

What is Express?

Express.js is a small framework that works on top of Node.js web server functionality to simplify its APIs and add helpful new features. It makes it easier to organize your application's functionality with middleware and routing. It adds helpful utilities to Node.js HTTP objects and facilitates the rendering of dynamic HTTP objects.

Why Express above node?

Express is a small framework that sits on top of Node JS's web server functionality to simplify its APIs and add helpful new features. It makes it easier to organize your application's functionality with middleware and routing. It adds helpful utilities to Node JS's HTTP objects.

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 our database from our Express application.

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 -y
```

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 --save express
```

To make our development process a lot easier, we will install a tool from npm, nodemon.

```
npm install -g nodemon
```

You can now start working on Express.

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.

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

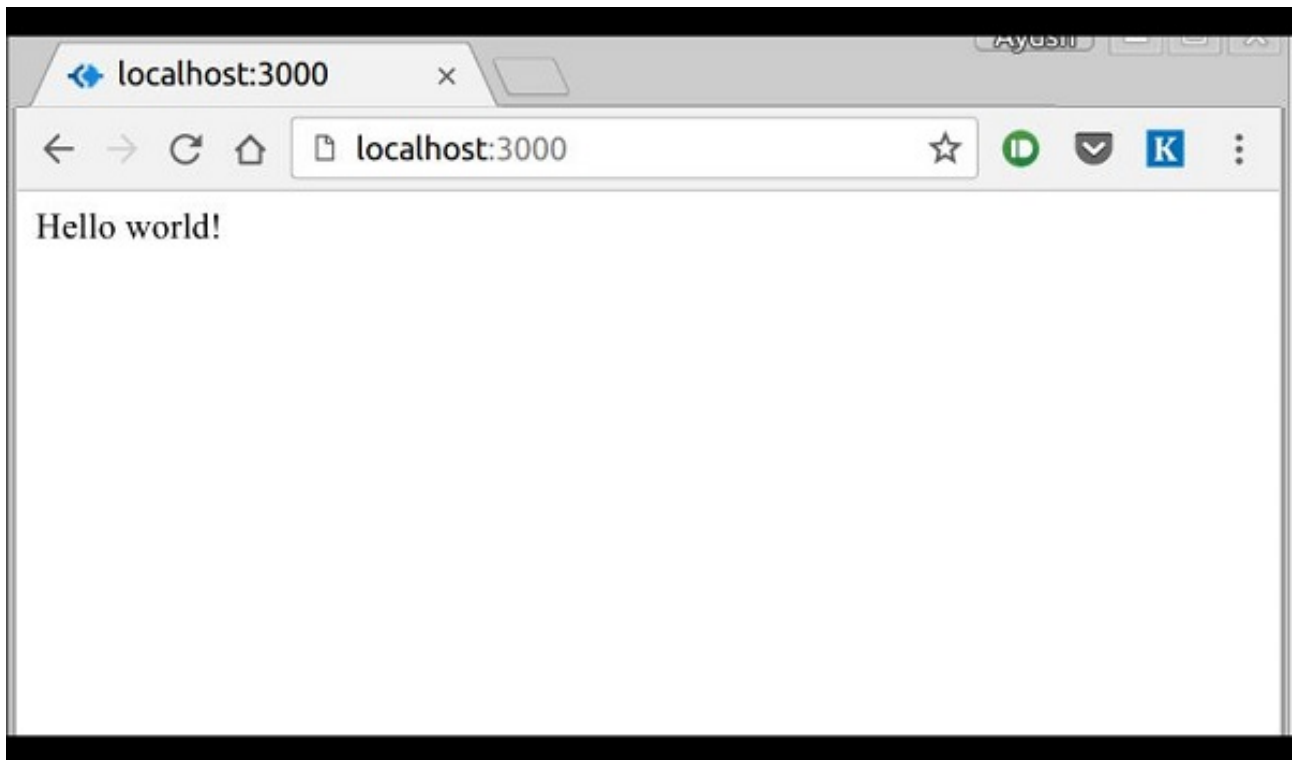
app.get('/', function(req, res){
  res.send("Hello world!");
});
```

```
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 as in the following screenshot.



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 var 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.

S.No.	Argument & Description
1	port A port number on which the server should accept incoming requests.
2	host Name of the domain. You need to set it when you deploy your apps to the cloud.
3	backlog The maximum number of queued pending connections. The default is 511.
4	callback An asynchronous function that is called when the server starts listening for requests.

`app.method(path, handler)`

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

Path is the route at which the request will run.

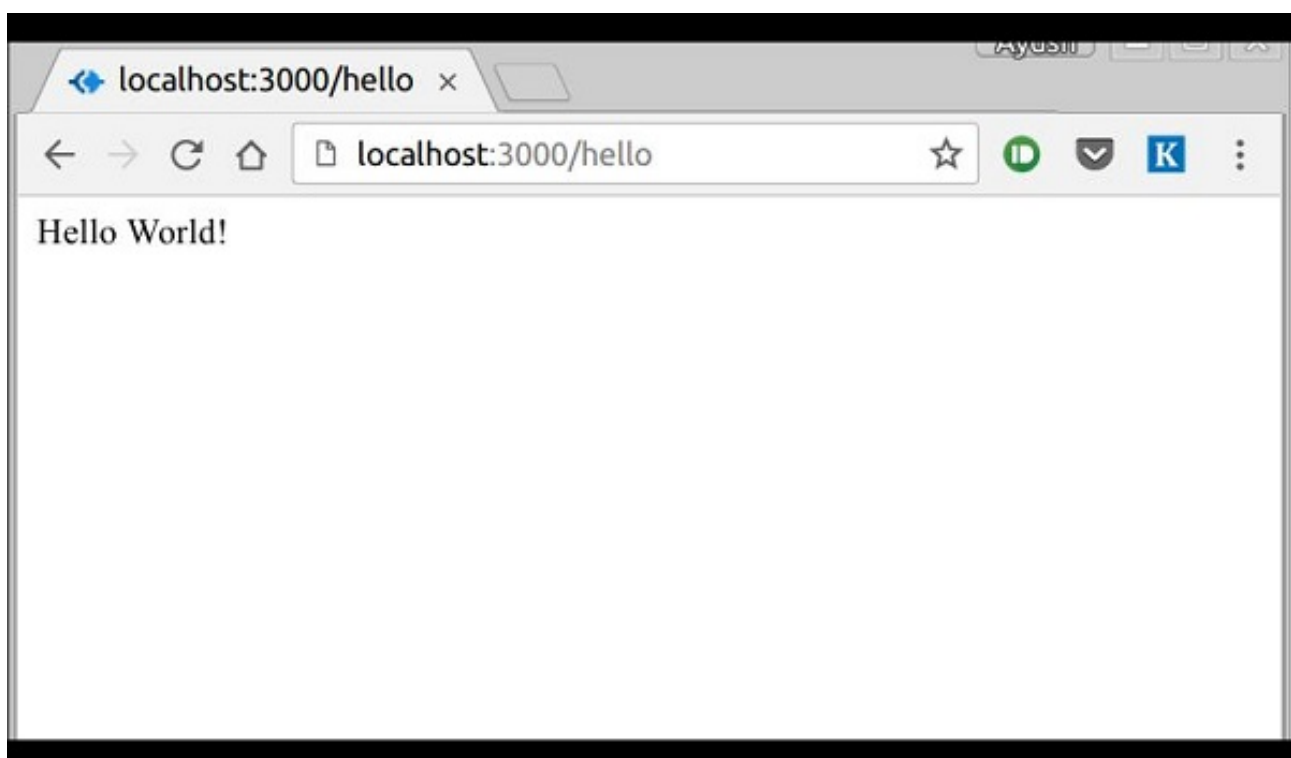
Handler is a callback function that executes when a matching request type is found on the relevant route. For example,

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



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.

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

router.get('/', function(req, res){
  res.send('GET route on things.');
```

```
});
router.post('/', function(req, res){
  res.send('POST route on things.');
```

```
});
```

```
//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.

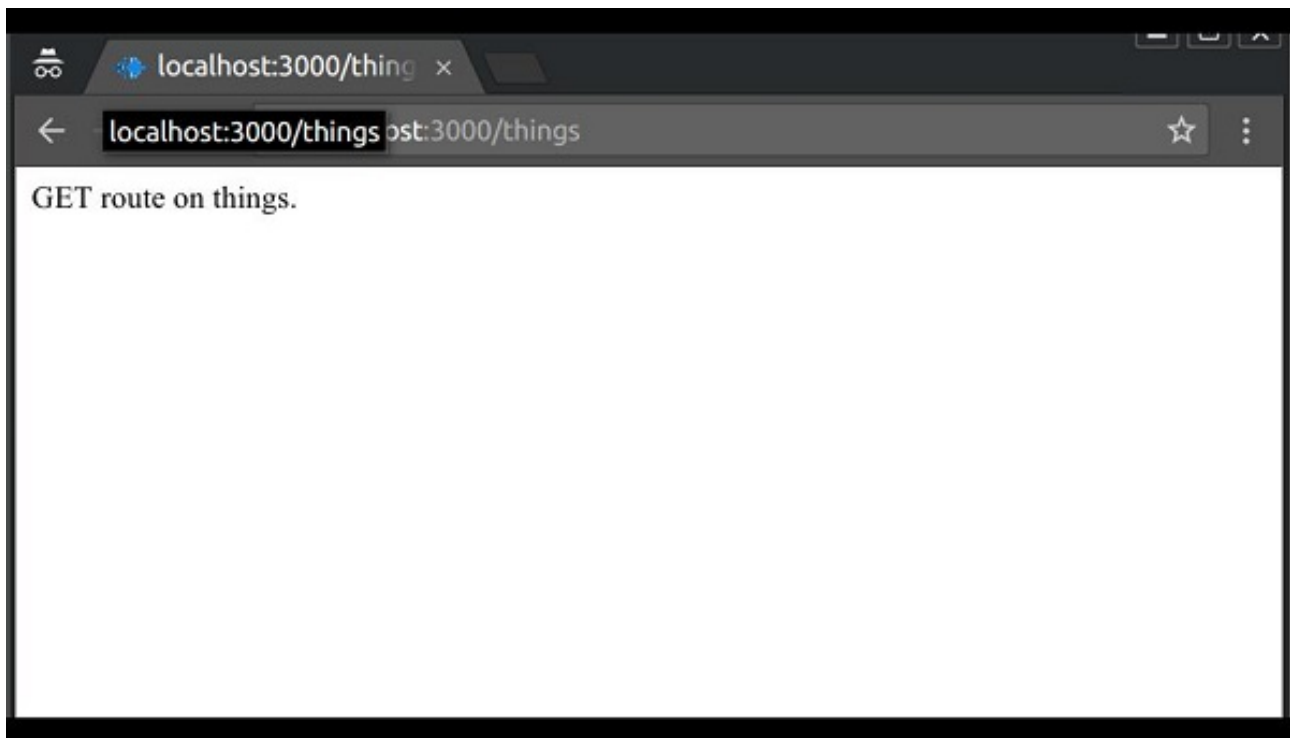
```
var express = require('Express');
var app = express();

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

```
//both index.js and things.js should be in same
directory
app.use('/things', things);
```

```
app.listen(3000);
```

The **app.use** function call on route **'/things'** 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 actually a subroute of **'/things'**. Visit localhost:3000/things/ and you will see the following output.



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 –

S.No.	Method & Description
1	GET The GET method requests a representation of the specified resource. Requests using GET should only retrieve data and should have no other effect.
2	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.
3	PUT The PUT method requests that the server accept the data enclosed in the request as a modification to existing object identified by the URI. If it does not exist then the PUT method should create one.

4	DELETE The DELETE method requests that the server delete the specified resource.
---	--

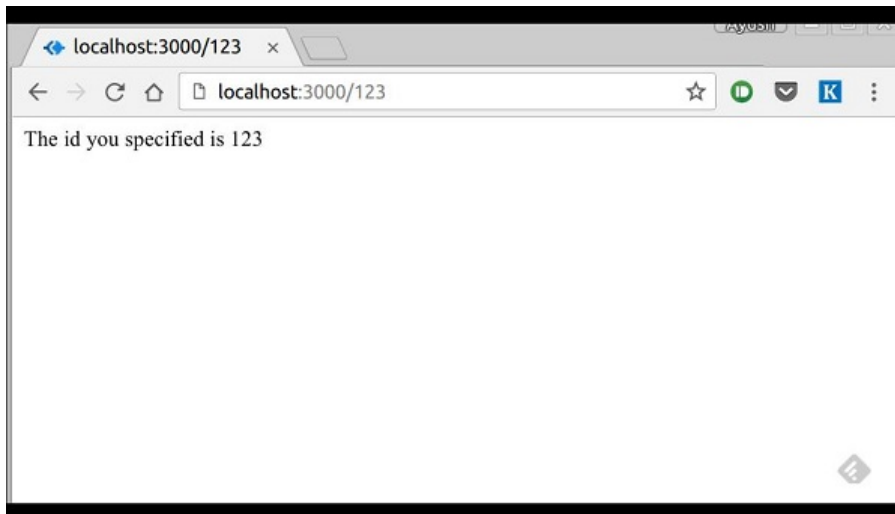
ExpressJS - URL Building

Here is an example of a dynamic route –

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

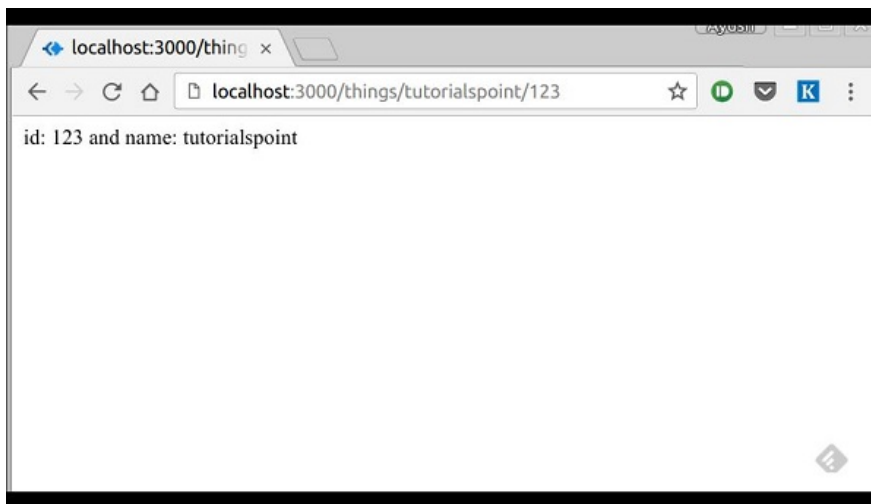
```
var express = require('express');  
var app = express();
```

```
app.get('/things/:name/:id', function(req, res) {
```



```
res.send('id: ' + req.params.id + ' and name: ' + req.params.name);
});
app.listen(3000);
```

To test the above code, go to **http://localhost:3000/things/Bhuvana/12345**.



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 –

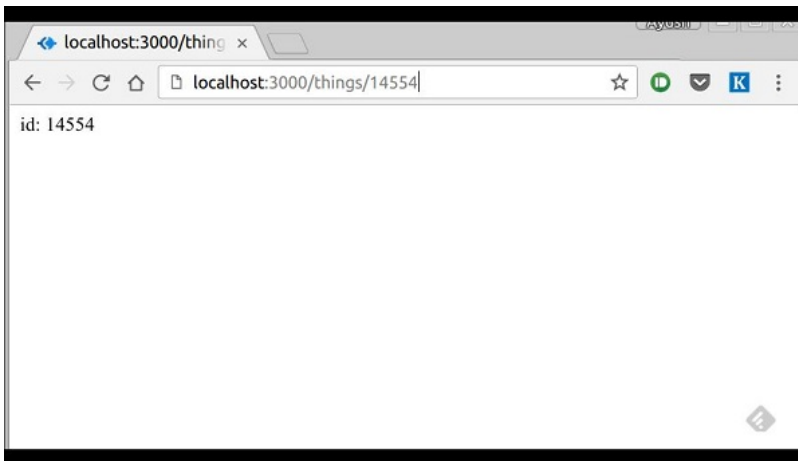
```
var express = require('express');
var app = express();

app.get('/things/:id([0-9]{5})', function(req, res){
  res.send('id: ' + req.params.id);
});

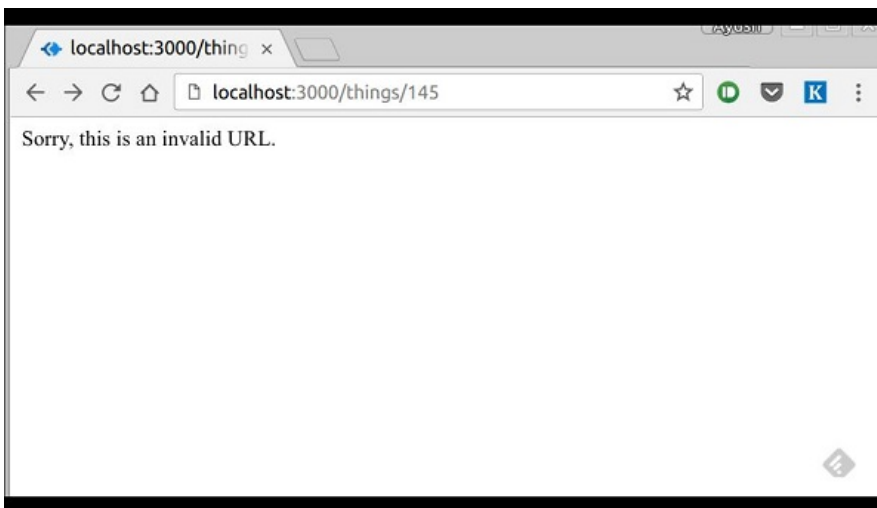
app.get('*', function(req, res){
  res.send('Sorry, this is an invalid URL.');
```

```
});
app.listen(3000);
```

For example, if we define the same routes as above, on requesting with a valid URL, the following output is displayed. –



While for an incorrect URL request, the following output is displayed.



ExpressJS - 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.

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

```
//Simple request time logger  
app.use(function(req, res, next){
```

```
console.log("A new request received at " +  
Date.now());
```

```
//This function call is very important. It  
tells that more processing is  
//required for the current request and is in  
the next middleware function route handler.  
next();  
});
```

```
app.listen(3000);
```

The above middleware is called for every request on the server. So after every request, we will get the following message in the console –

```
A new request received at 1467267512545
```

To restrict it to a specific route (and all its subroutes), provide that route as the first argument of **app.use()**. For Example,

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

```
//Middleware function to log request protocol  
app.use('/things', function(req, res, next){  
  console.log("A request for things received at "  
+ Date.now());  
  next();  
});
```

```
// Route handler that sends the response  
app.get('/things', function(req, res){  
  res.send('Things');  
});
```

```
app.listen(3000);
```

How to use pug with Express.js

Pug is an easy and simple template engine that could use with both NodeJs and web browsers. It allows us to write **HTML** using a simplified syntax that can improve code readability and boost productivity. With Pug, it's easy to create reusable HTML templates and to render data retrieved from databases or APIs dynamically.

Implementing Pug with Express.js

First, we create a file `app.js` and folder `views`, which contain the initially empty template file `index.pug`. We also have to make a `package.json` file for our app using the following command:

```
npm init -y
```

After creating files and running a command, our project structure is as follows:

```
app.js
package.json
>views
  index.pug
```

Then, we install Express and Pug with the following command:

```
npm install express pug
```

```
npm install pug
```

Code explanation

As previously mentioned, we have created an empty pug file, and we'll currently insert the following code into the `index.pug` file:

```
html
  head
    title = title
  body
```

h2 = message

- **Line 3:** We set the title of the HTML document to the value of the title variable, which is passed in from the Node.js/Express application. The `=` sign is used to indicate that the content of the title tag should be dynamically generated by the server-side code.
- **Line 5:** We create an `h2` heading element with the content of the `message` variable, which is also passed in from the Node.js/Express application.

Furthermore, we are now inserting the following code into the `app.js` file, which was previously empty.

```
const express = require("express");
const app = express();
```

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

```
app.set('view engine', 'pug');
```

```
app.get('/', (req, res)=>{
  res.render(
    'index',
    { title: 'PUG with Express', message: 'Hello from Imarticus!' }
  )
});
```

```
app.listen(3000);
```

- **Line 4:** We import the `pug` package, which is a templating engine for Node.js, and assigns it to a constant variable called `pug`.
- **Line 6:** We set the default `view engine` for the app to `pug` so that when the app renders views, it will use the `pug` templating engine.
- **Lines 9–11:** We render the `index` template using `pug` and passes in an object with two properties: `title` and `message`. These properties will be available to the `index` template and can be accessed using `pug` syntax.

Run:

```
Bhuvaneswaris-MacBook-Pro:express bhuvaneswarigokul$  
nodemon app.js
```

Output:

Hello from Imarticus!

Third Party Middleware

body-parser

The body-parser middleware converts text send through an HTTP request to a target format. 4 different parsers: text, JSON, URL encoded and raw. body-parser fails to parse if the content type of the request does not match that defined on the route.

This is used to parse the body of requests which have payloads attached to them. To mount body parser, we need to install it using **npm install --save body-parser** and to mount it, include the following lines in your index.js –

```
var bodyParser = require('body-parser');  
  
//To parse URL encoded data  
app.use(bodyParser.urlencoded({ extended: false })))  
  
//To parse json data  
app.use(bodyParser.json())
```

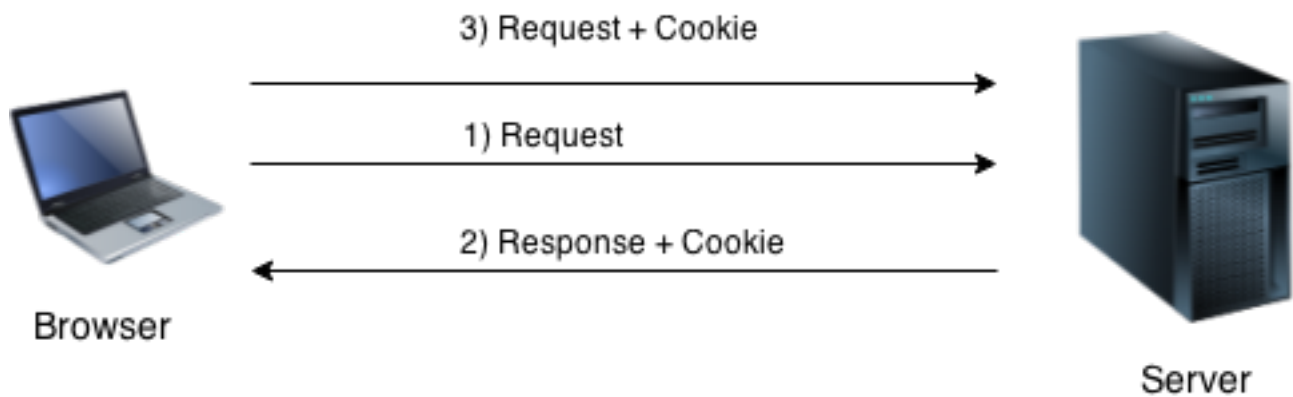
cookie-parser

It parses Cookie header and populate req.cookies with an object keyed by cookie names. To mount cookie parser, we need to install it using npm install --save cookie-parser and to mount it, include the following lines in your index.js –

```
var cookieParser = require('cookie-parser');  
app.use(cookieParser())
```

What are cookies

Cookies are small piece of information i.e. sent from a website and stored in user's web browser when user browses that website. Every time the user loads that website back, the browser sends that stored data back to website or server, to recognize user.



Install cookie

You have to acquire cookie abilities in Express.js. So, install cookie-parser middleware through npm by using the following command:

```
npm install --save cookie-parser
```

Import cookie-parser into your app.

```
var express = require('express');
var cookieParser = require('cookie-parser');
var app = express();
app.use(cookieParser());
```

File: cookies_example.js

```
var express = require('express');
var cookieParser = require('cookie-parser');
var app = express();
app.use(cookieParser());
app.get('/cookieset', function(req, res){
  res.cookie('cookie_name', 'cookie_value');
  res.cookie('company', 'Imarticus');
```

```

res.cookie('name', 'Bhuvana');

res.status(200).send('Cookie is set');
});
app.get('/cookieget', function(req, res) {
  res.status(200).send(req.cookies);
});
app.get('/', function (req, res) {
  res.status(200).send('Welcome to Imarticus Learning!');
});
var server = app.listen(8000, function () {
  var host = server.address().address;
  var port = server.address().port;
  console.log('Example app listening at http://%s:%s', host,
port);
});

```

Output:

```

Bhuvaneswaris-MacBook-Pro:express-database bhuvaneswarigokul$ node
cookie.js

```

http://localhost:8000/

Welcome to Imarticus Learning

http://localhost:8000/cookieset
 Cookie is set

http://localhost:8000/cookieget
 {"cookie_name":"cookie_value","company":"Imarticus","name":"Bhuvana"}

ExpressJS - Sessions

HTTP is **stateless**; in order to associate a request to any other request, you need a way to store user data between HTTP requests. Cookies and URL parameters are both suitable ways to transport data between the client and the server. But they are both readable and on the client side. Sessions solve exactly this problem. You assign the client an ID and it makes all further requests using that ID. Information associated with the client is stored on the server linked to this ID.

stateless - This protocol does not require the server to retain the server information or session details

stateful - This protocol require the server to save the status and session information

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

session.js

```
var express = require('express');
var cookieParser = require('cookie-parser');
var session = require('express-session');

var app = express();

app.use(cookieParser());
app.use(session({secret: "Shh, its a secret!"}));

app.get('/', function(req, res){
  if(req.session.page_views){
    req.session.page_views++;
    res.send("You visited this page " + req.session.page_views +
" times");
  } else {
    req.session.page_views = 1;
    res.send("Welcome to this page for the first time!");
  }
});
app.listen(3000);
```

Output

```
Bhuaneswaris-MacBook-Pro:express-database bhuaneswarigokul$ node
session.js
```

<http://localhost:3000/>

Welcome to this page for the first time!

If you refresh the page
You visited this page 2 time

If you refresh again
You visited this page 3 times

MIDDLEWARE IN EXPRESS.JS

In formal terms, a middleware function has access to the request object, response object, and access to the next function. This next

function is called when we want the next middleware or function in the sequence to be called.

We can use middleware to do the following tasks:

1. Execute code on our server
2. Modify request/response objects
3. End the current request/response cycle
4. Call next middleware

Throughout this article, we will learn how to do these tasks.

Let's start with some hands-on work. Create an empty folder and initialize a node project by running `npm init` and then install express by running `npm i express`.

Set up an express server by creating a file called `app.js` which has

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

```
app.listen(3000, () => {  
  console.log('Server started');  
});
```

On running the command `nodemon app.js`, the following should appear in your terminal.

```
Hp@srajan MINGW64 /g/Code Work/thrive/middlewares (master)  
$ nodemon app.js  
[nodemon] 2.0.6  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching path(s): *.*  
[nodemon] watching extensions: js,mjs,json  
[nodemon] starting `node app.js`  
Server started
```

Let's start by creating a middleware that gets executed for every request. Add the following code right after you initialize the `app` variable.

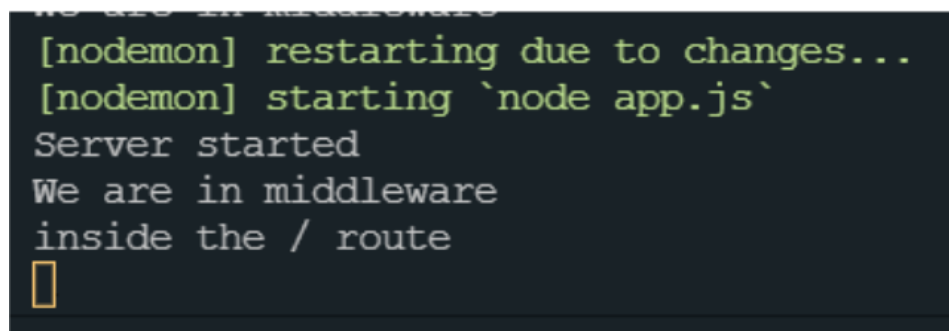
```
const middlewareFunction = function (req, res, next) {
```

```
console.log('We are in middleware');  
next();  
};
```

```
app.use(middlewareFunction);
```

```
app.get('/', (req, res) => {  
  res.send('inside the / route');  
});
```

First, we have created a function called `middlewareFunction` which takes the `req`, `res`, `next` as arguments. Whenever this middleware will get called it will print `We are in middleware` to the console and call the next function. Since we are using it here for each route with the help of this `app.use(middlewareFunction)`; code snippet, it will be called for each request. Let's now send a request to the <http://localhost:3000> using postman and see the console. Here, we are executing code on our server which is one of the tasks of middleware mentioned above.



```
[nodemon] restarting due to changes...  
[nodemon] starting `node app.js`  
Server started  
We are in middleware  
inside the / route  
█
```

As you can see, our middleware was executed before our request. There are several uses for this middleware. For example, you can store request logs in your database.

Now, we will make our middleware work for only specific requests. Remove the `app.use()` line and modify the API call like this:

```
app.get('/', middlewareFunction, (req, res) => {  
  console.log('inside the / route');  
  res.send('Route hit');  
});
```

Let's create another API without middleware.

```
app.get('/no-middleware', (req, res) => {  
  console.log('inside the /no-middleware route');  
  res.send('Route hit');  
});
```

Now let's make a request to both the routes.

1. Request to / route (middleware gets called)

```
[nodemon] 2.0.6  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching path(s): *.*  
[nodemon] watching extensions: js,mjs,json  
[nodemon] starting `node app.js`  
Server started  
We are in middleware  
inside the / route
```

2. Request to /no-middleware (middleware does not get called)

```
[nodemon] restarting due to changes...  
[nodemon] starting `node app.js`  
Server started  
inside the /no-middleware route
```

Let's try and create a protected route to filter out unauthenticated users using middleware.

Note: In the following example we will randomly select a value and according to that value we will judge whether the user is authorized or not. This is not how it is done in real applications, it is just to show you how middleware is used in the login process.

Modify your middleware function like this.

```
const middlewareFunction = function (req, res, next) {  
  const random = (min, max) => Math.floor(Math.random() * (max -  
min)) + min;  
  const isAuthenticated = random(0, 2);
```

```
    if (isAuthenticated)
      next();
    else res.send('User is not authorized');
  };
```

```
Math.floor(Math.random() * (max - min)) + min;
1 * (2-0) + 0 = 0.999999999998 * 2 = 1.99999998
```

Here we are assigning a random value to the isAuthenticated variable. If it is one then we are moving on with our process. Otherwise, we terminate this cycle by sending the message to the user.

In the end our app.js file looks like this.

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

const middlewareFunction = function (req, res, next) {
  const random = (min, max) => Math.floor(Math.random() * (max - min)) + min;
  const isAuthenticated = random(0, 2);
  if (isAuthenticated)
    next();
  else res.send('User is not authorized')
};

app.get('/', middlewareFunction, (req, res) => {
  console.log('inside the / route');
  res.send('Route hit');
});

app.get('/no-middleware', (req, res) => {
  console.log('inside the /no-middleware route');
  res.send('Route hit');
});

app.listen(3000, () => {
  console.log('Server started');
});
```

In the above piece of code, we are ending the request/response cycle for unauthenticated users and also modifying the response object by sending a message of the user being unauthenticated. We are calling the `next()` middleware for the authenticated users.

WORKING WITH ASYNCHRONOUS MIDDLEWARES:

Working with asynchronous middlewares is similar to working with non-asynchronous middlewares. All we have to do is add the `async-await` keywords in front of our middleware functions. Modify the middleware like this.

```
const middlewareFunction = async function (req, res, next) {  
  const random = (min, max) => Math.floor(Math.random() * (max -  
min)) + min;  
  let isAuthenticated;  
  await setTimeout(() => {  
    isAuthenticated = random(0, 2);  
    if (isAuthenticated) next();  
    else res.send("User is not authorized");  
  }, 3000);  
};
```

Here I have added an international delay of three seconds, which makes the function asynchronous.

We will still add it in our route the same way.

```
app.get('/', middlewareFunction, (req, res) => {  
  console.log('inside the / route');  
  res.send('Route hit');  
});
```

Now let's make a request.

You will see that it waits for some time before giving back the output, which means the code is working fine.

ADDING THIRD PARTY MIDDLEWARES:

If you want to add any third-party middleware to your code then you will have to work according to the documentation provided on the middleware's website.

For example:

1. Open localhost:3000 on browser
2. Open network tab in console.
3. If it's empty, then reload with the network tab open, you should see something like this.

Name	Status	Type	Initiator	Size	Time	Waterfall
localhost	200	docu...	Other	155 B	3.01 s	

4. Click on this request. You will see this.

▼ Response Headers	View source
Connection: keep-alive	
Date: Sun, 14 Nov 2021 09:33:13 GMT	
ETag: W/"16-6N18FL2Gk/Qvh/MV7DonruNF1jg"	
X-Powered-By: Express	

5. It is announcing to the world that you are using an express app at the backend, because of the X-Powered-By: Express object. People with malicious intent can do attacks that are known to work specifically against Express-based apps.

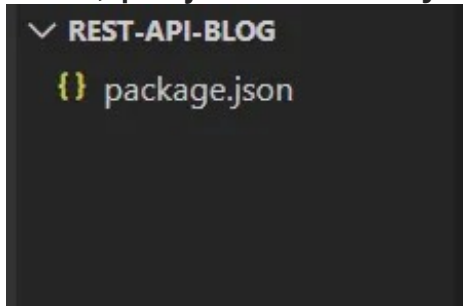


We will be making a REST API a Book store.

So, let's start with creating an empty project and running the command

```
npm init
```

Your, project directory will look like this now



Now, let's install our necessary dependencies:-

1. express `npm install express`
 2. body-parser `npm install body-parser`
- and create our server file `server.js`

Now, let's first create a file `data.js` where we will store our variable `data` which will act as a database for our tutorial.

In the file `data.js` add the code given below

```
const data = {}
```

```
module.exports = data
```

That's all we need in our file, now let's move to `server.js` and start loading our dependencies and our variable `data` from `data.js`.

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

```
const bodyParser = require('body-parser')
```

```
const data = require('./data')
```

Lets now create a `express` server by creating our `app` variable and using our `body-parser` middleware which will allow us to get our data under `req.body` variable and add the following code in `server.js`


```

const express = require('express')
const bodyParser = require('body-parser')

const data = require('./data')

const app = express()

app.use(bodyParser.json())

app.use(bodyParser.urlencoded({extended: false}))

app.listen(3000, () =>{
  console.log("App Started at Port 3000")
})

```

Let's decide our routes for CRUD operations, for all CRUD operations we will have one route each so our routes will be:-

`/api/add` (CREATE ROUTE)

This route will add a book to our database (data variable).

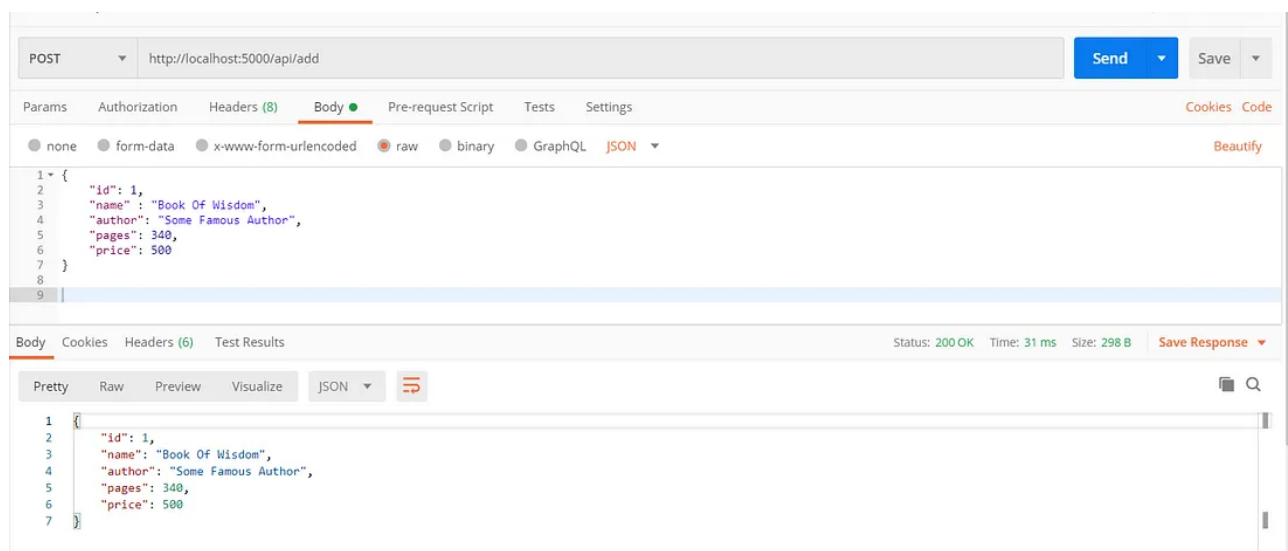
```

app.post('/api/add', (req, res) => {
  const {id, name, author, pages, prices} = req.body

  data[id] = {
    id, name, author, pages, prices
  }

  res.json(data[id])
})

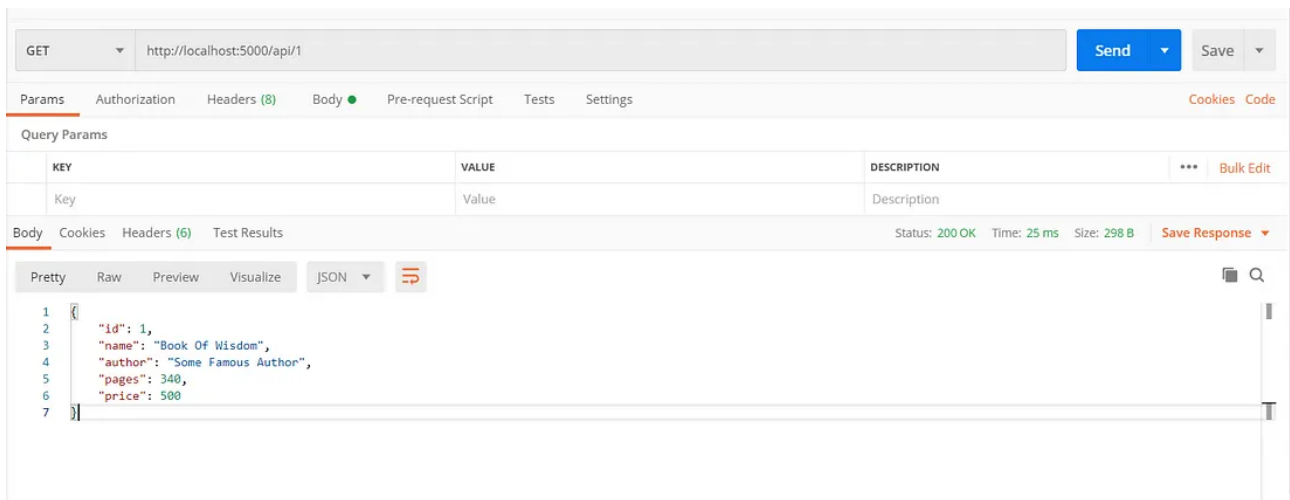
```



2. /api/:bookid (READ ROUTE)

This route will allow us to fetch the book record from the database using the book id.

```
app.get('/api/:bookid',(req,res) => {  
  const id = req.params.bookid  
  
  res.json(data[id])  
})
```



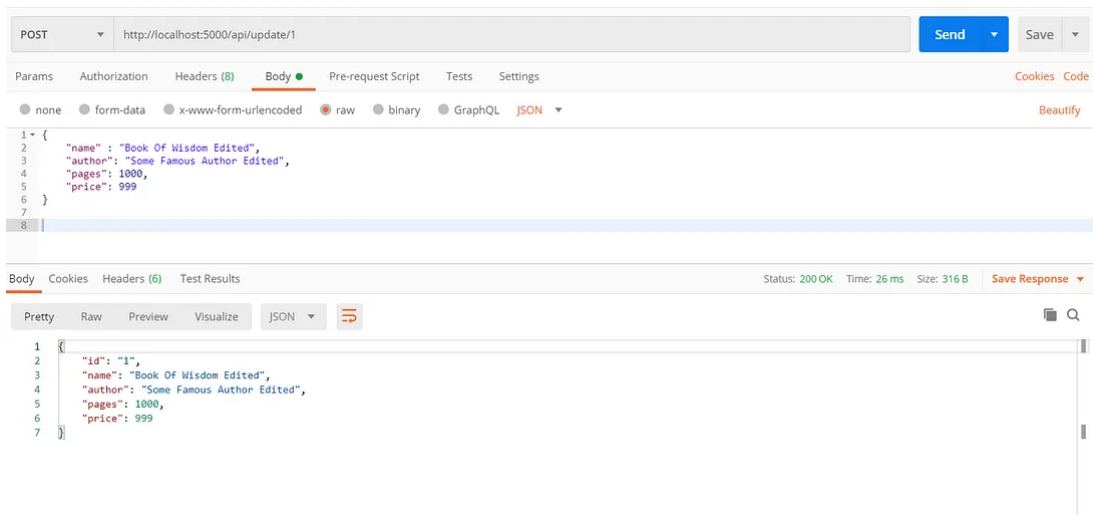
/api/update/:bookid (UPDATE ROUTE)

This route will allow us to update an existing record available at the database.

```
app.put('/api/update/:bookid',(req,res) =>{  
  const id = req.params.bookid  
  
  const {name,author,pages,prices} = req.body  
  data[id] = {id,name,author,pages,prices}  
  
  res.json(data[id])  
})
```

/api/delete/:bookname (DELETE ROUTE)

This route will allow us to delete a record from the database.



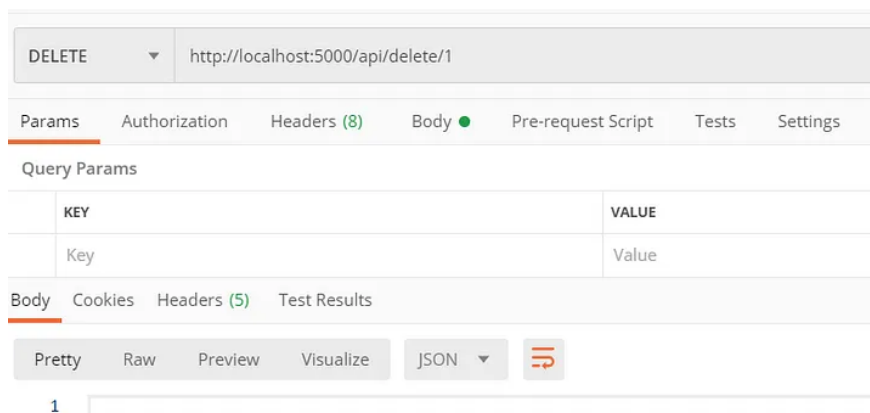
```

app.delete('/api/delete/:bookid', (req, res) => {
  const id = req.params.bookid

  delete data[id]

  res.json(data[id])
})

```



So, our record was deleted successfully, but let's verify from our READ ROUTE.

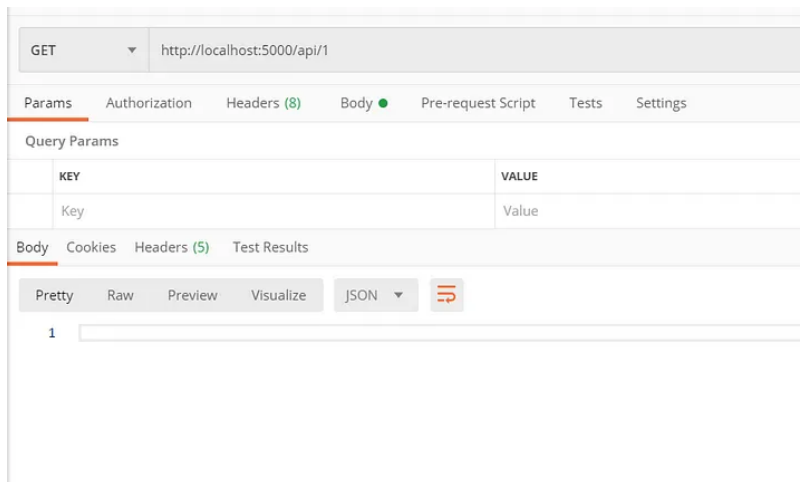
server.js

```

const express = require('express')
const bodyParser = require('body-parser')

const data = require('./data')

```



```
const app = express()

app.use(bodyParser.json())

app.use(bodyParser.urlencoded({extended: false}))

app.post('/api/add', (req, res) => {
  const {id, name, author, pages, prices} = req.body

  data[id] = {
    id, name, author, pages, prices
  }

  res.json(data[id])
})

app.get('/api/:bookid', (req, res) => {
  const id = req.params.bookid

  res.json(data[id])
})

app.put('/api/update/:bookid', (req, res) => {
  const id = req.params.bookid

  const {name, author, pages, prices} = req.body
  data[id] = {id, name, author, pages, prices}

  res.json(data[id])
})

app.delete('/api/delete/:bookid', (req, res) => {
  const {id} = req.params.bookid

  delete data[id]
})
```

```
    res.json(data[id])
  })

app.listen(3000, () =>{
  console.log("App Started at Port 3000")
})
```

Database integration in Express.js

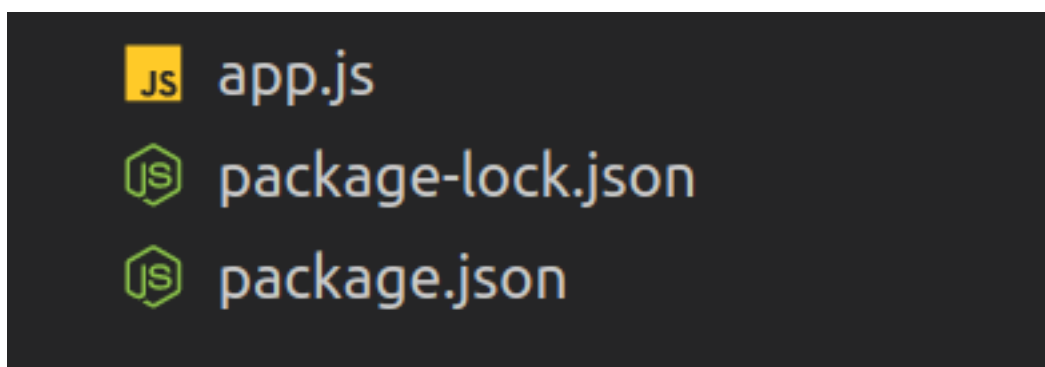
Express is a minimalistic framework that works along with Node.js and provides salient features to run a backend server and much more. As you know the database plays a vital role in a fully working application to persist data

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. You can skip this step if you are comfortable with this stuff.

Write these commands in your terminal to start a node app and then install express. Make sure that you have successfully installed npm. The **npm init** will ask you for some configuration about your project and that is super easy to provide.

```
npm init -y
npm install express
```



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

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

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

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

Run:

```
node app.js
```

```
Bhuvanewaris-MacBook-Pro:express-database
bhuvanewarigokul$ node app.js
Server is Successfully Running, and App is
listening on port 3000
```

Step 3: Integrate Database

Here comes the most interesting part which you are searching for, 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, so we are going to discuss only these two.

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
```

```
//Importing modules
```

```

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

const app = express();
const PORT = 3000;
//Connection to the mongodb database
mongoose.connect('mongodb://localhost:27017/First')
.then(()=>{
  app.listen(PORT, ()=>{
    console.log("Database connection is Ready "
      + "and Server is Listening on Port ", PORT);
  })
})
.catch((err)=>{
  console.log("A error has been occurred while"
    + " connecting to database.");
})
app.get('/', function(req, res){
  res.send("Hello world from database!");
});

```

Run:

```

Bhuvaneswaris-MacBook-Pro:express-database bhuvaneswarigokul$ node
db.js

```

```

Database connection is Ready and Server is Listening on Port 3000

```

React Redux-thunk-middleware1.

Introduction to Redux-Thunk

Redux-Thunk is a middleware for Redux that allows you to write action creators that return a function instead of an action object. This function receives the store's `dispatch` method and `getState` function as arguments, allowing it to dispatch multiple actions, perform asynchronous operations, and access the current state if needed before dispatching an action.

1.1. Understanding Middleware in Redux

Before diving into Redux-Thunk, let's briefly discuss what middleware is in the context of Redux.

Middleware provides a way to interact with actions dispatched to the Redux store before they reach the reducer. It sits between the action

dispatch and the reducer, allowing you to intercept, modify, or delay actions as needed.

It provides a way to extend Redux's functionality by intercepting and potentially modifying actions before they reach the reducers.

1.2. The Role of Redux-Thunk

The primary purpose of Redux-Thunk is to handle asynchronous actions in Redux. Asynchronous actions, such as fetching data from an API or performing asynchronous computations, are common in web applications.

Asynchronous actions: Redux Thunk dispatches things with Asynchronous actions, which is crucial when you fetch data from an API, perform network requests, or handle other asynchronous tasks while updating the state.

First, make sure you have Redux and Redux Thunk installed in your project:

```
npm install redux react-redux redux-thunk
```

now let's move it first and create a file "store.js",

```
// store.js
```

```
import { createStore, applyMiddleware } from 'redux';
```

```
import thunk from 'redux-thunk';
```

```
import rootReducer from './reducers'; // Assuming you have your rootReducer
```

```
const store = createStore(rootReducer, applyMiddleware(thunk));
```

```
export default store;
```

- **Integrate Redux Store with React:**

In your main application file (often `index.js` or `App.js`), wrap your application with the `Provider` component from `react-redux` and provide your Redux store with it.


```
// index.js
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store from './store'; // Import your Redux store
import App from './App'; // Your root application component
```

```
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

2: let's create a new file in the src folder named "actions.js"

```
// actions.js
import axios from 'axios';
```

// Action Types

```
export const FETCH_DATA_REQUEST = 'FETCH_DATA_REQUEST';
export const FETCH_DATA_SUCCESS = 'FETCH_DATA_SUCCESS';
export const FETCH_DATA_FAILURE = 'FETCH_DATA_FAILURE';
```

// Action Creators

```
const fetchDataRequest = () => {
  return {
    type: FETCH_DATA_REQUEST,
  };
};
```

```
const fetchDataSuccess = (data) => {
  return {
    type: FETCH_DATA_SUCCESS,
    payload: data,
  };
};
```

```
const fetchDataFailure = (error) => {
  return {
    type: FETCH_DATA_FAILURE,
    payload: error,
  };
};
```

```
};
```

```
// Thunk Action Creator
```

```
export const fetchData = () => {  
  return (dispatch) => {  
    dispatch(fetchDataRequest());  
    axios.get('https://api.example.com/data')  
      .then(response => {  
        const data = response.data;  
        dispatch(fetchDataSuccess(data));  
      })  
      .catch(error => {  
        dispatch(fetchDataFailure(error.message));  
      });  
  };  
};
```

3: So in the above code, we see three actions types (FETCH_DATA_REQUEST, FETCH_DATA_SUCCESS, and FETCH_DATA_FAILURE)

this is used to get data from APIs and dispatch this, so let's create a reducer file named “reducer.js”

```
// reducer.js
```

```
import {  
  FETCH_DATA_REQUEST,  
  FETCH_DATA_SUCCESS,  
  FETCH_DATA_FAILURE,  
} from './actions';
```

```
const initialState = {  
  data: [],  
  isLoading: false,  
  error: null,  
};
```

```
const reducer = (state = initialState, action) => {  
  switch (action.type) {  
    case FETCH_DATA_REQUEST:  
      return {  
        ...state,  
        isLoading: true,  
      };  
  }  
};
```

```

    error: null,
  };
  case FETCH_DATA_SUCCESS:
    return {
      ...state,
      isLoading: false,
      data: action.payload,
    };
  case FETCH_DATA_FAILURE:
    return {
      ...state,
      isLoading: false,
      error: action.payload,
    };
  default:
    return state;
}
};

```

```
export default reducer;
```

4: To use this in a React component, you would typically connect the component to Redux and dispatch the `fetchData` action when needed:

// YourComponent.js

```

import React, { useEffect } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { fetchData } from './actions';

const YourComponent = () => {
  const dispatch = useDispatch();
  const data = useSelector(state => state.data);
  const isLoading = useSelector(state => state.isLoading);
  const error = useSelector(state => state.error);

  useEffect(() => {
    dispatch(fetchData());
  }, [dispatch]);

  if (isLoading) {
    return <p>Loading...</p>;
  }
}

```

```
if (error) {  
  return <p>Error: {error}</p>;  
}
```

```
return (  
  <div>  
    { /* Render your data here */ }  
  </div>  
);  
};
```

```
export default YourComponent;
```