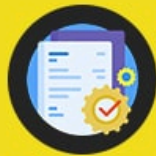
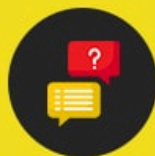




Advantages Of Express JS



Has Detailed
Documentation



Huge Supporting
Community



Supports Many
Third-Party Plugins



Very Simple & Fast



Powerful Routed API

<https://expressjs.com/>

About the Tutorial

Express.js, or simply Express, is a back end web application framework for building RESTful APIs with Node.js, released as free and open-source software under the MIT License. It is designed for building web applications and APIs. It has been called the standard server framework for Node.js

Table of Contents

1. EXPRESSJS - OVERVIEW

EXPRESSJS - ENVIRONMENT

2.

3. EXPRESSJS - HELLO WORLD

4. EXPRESSJS - ROUTING

5. EXPRESS JS - HTTP METHODS

6. EXPRESSJS - URL BUILDING

7. EXPRESSJS - MIDDLEWARE

8. EXPRESSJS - DATABASE

9. EXPRESSJS - AUTHENTICATION

10. EXPRESSJS - RESTFUL APIS

11. EXPRESSJS - ERROR HANDLING

1. EXPRESSJS – OVERVIEW

What is Express?

Express.js, or simply Express, is a back end web application framework for building RESTful APIs with Node.js, released as free and open-source software under the MIT License. It is designed for building web applications and APIs. It has been called the standard server framework for Node.js

Why Express?

Unlike its competitors like Ruby on Rails and Django, which have an opinionated way of building applications, Express has no "best way" to do something. It is very flexible and pluggable.

MongoDB and Mongoose

MongoDB is an open-source, document database designed for ease of development and scaling. This database is also used to store data.

Mongoose is a client API for **node.js** which makes it easy to access the MongoDB database from our Express application.

2. EXPRESSJS – ENVIRONMENT

Start developing and using the Express Framework. To start with, you should have the Node and the npm (node package manager) installed. If you don't already have these, go to the [Node setup](#) to install node js on your local system. Confirm that node and npm are installed by running the following commands in your terminal.

```
node --version  
npm --version
```

Node Package Manager (npm)

npm is the package manager for node. The npm Registry is a public collection of packages of open-source code for Node.js, front-end web apps, mobile apps, robots, routers, and countless other needs of the JavaScript community. npm allows us to access all these packages and install them locally. You can browse through the list of packages available on npm at [npmJS](#).

How to use npm?

There are two ways to install a package using npm: globally and locally.

- **Globally:** This method is generally used to install development tools and CLI based packages. To install a package globally, use the following code.

```
npm install -g <package-name>
```

- **Locally:** This method is generally used to install frameworks and libraries. A locally installed package can be used only within the directory it is installed. To install a package locally, use the same command as above without the **-g** flag.

```
npm install <package-name>
```

Whenever we create a project using npm, we need to provide a **package.json** file, which has all the details about our project. npm makes it easy for us to set up this file. Let us set up our development project.

Step 1: Start your terminal/cmd, create a new folder named hello-world and cd (create 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 at 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 express
```

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
```

You can now start working on Express.

3. EXPRESSJS – HELLO WORLD

We have set up the development, now it is time to start developing our first app using Express. Create a new file called **index.js** and type the following in it.

```
let express = require('express');  
let app = express();  
  
app.get('/', function(req, res){  
    res.send("Hello world!");  
});  
app.listen(3000);
```

Save the file, go to your terminal and type the following.

```
nodemon index.js
```

This will start the server. To test this app, open your browser and go to <http://localhost:3000> and a message will be displayed.

How the App Works?

The first line imports Express in our file, we have access to it through the variable Express. We use it to create an application and assign it to let app.

app.get(route, callback)

This function tells what to do when a **get** request at the given route is called. The callback function has 2 parameters, **request(req)** and **response(res)**. The request **object(req)** represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, etc. Similarly, the response object represents the HTTP response that the Express app sends when it receives an HTTP request.

res.send()

This function takes an object as input and it sends this to the requesting client. Here we are sending the string *"Hello World!"*.

app.listen(port, [host], [backlog], [callback])

This function binds and listens for connections on the specified host and port. Port is the only required parameter here, others are optional.

Argument	Description
port	A port number on which the server should accept incoming requests.
host	Name of the domain. You need to set it when you deploy your apps to the cloud.
backlog	The maximum number of queued pending connections. The default is 511.
callback	An asynchronous function that is called when the server starts listening for requests.

4.EXPRESSJS – ROUTING

Web frameworks provide resources such as HTML pages, scripts, images, etc. at different routes.

The following function is used to define routes in an Express application:

app.method(path, handler)

This METHOD can be applied to any one of the HTTP verbs - get, post, put, delete. An alternate method also exists, which executes independent of the request type.

Path is the route at which the request will run.

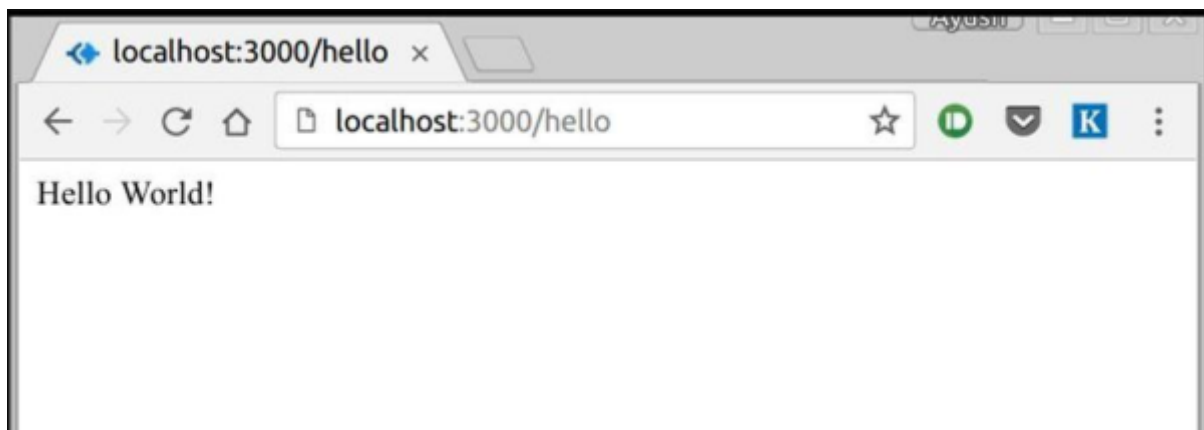
Handler is a callback function that executes when a matching request type is found on the relevant route. For example,

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

app.get('/hello', function(req, res){
    res.send("Hello World!");
});

app.listen(3000);
```

If we run our application and go to **localhost:3000/hello**, the server receives a get request at route **"/hello"**, our Express app executes the **callback** function attached to this route and sends **"Hello World!"** as the response.



We can also have multiple different methods on the same route. For example,

```
let express = require('express');
let app = express();
app.get('/mobile-data', function (req, res) {
    res.send("This is a message to get mobile Data");
});

app.get('/laptop-data', function (req, res) {
    res.send("This is a message to Get Laptop Data");
});
app.listen(3000);
```

A special method, **all**, is provided by Express to handle all types of http methods at a particular route using the same function. To use this method, try the following.

```
let express = require('express');
let app = express();
app.all('/test', function (req, res) {
    res.send("HTTP method doesn't have any effect on this route!");
});
app.listen(3000);
```

This method is generally used for defining middleware, which we'll discuss in the middleware chapter.

Routers

Defining routes like above is very tedious to maintain. 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.

routes.js

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

const router = express.Router();

// Define routes and middleware specific to this router

router.get('/', (req, res) => {

  res.send('Hello from the router!');

});

module.exports = router;

//export this router to use in our index.js module.exports =
router;
```

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

index.js

```
const express = require('express');
const app = express();
const port = 3000;

const router = require('./routes'); // path to your router file

app.use('/myrouter', router);

app.listen(port, () => {

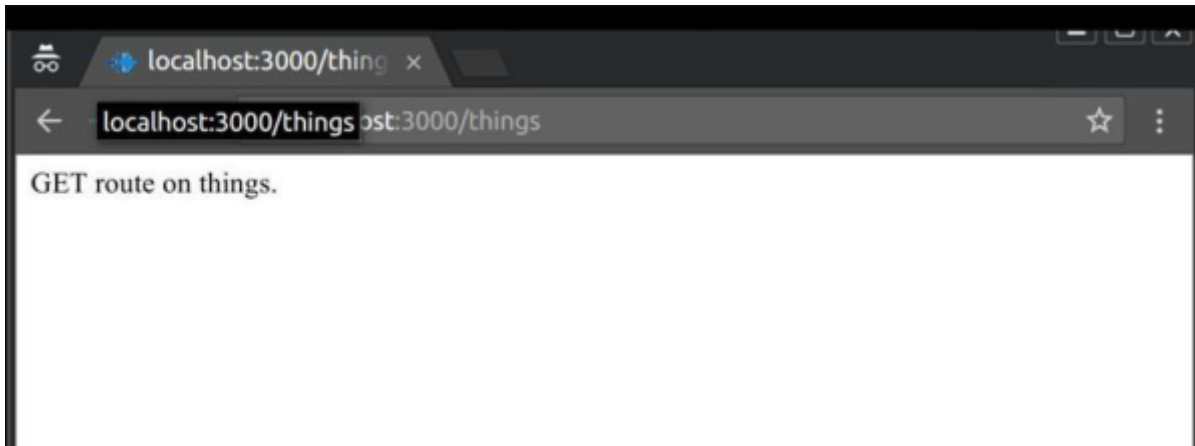
  console.log(`Server is running on port ${port}`);

});
```

goto - <http://localhost:3000/myrouter/>

The **app.use** function call on route **'/myrouter'** attaches the **'myrouter'** router with this route. Now whatever requests our app gets at the **'/myrouter'**, will be handled by our **routes.js** router. The **'/'** route in **routes.js** is actually a subroute of **'/myrouter'**. Visit **localhost:3000/myrouter/** and you will see the following output.

Routers are very helpful in separating concerns and keeping relevant portions of our code together. They help in building maintainable code. You should define your routes relating to an entity in a single file and include it using the above method in your **index.js** file.



5. EXPRESSJS – HTTP METHODS

The HTTP method is supplied in the request and specifies the operation that the client has requested. The following table lists the most used HTTP methods:

Method	Description
GET	The GET method requests a representation of the specified resource. Requests using GET should only retrieve data and should have no other effect.
POST	The POST method requests that the server accept the data enclosed in the request as a new object/entity of the resource identified by the URI.
PUT	The PUT method requests that the server accept the data enclosed in the request as a modification to an existing object identified by the URI. If it does not exist then the PUT method should create one.
DELETE	The DELETE method requests that the server delete the specified resource.

6. EXPRESSJS – URL BUILDING

We can now define routes, but those are static or fixed. To use the dynamic routes, we SHOULD provide different types of routes. Using dynamic routes allows us to pass parameters and processes based on them.

Here is an example of a dynamic route:

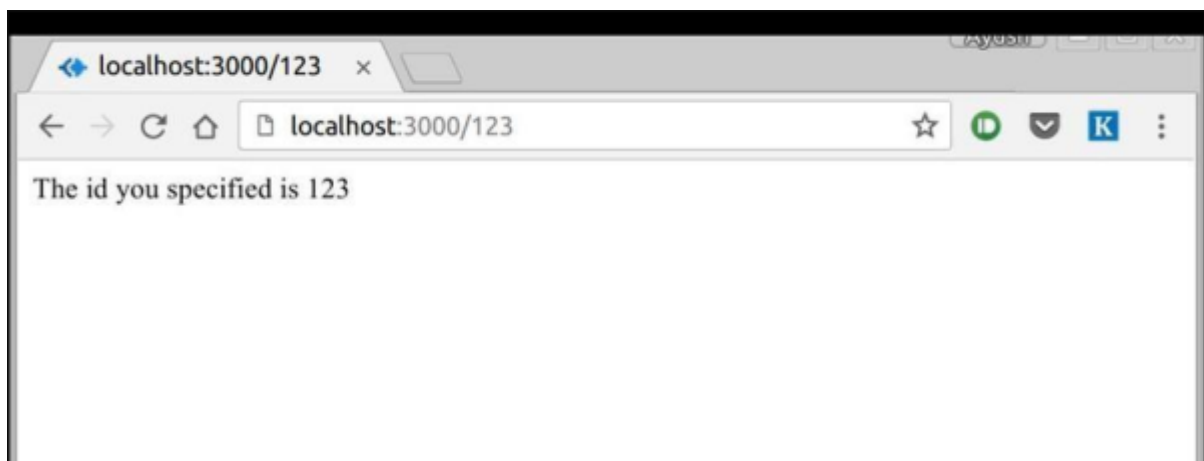
```
let express = require('express');

let app = express();

app.get('/:id', function (req, res) {
    res.send('The id you specified is ' + req.params.id);
});

app.listen(3000);
```

To test this go to <http://localhost:3000/123>. The following response will be displayed.



You can replace '123' in the URL with anything else and the change will reflect in the response. A more complex example of the above is:

6. EXPRESSJS – URL BUILDING

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

app.get('/men-casual/:name/:id', function (req, res) {
  res.send('name:' + req.params.name + ' and id: ' + req.params.id);
});
app.listen(3000, () => {
  console.log("Server is running on 3000")
});
```

//To Test above type : `http://localhost:3000/men-casual/footwear/bata_0123`

You can use the **req.params** object to access all the parameters you pass in the url. Note that the above 2 are different paths. They will never overlap. Also if you want to execute code when you get **'/things'** then you need to define it separately.

Pattern Matched Routes

You can also use **regex** to restrict URL parameter matching. Let us assume you need the **id** to be a 5-digit long number. You can use the following route definition:

```
let express = require('express');
let app = express();
app.get('/footwear/:id(\\d+)', (req, res) => {
  res.send('id: ' + req.params.id);
});
app.listen(3000);
```

Note that this will **only** match the requests that have a 5-digit long **id**. You can use more complex regex to match/validate your routes. If none of your routes match the request,

you'll get a **"Cannot GET <your-request-route>"** message as response. This message can be replaced by a 404 not found page using this simple route:

6. EXPRESSJS – URL BUILDING

Express Middleware

Middleware functions are functions that have access to the request object (req), the response object (res), and the **next** middleware function in the application's request-response cycle. These functions are used to modify req and res objects for tasks like parsing request bodies, adding response headers, etc.

A simple example of a middleware function in action -

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

//Middleware function (use , next) to log request protocol
app.use((req, res, next) => {
  console.log("A request for things received at " + Date.now());
  // next middleware Pass control to the next middleware
  next();// middleware function
});

// Route handler that sends the response
app.get('/users', (req, res) => {
  res.send('USERS');
});

app.listen(3000, () => {
  console.log("Server is running on 3000")
});
```


6. EXPRESSJS – URL BUILDING

Express js Model -

Models are responsible for creating a logical layer to communicate with the database. Model file in the express app contains information like key name, data type and constraints, relations, and triggers.

Models use the underlying MongoDB database to create and read documents.

syntax -

```
// Create a model based on the schema
const User = mongoose.model('User', userSchema);
```

Schema -

A schema is a structure of the data in the database. It defines the tables, fields, relationships, and way how data is stored.

For instance, if you are using a database system like MongoDB, Mongoose is a popular library for defining schemas. A Mongoose schema defines the shape of the documents within a collection in MongoDB.

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

const userSchema = new mongoose.Schema({
  name : {
    type : String,
    required : true,
  },
  email:{
    type : String,
    required : true,
    unique : true,
  },
  password : {
    type : String,
    required : true,
```

6. EXPRESSJS – URL BUILDING

```
    },  
  })
```

Example - (Passing hard code values in Model)

```
const mongoose = require('mongoose');  
// Define a schema  
const userSchema = new mongoose.Schema({  
  username: {  
    type: String, required: true  
  },  
  email: {  
    type: String, required: true, unique: true  
  },  
  password: {  
    type: String, required: true  
  },  
});  
  
// Create a model based on the schema  
const User = mongoose.model('User', userSchema);  
  
// Example usage  
const newUser = new User({  
  username: 'john_doe',  
  email: 'john@example.com',  
  password: 'securepassword',  
});  
  
// Save the user to the database  
newUser.save()  
  .then(() => {  
    console.log('User saved successfully');  
  })  
  .catch((err) => {  
    console.error('Error saving user:', err);  
  });
```

6. EXPRESSJS – URL BUILDING

why Model -

Models handle data: They are responsible for interacting with databases, handling data validation, and performing any necessary business logic.

Data storage and retrieval: Models are often used to interact with databases to store and retrieve data. This can involve creating, updating, deleting, **(CRUD)** and querying records in a database. Popular databases used with Express.js include MongoDB, MySQL, and PostgreSQL.

Encapsulation of logic: Business logic, such as data validation and manipulation, is encapsulated within models.

Reusable components: Models can be reused across different parts of your application.

Express Database

Step 1: Create an Express Application

Here we are going to create an express application because all of our work will be going to execute inside express.

Now create an empty .js file (let's name it app.js), This would be our folder structure

```
//Importing express module
const express = require('express');

const app = express();
const PORT = 3000;

app.listen(PORT, (error) =>{
  if(!error)
```

6. EXPRESSJS – URL BUILDING

```
    console.log("Server is Successfully Running, "
        + "and App is listening on port "+ PORT)
    else
        console.log("Error occurred, server can't start", error);
    }
};
```

Step 2: Run the server

To confirm whether our server is working or not. Write this command in the terminal to start the express server. The successful start of the server denotes that our express app is ready to listen to the connection on the specified path(localhost:3000 in our example).

```
>node app.js
```

Step 3: Integrate Database

Now we will integrate the database with express. But before that, we have to choose one of the database options i.e. MongoDB, MySQL, PostgreSQL,

Redis, SQLite, and Cassandra, etc.

MongoDB and MySQL are the most popular and used by numerous developers, we will be using MongoDB

step 1 -

login to MongoDB Atlas -> click on Connect Button

step 2 -



choose below option

6. EXPRESSJS – URL BUILDING

Connect to BlogCluster

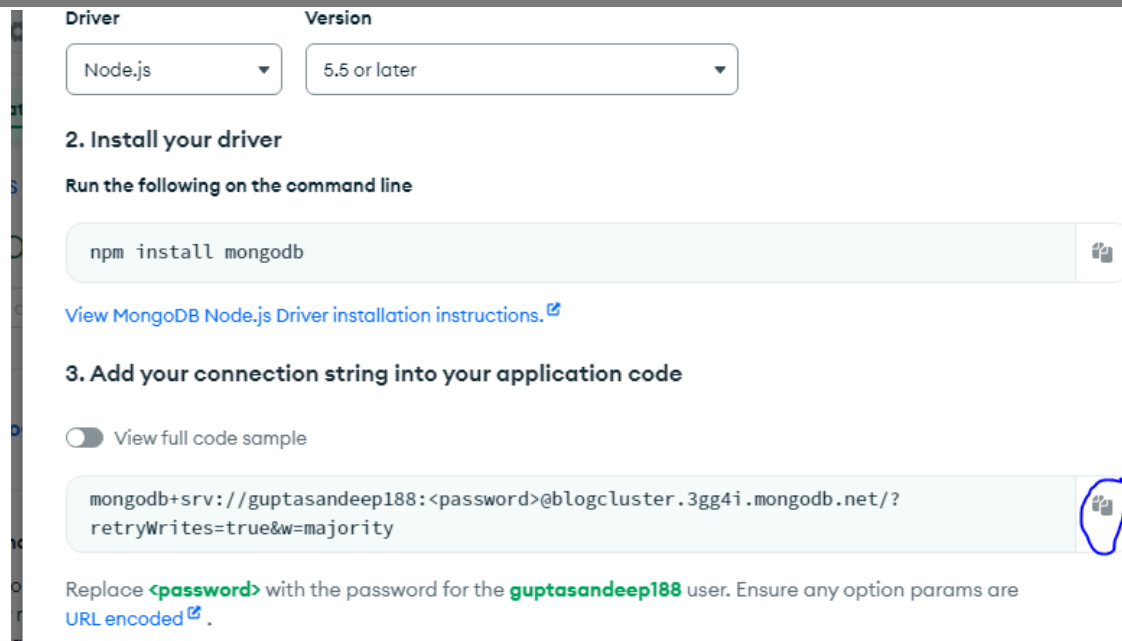


Connect to your application

 **Drivers** Access your Atlas data using MongoDB's native drivers (e.g. Node.js, Go, etc.) 

copy the connection string and paste inside `mongoose.connection()`

6. EXPRESSJS – URL BUILDING



Driver: Node.js | Version: 5.5 or later

2. Install your driver

Run the following on the command line

```
npm install mongodb
```

[View MongoDB Node.js Driver installation instructions.](#)

3. Add your connection string into your application code

☐ View full code sample

```
mongodb+srv://guptasandeep188:<password>@blogcluster.3gg4i.mongodb.net/?
retryWrites=true&w=majority
```

Replace **<password>** with the password for the **guptasandeep188** user. Ensure any option params are [URL encoded](#).

MongoDB

Install **mongoose**, a package built on the `mongodb` native driver, to interact with the MongoDB instance and model the data from express/node application.

```
>npm install mongoose
```

Explanation -

First of all we will import the mongoose module and then later we call the

connect method, it accepts a connection string, an object with some configurations, The "ExpressIntegration" written inside the connection string is the random name of the Database, the rest all stuff comes under syntax. This method provides a promise in return that's why we are using them and catching blocks. On the successful connection with the database, it will call the listen to method with the app object to start the express server otherwise simply will execute a console log on failure.

app.js

```
const express = require("express");
const mongoose = require("mongoose")
const app = express();
const PORT = 3000;
```

6. EXPRESSJS – URL BUILDING

```
const MONGO_URI =
"mongodb+srv://guptasandeep188:guptasandeep188@blogcluster.3gg4i.mongodb
.net/?retryWrites=true&w=majority";
//Connection to the mongodb database
mongoose.connect(MONGO_URI)
.then((conn)=>{
  console.log("MongoDB cluster is not connected. Cluster Name = ",
conn.connection.host)
  app.listen(PORT, ()=>{
    console.log("Database connection is Ready "
+ "and Server is Listening on Port ", PORT);
  })
})
.catch((err)=>{
  console.log("AN error has been occurred while"
+ " connecting to database.");
})
```

Output:

Start the server with the `node app.js` command and this will be output in the terminal, which means we are successfully able to integrate the MongoDB database.

Express Authentication

How to implement JWT authentication in Express.js app ?

JSON Web Token

A JSON web token(JWT) is a JSON Object which is used to securely transfer information over the web(between two parties). It is generally used for authentication systems and can also be used for information exchange.

This is used to transfer data with encryption over the internet also these tokens can be more secured by using an additional signature.

6. EXPRESSJS – URL BUILDING

Below is the sample example of JSON Web Token.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiI2MTU0OWIwMTIwMWUyZjMzZWE3NmFkZjYiLCJlbWFpbCI6InNtdHdpbmtsZTQ1MkNnbWFpbC5jb20iLCJpYXQiOiJE2MzI5MzQ2NTgsImV4cCI6MTYzMjkzODI1OH0. _oHr3REme2pjDDdRliArAeVG_HuimbdM5suTw8HI7uc
```

Implementation of JWT authentication: JWT is much popular for authentication and authorization via HTTP. These tokens can be used as credentials to grant permission to access server resources.

Why do we need JWT in the authentication:

Features -

1. Compact and Self-Contained.
2. Stateless and Scalable
3. Authentication and Authorization: JWTs are commonly used for user authentication. After a user logs in, the server can generate a JWT and send it to the client. The client includes this token in subsequent requests, allowing the server to verify the user's identity. Additionally, JWTs can carry claims (pieces of information) about the user, enabling authorization checks.
4. Standardized and Widely Supported.

Step 1: Install JWT Package.

```
>npm install jsonwebtoken
```

postAPIWithJwt.js

```
let express = require("express");
let mongoose = require("mongoose");
let jwt = require("jsonwebtoken");
let mongodb_conn =
  "mongodb+srv://guptasandeep188:guptasandeep188@blogcluster.3gg4i.mongodb
  .net/?retryWrites=true&w=majority"
let app = express();
let secret_key = "sec@123"; // must parameter

let verifyToken = (req, resp, next) => {
  let token = req.header("Authorization");
  if (!token) {
    //Status 401, The client request has not been completed because
    //it lacks valid authentication credentials for the requested
```


6. EXPRESSJS – URL BUILDING

```
resource.  
    return resp.status(401).json({ message: "No token is provided"  
  })  
}  
  
/*jwt.verify() is a function used to verify the integrity and  
authenticity of a JSON Web Token (JWT).  
When you receive a JWT, it's crucial to ensure that it hasn't been  
tampered with and that  
it was indeed signed by a trusted entity.  
The jwt.verify() function is responsible for performing this  
verification process.  
*/  
jwt.verify(token, secret_key, (err, data) => {  
  if (err) {  
    return resp.status(401).json({ message: "Invalid Token" })  
  }  
  req.user = data;  
  next();  
})  
}  
  
app.post('/login', (req, resp) => {  
  let user = {  
    email: "abc@gmail.com",  
    username: "abc"  
  };  
  /*  
  jwt.sign() is a function commonly used in web development to  
generate JSON Web Tokens (JWTs).  
  JWTs are a compact, URL-safe means of representing claims to be  
transferred between two parties.  
  These tokens are often used for authentication and authorization  
purposes in web applications.  
  */  
  
  jwt.sign({ user },  
    secret_key,
```

6. EXPRESSJS – URL BUILDING

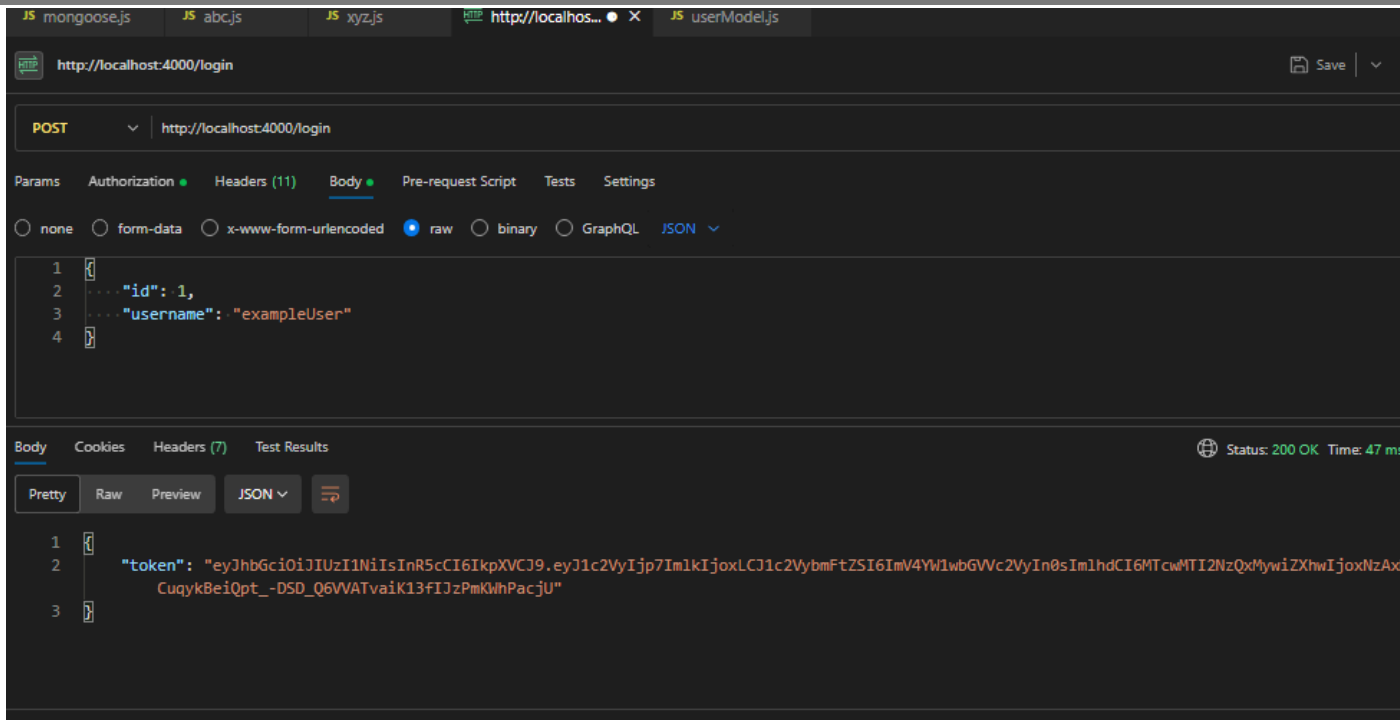
```
    { expiresIn: "1h" },
    (err, token) => {
      if (err) {
        return resp.status(401).json({ message: "Not able to
create token" })
      }
      resp.json({ token })
    })
  });

app.get('/userprofile', verifyToken, (req, resp) => {
  resp.json({ message: "your details are = ", user: req.user })
})

mongoose.connect(mongodb_conn)
  .then((conn) => {
    console.log("cluster name =", conn.connection.host);
    app.listen(4000, () => {
      console.log("server is running on port 4000")
    })
  })
  .catch(err => console.log(err))
```

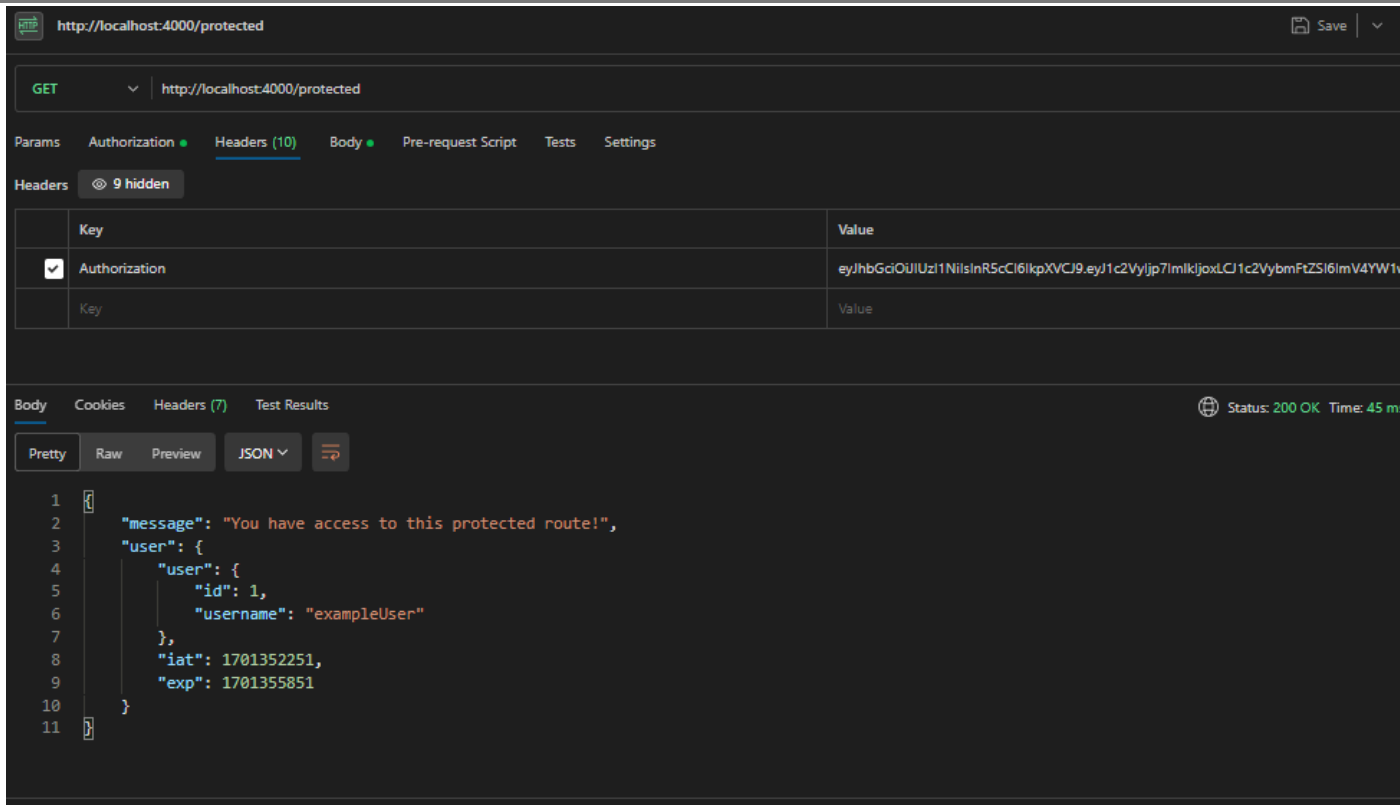
Check in Postman-

6. EXPRESSJS – URL BUILDING



Get Request -

6. EXPRESSJS – URL BUILDING



Flow Explanation -

We have imported the `express`, `mongoose`, and `jsonwebtoken` in the first line, and also the `User` model because that would be necessary to interact with the database. In the next line, we have called the `express` method which returns an app with which we can configure our server. We are using

`express.json()` middleware in the starting because the server can identify the incoming requests as JSON objects.

After that we have created routes for login.

Inside the login route, we have extracted details like id and user name to register users in the database by the mongoose.

Finally, we have created a token with the 1-hour expiry by providing payload as user id and user name because only this is sufficient to extract the user information. The sign method accepts payload, secret jwt key, and expiry time then generates a token.

jwt.sign() is a function commonly used in web development to generate JSON Web Tokens (JWTs). JWTs are a compact, URL-safe means of representing claims to be transferred between two parties. These tokens are often used for authentication and authorization purposes in web applications.

6. EXPRESSJS – URL BUILDING

jwt.verify() is a function used to verify the integrity and authenticity of a JSON Web Token (JWT). When you receive a JWT, it's crucial to ensure that it hasn't been tampered with and that it was indeed signed by a trusted entity. The `jwt.verify()` function is responsible for performing this verification process.

Output: Testing API with the Postman, provided the data for login in the request body and finally, got our token with some other details.

Signup API with dynamic data and JWT

index.js

```
// Importing modules
const express = require("express");
const mongoose = require("mongoose");
const jwt = require("jsonwebtoken");
const User = require("../signupUserModel");
const app = express();
app.use(express.json());

let secret_key = "secret@123"

let authenticateToken = (req, resp, next) => {
  let token = req.header("Authorization");
  if (!token) {
    return resp.status(401).json({ message: "No token is provided" })
  }
  jwt.verify(token, secret_key, (err, data) => {
    if (err) {
      return resp.status(401).json({ message: "Invalid Token" })
    }
    req.user = data;
    next();
  })
}
```

6. EXPRESSJS – URL BUILDING

```
// Handling post request
app.post("/signup", (req, res, next) => {
  const { name, email, password } = req.body;
  const newUser = User({
    name,
    email,
    password,
  });
  try {
    newUser.save();
    console.log('newUser', newUser);
  } catch {
    const error = new Error("Error! Something went wrong.");
    return next(error);
  }
  let token;
  try {
    token = jwt.sign(
      {
        name: newUser.name,
        email: newUser.email,
        password: newUser.password
      },
      secret_key,
      { expiresIn: "1h" }
    );
  } catch (err) {
    const error = new Error("Error! Something went wrong.");
    console.log('error', error);
    return next(error);
  }
  res
    .status(201) //request has succeeded and has been created.
    .json({
      success: true,
      data: {
        name: newUser.name,
        email: newUser.email,
```

6. EXPRESSJS – URL BUILDING

```
        password: newUser.password,
        token: token
      }
    });
  });
// GET API , authenticateToken
app.get('/userinfo', authenticateToken, (req, res) => {
  res.json({ message: "your details are = ", user: req.user });
});

//Connecting to the database
mongoose
  .connect("mongodb+srv://guptasandeep188:guptasandeep188@blogcluster.3gg4i.mongodb.net/?retryWrites=true&w=majority")
  .then((conn) => {
    console.log("cluster =" + conn.connection.host);
    app.listen("4000", () => {
      console.log("Server is listening on port 4000");
    });
  })
  .catch((err) => {
    console.log("Error Occurred", err);
  });
});
```

userModel.js

```
// user.js

const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true
  },
  email: {
```

6. EXPRESSJS – URL BUILDING

```
    type: String,
    required: true,
    unique: true
  },
  password: {
    type: String,
    required: true
  }
});

const User = mongoose.model('User', userSchema);
module.exports = User;
```

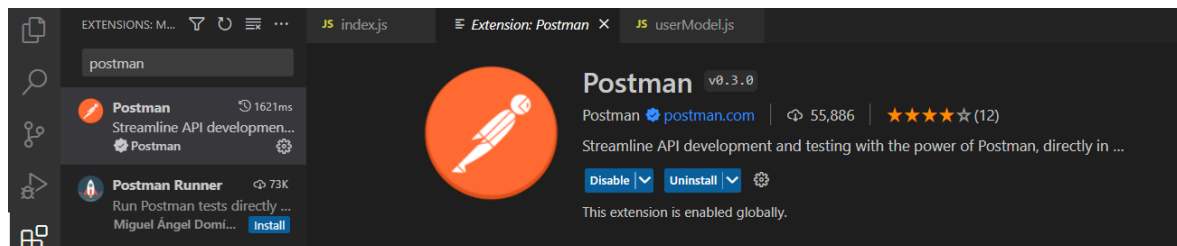
output -

run index.js

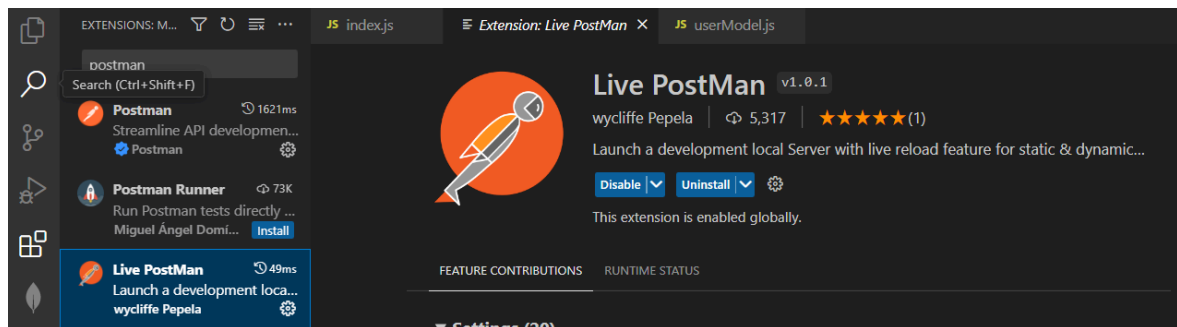
> node index.js

open PostMan in VS Code (Install if not installed both)

Postman -



Live Postman -



select Method : Post

URL : <http://localhost:4000/signup>

select Body

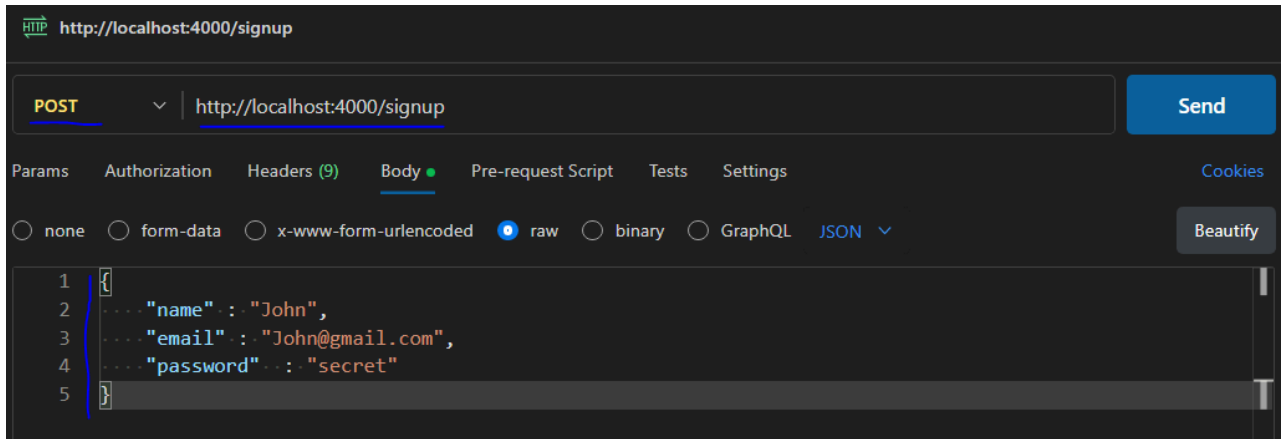
pass below data

```
{
```

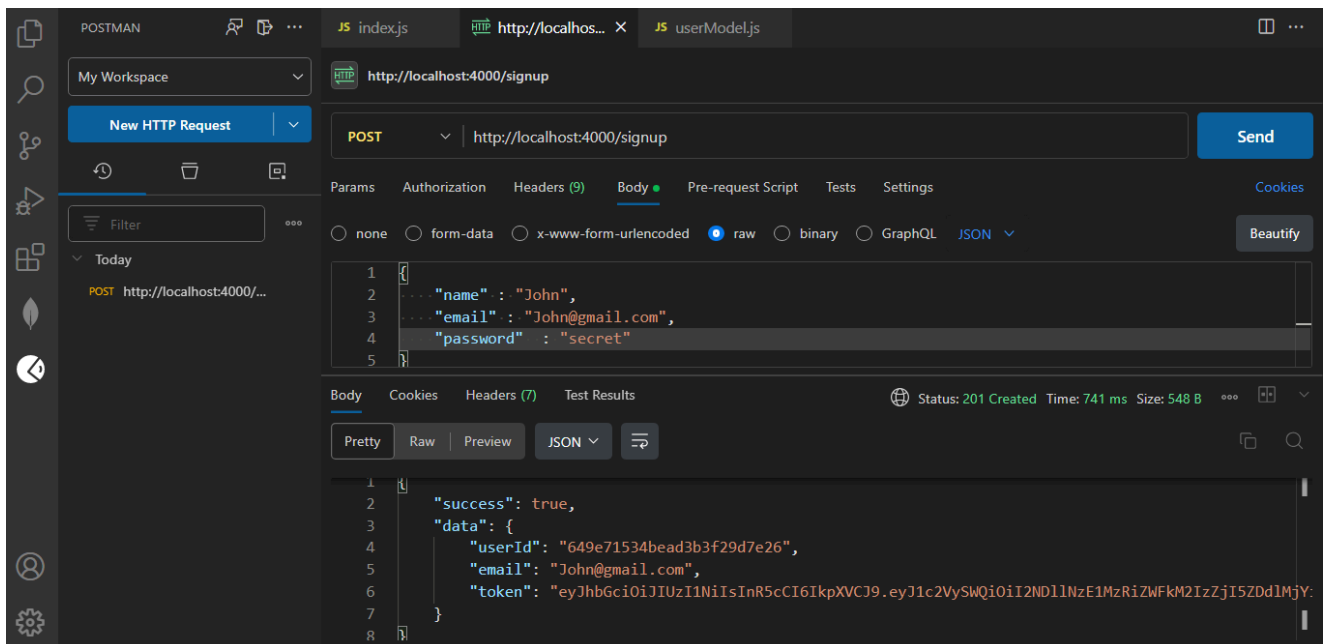

6. EXPRESSJS – URL BUILDING

```
"name" : "John",
"email" : "John@gmail.com",
"password" : "secret"
}
```

In Dropdown Select JSON.



Click on send and you will get below response -



6. EXPRESSJS – URL BUILDING

Flow Explanation -

We have imported the `express`, `mongoose`, and `jsonwebtoken` in the first line, and also the `User` model because that would be necessary to interact with the database. In the next line, we have called the `express` method which returns an app with which we can configure our server. We are using

`express.json()` middleware in the starting because the server can identify the incoming requests as JSON objects.

After that we have created routes for `signup`.

Inside the `signup` route, we have extracted details like `name`, `email`, and `password` to register users in the database, with the `save` method provided

by the `mongoose`.

Finally, we have created a token with the 1-hour expiry by providing payload as `user id` and `email` because only this is sufficient to extract the user information. The `sign` method accepts payload, secret `jwt` key, and expiry time then generates a token.

Output:

Testing API with the Postman, provided the data for `signup` in the request body and finally, got our token with some other details.

Restful API

An API is always needed to create mobile applications, single page applications, use AJAX calls and provide data to clients. A popular architectural style of how to structure and name these APIs and the endpoints is called REST (Representational Transfer State).

RESTful URIs (Uniform Resource Indicator) and methods provide us with almost all information we need to process a request. The table given below summarizes how the various verbs should be used and how URIs should be named.

6. EXPRESSJS – URL BUILDING

Method	URI	Details	Function
GET	/movies	Safe, cachable	Gets the list of all movies and their details
GET	/movies/1234	Safe, cachable	Gets the details of Movie id 1234
POST	/movies	N/A	Creates a new movie with the details provided. Response contains the URI for this newly created resource.
PUT	/movies/1234	Idempotent	Modifies movie id 1234(creates one if it doesn't already exist). Response contains the URI for this newly created resource.
DELETE	/movies/1234	Idempotent	Movie id 1234 should be deleted, if it exists. Response should contain the status of the request.
DELETE or PUT	/movies	Invalid	Should be invalid. DELETE and PUT should specify which resource they are working on.