|  |  |  |
| --- | --- | --- |
| ExpressJS | December 16  2021 | |
|  | |  |

# ExpressJS - Overview

## What is Express?

Express provides a minimal interface to build our applications. It provides us the tools that are required to build our app. It is flexible as there are numerous modules available on **npm**, which can be directly plugged into Express.

Express was developed by **TJ Holowaychuk** and is maintained by the [Node.js](https://nodejs.org/en/) foundation and numerous open source contributors.

# ExpressJS - Environment

**Installing**

Assuming you’ve already installed [Node.js](https://nodejs.org/), create a directory to hold your application, and make that your working directory.

$ mkdir myapp

$ cd myapp

Use the npm init command to create a package.json file for your application. For more information on how package.json works, see [Specifics of npm’s package.json handling](https://docs.npmjs.com/files/package.json).

$ npm init

This command prompts you for a number of things, such as the name and version of your application. For now, you can simply hit RETURN to accept the defaults for most of them, with the following exception:

entry point: (index.js)

Enter app.js, or whatever you want the name of the main file to be. If you want it to be index.js, hit RETURN to accept the suggested default file name.

Now install Express in the myapp directory and save it in the dependencies list. For example:

$ npm install express --save

To install Express temporarily and not add it to the dependencies list:

$ npm install express --no-save

# Hello world example

const express = require('express')

const app = express()

const port = 3000

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

res.send('Hello World!')

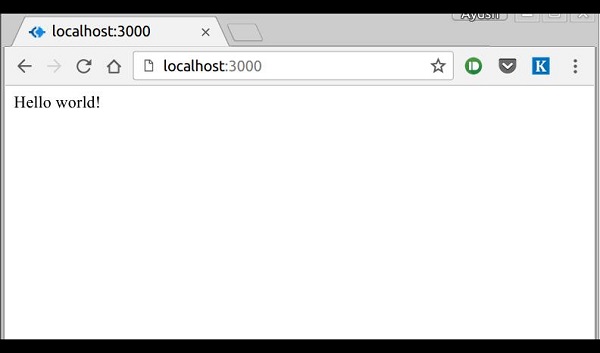
})

app.listen(port, () => {

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

}

**This app starts a server and listens on port 3000 for connections. The app responds with “Hello World!”**



## How the App Works?

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

### app.get(route, callback)

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

### res.send()

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

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

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

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

# Basic 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)**

**Where:**

* app is an instance of express.
* METHOD is an [HTTP request method](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods), in lowercase.
* PATH is a path on the server.
* HANDLER is the function executed when the route is matched.

The following examples illustrate defining simple routes.

Respond with Hello World! on the homepage:

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

res.send('Hello World!')

})

Respond to POST request on the root route (/), the application’s home page:

app.post('/', function (req, res) {

res.send('Got a POST request')

})

Respond to a PUT request to the /user route:

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

res.send('Got a PUT request at /user')

})

Respond to a DELETE request to the /user route:

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

res.send('Got a DELETE request at /user')

})

# ExpressJS - HTTP Methods

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

|  |  |
| --- | --- |
| **S.No.** | **Method & Description** |
| 1 | **GET**  The GET method requests a representation of the specified resource. Requests using GET should only retrieve data and should have no other effect. |
| 2 | **POST**  The POST method requests that the server accept the data enclosed in the request as a new object/entity of the resource identified by the URI. |
| 3 | **PUT**  The PUT method requests that the server accept the data enclosed in the request as a modification to existing object identified by the URI. If it does not exist then the PUT method should create one. |
| 4 | **DELETE**  The DELETE method requests that the server delete the specified resource. |

# ExpressJS - URL Building

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

Here is an example of a dynamic route −

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

**var app = express();**

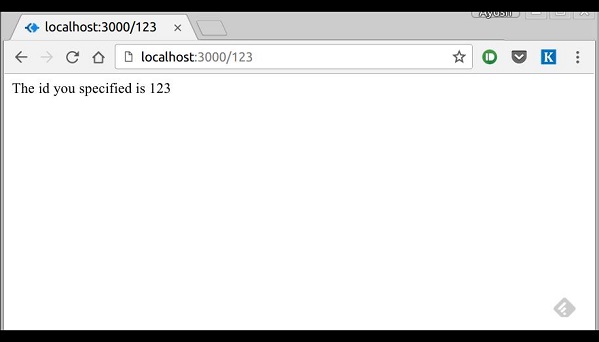
**app.get('/:id', function(req, res){**

**res.send('The id you specified is ' + req.params.id);**

**});**

**app.listen(3000);**

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



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

var express = require('express');

var app = express();

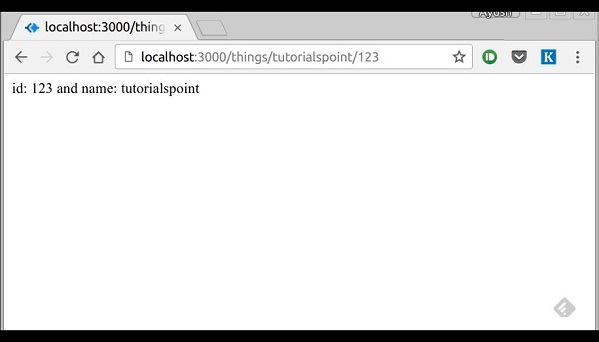
app.get('/things/:name/:id', function(req, res) {

res.send('id: ' + req.params.id + ' and name: ' + req.params.name);

});

app.listen(3000);

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



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

## Pattern Matched Routes

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

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

**var app = express();**

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

**res.send('id: ' + req.params.id);**

**});**

**app.listen(3000);**

Note that this will **only** match the requests that have a 5-digit long **id**. You can use more complex regexes to match/validate your routes. If none of your routes match the request, you'll get a ***"Cannot GET <your-request-route>"*** message as response. This message be replaced by a 404 not found page using this simple route −

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

**var app = express();**

**//Other routes here**

**app.get('\*', function(req, res){**

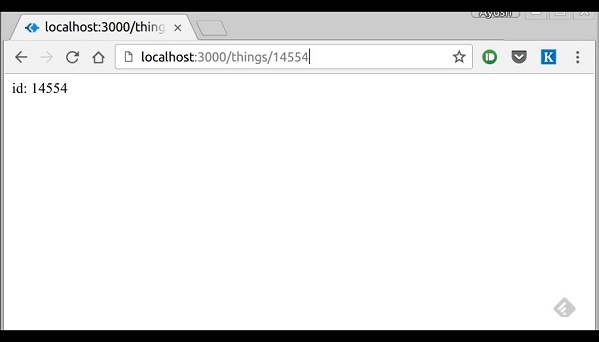
**res.send('Sorry, this is an invalid URL.');**

**});**

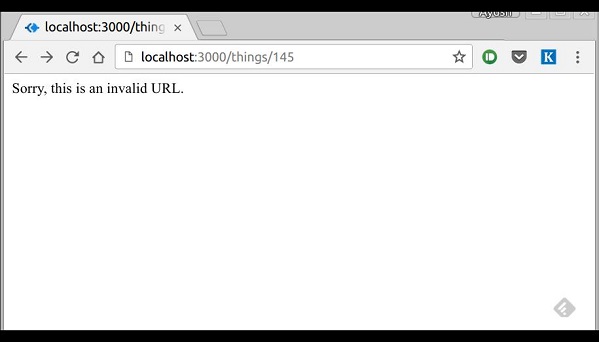
**app.listen(3000);**

**Important** − This should be placed after all your routes, as Express matches routes from start to end of the **index.js** file, including the external routers you required.

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



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



# ExpressJS - Middleware

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

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

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

**var app = express();**

**//Simple request time logger**

**app.use(function(req, res, next){**

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

**//This function call is very important. It tells that more processing is**

**//required for the current request and is in the next middleware**

**function route handler.**

**next();**

**});**

**app.listen(3000);**

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

A new request received at 1467267512545

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

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

**var app = express();**

**//Middleware function to log request protocol**

**app.use('/things', function(req, res, next){**

**console.log("A request for things received at " + Date.now());**

**next();**

**});**

**// Route handler that sends the response**

**app.get('/things', function(req, res){**

**res.send('Things');**

**});**

**app.listen(3000);**

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

### Order of Middleware Calls

One of the most important things about middleware in Express is the order in which they are written/included in your file; the order in which they are executed, given that the route matches also needs to be considered.

For example, in the following code snippet, the first function executes first, then the route handler and then the end function. This example summarizes how to use middleware before and after route handler; also how a route handler can be used as a middleware itself.

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

**var app = express();**

**//First middleware before response is sent**

**app.use(function(req, res, next){**

**console.log("Start");**

**next();**

**});**

**//Route handler**

**app.get('/', function(req, res, next){**

**res.send("Middle");**

**next();**

**});**

**app.use('/', function(req, res){**

**console.log('End');**

**});**

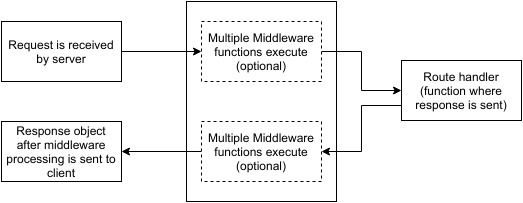
**app.listen(3000);**

When we visit '/' after running this code, we receive the response as **Middle** and on our console −

Start

End

The following diagram summarizes what we have learnt about middleware −



# ExpressJS - Templating

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.

## HandleBars Templating in ExpressJS

Handlebars.js is a templating engine similar to the ejs module in node.js, but more powerful and simple to use. It ensures minimum templating and is a logicless engine that keeps the view and the code separated. It can be used with express as the hbs module, available through npm. HandleBars can be used to render web pages to the client side from data on the server-side.

**Command to install hbs module:**

**npm i hbs**

To use handlebars in express, we need to store HTML code into a .hbs extension in the ‘views’ folder in the source directory as hbs looks for the pages in the views folder.

The first thing we need to do in index.js file is to require the hbs module

var express = require('express')

var hbs = require('hbs')

var app = express()

**Now, we need to change the default view engine.**

app.set('view engine', 'hbs')

**In case the views directory is undesirable, you can change the viewpath by the following command:**

app.set('views', <pathname>)

**Now let us create a demo.hbs file in our views directory with the following content:**

<!DOCTYPE html>

<html>

<body>

<p>This is a Demo Page on localhost!</p>

</body>

</html>

**Now, we render our webpage through express to the local server.**

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

res.render('demo')

})

app.listen(3000)

Now, open your browser and type *localhost:3000* on web address and verify the webpage at your server.

Now we will see how we can dynamically link the pages to server-side data.  
In the index.js, we declare a demo object, in practice, the object can be a result of the request body and/or database query.

var demo = {

name : 'Rohan',

age : 26

}

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

res.render('dynamic', {d

emo : demo})

Here we send the demo object as a demo to our hbs page. We can retrieve the information in dynamic.hbs present in the views folder.})

<!DOCTYPE html>

<html>

<body>

<p>{{demo.name}} is {{demo.age}} years old.</p>

</body>

</html>

**Output:**

Rohan is 26 years old

Given multiple values, we can iterate over all of them to perform the same functionality/display for each of the elements.

Let’s take an example, add the following code to your index.js and run the server and get a response.

var projects = {

name : 'Rahul',

skills : ['Data Mining', 'BlockChain Dev', 'node.js']

}

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

res.render('projects', {project : project});

})

where out views/projects.hbs looks something like:

|  |
| --- |
| <!DOCTYPE html>  <html>      <body>          {{projects.name}} has the following skills : <br>          {{#each projects.skills}}              {{this}} <br>          {{/each}}      </body>  </html>  **Output:**  Rahul has the following skills :  Data Mining  BlockChain Dev  node.js |