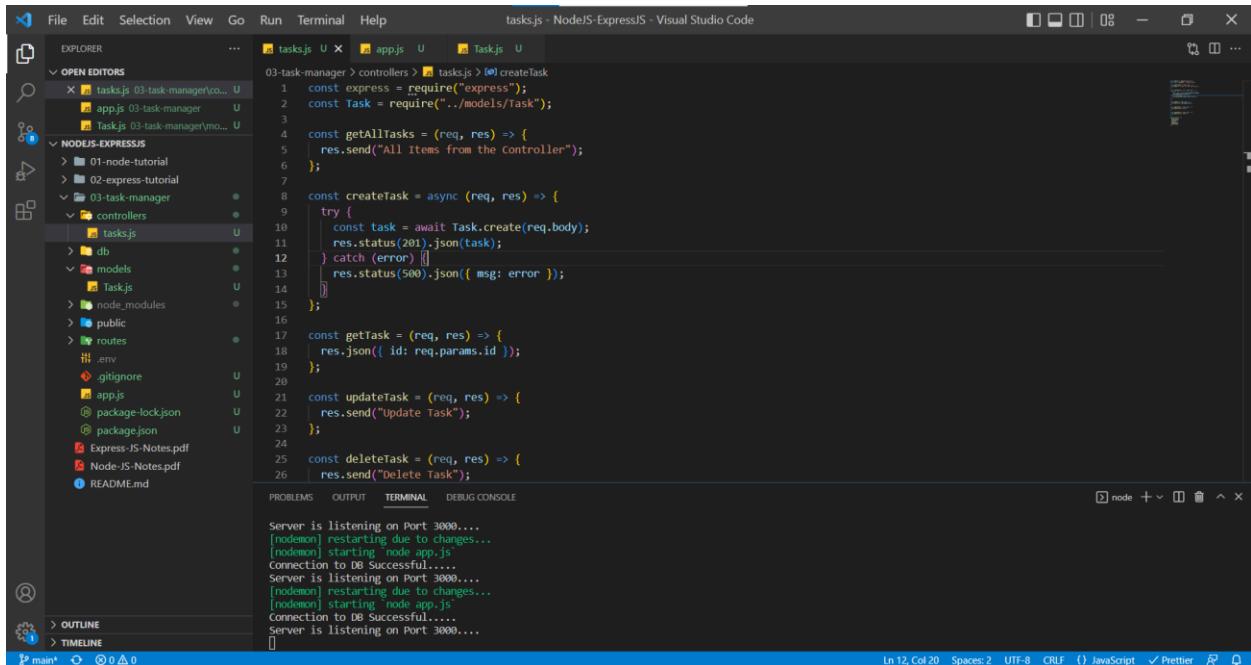
The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure with files like tasks.js, Task.js, app.js, routes, db, and models.
- 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, trim, max length of 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, node.js is running, and other details like spaces, tabs, and 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.

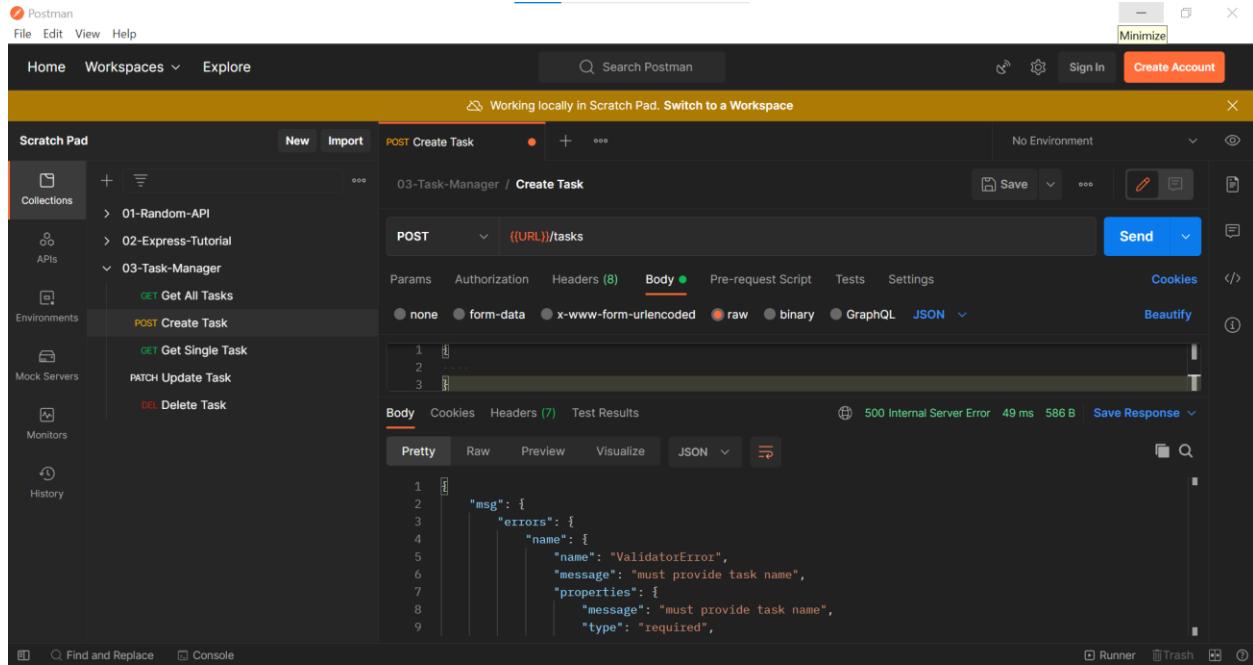
app.js



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

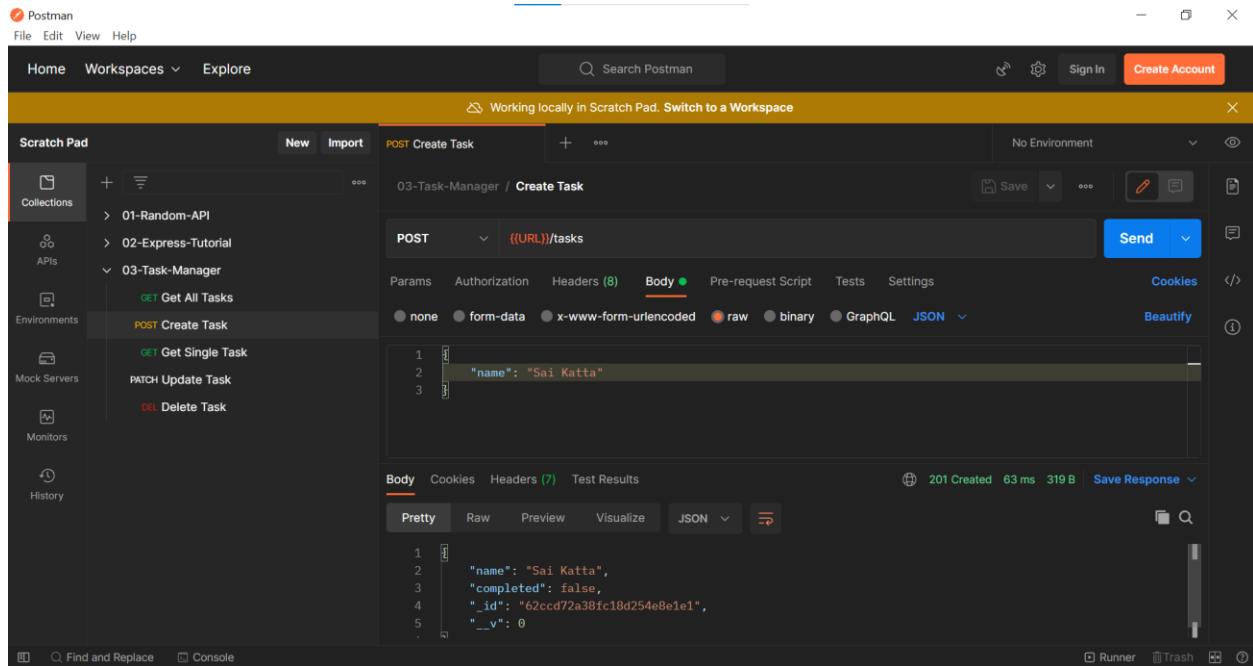
- File Explorer:** Shows the project structure with files like tasks.js, app.js, routes, db, and models.
- Code Editor:** Displays the content of app.js, which contains several middleware functions (express, task, getAllTasks, createTask, getTask, updateTask, deleteTask) for handling requests to the /tasks endpoint.
- 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, node.js is running, and other details like spaces, tabs, and 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 empty. 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 contains a single key 'name' with the value '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"});*

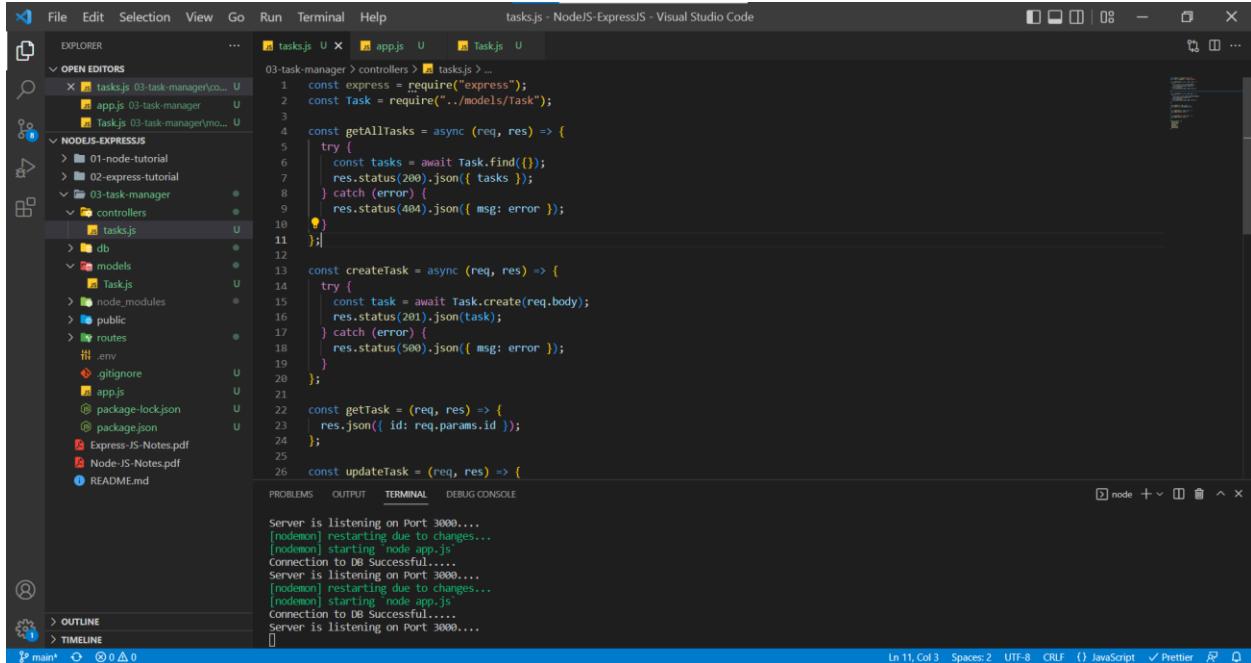
```
res.status(400).json({error});
```

*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.

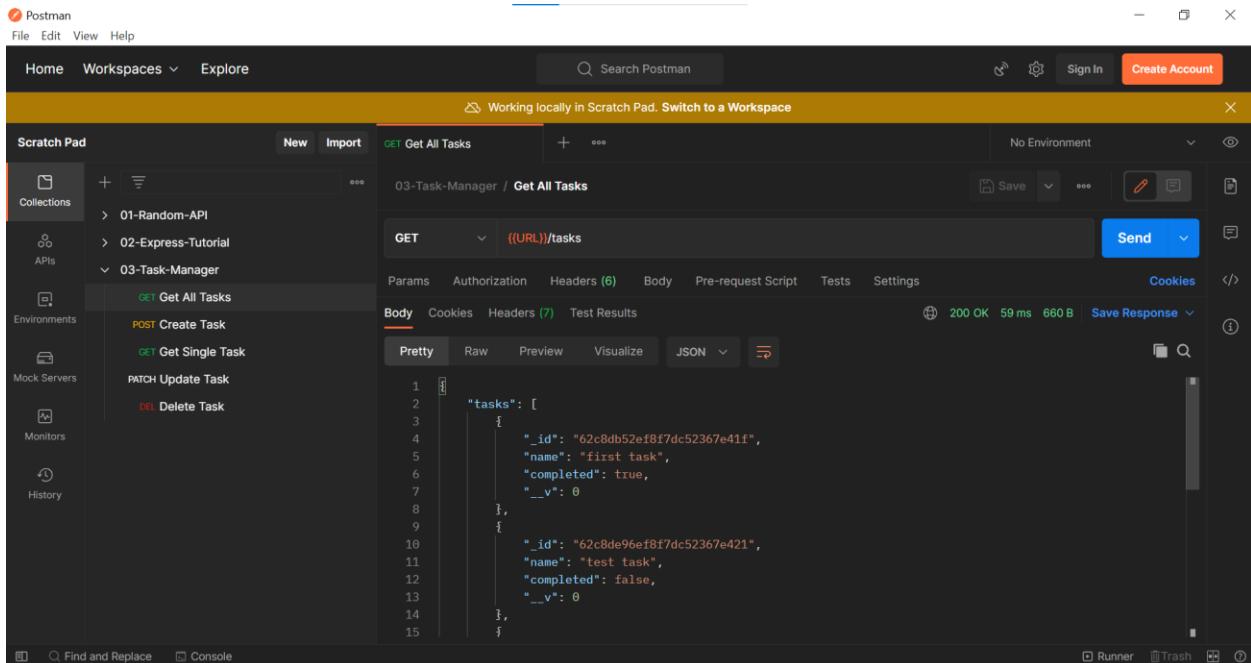
## Get All Tasks



```
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   ]
```

## 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 (Left):** Shows the project structure under "NODES-EXPRESS".
  - 01-node-tutorial
  - 02-express-tutorial
  - 03-task-manager
    - controllers
      - tasks.js
    - db
    - models
      - Task.js
    - routes
      - env
      - gitignore
      - app.js
    - node\_modules
    - public
    - routes
      - Express-JS-Notes.pdf
      - Node-JS-Notes.pdf
    - README.md
- Code Editor (Top Center):** The file "tasks.js" is open, showing Node.js code for a task manager.

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

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

42 }
```
- Terminal (Bottom):** Shows the terminal output of the application.

```
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 'Postman' (with a logo), 'File', 'Edit', 'View', 'Help', 'Home', 'Workspaces', 'Explore', a search bar ('Search Postman'), and user account options ('Sign In', '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', 'APIs', 'Environments', 'Mock Servers', 'Monitors', and 'History'. The main workspace is titled 'Scratch Pad' and shows a '03-Task-Manager / Get Single Task' request. The request details are as follows:

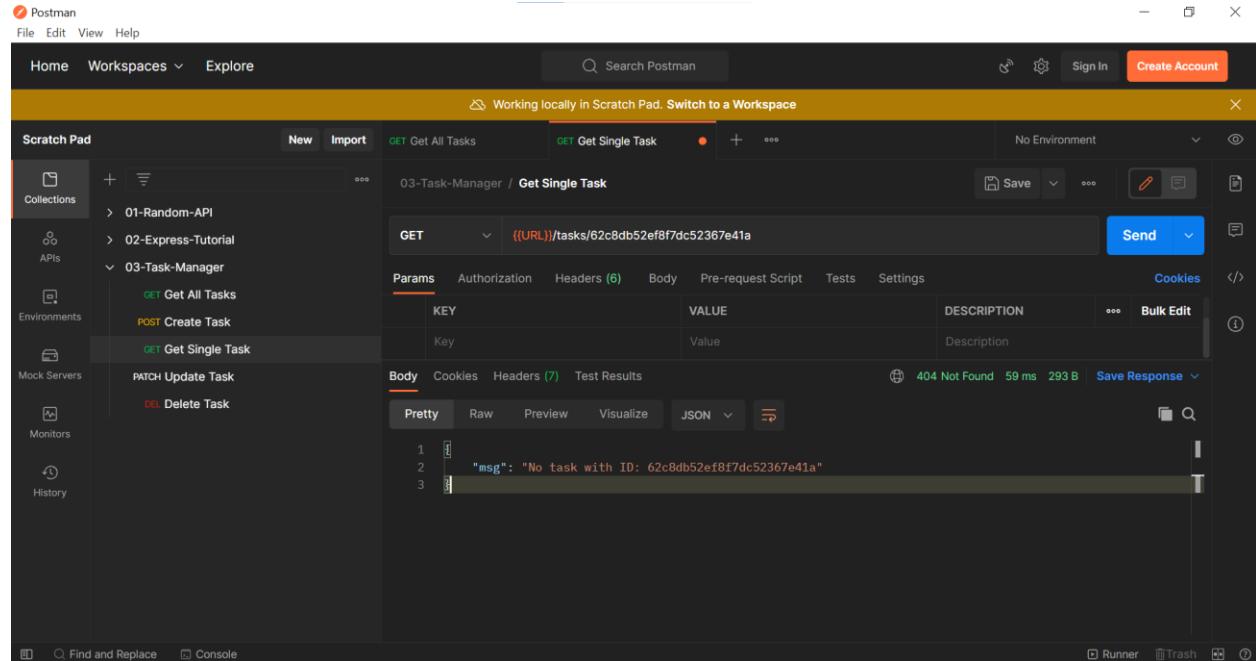
- Method: GET
- URL: `((URL))/tasks/62c8db52ef8f7dc52367e41f`
- Params tab (selected):

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description	...	
- Body tab:
  - Pretty (selected)
  - Raw
  - Preview
  - Visualize
  - JSON

```
1 [ { "task": { "id": "62c8db52ef8f7dc52367e41f", "name": "first task", "completed": true, "__v": 0 } } ]
```
- Response status: 200 OK, 106 ms, 323 B, Save Response

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

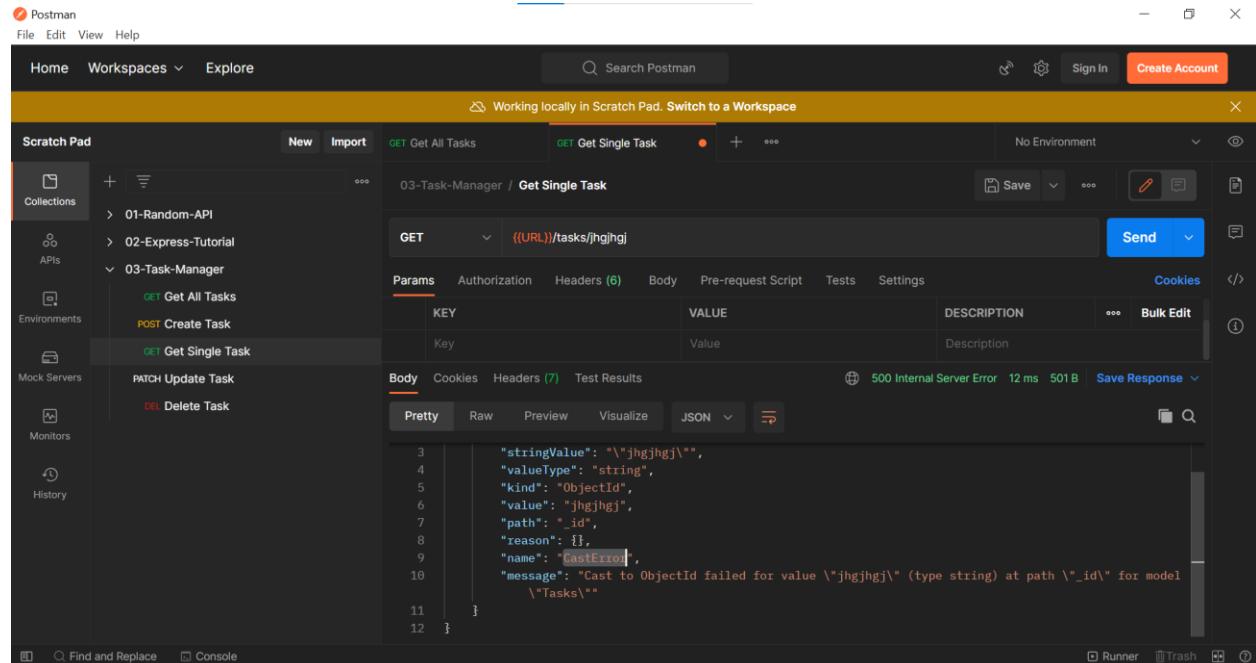
### Response 404 Status Code



The screenshot shows the Postman application interface. In the center, there's a request configuration window for a 'GET Get Single Task' operation. The URL is set to `((URL))/tasks/62c8db52ef8f7dc52367e41a`. Below the URL, under the 'Params' tab, there's a table with one row: 'Key' (empty) and 'Value' (empty). The 'Body' tab is selected, showing a JSON response with a single line: `"msg": "No task with ID: 62c8db52ef8f7dc52367e41a"`. At the bottom of the response pane, it says '404 Not Found 59 ms 293 B'. The status bar at the bottom right indicates 'Save Response'.

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



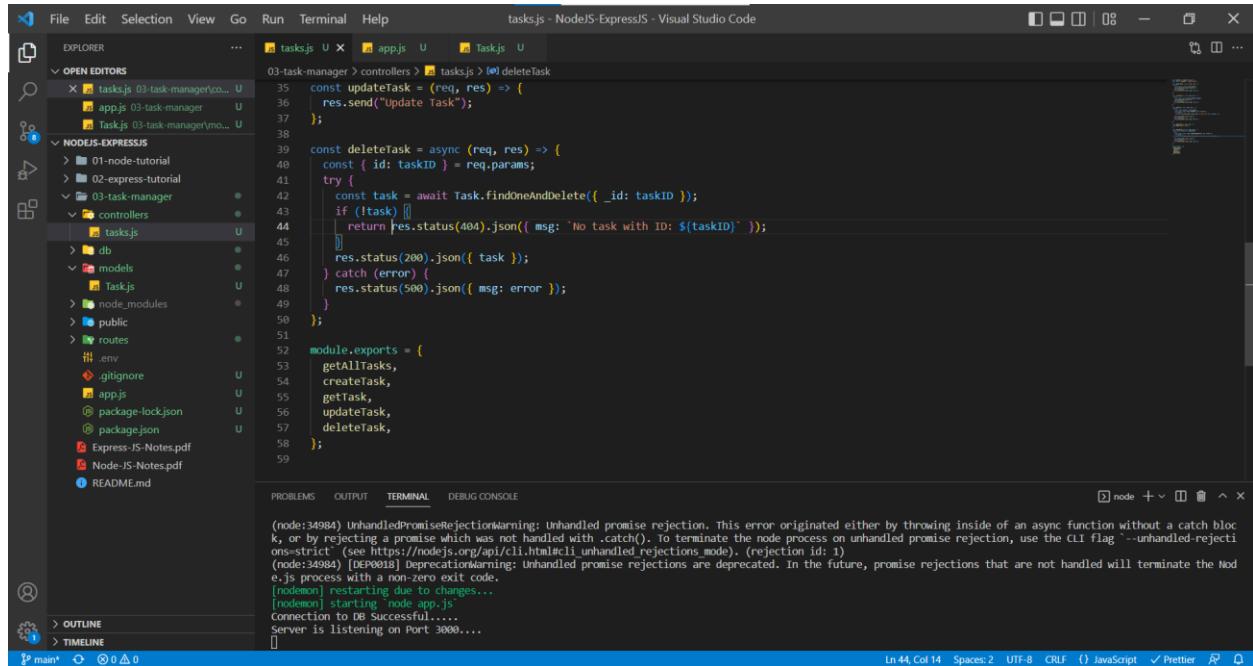
The screenshot shows the Postman application interface. In the center, there's a request configuration window for a 'GET Get Single Task' operation. The URL is set to `((URL))/tasks/jhgjhgj`. Below the URL, under the 'Params' tab, there's a table with one row: 'Key' (empty) and 'Value' (empty). The 'Body' tab is selected, showing a JSON response with multiple lines. Lines 3 through 12 of the JSON output are:  `"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\""`. At the bottom of the response pane, it says '500 Internal Server Error 12 ms 501 B'. The status bar at the bottom right indicates 'Save Response'.

## 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.

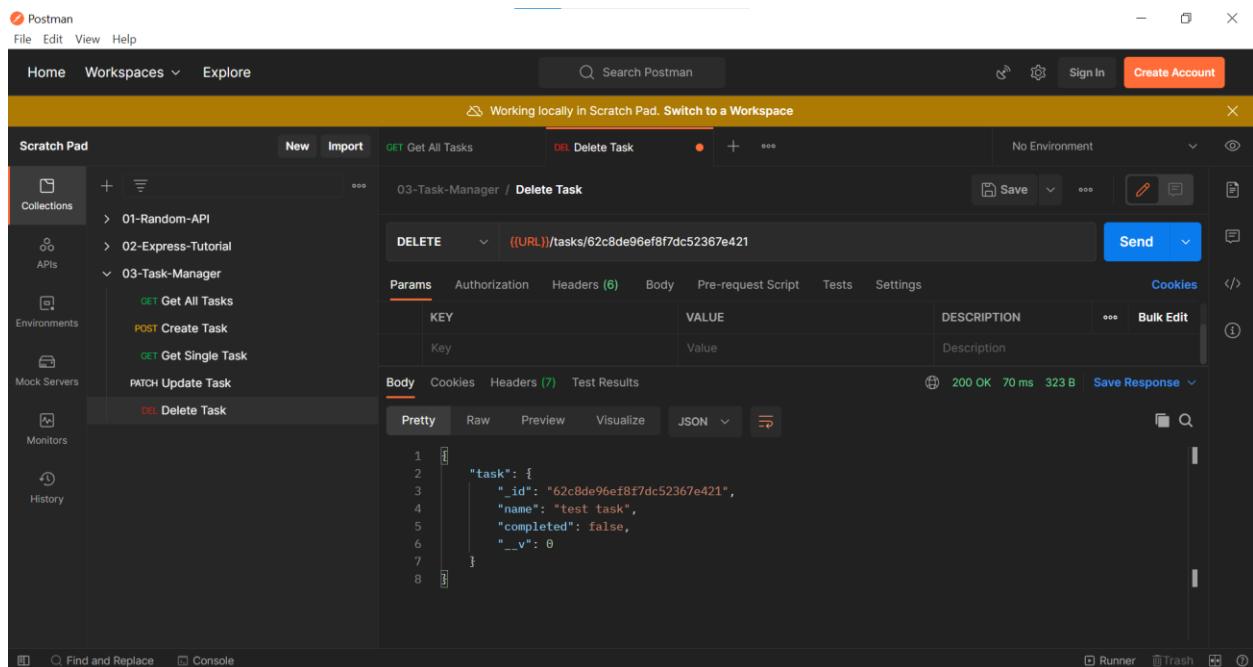


```
const updateTask = (req, res) => {
  res.send("Update Task");
};

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



Working locally in Scratch Pad. Switch to a Workspace

Scratch Pad

03-Task-Manager / Delete Task

DELETE `((URL))/tasks/62c8de96ef8f7dc52367e421`

Params

KEY	VALUE	DESCRIPTION	Bulk Edit
Key	Value	Description	

Body

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

200 OK 70 ms 323 B Save Response

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, which is currently selected. The "DELETE" tab is active, showing the URL as "DELETE {{URL}}/tasks/62c8de96ef8f7dc52367e422". The "Params" tab is selected, showing a table with one row:

KEY	VALUE	DESCRIPTION	Bulk Edit
Key	Value	Description	

The "Body" tab is also visible. At the bottom of the interface, the status bar shows "404 Not Found 65 ms 293 B Save Response".

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:

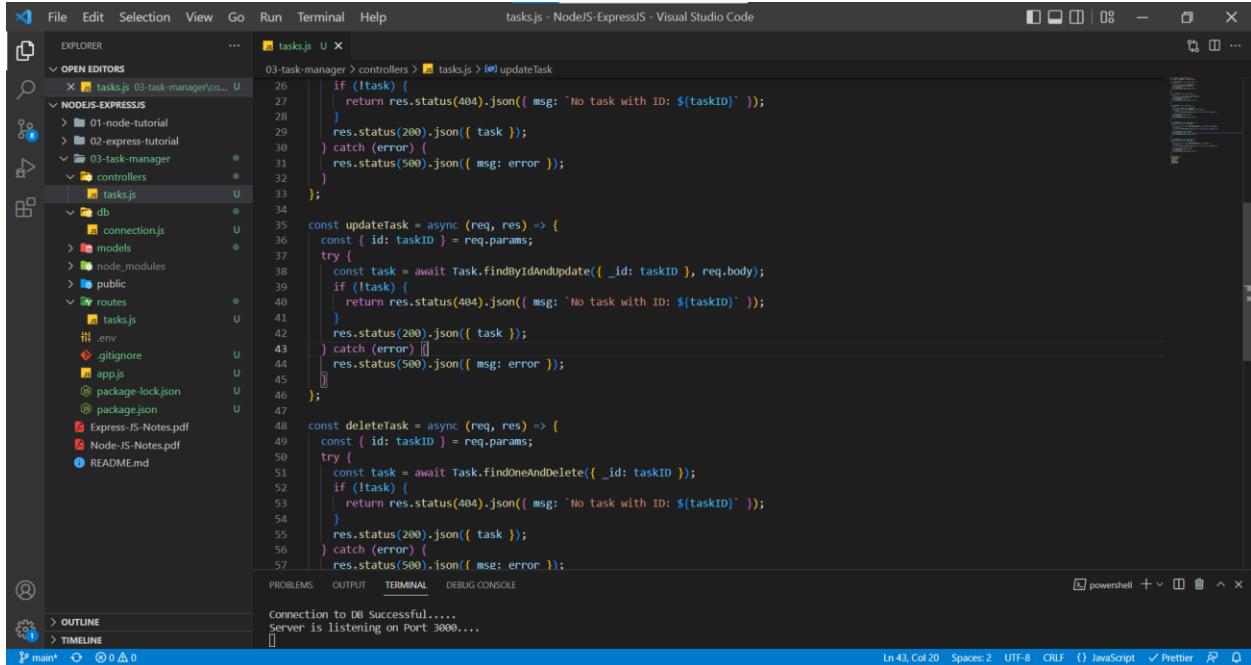
```
3 |     "stringValue": "\"62c8de96ef8f7dc52367e422jjhgjhgjh\"",
4 |     "valueType": "string",
5 |     "kind": "ObjectId",
6 |     "value": "62c8de96ef8f7dc52367e422jjhgjhgjh",
7 |     "path": "_id",
8 |     "reason": {},
9 |     "name": "CastError",
10 |    "message": "Cast to ObjectId failed for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



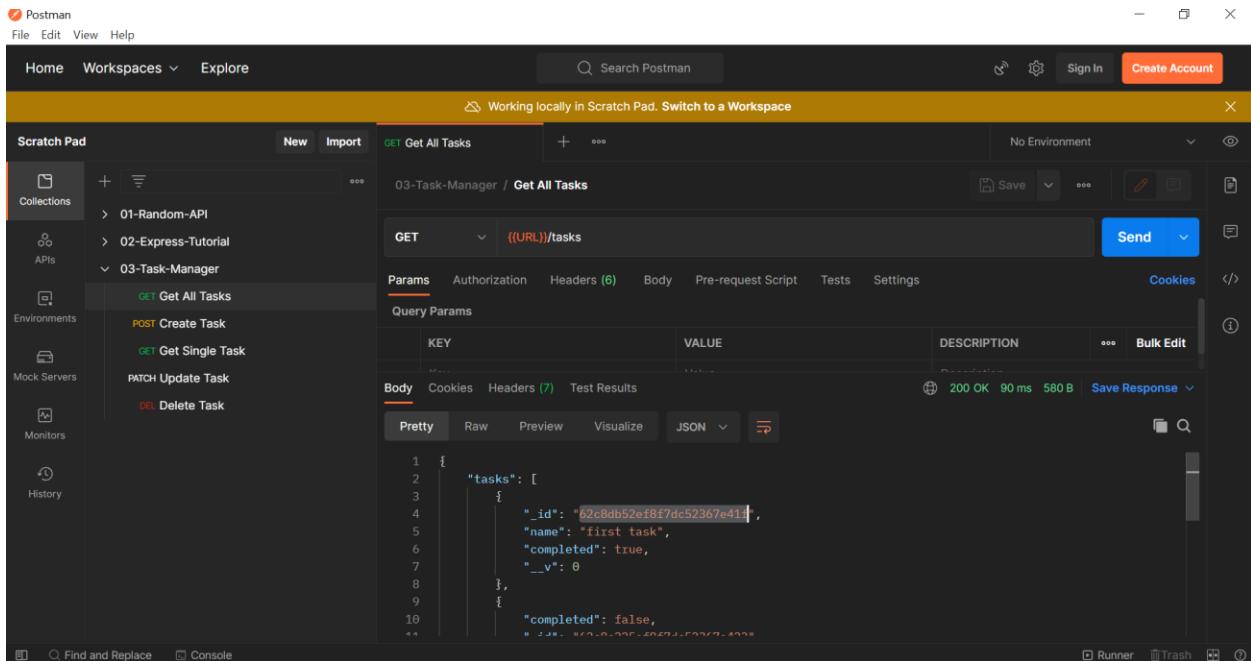
The screenshot shows the Visual Studio Code interface with the tasks.js file open in the editor. The file contains code for updating, deleting, and getting tasks from a database. The code uses async/await syntax and the Task model. The terminal at the bottom shows a successful connection to the database and the server listening on port 3000.

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

EXPLORER tasks.js 03-task-manager/controllers/tasks.js [10] updateTask
OPEN EDITORS tasks.js 03-task-manager/controllers/tasks.js [10] updateTask
NODEJS-EXPRESSJS
  01-node-tutorial
  02-express-tutorial
  03-task-manager
    controllers
      tasks.js
    db
      connection.js
    models
    node_modules
    public
    routes
      tasks.js
    .env
    .gitignore
    app.js
    package-lock.json
    package.json
    Express-JS-Notes.pdf
    Node-JS-Notes.pdf
    README.md

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
Connection to DB Successful.....
Server is listening on Port 3000...
Ln 43, Col 20  Spaces: 2  UTF-8  CHRF  () JavaScript  ✓ Prettier  🔍  ⌂
```

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



The screenshot shows the Postman interface with a collection named "03-Task-Manager". A GET request is selected to "Get All Tasks" with the URL {{URL}}/tasks. The "Params" tab shows a query parameter "id" with the value "62c8db52ef8f7dc52367e41f". The response body is displayed in JSON format, showing a list of tasks where the first task has an \_id of "62c8db52ef8f7dc52367e41f", a name of "first task", and a completed status of true.

```
GET {{URL}}/tasks
Params:
  KEY: id, VALUE: 62c8db52ef8f7dc52367e41f
Body:
  PRETTY
  [
    {
      "tasks": [
        {
          "_id": "62c8db52ef8f7dc52367e41f",
          "name": "first task",
          "completed": true,
          "__v": 0
        },
        {
          "completed": false,
          "__v": 0
        }
      ]
    }
  ]
```

## 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 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', 'Get Single Task', 'Update Task', and 'Delete Task'), 'Environments', 'Mock Servers', 'Monitors', and 'History'. The main workspace is titled '03-Task-Manager / Get All Tasks'. It shows a 'GET' request to '((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' latency and '590 B' size. Buttons for 'Send', 'Save', and 'Bulk Edit' are visible. The bottom navigation bar includes 'Find and Replace', 'Console', 'Runner', 'Trash', and other icons.

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   ...
6   "completed": "false"
7 }
```

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

```
1
2   ...
3   ...
4   "task": {
5     "_id": "62c8db52ef8f7dc52367e41f",
6     "name": "testing update func",
7     "completed": false,
8     ...
9   }
10 }
```

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 a duration of 65 ms and a size of 571 B. The response body is displayed in JSON format:

```
1
2   ...
3   ...
4   "tasks": [
5     {
6       ...
7     },
8     {
9       ...
10      ...
11    }
12 }
```

***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

```
File Edit Selection View Go Run Terminal Help tasksjs - NodeJS-ExpressJS - Visual Studio Code

EXPLORER tasks.js U x
OPEN EDITORS tasks.js 03-task-manager\co... U
NODEJS-EXPRESS
  1 node-tutorial
  2 express-tutorial
  3 03-task-manager
    controllers tasksjs U
      db connection.js U
      models U
      public U
      routes tasksjs U
        tasksjs U
        .env U
        gitignore U
        app.js U
        package-lock.json U
        package.json U
        ExpressJS-Notes.pdf U
        NodeJS-Notes.pdf U
        README.md U

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
powershell +  terminal ^ x

tasks.js U x
03-task-manager > controllers > tasks.js > updateTask > task
29   res.status(200).json({ task });
30 } catch (error) {
31   res.status(500).json({ msg: error });
32 }
33 };
34
35 const updateTask = async (req, res) => {
36   const { id: taskID } = req.params;
37   try {
38     const task = await Task.findByIdAndUpdate({ _id: taskID }, req.body, [
39       new: true,
40       runValidators: true,
41     ]);
42     if (!task) {
43       return res.status(404).json({ msg: `No task with ID: ${taskID}` });
44     }
45     res.status(200).json({ task });
46   } catch (error) {
47     res.status(500).json({ msg: error });
48   }
49 };
50
51 const deleteTask = async (req, res) => {
52   const { id: taskID } = req.params;
53   try {
54     const task = await Task.findOneAndDelete({ _id: taskID });
55     if (!task) {
56       return res.status(404).json({ msg: `No task with ID: ${taskID}` });
57     }
58     res.status(200).json({ task });
59   } catch (error) {
60     res.status(500).json({ msg: error });
61   }
62 };

Connection to DB Successful.....
Server is listening on Port 3000.....
[nodemon] restarting due to changes...
```

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

The screenshot shows the Postman application interface. The top navigation bar includes 'File', 'Edit', 'View', and 'Help' options, along with 'Sign In' and 'Create Account' buttons. The main header features 'Home', 'Workspaces', and 'Explore' buttons, and a search bar labeled 'Search Postman'. A yellow banner at the top center reads 'Working locally in Scratch Pad. Switch to a Workspace'. The left sidebar contains sections for 'Collections' (with items '01-Random-API', '02-Express-Tutorial', and '03-Task-Manager'), 'APIs', 'Environments', 'Mock Servers', 'Monitors', and 'History'. The central workspace is titled '03-Task-Manager / Update Task'. It shows a 'PATCH' request with the URL `((URL))/tasks/62c8db52ef8f7dc52367e41f`. The 'Body' tab is selected, showing the following JSON payload:

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

The 'Body' tab also displays the response: `200 OK`, `163 ms`, `331 B`, and a 'Save Response' button. Below the body, there are tabs for 'Pretty', 'Raw', 'Preview', 'Visualize', and 'JSON'. The bottom status bar includes icons for 'Find and Replace', 'Console', 'Runner', 'Trash', and other settings.

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

The screenshot shows the Postman interface with the following details:

- Header Bar:** Postman, File, Edit, View, Help, Search Postman, Sign In, Create Account.
- Left Sidebar (Scratch Pad):** Collections (01-Random-API, 02-Express-Tutorial, 03-Task-Manager), APIs, Environments, Mock Servers, Monitors, History.
- Central Area:**
  - Request URL:** PATCH ((URL))/tasks/62c8db52ef8f7dc52367e41f
  - Method:** PATCH
  - Body:** Body tab selected, Content Type: application/x-www-form-urlencoded, Data:

```
1 ... "name": "",  
2 ... "completed": "false"
```
  - Response Headers:** 500 Internal Server Error, 66 ms, 596 B, Save Response.
  - Response Body (Pretty JSON):**

```
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": ""
```
- Bottom Bar:** Find and Replace, Console, Runner, Trash.