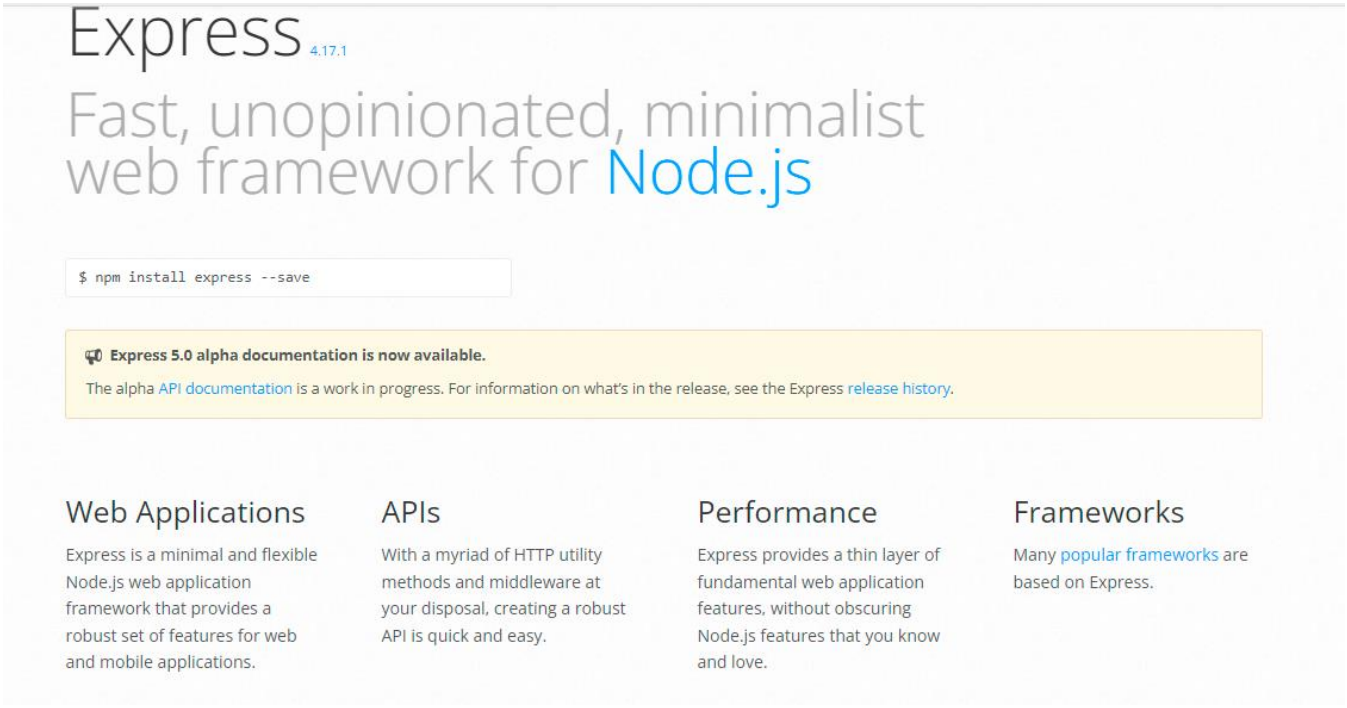


https://expressjs.com/

About the Tutorial

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. It is an open source framework developed and maintained by the Node.js foundation.

Audience

This tutorial has been created for anyone who has a basic knowledge of HTML, Javascript and how client-servers work. After completing this tutorial, you will be able to build moderately complex websites and back-ends for you mobile applications.

Prerequisites

You should have basic knowledge of Javascript and HTML. If you are not acquainted with these, we suggest you to go through tutorials on those areas first. It will definitely help, if you have some exposure to HTTP, although it is not mandatory. Having a basic knowledge of MongoDB will help you with the Database chapter.

Table of Contents

1. EXPRESSJS – OVERVIEW

ExpressJS is a web application framework that provides you with a simple API to build websites, web apps and back ends. With ExpressJS, you need not worry about low level protocols, processes, etc.

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 foundation and numerous open source contributors.

Why Express?

Unlike its competitors like Rails and Django, which have an opinionated way of building applications, Express has no "best way" to do something. It is very flexible and pluggable.

Pug

Pug (earlier known as Jade) is a terse language for writing HTML templates. It -

- Produces HTML
- Supports dynamic code
- Supports reusability (DRY)

It is one of the most popular template language used with Express.

MongoDB and Mongoose

MongoDB is an open-source, document database designed for ease of development and scaling. This database is also used to store data.

Mongoose is a client API for **node.js** which makes it easy to access our database from our Express application.

2. EXPRESSJS – ENVIRONMENT

In this chapter, we will learn how to start developing and using the Express Framework. To start with, you should have the Node and the npm (node package manager) installed. If you don't already have these, go to the Node setup to install node on your local system. Confirm that node and npm are installed by running the following commands in your terminal.

```
node --version
npm --version
```

You should get an output similar to the following.

```
v5.0.0
3.5.2
```

Now that we have Node and **npm** set up, let us understand what **npm** is and how to use it.

Node Package Manager(npm)

npm is the package manager for node. The npm Registry is a public collection of packages of open-source code for Node.js, front-end web apps, mobile apps, robots, routers, and countless other needs of the JavaScript community. npm allows us to access all these packages and install them locally. You can browse through the list of packages available on npm at npmJS.

How to use npm?

There are two ways to install a package using npm: globally and locally.

- **Globally**: This method is generally used to install development tools and CLI based packages. To install a package globally, use the following code.

```
npm install -g <package-name>
```

- **Locally**: This method is generally used to install frameworks and libraries. A locally installed package can be used only within the directory it is installed. To install a package locally, use the same command as above without the **-g** flag.

```
npm install <package-name>
```

Whenever we create a project using npm, we need to provide a **package.json** file, which has all the details about our project. npm makes it easy for us to set up this file. Let us set up our development project.

Step 1: Start your terminal/cmd, create a new folder named hello-world and cd (create directory) into it:

```
ayushgp@dell:~$ mkdir hello-world
ayushgp@dell:~$ cd hello-world/
ayushgp@dell:~/hello-world$
```

Step 2: Now to create the package.json file using npm, use the following code.

```
npm init
```

It will ask you for the following information.

```
Press ^C at any time to quit.
name: (hello-world)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author: Ayush Gupta
license: (ISC)
About to write to /home/ayushgp/hello-world/package.json:

{
  "name": "hello-world",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Ayush Gupta",
  "license": "ISC"
}


Is this ok? (yes) yes
ayushgp@dell:~/hello-world$
```

Just keep pressing enter, and enter your name at the "author name" field.

Step 3: Now we have our 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 --save express
```

To confirm that Express has installed correctly, run the following code.

```
ls node_modules #(dir node_modules for windows)
```

Tip: The **--save** flag can be replaced by the **-S** flag. This flag ensures that Express is added as a dependency to our **package.json** file. This has an advantage, the next time we need to install all the dependencies of our project we can just run the command *npm install* and it will find the dependencies in this file and install them for us.

This is all we need to start development using the Express framework. 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
```

You can now start working on Express.

3. EXPRESSJS – HELLO WORLD

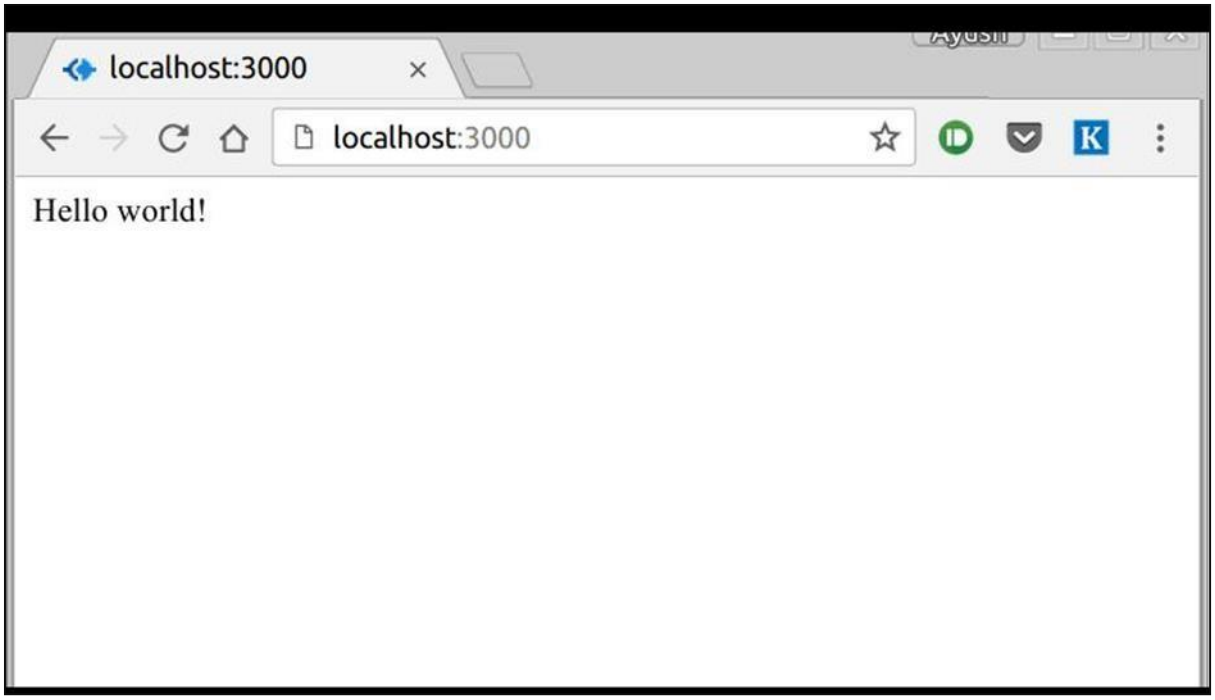
We have set up the development, now it is time to start developing our first app using Express. Create a new file called **index.js** and type the following in it.

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

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

app.listen(3000);
```

Save the file, go to your terminal and type the following.

```
nodemon index.js
```

This will start the server. To test this app, open your browser and go to http://localhost:3000 and a message will be displayed as in the following screenshot.



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.

Argument	Description
port	A port number on which the server should accept incoming requests.
host	Name of the domain. You need to set it when you deploy your apps to the cloud.
backlog	The maximum number of queued pending connections. The default is 511.
callback	An asynchronous function that is called when the server starts listening for requests.

4. EXPRESSJS – ROUTING

Web frameworks provide resources such as HTML pages, scripts, images, etc. at different routes.

The following function is used to define routes in an Express application:

app.method(path, handler)

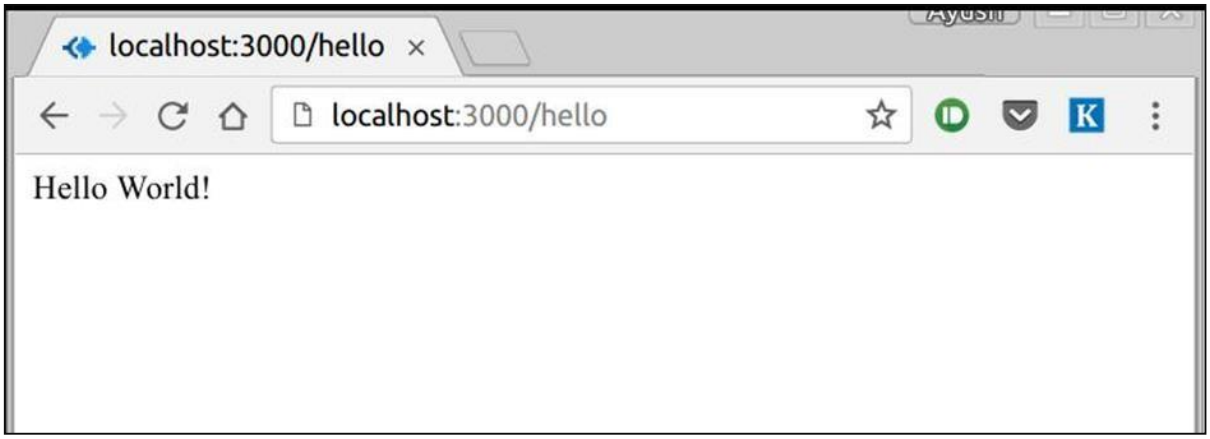
This METHOD can be applied to any one of the HTTP verbs – get, set, put, delete. An alternate method also exists, which executes independent of the request type.

Path is the route at which the request will run.

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

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

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

app.listen(3000);
```

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



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"
```

```
ayushgp@swaggy:~/hello-world$ curl -X POST "http://localhost:3000/hello"
You just called the post method at '/hello'!
ayushgp@swaggy:~/hello-world$
```

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

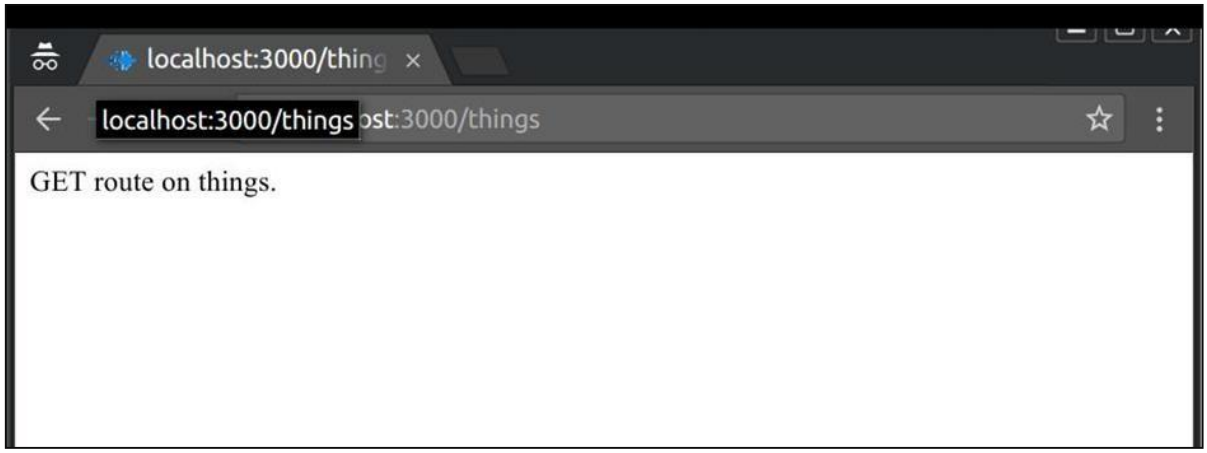
Defining routes like above is very tedious to maintain. To separate the routes from our main **index.js** file, we will use **Express.Router**. Create a new file called **things.js** and type the following in it.

```
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.

5. EXPRESSJS–HTTPMETHODS

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.

6. 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);
```

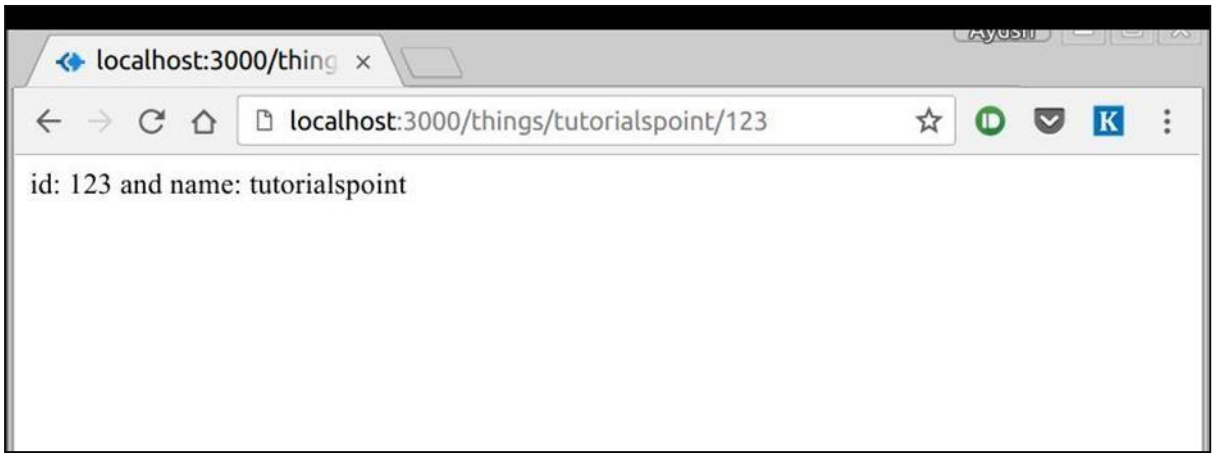
To test this go to http://localhost:3000/123. The following response will be displayed.

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 can 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){
```