SmartCliff

CAREER MOBILITY SOLUTIONS

We are on a mission to address the digital skills gap for 10 Million+ young professionals, train and empower them to forge a career path into future tech

Express.js

Contents

- Introduction to ExpressJS
- Routing and HTTP Methods
- Express.js Middleware
- Express.js Templating
- Express.js Scaffolding

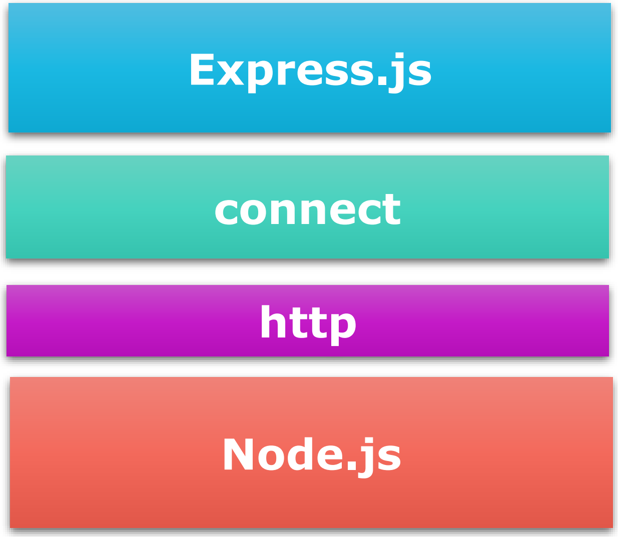
Introduction to Express.js

Getting started with Express.js

- Express.js is a **Node js web application server framework.**
- Express was developed by **TJ Holowaychuk** and is maintained by the **Node.js** foundation and numerous open source contributors.
- It provides various features that make web application development **fast and easy** which otherwise takes more time using only Node.js.
- It is specifically designed for **building single-page, multi-page, and hybrid web applications**.
- The Express.js framework makes it very easy to develop an application which can be used to handle multiple types of requests like the **GET, PUT, and POST and DELETE requests.**

Getting started with Express.js

- Express.js is based on the **Node.js** middleware module called **connect** which in turn uses **http** module.
- So, any middleware which is based on **connect** will also work with **Express.js**.



Express.js

connect

http

Node.js

Advantages of Express.js

- Makes Node.js web application development fast and easy.
- Easy to configure and customize.
- Allows to define routes of your application based on HTTP methods and URLs.
- Includes various middleware modules which can use to perform additional tasks on request and response.
- Easy to integrate with different template engines like Jade, Vash, EJS etc.
- Allows to define an error handling middleware.
- Easy to serve static files and resources of our application.
- Allows to create REST API server.
- Easy to connect with databases such as MongoDB, Redis, MySQL

Install Express.js

- Express.js can be installed using **npm**.
- The following command will install latest version of express.js globally on the machine so that every Node.js application on the machine can use it.

```
npm install -g express
```

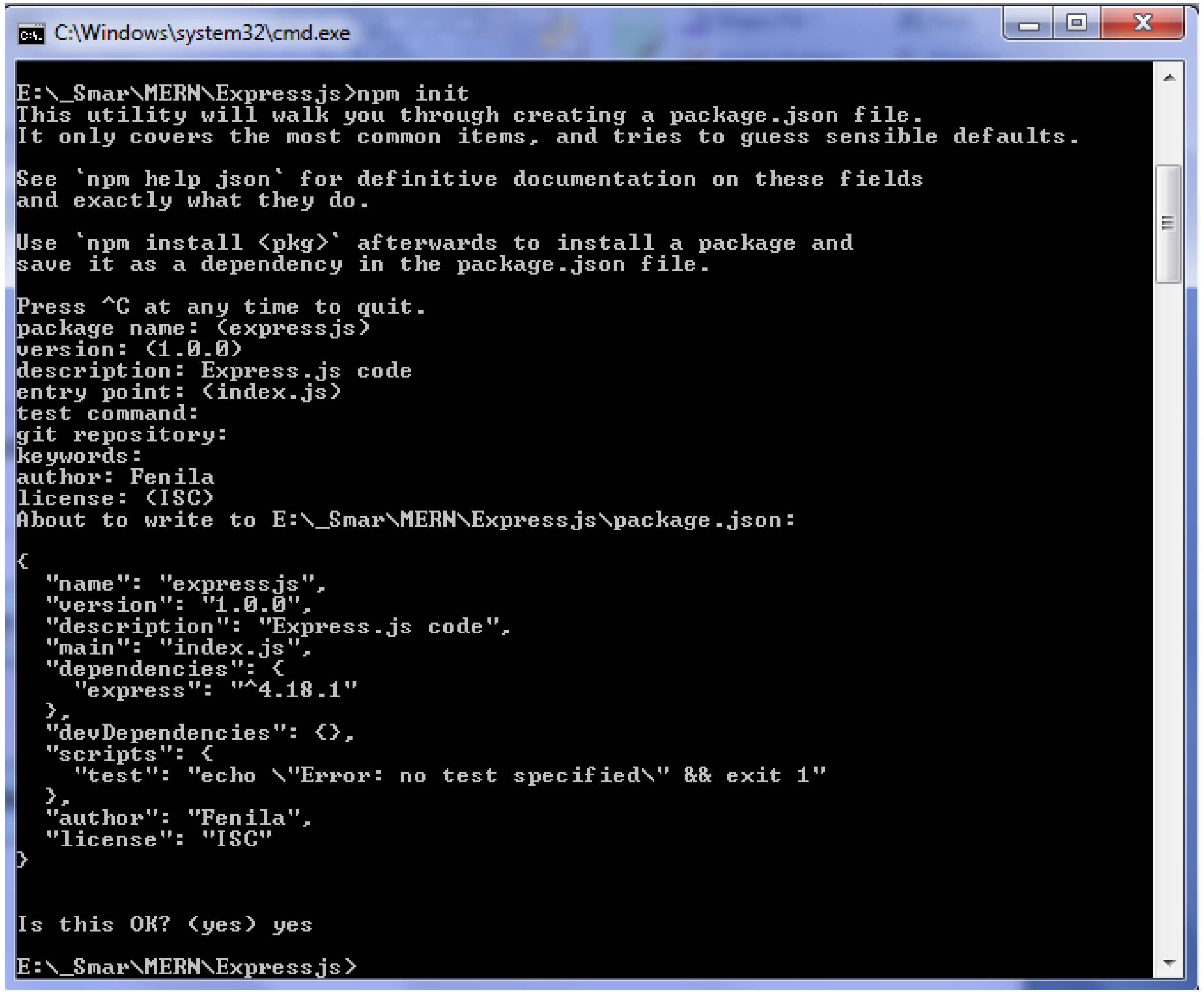
- The following command will install latest version of express.js local to the project folder.

```
E:\_Smart\MERN\Expressjs>   npm install express --save
```

- --save will update the **package.json** file by specifying express.js dependency.
- **npm** creates a **package-lock.json** file that includes detailed information about the modules that have installed which helps keep things fast and secure.

Install Express.js

- It will ask the following information:



```
E:\_Smar\MERN\Expressjs>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (expressjs)
version: (1.0.0)
description: Express.js code
entry point: (index.js)
test command:
git repository:
keywords:
author: Fenila
license: (ISC)
About to write to E:\_Smar\MERN\Expressjs\package.json:

{
  "name": "expressjs",
  "version": "1.0.0",
  "description": "Express.js code",
  "main": "index.js",
  "dependencies": {
    "express": "^4.18.1"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Fenila",
  "license": "ISC"
}


Is this OK? (yes) yes

E:\_Smar\MERN\Expressjs>
```

Note:

Just keep pressing enter, and enter the details whatever we need to add like description, author name, etc.

Install Express.js

- Now **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 --save
```

- 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
```

Basic Express.js app

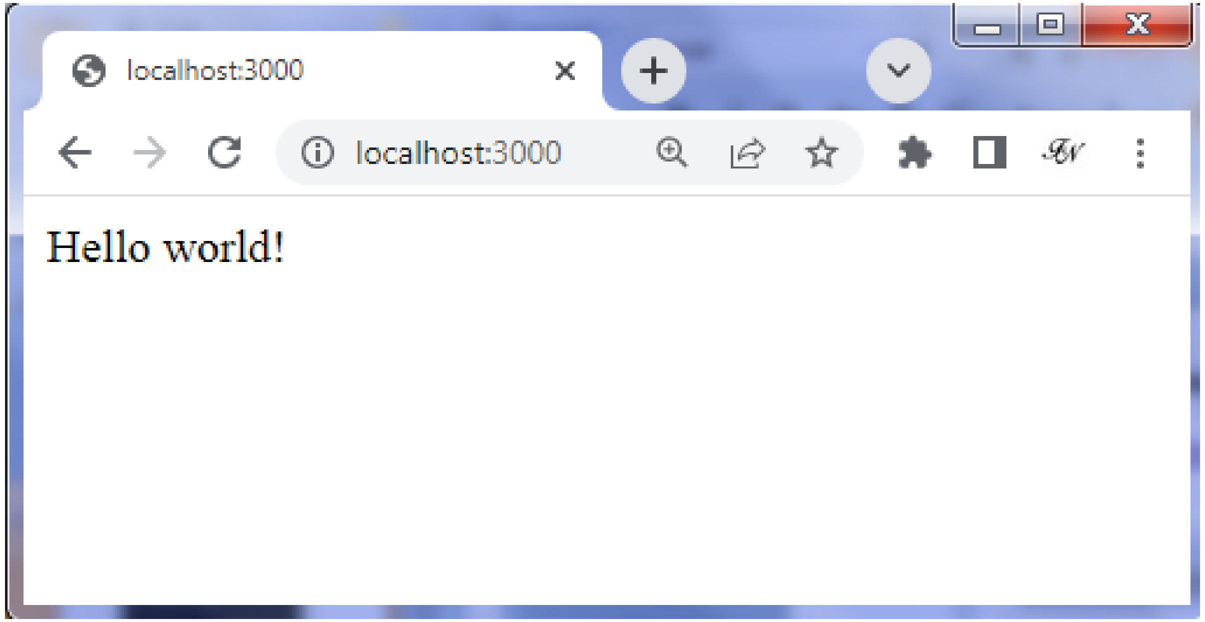
- Lets create a file called **index.js** and type the following in it.

Filename: index.js

```
var express = require('express');  //imported Express.js module using require() function
var app = express();
app.get('/', function(req, res){
        res.send("Hello world!");
});
app.listen(3000);
```

Basic Express.js app

- Run the command **nodemon index.js** in the terminal.
- This will start the server. To test this app, open the browser and go to **http://localhost:3000** and a message will be displayed as in the following screenshot.



```
Hello world!
```

Basic Express.js app

How the App Works?

- The first line imports **Express i**n 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)** 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.

Basic Express.js app

How the App Works?

- **res.send()** 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]])** function binds and listens for connections on the specified host and port. Port is the only required parameter here.

Routing and HTTP Methods

Routing

- ***Routing*** refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).
- Each route can have one or more handler functions, which are executed when the route is matched.
- Route definition takes the following structure:

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

  - **app** is an instance of express.
  - **METHOD** is an HTTP request method, in lowercase get, post, put, delete.
  - **PATH** is a path on the server.
  - **HANDLER** is the callback function that is executed when the matching request type is found on the relevant route.

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 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.

Routing - GET route

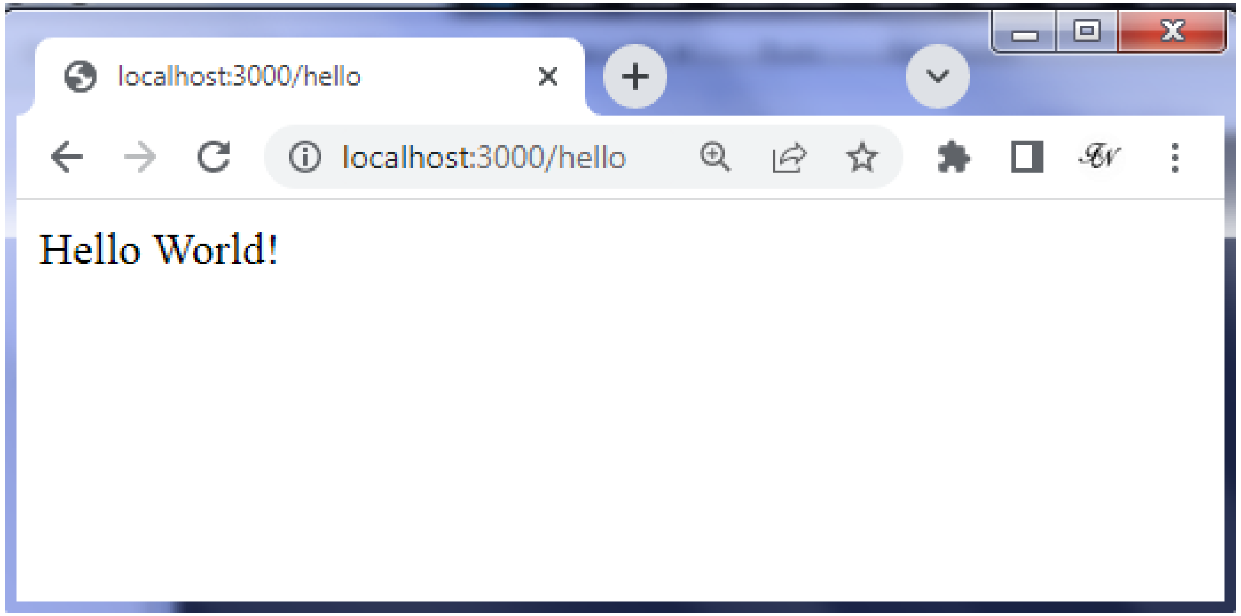
- Lets create a file called **get_route.js** and use app object to define GET route

Filename: get_route.js

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

Routing - Example

- 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.

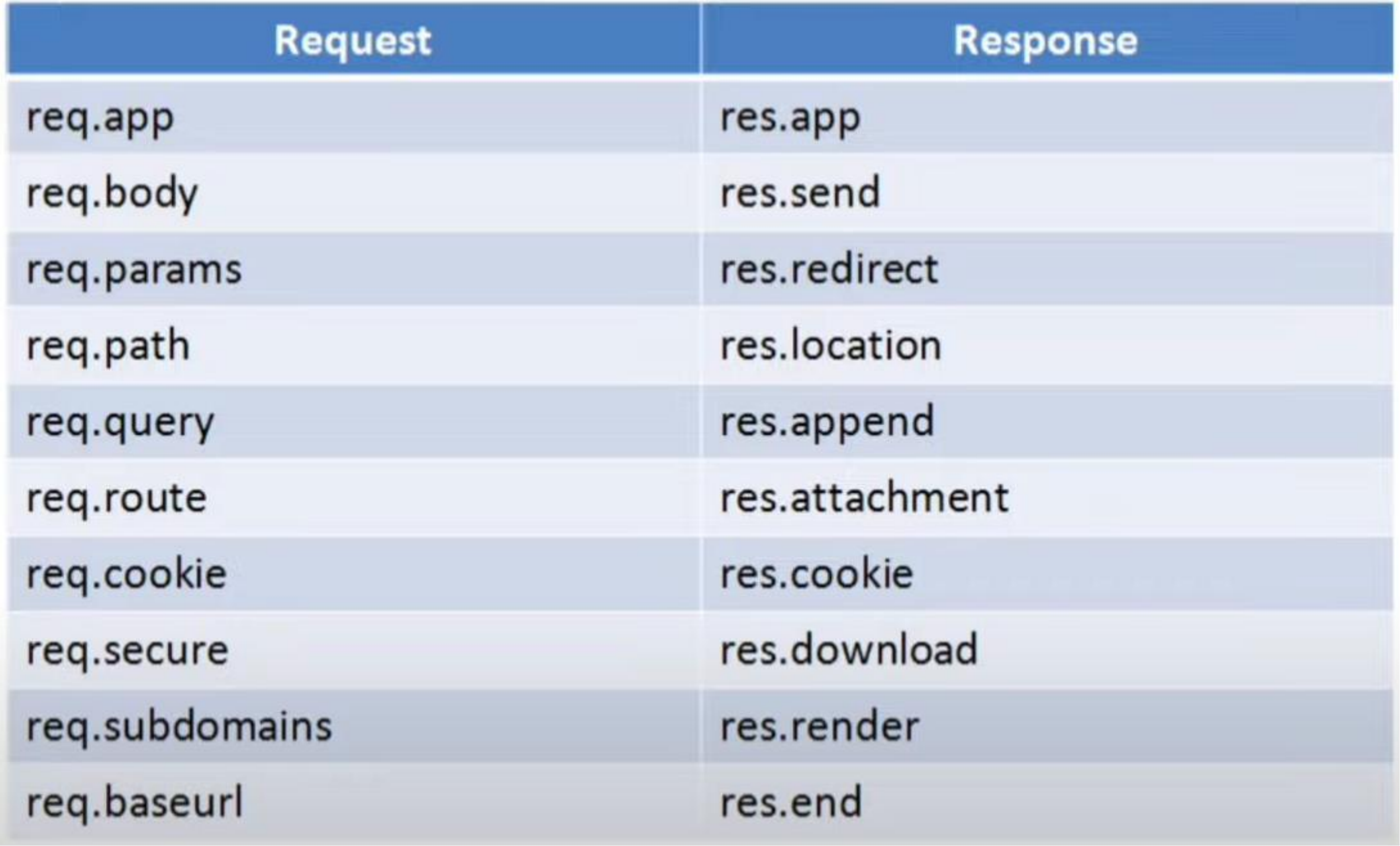


Hello World!

Routing - GET route

- **app.get()** is a function that tells the server what to do when a get request at the given route is called.
- It has a callback function (req, res) that listen to the incoming request req object and respond accordingly using res response object.
- Both **req** and **res** are made available to us by the Express framework.
- The **req** object represents the HTTP request and has properties for the request query string, parameters, body, and HTTP headers.
- The **res** object represents the HTTP response that an Express app sends when it gets an HTTP request.
- In our case, we are sending a text Hello World whenever a request is made to the route /.
- Lastly, app.listen() is the function that starts a port and host, in our case the localhost for the connections to listen to incoming requests from a client. We can define the port number such as 3000.

Request and Response Object Properties



Request	Response
req.app	res.app
req.body	res.send
req.params	res.redirect
req.path	res.location
req.query	res.append
req.route	res.attachment
req.cookie	res.cookie
req.secure	res.download
req.subdomains	res.render
req.baseurl	res.end

Response Object Properties

- Some properties of response object:

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

Response Object Methods

Response Append Method

- Syntax:

```
res.append(field, [value])
```

- Response append method appends the specified value to the HTTP response header field. That means if the specified value is not appropriate so this method redress that.

- Example :

```
res.append('Link', ['<http://localhost/>', '<http://localhost:3000/>']);
res.append('Warning', '299 Miscellaneous warning');
```

Response Object Methods

Response Get Method

- Syntax **:**

```
res.get(field)
```

- res.get method provides HTTP response header specified by field.
- Example **:**

```
res.get('Content-Type');
```

Response Attachment Method

- Syntax :

```
res.attachment(‘path/to/js_pic.png’);
```

- Response attachment method allows you to send a file as an attachment in the HTTP response.
- Example :

```
res.attachment('path/to/js_pic.png');
```

Response Object Methods

Response Download Method

- Syntax:

```
res.download(path [, filename] [, fn])
```

- res.download method is transfer file at path as an “attachment” and the browser to prompt user for download
- Example:

```
res.download('/report-12345.pdf');
```

Response Cookie Method

- Syntax:

```
res.cookie(name, value [, options])
```

- It is used to set a cookie name to value. The value can be a string or object converted to JSON.
- Example :

```
res.cookie('name', 'alish', { domain: '.google.com', path: '/admin', secure: true });

res.cookie('Section', { Names: [sonica,riya,ronak] });

res.cookie('Cart', { items: [1,2,3] }, { maxAge: 900000 });
```

Response Object Methods

Response End Method

- Syntax:

```
res.end([data] [, encoding])
```

- Response end method is used to end the response process.
- Example :

```
res.end();

res.status(404).end();
```

Response JSON Method

- Syntax :

```
res.json([body])
```

- Response JSON method returns the response in JSON format.
- Example :

```
res.json(null)

res.json({ name: 'alish' })
```

Response Object Methods

Response Render Method

- Syntax

```
:   res.render(view [, locals] [, callback])
```

- Response render method renders a view and sends the rendered HTML string to the client.
- Example :

```
// send the rendered view to the client
res.render('index');
// pass a local variable to the view
res.render('user', { name: 'monika' },
    function(err, html) {
 // ...
});
```

Response Object Methods

Response Status Method

- Syntax:

```
res.status(code)
```

- res.status method sets an HTTP status for the response.
- Example :

```
res.status(403).end();

res.status(400).send('Bad Request');
```

Response Type Method

- Syntax :

```
res.type(type)
```

- res.type method sets the content-type HTTP header to the MIME type.
- Example :

```
res.type('.html');          // => 'text/html'

res.type('html');           // => 'text/html'

res.type('json');           // => 'application/json'
```

```
res.type('application/json');

// => 'application/json'

res.type('png');            // => image/png:
```

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) {

  // --

})
```

Request Object Properties

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

Properties	Description
req.app	This is used to hold a reference to the instance of the express application that is using the middleware.
req.baseurl	It specifies the URL path on which a router instance was mounted.
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.
req.cookies	When we use cookie-parser middleware, this property is an object that contains cookies sent by the request.
req.fresh	It specifies that the request is "fresh." it is the opposite of req.stale.
req.hostname	It contains the hostname from the "host" http header.
req.ip	It specifies the remote IP address of the request.

Request Object Properties

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

Properties	Description
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.
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.
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 {}.
req.path	It contains the path part of the request URL.
req.protocol	The request protocol string, "http" or "https" when requested with TLS.
req.query	An object containing a property for each query string parameter in the route.
req.route	The currently-matched route, a string.

Request Object Properties

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

Properties	Description
req.secure	A Boolean that is true if a TLS connection is established.
req.signedcookies	When using cookie-parser middleware, this property contains signed cookies sent by the request, unsigned and ready for use.
req.stale	It indicates whether the request is "stale," and is the opposite of req.fresh.
req.subdomains	It represents an array of subdomains in the domain name of the request.
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

There are various types of request object method, these methods are:

req.is(type)

- If the incoming request is “CONTENT-TYPE”, this method returns true. HTTP header field matches the MIME type by the type parameter.
- Example:

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

Request Object Methods

req.param(name [, defaultValue])

- req.param method is used to fetch the value of param name when present.
- Example :

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

URL Building: Dynamic Routing

- Dynamic Routing allows us to pass parameters and values in routes on the basis of which the server processes our request and sends back the response.
- This is not fixed or static.
- The value of parameter can be changed.
- For understanding purpose, let’s see Facebook profile link https://fb.com/itsaareez1 where itsaareez1 is profile’s username.
- Every user has a unique username so, this value will vary according to user. This is dynamic routing.
- We use : to make any route dynamic

URL Building: Dynamic Routing

Example :

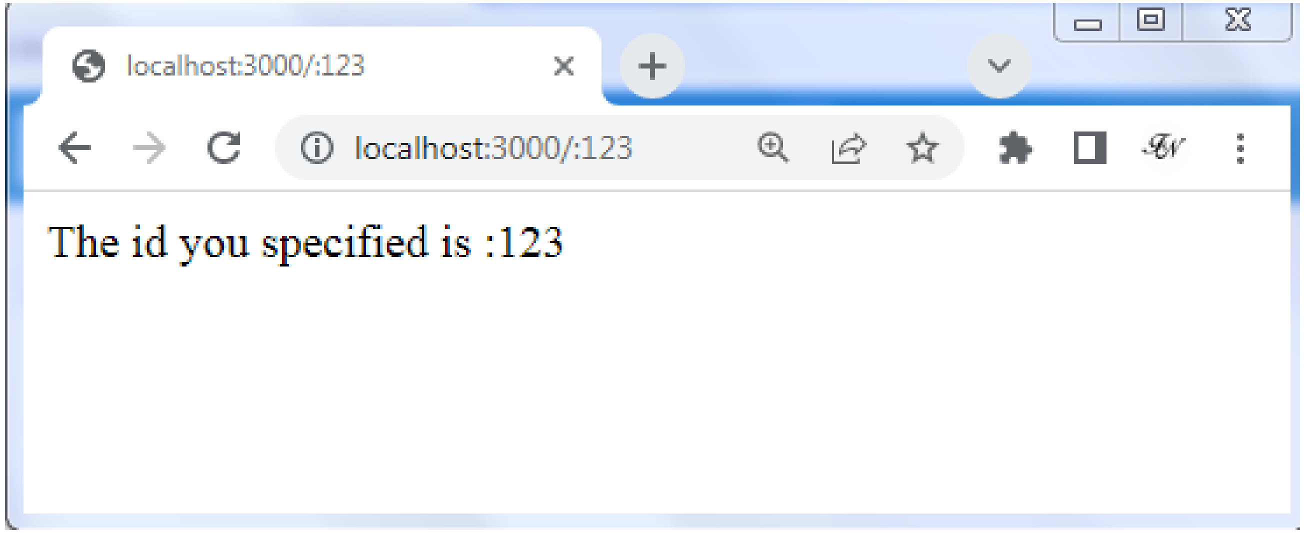
Filename: dynamic_route.js

```
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);
```

URL Building: Dynamic Routing

Run the express app using command nodemon dynamic_route.js

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



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

URL Building: Dynamic Routing

- We 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
- Example

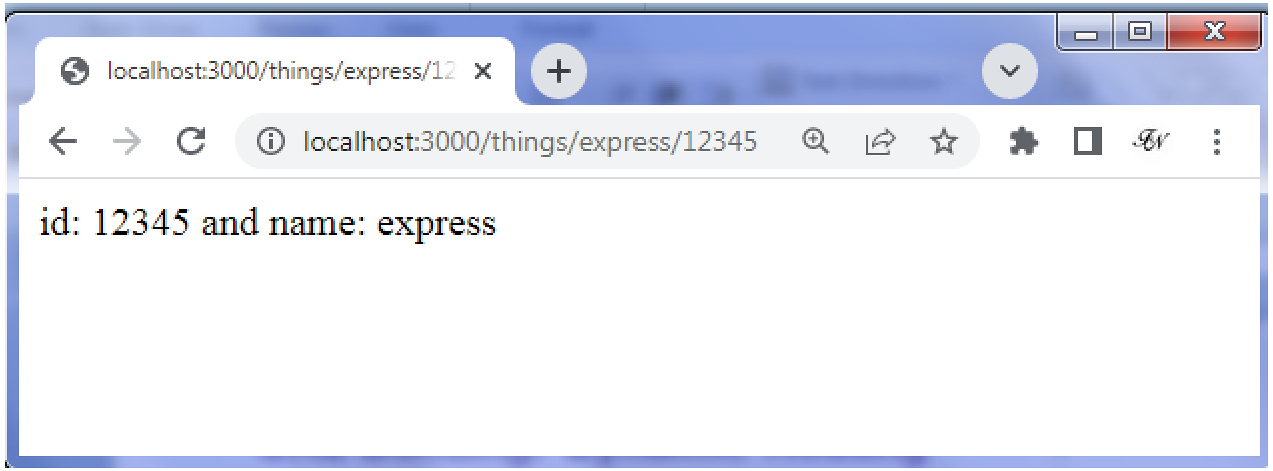
Filename: dynamic_route2.js

```
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);
```

URL Building: Dynamic Routing

Run the express app using command nodemon dynamic_route2.js

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



localhost:3000/things/express/12
localhost:3000/things/express/12345

```
id: 12345 and name: express
```

We 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.

Configure Routes

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

- Like if we have POST, GET, PUT, DELETE, etc, request made to any specific route, let say */user*, so instead to defining different API's like app.post('/user'), app.get('/user'), etc, we can define single API **app.all('/user')** which will accept all type of HTTP request.

```
app.all( path, callback )
```

  - **Path**: It is the path for which the middleware function is called.
  - **Callback** is the function executed when the route is matched.

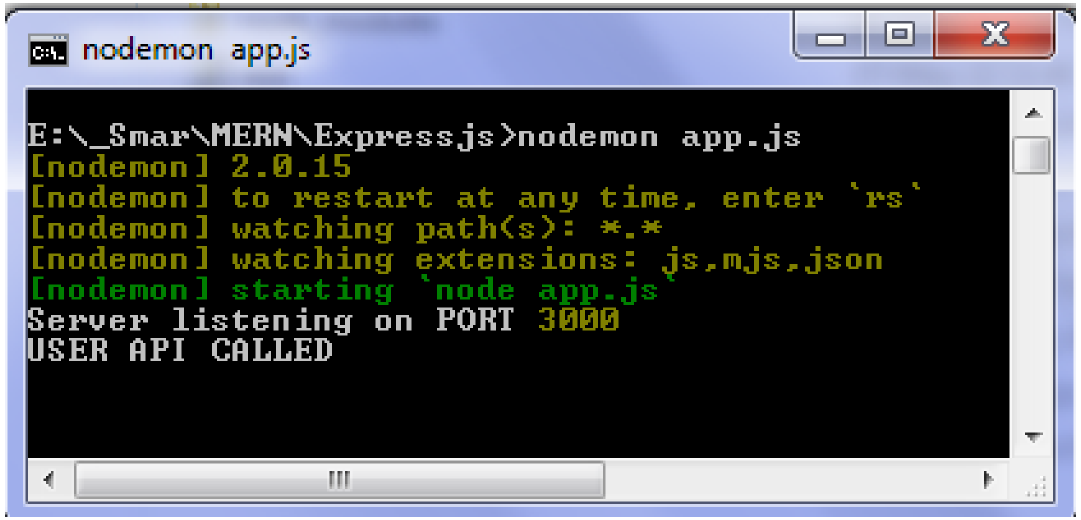
Configure Routes

Filename: app.js

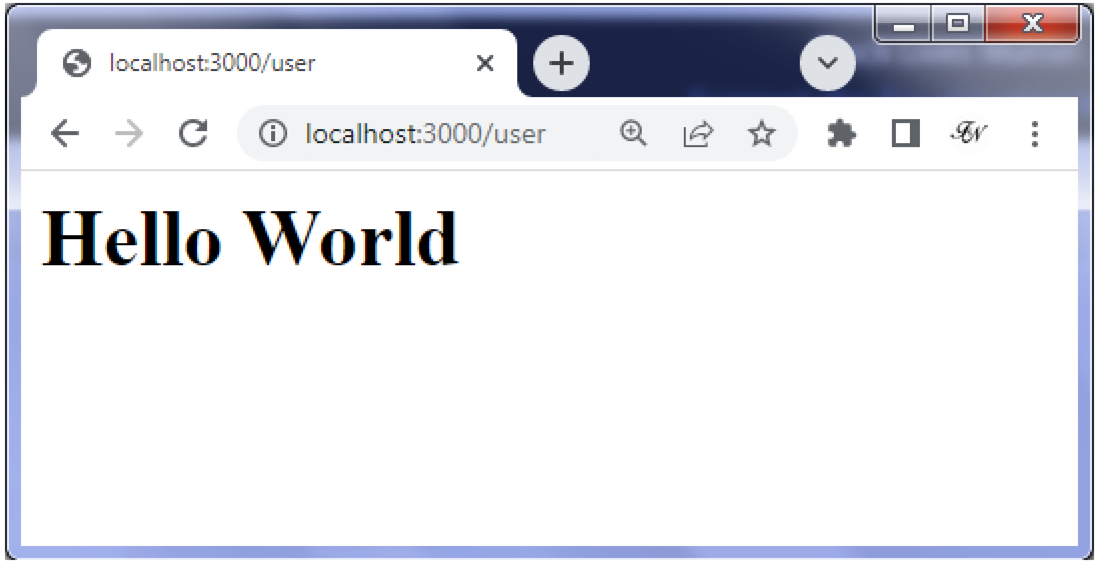
```
var express = require('express');

var app = express();

var PORT = 3000;

app.all('/user', function (req, res) {

    console.log('USER API CALLED');

    res.send('<html><body><h1>Hello World</h1></body></html>');

});


app.listen(PORT, function(){

    console.log("Server listening on PORT", PORT);

});
```

Configure Routes

- Run the command **nodemon app.js** and point the browser to ***http://localhost:3000*** and see the following result.



```
E:\_Smar\MERN\Expressjs>nodemon app.js
[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node app.js`
Server listening on PORT 3000
USER API CALLED
```



Hello World

Route parameters

- Route parameters are named URL segments that are used to capture the values specified at their position in the URL. The name of route parameters must be made up of “word characters” ([A-Za-z0-9_]).
- The captured values are populated in the req.params object, with the name of the route parameter specified in the path as their respective keys.

Route path: /users/:userId/books/:bookId

Request URL: http://localhost:3000/users/34/books/8989

req.params: { "userId": "34", "bookId": "8989" }

- To define routes with route parameters, simply specify the route parameters in the path of the route:

```
app.get('/users/:userId/books/:bookId', function(req, res) {
        res.send(req.params)
})
```

Route parameters

- Since the hyphen (-) and the dot (.) are interpreted literally, they can be used along with route parameters for useful purposes.

Route path: **/flights/:from-:to**

Request URL: **http://localhost:3000/flights/LAX-SFO**

req.params: **{ "from": "LAX", "to": "SFO" }**

Route path: **/plantae/:genus.:species**

Request URL: **http://localhost:3000/plantae/Prunus.persica**

req.params: **{ "genus": "Prunus", "species": "persica" }**

Route parameters

- To have more control over the exact string that can be matched by a route parameter, we can append a regular expression in parentheses (**()**):

Route path: **/user/:userId(\d+)**

Request URL: **http://localhost:3000/user/42**

req.params: **{"userId": "42"}**

- Because the regular expression is usually part of a literal string, be sure to escape any \ characters with an additional backslash, for example \\d+.

Route parameters

Filename: route-params.js

```
var express = require('express');
var app = express();
app.get('/user/:userId/books/:bookid', (req, res) => {
  req.params; // { userId: '42' }
  res.json(req.params);
});
app.get('/flights/:from-:to', (req, res) => {
  req.params; // { "from": "LAX", "to": "SFO" }
  res.json(req.params);
});
```

Route parameters

```
app.get('/plantae/:genus.:species', (req, res) => {
  req.params; // { "genus": "Prunus", "species": "persica" }
  res.json(req.params);
});
app.get('/user/:userId', (req, res) => {
  req.params; // {"userId": "42"}
  res.json(req.params);
});
var server = app.listen(3000, function () {
    console.log('Node server is running..');
});
```

Handle POST Request

- To handle HTTP POST request and get data from the submitted form.
- First, create index.html file in the root folder of your application and write the following HTML code in it.

Filename: index.html

```
<!DOCTYPE html>
<html >
 <head>
 <meta charset="utf-8" />
<title></title>
</head>
<body>
<form action="/submit-data" method="post">
 First Name: <input name="firstName" type="text" /> <br />
Last Name: <input name="lastName" type="text" /> <br />
<input type="submit" /> </form> </body> </html>
```

Handle POST Request

Body Parser

- To handle HTTP POST request in Express.js version 4 and above, we need to install middleware module called **body-parser.**
- The middleware was a part of Express.js earlier but now we have to install it separately.
- This body-parser module parses the JSON, buffer, string and url encoded data submitted using HTTP POST request. Install body-parser using NPM as shown below.

```
npm install body-parser --save
```

Handle POST Request

Import body-parser and get the POST request data as shown below.

Filename: post_route.js

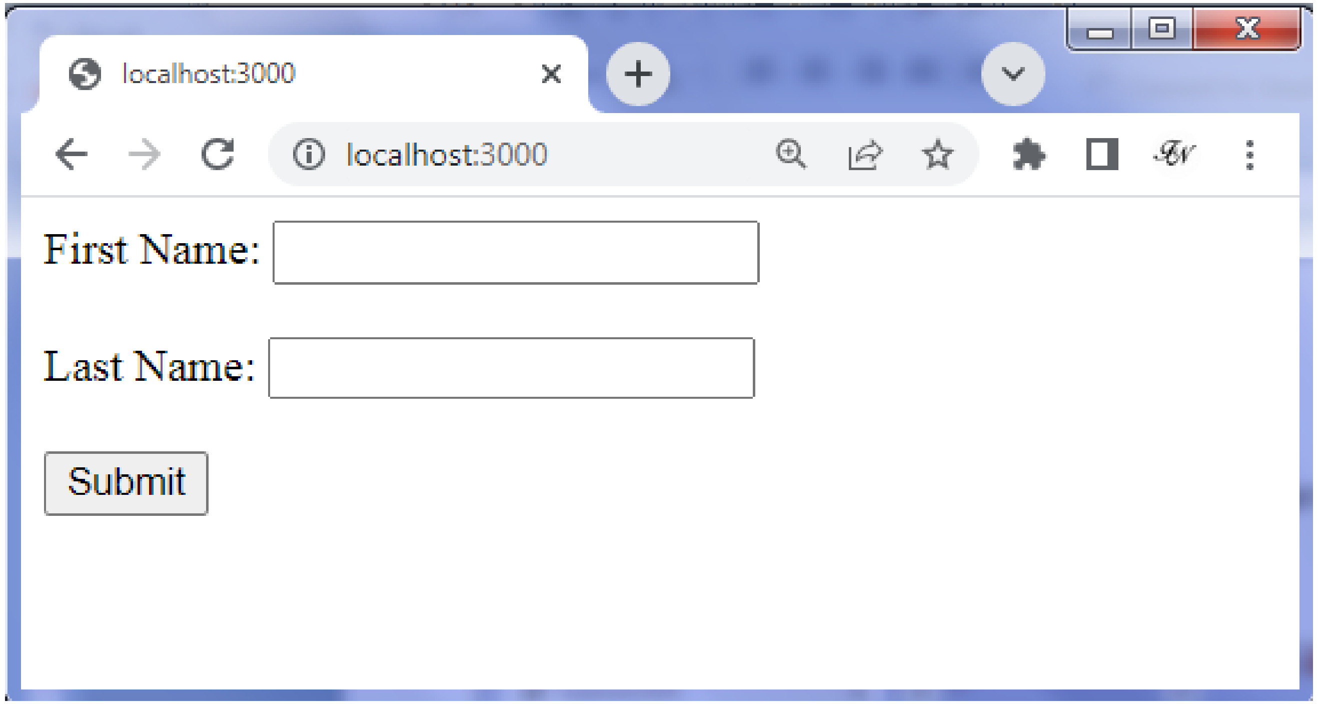
```
var express = require('express');
var app = express();
var bodyParser = require("body-parser");
app.use ( bodyParser.urlencoded ( { extended: false }));

app.get('/', function (req, res) {
   res.sendFile('E:/_Smar/MERN/Expressjs/index.html');
});

app.post('/submit-data', function (req, res) {
   var name = req.body.firstName + ' ' + req.body.lastName;

   res.send(name + ' Submitted Successfully!');
});

var server = app.listen(3000, function () {
   console.log('Node server is running..');
});
```

In this example,

- POST data can be accessed using req.body.
- The req.body is an object that includes properties for each submitted form. Index.html contains firstName and lastName input types, so we can access it using req.body.firstName and req.body.lastName.

Handle POST Request

- Run the command **nodemon post_route.js** and point the browser to ***http://localhost:3000*** and see the following result.



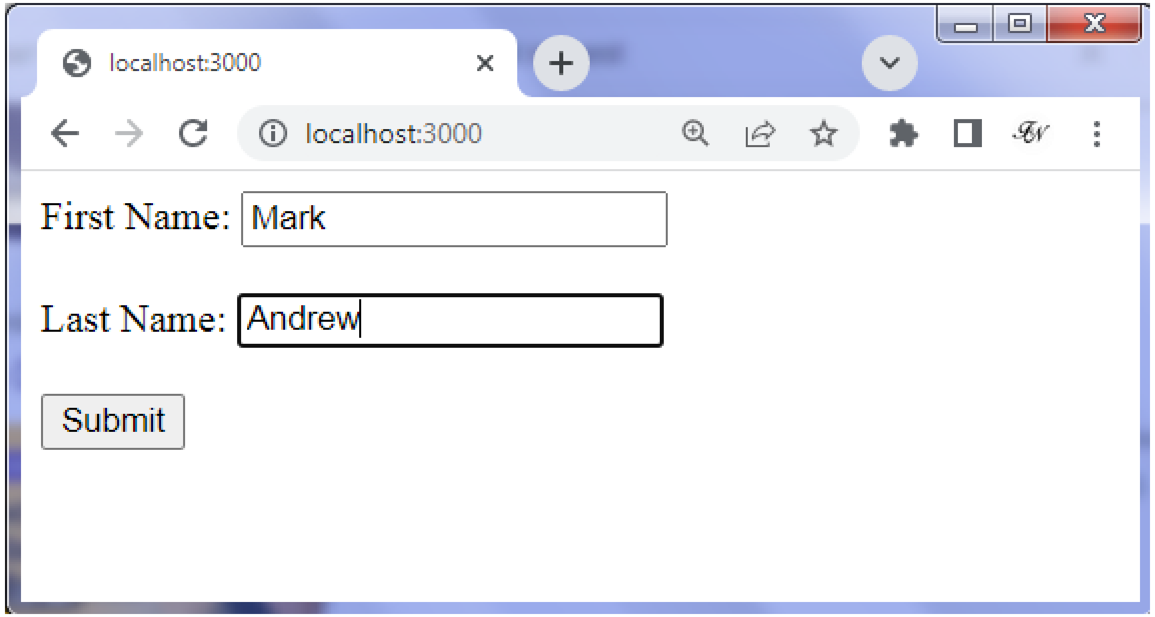
First Name:

Last Name:

Submit

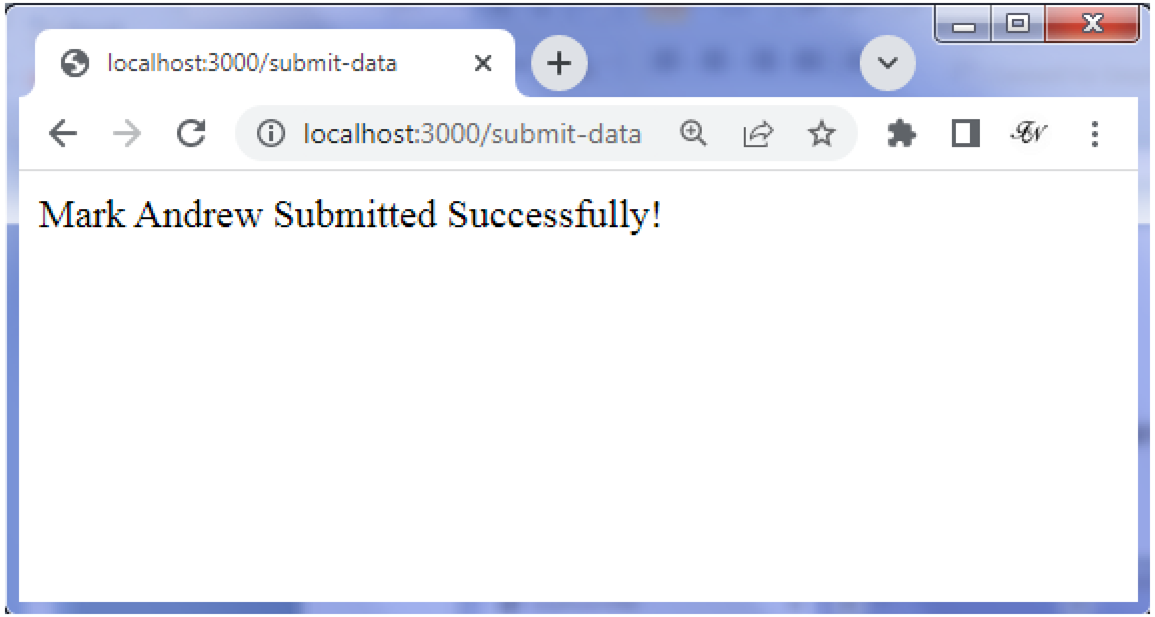
Handle POST Request

- Fill the First Name and Last Name in the above example and click on **submit**.
- For example, enter “Mark" in First Name textbox and “Andrew" in Last Name textbox and click the submit button. The following result is displayed.



First Name: Mark

Last Name: Andrew



Mark Andrew Submitted Successfully!

Routers

- Defining routes in the same file 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 **books.js** and type the following in it.

Filename: books.js

```
var express = require('express');
var router = express.Router();
router.get('/', function(req, res){
res.send('GET random book.');
});
router.post('/', function(req, res){
res.send(' Add a book.');
 });
//export this router to use in our index.js
module.exports = router;
```

Routers

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

Filename: index.js

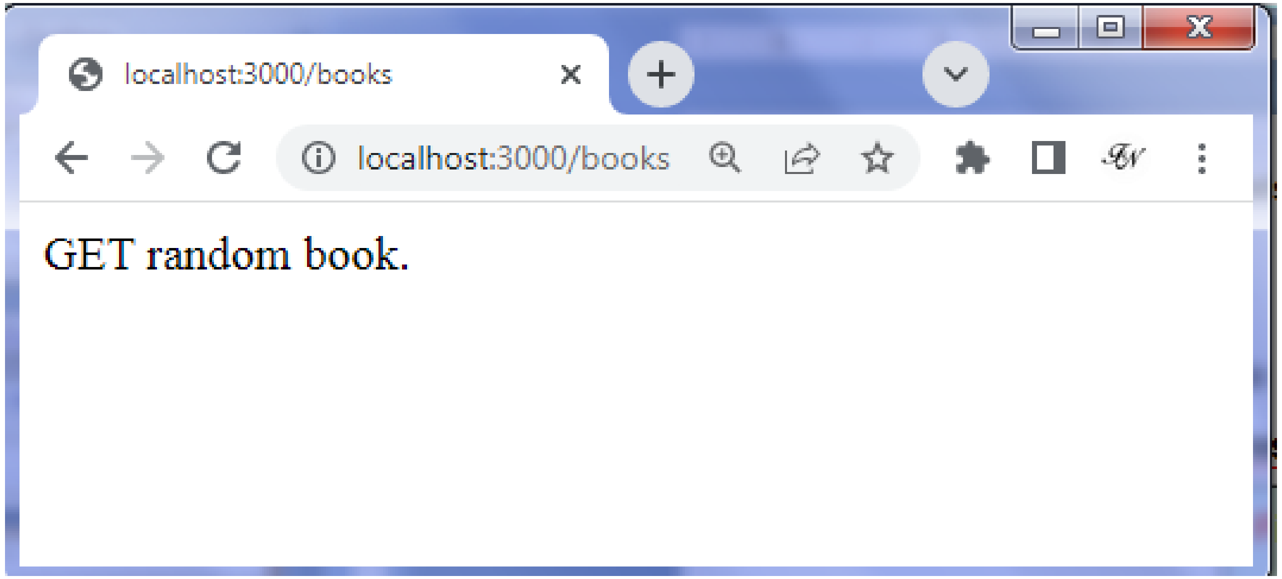
```
var express = require(‘Express');
var app = express();
var PORT = 3000;
var books = require('./books.js');
//both index.js and books.js should be in same directory
app.use('/books', books);
app.listen(PORT, function(err){
    if (err) console.log(err);
    console.log("Server listening on PORT", PORT);
});
```

In this example,

- The ***app.use*** function call on route **'/books'** attaches the **books** router with this route.
- Now whatever requests our app gets at the '/books', will be handled by our things.js router.
- The **'/'** route in books.js is actually a subroute of '/books'.

Routers

- Run the command **nodemon index.js** and point the browser to ***http://localhost:3000*** and see the following result.

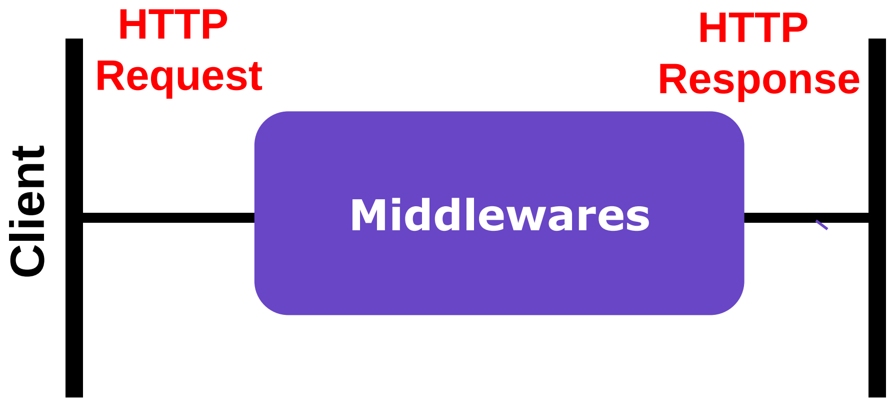


```
GET random book.
```

Express.js Middleware

Middleware

- Express is a routing and middleware web framework that has minimal functionality of its own.
- An Express application is essentially a series of middleware function calls.
- ***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.
- An Express-based application is a series of middleware function calls.



Middleware

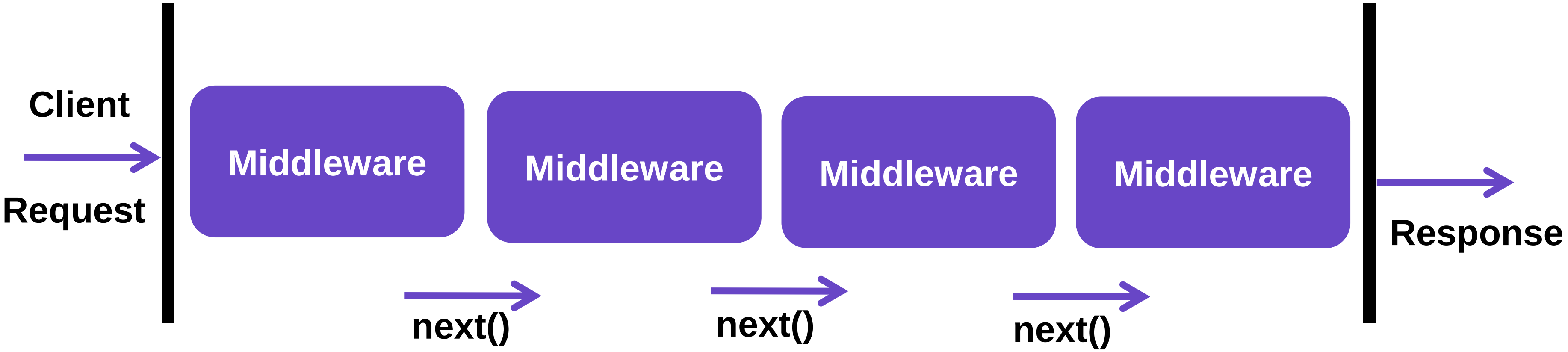
- Middleware functions can perform the following tasks:
  - Execute any code.
  - Make changes to the request and the response objects.
  - End the request-response cycle.
  - Call the next middleware function in the stack.
- The basic syntax for the middleware functions are as follows –

```
app.get(path, (req, res, next) => {})
```

- The middle part **(req,res,next)=>{}** is the middleware function.

Middleware Chaining

- **Middleware** can be chained from one to another, hence creating a chain of functions that are executed in order.
- The last function sends the response back to the browser.
- So, before sending the response back to the browser the different middleware process the request.
- The next middleware function is commonly denoted by a variable named **next**.
- If the current middleware function does not end the request-response cycle, it must call **next()** to pass control to the next middleware function. Otherwise, the request will be left hanging.



Middleware Types

An Express application can use the following types of middleware:

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

Note:

- We can load application-level and router-level middleware with an optional mount path.
- We can also load a series of middleware functions together, which creates a sub-stack of the middleware system at a mount point.

Application-level middleware

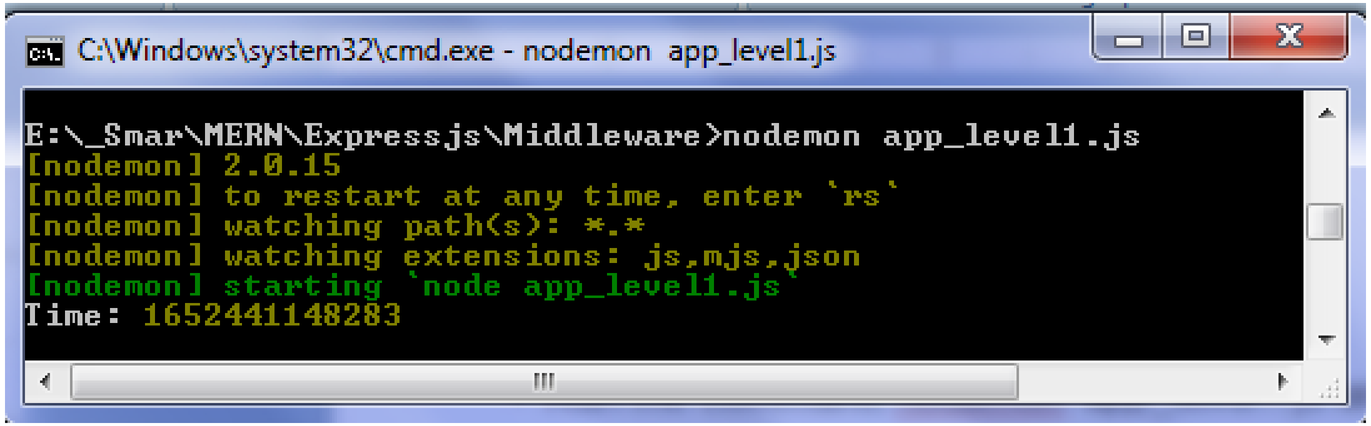
- Bind application-level middleware to an instance of the **app object** by using the **app.use()** and **app.METHOD()** functions,
  - METHOD is the HTTP method of the request that the middleware function handles (such as GET, PUT, or POST) in lowercase.
- The following example shows a middleware function with no mount path. The function is executed every time the app receives a request.

Filename: app_level1.js

```
const express = require('express')
const app = express()
app.use(function(req, res, next) {
        console.log('Time:', Date.now())
        next()
}); app.listen(3000);
```

Application-level middleware

- Run the command nodemon app_level1.js and point the browser to *http://localhost:3000* and see the following result in the console



```
E:\_Smar\MERN\Expressjs\Middleware>nodemon app_level1.js
[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node app_level1.js`
Time: 1652441148283
```

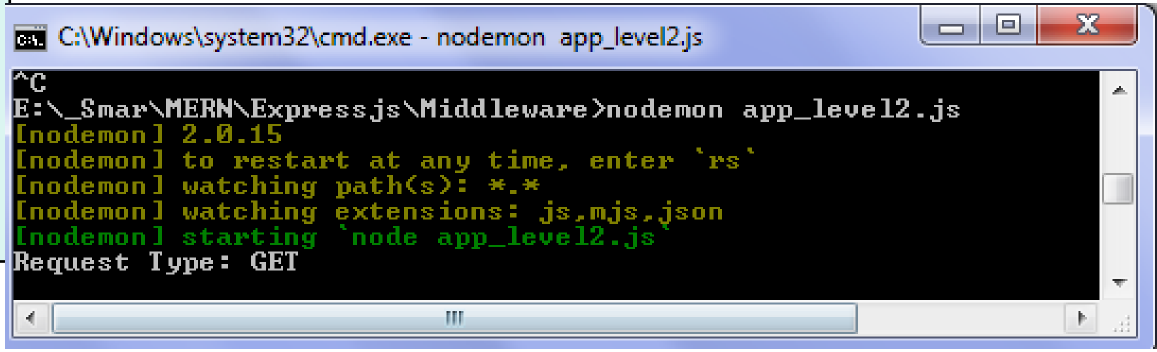
Application-level middleware

- The following example shows a middleware function mounted on the **/user/:id** path. The function is executed for any type of HTTP request on the **/user/:id** path.

Filename: app_level2.js

```
const express = require('express')
const app = express()
app.use('/user/:id',function (req, res, next) {
        console.log('Request Type:', req.method)
        next()
});
app.listen(3000);
```

Run the command **nodemon app_level2.js** and point the browser to *http://localhost:3000/user/:id* and see the following result in the console:



```
^C
E:\_Smar\MERN\Expressjs\Middleware>nodemon app_level2.js
[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node app_level2.js`
Request Type: GET
```

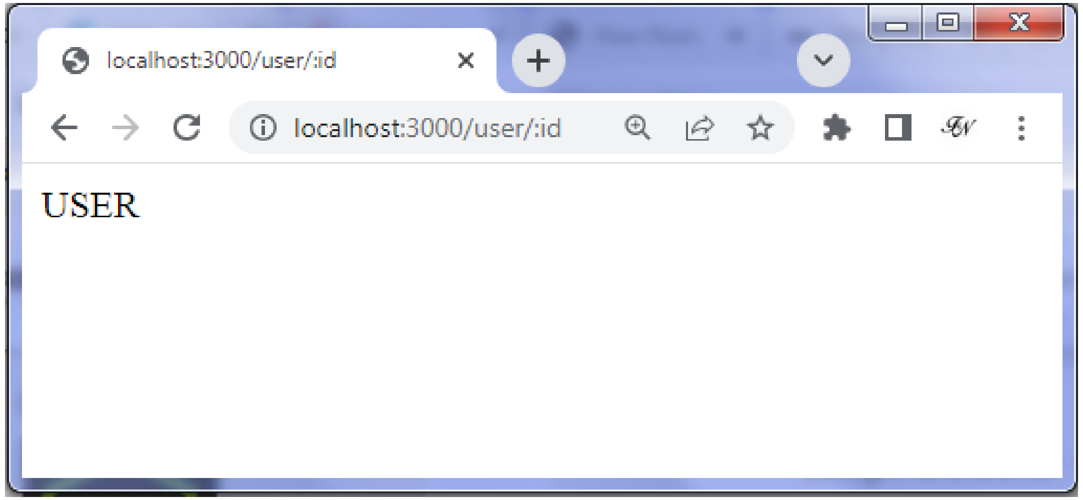
Application-level middleware

- The following example shows a route and its handler function (middleware system).
- The function handles GET requests to the **/user/:id** path.

Filename: app_level3.js

```
const express = require('express')
const app = express()
app.get('/user/:id',function (req, res, next) {
        res.send('User',)
        next()
})
app.listen(3000);
```

Run the command **nodemon app_level3.js** and point the browser to http://localhost:3000/user/:id and see the following result in the browser:



USER

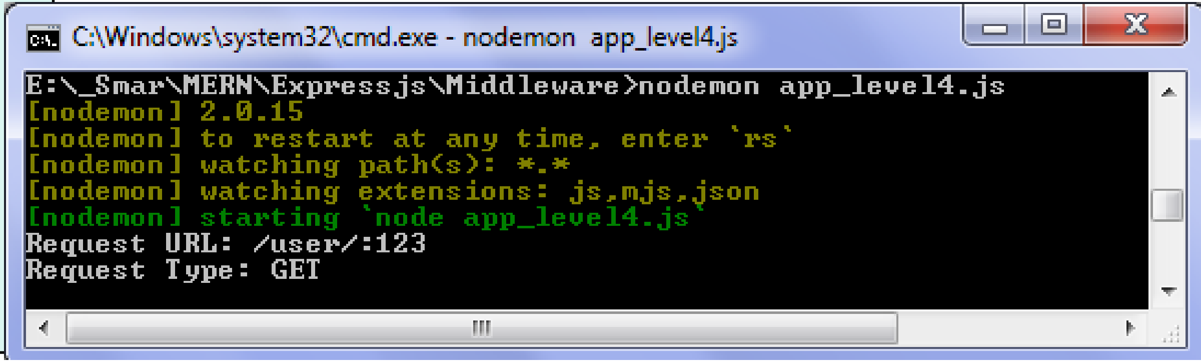
Application-level middleware

- Here is an example of loading a series of middleware functions at a mount point, with a mount path.
- It illustrates a middleware sub-stack that prints request info for any type of HTTP request to the **/user/:id** path.

Filename: app_level4.js

```
const express = require('express')
const app = express()
app.use('/user/:id', (req, res, next) => {
    console.log('Request URL:', req.originalUrl)
    next()
  },
  (req, res, next) => {
    console.log('Request Type:', req.method)
    next()
  })
app.listen(3000);
```

Run the command **nodemon app_level4.js** and point the browser to *http://localhost:3000/user/:123* and see the following result in the console:



C:\Windows\system32\cmd.exe - nodemon app_level4.js

```
E:\_Smar\MERN\Express.js\Middleware>nodemon app_level4.js
[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node app_level4.js`
Request URL: /user/:123
Request Type: GET
```

Application-level middleware

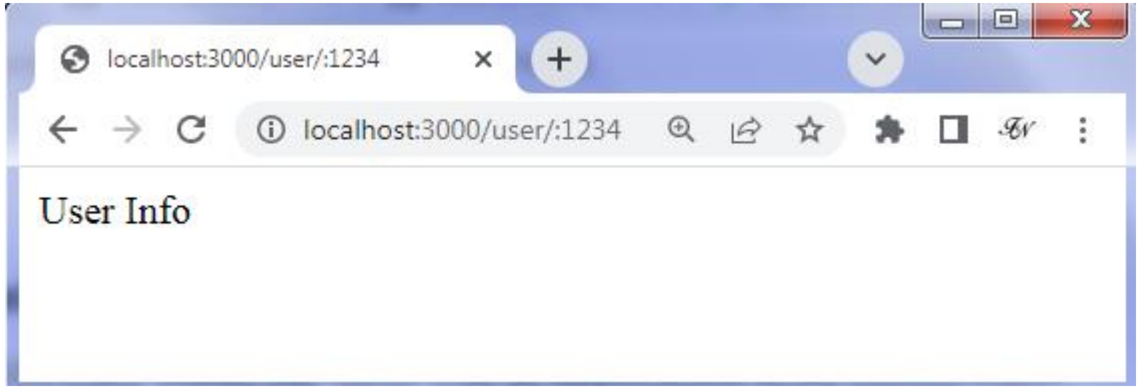
- Route handlers enable you to define multiple routes for a path.
- The example below defines two routes for GET requests to the /user/:id path.
- The second route will not cause any problems, but it will never get called because the first route ends the request-response cycle.
- The following example shows a middleware sub-stack that handles GET requests to the /user/:id path.

Filename: app_level5.js

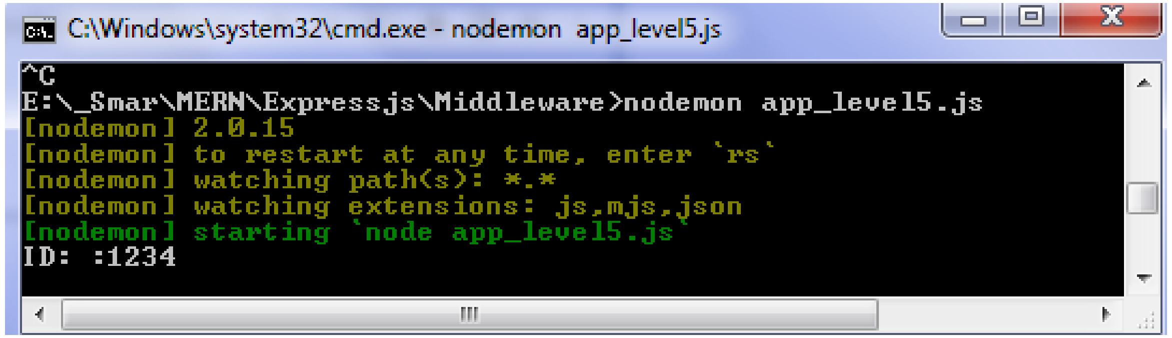
```
const express = require('express')
const app = express()
app.get('/user/:id', (req, res, next) => {
        console.log('ID:', req.params.id)
        next()
},
 (req, res, next) => {
      res.send('User Info')
})
// handler for the /user/:id path, which prints the user ID
app.get('/user/:id', (req, res, next) => {
        res.send(req.params.id)
})app.listen(3000);
```

Application-level middleware

- Run the command nodemon app_level5.js and point the browser to *http://localhost:3000/user:1234* and see the following result.



User Info



```
^C
E:\_Smar\MERN\Expressjs\Middleware>nodemon app_level5.js
[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node app_level5.js`
ID: :1234
```

Application-level middleware

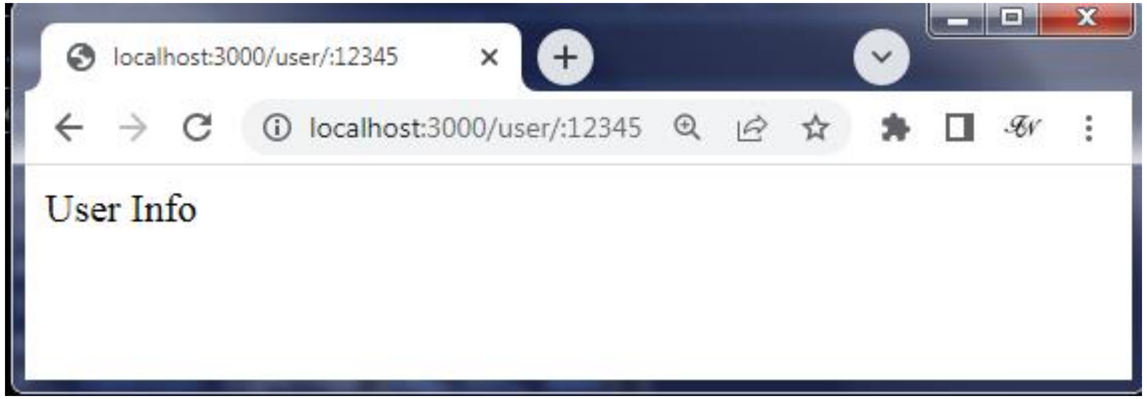
- Middleware can also be declared in an array for reusability.
- The following example shows an array with a middleware sub-stack that handles GET requests to the /user/:id path

Filename: app_level6.js

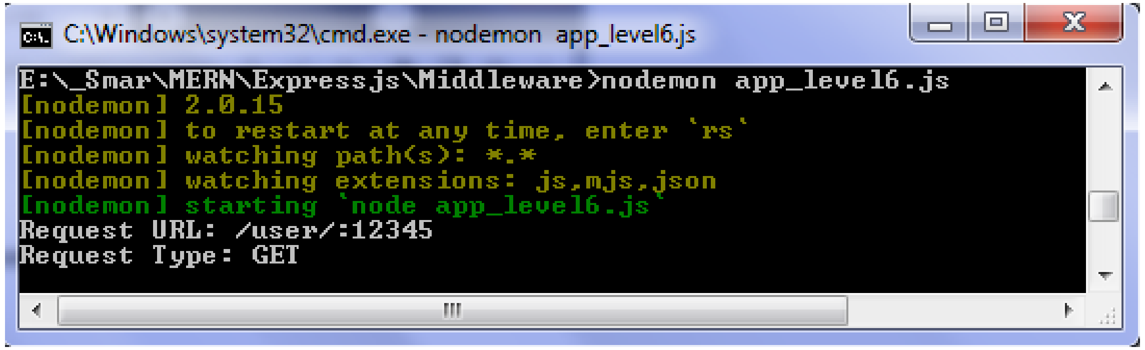
```
const express = require('express')
const app = express()
function logOriginalUrl (req, res, next) {
    console.log('Request URL:', req.originalUrl)
    next()
  }
function logMethod (req, res, next) {
    console.log('Request Type:', req.method)
    next()
  }
const logStuff = [logOriginalUrl, logMethod]
      app.get('/user/:id', logStuff, (req, res, next) =>
  {
    res.send('User Info')
  })
app.listen(3000);
```

Application-level middleware

- Run the command nodemon app_level6.js and point the browser to *http://localhost:3000/user:12345* and see the following result.



User Info



C:\Windows\system32\cmd.exe - nodemon app_level6.js

```
E:\_Smar\MERN\Expressjs\Middleware>nodemon app_level6.js
[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node app_level6.js`
Request URL: /user/:12345
Request Type: GET
```

Router-level middleware

- Router-level middleware works in the same way as application-level middleware, except it is bound to an instance of express.Router().

```
const router = express.Router()
```

- Load router-level middleware by using the router.use() and router.METHOD() functions.

Router-level middleware

- The following example code replicates the middleware system by using router-level middleware:

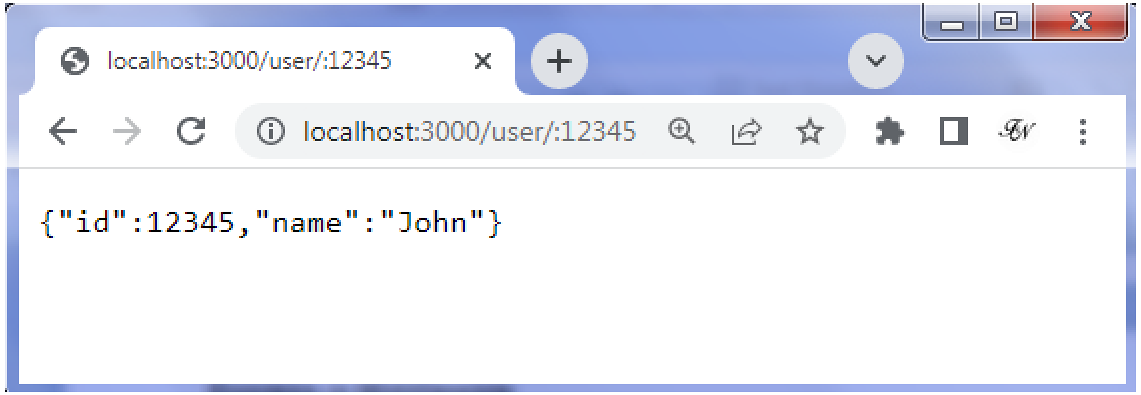
Filename:router_level.js

```
var express = require('express')
var app = express()
var router = express.Router()
// a middleware function with no mount path.
// This code is executed for every request to the router
router.use(function (req, res, next) {
  console.log('Router level middleware ...')
  next()
})
```

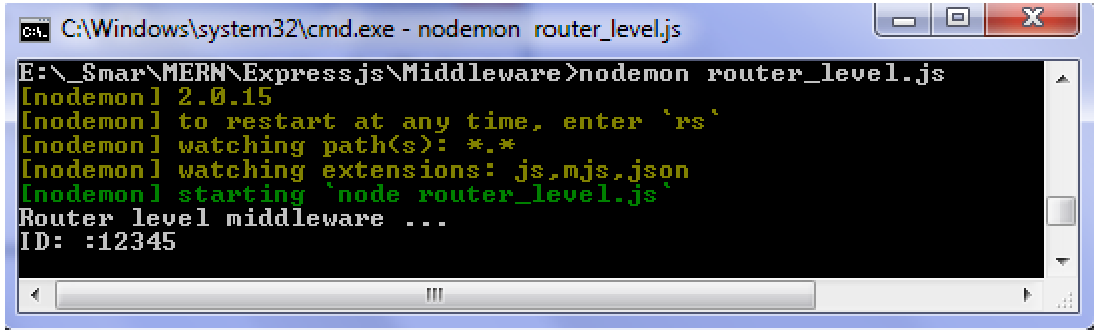
```
router.get('/user/:id', function (req, res, next) {
  console.log(req.params.id)
  res.send({id: 1, name: 'John'})
})
// mount the router on the app
app.use('/', router)

app.listen(3000);
```

Router-level middleware

- Run the command nodemon router_level.js and point the browser to *http://localhost:3000/user:12345* and see the following result.



```
{"id":12345,"name":"John"}
```



```
E:\_Smar\MERN\Express.js\Middleware>nodemon router_level.js
[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node router_level.js`
Router level middleware ...
ID: :12345
```

Router-level middleware

- We mostly use this kind of middleware for medium- to large-scale Express applications.
- When our allocation grows, we usually split our whole application code by features or resources (in REST API terms).
- If we have two resources in our application API, like **users** and **posts**, then we will create two router instances.

Filename: users.js

```
const express = require('express');
const router = express.Router();
router.get('/:id', (req, res) => {
        res.send(“ID: “+ req.params.id);
});
module.exports = router;
```

Router-level middleware

Filename: posts.js

```
const express = require('express');
const router = express.Router();
router.get('/', (req, res) => {
        res.send(“GET request”);
});
router.get('/:id', (req, res) => {
         res.send(“ID: “ + req.params.id );
});
module.exports = router;
```

Router-level middleware

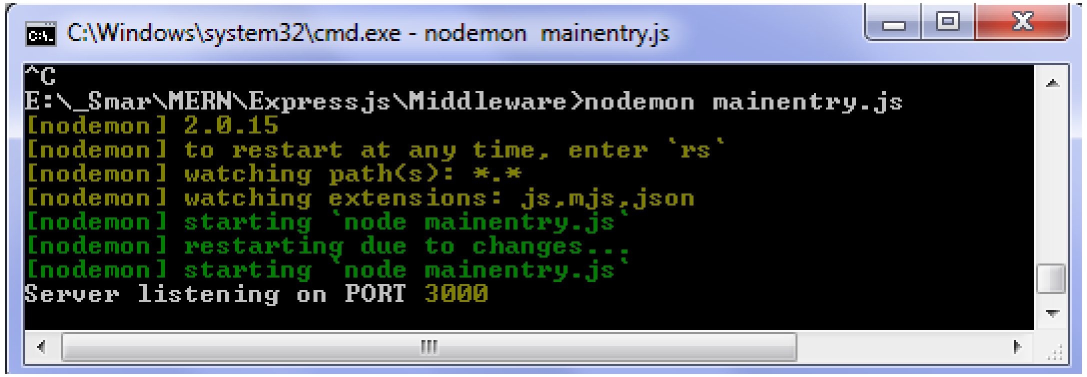
- Now in the main entry point file, we will tell Express which path we want to mount the users and posts router with:

Filename: mainentry.js

```
const express = require('express');
const posts = require('./routes/posts');
const users = require('./routes/users');
const app = express();
const PORT = 3000;
app.use('/api/posts', posts);
app.use('/api/users', users);
app.listen(PORT, function()
{
    console.log("Server listening on PORT", PORT);
});
```

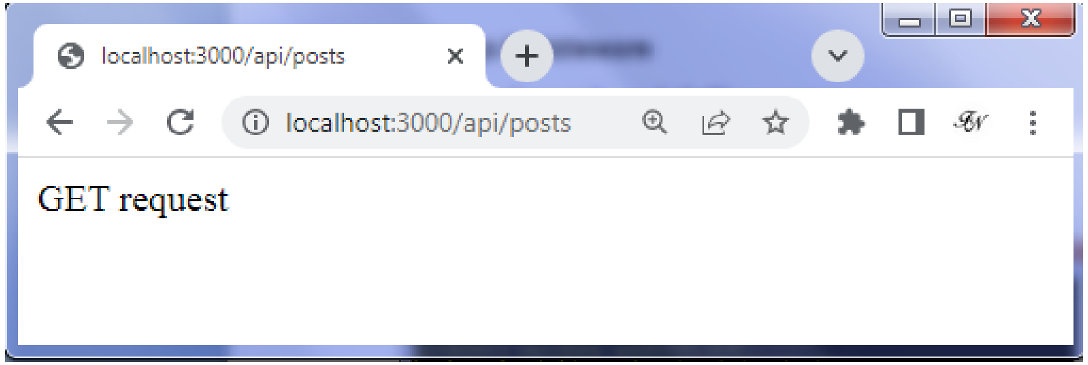
Router-level middleware

- Run the command nodemon mainentry.js



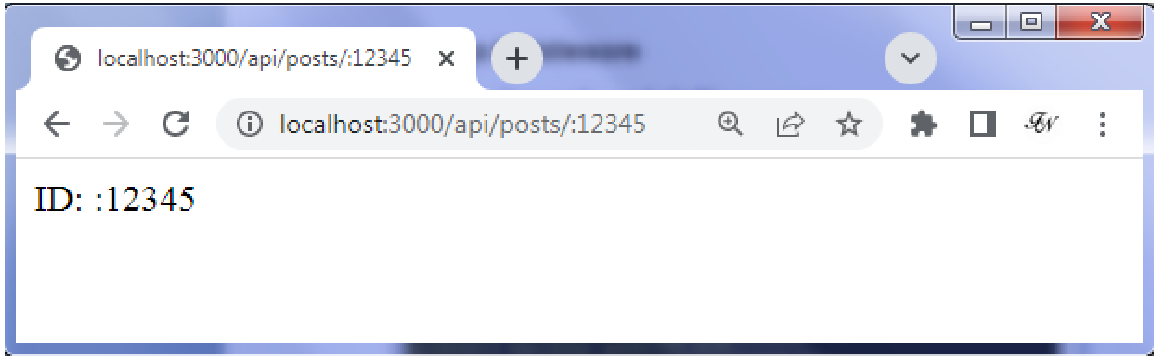
```
^C
E:\_Smar\MERN\Expressjs\Middleware>nodemon mainentry.js
[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node mainentry.js`
[nodemon] restarting due to changes...
[nodemon] starting `node mainentry.js`
Server listening on PORT 3000
```

Point the browser to *http://localhost:3000/api/posts*



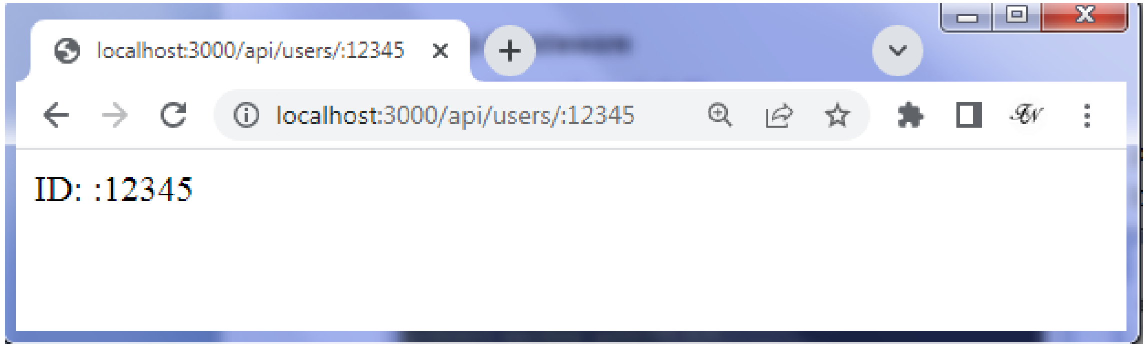
GET request

Point the browser to *http://localhost:3000/api/posts/:12345*


localhost:3000/api/posts/:12345
localhost:3000/api/posts/:12345

ID: :12345

Point the browser to *http://localhost:3000/api/users/:12345*


localhost:3000/api/users/:12345
localhost:3000/api/users/:12345

ID: :12345

Error handling middleware

- Error-handling middleware always takes ***four*** arguments.
- We must provide four arguments to identify it as an error-handling middleware function.
  - Even if we don’t need to use the next object, we must specify it to maintain the signature.
  - Otherwise, the next object will be interpreted as regular middleware and will fail to handle errors.
- Define error-handling middleware functions in the same way as other middleware functions, except with four arguments instead of three, specifically with the signature **(err, req, res, next))**

Error handling middleware

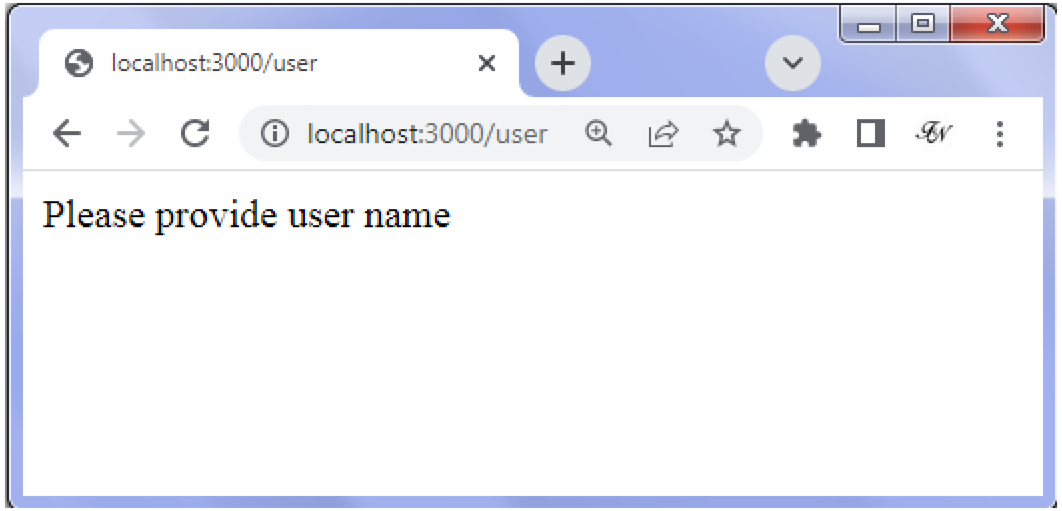
Filename:error_handling.js

```
const express = require('express');
const app = express();
// http://localhost:3000/user?name=kenny
app.get('/user', (req, res, next) => {
    if (!req.query.name) {
        const err = new Error("Please provide user name");
        return next(err);
    }
    console.log('User name: ', req.query.name);
    res.send('success');
});
```

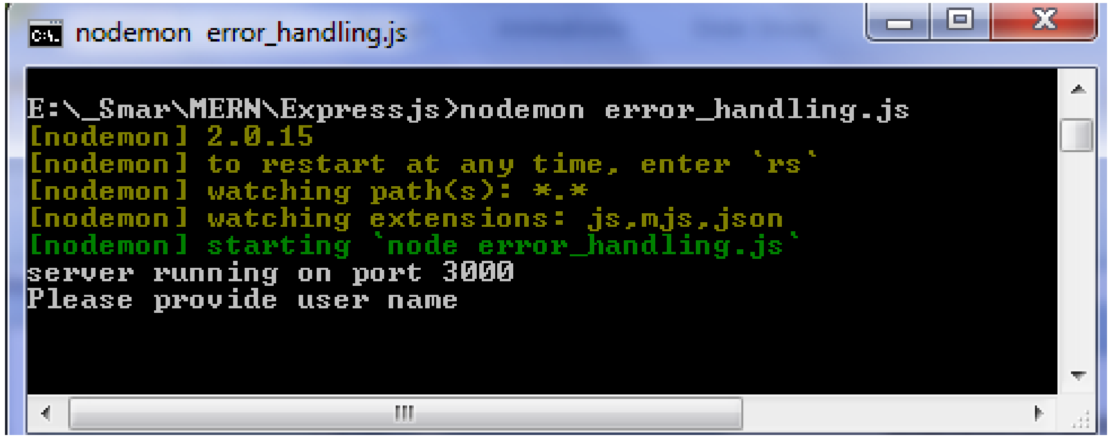
```
app.use((err, req, res, next) => {
    console.log(err.message);
    res.status(500);
    res.send(err.message)
});

app.listen(3000, (req, res) => {
    console.log(‘Server running on port 3000');
});
```

Error handling middleware

- Run the command nodemon error_handling.js and point the browser to *http://localhost:3000/user* and see the following result.



Please provide user name

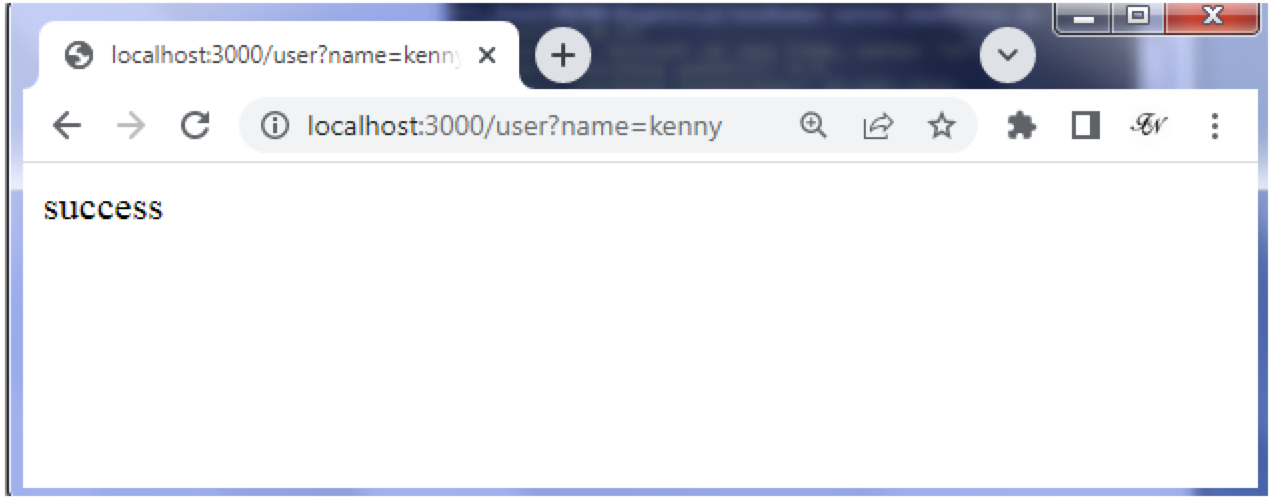


```
E:\_Smar\MERN\Expressjs>nodemon error_handling.js
[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node error_handling.js`
server running on port 3000
Please provide user name
```

Error in error handler

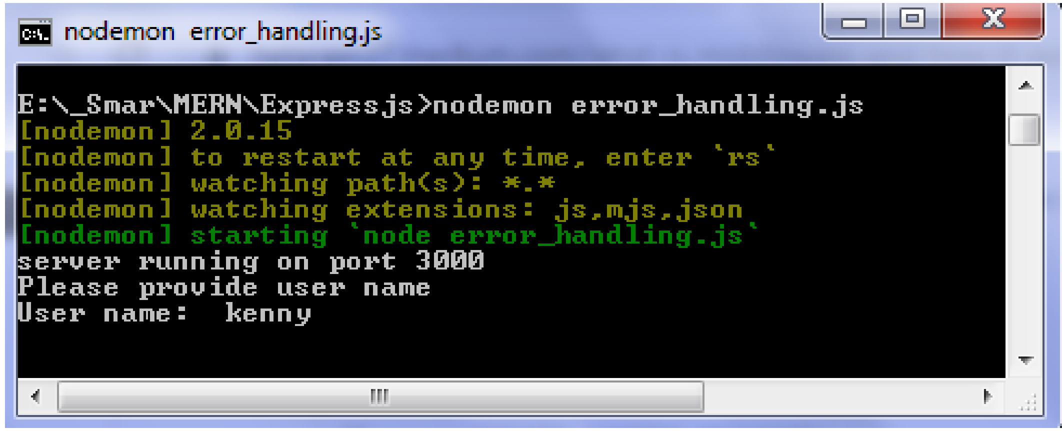
Error handling middleware

- Point the browser to *http://localhost:3000/user?name=kenny* and see the following result.



localhost:3000/user?name=kenny

```
success
```



```
E:\_Smar\MERN\Express.js>nodemon error_handling.js
[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node error_handling.js`
server running on port 3000
Please provide user name
User name:  kenny
```

- We should add our error handler middleware after all other routes, so if we get any error and we want to throw it using our error handler then with the next() it will get to the error handler and if we did not get any error then we will use res.send() so it will provide final response and close the request.

Built-in middleware

Express has the following built-in middleware functions:

- **express.static** serves static assets such as HTML files, images, and so on.
- **express.json** parses incoming requests with JSON payloads. NOTE: Available with Express 4.16.0+
- **express.urlencoded** parses incoming requests with URL-encoded payloads. NOTE: Available with Express 4.16.0+

Built-in middleware: express.static

- The express.static() function is a built-in middleware function in Express. It serves static files and is based on serve-static.

Syntax:

```
express.static(root, [options])
```

- Parameters: The root parameter describes the root directory from which to serve static assets.
- Return Value: It returns an Object.

Built-in middleware: express.static

Creating Express Server

- In index.js file, require in an Express instance and implement a GET request:

Filename: index.js

```
var express = require('express');
var app = express();
var path = require('path');
// Static Middleware
var public = path.join(__dirname, 'public');
app.get('/', function(req, res) {
    res.sendFile(path.join(public, 'app.html'));
});
//express provides a built-in method to serve static files
app.use('/', express.static(public));
app.listen(3000);
```

In this example,

- When we call app.use(), we are telling Express to use a piece of middleware.
- *Middleware* is a function that Express passes requests through before sending them to routing functions, such as app.get('/') route.
- express.static() finds and returns the static files requested.
- The argument we pass into express.static() is the name of the directory we want Express to serve files. Here, the public directory.

Built-in middleware: express.static

Structuring the Files

- To store our files on the client-side, create a public directory and include an app.html file along with an image. File structure will look like this:

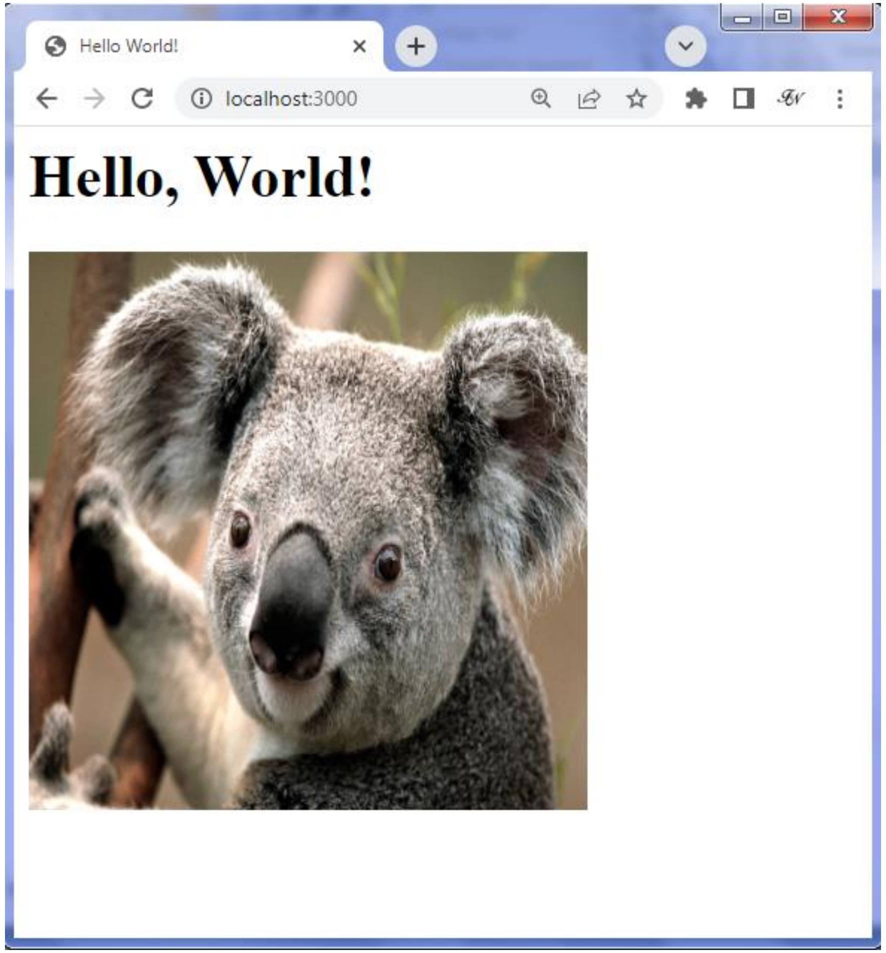
```
Middleware
 |- index.js
 |- public
     |-  app.html
     |-  Koala.jpg
```

Navigate to **app.html** file in the public directory and populate the file with body and image elements:

```
<html>
  <head>
    <title>Hello World!</title>
  </head>
  <body>
    <h1>Hello, World!</h1>
    <img src="Koala.jpg" alt="Koala" width="300" height="300">
  </body>
</html>
```

Built-in middleware: express.static

- Run the command nodemon index.js and point the browser to *http://localhost:3000* and see the following result.



Hello, World!

Built-in middleware: express.json

- The express.json() function is a built-in middleware function in Express.
- It parses incoming requests with JSON payloads and is based on body-parser.

Syntax:

```
express.json( [options] )
```

Parameters:

- The options parameter have various property like inflate, limit, type, etc.
- Return Value: It returns an Object.

Built-in middleware: express.json

What is Axios?

- Axios js is a promise-based HTTP library that lets you consume an API service. It offers different ways of making HTTP requests such as GET, POST, PUT, and DELETE.

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

- Interacting with REST APIs and performing CRUD operations are made simple with Axios. JavaScript libraries such as Vue, React, and Angular can use Axios.
- We need to import the Axios to display the output in Json format.

Built-in middleware: express.json

Filename: express1.js

```
const express = require('express');
const axios = require('axios');
const app = express();
app.use(express.json());
app.post('/', function(req,res) {
  // Without `express.json()`, `req.body` is undefined.
  console.log(req.body);
});
app.listen(4000);
// Prints "{ Name: ’Karthik’, Age: 36 }"
axios.post('http://localhost:4000', { Name: 'Karthik’, Age: 36 });
```

Built-in middleware: express.urlencoded

- The express.urlencoded() function is a built-in middleware function in Express.
- It parses incoming requests with urlencoded payloads and is based on body-parser.

Syntax:

```
express.urlencoded( [options] )
```

Parameter:

- The options parameter contains various property like extended, inflate, limit, verify etc.
- Return Value: It returns an Object.

Built-in middleware: express.json

Filename: index.html

```
<!DOCTYPE html>
<html >
 <head>
 <meta charset="utf-8" />
<title></title>
</head>
<body>
<form action="/submit-data" method="post">
 First Name: <input name="firstName" type="text" /> <br />
Last Name: <input name="lastName" type="text" /> <br />
<input type="submit" /> </form> </body> </html>
```

Built-in middleware: express.json

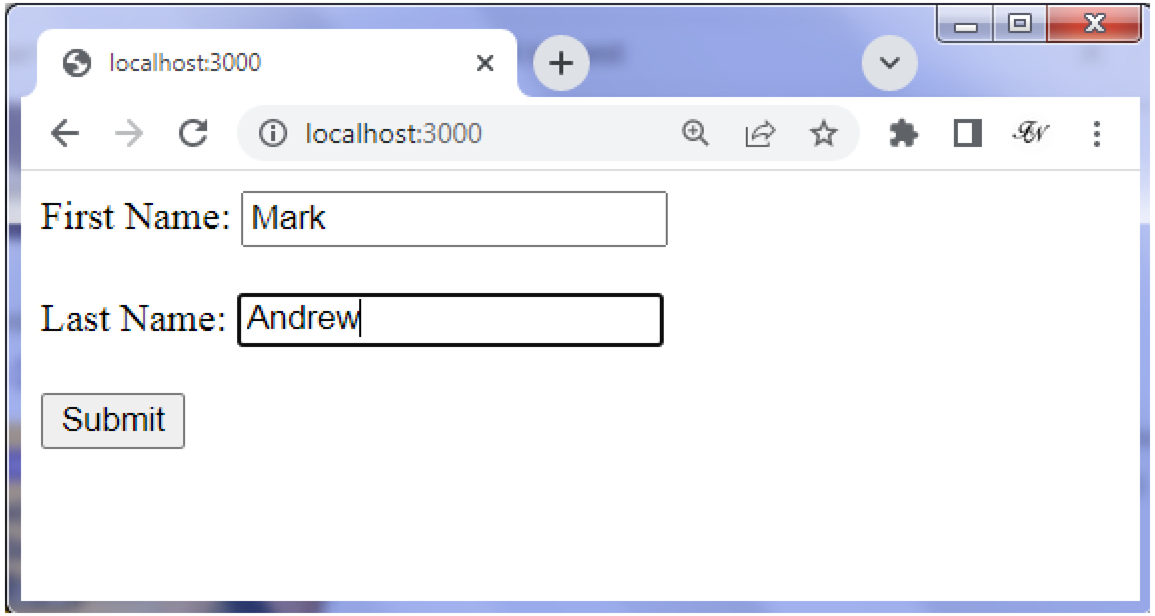
Filename: index.js

```
const express = require('express');
const path = require('path')
const app = express();
var PORT = 3000;
app.use(express.urlencoded({ extended: true }));
app.get('/', function (req, res,next) {
res.sendFile('D:/All_Materials/Express/Middleware/ index.html');
});
```

```
app.post('/submit-data', function (req, res) {
  var name = req.body.firstName + ' ' + req.body.lastName;
  res.send(name + ' Submitted Successfully!');
  console.log(req.body)
});

app.listen(PORT, function(err){
  if (err) console.log(err);
  console.log("Server listening on PORT", PORT);
});
```

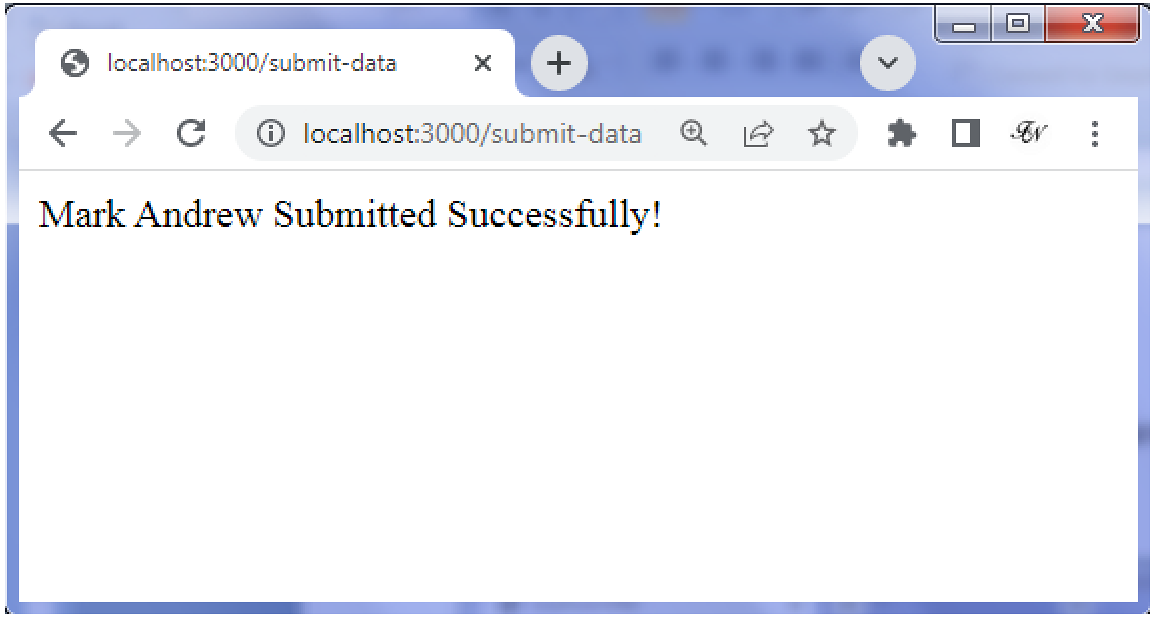
Built-in middleware: express.json

- Fill the First Name and Last Name in the above example and click on submit.
- For example, enter “Mark" in First Name textbox and “Andrew" in Last Name textbox and click the submit button. The following result is displayed.



First Name: Mark

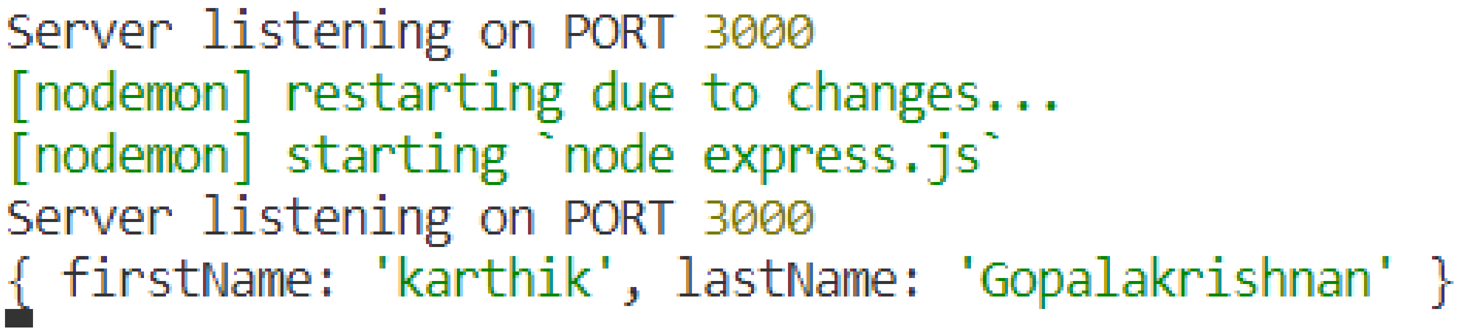
Last Name: Andrew



Mark Andrew Submitted Successfully!

Built-in middleware: express.json

- In terminal the output will displayed in the format of Json.



```
Server listening on PORT 3000
[nodemon] restarting due to changes...
[nodemon] starting `node express.js`
Server listening on PORT 3000
{ firstName: 'karthik', lastName: 'Gopalakrishnan' }
```

Third-party middleware

- There are a number of third party middleware, such as body-parser, mongoose, morgan and so on.
- They are like plugins that can be installed using npm and this is why Express is flexible.

Installation

```
npm install <module name>
```

- Third-party middlewares are a good alternative when there is a package available out there offering functionality that our API will need such as allowing client access from different origins and parsing the incoming request body data.

Third-party middleware: bodyParser

- It allows developers to process incoming data, such as body payload. The payload is just the data we are receiving from the client to be processed on. Most useful with POST methods.
- It is installed using:

```
npm install --save body-parser
```

- Usage:

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

Third-party middleware: cookie-parser

- 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
- It parses Cookie header and populate req.cookies with an object keyed by cookie names. To install it,

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

Third-party middleware: cookie-parser

- Usage:

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

const app = express()

const cookieParser = require('cookie-parser')

 // load the cookie-parsing middleware

app.use(cookieParser())
```

Third-party middleware: express-session

- HTTP is stateless;
- In order to associate a request to any other request, we need a way to store user data between HTTP requests.
- Sessions solve exactly this problem. We will need the Express-session, so install it using the following code.

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

Third-party middleware: express-session

- Usage: express-session.js

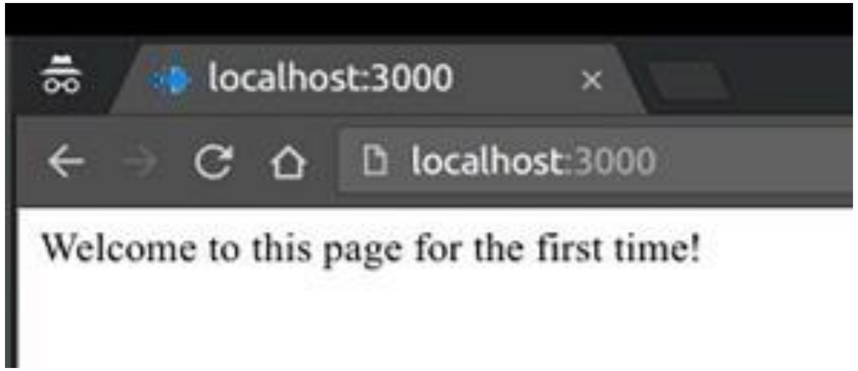
```
var express = require('express');
var cookieParser = require('cookie-parser');
var session = require('express-session');
var app = express();
app.use(cookieParser());
app.use(session({secret: "Shh, its a secret!"}));
app.get('/', function(req, res){
  if(req.session.page_views){
    req.session.page_views++;
    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);
```

In this example,

When a user visits the site, it creates a new session for the user and assigns them a cookie. Next time the user comes, the cookie is checked and the page_view session variable is updated accordingly.

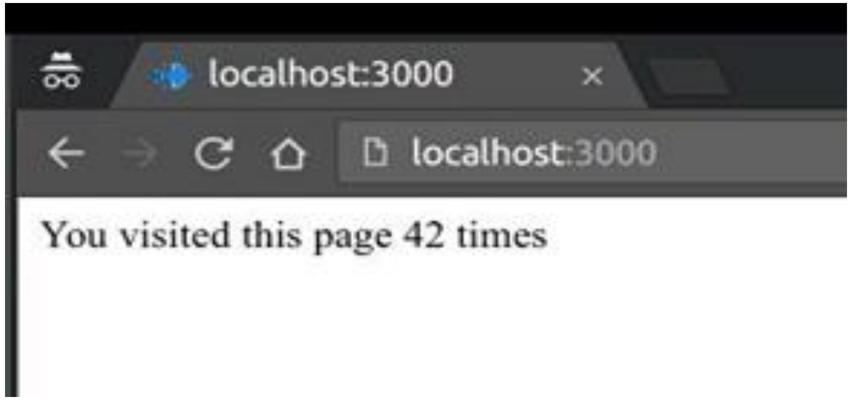
Third-party middleware: express-session

- Run the command nodemon express-session.js and Point the browser to http://localhost:3000 and see the following result.



Welcome to this page for the first time!

- If we revisit the page, the page counter will increase. The page in the following screenshot was refreshed 42 times.



You visited this page 42 times

Express.js Templating

Template Engine

- A template engine facilitates to use static template files in the applications.
- At runtime, it replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client.
- So this approach is preferred to design HTML pages easily.
- There are number of template engines available some of them are given below:-
  - EJS
  - Pug (formerly known as jade)
  - Vash
  - Mustache
  - Dust.js
  - Nunjucks
  - Handlebars

Using template engines with Express

- Template engine makes us able to use static template files in our application.
- To render template files we have to set the following application setting properties:
- **Views:** It specifies a directory where the template files are located.

  For example: app.set('views', './views').
- **view engine:** It specifies the template engine that you use.

  For example, to use the Pug template engine: app.set('view engine', 'pug')

EJS template engine

- EJS is a simple templating language that lets you generate HTML markup with plain JavaScript.
- EJS is much more similar to HTML than Pug is, retaining the usual method of opening and closing tags as well as specifying attributes.
- Variables are declared using angle brackets and the percent sign in this manner: <%= name %>.
- EJS tags can be used in different ways:
  - <%= – Escape the provided value, and output it to the template
  - <%- – Output the provided value without escaping. It is advised you escape all HTML variables before rendering to prevent cross-site scripting (XSS) attacks
  - <% – Perform control operations such as using a conditional or loop

EJS template engine

Integrating EJS into Express

- First, install the ejs package from npm:

```
npm i ejs
```

- Create a folder named “Views” in project’s root directory Because by default Express.js searches all the views (templates) in the views folder under the root folder.
- If we don’t want your folder to be named as views we can change name using views property in express.
- Eg : the name of your folder is myviews then in our main file in this case app.js we write app.set(‘views’,./myviews);
- All the templates file are saved with template engine’s extension and in views folder. Here we are using EJS so all templates are saved with .ejs extension in views folder.

EJS template engine

- EJS files are simple to write because in this file we write simple HTML and write logic, backend variables between these tags:-

  <% %>(to write logics)

  <%= %>(when we want to display content use this)

  <%- %>

EJS template engine

- For demonstration purpose, we have two files one is main JS file i.e. app.js , which creates a server and one template file(display.ejs).

Filename:app.js

```
// To create a basic server like
var express = require("express"); //requiring express module
var app = express(); //creating express instance

//To send templates to frontend, first we set the view engine as follows:
app.set("view engine","ejs");

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

app.listen(3000, function() {
          console.log("Node server is running..");
});
```

EJS template engine

- After setting the view engine we create a simple display.ejs file in view folder with a simple ‘hello user’ message in it.

Filename:display.ejs

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-
     scale=1.0">
   <meta http-equiv="X-UA-Compatible" content="ie=edge">
   <title>templates</title>
</head>
<body>
   <p>Hello User</p>
</body>
</html>
```

EJS template engine

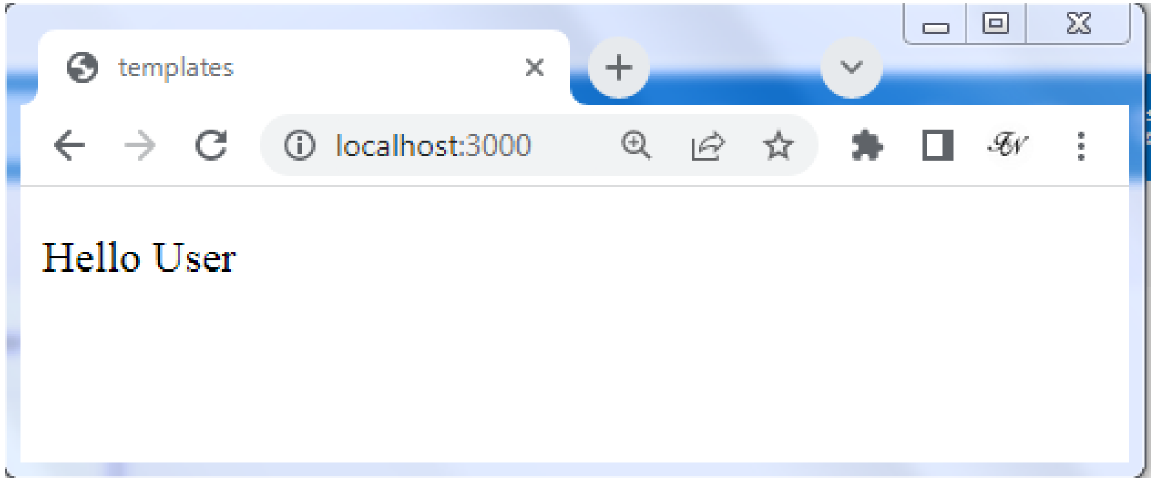
- Now to send the file to frontend we have a function called i.e. res.render() , it renders a view and sends the rendered HTML string to the client.
- To render our file in previous code we replace res.send(“hello”); with res.render(‘display’) ;

Filename:app.js

```
// To create a basic server like
var express = require("express"); //requiring express module
var app = express(); //creating express instance

//To send templates to frontend, first we set the view engine as follows:
app.set("view engine","ejs");

app.get("/", function(req, res) {
  res.render('display');
});
app.listen(3000, function() {
          console.log("Node server is running..");
});
```

EJS template engine

- Run the following code as “nodemon app.js” in terminal and point to browser to http://localhost:3000 and see the message “Hello User” will be printed



Hello User

EJS template engine

Sending variables to EJS and show on frontend

- If we want to send data from backend and use it in EJS file we can do this in the following manner:-
- Here instead of user in our display.ejs file I want to specify the name of the user. We can send data where we are rendering the file.
- Eg:-

```
res.render(‘display’,{variable_name:’value’});
```

- Say the username is ‘Alice’ then our code will be:-

```
res.render('display',{user:'Alice'});
```

- And to display this on front end we write this user variable inside <%= %> like this:-

```
<p>Hello <%= user%></p>
```

- Now restart the server and run the code again and on refreshing the page you will see Hello Alice instead of Hello User.

EJS template engine

Filename:display.ejs

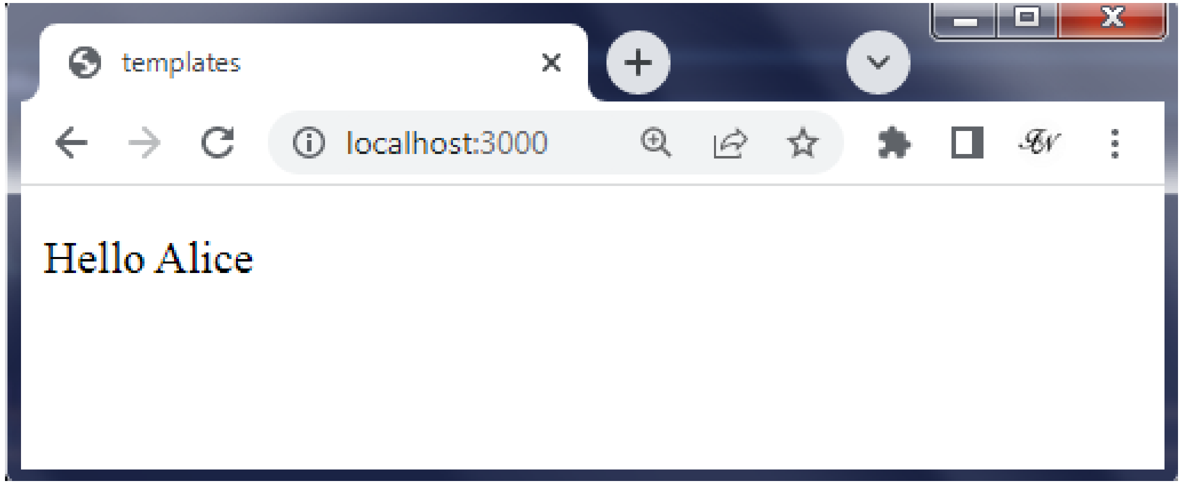
```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport"
     content="width=device-width, initial-
     scale=1.0">
   <meta http-equiv="X-UA-Compatible"
     content="ie=edge">
   <title>templates</title>
</head>
<body>
   <p>Hello <%= user%></p>
</body>
</html>
```

Filename:app.js

```
// To create a basic server like
var express = require("express"); //requiring
     express module
var app = express(); //creating express instance

//To send templates to frontend, first we set the
     view engine as follows:
app.set("view engine","ejs");

app.get("/", function(req, res) {
  res.render('display',{user:'Alice'});
});
app.listen(3000, function() {
          console.log("Node server is running..");
});
```

EJS template engine

- Run the command as “nodemon app.js” in terminal and point to browser to http://localhost:3000 and see the message “Hello User” will be printed



Hello Alice

EJS template engine

Sending Arrays or Objects(JSON data)

- We can send any sort of data to the frontend. Suppose if we have a JSON data having an array named authors[] of author object having name and article, and I want to display it on frontend via a list, I can send an array to EJS and then loop through it and display the data. For this follow the given code in app.js:-

Filename:app_authors.js

```
// example for Sending Arrays or Objects(JSON data)
var express = require("express");
var app = express(); /
app.set("view engine","ejs");
var authors = [
    {
      name: 'Andrew',
      article: 'node.js',
    },
    {
      name: 'Brad',
      article: 'express.js',
    },
    {
      name: 'John',
      article: 'mongoose',
    },
  ]
  app.get('/authors', function (req, res) {
    res.render('displayAuthors', { authors })
  })
  app.listen(3000);
```

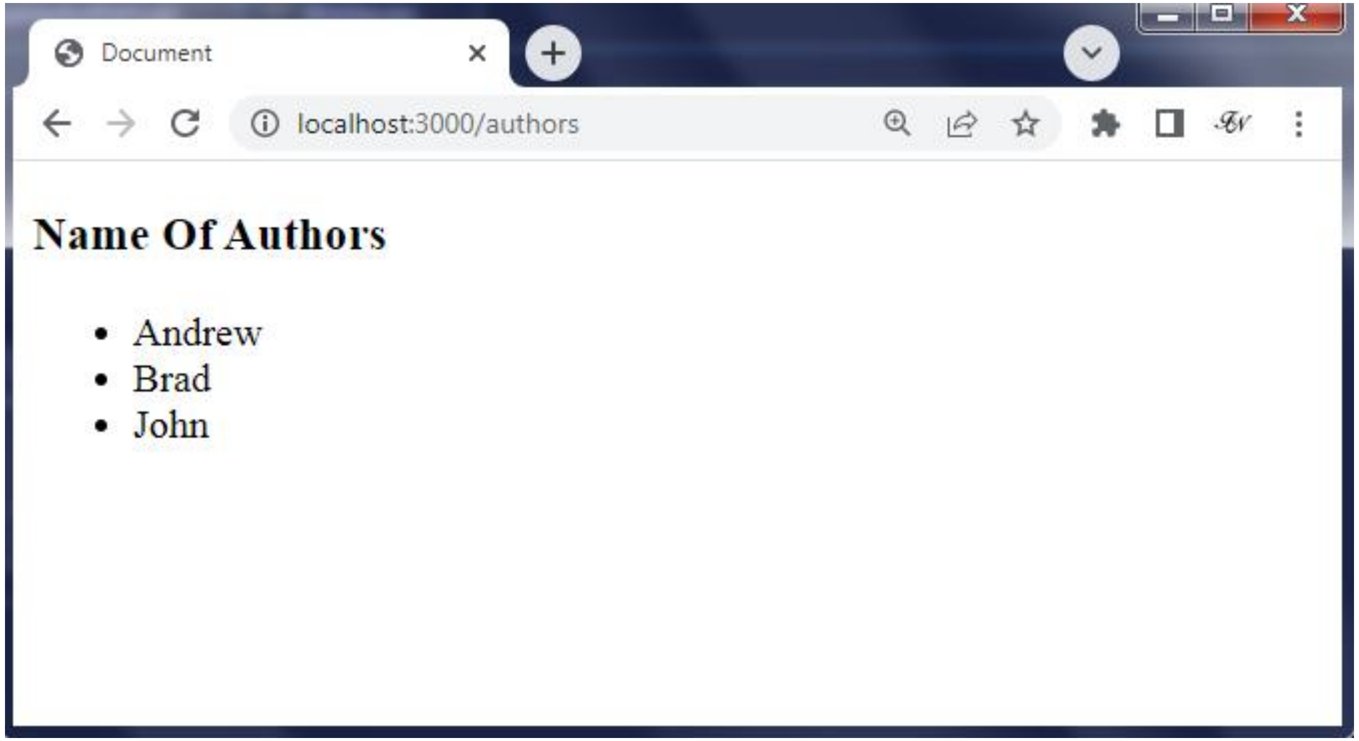
EJS template engine

- Now to access the author array in EJS follow the given EJS code:-

```
Filename:display_author.ejs
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
  <h3>Name Of Authors</h3>
  <ul>
    <% authors.forEach(function(author){ %>      <!-- looping through authors array  -->
      <li>
        <%= author.name %>
      </li>
    <% }) %> </ul></body></html>
```

EJS template engine

- Here In the above EJS code, we have used a forEach loop to iterate through “authors” array and to access data of authors array one by one.
- Save app.js and display.ejs with the following code and then run the code, now as we refresh the web page(http://localhost:3000/authors) you will see the following output.



Name Of Authors

- Andrew
- Brad
- John

EJS template engine

Using partials in EJS

- Suppose we have some code snippets, such as navigation bar snippet, needed to be added in multiple pages, we can do it by creating partials (partial code snippets).
- For an instance, suppose we want to create a common navigation bar for all the pages of an application without repeating code on each pages. For this we need to create a folder named partials which will contain all the partial files.

EJS template engine

- For demonstration purpose, let’s create an index.ejs which contains our page content.

Filename:index.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width"
/>
    <link
      rel="stylesheet"
      href=
"https://stackpath.bootstrapcdn.com/bootstrap/4.4.1
/css/bootstrap.min.css"
    />
    <title><%= subject %></title>
  </head>
  <body>

     <%- include('../partials/_nav'); %> // partial included here
    <div class="container mt-2">
      <header>
        <h2>Welcome</h2>
        <p>Here is the homepage for <%= name %></p>
      </header>
```

```
<section>
        <h2>Here is the body</h2>
        <p>
                Lorem ipsum dolor sit, amet
consectetur adipisicing elit. Totam,
                repellendus!
        </p>
      </section>

      <footer>
        <h2>Here is the footer</h2>
        <p>
<a href="<%= link %>">
Unsubscribe
</a>
</p>
      </footer>
    </div>
  </body>
</html>
```

EJS template engine

- Inside the partials subfolder, create a nav.ejs file and add this code:

Filename:nav.ejs

```
<nav class="navbar navbar-dark bg-primary navbar-expand">
  <div class="container"><a class="navbar-brand" href="#">TMP</a>
    <ul class="navbar-nav mr-auto">
      <li class="nav-item"><a class="nav-link" href="#"><span>Home</span></a></li>
      <li class="nav-item"><a class="nav-link" href="#"><span>About</span></a></li>
      <li class="nav-item"><a class="nav-link" href="#"><span>Menu</span></a></li>
      <li class="nav-item"><a class="nav-link" href="#"><span>Contact</span></a></li>
    </ul><span class="navbar-text"><a class="nav-link" href="#"><span>
     Login</span></a></span>
  </div>
</nav>
```

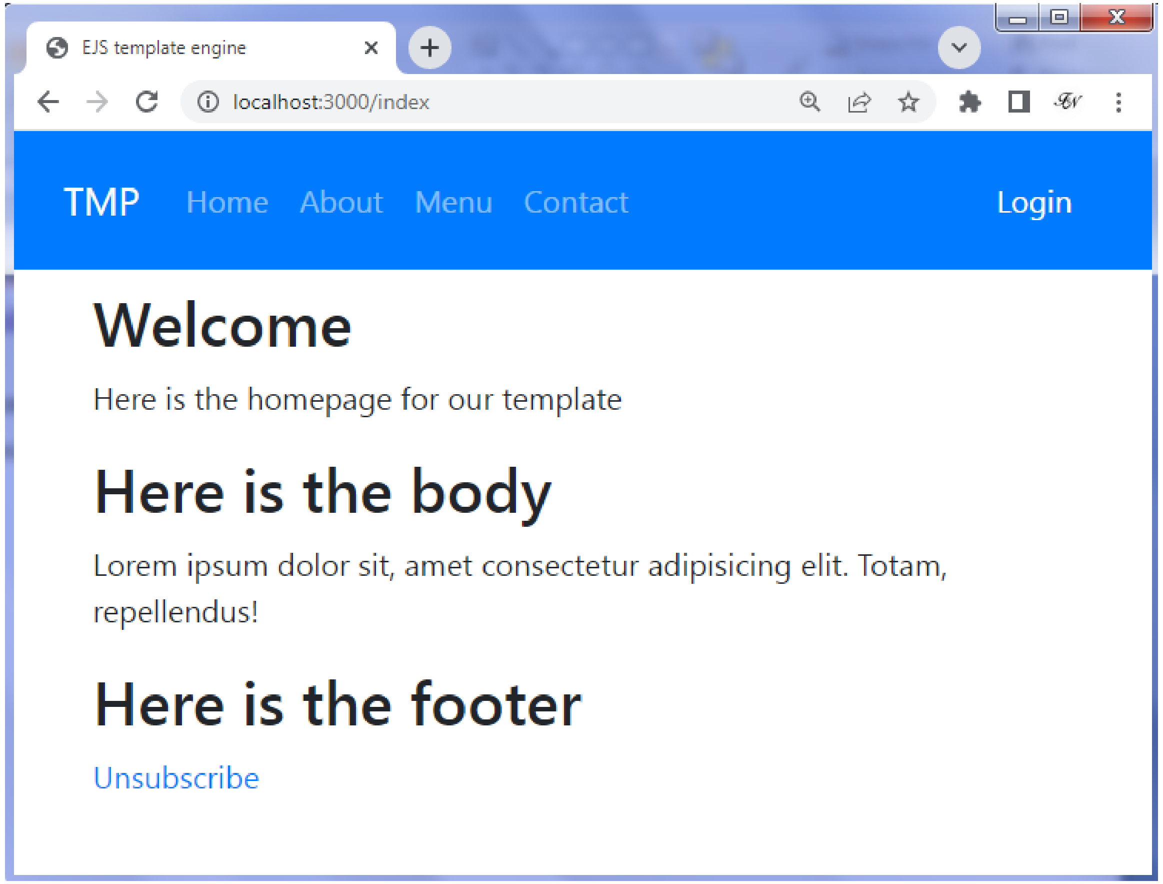
EJS template engine

- Add the following route to the app.js file:

```
Filename:app_nav.js
var express = require("express");
var app = express(); /
app.set("view engine","ejs");
app.get('/index', function (req, res){
    res.render('index', {
      subject: 'EJS template engine',
      name: 'our template',
      link: 'https://google.com'
    });
  });
app.listen(3000, function() {
    console.log("Node server is running..");
});
```

EJS template engine

- Run the command as “nodemon app_nav.js” in terminal and point to browser to http://localhost:3000 and see the result



Welcome

Here is the homepage for our template

Here is the body

Lorem ipsum dolor sit, amet consectetur adipisicing elit. Totam, repellendus!

Here is the footer

Unsubscribe

EJS template engine

Serving static files in Express

- To serve static files such as images, CSS files, and JavaScript files, use the express.static built-in middleware function in Express.
- The function signature is:

```
express.static(root, [options])
```

- The root argument specifies the root directory from which to serve static assets. For more information on the options argument, see express.static.
- For example, use the following code to serve images, CSS files, and JavaScript files in a directory named public:

```
app.use(express.static('public'))
```

EJS template engine

- For static files, we create a folder with name such as ‘public’. When we set our static files folder as ‘public’ using express.static middleware, all the static files will be served from this public folder itself.
- Suppose, we create a route to display a frontend page, with an image, as shown below:

Filename:app_image.js

```
var express = require("express");
var app = express();
app.set("view engine","ejs");
app.use(express.static('public'))
app.get('/image', (req, res) => {
    res.render('displayImage')
  })
app.listen(3000);
```

EJS template engine

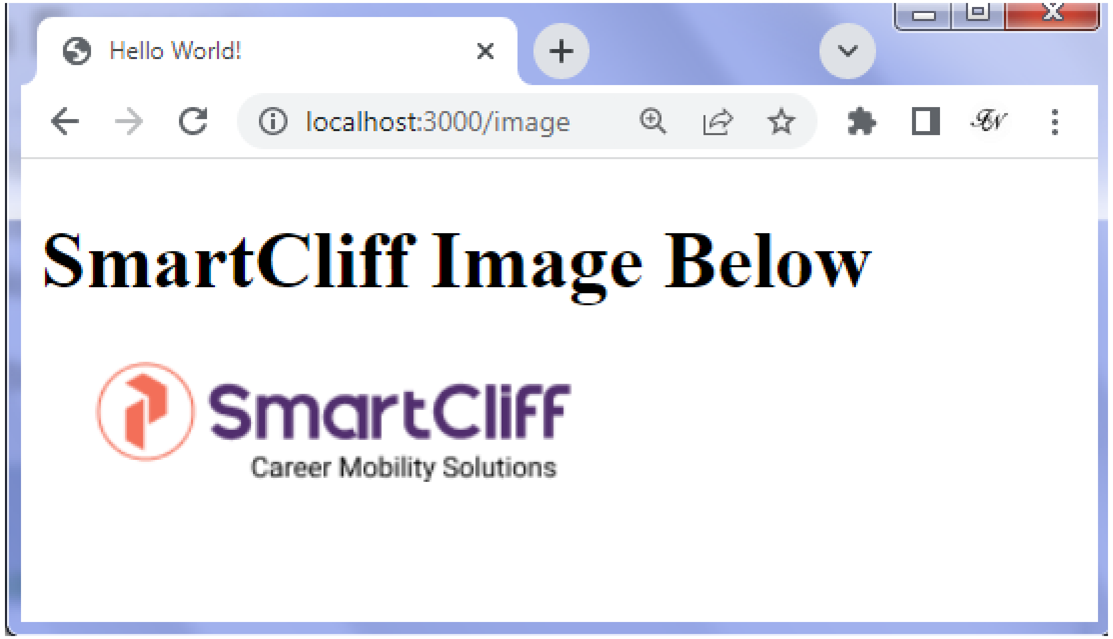
- For demonstration purpose, let’s take a jpg image file named ‘smartcliff.jpg’ in our public folder. And a displayImage.ejs file in our view folder which has the following code in it :

```
<html>
  <head>
    <title>Hello World!</title>
  </head>
  <body>
    <h1>SmartCliff Image Below</h1>
    <img src=“logo.png" alt=“LOGO" />
  </body>
</html>
```

- Notice the image element source to montbleu.jpg . Since you’ve served the public directory through Express, you can add the file name as your image source’s value.

EJS template engine

- Run the command as “nodemon app_image.js” in terminal and point to browser to http://localhost:3000 /image and see the result



SmartCliff Image Below

EJS template engine

Multiple Static Directories

- We can also set multiple static assets directories using the following program −

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

app.use(express.static('public'));
app.use(express.static('images'));

app.listen(3000);
```

EJS template engine

Virtual Path Prefix

- We can also provide a path prefix for serving static files. For example, if you want to provide a path prefix like '/static', you need to include the following code in our .js file −

```
var express = require('express');
var app = express();
app.use('/static', express.static('public'));
app.listen(3000);
```

- Now whenever we need to include a file, for example, a script file called .ejs residing in our public directory, use the following script tag −

```
<script src = "/static/main.js" />
```

- This technique can come in handy when providing multiple directories as static files. These prefixes can help distinguish between multiple directories.

Form Handling in Express

- In express, to handle form submission, we use ‘body-parser’ to parse the body received in our route request of form submission.
- A simple demonstration of form handling using a simple form page rendered through GET request on route ‘/formPage’ and post request on route ‘/form-submit’

Form Handling in Express

- Frontend

```
Filename:  formPage.ejs
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible"
content="IE=edge" />
    <meta name="viewport"
content="width=device-width, initial-scale=1.0" />
    <title>Form Page</title>
  </head>
  <body>
```

```
Page : <%= heading %>
    <form method="POST" action="/submit-form">
      <label for="name">Name:</label><br />
      <input type="text" id="name" name="name" /><br
/><br />
      <label for="email">Email:</label><br />
      <input type="email" id="email" name="email" /><br
/><br />
      <label for="password">Password:</label><br />
      <input type="password" id="password"
name="password" /><br /><br />
      <input type="submit" value="Submit" />
    </form></body></html>
```

Form Handling in Express

- Backend

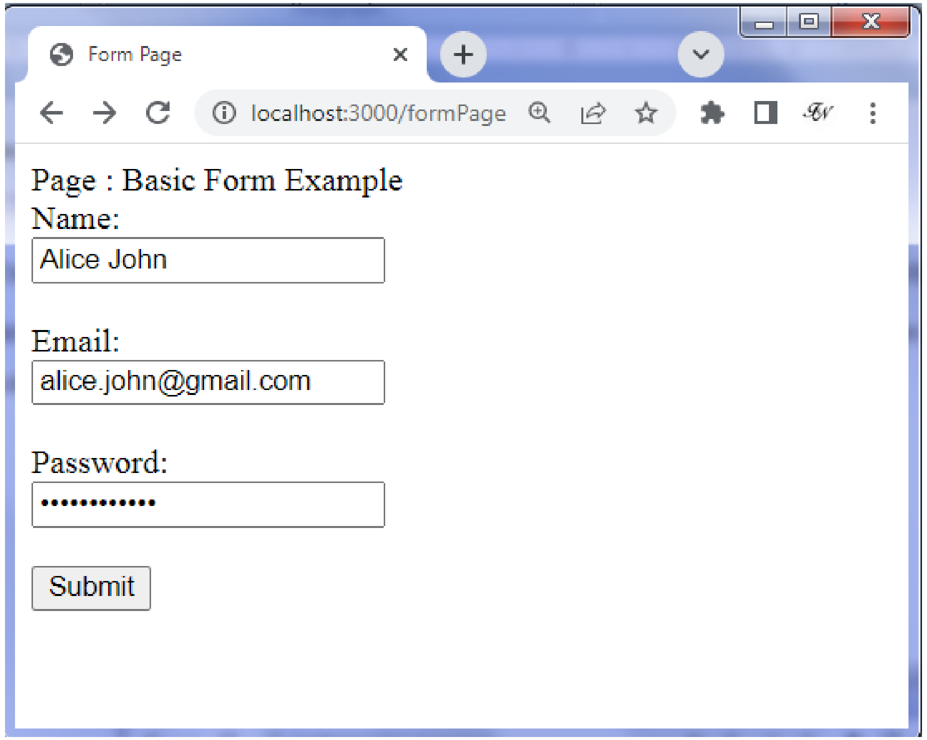
Filename: form.js

```
var express = require("express");
var app = express();
const bodyParser = require('body-parser')
app.set("view engine","ejs");
app.use(bodyParser.urlencoded({ extended: false }))
app.use(bodyParser.json())
app.use(express.json({ extended: false }))
app.get('/formPage', function (req, res) {
  res.render('formPage', {
    heading: 'Basic Form Example',
  })
})
```

```
app.post('/submit-form', (req, res) => {
  res.end(JSON.stringify(req.body))
})
app.listen(3000, function() {
    console.log("Node server is running..");
});
```

Form Handling in Express

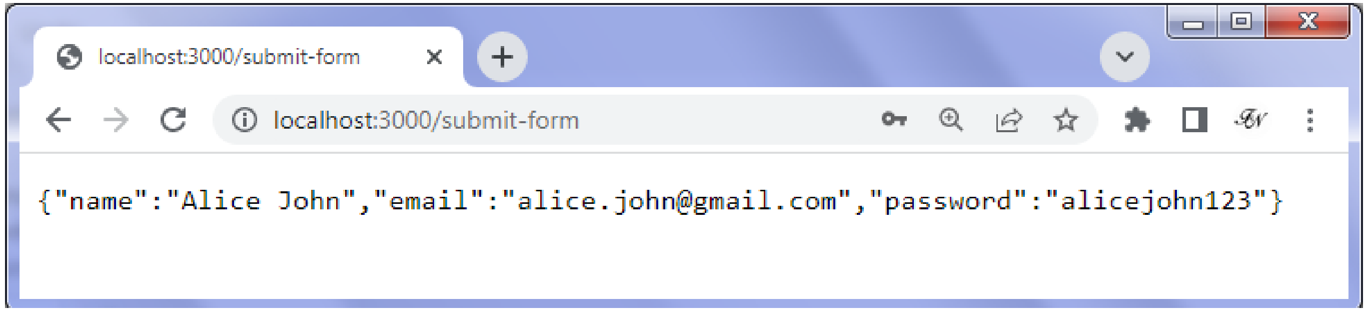
- The form is rendered as shown below on left on route http://localhost:3000/formPage.
- And on filling and submitting the form, we get a response from the server on route http://localhost:3000/submit-form as shown below on right.



Page : Basic Form Example
Name:
Alice John

Email:
alice.john@gmail.com

Password:
••••••••••••



```
{"name":"Alice John","email":"alice.john@gmail.com","password":"alicejohn123"}
```

Express.js Scaffolding

Express.js Scaffolding

- Scaffolding is creating the skeleton structure of application. It allows users to create own public directories, routes, views etc. Once the structure for app is built, user can start building it.
- If you will create API we will have to go through a number of operations, like we will have to manually make our public directory, add middleware, make separate configuration file, and so forth.
- A **scaffolding** device sets up every one of these things for us so we can straightforwardly begin with building our application.
- It allows us to create a skeleton for a web application when we work with ExpressJS so that we can directly get started with building our application.
- This is supported by many MVC frameworks.
- This reduces lots of development cost, as it has predefined templates that we can use for CRUD operations.

Get started

- Using **express-generator** package to install **‘express’** command line tool. **express-generator** is used to create the structure of application.

Installing express-generator:

Steps:

1. Navigate to the folder where app is to be built out using the terminal.
2. Now in the terminal, install **express-generator** using the following command.

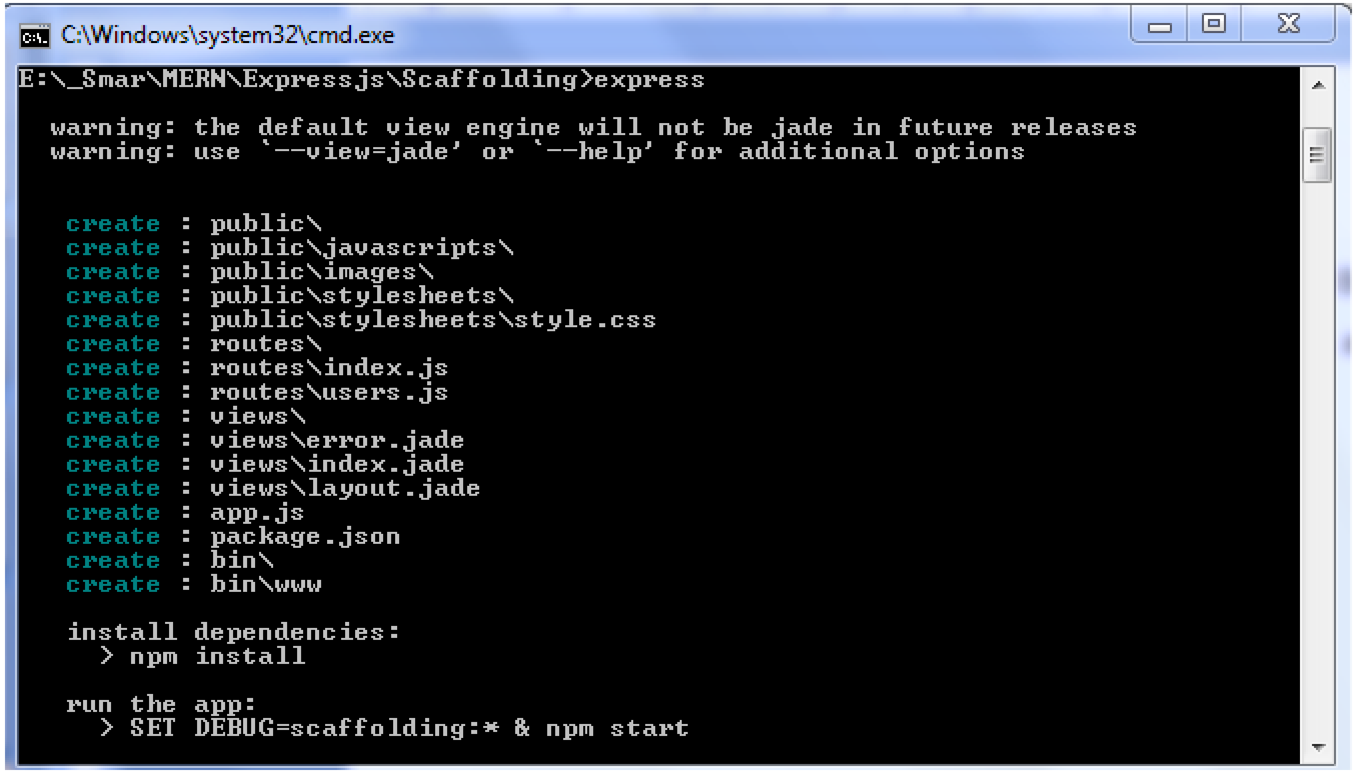
```
npm install express-generator -g
```

Note: npm installs the packages in two ways: **locally (default)** and **globally**

The packages installed locally are local to the current project and the packages installed globally are global packages which once installed can be used anywhere in your system.

Scaffolding the app

- Below image shows the scaffolding of the application. Basic structure of the application is being created if observed. Public directories, paths, routes, views, etc. are being created which would form the structure of application.



```
E:\_Smar\MERN\Express.js\Scaffolding>express

  warning: the default view engine will not be jade in future releases
  warning: use `--view=jade' or `--help' for additional options


   create : public\
   create : public\javascripts\
   create : public\images\
   create : public\stylesheets\
   create : public\stylesheets\style.css
   create : routes\
   create : routes\index.js
   create : routes\users.js
   create : views\
   create : views\error.jade
   create : views\index.jade
   create : views\layout.jade
   create : app.js
   create : package.json
   create : bin\
   create : bin\www

   install dependencies:
     > npm install

   run the app:
     > SET DEBUG=scaffolding:* & npm start
```

Scaffolding the app

- The project folder is constituted of various folders/files as shown below. Comparing the scaffolding structure and the project structure it can be clearly seen that the folders/files created in structural mode are present in project folder which was the purpose of scaffolding the application.

```
Express.js Scaffolding
 |- newapp
    |- bin
    |- public
    |- routes
    |- views
    |- app.js
    |- package.json
 |- sacf_app
    |- bin
```

```
    |- public
    |- routes
    |- views
    |- app.js
    |- package.json
 |- sacf_app2
    |- bin
    …..
    |- package.json
……
```

Scaffolding the app

Explaining the files/folders in project.

- **bin:** The file inside bin called www is the main configuration file of our app.
- **public:** The public folder contains the files which are to be made public for use like JavaScript files, CSS files, images etc.
- **routes:** The routes folder contains files which contain methods to help in navigating to different areas of the map. It contains various js files.
- **views:** The view folder contains various files which form the views part of the application. Example: homepage, the registration page, etc.

Scaffolding the app

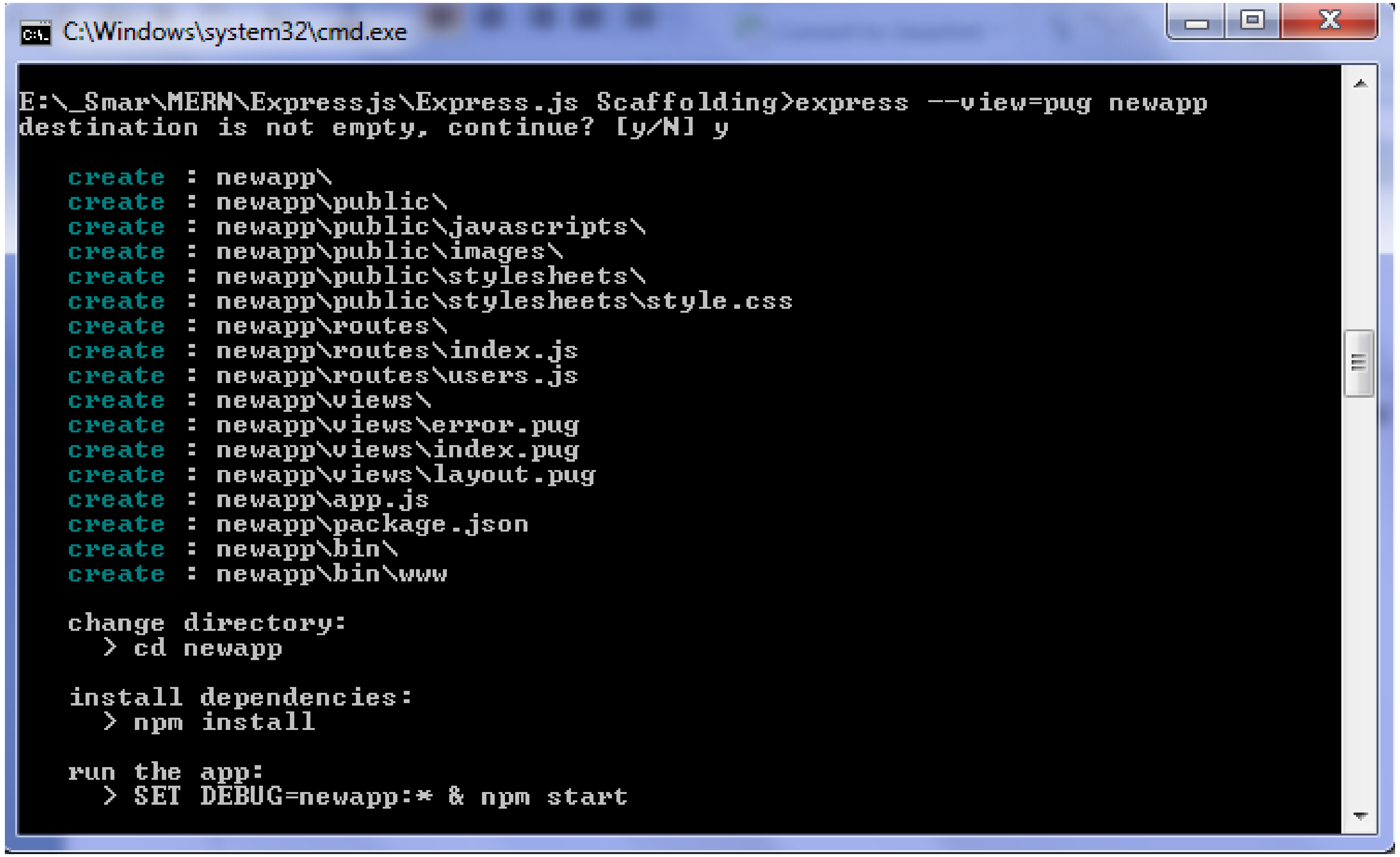
Explaining the files/folders in project.

- **app.js**: The app.js file is the main file which is the head of all the other files. The various packages installed have to be ‘required’ here. Besides this, it serves many other purposes like handling routers, middle-wares etc.
- **package.json**: package.json file is the manifest file of any Node.js project and express.js application.

  It contains the metadata of the project such as the packages and their versions used in the app (called dependencies), various scripts like start and test (run from terminal as ‘npm start’), name of the app, description of the app, version of the app, etc.

Scaffolding the app

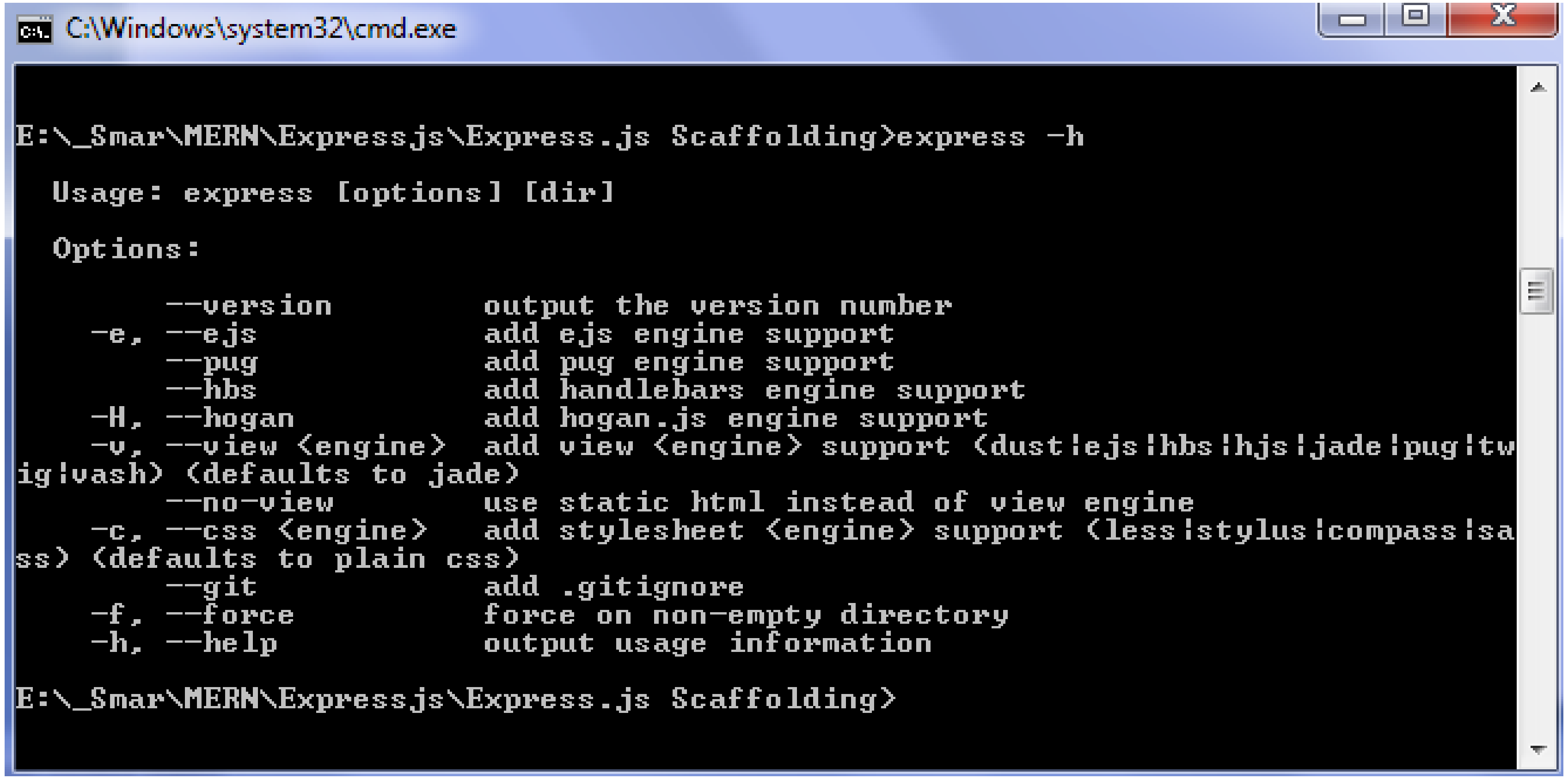
- The extension of the files at the time of writing this article is **.jade**. Change these file extensions to **.pug** as the jade project has changed to pug. Pug is a template engine for Node.js.



```
E:\_Smar\MERN\Expressjs\Express.js Scaffolding>express --view=pug newapp
destination is not empty, continue? [y/N] y

   create : newapp\
   create : newapp\public\
   create : newapp\public\javascripts\
   create : newapp\public\images\
   create : newapp\public\stylesheets\
   create : newapp\public\stylesheets\style.css
   create : newapp\routes\
   create : newapp\routes\index.js
   create : newapp\routes\users.js
   create : newapp\views\
   create : newapp\views\error.pug
   create : newapp\views\index.pug
   create : newapp\views\layout.pug
   create : newapp\app.js
   create : newapp\package.json
   create : newapp\bin\
   create : newapp\bin\www

   change directory:
     > cd newapp

   install dependencies:
     > npm install

   run the app:
     > SET DEBUG=newapp:* & npm start
```

Scaffolding the app

- This is the lists of express generator command options. Suppose we want to use a pug view engine to create new applications.



```
E:\_Smar\MERN\Expressjs\Express.js Scaffolding>express -h

  Usage: express [options] [dir]

  Options:

        --version        output the version number
    -e, --ejs            add ejs engine support
        --pug            add pug engine support
        --hbs            add handlebars engine support
    -H, --hogan          add hogan.js engine support
    -v, --view <engine>  add view <engine> support (dust|ejs|hbs|hjs|jade|pug|tw
ig|vash) (defaults to jade)
        --no-view        use static html instead of view engine
    -c, --css <engine>   add stylesheet <engine> support (less|stylus|compass|sa
ss) (defaults to plain css)
        --git            add .gitignore
    -f, --force          force on non-empty directory
    -h, --help           output usage information

E:\_Smar\MERN\Expressjs\Express.js Scaffolding>
```

Scaffolding the app

Running the Scaffold app:

- Install all the dependencies mentioned in the package.json file required to run the app using the following command:

```
npm install
```

- After the dependencies are installed, run the following command to start the ExpressJs app:

```
npm start
```

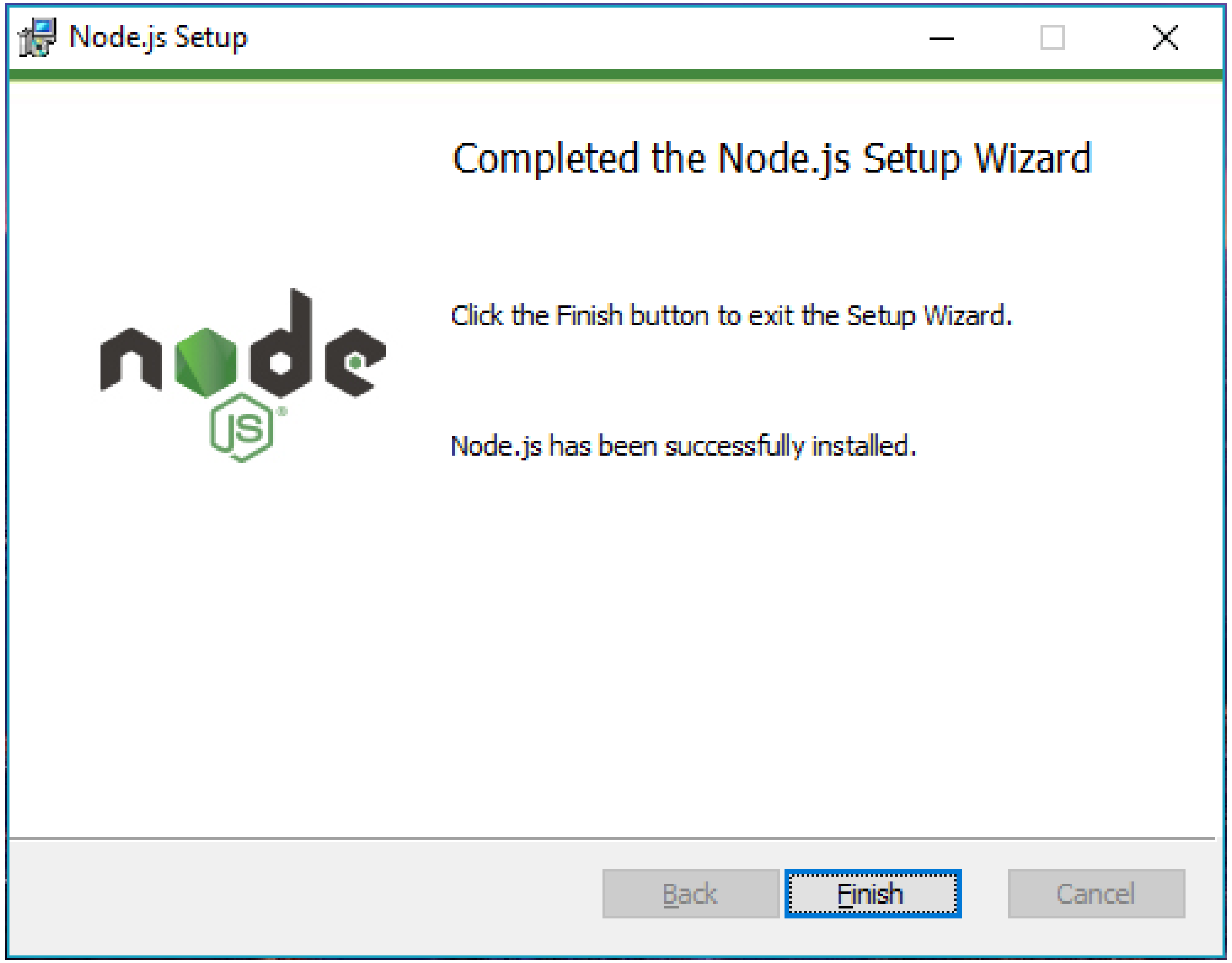
- Server will get started now. We can open the browser and can make a request at port 3000 as this is the default port number for us now.
- All the scaffolding is done now. All the required directories have been created. We can obviously create more directories and can make changes according to our needs but the basic structure is done now.

Database Access Step by Step

Prerequesties

- Node JS Installation
- Visual Studio Code Installation
- Express Installation
- MySQL Driver Installation

Node JS Installation



Completed the Node.js Setup Wizard

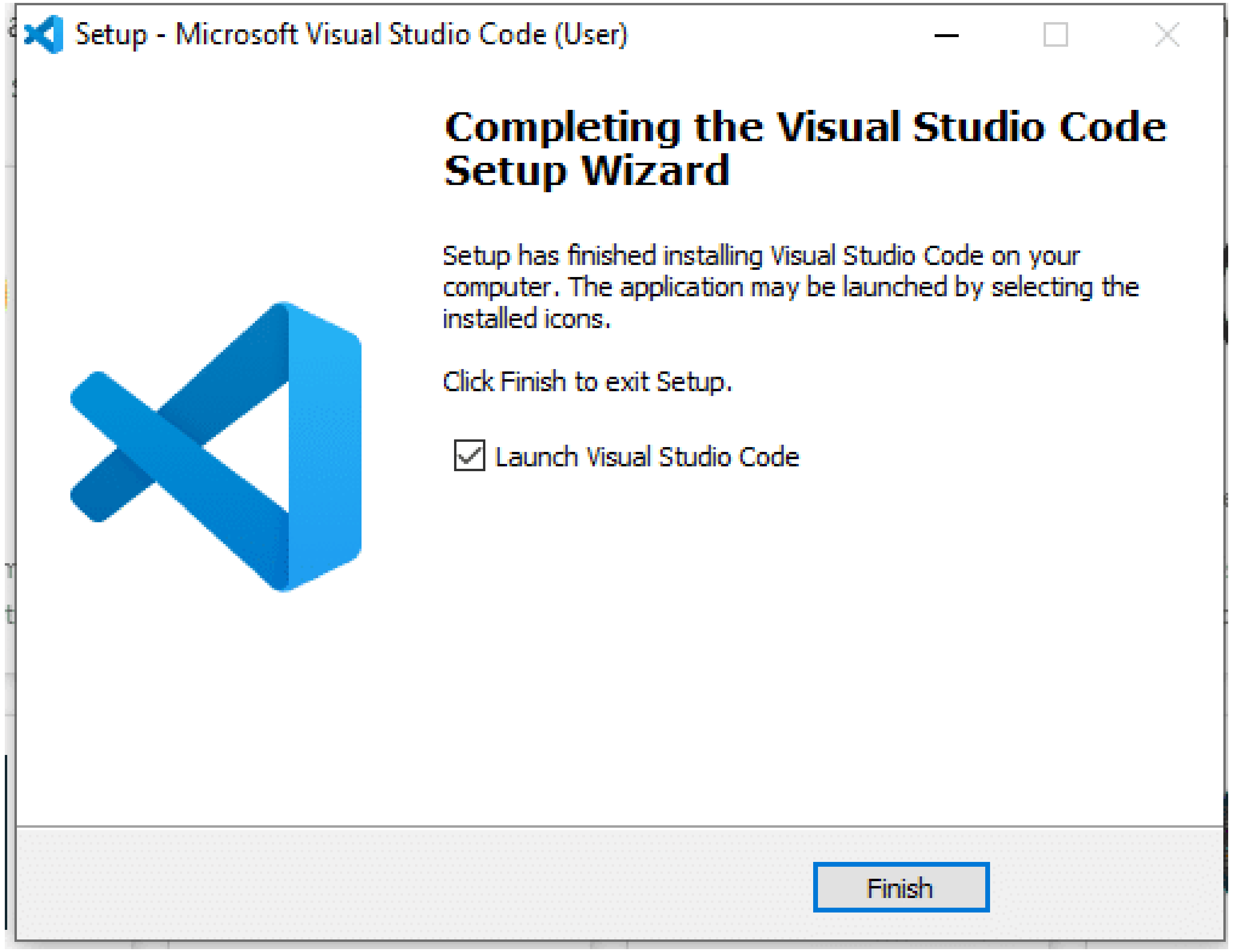
Click the Finish button to exit the Setup Wizard.

Node.js has been successfully installed.

✓ → Done

Database Access Step by Step

Visual Studio Code Installation



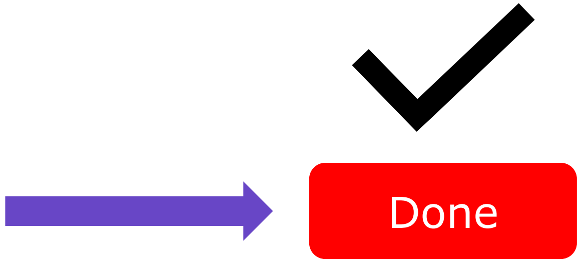
Setup - Microsoft Visual Studio Code (User)

Completing the Visual Studio Code Setup Wizard

Setup has finished installing Visual Studio Code on your computer. The application may be launched by selecting the installed icons.

Click Finish to exit Setup.

☑ Launch Visual Studio Code



Installation of Express.js

- Express.js can be installed using **npm**.

```
npm install -g express
```

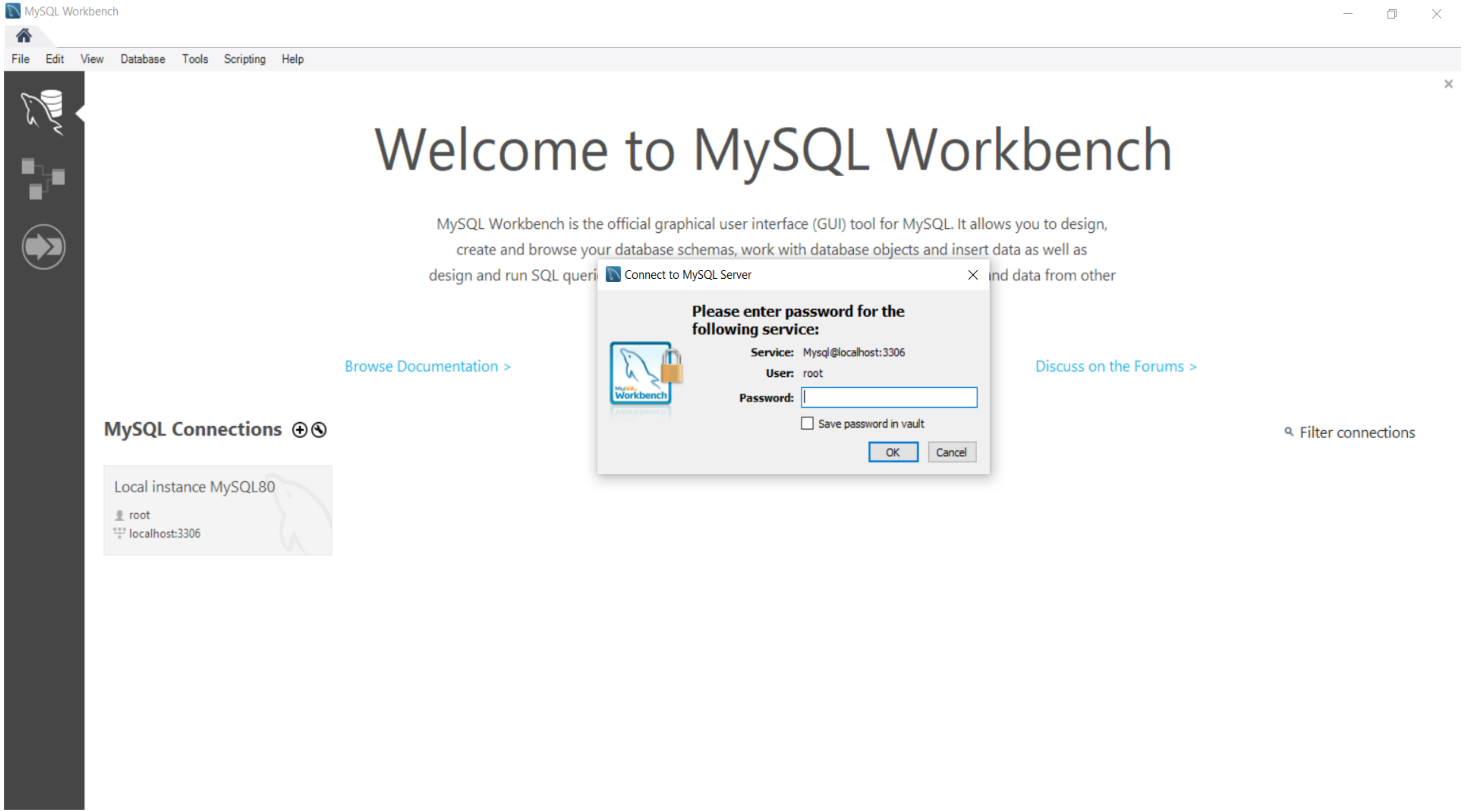
- The following command will install latest version of express.js local to the project folder.

```
E:\_Smart\MERN\Expressjs>   npm install express --save
```

✓ Done

- --save will update the **package.json** file by specifying express.js dependency.
- **npm** creates a **package-lock.json** file that includes detailed information about the modules that have installed which helps keep things fast and secure.

Installation of MySQL



Welcome to MySQL Workbench

MySQL Workbench is the official graphical user interface (GUI) tool for MySQL. It allows you to design, create and browse your database schemas, work with database objects and insert data as well as design and run SQL queri… …nd data from other

Connect to MySQL Server

Please enter password for the following service:

Service: Mysql@localhost:3306

User: root

Password:

☐ Save password in vault

OK Cancel

✓

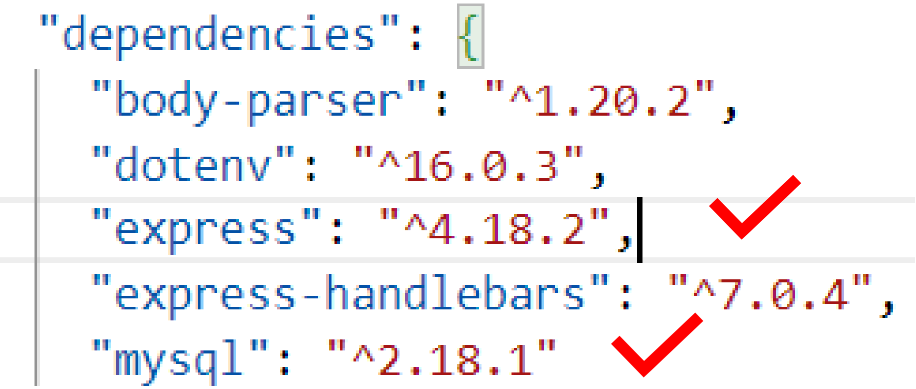
Done

Installation of MySQL Driver

- MySQL can be installed using **npm**.

npm install mysql → Done ✓

- Once done these all installation process, check the package.json file to see, whether all the dependencies installed properly or not.



```
"dependencies": {
  "body-parser": "^1.20.2",
  "dotenv": "^16.0.3",
  "express": "^4.18.2",
  "express-handlebars": "^7.0.4",
  "mysql": "^2.18.1"
```

Database Connectivity

- To create a database connectivity in MySQL.

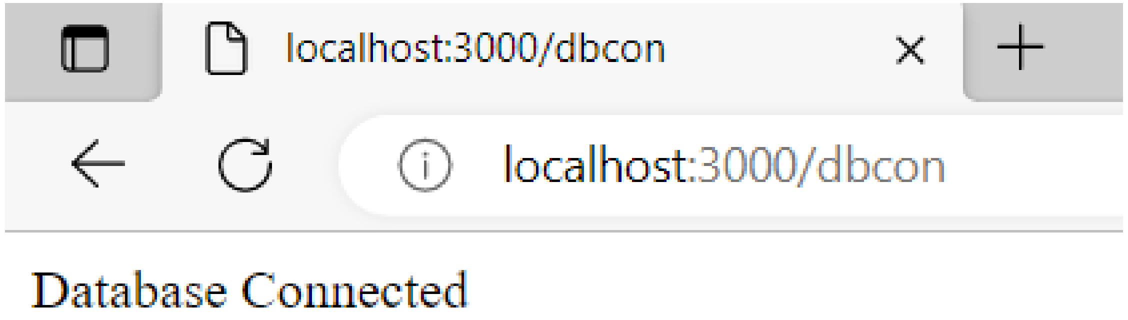
Create.js

```
const mysql=require('mysql');
//MySQL
const con=mysql.createPool({
    host :'localhost',
    user : 'root',
    password :'root@123',
    database :'crud_contact’
}); app.get('/dbcon',(req,res)=>{
con.getConnection((err) => {
```

Database Connectivity

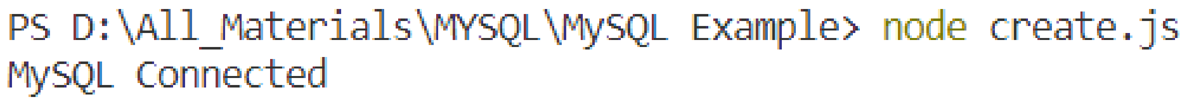
- To create a database connectivity in MySQL.

```
if(err)
   {        throw err;      }
   res.send('Database Connected');
   console.log('MySQL Connected');
}); })
app.listen('3000',()=>{
   console.log('Server started');
});
```



Database Connected

Output:



```
PS D:\All_Materials\MYSQL\MySQL Example> node create.js
MySQL Connected
```

Create Database

- To create a database connectivity in MySQL.

Create.js

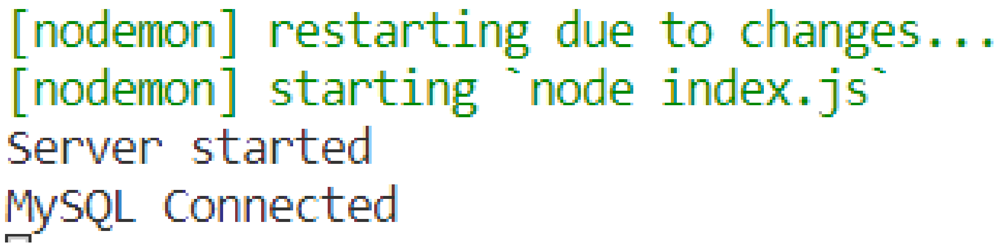
```
const mysql=require('mysql');
//MySQL
const con=mysql.createPool({
    host :'localhost',
    user : 'root',
    password :'root@123',
});
const app=express();
```

Create Database

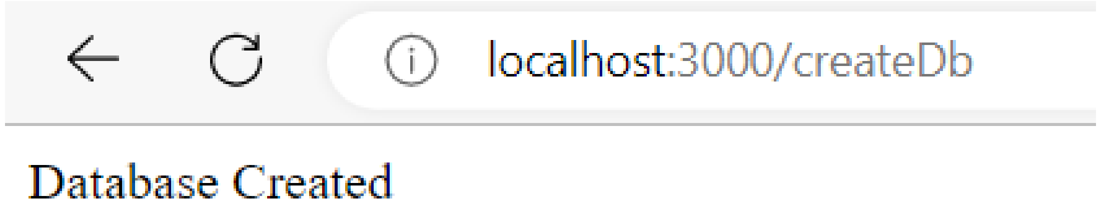
- To create a database connectivity in MySQL.

```
app.get('/createDb',(req,res)=>{
    let sql='CREATE DATABASE teachers';
    db.query(sql,(err,result)=>{
        res.send('Database Created');
    });
});
app.listen('3000',()=>{
    console.log('Server started');
});
```

//Output



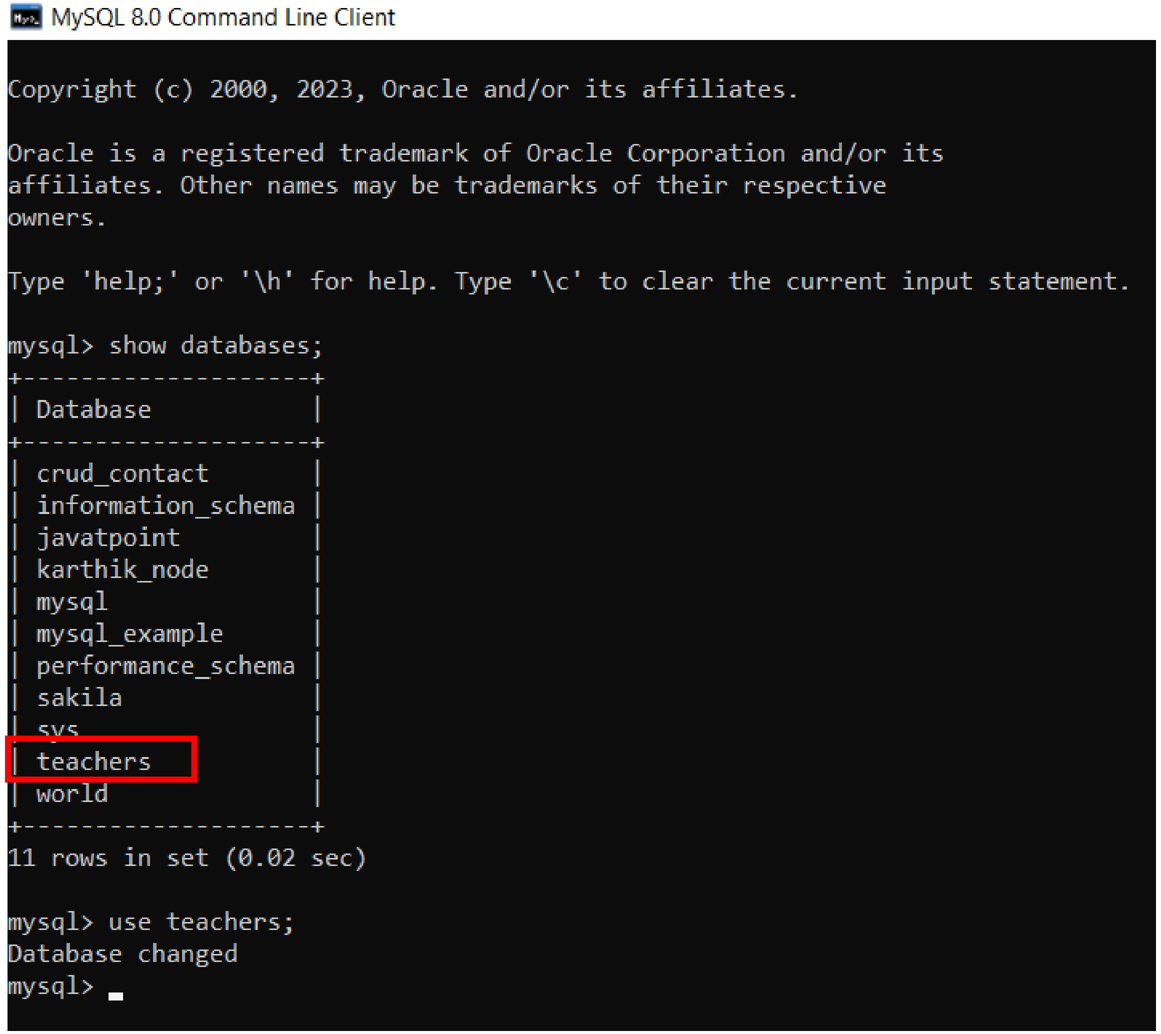
```
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
Server started
MySQL Connected
```



localhost:3000/createDb

Database Created

Create Table



MySQL 8.0 Command Line Client

```
Copyright (c) 2000, 2023, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| crud_contact       |
| information_schema |
| javatpoint         |
| karthik_node       |
| mysql              |
| mysql_example      |
| performance_schema |
| sakila             |
| sys                |
| teachers           |
| world              |
+--------------------+
11 rows in set (0.02 sec)

mysql> use teachers;
Database changed
mysql> _
```

Create Table

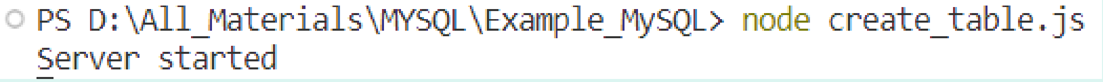
- To create a table in MySQL, use the "CREATE TABLE" statement.
- Make sure you define the name of the database when you create the connection:

Create.js

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


//MySQL
const con=mysql.createPool({
    host :'localhost',
    user : 'root',
    password :'root@123',
    database :'crud_contact’
});
```

Create Table

```
const app=express();


app.get('/createTable',(req,res)=>{
    let sql='CREATE TABLE users (ID int, NAME varchar(150), AGE int, CITY varchar(150)';
    db.query(sql,(err,result)=>{
        res.send('Table Created');
    });
});
app.listen('3000',()=>{
    console.log('Server started');
});
```



```
PS D:\All_Materials\MYSQL\Example_MySQL> node create_table.js
Server started
```

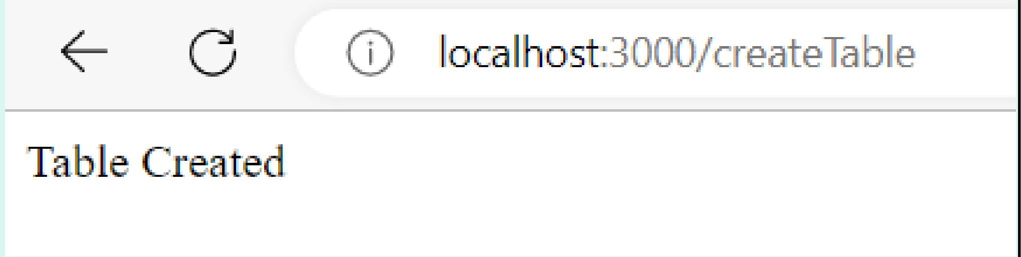
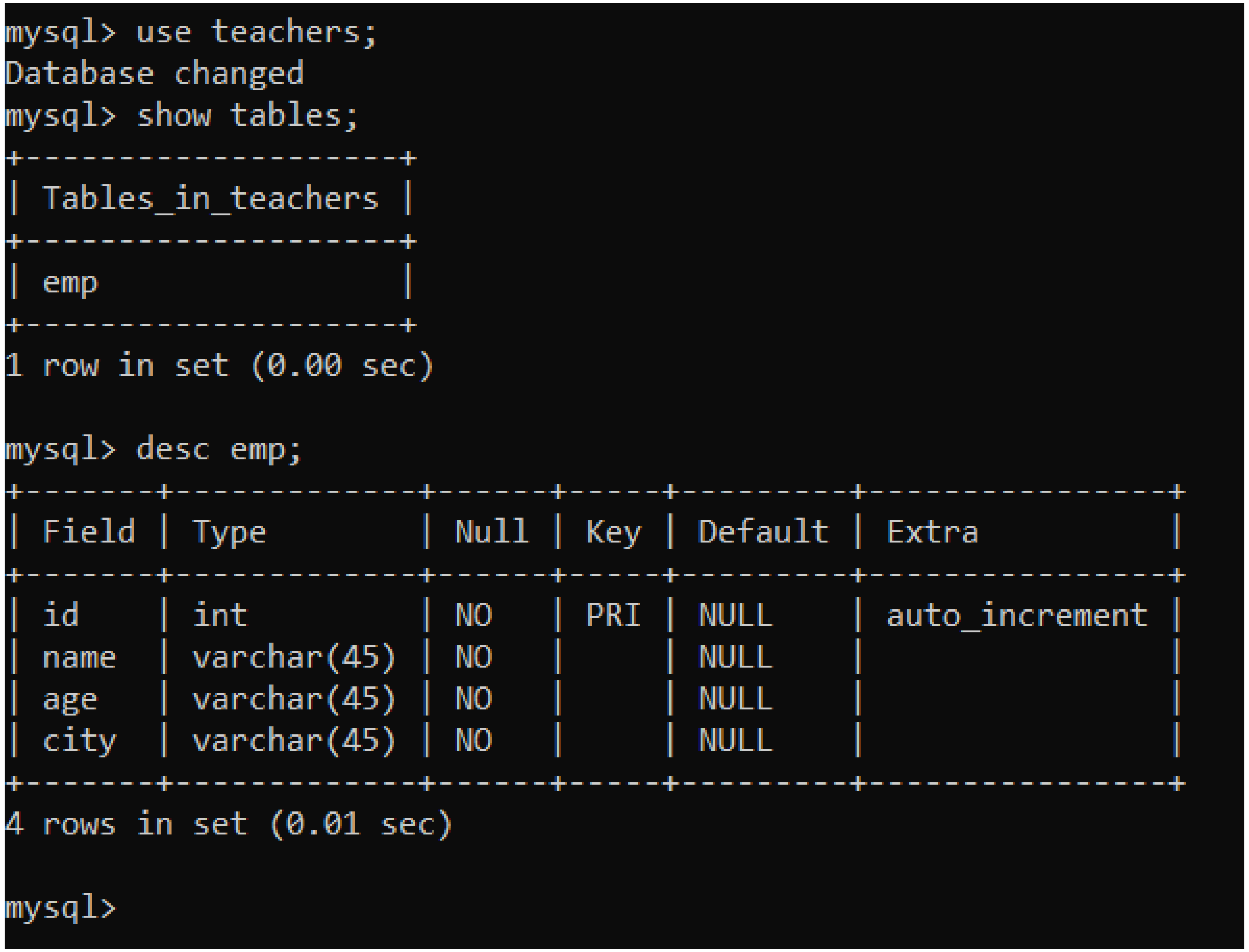


Table Created

Create Table



```
mysql> use teachers;
Database changed
mysql> show tables;
+--------------------+
| Tables_in_teachers |
+--------------------+
| emp                |
+--------------------+
1 row in set (0.00 sec)

mysql> desc emp;
+-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| id    | int         | NO   | PRI | NULL    | auto_increment |
| name  | varchar(45) | NO   |     | NULL    |                |
| age   | varchar(45) | NO   |     | NULL    |                |
| city  | varchar(45) | NO   |     | NULL    |                |
+-------+-------------+------+-----+---------+----------------+
4 rows in set (0.01 sec)

mysql>
```

Insert value into Table

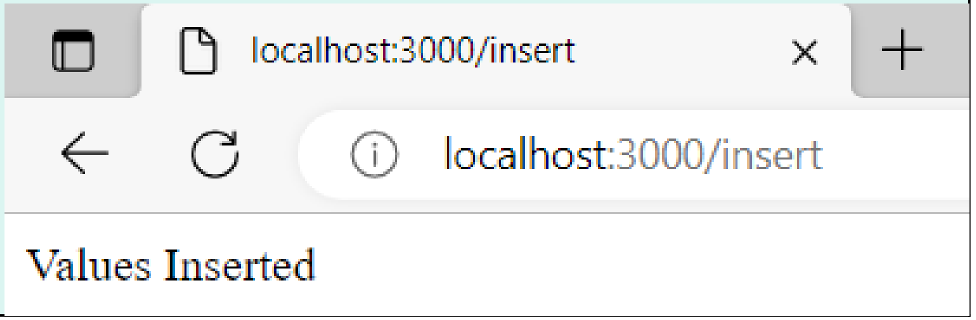
- In this studentscontroller.js file we have MySQL manipulation operations.

insert.js

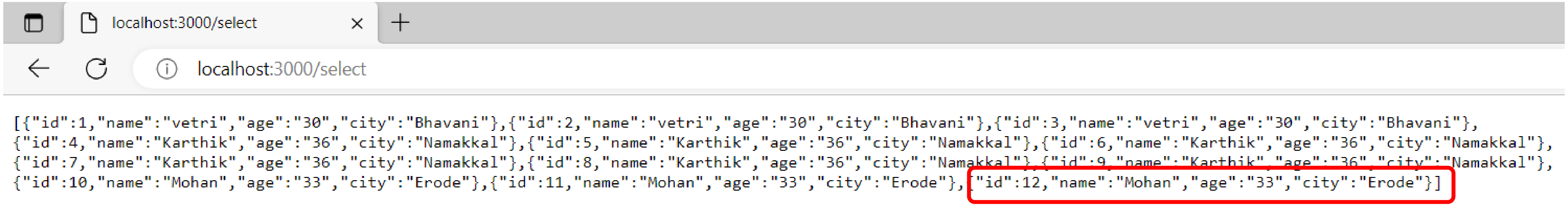
```
const mysql=require(‘mysql’);

//MySQL
const con=mysql.createPool
({
    host :'localhost',
    user : 'root',
    password :'root@123',
    database :'crud_contact’
});
const app=express();

app.get('/insert',(req,res)=>
{
```

Insert value into Table

```
let id = req.params.id;
    let sql='INSERT INTO emp(NAME,AGE,CITY) VALUES("Mohan",33,"Erode")';
    db.query(sql,[id],(err,result)=>{
        res.send('Values Inserted');
        res.end();
    });
});
app.listen('3000',()=>{
    console.log('Server started');
});
```



Values Inserted



```
[{"id":1,"name":"vetri","age":"30","city":"Bhavani"},{"id":2,"name":"vetri","age":"30","city":"Bhavani"},{"id":3,"name":"vetri","age":"30","city":"Bhavani"},
{"id":4,"name":"Karthik","age":"36","city":"Namakkal"},{"id":5,"name":"Karthik","age":"36","city":"Namakkal"},{"id":6,"name":"Karthik","age":"36","city":"Namakkal"},
{"id":7,"name":"Karthik","age":"36","city":"Namakkal"},{"id":8,"name":"Karthik","age":"36","city":"Namakkal"},{"id":9,"name":"Karthik","age":"36","city":"Namakkal"},
{"id":10,"name":"Mohan","age":"33","city":"Erode"},{"id":11,"name":"Mohan","age":"33","city":"Erode"},{"id":12,"name":"Mohan","age":"33","city":"Erode"}]
```

Select Statement

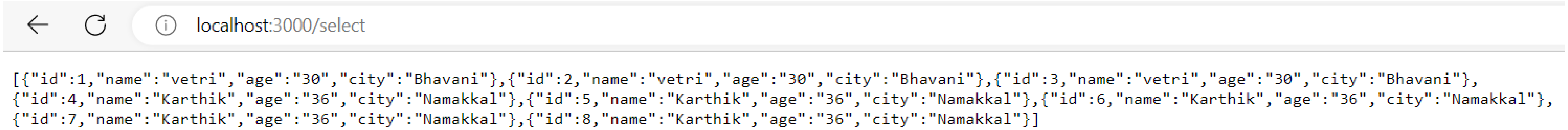
- Select * from statement is to display all the records in the table.

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

const db=mysql.createConnection({
    host:'localhost',
    user:'root',
    password:'root@123',
    database:'teachers'
});
const app=express();

app.get('/select',(req,res)=>{
    res.writeHead(200,{'content-type':'text/json'});
    let id = req.params.id;
```

Select Statement

```
 let sql='select * from emp';
    db.query(sql,[id],(err,result)=>{
        res.write(JSON.stringify(result));
        res.end();
    });
});
app.listen('3000',()=>{
    console.log('Server started');
});
```

Output:



```
[{"id":1,"name":"vetri","age":"30","city":"Bhavani"},{"id":2,"name":"vetri","age":"30","city":"Bhavani"},{"id":3,"name":"vetri","age":"30","city":"Bhavani"},
{"id":4,"name":"Karthik","age":"36","city":"Namakkal"},{"id":5,"name":"Karthik","age":"36","city":"Namakkal"},{"id":6,"name":"Karthik","age":"36","city":"Namakkal"},
{"id":7,"name":"Karthik","age":"36","city":"Namakkal"},{"id":8,"name":"Karthik","age":"36","city":"Namakkal"}]
```

Update value into Table

- To update values in the table.

Update.js

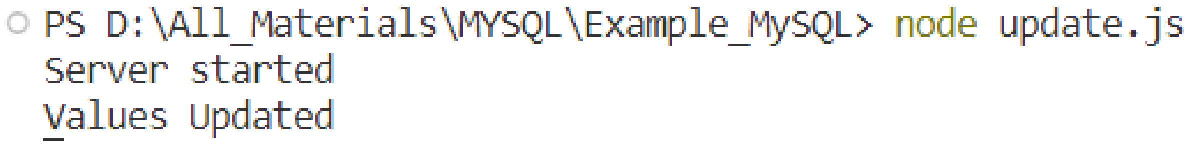
```
const express=require('express');
const mysql=require('mysql');

const db=mysql.createConnection({
    host:'localhost',
    user:'root',
    password:'root@123',
    database:'teachers'
});

const app=express();
```

Update value into Table

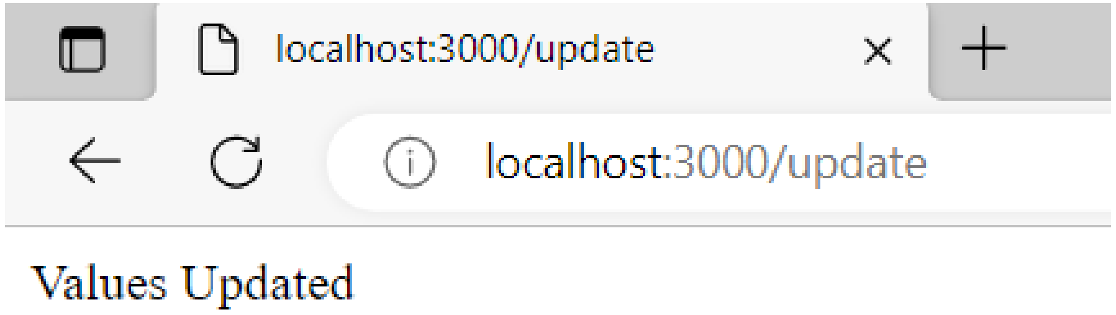
```
app.get('/update',(req,res)=>
{
let id = req.query.id;
let sql="UPDATE emp SET name = 'Mohankumar' WHERE city = 'Erode'";;
    db.query (sql,[id],(err,result)=>
{
        if(!err)
{
        res.write('Updated');
    }
else{
        res.send(err);
    }
        res.end();
    });
 });
```

Update value into Table

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



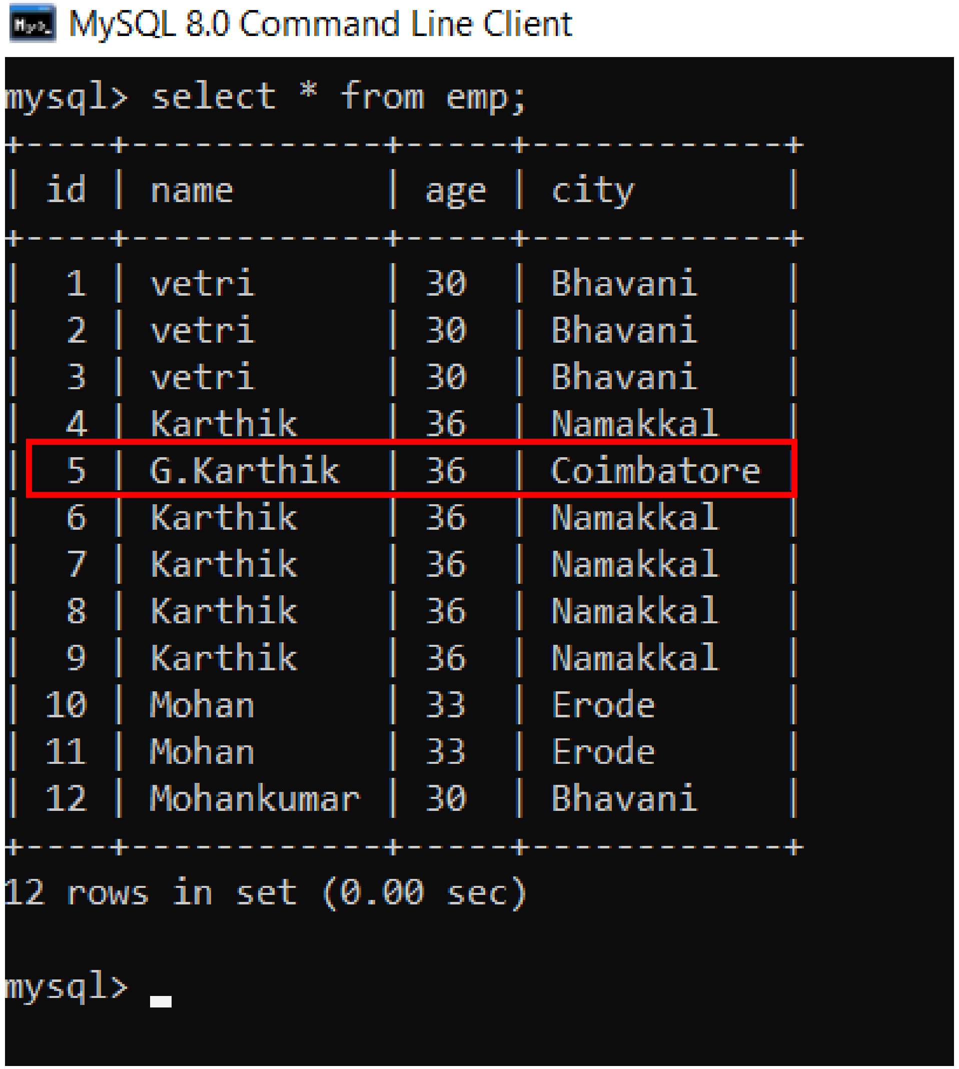
```
○ PS D:\All_Materials\MYSQL\Example_MySQL> node update.js
  Server started
  Values Updated
```



localhost:3000/update

Values Updated

Update value into Table



MySQL 8.0 Command Line Client

```
mysql> select * from emp;
+----+------------+-----+------------+
| id | name       | age | city       |
+----+------------+-----+------------+
|  1 | vetri      | 30  | Bhavani    |
|  2 | vetri      | 30  | Bhavani    |
|  3 | vetri      | 30  | Bhavani    |
|  4 | Karthik    | 36  | Namakkal   |
|  5 | G.Karthik  | 36  | Coimbatore |
|  6 | Karthik    | 36  | Namakkal   |
|  7 | Karthik    | 36  | Namakkal   |
|  8 | Karthik    | 36  | Namakkal   |
|  9 | Karthik    | 36  | Namakkal   |
| 10 | Mohan      | 33  | Erode      |
| 11 | Mohan      | 33  | Erode      |
| 12 | Mohankumar | 30  | Bhavani    |
+----+------------+-----+------------+
12 rows in set (0.00 sec)

mysql> _
```

Delete student record

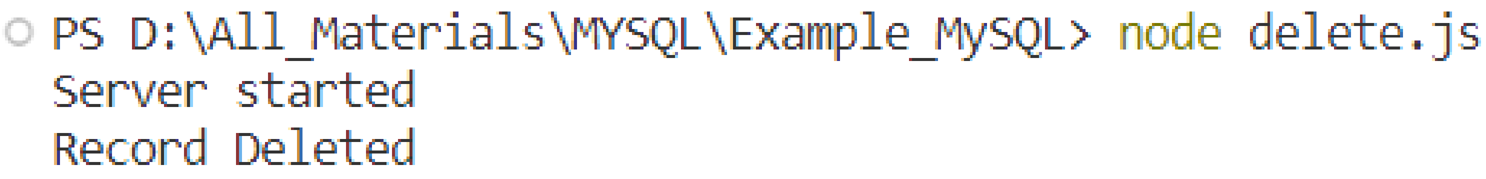
studentscontroller.js

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


//MySQL
const con=mysql.createPool({
    host :'localhost',
    user : 'root',
    password :'root@123',
    database :'crud_contact’
});
```

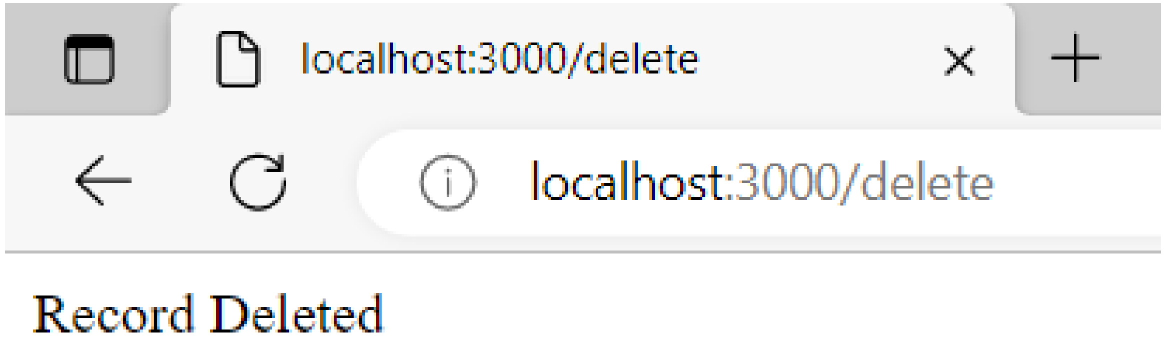
Delete student record

```
app.get('/delete',(req,res)=>
{
let id = req.params.id;
    let sql='DELETE FROM emp WHERE id =14';
    db.query(sql,[id],(err,result)=>
      {
        if(err) throw err;
        res.send('Record Deleted');
        res.end();
    });
});
app.listen('3000',()=>{
    console.log('Server started');
});
```

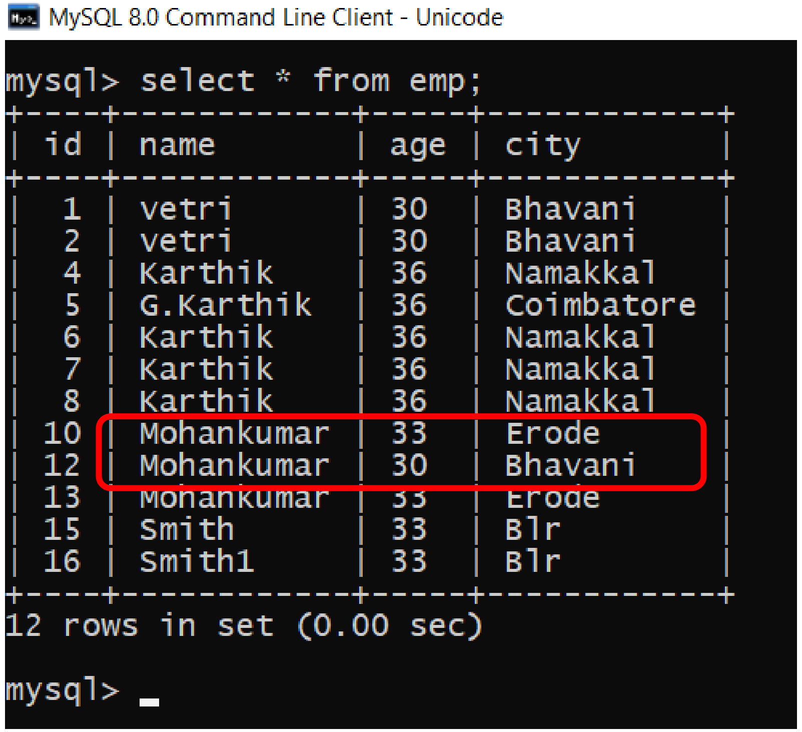
Delete student record



```
PS D:\All_Materials\MYSQL\Example_MySQL> node delete.js
Server started
Record Deleted
```



Record Deleted



```
mysql> select * from emp;
+----+-------------+-----+------------+
| id | name        | age | city       |
+----+-------------+-----+------------+
|  1 | vetri       | 30  | Bhavani    |
|  2 | vetri       | 30  | Bhavani    |
|  4 | Karthik     | 36  | Namakkal   |
|  5 | G.Karthik   | 36  | Coimbatore |
|  6 | Karthik     | 36  | Namakkal   |
|  7 | Karthik     | 36  | Namakkal   |
|  8 | Karthik     | 36  | Namakkal   |
| 10 | Mohankumar  | 33  | Erode      |
| 12 | Mohankumar  | 30  | Bhavani    |
| 13 | Mohankumar  | 33  | Erode      |
| 15 | Smith       | 33  | Blr        |
| 16 | Smith1      | 33  | Blr        |
+----+-------------+-----+------------+
12 rows in set (0.00 sec)

mysql> _
```

THANK YOU

SmartCliff
Career Mobility Solutions