**Express** is built on top of **Node**, so yes **Express** adds more features. **Node** is just a Javascript environment with libraries to make it easy to write software, where **Express** extends **Node** specifically to make webservers easy to write. ...

The comparison is not entirely correct. The difference between node.js and express.js in the level of abstraction:

1. Node.js is a *platform* for building server-side *event-driven i/o application* using javascript.
2. Express.js is a *framework based on node.js* for building *web-application* using principles and approaches of node.js

npm install --save express

var express = require('express');

var app = express();

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

res.send("Hello world!");

});

app.listen(3000);

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. |

# ExpressJS - Routing

app.method(path, handler)

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

Path is the route at which the request will run.

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

var express = require('express');

var app = express();

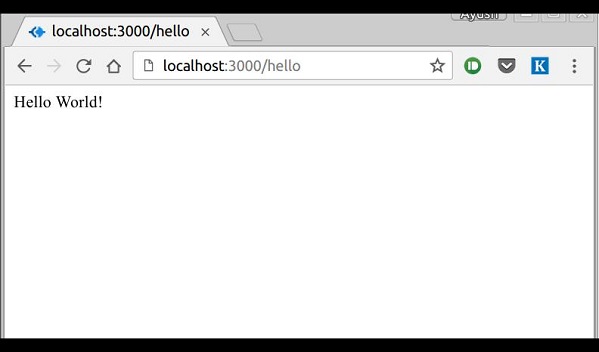
app.get('/hello', function(req, res){

res.send("Hello World!");

});

app.listen(3000);

If we run our application and go to **localhost:3000/hello**, the server receives a get request at route **"/hello"**, our Express app executes the **callback** function attached to this route and sends **"Hello World!"** as the response.



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

var express = require('express');

var app = express();

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

res.send("Hello World!");

});

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

res.send("You just called the post method at '/hello'!\n");

});

app.listen(3000);

To test this request, open up your terminal and use cURL to execute the following request −

curl -X POST "http://localhost:3000/hello"

Curl request

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

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

res.send("HTTP method doesn't have any effect on this route!");

});

This method is generally used for defining middleware, which we'll discuss in the middleware chapter.

Routers

Defining routes like above is very tedious to maintain. To separate the routes from our main **index.js** file, we will use **Express.Router**. Create a new file called **things.js** and type the following in it.

var express = require('express');

var router = express.Router();

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

res.send('GET route on things.');

});

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

res.send('POST route on things.');

});

//export this router to use in our index.js

module.exports = router;

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

var express = require('Express');

var app = express();

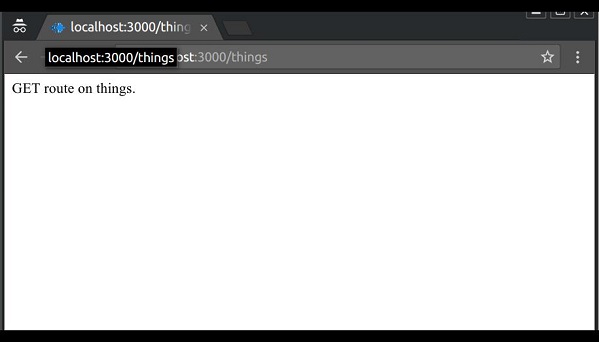
var things = require('./things.js');

//both index.js and things.js should be in same directory

app.use('/things', things);

app.listen(3000);

The ***app.use*** function call on route **'/things'** attaches the **things** router with this route. Now whatever requests our app gets at the '/things', will be handled by our things.js router. The **'/'** route in things.js is actually a subroute of '/things'. Visit localhost:3000/things/ and you will see the following output.



Routers are very helpful in separating concerns and keep relevant portions of our code together. They help in building maintainable code. You should define your routes relating to an entity in a single file and include it using the above method in your **index.js** file.

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. |

# RL Building

Here is an example of a dynamic route −

var express = require('express');

var app = express();

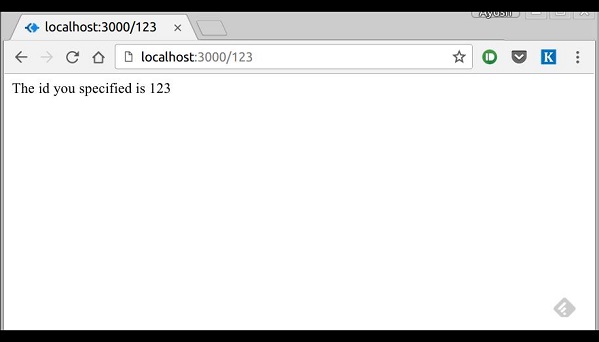
app.get('/:id', function(req, res){

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

});

app.listen(3000);

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



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

var express = require('express');

var app = express();

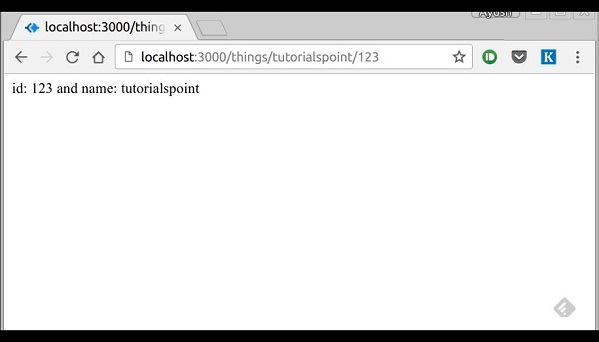
app.get('/things/:name/:id', function(req, res) {

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

});

app.listen(3000);

To test the above code, go to **http://localhost:3000/things/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){

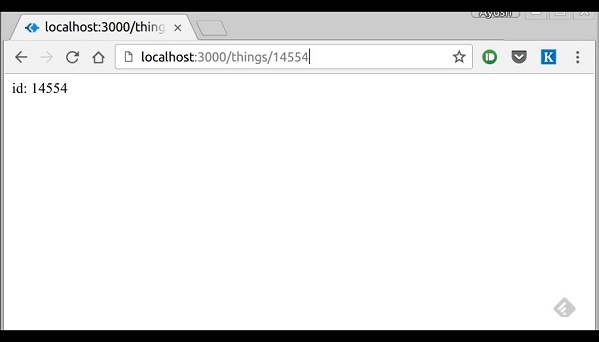
res.send('Sorry, this is an invalid URL.');

});

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.

# 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');

});

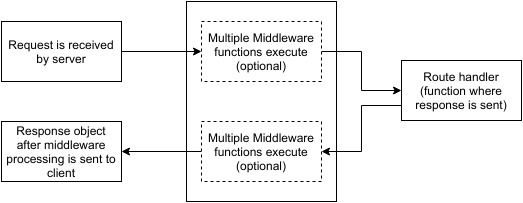
app.listen(3000);

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 −



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

Third Party Middleware

A list of Third party middleware for Express is available [here](http://expressjs.com/en/resources/middleware.html). Following are some of the most commonly used middleware; we will also learn how to use/mount these −

body-parser

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

var bodyParser = require('body-parser');

//To parse URL encoded data

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

//To parse json data

app.use(bodyParser.json())

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

cookie-parser

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

var cookieParser = require('cookie-parser');

app.use(cookieParser())

express-session

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

We have many other third party middleware in ExpressJS. However, we have discussed only a few important ones here.