express.js

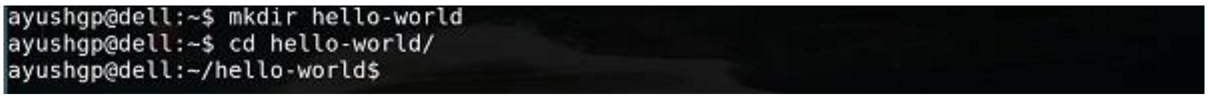
ExpressJS is a web application framework that provides you with a simple API to build websites, web apps, and backends.

What is Express?

- ✓ Express provides a minimal interface to build our applications.
- ✓ It provides us with the tools that are required to build our app.
- ✓ It is flexible as there are numerous modules available on **npm**, which can be directly plugged into Express.
- ✓ Express was developed by **TJ Holowaychuk** and is maintained by the node.js foundation and numerous open-source contributors.

Express Development Environment

Step 1 – Start your terminal/cmd, create a new folder named hello-world, and cd (create the directory) into it –

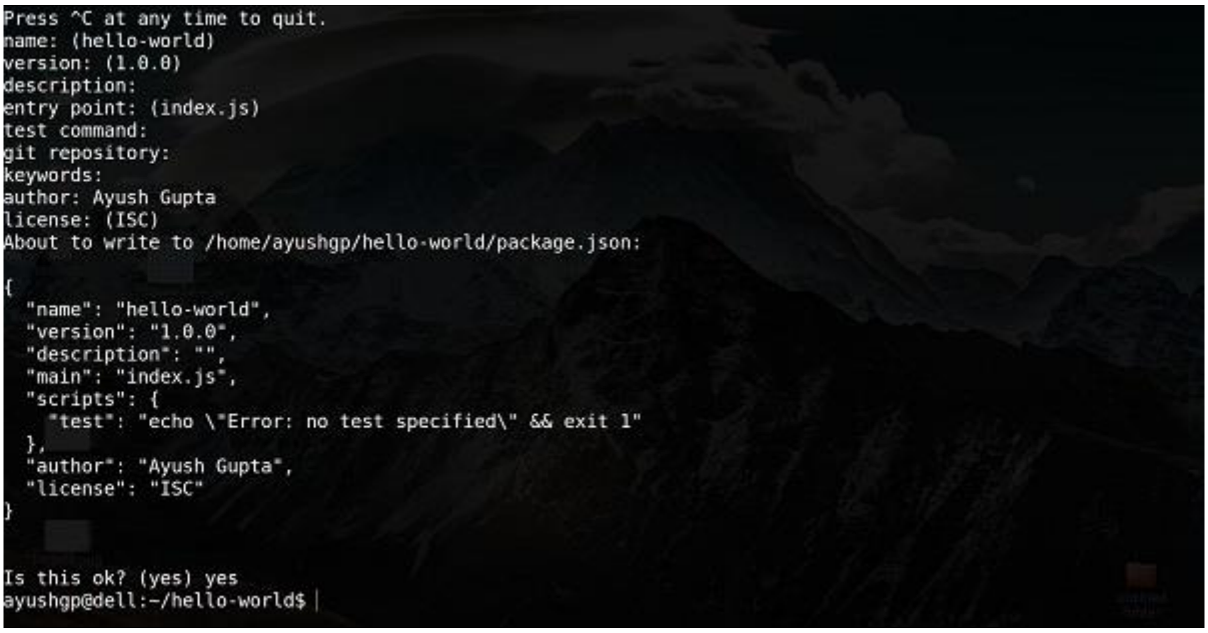


```
ayushgp@dell:~$ mkdir hello-world
ayushgp@dell:~$ cd hello-world/
ayushgp@dell:~/hello-world$
```

Step 2 – Now to create the package.json file using npm, use the following code.

```
npm init
```

It will ask you for the following information.



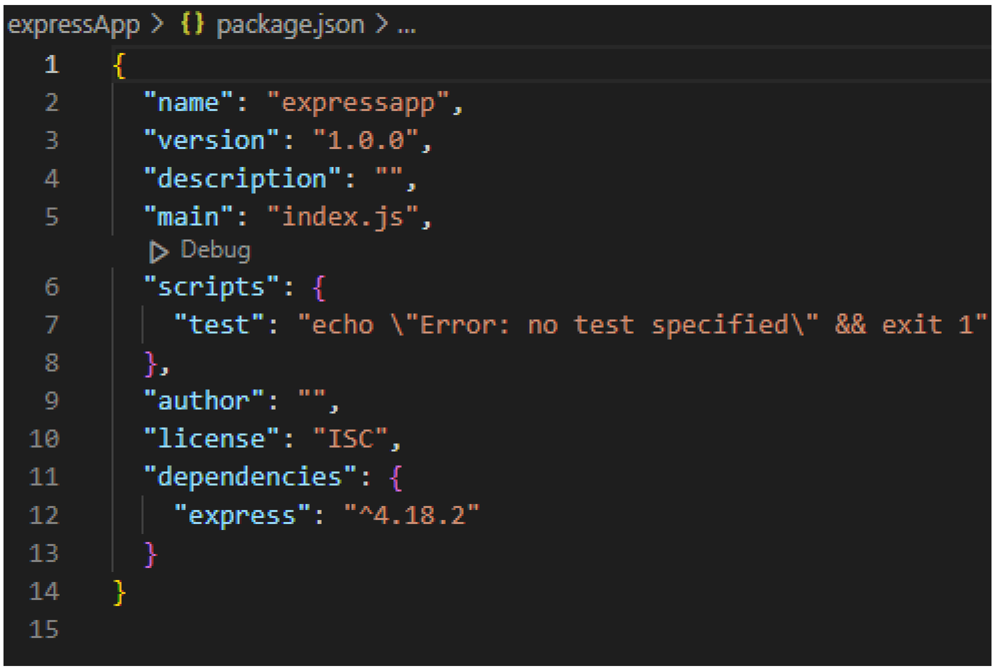
```
Press ^C at any time to quit.
name: (hello-world)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author: Ayush Gupta
license: (ISC)
About to write to /home/ayushgp/hello-world/package.json:

{
  "name": "hello-world",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Ayush Gupta",
  "license": "ISC"
}


Is this ok? (yes) yes
ayushgp@dell:~/hello-world$
```

Just keep pressing enter, and enter your name in the “author name” field.

Step 3 – Now we have our package.json file set up, we will further install Express. To install Express and add it to our package.json file, use the following command –

```
npm install --save express
```

To confirm that Express has been installed correctly, run the following code.



```
expressApp > {} package.json > ...
{
  "name": "expressapp",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  ▷ Debug
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

- ✓ This is all we need to start development using the Express framework.
- ✓ To make our development process a lot easier, we will install a tool from npm, nodemon. This tool restarts our server as soon as we make a change in any of our files, otherwise, we need to restart the server manually after each file modification.
- ✓ To install nodemon, use the following command –

```
npm install -g nodemon
```

Defining a route

- ✓ Routing is a method that refers to determining how an application responds to a client request to a particular path and a specific HTTP request method (GET, POST, and so on).
- ✓ In simple terms, routing is controlling which function gets invoked whenever the user navigates to a particular URL. In this context, URL refers to any path or route.

Defining Routing Methods

```
app.METHOD(PATH,CALLBACK)
```

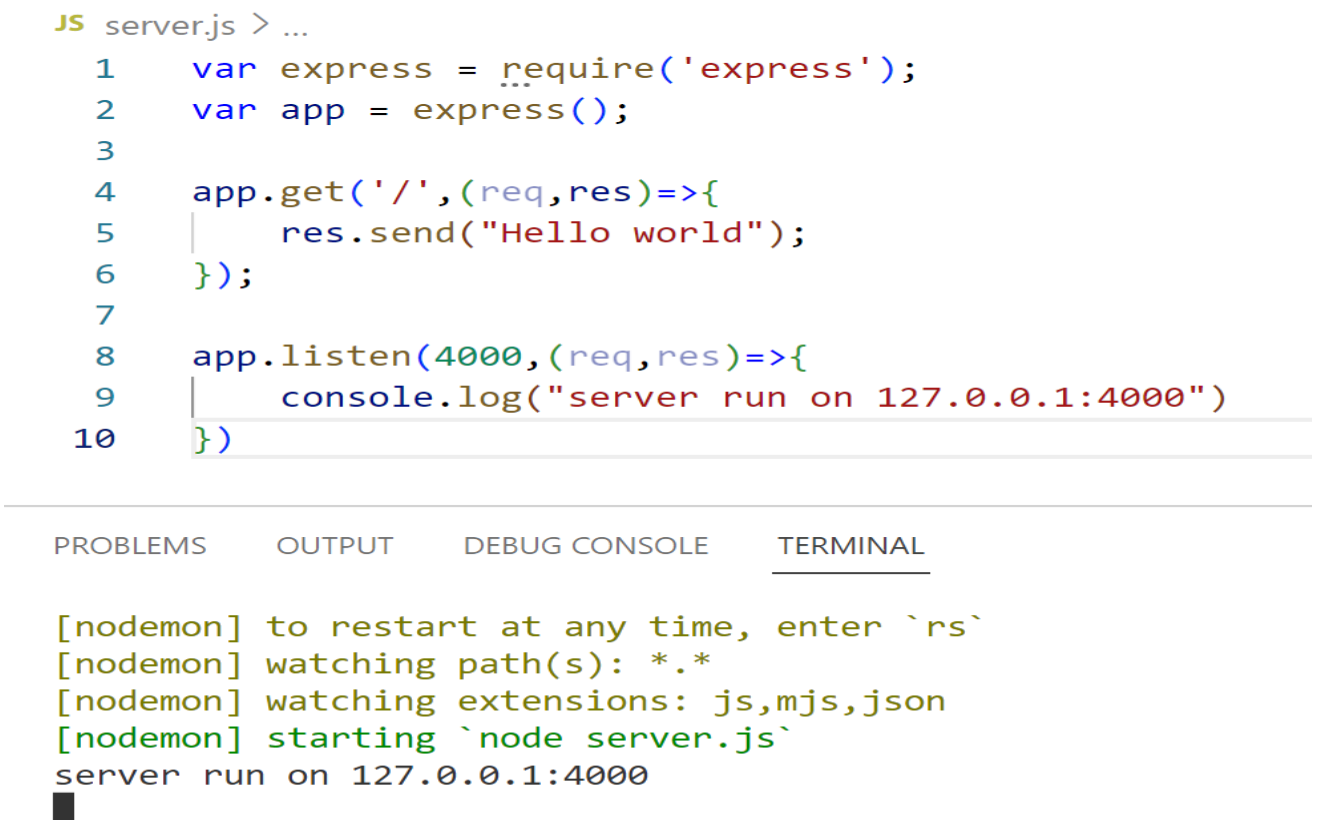
- ✓ This function tells the server, "If a user navigates to PATH, then perform the following CALLBACK function and perform an HTTPMETHOD request."
- ✓ Here, the most commonly used verbs in place of METHOD are:
- ✓ get: To handle GET requests (i.e. to request/GET data from a specified resource).
- ✓ post: To send data to a server to create/update a resource.
- ✓ put: To send data to a server to create/update a resource. The difference between POST and PUT requests is that the latter are idempotent. This means that PUT requests have no additional effect if they are called multiple times. In contrast, if you call a

POST method more than once, your program will have side effects. Therefore, bear in mind that POST requests are not to be called more than once.

- ✓ delete: For removal of a specified resource.

Basic Hello World

- ✓ As a basic example, let's define an HTTP GET method and display the words "Hello World" if the user navigates to the '/' path:

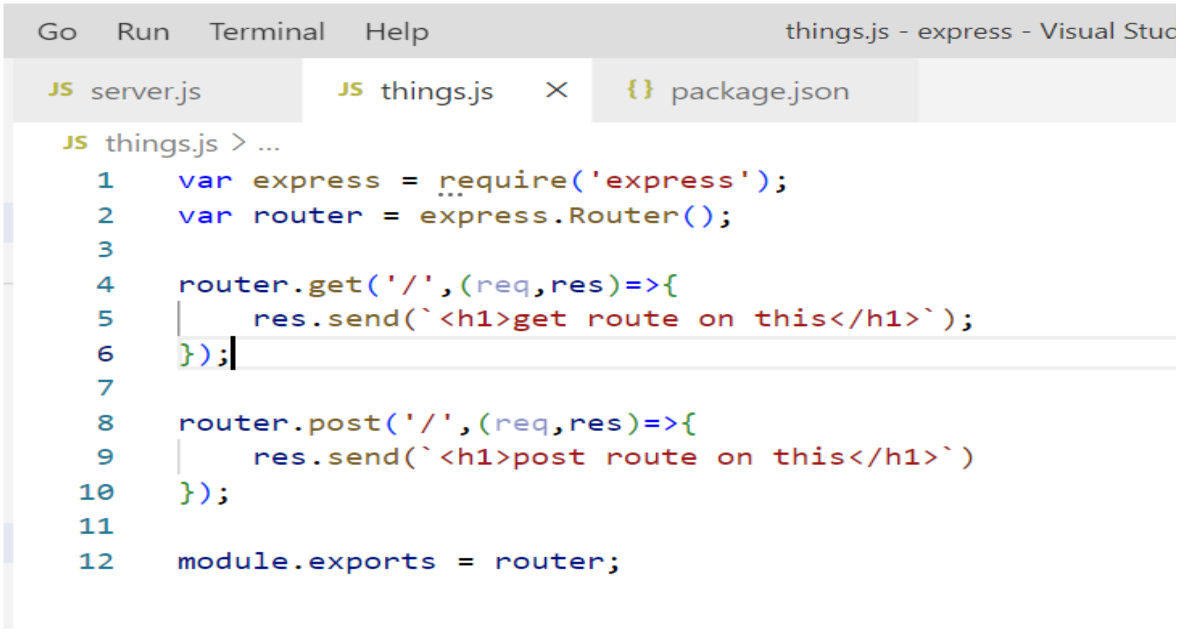


```
var express = require('express');
var app = express();

app.get('/',(req,res)=>{
    res.send("Hello world");
});

app.listen(4000,(req,res)=>{
    console.log("server run on 127.0.0.1:4000")
})
```

```
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
server run on 127.0.0.1:4000
```

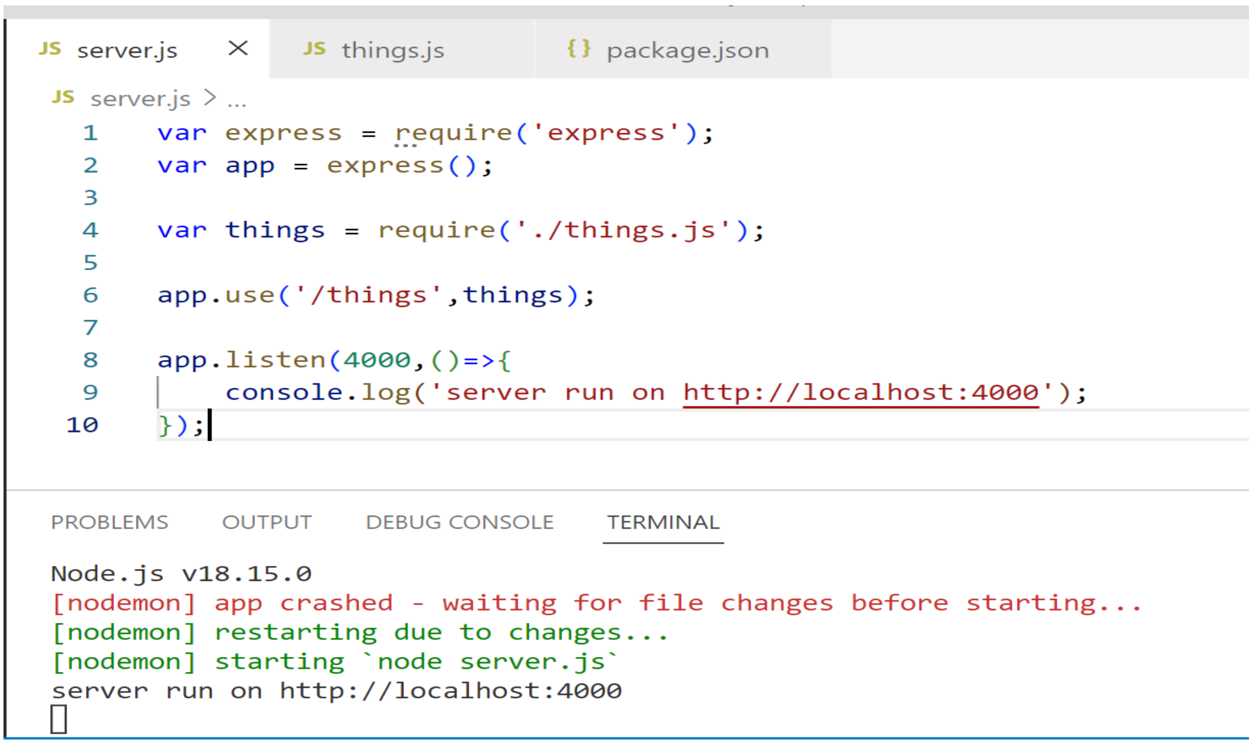
Routers

- ✓ To separate the routes from our main index.js file, we will use Express.Router.
- ✓ Create a new file called things.js and type the following in it.



```
var express = require('express');
var router = express.Router();

router.get('/',(req,res)=>{
    res.send(`<h1>get route on this</h1>`);
});

router.post('/',(req,res)=>{
    res.send(`<h1>post route on this</h1>`)
});

module.exports = router;
```

- ✓ Now to use this router in our index.js, type in the following before the app.listen function call.



server.js | things.js | package.json

server.js > ...

```
var express = require('express');
var app = express();

var things = require('./things.js');

app.use('/things',things);

app.listen(4000,()=>{
    console.log('server run on http://localhost:4000');
});
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
Node.js v18.15.0
[nodemon] app crashed - waiting for file changes before starting...
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
server run on http://localhost:4000
```

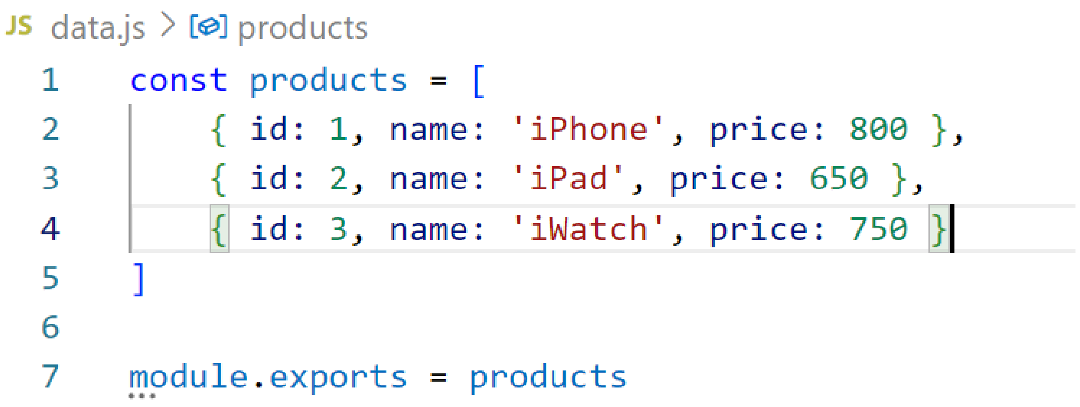
- ✓ The app.use() a function call on route '/things' and attaches the things router with this route.
- ✓ Now whatever requests our app gets at the '/things', will be handled by our things.js router.
- ✓ The '/' route in things.js is a subroutine of '/things'.
- ✓ Visit 127.0.0.1:3000/things/

Route Parameters

- ✓ router parameters are essentially variables derived from named sections of the URL.
- ✓ Express captures the value in the named section and stores it in the req.params property.

  Example

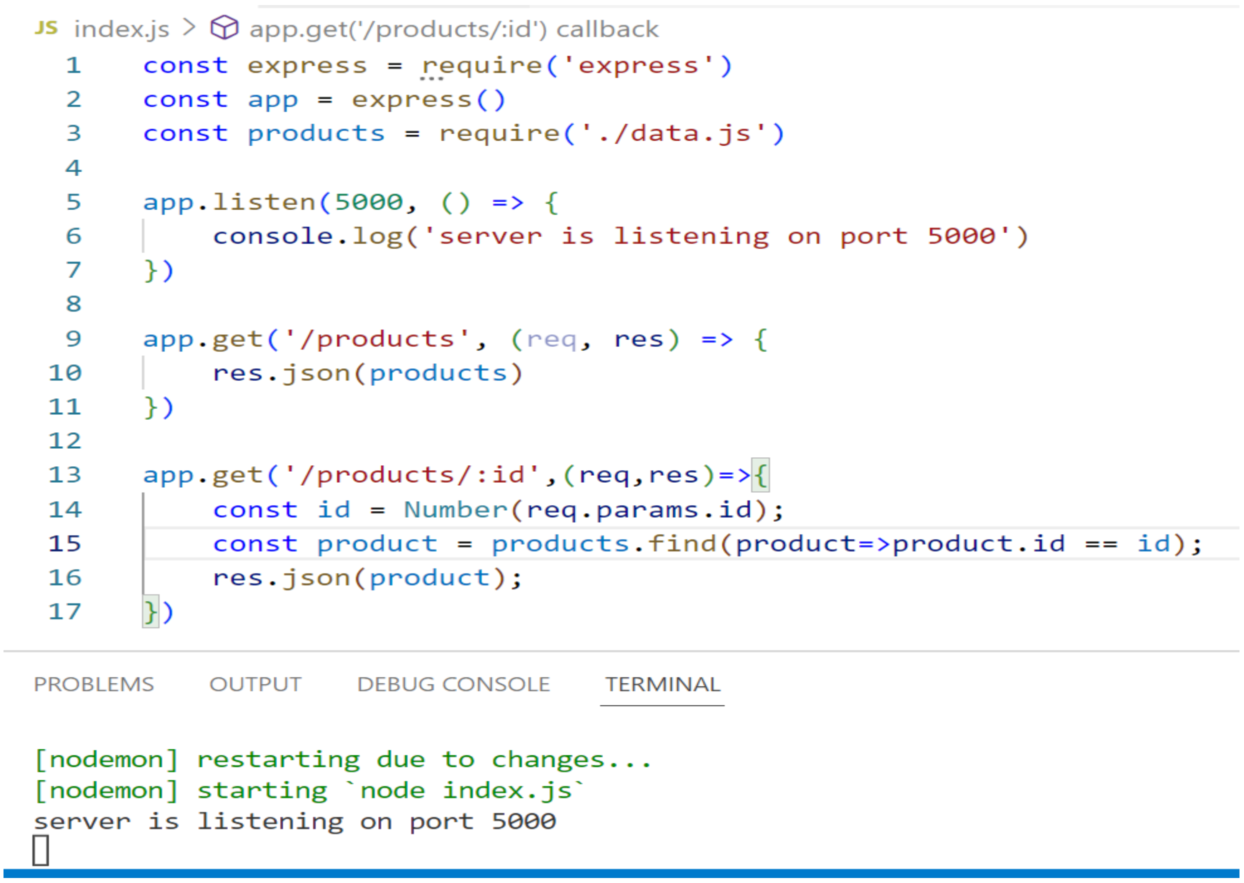
  Data.js



data.js > products

```
const products = [
    { id: 1, name: 'iPhone', price: 800 },
    { id: 2, name: 'iPad', price: 650 },
    { id: 3, name: 'iWatch', price: 750 }
]

module.exports = products
```

Index.js



```
const express = require('express')
const app = express()
const products = require('./data.js')

app.listen(5000, () => {
    console.log('server is listening on port 5000')
})

app.get('/products', (req, res) => {
    res.json(products)
})

app.get('/products/:id',(req,res)=>{
    const id = Number(req.params.id);
    const product = products.find(product=>product.id == id);
    res.json(product);
})
```

```
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
server is listening on port 5000
```

- ✓ What if you want to load only products with id 1.
- ✓ The convention wants to route to be something like: products/1

Query Parameters

- ✓ The Query String portion of a URL is part of the URL after the question mark ?. For example:

```
?answer=42
```

- ✓ Each key=value pair is called a *query parameter*. If your query string has multiple query parameters, they're separated by &. For example, the below string has 2 query parameters, a and b.

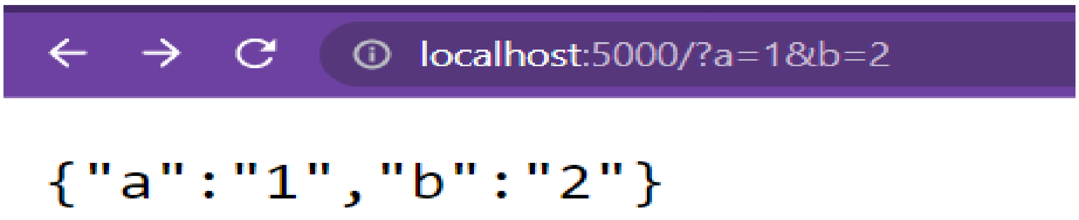
```
?a=1&b=2
```

- ✓ Express automatically parses query parameters for you and stores them on the request object as req.query.

Example:

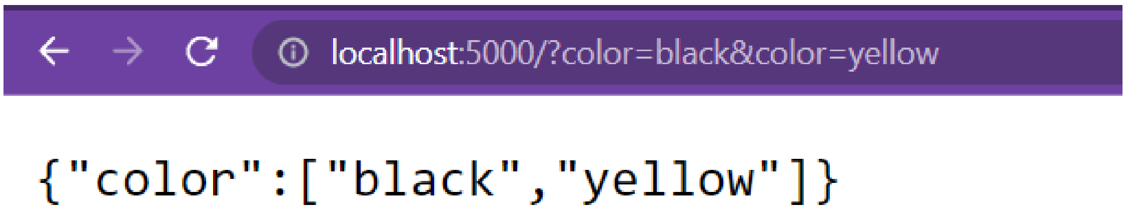
```
app.get('/',(req,res)=>{
    //req.query;
    res.json(req.query);
})
```

Sample output-1:



```
localhost:5000/?a=1&b=2

{"a":"1","b":"2"}
```

Sample output-2:



```
localhost:5000/?color=black&color=yellow

{"color":["black","yellow"]}
```

REST API with Node, Express, and MongoDB

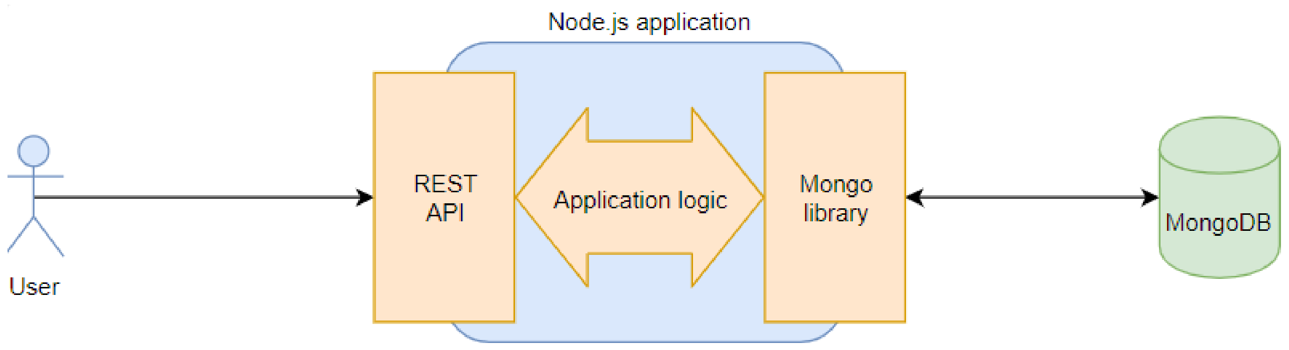
- ✓ **API** means **Application Programming Interface,** which is a set of clearly defined methods of communication between the front end and the database.
- ✓ **REST** which stands for **Representational State Transfer** is an architectural style for providing standards between computer systems on the web, making it easier for systems to communicate with each other.
- ✓ REST APIs mostly use **JSON** as the preferred choice for transferring data as they are easy to understand and readable.

we will use Node, Express, and MongoDB to create a REST API that would support the four operations — **GET, POST, PUT,** and **DELETE.**

1. **GET** — GET means to read the data. The function of this operation is to retrieve the data from the database and present it to the user.
2. **POST** — POST, as the name suggests, is used to post/add new data to the database. It allows users to add new data to the database.
3. **PUT** — PUT means to update the data already present in the database.
4. **DELETE** — It is used to delete any existing data from the database.

Architecture Overview

1. We will define a RESTful API to act as an interface between the user and our application
2. The Node.js application will host an HTTP server and the application logic of our APIs
3. We will use MongoDB as a database to store and retrieve our data.



Step 1: create a new Node.js project and initialize it npm.

Open your terminal or command prompt and run the following commands

```
mkdir my-express-app
cd my-express-app
npm init -y
```

Step 2: Install Express.js and any other dependencies you need for your project

```
npm install express mongoose
```

Step 3: Set up your MongoDB Atlas Cluster

1. Log in to your MongoDB Atlas account.
2. Create a new cluster or use an existing one.
3. Whitelist your IP address to allow connections to your cluster.
4. Create a MongoDB user with appropriate permissions for your database.

Step 4: create a web application project

1. HTML Frontend: HTML for the user interface.
2. Node.js with Express Backend: Handles HTTP requests, interacts with the database, and serves HTML files.
3. MongoDB Atlas: Cloud-hosted MongoDB database.

1. HTML Frontend (index.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Sample Form</title>
</head>
<body>
   <h1>Sample Form</h1>
   <form action="/submit" method="POST">
      <input type="text" name="name" placeholder="Enter your name" required>
      <button type="submit">Submit</button>
   </form>
</body>
</html>
```

2. Node.js with Express Backend (index.js)

Step 1: Require dependencies

```
const express = require('express');
const mongoose = require('mongoose');
const app = express();
```

Here, we import the express framework and mongoose library. We also create an instance of the Express application.

Step 2: Connect to MongoDB Atlas

```
const uri = 'YOUR_MONGODB_ATLAS_CONNECTION_URI';
mongoose.connect(uri,{dbName:"CT"}).then(() => {
     console.log('Atlas connection established');
     }).catch((err) => { console.error('Error connecting to
Atlas:', err.message);
     });
```

Replace 'YOUR_MONGODB_ATLAS_CONNECTION_URI' with your actual MongoDB Atlas connection URI. This code establishes a connection to MongoDB Atlas using Mongoose. It uses the connect method with the URI and some options. If the connection is successful, it logs a success message; otherwise, it logs an error message.

Step 3: Define MongoDB schema and model

```
const UserSchema = new mongoose.Schema({
    name: String
});
const User = mongoose.model('User', UserSchema);
```

Here, we define a simple schema for a user with a single field name. We then create a Mongoose model named User based on this schema.

Step 4: Middleware to parse form data

```
app.use(express.urlencoded({ extended: true }));
```

This line adds a middleware to parse incoming request bodies containing URL-encoded data. It allows our application to access form data submitted via POST requests.

Step 5: Serve HTML file

```
app.get('/', (req, res) => {
    res.sendFile(__dirname + '/index.html');
});
```

This route handler serves an HTML file located at /index.html when a GET request is made to the root URL (/). __dirname is a global variable representing the current directory path.

Step 6: Handle form submission

```
app.post('/submit', async (req, res) => {
    const { name } = req.body;
    try {
        const newUser = new User({ name });
        await newUser.save();
        res.send('Data saved successfully!');
    } catch (err) {
        res.status(500).send('Error saving data.');
    }
});
```

This route handler processes POST requests sent to /submit. It extracts the name field from the request body, creates a new User document using Mongoose, saves it to the database, and sends a response indicating success or failure.

Step 7: Start the server

```
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
   console.log(`Server is running on port ${PORT}`);
});
```

This code starts the Express server and listens on the specified port (or default port 3000 if not provided). It logs a message indicating that the server is running.

Fetch all documents:

```
app.get('/all', async (req, res) => {
   try {
      const allUsers = await users.find();
      res.json(allUsers);
   } catch (error) {
      console.error('Error retrieving users:', error.message);
      res.status(500).send('Error retrieving users');
   }
});
```