

Express.js



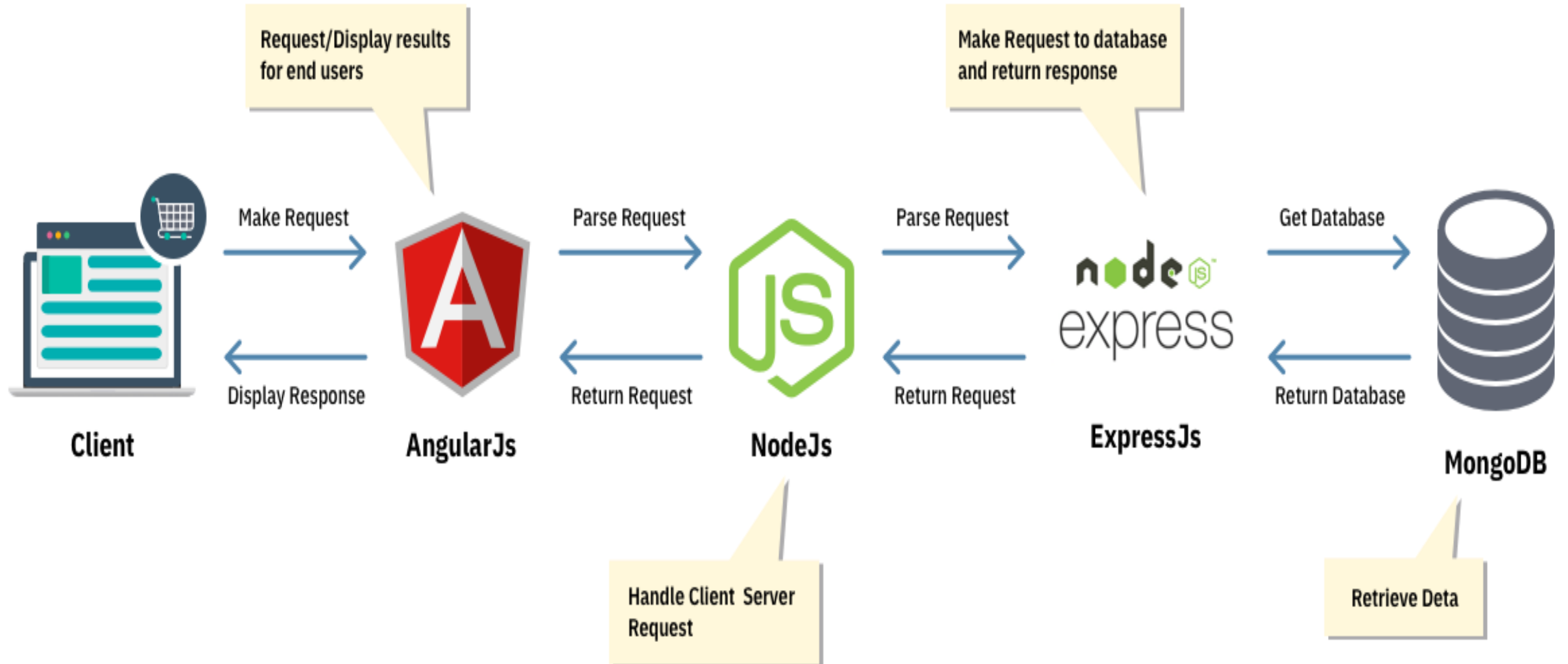
EXPRESS.JS

- Express.js is the **most popular** Node.js web application framework used today.
- Express.js is a **minimal** yet **flexible** and **powerful** web development framework inspired by **Sinatra**.
- Features of Express.js include:
 - Robust **routing**
 - Focus on **high performance**
 - View system supporting several **template engines**
 - Content negotiation
 - Executable for **generating applications** quickly

EXPRESS

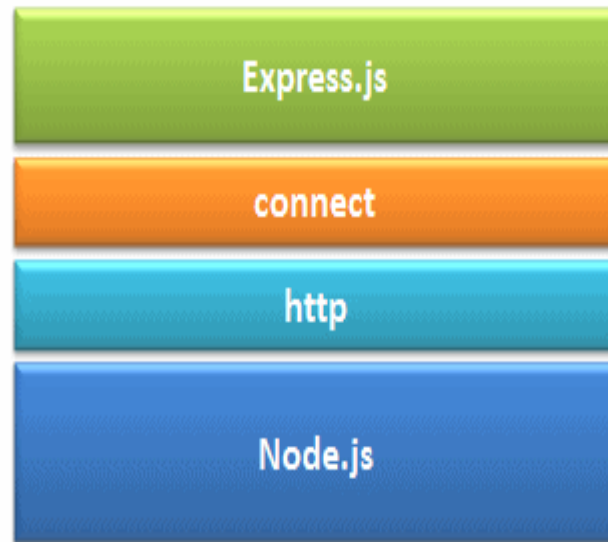
- ❑ Express.js is a fast and lightweight framework used majorly for web application development and the Node.js Developers all over the world are totally in love with this framework. Express.js provides all the features of web application without overshadowing the Node.js features.
- ❑ Express.js is a web application framework that is built on top of Node.js. It provides a minimal interface with all the tools required to build a web application.
- ❑ Express.js adds flexibility to an application with a huge range of modules available on npm that you can directly plug into Express as per requirement. It helps in easy management of the flow of data between server and routes in the server-side applications.
- ❑ Express is majorly responsible for handling the backend part in the MEAN or MERN stack.

Express with MEAN



EXPRESS

- ❑ Express.js is a web application framework for Node.js. It provides various features that make web application development fast and easy which otherwise takes more time using only Node.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

Features of Express.js

- ❑ Express quickens the development pace of a web application.
- ❑ It also helps in creating mobile and web application of single-page, multi-page, and hybrid types
- ❑ Express can work with various templating engines such as Pug, Mustache, and EJS.
- ❑ Express follows the Model-View-Controller (MVC) architecture.
- ❑ It makes the integration process with databases such as MongoDB, Redis, MySQL effortless.
- ❑ Express also defines an error-handling middleware.
- ❑ It helps in simplifying the configuration and customization steps for the application.

Advantages of Express.js

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

EXPRESS

Database Integration

Database systems supported by Express:

- Cassandra
- Couchbase
- CouchDB
- LevelDB
- MySQL
- MongoDB
- Neo4j
- PostgreSQL
- Redis
- SQL Server
- SQLite
- ElasticSearch



Express Installation

In order to install Express.js in your system, first, you need to make sure that you have Node.js already installed. Once you are done with Node.js installation, the next step is to install Express.

To install Express.js, first, you need to create a project directory and create a package.json file which will be holding the project dependencies. Below is the code to perform the same:

npm init

Now, you can install the express.js package in your system. To install it globally, you can use the below command:

npm install -g express

Or, if you want to install it locally into your project folder, you need to execute the below command:

npm install express --save

Hello world example

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

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`)
})
```

Hello world example

- ❑ This app starts a server and listens on port 3000 for connections. The app responds with “Hello World!” for requests to the root URL (/) or route. For every other path, it will respond with a 404 Not Found.
- ❑ The example above is actually a working server: Go ahead and click on the URL shown. You’ll get a response, with real-time logs on the page, and any changes you make will be reflected in real time.

Create Server in Express

Express.js provides an easy way to create web server and render HTML pages for different HTTP requests by configuring routes for your application.

Web Server

First of all, import the Express.js module and create the web server as shown below.

```
//app.js: Express.js Web Server
```

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

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

```
// define routes here..
```

```
var server = app.listen(5000, function () {
```

```
    console.log('Node server is running..');
```

```
});
```

Create Server in Express

- ❑ In the above example, we imported Express.js module using `require()` function. The express module returns a function. This function returns an object which can be used to configure Express application (app in the above example).
- ❑ The app object includes methods for routing HTTP requests, configuring middleware, rendering HTML views and registering a template engine.
- ❑ The `app.listen()` function creates the Node.js web server at the specified host and port. It is identical to Node's `http.Server.listen()` method.
- ❑ Run the above example using `node app.js` command and point your browser to `http://localhost:5000`. It will display Cannot GET / because we have not configured any routes yet.

Create Server in Express

// Import the top-level function of express

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

// Creates an Express application using the top-level function

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

// Define port number as 3000

```
const port = 3000;
```

// Routes HTTP GET requests to the specified path "/" with the specified callback function

```
app.get('/', function(request, response) {
```

```
  response.send('Hello, World!');
```

```
});
```

// Make the app listen on port 3000

```
app.listen(port, function() {
```

```
  console.log('Server listening on http://localhost:' + port);
```

```
});
```

Test Express Api on Postman

The screenshot shows the Postman application interface. At the top, there is a blue header bar. Below it, the request method is set to 'GET' and the URL is 'http://localhost:3000'. A 'Send' button is visible on the right. The 'Params' tab is selected, showing a table with columns 'KEY', 'VALUE', and 'DESCRIPTION'. The 'Body' tab is also visible. The response status is '200 OK' with a time of '650 ms' and a size of '215 B'. The response body is displayed in the 'Pretty' view, showing 'Hello World'.

GET Send

Params Authorization Headers Body Pre-request Script Tests Cookies Code

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (6) Test Results Status: 200 OK Time: 650 ms Size: 215 B

Pretty Raw Preview HTML

i 1 Hello World

EXPRESS

Routes

Routing refers to the definition of application end points (URIs) and how they respond to client requests

A **route method** is derived from one of the HTTP methods, and is attached to an instance of the express class

```
1 var express = require('express');
2
3 var app = express();
4
5 //GET method route
6 app.get('/', function(req, res){
7
8   res.send('GET request to the homepage');
9
10 });
11
12 //POST method route
13 app.post('/', function(req, res){
14
15   res.send('POST request to the homepage');
16
17 }); |
```

Importing Express Module

Creating Express Instance

Callback function

Path (route)

Route Methods

HTTP request

HTTP response

EXPRESS

Route Handlers

Provide multiple callback functions which behave like middleware to handle a request

Exception -> Callbacks might invoke `next('route')` to bypass the remaining route callbacks

```
1 app.get('/example/a', function (req, res) {  
2   res.send('Hello from A!')  
3 }) |
```

A single callback function can handle a route

```
1 app.get('/example/b', function (req, res, next) {  
2   console.log('the response will be sent by the next function ...')  
3   next()  
4 }, function (req, res) {  
5   res.send('Hello from B!')  
6 }) |
```

Next to bypass the remaining route callbacks



Used to impose pre-conditions on a route, then pass control to subsequent routes (if no reason to proceed with the current route)

EXPRESS

Routes (app.route)

- Create chainable route handlers for a route path by using `app.route()`
- Path is specified at a single location
- Creating modular routes is helpful, as it reduces redundancy and typos

Creating chainable route

GET Method

POST Method

PUT Method

```
1 app.route('/book')
2   .get(function (req, res) {
3     res.send('Get a random book')
4   })
5   .post(function (req, res) {
6     res.send('Add a book')
7   })
8   .put(function (req, res) {
9     res.send('Update the book')
10  }) |
```

EXPRESS

Routes (express.Router)

```
1 var express = require('express')
2 var router = express.Router()
3
4 // middleware that is specific to this router
5 router.use(function timeLog (req, res, next) {
6   console.log('Time: ', Date.now())
7   next()
8 })
9 // define the home page route
10 router.get('/', function (req, res) {
11   res.send('Birds home page')
12 })
13 // define the about route
14 router.get('/about', function (req, res) {
15   res.send('About birds')
16 })
17
18 module.exports = router
```

Creates a router
as a module

Loads a
middleware
function

Defines
Home Route

Define About
Route

- Router class to create modular, mountable route handlers
- A Router instance is a complete middleware and routing system

ROUTE EXAMPLE

router1.js

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

var app = express();

app.get('/', function (req, res) {
  res.send('<html><body><h1>Hello World</h1></body></html>');
});

app.post('/submit-data', function (req, res) {
  res.send('POST Request ! Just For Testing');
});

app.put('/update-data', function (req, res) {
  res.send('PUT Request! Just For Testing');
});

app.delete('/delete-data', function (req, res) {
  res.send('DELETE Request ! Just For Testing');
});

var server = app.listen(5000, function () {
  console.log('Node server is running..');
});
```

ROUTE EXAMPLE

Test the program using api testing tool postman

In the above example, `app.get()`, `app.post()`, `app.put()` and `app.delete()` methods define routes for HTTP GET, POST, PUT, DELETE respectively. The first parameter is a path of a route which will start after base URL. The callback function includes request and response object which will be executed on each request.

Run the above example using `node server.js` command, and point your browser to `http://localhost:5000` and you will see the following result.

ROUTE EXAMPLE

router2.js

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

const app = express()

const port = 3000
app.get('/', (req, res) => res.send('Hello World!'))

app.route('/book')

  .get(function (req, res) {
    res.send('Get a random book')
  })

  .post(function (req, res) {
    res.send('Add a book')
  })

  .put(function (req, res) {
    res.send('Update the book')
  })

  .delete(function (req, res) {
    res.send('Delete the book')
  })

app.listen(port, () => console.log(`Example app listening on port port!`))

// Test the program using api testing tool postman
```


DYNAMIC ROUTE



Express JS : Dynamic Routes

1. ExpressJS allows to build URL's dynamically as well
2. Often we need to work with content/id's which are dynamic in nature and NOT static values
3. Using dynamic routes allows us to pass parameters and process based on them
4. E.g.

```
app.get('/:id', function(req, res){  
  res.send('Dynamic Value is ' + req.params.id);  
});
```

DYNAMIC ROUTE

Dynamic Routing and URL Building in Express JS

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.

ROUTE EXAMPLE

droute1.js

```
//imports express module
```

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

```
//initializes express app
```

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

```
//where username is variable starting with colon
```

```
app.get('/profile/:username', function(request, response){
```

```
    response.send('Welcome to '+request.params.username+' profile.');
```

```
});
```

```
//listens to server at port 3000
```

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

```
// Test the program using api testing tool postman http://localhost:3000/profile/Tapan
```

ROUTE EXAMPLE

droute2.js

```
//imports express module
```

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

```
//initializes express app
```

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

```
//where username is variable starting with colon
```

```
app.get('/profile/:username/:post_id', function(request, response){
```

```
    response.send('Welcome to ' + request.params.username + ' profile. We are fetching post_id'  
    + request.params.post_id);
```

```
});
```

```
//listens to server at port 3000
```

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

// Test the program using api testing tool postman <http://localhost:3000/profile/Tapan/20>

Handle POST Request

Body Parser

To handle HTTP POST request in Express.js version 4 and above, you need to install middleware module called body-parser. The middleware was a part of Express.js earlier but now you 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
```

EXPRESS

Handle POST Request

Here, you will learn how 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.

Handle POST Request

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta charset="utf-8" />
  <title></title>
</head>
<body>
  <form action="/submit-student-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

Now, import body-parser and get the POST request data as shown below.

```
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(__dirname + '/index.html');
});
app.post('/submit-student-data', function (req, res) {
  var name = req.body.firstName + ' ' + req.body.lastName;

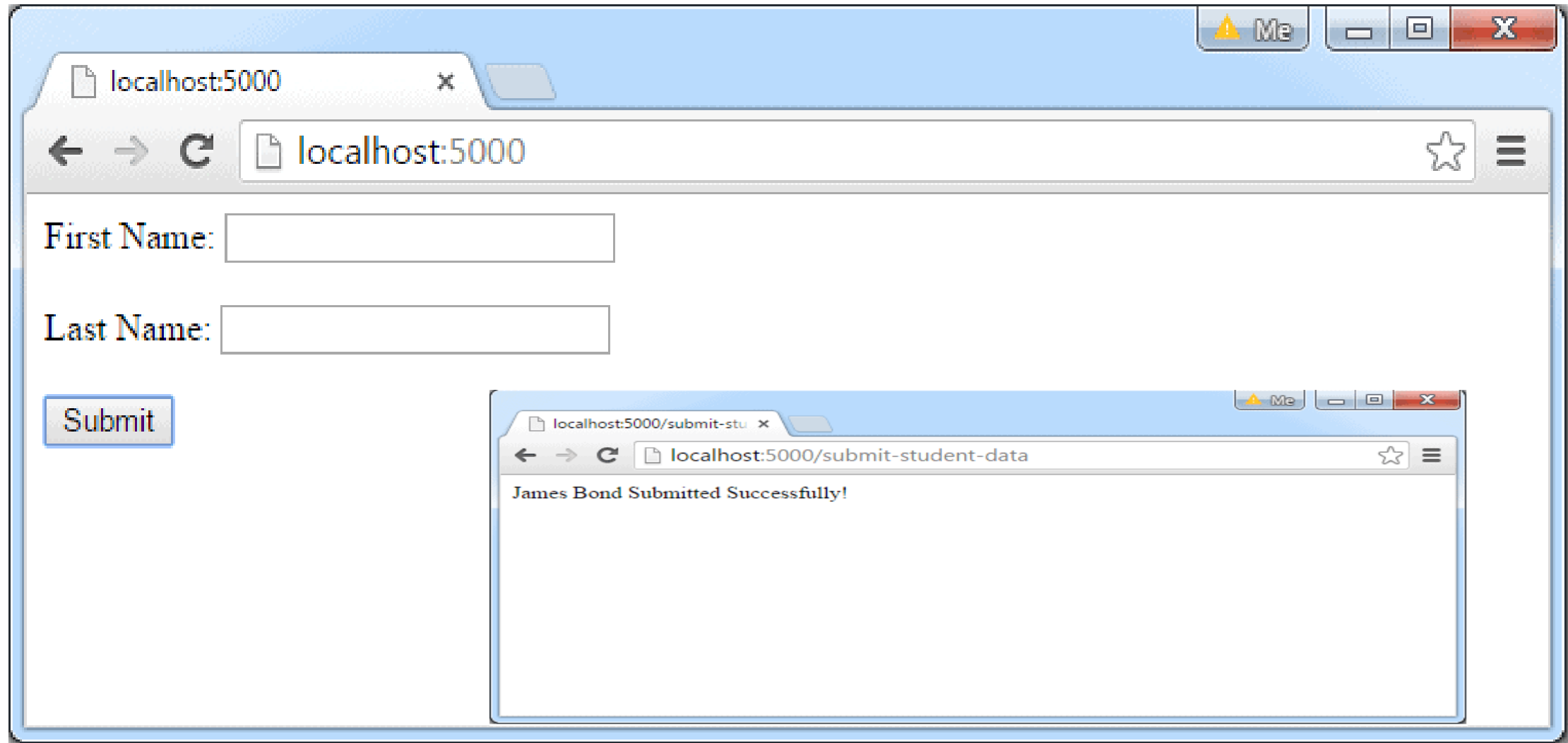
  res.send(name + ' Submitted Successfully!');
});
var server = app.listen(5000, function () {
  console.log('Node server is running..');
});
```

Handle POST Request

In the above 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 you can access it using `req.body.firstName` and `req.body.lastName`.

Now, run the above example using `node server.js` command, point your browser to `http://localhost:5000` and see the following result.

Handle POST Request



Express Calculator

```
//calculator.html
<!doctype html>
<html>
<head>

    <title> EXPRESSJS</title>

</head>

<body>

    <H1> EXPRESSJS CALCULATOR</H1>
    <form method="post" action="/submit-form">
        1st Value <input type="text" name="t1"><br>
        2nd Value <input type="text" name="t2"><br>
        <button type="submit">Calculate </button>
    </form>
</body>

</html>
```

Express Calculator

calculator.js

```
var express = require('express');
const bodyParser=require('body-parser');
var http = require('http');
var app = express();
app.use(bodyParser.urlencoded({extended:true}));
app.get('/calculator', function(req, res){
  res.sendFile(__dirname+"/calculator.html");
});
app.post("/submit-form", function(req, res) {
  console.log(req.body);
  //res.send('Calculation Ready Now!');
  let n1=Number(req.body.t1);
  let n2=Number(req.body.t2);
  let sum=n1+n2;
  res.send("Total Calculation:"+sum);
})
// Create a server
```

Middleware with Express

- ❑ Express is a Node module that gives us middleware. Execution flows through middleware, and a response drops out the bottom.
- ❑ Express is a node module that which provides us with middleware. Middleware is like a pipeline. Requests are passed through each middleware function in turn top to bottom. Each middleware function receives the request and response objects, and can modify them.

Middleware functions can do things like:

- Log a request
- Check authorization
- Serve a static file
- Inspect the URL, parse out URL parameters, and save them in the request
- Compile a SASS file and serve the result as CSS
- Serve a web page
- Serve an error or 404

EXPRESS

Middleware

```
1  var express = require('express')
2  var app = express()
3
4  var requestTime = function (req, res, next) {
5    req.requestTime = Date.now()
6    next()
7  }
8
9  app.use(requestTime)
10
11 app.get('/', function (req, res) {
12   var responseText = 'Hello World!<br>'
13   responseText += '<small>Requested at: ' + req.requestTime + '</small>'
14   res.send(responseText)
15 })
16
17 app.listen(3000); |
```

Uses the *requestTime* middleware function

EXPRESS

Router-level middleware

- Router-level middleware binds to an instance of `express.Router()`
- Works in the same way as application-level middleware

```
1 var app = express()
2 var router = express.Router()
3
4 // a middleware function with no mount path. This code is executed for every request to the router
5 router.use(function (req, res, next) {
6   console.log('Time:', Date.now())
7   next()
8 })
9
10 // a middleware sub-stack shows request info for any type of HTTP request to the /user/:id path
11 router.use('/user/:id', function (req, res, next) {
12   console.log('Request URL:', req.originalUrl)
13   next()
14 }, function (req, res, next) {
15   console.log('Request Type:', req.method)
16   next()
17 }) |
```

Router middleware i.e.
executed for every router
request

Router middleware i.e.
executed for given path

EXPRESS

Error-handling middleware

- Error-handling middleware always takes **four** arguments: (*err*, *req*, *res*, *next*)
- *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

```
1 app.use(function (err, req, res, next) {  
2   console.error(err.stack)  
3   res.status(500).send('Something broke!')  
4 }) |
```

Error Object

Request Object

Response Object

Next Object

Middleware

In express, middleware functions are the functions which have access to the request and response objects along with the next function present in the application's request-response cycle. These functions are capable of performing the below-listed tasks:

- Execution of any code
- Modify the request and the response objects.
- End applications request-response cycle.
- Call the next middleware present in the cycle.

Note that, in case, the current function doesn't terminate the request-response cycle then it must invoke the `next()` function in order to pass on the control to the next available middleware function. If not done, then the request will be left incomplete. Below I have listed down the majorly used middlewares in any Express.js application:

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

Middleware : Example

middleware.js

```
var express = require('express')
var app = express()
var requestDate = function (req, res, next) {
  req.requestDate = Date()
  next()
}
app.use(requestDate)
app.get('/', function (req, res) {
  var responseMsg = '<h2 style="font-family: Verdana; color: coral;">Hello Learners!!</h2>'
  responseMsg += '<small>Request genrated at: ' + req.requestDate + '</small>'
  res.send(responseMsg)
})
//PORT ENVIRONMENT VARIABLE
const port = process.env.PORT || 8080;
app.listen(port, () => console.log(`Listening on port ${port}..`));
```


Module nodemon

- ✓ nodemon is a tool that helps develop node.js based applications by automatically restarting the node application when file changes in the directory are detected.
- ✓ nodemon does **not** require *any* additional changes to your code or method of development.
- ✓ nodemon is a replacement wrapper for node. To use nodemon, replace the word node on the command line when executing your script.

npm install -g nodemon

- ✓ With a local installation, nodemon will not be available in your system path. Instead, the local installation of nodemon can be run by calling it from within an npm script (such as npm start) or using npx nodemon.

Manual restarting

- ✓ While nodemon is running, if you need to manually restart your application, instead of stopping and restart nodemon, you can type rs with a carriage return, and nodemon will restart your process.

Package.json

```
{  
  "name": "nodemon-import",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "type": "module", // now instead of writing const express = require('express') we can also write import express from 'express';  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1",  
    "start": "node index.js" // now we can use npm start  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "express": "^4.17.1"  
  }  
}
```

ExpressSnippet

❑ Install **ExpressSnippet** using Visual Studio Extension or just google

Express snippets

<https://marketplace.visualstudio.com/items?itemName=chris-noring.node-snippets>

To initiate just type out express commands and the rest will be autocompleted.

node-express

Will generate :

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

```
const app = express()
```

```
const port = 3000
```

```
app.get('/', (req, res) => res.send('Hello World!'))
```

```
app.listen(port, () => console.log(`Example app listening on port port!`))
```

ExpressSnippet

node-http-server

will generate

```
var http = require('http');  
http.createServer(function (request, response) {  
  response.writeHead(200, {'Content-Type': 'text/plain'});  
  response.end('Hello World');  
}).listen(8081);  
  
console.log('Server running at http://127.0.0.1:8081/');
```

ExpressSnippet

Command	Description
app.all	This method is like the standard app.METHOD() methods, except it matches all HTTP verbs.
app.disable	Sets the Boolean setting name to false, where name is one of the properties from the app settings table.
app.disabled	Returns true if the Boolean setting name is disabled (false), where name is one of the properties from the app settings table.
app.get	Express GET request
app.listen	Binds and listens for connections on the specified host and port. This method is identical to Node's http.Server.listen().
app.param	Add callback triggers to route parameters, where name is the name of the parameter or an array of them, and callback is the callback function.
app.patch	Routes HTTP PATCH request to the specifed path with the specified callback functions.

ExpressSnippet

Command	Description
<code>app.post</code>	Express POST request
<code>app.put</code>	Express PUT request
<code>app.delete</code>	Express DELETE request
<code>app.render</code>	Returns the rendered HTML of a view via the callback function. It accepts an optional parameter that is an object containing local variables for the view.
<code>app.route</code>	Returns an instance of a single route, which you can then use to handle HTTP verbs with optional middleware. Use <code>app.route()</code> to avoid duplicate route names (and thus typo errors).
<code>app.set</code>	Assigns setting name to value, where name is one of the properties from the app settings table.
<code>app.use</code>	Mounts the specified middleware function or functions at the specified path. If path is not specified, it defaults to <code>"/</code> ".
<code>res.send</code>	Express RESPONSE object

Templating In Express

- **What is a template engine**
- A template engine facilitates you to use static template files in your 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.
- **What Template Engines Should I Use?**
- There's a wide variety of template engines that work with Express. The default template engine found in Express is Jade, which is now known as Pug. However, the default Jade installed in Express is still using the old version.

Templating In Express

A template engine facilitates you to use static template files in your 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.

Following is a list of some popular template engines that work with Express.js:

- **Pug** (formerly known as jade , Default Template for Express)
- **mustache**
- dust
- atpl
- ejs

Templating In Express

Template engine makes you able to use static template files in your application. To render template files you 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')`.

Step To Create Pug Teampplate Engine

How To Use Pug

To install Pug and use it in our Express app, we have to use npm to install it first:

Steps :

Create a folder for the pug based project

1. `md expresspug`
 2. `npm init`
 3. `npm install express - - save`
 4. `npm install nodemon -g`
 5. `npm install pug - -save`
- create the folder views
 - Create the file template1.pug inside the views

Step To Create Pug Teamplate Engine

template1.pug (file inside the views folder)

```
html
```

```
  head
```

```
    title=title
```

```
  body
```

```
    h1=message
```

Step To Create Pug Teampplate Engine

Create server.js (take support of node-express snippet)

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

//app.get('/', (req, res) => res.send('Hello World!'))
app.set('view engine', 'pug')
app.get('/', function (req, res) {
  res.render('template1', { title: ' Exploring the Pug', message: 'pug Template' })
})
app.listen(port, () => console.log(`Example app listening on port port!{port}`))
```

Step To Create Pug Teampplate Engine

- Run the program **node server.js**
- See the Output with the Pug Template
- (Message will be h1 & page title will be with Hey)

PUG : Example

Indentation is very crucial in Pug, it makes it easy for Pug template engine to know which part of the code is nested. In the code above note how I added `h1` to a paragraph. When code above is compiled, this is what it translates to:

```
<!DOCTYPE html>
<html>
  <head>
    <title>
      Exploring the Pug
    </title>
  </head>
  <body>
    <h1 >
      pug template
    </h1>
  </body>
</html>
```

PUG : Example

- ✓ Pug is a template engine for Node.js. Pug uses whitespaces and indentation as the part of the syntax. Its syntax is aesy to learn.
- ✓ Pug template must be written inside .pug file and all .pug files must be put inside views folder in the root folder of Node.js application.
- ✓ **Note:** By default Express.js searches all the views in the views folder under the root folder. you can also set to another folder using views property in express.

PUG : Example

The pug template engine takes the input in a simple way and produces the output in HTML.
See how it renders HTML:

Simple input:

```
doctype html
html
  head
    title A simple pug example
  body
    h1 This page is produced by pug template engine
    p some paragraph here..
```

Output produced by pug template:

```
<!DOCTYPE html>
<html>
  <head>
    <title>A simple pug example</title>
  </head>
  <body>
    <h1>This page is produced by pug template engine</h1>
    <p>some paragraph here..</p>
  </body>
</html>
```


Scaffolding In Express

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.

Express js Scaffolding

Scaffolding is used to create a skeleton structure for 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.

A ***template engine*** enables you to use static template files in your application. At runtime, the template engine replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client. This approach makes it easier to design an HTML page.

Scaffolding In Express : Steps

- ❑ `npm init`
- ❑ `npm install express --save`
- ❑ `npm -g nodemon`
- ❑ `npm install -g express-generator`
- ❑ `npm install`
- ❑ `express`

Scaffolding In Express with pug

- ❑ `npm init`
- ❑ `npm install express --save`
- ❑ `npm -g nodemon`
- ❑ `npm install -g express-generator`
- ❑ `npm install pug --save`
- ❑ `npm install`
- ❑ `Express --view=pug`

Scaffolding In Express : Steps

- ✓ views, the directory where the template files are located. Eg: `app.set('views', './views')`. This defaults to the views directory in the application root directory.
- ✓ view engine, the template engine to use. For example, to use the Pug template engine: `app.set('view engine', 'pug')`.
- ✓ After the view engine is set, you don't have to specify the engine or load the template engine module in your app; Express loads the module internally, as shown below (for the above example).
- ✓ `app.set('view engine', 'pug')`

Scaffolding In Express : Steps

- ✓ Create a Pug template file named index.pug in the views directory, with the following content:

```
html
  head
    title= title
  body
    h1= message
```

- ✓ Then create a route to render the index.pug file. If the view engine property is not set, you must specify the extension of the view file. Otherwise, you can omit it.

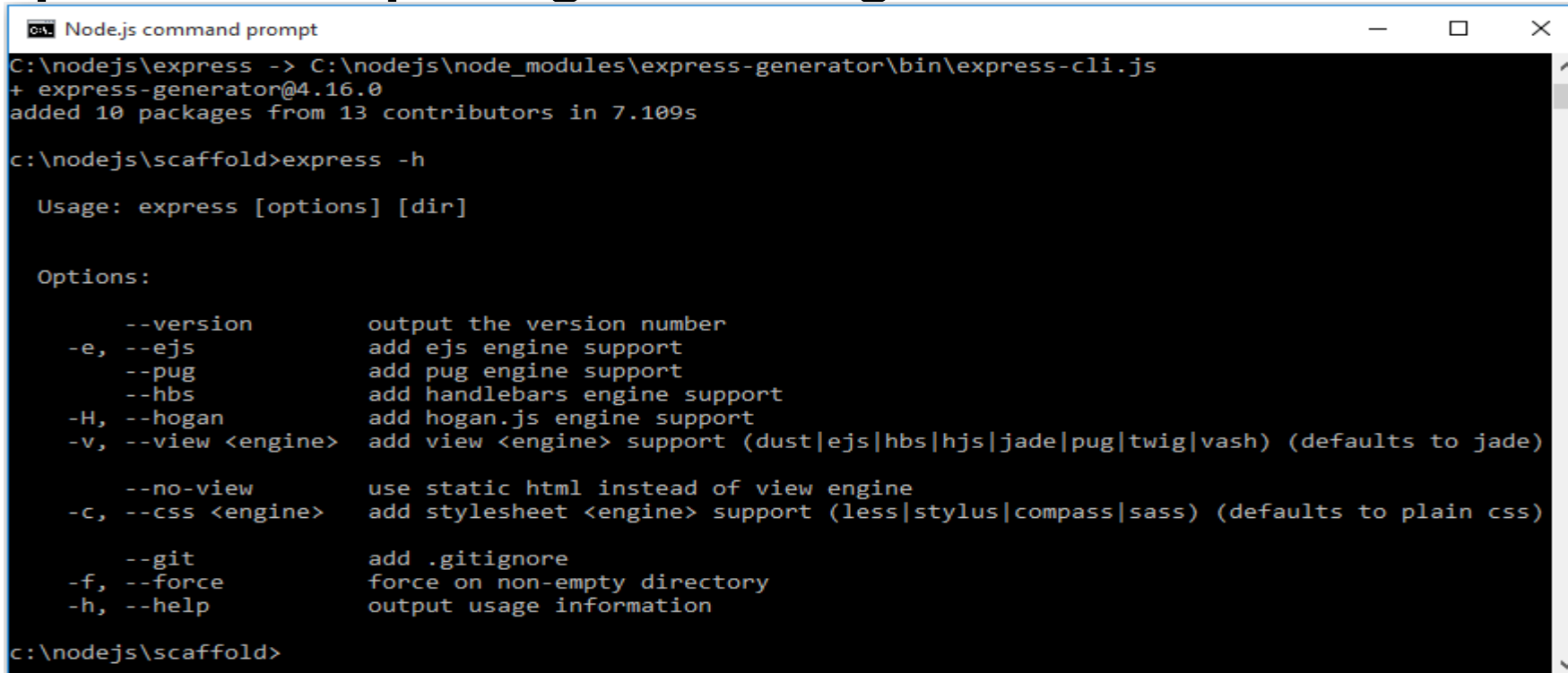
```
app.get('/', function (req, res) {
  res.render('index', { title: 'Hey', message: 'Hello there!' })
})
```

- ✓ When you make a request to the home page, the index.pug file will be rendered as HTML.

Express Generator

This package contains an express command line tool, run the following command to install express-generator package.

npm install express-generator -g



```
Node.js command prompt
C:\nodejs\express -> C:\nodejs\node_modules\express-generator\bin\express-cli.js
+ express-generator@4.16.0
added 10 packages from 13 contributors in 7.109s

c:\nodejs\scaffold>express -h

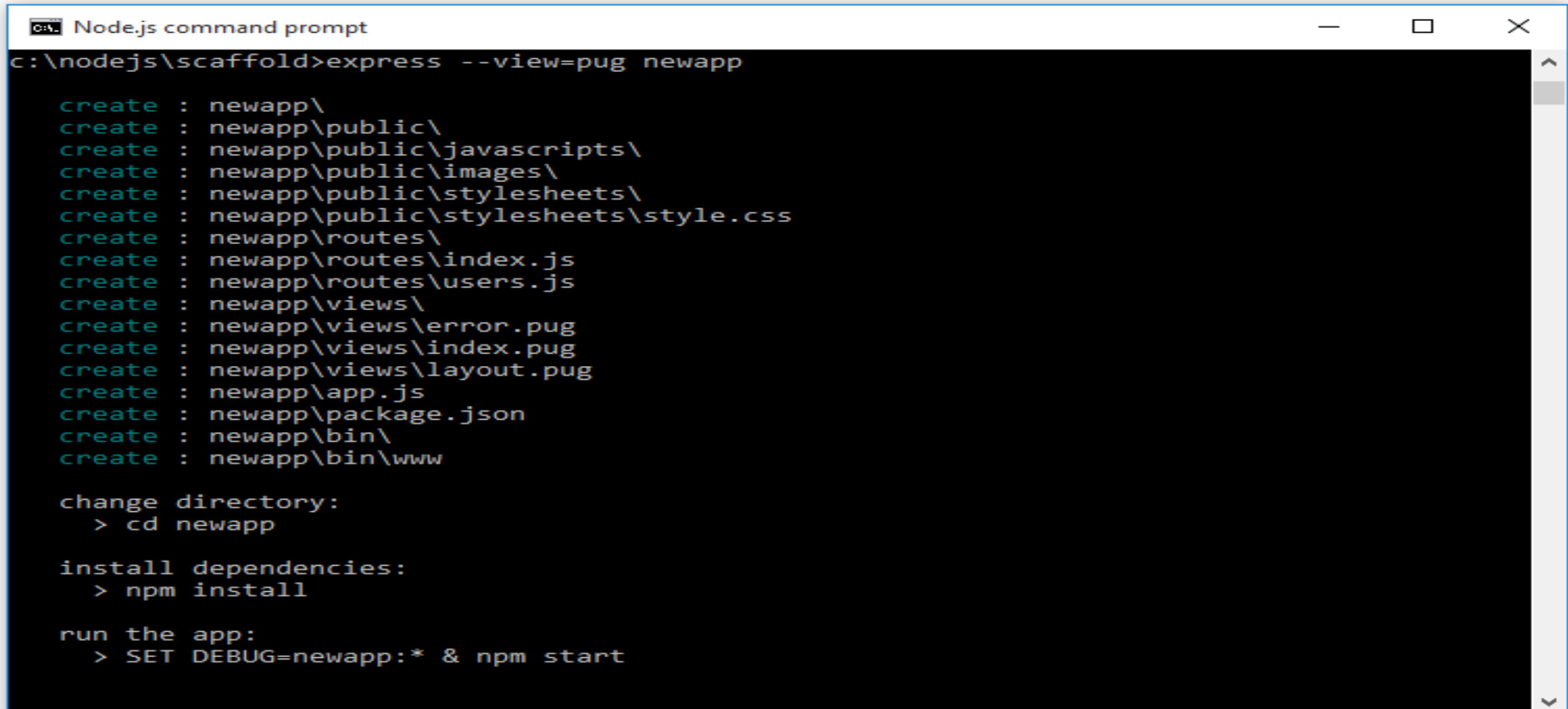
Usage: express [options] [dir]

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|twig|vash) (defaults to jade)
  --no-view         use static html instead of view engine
  -c, --css <engine> add stylesheet <engine> support (less|stylus|compass|sass) (defaults to plain css)
  --git             add .gitignore
  -f, --force        force on non-empty directory
  -h, --help         output usage information

c:\nodejs\scaffold>
```

Express Generator

Suppose we want to use a pug view engine to create new applications.

A screenshot of a Windows command prompt window titled "Node.js command prompt". The window shows the execution of the command "express --view=pug newapp" in the directory "c:\nodejs\scaffold". The output lists the files and directories created by the Express generator using the pug view engine. The files and directories are: newapp\, newapp\public\, newapp\public\javascripts\, newapp\public\images\, newapp\public\stylesheets\, newapp\public\stylesheets\style.css, newapp\routes\, newapp\routes\index.js, newapp\routes\users.js, newapp\views\, newapp\views\error.pug, newapp\views\index.pug, newapp\views\layout.pug, newapp\app.js, newapp\package.json, newapp\bin\, and newapp\bin\www. Below the file list, the prompt provides instructions on how to change the directory, install dependencies, and run the application.

```
C:\nodejs\scaffold>express --view=pug newapp

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
```

Express Generator

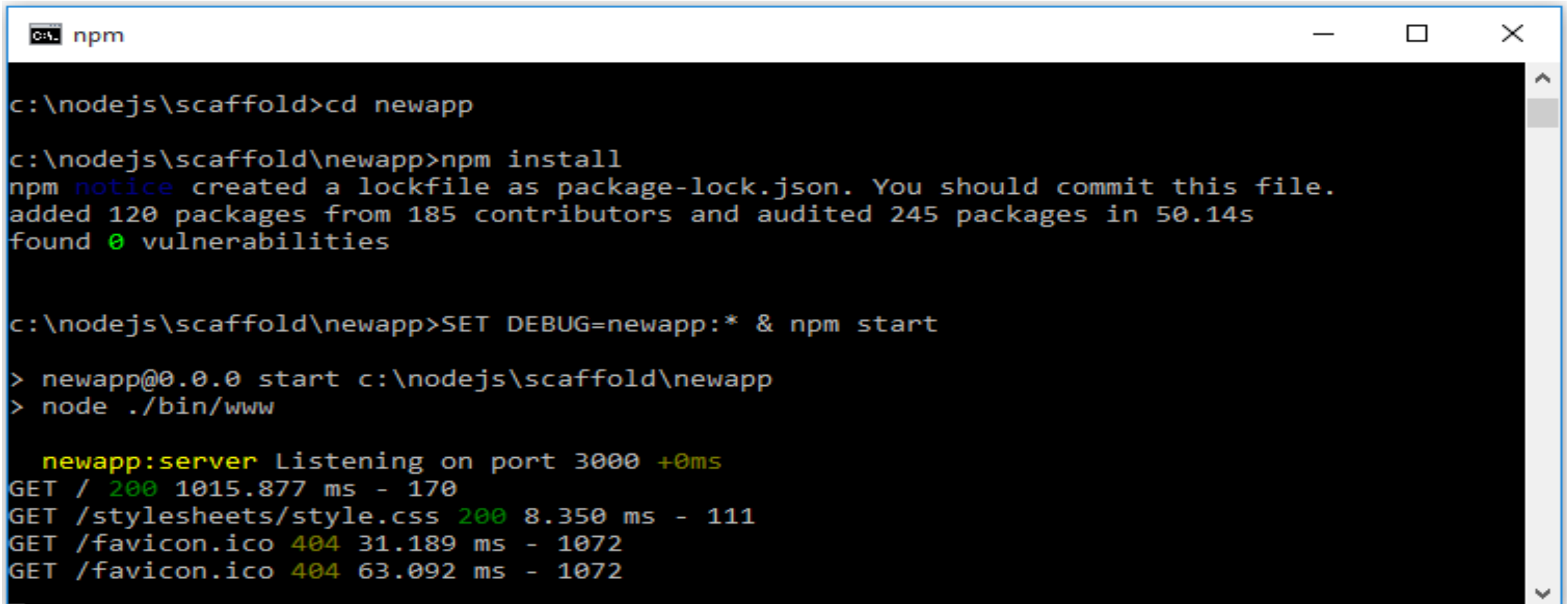
```
.
├── app.js
├── bin
│   └── www
├── package.json
├── public
│   ├── images
│   ├── javascripts
│   └── stylesheets
│       └── style.css
├── routes
│   ├── index.js
│   └── users.js
└── views
    ├── error.pug
    ├── index.pug
    └── layout.pug
```

7 directories, 9 files

Express Generator

Then, also commands these following-

```
>cd newapp>npm install  
>SET DEBUG=newapp:* & npm start
```



```
C:\> npm  
c:\nodejs\scaffold>cd newapp  
c:\nodejs\scaffold\newapp>npm install  
npm notice created a lockfile as package-lock.json. You should commit this file.  
added 120 packages from 185 contributors and audited 245 packages in 50.14s  
found 0 vulnerabilities  
  
c:\nodejs\scaffold\newapp>SET DEBUG=newapp:* & npm start  
  
> newapp@0.0.0 start c:\nodejs\scaffold\newapp  
> node ./bin/www  
  
newapp:server Listening on port 3000 +0ms  
GET / 200 1015.877 ms - 170  
GET /stylesheets/style.css 200 8.350 ms - 111  
GET /favicon.ico 404 31.189 ms - 1072  
GET /favicon.ico 404 63.092 ms - 1072
```

Project Structure : Explanation

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.

```
Parikshit@DESKTOP-F3EF709 MINGW64 ~/Desktop/GeeksForGeeks
$ express GeeksForGeeksSampleApp

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

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

change directory:
$ cd GeeksForGeeksSampleApp

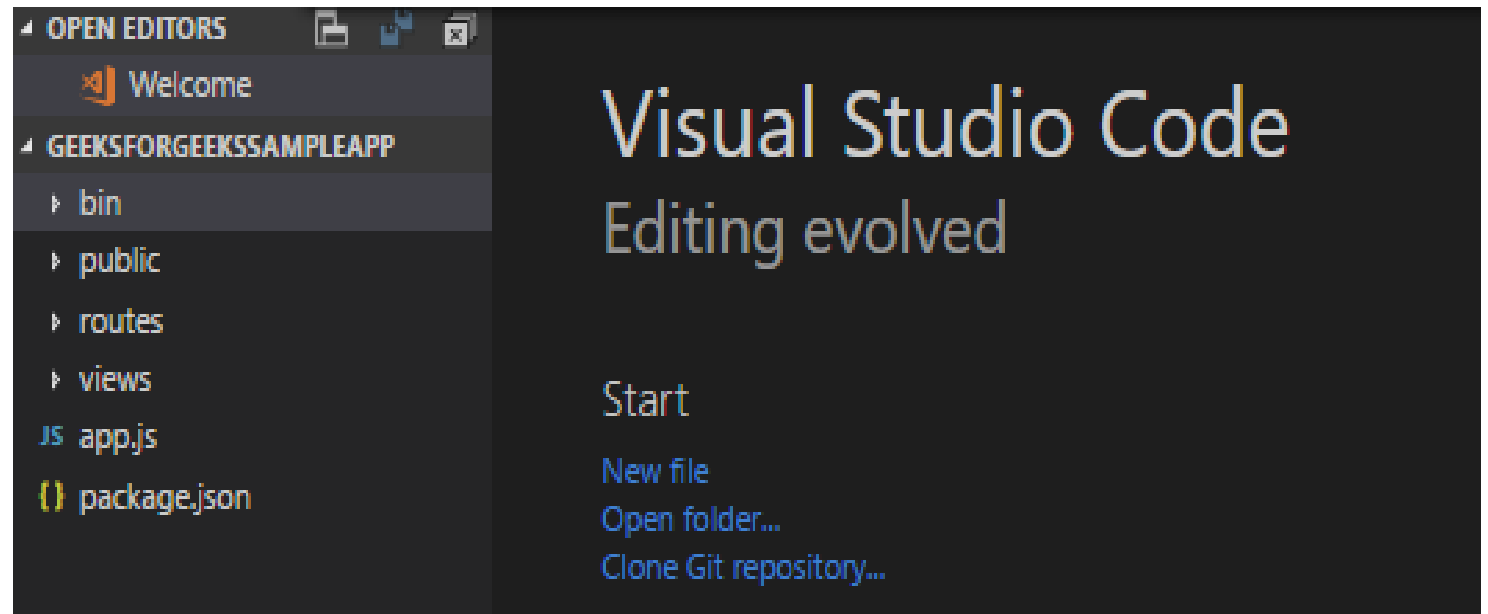
install dependencies:
$ npm install

run the app:
$ DEBUG=geeksforgEEKSSampleApp:* npm start
```

Project Structure : Explanation

Project Folder:

The project folder is constituted of various folders/files which can be seen in image. 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.



Project Structure : Explanation

Explanation:

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

Project Structure : Explanation

Note: The extension of the files at the time of writing is **.jade**. Change these file extensions to **.pug** as the jade project has changed to pug.

In the app.js file, change the following code:

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

to :

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

This will change the view engine to pug.

Project Structure : Explanation

5. 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.

6. 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.

Project Structure : Explanation

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`

Creating Service in ExpressJS

Building a REST API with Node and Express

What is a REST API?

REpresentation **S**tate **T**ransfer (REST) is a set of methods in which HTTP clients can request information from the server via the HTTP protocol.

Creating Service in ExpressJS

HTTP Request Types

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

- **POST**: Used to submit data, typically used to *create* new entities or edit already existing entities
- **GET**: Used to request data from the server, typically used to *read* data
- **PUT**: Used to completely replace the resource with the submitted resource, typically used to *update* data
- **DELETE**: Used to *delete* an entity from the server

Creating Service in ExpressJS

Steps:

- ☐ Create a folder for the service
- ☐ Create the package.json `//npm init`
- ☐ `npm install express --save`
- ☐ `Npm install -g nodemon`

Creating Service in ExpressJS

Creating the app.

```
const express = require('express')
const app = express()
const port = 3000
app.get('/', (req, res) => res.send('Hello World!'))
app.get("/url", (req, res, next) => {
  res.json(["Tony","Lisa","Michael","Ginger","Food"]);
});
app.listen(port, () => console.log(`Example app listening on port port!`))

// Test the Service on Testing API Tool postman
```

Creating Service in ExpressJS

- **Setting request handlers.**
- A server receives requests, processes them and returns a response. So you need to use routes to handle this requests. The requests have three major types, a GET request that get's data, a POST request that sends data securely, a PUT request that updates data and a DELETE request that deletes data.

Creating Service in ExpressJS

Let's create a simple GET request that returns a list of users. Under `var app = express()`, write down the following code.

```
app.get("/url", (req, res, next) => {  
  res.json(["Tony", "Lisa", "Michael", "Ginger", "Food"]);  
});
```

This simple function makes the express app to use the url handle “/url” to trigger the callback that follows it. The callback accepts three parameters, req is the request body and carries information about the request. The res is the response body and is used to handle response functions like `.json()` to return json data.

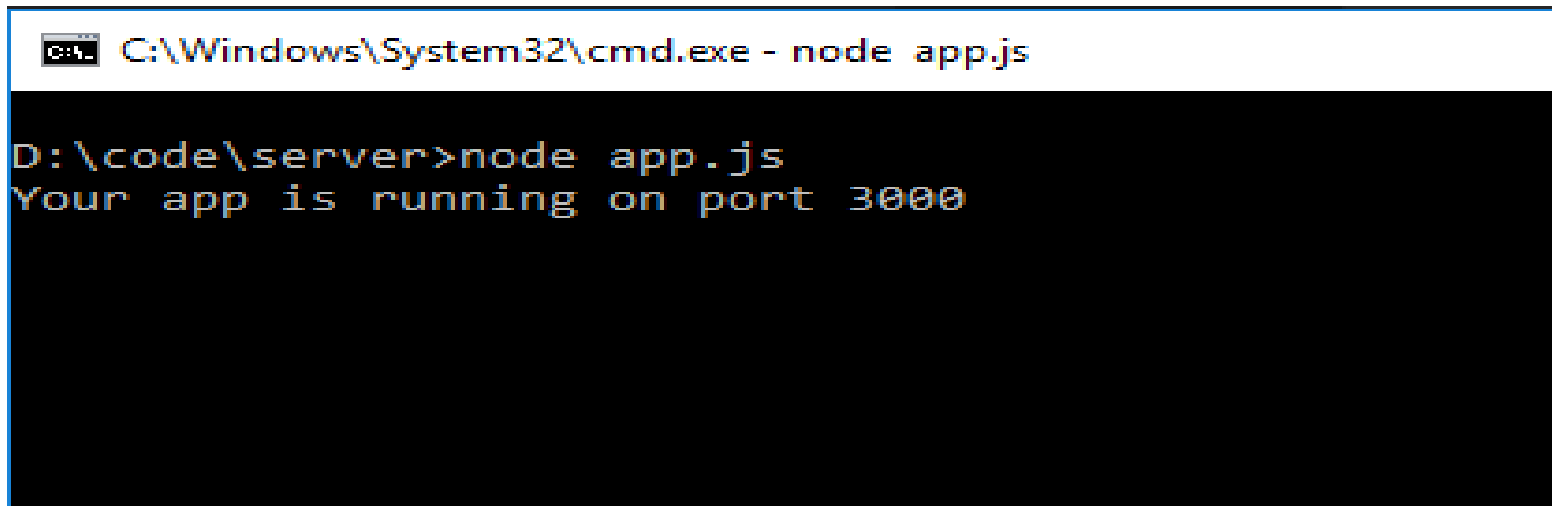
Creating Service in ExpressJS

Running your app.

To run your app, use the command below.

`node app.js`

This is what your cmd should look like after running this command.

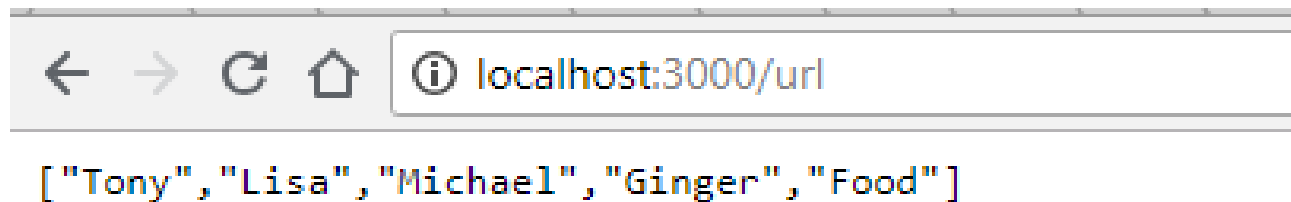


```
C:\Windows\System32\cmd.exe - node app.js

D:\code\server>node app.js
Your app is running on port 3000
```

Creating Service in ExpressJS

This means our app is now successfully running on port 3000. To view our data, open up your browser and enter <http://localhost:3000/url>. You'll expect to see something like this on your browser.



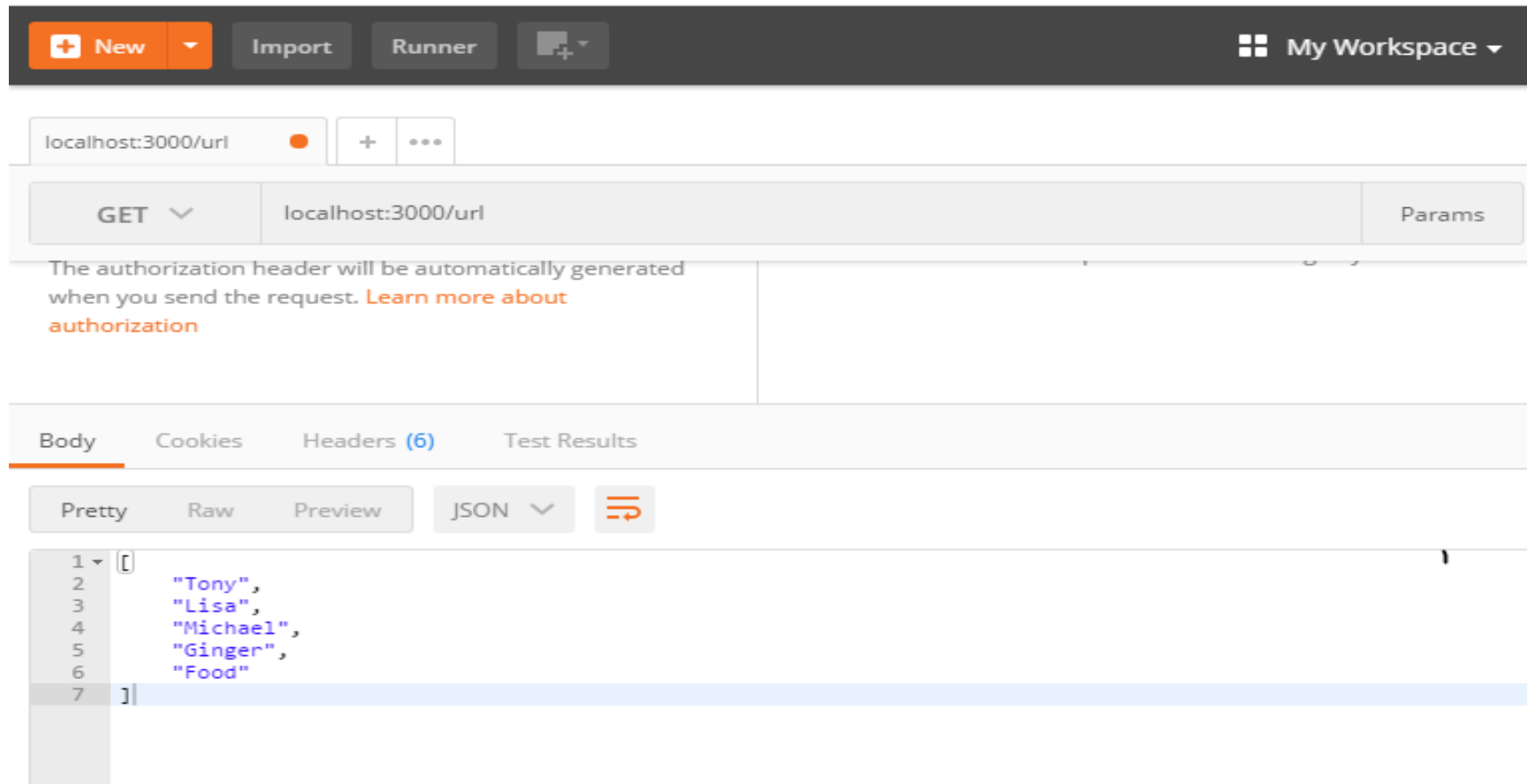
Creating Service in ExpressJS

How it all fits as a REST based API.

- ❑ You might be wondering where the REST attribute comes in. REST stands for Representational State Transfer. This means there is no state between the client and the server. There are no webpages served to be parsed, just data. And this gives you all the freedom you need. All you need to do is write some logic on a specific URL that connects to a database, uses its logic to process the data and return it in JSON format. Your client can now be an Android app made in Java, or a Windows desktop app made in C# or Java project.
- ❑ This is the whole point of using REST, it makes the connection stateless therefore any client that utilizes the HTTP protocol can access this data. Now you can iterate through the data and display it anywhere you want.

Creating Service in ExpressJS

Below is an image of my Postman app querying the same server



Creating Service in ExpressJS

eservice2.js

```
const express = require('express')
const app = express()
const port = 3000
app.get('/', (req, res) => res.send('Hello World!'))
const games = [{
  id: 1,
  title: 'Mario'
},
{
  id: 2,
  title: 'Zelda'
},
{
  id: 3,
  title: 'Donkey Kong'
}]
```

Creating Service in ExpressJS

eservice2.js

```
app.get('/api/games', (req, res) => {  
  res.send/games);  
});  
app.get("/url", (req, res, next) => {  
  res.json(["Tony", "Lisa", "Michael", "Ginger", "Food"]);  
});  
app.listen(port, () => console.log(`Example app listening on port port!`))
```

ExpressJs – Error Handling & Debugging

- ❑ Error Handling refers to how Express catches and processes errors that occur both synchronously and asynchronously. Express comes with a default error handler so you don't need to write your own to get started.

Catching Errors

It's important to ensure that Express catches all errors that occur while running route handlers and middleware.

Errors that occur in synchronous code inside route handlers and middleware require no extra work. If synchronous code throws an error, then Express will catch and process it. For example:

```
app.get('/', function (req, res) {  
  throw new Error('BROKEN') // Express will catch this on its own.  
})
```

ExpressJs – Error Handling & Debugging

Error handling in Express is done using middleware. But this middleware has special properties. The error handling middleware are defined in the same way as other middleware functions, except that error-handling functions MUST have four arguments instead of three – err, req, res, next. For example, to send a response on any error, we can use –

```
app.use(function(err, req, res, next) {  
  console.error(err.stack);  
  res.status(500).send('Something broke!');  
});
```

ExpressJs – Error Handling & Debugging

For error handling, we have the **next(err)** function. A call to this function skips all middleware and matches us to the next error handler for that route. Let us understand this through an example.

```
var express = require('express');
var app = express();
app.get('/', function(req, res){
  //Create an error and pass it to the next function
  var err = new Error("Something went wrong");
  next(err);
});
app.use(function(err, req, res, next) {
  res.status(500);
  res.send("Oops, something went wrong.")
});
app.listen(3000);
```

ExpressJs – Error Handling & Debugging

Catching Errors in Express

If synchronous code having route handlers and middleware throws any error, then without any effort and extra work, Express solves it by catching and processing the error without requiring any of our consent. Have a look at the below code –

```
app.get('/', function (req, res) {  
  
  // Express catches this on its own  
  throw new Error('Died')  
})
```

ExpressJs – Error Handling & Debugging

If a route handler and middleware invokes asynchronous function which in turn produces some errors, then we have to explicitly pass the error to the `next()` function, where Express will catch and process them. The following illustration will help you to understand

```
app.get('/', function (req, res, next) {  
  fs.readFile('/file-is-not-available',  
    function (err, data) {  
    if (err) {  
      // Passing errors to  
      // Express explicitly  
      next(err)  
    } else {  
      res.send(data)  
    }  
  })  
})
```


ExpressJs –Debugging

Express uses the Debug module to internally log information about route matching, middleware functions, application mode, etc.

To see all internal logs used in Express, set the DEBUG environment variable to Express:* when starting the app –

DEBUG = express:* node index.js

`SET DEBUG=newapp:* & npm start`

Configure package.json for Debugging

```
"scripts": {  
  "start": "set DEBUG=abc:* & node ./bin/www"  
}
```

we just need to put the set command in the package.json file itself, In that way you can store as many as variable you want like port also.

Note : You can also Debug the ExpressJs Code inside the Visual Studio Code

Cookies

Cookies

Cookies in Express.js are the small files of information which are stored within a user's computer. They hold a decent amount of data specific to a particular user and the websites he visits. Each time a client loads a website on his browser, the browser will implicitly send the locally stored data back to the website/server, in order to recognize the user. A cookie generally consists of the following:

Name

Value

Zero or more attributes in key-value format. These attributes can include cookie's expiration, domain, and flags, etc.

Cookies

Cookies

Now in order to use cookies in express, first you need to install the cookie-parser middleware through [npm](#) into the ***node_modules*** folder that is already present in your project folder. Below is the code to perform the same:

```
npm install cookie-parser
```

Once done, the next step is to import the cookie-parser into your application. Below is code to do the same:

Cookies

```
var express = require('express');
var cookie_parser = require('cookie-parser');
var app = express();
app.use(cookie_parser());
//Setting up a Cookie
app.get('/setcookie',function(req, res){
  res.cookie('cookie_name', 'Express_Demo');
  res.cookie('organization', 'ABC');
  res.cookie('name', 'Swatee');

  res.status(200).send('<h4 style="font-family: Tahoma; color: coral; text-align: center;">Setting up the Cookie</h4>');
});

//Checking cookie is set or not
app.get('/getcookie', function(req, res) {
  res.status(200).send(req.cookies);
});

//Welcome Message
app.get('/', function (req, res) {
  res.status(200).send('<h2 style="font-family: cursive; color: lightseagreen; text-align: center; ">Welcome to Express!</h2>');
});
//PORT ENVIRONMENT VARIABLE
const port = process.env.PORT || 8080;
app.listen(port, () => console.log(`Listening on port ${port}..`));
```

Cookies

Now check the URL localhost:8080

Result Will Be : **Welcome to Express!**

Now check the url localhost:8080/setcookie

Result will be :

The screenshot shows a web browser with the address bar displaying `localhost:8080/setcookie`. The **Application** tab is selected, showing the **Setting up the Cookie** section. The left sidebar lists various storage and background services, with **Cookies** expanded and **http://localhost:8080** selected. The main area displays a table of cookies:

Name	Value	Domain	Path	Expires / Max-A...	Size	HttpOnly	Secure	S...	Priority
name	Swatee	localhost	/	Session	10				Medium
organization	ABC	localhost	/	Session	15				Medium
cookie_name	Express_Demo	localhost	/	Session	23				Medium

Below the table, the cookie **Express_Demo** is expanded, showing its details. The bottom of the browser shows the Windows taskbar with the search bar and various application icons.

Cookies

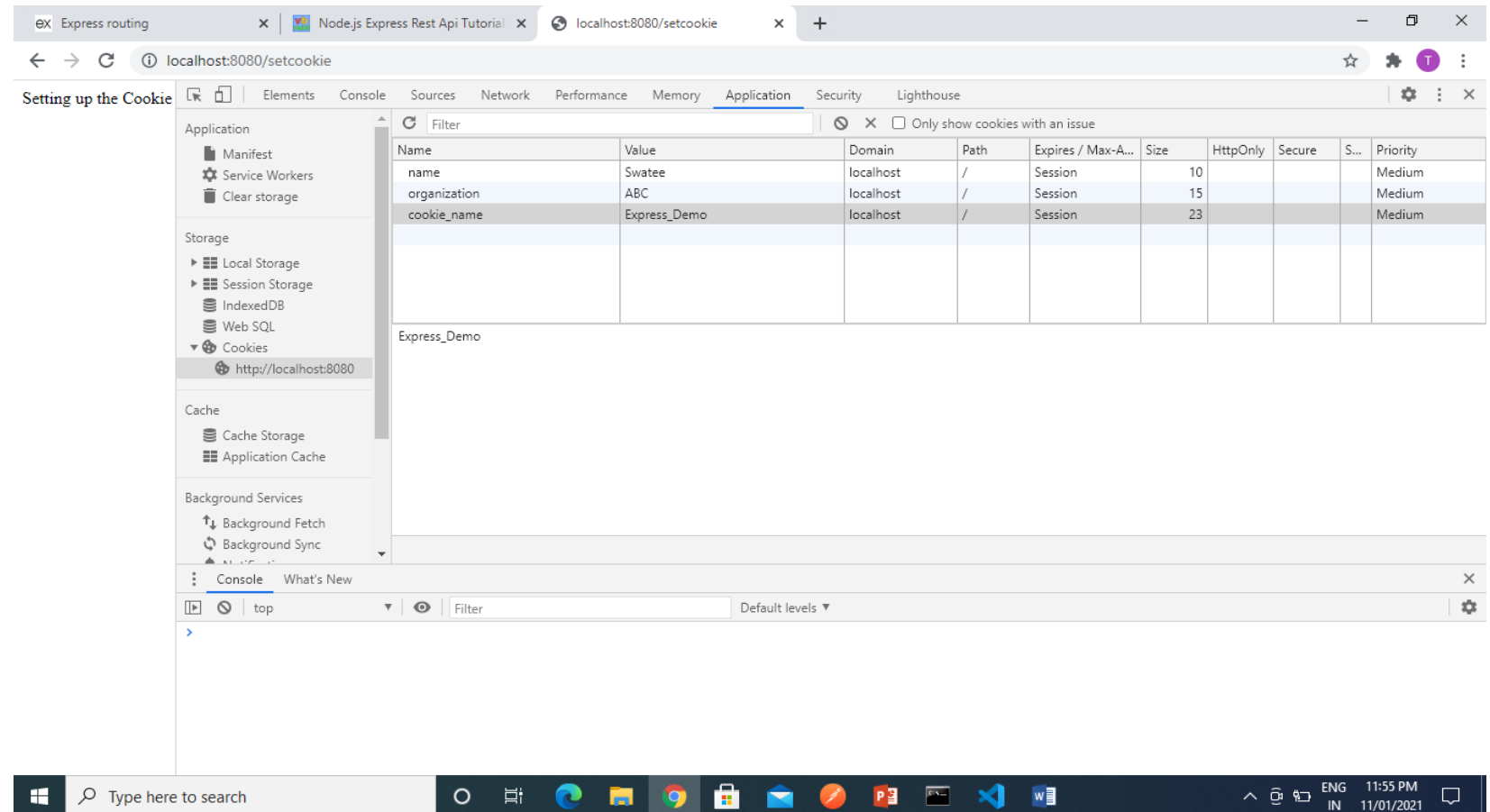
Now check the URL localhost:8080

Result Will Be : **Welcome to Express!**

Now check the url localhost:8080/getcookie

Result will be :

```
{"cookie_name":"Express_Demo",  
"organization":"ABC","name":"Swatee"}
```



The screenshot shows a web browser with the address bar displaying `localhost:8080/setcookie`. The **Application** tab is selected, showing a list of cookies for the domain `http://localhost:8080`. The cookies are:

Name	Value	Domain	Path	Expires / Max-A...	Size	HttpOnly	Secure	S...	Priority
name	Swatee	localhost	/	Session	10				Medium
organization	ABC	localhost	/	Session	15				Medium
cookie_name	Express_Demo	localhost	/	Session	23				Medium

Below the table, the cookie `Express_Demo` is expanded, showing its value as `Express_Demo`. The bottom of the screenshot shows the Windows taskbar with the time `11:55 PM` and date `11/01/2021`.