Express.js

Express.js is a web framework for Node.js. It is a fast, robust and asynchronous in nature.

What is Express.js

Express is a fast, assertive, essential and moderate web framework of Node.js. You can assume express as a layer built on the top of the Node.js that helps manage a server and routes. It provides a robust set of features to develop web and mobile applications.

Core Features of Express framework:

- It can be used to design single-page, multi-page and hybrid web applications.
- It allows to setup middlewares to respond to HTTP Requests.
- It defines a routing table which is used to perform different actions based on HTTP method and URL.
- It allows to dynamically render HTML Pages based on passing arguments to templates.

Why use Express

- Ultra fast I/O
- Asynchronous and single threaded
- MVC like structure
- Robust API makes routing easy

Commands

- npm init
- npm install express --save **//install express**
- node app.js **// to run**
- http://localhost:5500/ // **– to run & view the page in browser where 5500 is the port number in which the server runs**

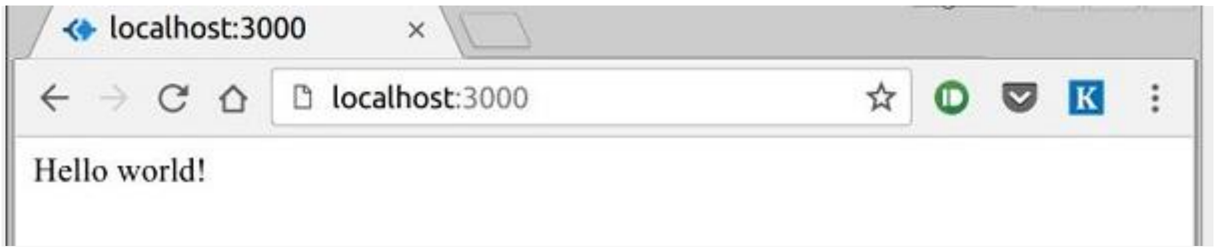
ExpressJS - Hello World

index.js

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

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

app.listen(3000);
node index.js // to run
```

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.



Hello world!

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.

Some key differences between Express.js and Node.js:

Feature	Express.js	Node.js
Definition	Express.js is a lightweight and fast backend web application framework for Node.js.	Node.js is an open-source and cross-platform that is used to execute JavaScript code outside of a browser.
Usage	Express.js is used to develop complete web applications such as single-page, multi-page, and hybrid web applications and APIs. It uses approaches and principles of Node.js.	Node.js is used to build server-side, input-output, event-driven apps.
Features	Express has more features than Node.js.	Node.js has fewer features as compared to Express.js.
Building Block	Express.js is built on Node.js.	Node.js is built on Google's V8 engine.
Written in	Express.js is written in JavaScript only.	Node.js is written in C, C++, and JavaScript language.
Framework/Platform	Express.js is a framework of Node.js based on its functionalities.	Node.js is a run-time platform or environment designed for server-side execution of JavaScript.
Controllers	Express.js is assigned with controllers.	Node.js is assigned with controllers.

Routing	Routing is provided in Express.js.	Routing is not provided in Node.js.
Middleware	Express.js uses middleware to arrange the functions systematically on the server-side.	Node.js doesn't use any such provision of middleware.
Coding	Express is easy to code and requires less coding time.	Node.js requires more coding time as compare to Express.js.

Express.js Request Object

Express.js Request and Response objects are the parameters of the callback function which is used in Express applications.

The express.js request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.

Syntax:

```
app.get('/', function (req, res) {
   // --
})
```

Express.js Request Object Properties

The following table specifies some of the properties associated with request object.

Index	Properties	Description
1.	req.app	This is used to hold a reference to the instance of the express application that is using the middleware.
2.	req.baseurl	It specifies the URL path on which a router instance was mounted.
3.	req.body	It contains key-value pairs of data submitted in the request body. By default, it is undefined, and is populated when you use body-parsing middleware such as body-parser.
4.	req.cookies	When we use cookie-parser middleware, this property is an object that contains cookies sent by the request.

5.	req.fresh	It specifies that the request is "fresh." it is the opposite of req.stale.
6.	req.hostname	It contains the hostname from the "host" http header.
7.	req.ip	It specifies the remote IP address of the request.
8.	req.ips	When the trust proxy setting is true, this property contains an array of IP addresses specified in the ?x-forwarded-for? request header.
9.	req.originalurl	This property is much like req.url; however, it retains the original request URL, allowing you to rewrite req.url freely for internal routing purposes.
10.	req.params	An object containing properties mapped to the named route ?parameters?. For example, if you have the route /user/:name, then the "name" property is available as req.params.name. This object defaults to {}.
11.	req.path	It contains the path part of the request URL.
12.	req.protocol	The request protocol string, "http" or "https" when requested with TLS.
13.	req.query	An object containing a property for each query string parameter in the route.
14.	req.route	The currently-matched route, a string.
15.	req.secure	A Boolean that is true if a TLS connection is established.
16.	req.signedcookies	When using cookie-parser middleware, this property contains signed cookies sent by the request, unsigned and ready for use.
17.	req.stale	It indicates whether the request is "stale," and is the opposite of req.fresh.
18.	req.subdomains	It represents an array of subdomains in the domain name of the request.
19.	req.xhr	A Boolean value that is true if the request's "x-requested-with" header field is "xmlhttprequest", indicating that the request was issued by a client library such as jQuery

Request Object Methods

Following is a list of some generally used request object methods:

req.accepts (types)

This method is used to check whether the specified content types are acceptable, based on the request's Accept HTTP header field.

Examples:

```
req.accepts('html');
//=>?html?
req.accepts('text/html');
// => ?text/html?
```

req.get(field)

This method returns the specified HTTP request header field.

Examples:

```
req.get('Content-Type');
// => "text/plain"
req.get('content-type');
// => "text/plain"
req.get('Something');
// => undefined
```

req.is(type)

This method returns true if the incoming request's "Content-Type" HTTP header field matches the MIME type specified by the type parameter.

Examples:

```
// With Content-Type: text/html; charset=utf-8
req.is('html');
req.is('text/html');
req.is('text/*');
// => true
```

req.param(name [, defaultValue])

This method is used to fetch the value of param name when present.

Examples:

```
// ?name=sasha
req.param('name')
// => "sasha"
// POST name=sasha
req.param('name')
// => "sasha"
// /user/sasha for /user/:name
req.param('name')
// => "sasha"
```

Express.js Response Object

The Response object (res) specifies the HTTP response which is sent by an Express app when it gets an HTTP request.

What it does

- It sends response back to the client browser.
- It facilitates you to put new cookies value and that will write to the client browser (under cross domain rule).
- Once you res.send() or res.redirect() or res.render(), you cannot do it again, otherwise, there will be uncaught error.

Response Object Properties

Index	Properties	Description
1.	res.app	It holds a reference to the instance of the express application that is using the middleware.
2.	res.headersSent	It is a Boolean property that indicates if the app sent HTTP headers for the response.
3.	res.locals	It specifies an object that contains response local variables scoped to the request

Response Object Methods

Refer: https://www.javatpoint.com/expressjs-response

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.

Express.js GET Request

GET and POST both are two common HTTP requests used for building REST API's. GET requests are used to send only limited amount of data because data is sent into header while POST requests are used to send large amount of data because data is sent in the body.

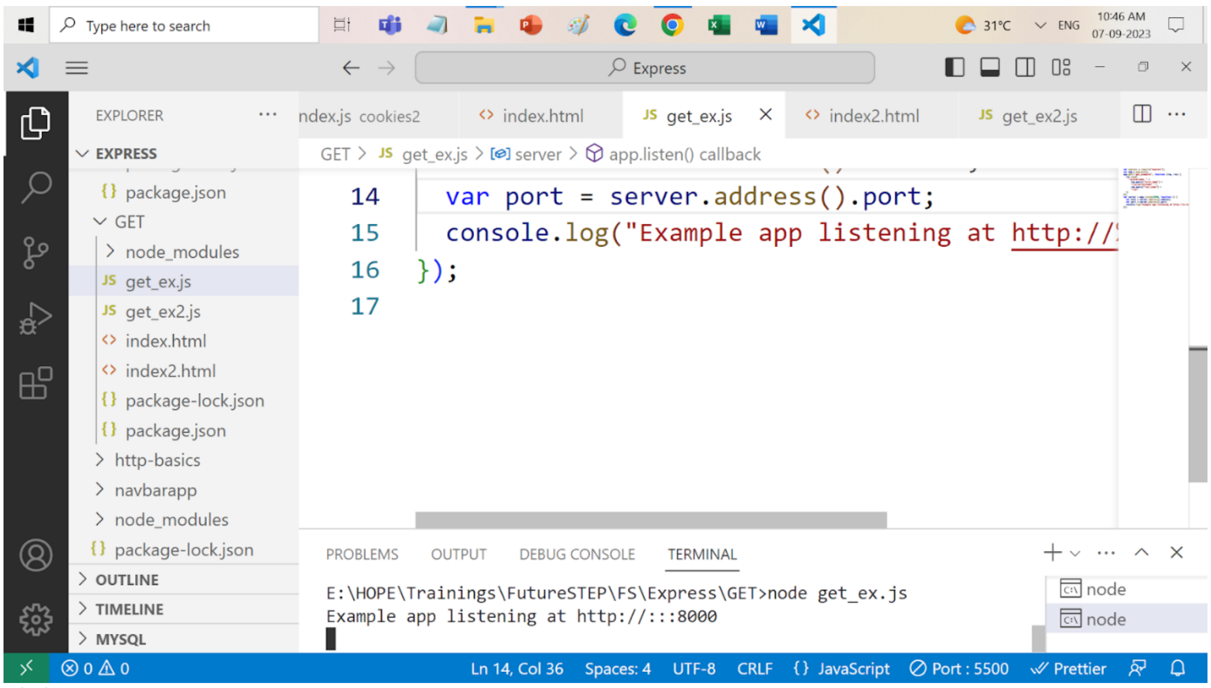
Express.js facilitates you to handle GET and POST requests using the instance of express.

Express.js GET Method Examples

```
mkdir GET
cd GET
npm init
npm install express --save
```

Example 1

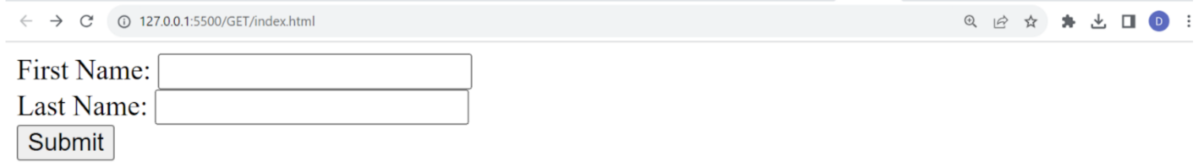
```
index.html
<html>
  <body>
    <form
action="http://127.0.0.1:8000/get_example2"
method="GET">
      First Name: <input type="text"
name="first_name" /> <br />
      Last Name: <input type="text"
name="last_name" /><br />
      <input type="submit" value="Submit" />
    </form>
  </body>
</html>
get_ex.js
var express = require("express");
var app = express();
app.get("/get_example2", function (req, res) {
  res.send(
    "<p>Username: " +
      req.query["first_name"] +
      "</p><p>Lastname: " +
      req.query["last_name"] +
      "</p>"
  );
});
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);
});

//To run
node get_ex.js
//Console.log output:
```



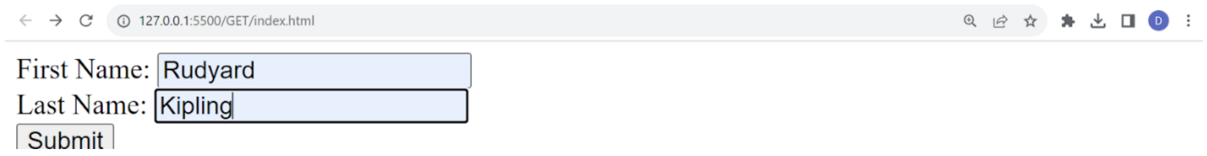
```
    var port = server.address().port;
    console.log("Example app listening at http://
});
```

```
E:\HOPE\Trainings\FutureSTEP\FS\Express\GET>node get_ex.js
Example app listening at http://:::8000
```

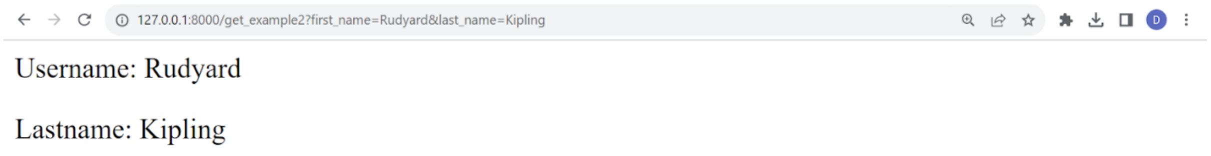
//In browser
http://127.0.0.1:5500/GET/index.html


127.0.0.1:5500/GET/index.html

First Name:
Last Name:
Submit


127.0.0.1:5500/GET/index.html

First Name: Rudyard
Last Name: Kipling
Submit


127.0.0.1:8000/get_example2?first_name=Rudyard&last_name=Kipling

Username: Rudyard

Lastname: Kipling

Query String
http://127.0.0.1:8000/get_example2?first_name=Rudyard&last_name=Kipling

Example 2

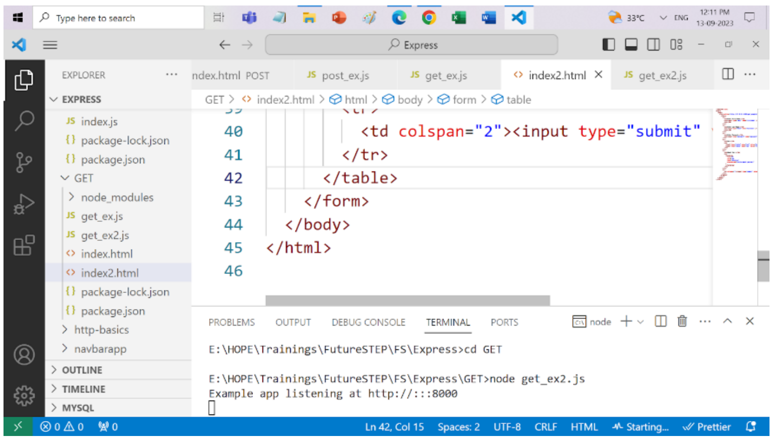
index.html

```
<!DOCTYPE html>
<html>
  <body>
    <form
action="http://127.0.0.1:8000/get_example3">
```

```
      <table>
        <tr>
          <td>Enter First Name:</td>
          <td><input type="text" name="firstname"
/></td>
          <td></td>
        </tr>
        <tr>
          <td>Enter Last Name:</td>
          <td><input type="text" name="lastname"
/></td>
          <td></td>
        </tr>
        <tr>
          <td>Enter Password:</td>
          <td><input type="password"
name="password" /></td>
        </tr>
        <tr>
          <td>Sex:</td>
          <td>
            <input type="radio" name="sex"
value="male" /> Male
            <input type="radio" name="sex"
value="female" />Female
          </td>
        </tr>
        <tr>
          <td>About You :</td>
          <td>
            <textarea
              rows="5"
              cols="40"
              name="aboutyou"
              placeholder="Write about yourself"
            >
            </textarea>
          </td>
        </tr>
        <tr>
```

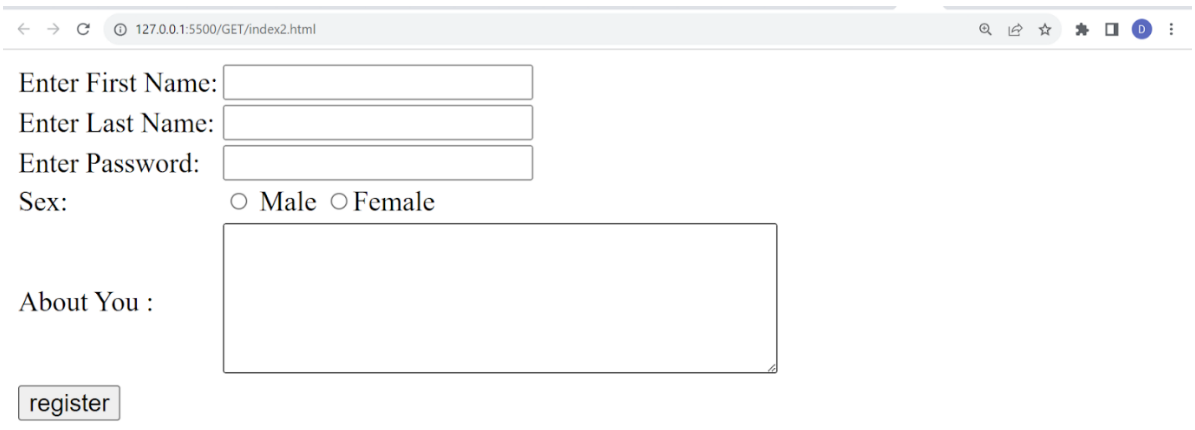
```
            <td colspan="2"><input type="submit"
value="register" /></td>
          </tr>
        </table>
      </form>
    </body>
</html>
```

```
get_ex2.js
var express = require("express");
var app = express();

app.get("/get_example3", function (req, res) {
  res.send(
    "<p>Firstname: " +
      req.query["firstname"] +
      "</p><p>Lastname: " +
      req.query["lastname"] +
      "</p><p>Password: " +
      req.query["password"] +
      "</p><p>AboutYou: " +
      req.query["aboutyou"] +
      "</p>"
  );
});

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);
});
```

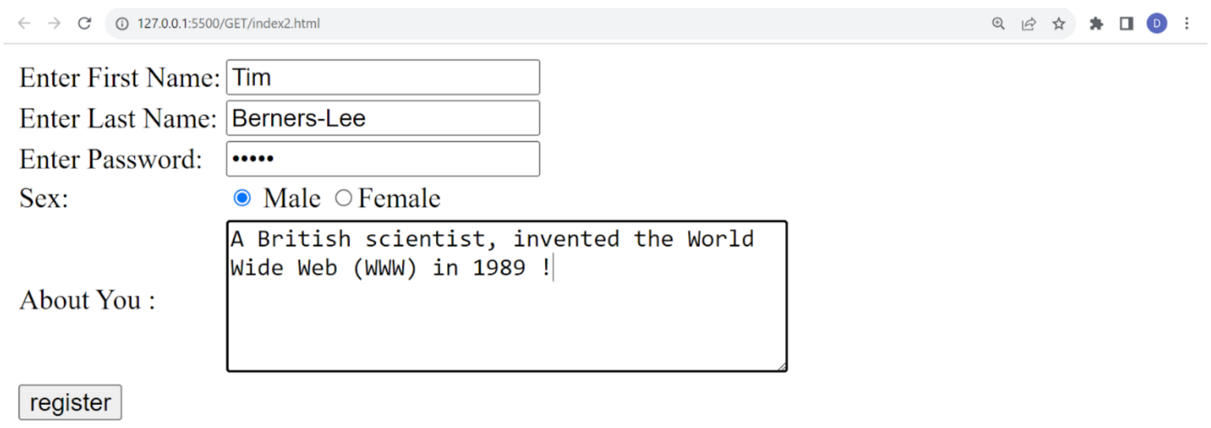


```
          <td colspan="2"><input type="submit"
        </tr>
      </table>
    </form>
  </body>
</html>
```

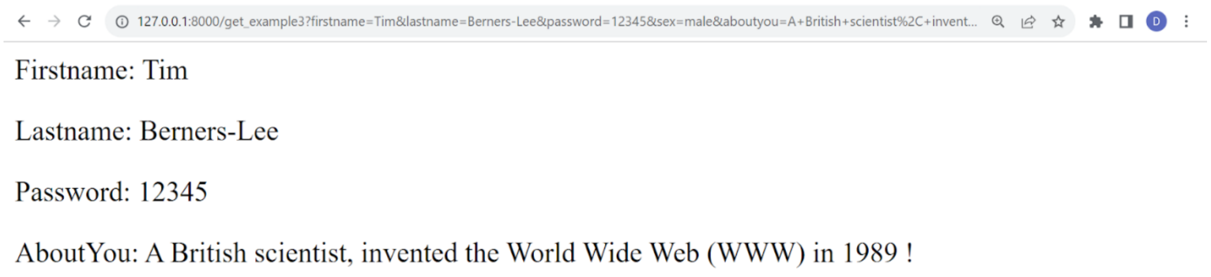
```
E:\HOPE\Trainings\FutureSTEP\FS\Express>cd GET

E:\HOPE\Trainings\FutureSTEP\FS\Express\GET>node get_ex2.js
Example app listening at http://:::8000
```



Enter First Name:	
Enter Last Name:	
Enter Password:	
Sex:	○ Male ○ Female
About You :	



Enter First Name:	Tim
Enter Last Name:	Berners-Lee
Enter Password:	•••••
Sex:	[✓ Male] ○ Female
About You :	A British scientist, invented the World Wide Web (WWW) in 1989 !


127.0.0.1:8000/get_example3?firstname=Tim&lastname=Berners-Lee&password=12345&sex=male&aboutyou=A+British+scientist%2C+invent...

```
Firstname: Tim

Lastname: Berners-Lee

Password: 12345

AboutYou: A British scientist, invented the World Wide Web (WWW) in 1989 !
```

Query String / URL
http://127.0.0.1:8000/get_example3?firstname=Tim&lastname=Berners-Lee&password=12345&sex=male&aboutyou=A+British+scientist%2C+invented+the+World+Wide+Web+%28WWW%29+in+1989+%21+++

Express.js POST Request

GET and POST both are two common HTTP requests used for building REST API's. POST requests are used to send large amount of data.

Express.js facilitates you to handle GET and POST requests using the instance of express.

Express.js POST Method

Post method facilitates you to send large amount of data because data is send in the body. Post method is secure because data is not visible in URL bar but it is not used as popularly as GET method. On the other hand GET method is more efficient and used more than POST.

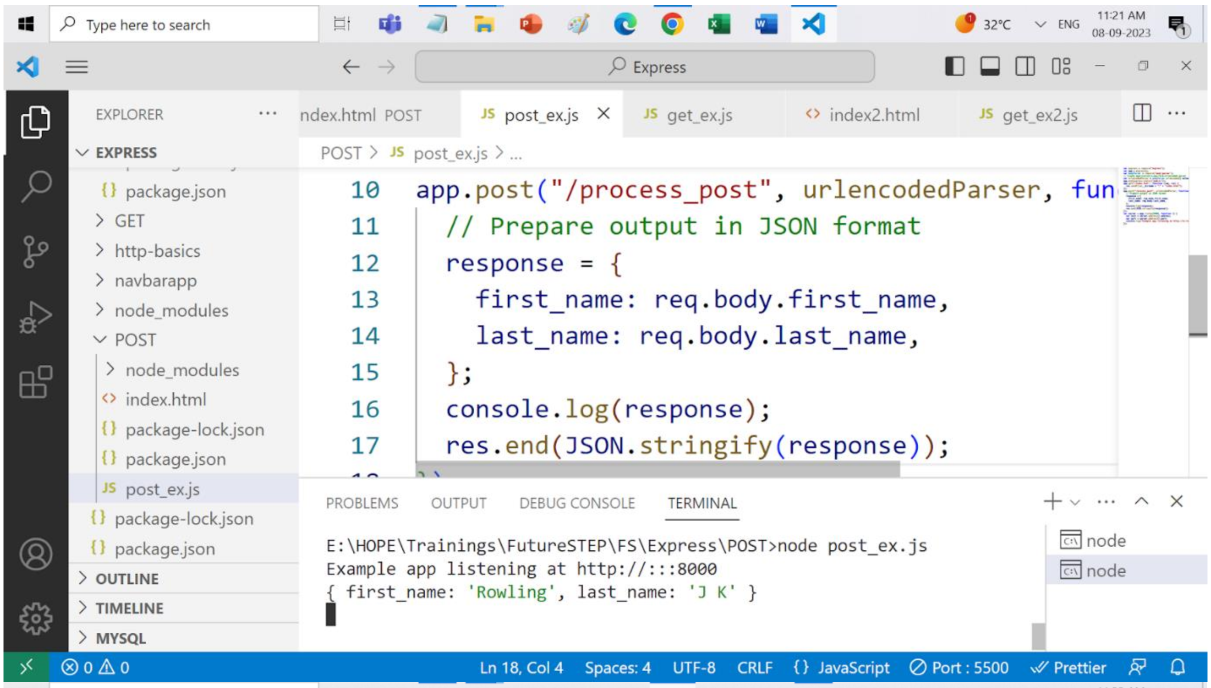
Let's take an example to demonstrate POST method.

index.html

```
<html>
  <body>
    <form
action="http://127.0.0.1:8000/process_post"
method="POST">
      First Name: <input type="text"
name="first_name" /> <br />
      Last Name: <input type="text"
name="last_name" />
      <input type="submit" value="Submit" />
    </form>
  </body>
</html>
```

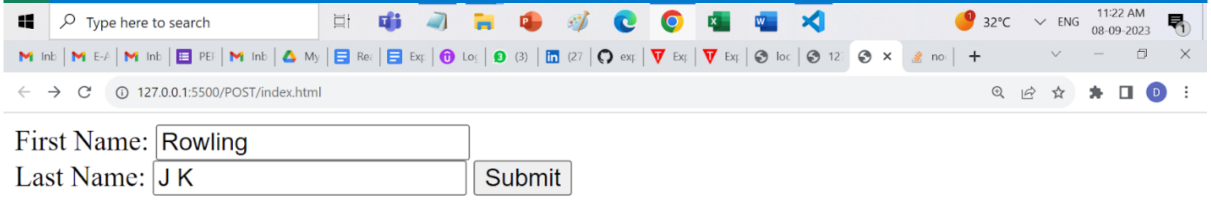
post_ex.js

```
var express = require("express");
var app = express();
var bodyParser = require("body-parser");
// Create application/x-www-form-urlencoded parser
var urlencodedParser = bodyParser.urlencoded({
extended: false });
app.use(express.static("public"));
app.get("/index.html", function (req, res) {
  res.sendFile(__dirname + "/" + "index.html");
});
app.post("/process_post", urlencodedParser,
function (req, res) {
  // Prepare output in JSON format
  response = {
    first_name: req.body.first_name,
    last_name: req.body.last_name,
  };
  console.log(response);
  res.end(JSON.stringify(response));
});
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);
});
```



```
app.post("/process_post", urlencodedParser, fun
  // Prepare output in JSON format
  response = {
    first_name: req.body.first_name,
    last_name: req.body.last_name,
  };
  console.log(response);
  res.end(JSON.stringify(response));
```

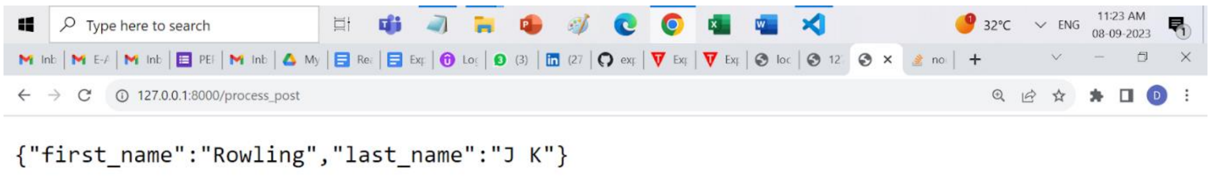
```
E:\HOPE\Trainings\FutureSTEP\FS\Express\POST>node post_ex.js
Example app listening at http://:::8000
{ first_name: 'Rowling', last_name: 'J K' }
```



127.0.0.1:5500/POST/index.html

First Name: Rowling
Last Name: J K Submit

http://127.0.0.1:8000/process_post



127.0.0.1:8000/process_post

```
{"first_name":"Rowling","last_name":"J K"}
```

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)

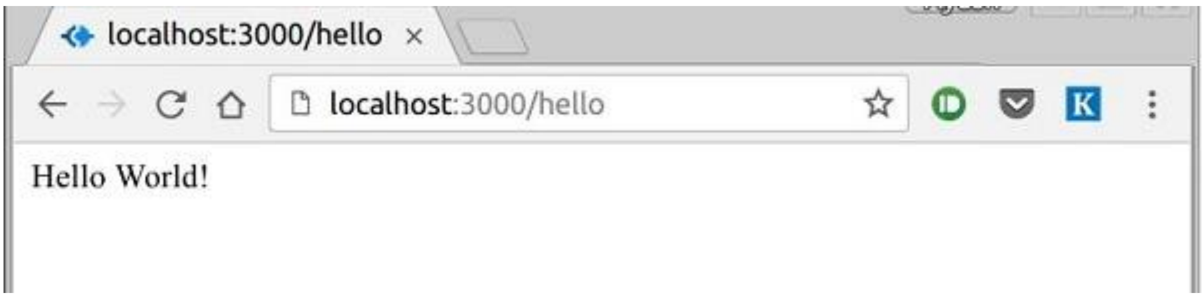
This METHOD can be applied to any one of the HTTP verbs – get, set, put, delete. An alternate method also exists, which executes independent of the request type.
Path is the route at which the request will run.

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.

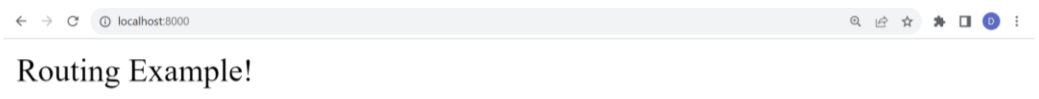


```
Hello World!
```

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

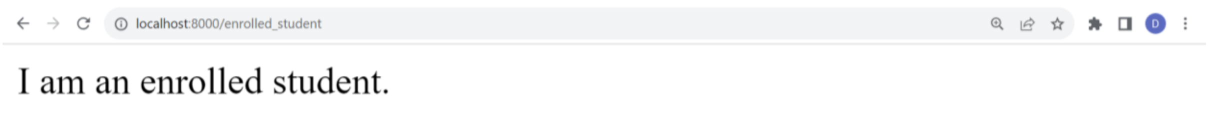
```
var express = require("express");
var app = express();
app.get("/", function (req, res) {
  console.log("Got a GET request for the homepage");
  res.send("Routing Example!");
});
app.post("/post", function (req, res) {
  console.log("Got a POST request for the homepage");
  res.send("I am Impossible! ");
});
app.delete("/del_student", function (req, res) {
  console.log("Got a DELETE request for
/del_student");
  res.send("I am Deleted!");
});
app.get("/enrolled_student", function (req, res) {
  console.log("Got a GET request for
/enrolled_student");
  res.send("I am an enrolled student.");
});
// This responds a GET request for abcd, abxcd,
ab123cd, and so on
app.get("/ab*cd", function (req, res) {
  console.log("Got a GET request for /ab*cd");
  res.send("Pattern Matched.");
});
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);
});
```

http://localhost:8000/



Routing Example!

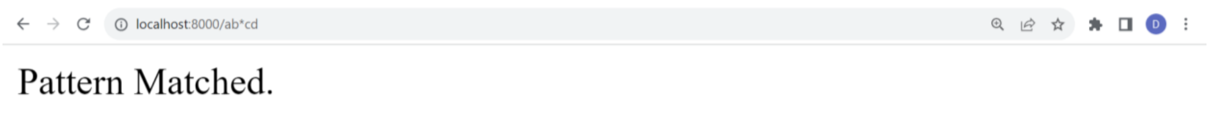
http://localhost:8000/enrolled_student



localhost:8000/enrolled_student

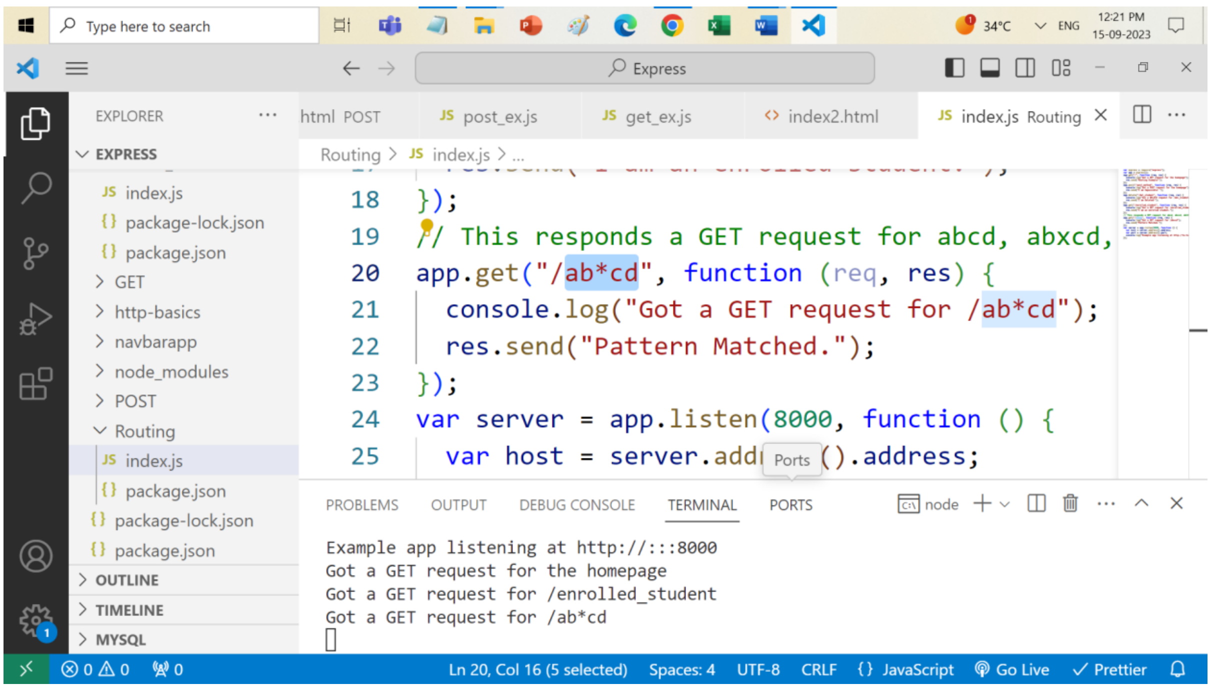
I am an enrolled student.

http://localhost:8000/ab*cd



localhost:8000/ab*cd

Pattern Matched.



```
});
// This responds a GET request for abcd, abxcd,
app.get("/ab*cd", function (req, res) {
  console.log("Got a GET request for /ab*cd");
  res.send("Pattern Matched.");
});
var server = app.listen(8000, function () {
  var host = server.address().address;
```

```
Example app listening at http://:::8000
Got a GET request for the homepage
Got a GET request for /enrolled_student
Got a GET request for /ab*cd
```

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.

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

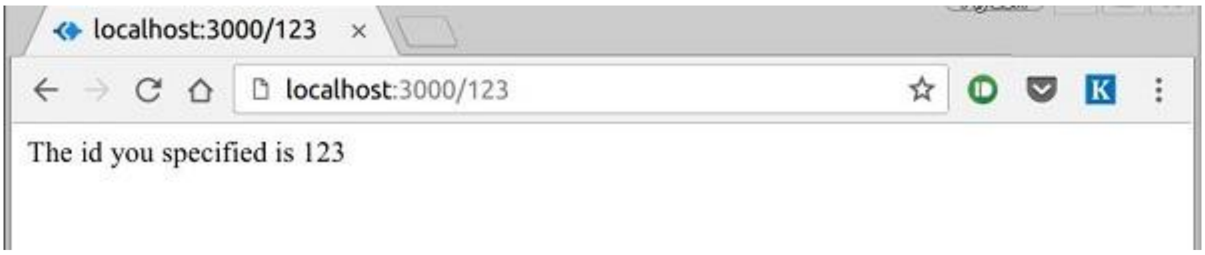
ExpressJS - URL Building

We can 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 process based on them.

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.

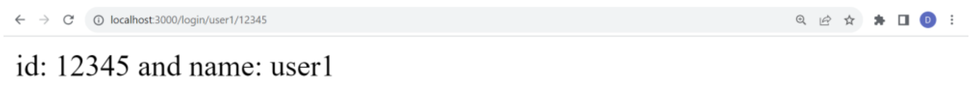


```
The id you specified is 123
```

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('/login/: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/login/user1/12345



```
id: 12345 and name: user1
```

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 '/login' then you need to define it separately.

ExpressJS - Cookies

Cookies are simple, small files/data that are sent to client with a server request and stored on the client side. Every time the user loads the website back, this cookie is sent with the request. This helps us keep track of the user's actions.

The following are the numerous uses of the HTTP Cookies −

- Session management
- Personalization(Recommendation systems)
- User tracking

To use cookies with Express, we need the cookie-parser middleware. To install it, use the following code −

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

Now to use cookies with Express, we will require the cookie-parser. cookie-parser is a middleware which parses cookies attached to the client request object. To use it, we will require it in our index.js file; this can be used the same way as we use other middleware. Here, we will use the following code.

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

cookie-parser parses Cookie header and populates req.cookies with an object keyed by the cookie names. To set a new cookie, let us define a new route in your Express app like −

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

app.get('/', function(req, res){
  res.cookie('name', 'express').send('cookie set'); //Sets name = express
});

app.listen(3000);
```

To check if your cookie is set or not, just go to your browser, fire up the console, and enter −

```
console.log(document.cookie);
```

You will get the output like (you may have more cookies set maybe due to extensions in your browser) −

```
"name = express"
```

The browser also sends back cookies every time it queries the server. To view cookies from your server, on the server console in a route, add the following code to that route.

```
console.log('Cookies: ', req.cookies);
```

Next time you send a request to this route, you will receive the following output.

```
Cookies: { name: 'express' }
```

Adding Cookies with Expiration Time

You can add cookies that expire. To add a cookie that expires, just pass an object with property 'expire' set to the time when you want it to expire. For example,

```
//Expires after 360000 ms from the time it is set.
res.cookie(name, 'value', {expire: 360000 + Date.now()});
```

Another way to set expiration time is using 'maxAge' property. Using this property, we can provide relative time instead of absolute time. Following is an example of this method.

```
//This cookie also expires after 360000 ms from the time it is set.
res.cookie(name, 'value', {maxAge: 360000});
```

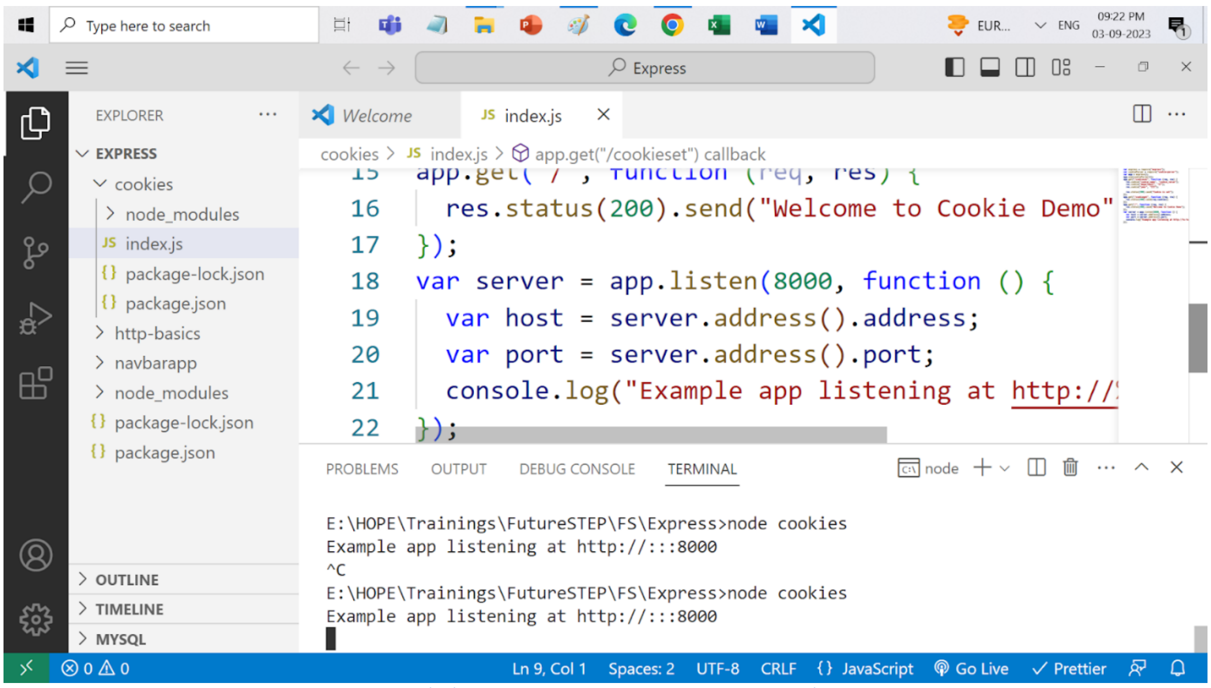
Deleting Existing Cookies

To delete a cookie, use the clearCookie function. For example, if you need to clear a cookie named foo, use the following code.

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

app.get('/clear_cookie_foo', function(req, res){
  res.clearCookie('foo');
  res.send('cookie foo cleared');
});

app.listen(3000);
```

Cookies Sample Code

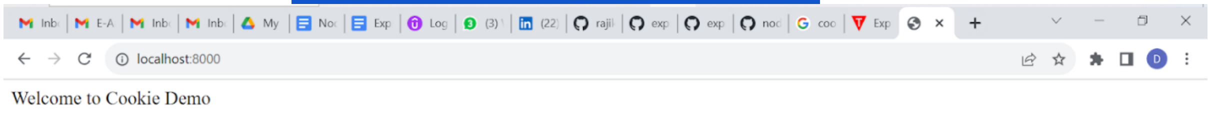
```
mkdir cookies
npm init

npm install cookie-parser
Create file index.js
node cookies //to run
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("department", "IT");
  res.cookie("year", "III");
  res.cookie("likes", "66");

  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 Cookie Demo");
});
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);
});
```



```
app.get("/", function (req, res) {
  res.status(200).send("Welcome to Cookie Demo"
});
var server = app.listen(8000, function () {
  var host = server.address().address;
  var port = server.address().port;
  console.log("Example app listening at http://
});
```

```
E:\HOPE\Trainings\FutureSTEP\FS\Express>node cookies
Example app listening at http://:::8000
^C
E:\HOPE\Trainings\FutureSTEP\FS\Express>node cookies
Example app listening at http://:::8000
```

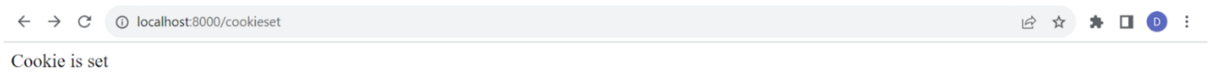
In Browser: http://localhost:8000/



localhost:8000

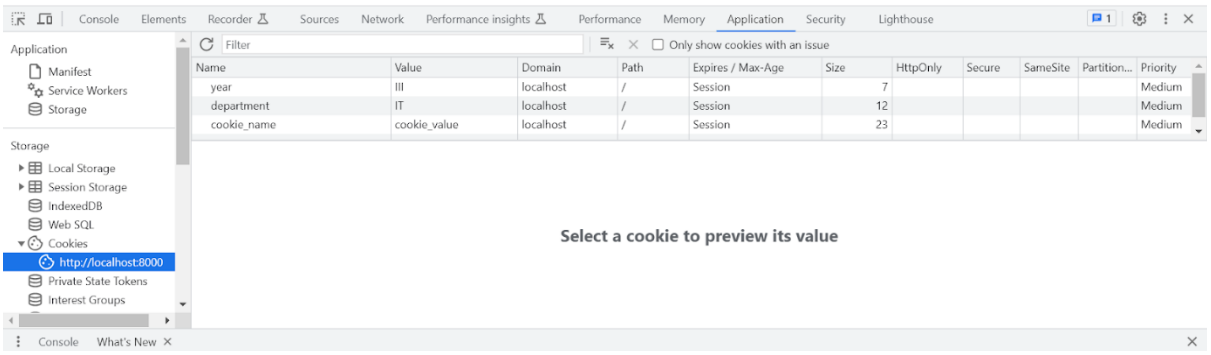
Welcome to Cookie Demo

http://localhost:8000/cookieset



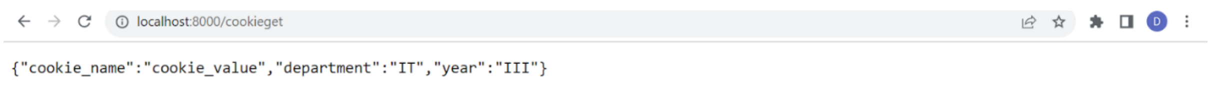
localhost:8000/cookieset

Cookie is set

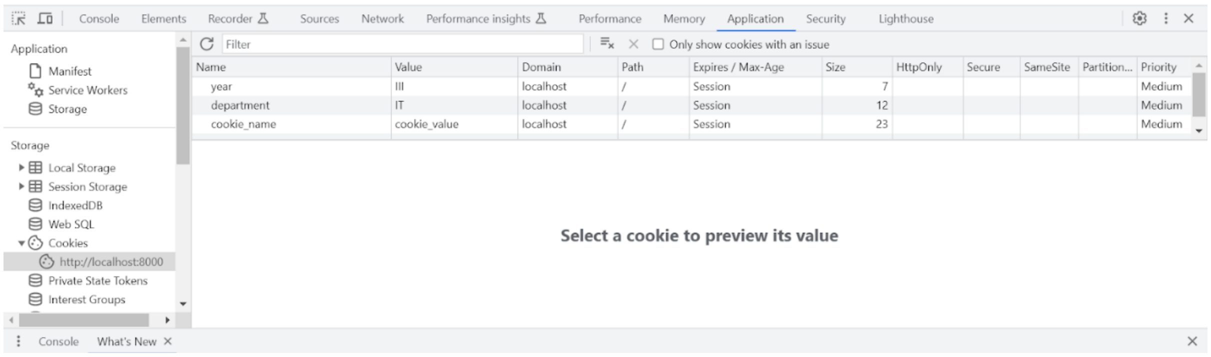


Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Partition...	Priority
year	III	localhost	/	Session	7					Medium
department	IT	localhost	/	Session	12					Medium
cookie_name	cookie_value	localhost	/	Session	23					Medium

Select a cookie to preview its value



```
{"cookie_name":"cookie_value","department":"IT","year":"III"}
```



Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Partition...	Priority
year	III	localhost	/	Session	7					Medium
department	IT	localhost	/	Session	12					Medium
cookie_name	cookie_value	localhost	/	Session	23					Medium

Select a cookie to preview its value

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.

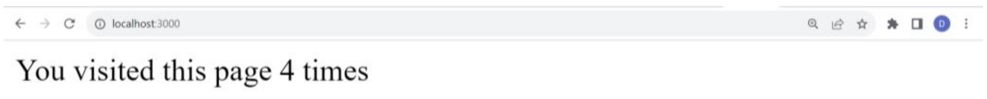
To enable Express-session, so install it using the following code:

npm install --save express-session

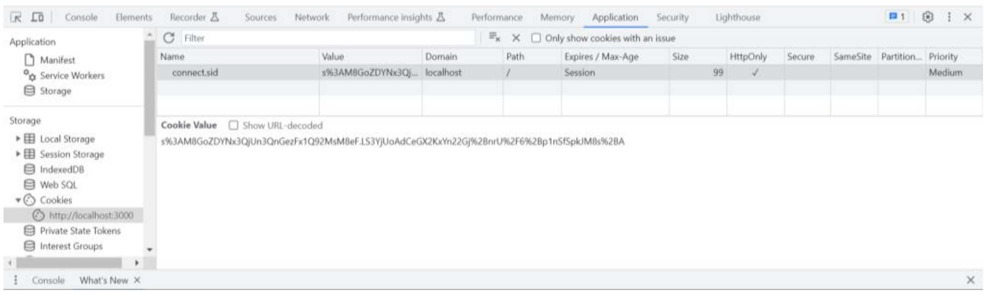
The session and cookie-parser middleware should be put in place. In this example, a default store for storing sessions, i.e., MemoryStore is used. Never use this in production environments. The session middleware handles all things for us, i.e., creating the session, setting the session cookie and creating the session object in req object.

Whenever we make a request from the same client again, we will have their session information stored with us (given that the server was not restarted). We can add more properties to the session object. In the following example, a view counter for a client will be created.

```
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!",
    resave: true,
    saveUninitialized: true,
  })
);
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);
```



You visited this page 4 times



Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Partition...	Priority
connect.sid	s%3AM8GoZDYNx3Qj...	localhost	/	Session	99	✓				Medium

Cookie Value

s%3AM8GoZDYNx3QjUn3QnGezFx1Q92MsM8eF.LS3YjUoAdCeGX2KxYn22Gj%2BnrU%2F6%2Bp1nSfSpkJM8s%2BA

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.

What is a Middleware function

Middleware functions are the functions that access to the request and response object (req, res) in request-response cycle.
A middleware function can perform the following tasks:

- It can execute any code.
- It can make changes to the request and the response objects.
- It can end the request-response cycle.
- It can call the next middleware function in the stack.

Following is a list of possibly used middleware in Express.js app:

- Application-level middleware
- Router-level middleware
- Error-handling middleware
- Built-in middleware
- Third-party middleware

Here is a simple example of a middleware function in action −

```
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);
```

- Now whenever you request any subroute of '/things', only then it will log the time.

Order of Middleware Calls

- One of the most important things about middleware in Express is the order in which they are written/included in your file; the order in which they are executed, given that the route matches also needs to be considered.
- For example, in the following code snippet, the first function executes first, then the route handler and then the end function. This example summarizes how to use middleware before and after route handler; also how a route handler can be used as a middleware itself.

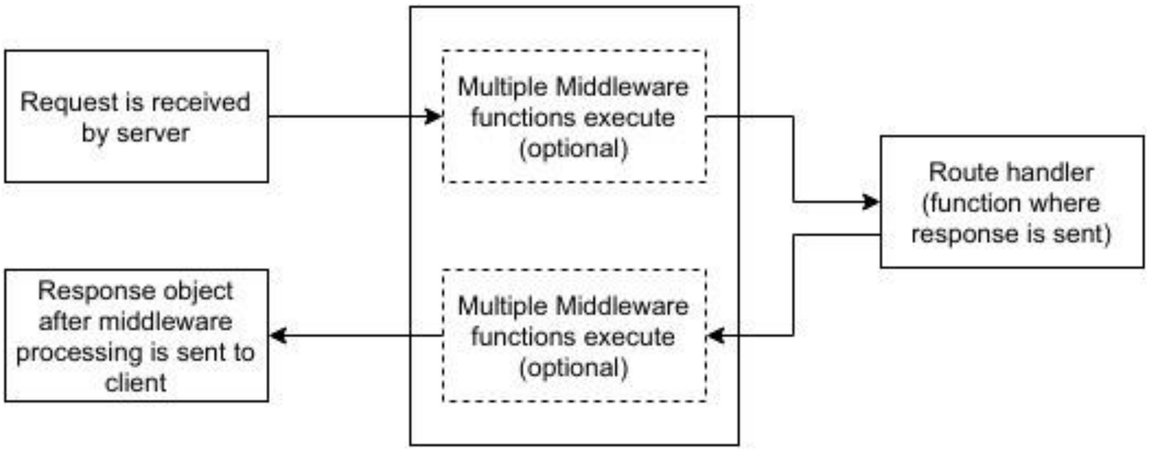
```
var express = require('express');
var app = express();

//First middleware before response is sent
app.use(function(req, res, next){
  console.log("Start");
  next();
});

//Route handler
app.get('/', function(req, res, next){
  res.send("Middle");
  next();
});

app.use('/', function(req, res){
  console.log('End');
});

app.listen(3000);
```

- When we visit '/' after running this code, we receive the response

  as Middle and on our console –

```
Start
End
```

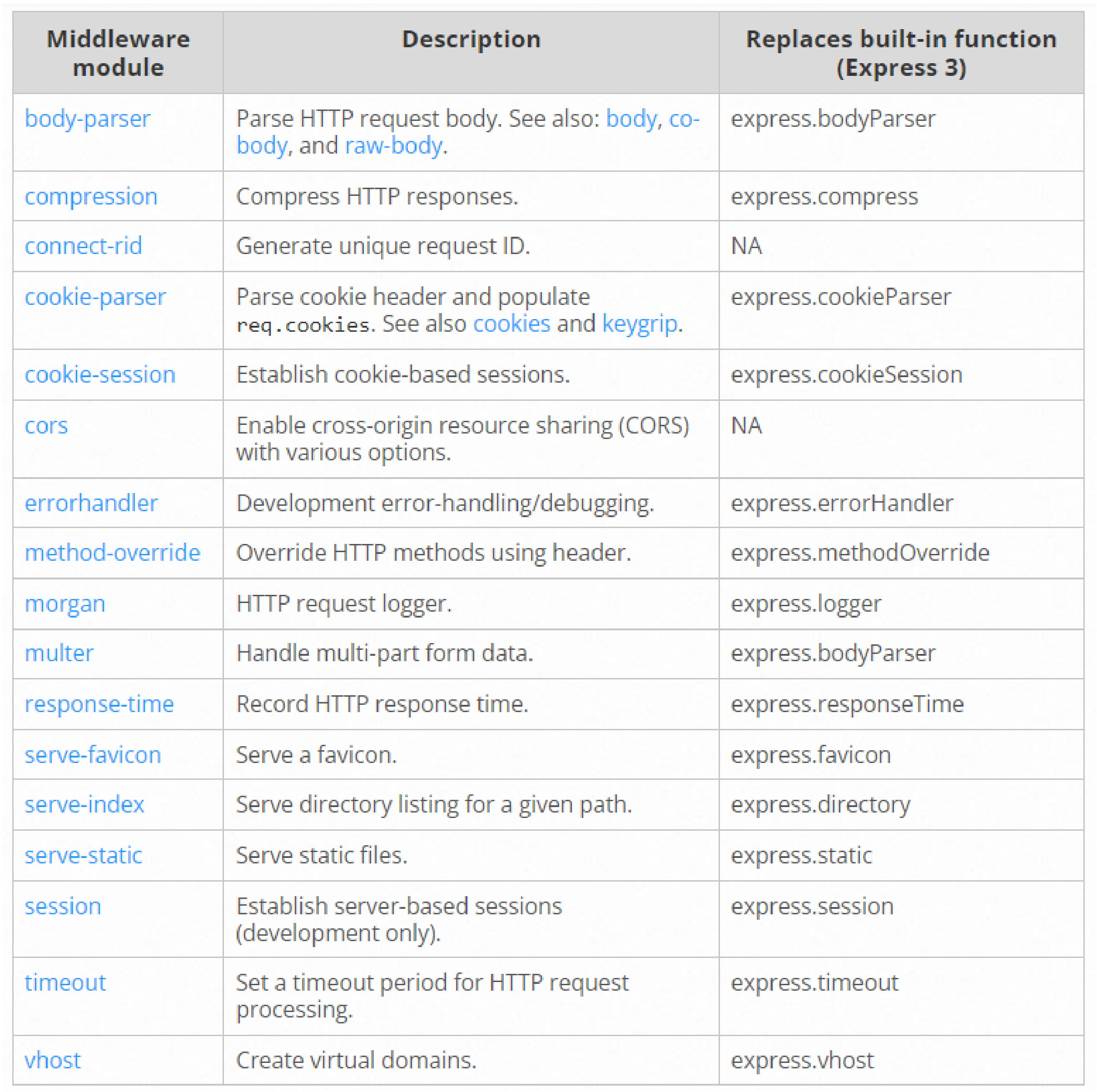
- The following diagram summarizes what we have learnt about middleware –


Request is received by server
Multiple Middleware functions execute (optional)
Route handler (function where response is sent)
Multiple Middleware functions execute (optional)
Response object after middleware processing is sent to client

- Now that we have covered how to create our own middleware, let us discuss some of the most used community created middleware.

Third Party Middleware

- A list of third party middleware for Express.



Middleware module	Description	Replaces built-in function (Express 3)
body-parser	Parse HTTP request body. See also: body, co-body, and raw-body.	express.bodyParser
compression	Compress HTTP responses.	express.compress
connect-rid	Generate unique request ID.	NA
cookie-parser	Parse cookie header and populate `req.cookies`. See also cookies and keygrip.	express.cookieParser
cookie-session	Establish cookie-based sessions.	express.cookieSession
cors	Enable cross-origin resource sharing (CORS) with various options.	NA
errorhandler	Development error-handling/debugging.	express.errorHandler
method-override	Override HTTP methods using header.	express.methodOverride
morgan	HTTP request logger.	express.logger
multer	Handle multi-part form data.	express.bodyParser
response-time	Record HTTP response time.	express.responseTime
serve-favicon	Serve a favicon.	express.favicon
serve-index	Serve directory listing for a given path.	express.directory
serve-static	Serve static files.	express.static
session	Establish server-based sessions (development only).	express.session
timeout	Set a timeout period for HTTP request processing.	express.timeout
vhost	Create virtual domains.	express.vhost

- Following are some of the most commonly used middleware; we will also learn how to use/mount these –

body-parser

- 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())
```

- To view all available options for body-parser, visit its github page.

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())
```

express-session

- It creates a session middleware with the given options. We will discuss its usage in the Sessions section.

Building a REST API & Rendering JSON

REST (Representational State Transfer) is a standard architecture for building and communicating with web services. It typically mandates resources on the web are represented in a text format (like JSON, HTML, or XML) and can be accessed or modified by a predetermined set of operations. Given that we typically build REST APIs to leverage HTTP instead of other protocols, these operations correspond to HTTP methods like GET, POST, or PUT.

On a collection of data, like books for example, there are a few actions we'll need to perform frequently, which boil down to - Create, Read, Update and Delete (also known as CRUD Functionality).

An API (Application Programming Interface), as the name suggests, is an interface that defines the interaction between different software components. Web APIs define what requests can be made to a component (for example, an endpoint to get a list of books), how to make them (for example, a GET request), and their expected responses.

There are a few types of HTTP methods that we need to grasp before building a REST API. These are the methods that correspond to the CRUD tasks:

POST: Used to submit data, typically used to create new entities or edit already existing entities.

GET: Used to request data from the server, typically used to read data.

PUT: Used to completely replace the resource with the submitted resource, typically used to update data.

DELETE: Used to delete an entity from the server.

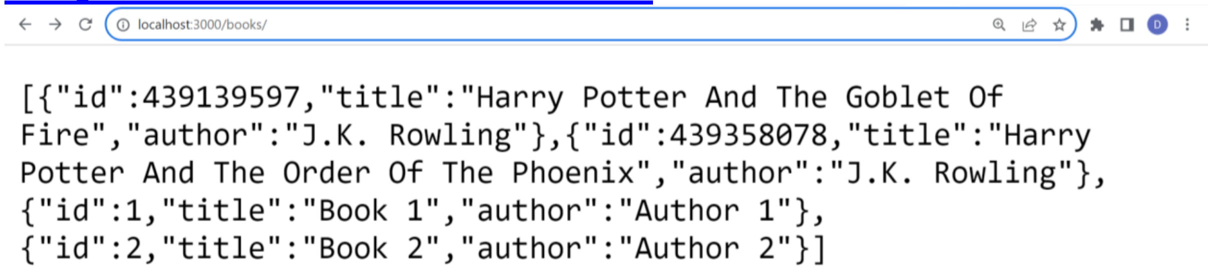
Building a REST API with Node and Express – Books API

Books.js

```
const express = require("express");
const bodyParser = require("body-parser");
const app = express();
const port = 3000;

// Sample data for books
let books = [
  {
    id: 439139597,
    title: "Harry Potter And The Goblet Of Fire",
    author: "J.K. Rowling",
  },
  {
    id: 439358078,
    title: "Harry Potter And The Order Of The
Phoenix",
    author: "J.K. Rowling",
  },
  { id: 1, title: "Book 1", author: "Author 1" },
  { id: 2, title: "Book 2", author: "Author 2" },
];

app.use(bodyParser.json());
```

```
// Get all books
app.get("/books", (req, res) => {
  res.json(books);
});

// Get a specific book by ID
app.get("/books/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const book = books.find((b) => b.id === id);
  if (book) {
    res.json(book);
  } else {
    res.status(404).send("Book not found");
  }
});

// Add a new book
app.post("/books", (req, res) => {
  const newBook = req.body;
  books.push(newBook);
  res.status(201).json(newBook);
});

// Update a book by ID
app.put("/books/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const updatedBook = req.body;
  const index = books.findIndex((b) => b.id === id);
  if (index !== -1) {
    books[index] = updatedBook;
    res.json(updatedBook);
  } else {
    res.status(404).send("Book not found");
  }
});
// Delete a book by ID
app.delete("/books/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const index = books.findIndex((b) => b.id === id);
  if (index !== -1) {
    books.splice(index, 1);
    res.sendStatus(204);
  } else {
    res.status(404).send("Book not found");
  }
});
app.listen(port, () => {
  console.log(`Server is running on port ${port}`);
});
```

http://localhost:3000/books



```
[{"id":439139597,"title":"Harry Potter And The Goblet Of
Fire","author":"J.K. Rowling"},{"id":439358078,"title":"Harry
Potter And The Order Of The Phoenix","author":"J.K. Rowling"},
{"id":1,"title":"Book 1","author":"Author 1"},
{"id":2,"title":"Book 2","author":"Author 2"}]
```

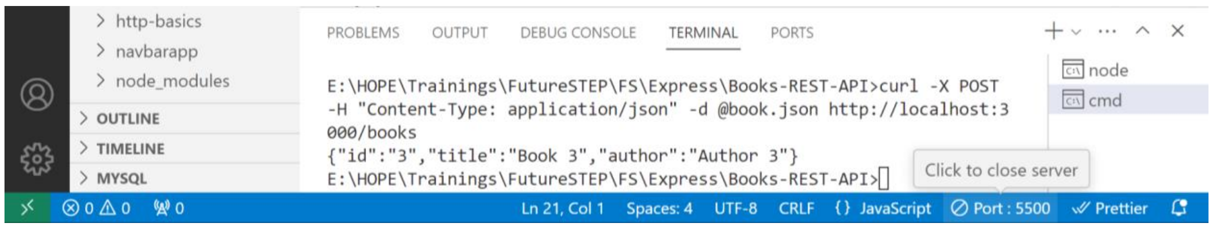
//CREATE

Create a JSON file (e.g., book.json) with the following content:

```
{
  "id": "3",
  "title": "Book 3",
  "author": "Author 3"
}
```

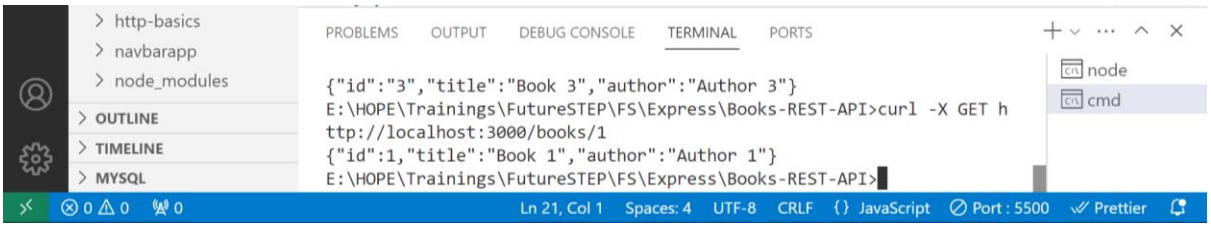
Use curl to send the data from the file as follows:

```
curl -X POST -H "Content-Type: application/json" -d @book.json http://localhost:3000/books
```



```
E:\HOPE\Trainings\FutureSTEP\FS\Express\Books-REST-API>curl -X POST
-H "Content-Type: application/json" -d @book.json http://localhost:3
000/books
{"id":"3","title":"Book 3","author":"Author 3"}
E:\HOPE\Trainings\FutureSTEP\FS\Express\Books-REST-API>
```

//READ or FIND



```
{"id":"3","title":"Book 3","author":"Author 3"}
E:\HOPE\Trainings\FutureSTEP\FS\Express\Books-REST-API>curl -X GET h
ttp://localhost:3000/books/1
{"id":1,"title":"Book 1","author":"Author 1"}
E:\HOPE\Trainings\FutureSTEP\FS\Express\Books-REST-API>
```

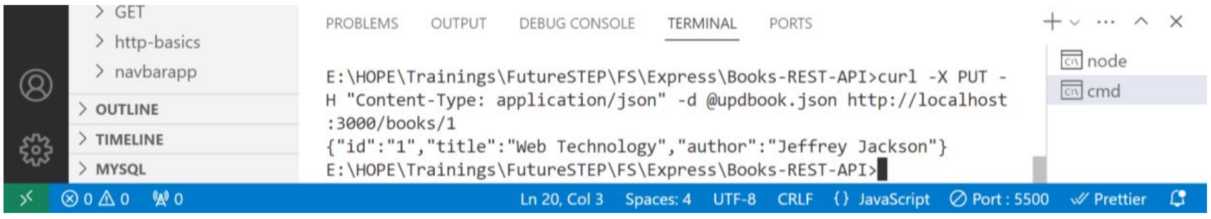
http-basics
navbarapp
node_modules
OUTLINE
TIMELINE
MYSQL
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
node
cmd
Ln 21, Col 1 Spaces: 4 UTF-8 CRLF {} JavaScript Port : 5500 Prettier

//UPDATE

Create a JSON file (e.g., updbook.json) with the following content:

```
{
  "id": "1",
  "title": "Web Technology",
  "author": "Jeffrey Jackson"
}
```

```
curl -X PUT -H "Content-Type: application/json" -d
@updbook.json http://localhost:3000/books/1
```

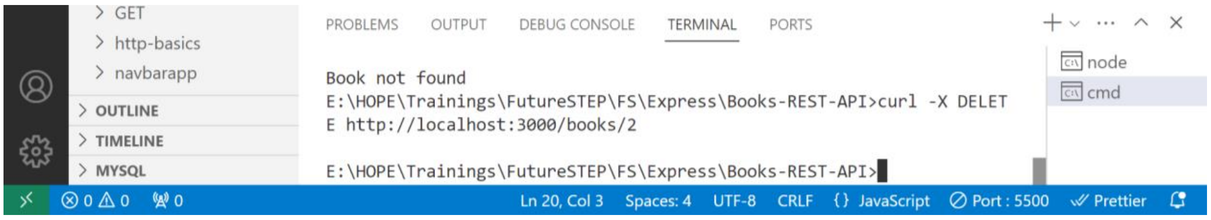


```
E:\HOPE\Trainings\FutureSTEP\FS\Express\Books-REST-API>curl -X PUT -
H "Content-Type: application/json" -d @updbook.json http://localhost
:3000/books/1
{"id":"1","title":"Web Technology","author":"Jeffrey Jackson"}
E:\HOPE\Trainings\FutureSTEP\FS\Express\Books-REST-API>
```

GET
http-basics
navbarapp
OUTLINE
TIMELINE
MYSQL
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
node
cmd
Ln 20, Col 3 Spaces: 4 UTF-8 CRLF {} JavaScript Port : 5500 Prettier

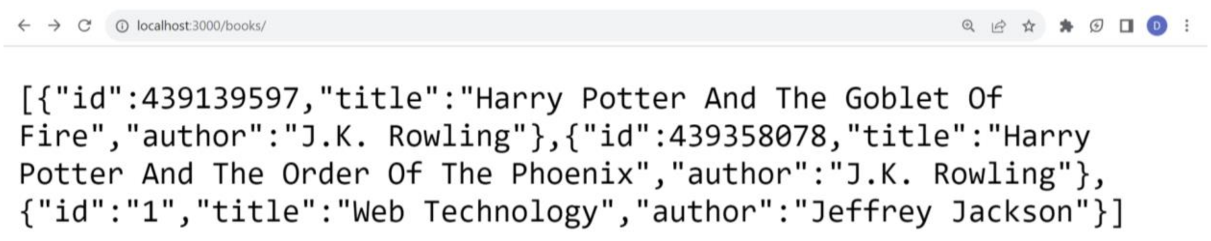
//DELETE

```
curl -X DELETE http://localhost:3000/books/2
```



```
Book not found
E:\HOPE\Trainings\FutureSTEP\FS\Express\Books-REST-API>curl -X DELET
E http://localhost:3000/books/2

E:\HOPE\Trainings\FutureSTEP\FS\Express\Books-REST-API>
```

GET
http-basics
navbarapp
OUTLINE
TIMELINE
MYSQL
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
node
cmd
Ln 20, Col 3 Spaces: 4 UTF-8 CRLF {} JavaScript Port : 5500 Prettier



```
[{"id":439139597,"title":"Harry Potter And The Goblet Of
Fire","author":"J.K. Rowling"},{"id":439358078,"title":"Harry
Potter And The Order Of The Phoenix","author":"J.K. Rowling"},
{"id":"1","title":"Web Technology","author":"Jeffrey Jackson"}]
```