Express.js

Getting MEAN 2 – A Practical Workshop

Sharif Malik | 2017

TABLE OF CONTENT

INTRODUCTION **3**
What is ExpressJS 3
Why to use 3
Where it is used 3
Installing Express 3
Tools/packages: Express Application generator **5**
Routing **7**
Definition 7
Function definition 7
Example 7
POSTMAN Chrome Extension **10**
Router **12**
Definition 12
Examples: 12
Middleware Functions **14**
Definition 14
Example 14
Difference between app.use() and app.all() 16
RESTful Web Services **17**
What is REST Architecture 17
HTTP Request methods 18
The rules of REST API 18
References **20**

INTRODUCTION

What is ExpressJS

- Express is a **minimal** and **flexible web application framework for Node.js** that provides a robust set of features for web and mobile applications.

- Great for building Web APIs and facilitates the rapid development of Node-based web applications.

Why to use

Following are some of the core features of Express framework:

- Allows to set up middleware's to respond to HTTP requests.

- Defines a routing table which is used to perform different actions based on HTTP method and URL.

- Allows to dynamically render HTML pages using different templates engines (https://expressjs.com/en/guide/using-template-engines.html)

Where it is used

- Popular services built on Express.js such as MySpace (https://myspace.com/), Ghost(https://ghost.org/)and many more.

- Foundation for other tools and frameworks like kranken (http://krakenjs.com/) and Sails(http://sailsjs.com/).

Installing Express

Assuming you have already installed Node.js, create a directory to hold your application, make that your working directory.

- **Step1**: make directory structure to work with (optional)

```
$ mkdir myApp
$ cd myApp
```

Use the npm init command to create a package.json file for your application.

For more information over **package.json**, please have a look here (https://docs.npmjs.com/getting-started/using-a-package.json)

$ npm init
This command prompts you for several things, such as the name and version of your application.

For now, you can hit ENTER key to accept the defaults for most of them, with the following **exception**.
entry point: (index.js)
Note: Enter app.js, or whatever you want the name of the **main file** to be. If you want it to be index.js, hit ENTER to accept the suggested default file name.

❖ **Step 2:** Install express package through npm

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 of package.json, omit the **--save** option:
$ npm install express

Note: Node modules installed with the **--save** option are added to the dependencies list in the **package.json** file. Afterwards, running **npm install** in the app directory will automatically install modules in the dependencies list.

Tools/packages: Express Application generator

Use the application generator tool, **express-generator**, to quickly create an application skeleton.

❖ **Step 1:** Install npm module to use express generator for creating project

Install express-generator with the following command.
$npm install express-generator -g

❖ **Step 2:** Use express command to create new project of express
After express-generator installation, you can go ahead to install express based app.
Example:
$ express myApp
create : myApp
create : myApp/package.json
create : myApp/app.js
create : myApp/public
create : myApp/routes
create : myApp/routes/index.js
create : myApp/routes/users.js
create : myApp/views
create : myApp/views/index.jade
create : myApp/views/layout.jade
create : myApp/views/error.jade
create : myApp/bin
create : myApp/bin/www
create : myApp/public/javascripts
create : myApp/public/images
create : myApp/public/stylesheets
create : myApp/public/stylesheets/style.css

❖ **Step 3:** Install dependencies

You can move the directory and install all required dependencies using npm:
$ cd myApp $ npm install

❖ **Step 4:** Run/ start the express project

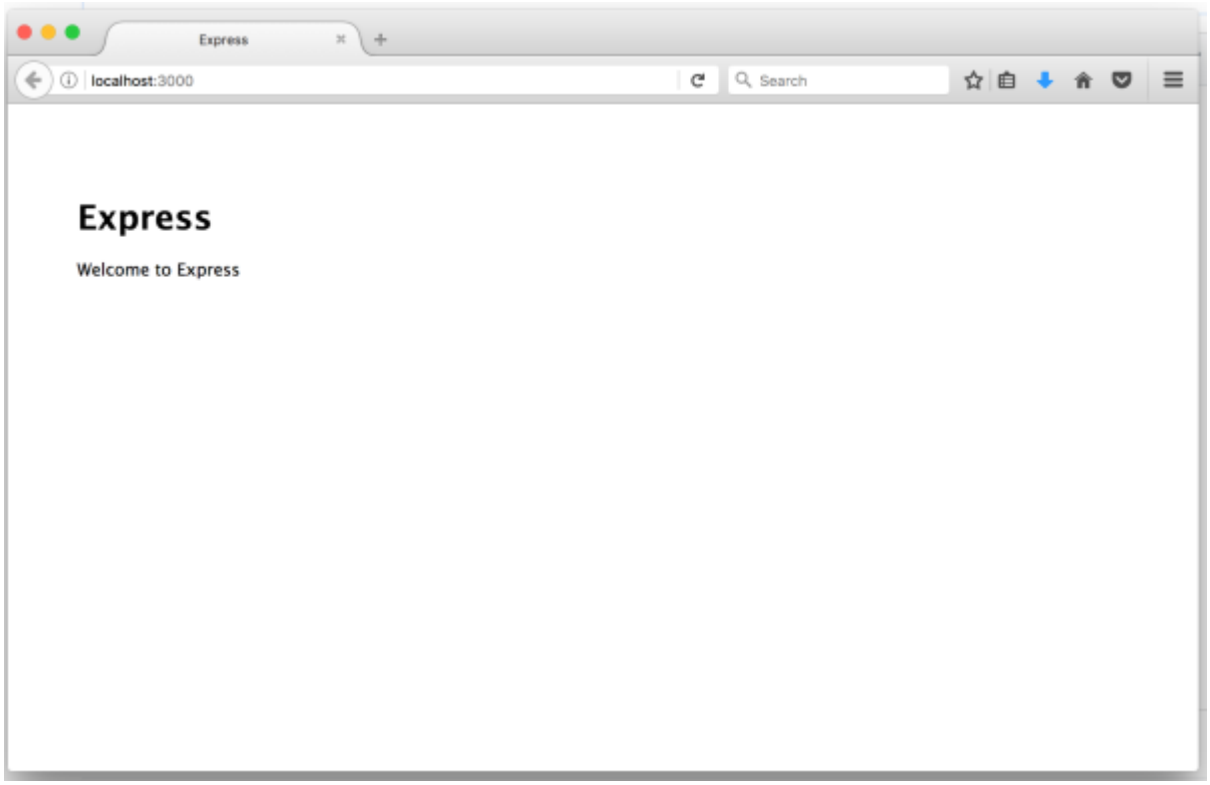
To run the application, use the below command:
$npm start

❖ **Step 5:** Verify the output

```
sharif@MacBook-Pro demoExpressGenerator $ npm start

> demoexpressgenerator@0.0.0 start
/Users/gitrepo/mean2/demo/demoExpressGenerator
> node ./bin/www
```

Verify the output on browser: http://localhost:3000



Have a look on generate directory structure and try to understand all the files and flow of the code. For more explanation over express-geenerator, please have a look here (https://expressjs.com/en/starter/generator.html)

Routing

Definition

- 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.

Function definition

Route function definition has the structure like this:

app.method(path,handler)
where:
app is an instance of express
method– is an HTTP request Method
path– is a path on the server
handler– is the function reference which should executed whenever the route is matched.

Example

Hello Express Js Framework
Example 1:

- Code:

You can download the source code (https://github.com/virtualSharif/gettingMEAN2/blob/master/express/examples/example1.js) or create example1.js file which have contents as the following :

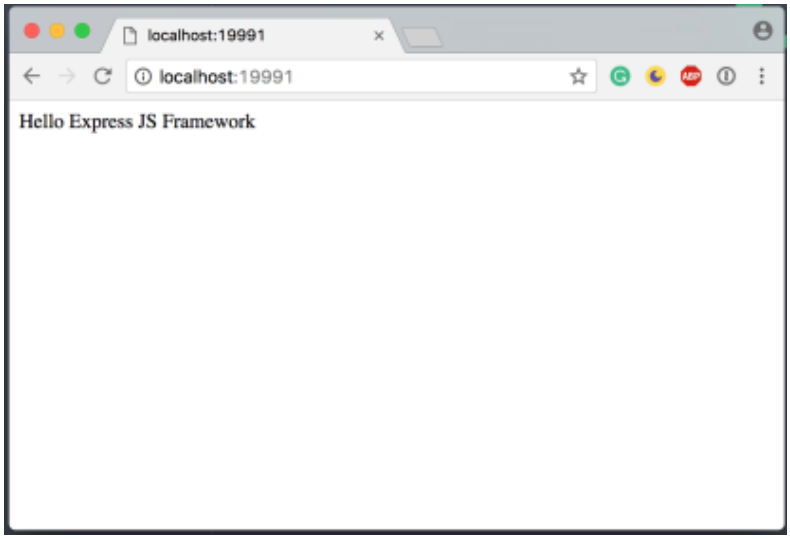
```
1.  var express = require('express');
2.  var app = express();
3.
4.  app.get('/', function (req, res) {
5.    res.send('Hello Express JS Framework');
6.  });
7.
8.  app.listen(19991, function () {
9.    console.log('Server is running on http://localhost:19991');
10. });
```

- To run the application:

$ node example1.js

❖ Verify the output :
Example app listening on port 19991!

Then, load http://localhost:19991/ in any browser to see the output:



Example 2:
Multiple HTTP methods at same URL

❖ Code:
You can download the source code (https://github.com/virualSharif/gettingMEAN2/blob/master/express/examples/example2.js)
or create **example2.js** le which have contents as the following :

```
1.  var express = require('express');
2.  var app = express();
3.
4.  //Respond with Hello World! on the homepage:
5.  app.get('/', function (req, res) {
6.    res.send('Hello Express JS Framework!');
7.  });
8.
9.  //Respond to GET request on the /user route
10. app.get('/users', function (req, res) {
11.   res.send('Got a Get request at /user');
12. });
13.
14. //Respond to POST request to the /user route
15.
16. app.post('/users', function (req, res) {
17.   res.send('Got a POST request at /user');
18. });
19.
20. //Respond to a PUT request to the /user route:
```

```
21.
22. app.put('/users', function (req, res) {
23.   res.send('Got a PUT request at /user');
24. });
25.
26. //Respond to a DELETE request to the /user route:
27.
28. app.delete('/users', function (req, res) {
29.   res.send('Got a DELETE request at /user');
30. });
31.
32. app.listen(19991, function () {
33.   console.log('Server is running on http://localhost:19991');
34. });
```

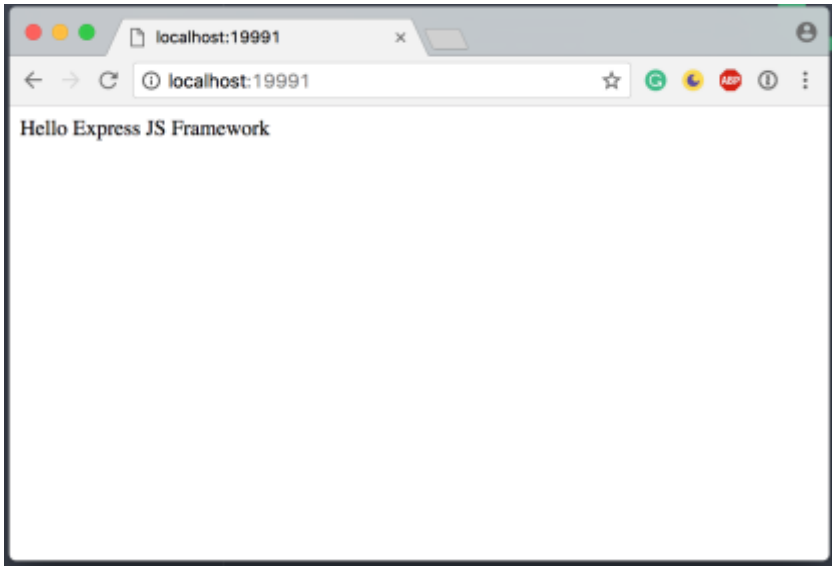
❖ To run the application :

$ node example1.js

❖ Verify the output :

Example app listening on port 19991!

Then , load http://localhost:19991/ in any browser to see the output:

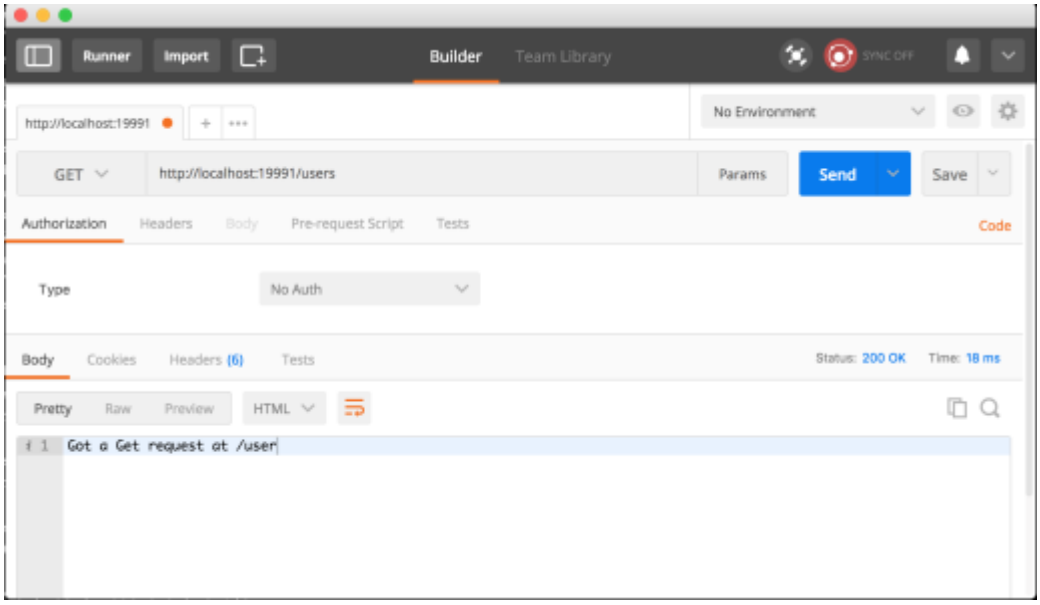


But now, to check the other URL you need to have some application which can send different type of HTTP request to our server.

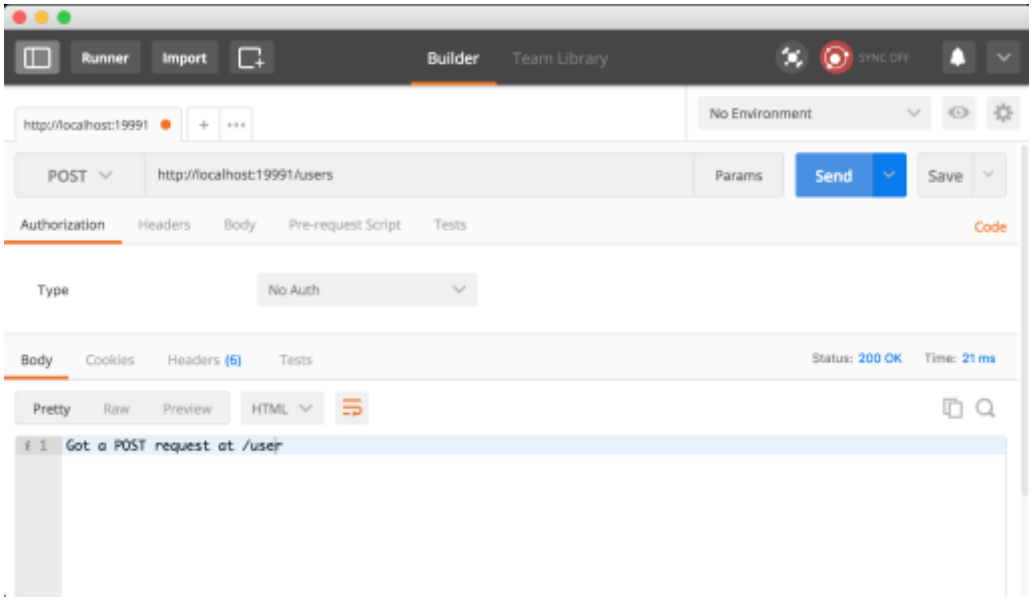
POSTMAN Chrome Extension

To check the different HTTP calls, use postman chrome app. (https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop?hl=en)
After installation of POSTMAN chrome extension:
You can send different types of HTTP requests to our server.

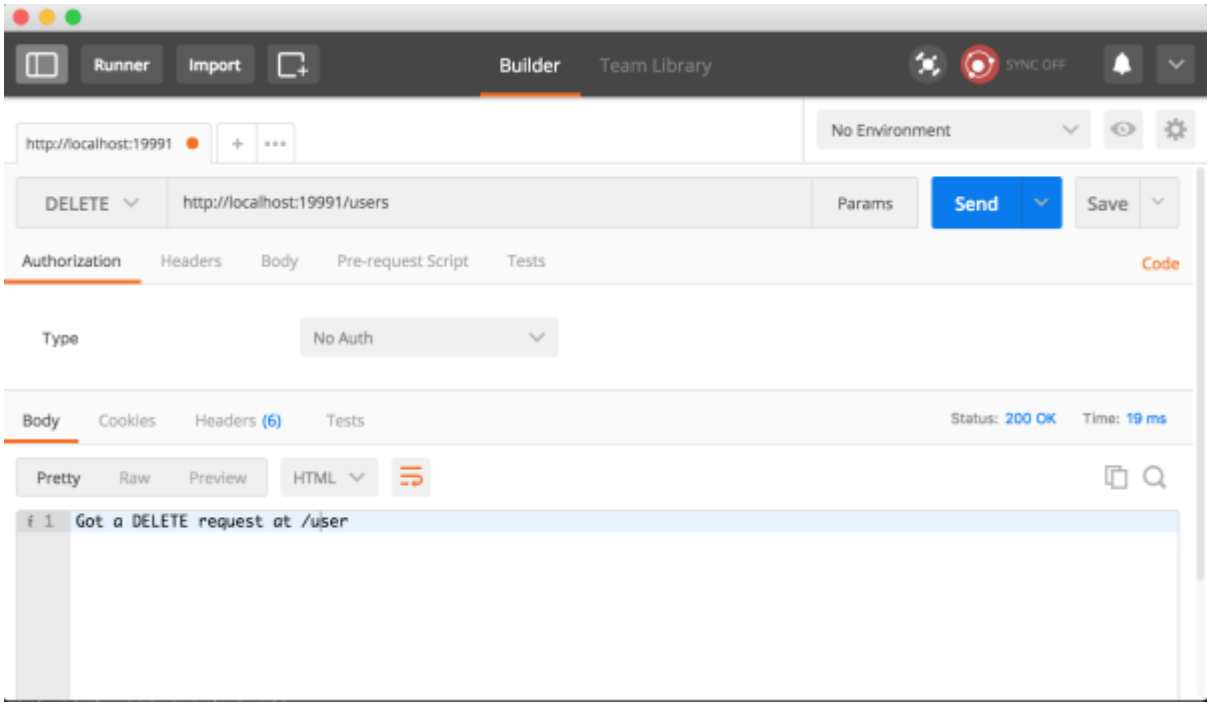
i. HTTP GET Method



ii. HTTP POST Method



iii. HTTP DELETE Method



Pointers: To know more about the postman chrome extension, please go through the link (https://www.getpostman.com/docs/)

Router

Definition

- **express.Router:** use the express.Router class to create modular, mountable route handlers.
- A Router instance is a complete middleware and routing system; for this reason, it is often referred to as a "mini-app".
- The top-level express object has a Router() method that creates a new router object.
- Once you've created a router object, you can add middleware and HTTP method routes (such as get, put, post, and so on) to it just like an application.

Examples:

The following example creates a router as a module, loads a middleware function in it, defines some routes, and mounts the router module on a path in the main app.

You can download the code from here (https://github.com/virtualSharif/gettingMEAN2/blob/master/express/examples/example3/birds.js)
or Create a **router file** named **birds.js** in the example3 directory, with the following content:

```
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.
10. router.get('/',function(req, res){
11.   res.send('You have hit get call on /birds');
12. });
13.
14. router.get('/about',function(req, res){
15.   res.send('You have hit get call on /birds/about');
16. });
17.
18. module.exports = router;
```

Then, create app.js file and load the router module birds.js in the **app.js**:

You can download the code from here(https://github.com/virtualSharif/gettingMEAN2/blob/master/express/examples/example3/app.js)

```
1.  var express = require('express');
2.  var app = express();
3.  var birds = require('./birds');
4.
5.  app.use('/birds', birds);
6.
7.  app.get('/', function (req, res) {
8.    res.send('Hello Express JS Framework');
9.  });
10.
11. app.listen(19991, function () {
12.   console.log('Server is running on http://localhost:19991');
13. });
```

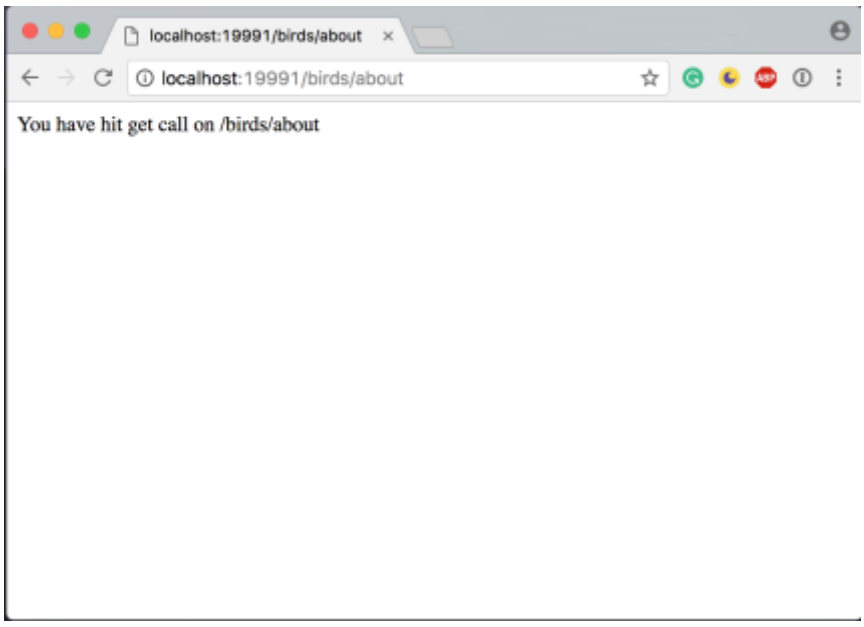
The app will now be able to handle requests to /birds and /birds/about, as well as call the timeLog middleware function that is specific to the route.

- To run the application :

$ node app.js

- Verify the output :

Example app listening on port 19991!
Then, load http://localhost:19991/birds in any browser to see the output:



Middleware Functions

Definition

- A Middleware is a callback that sits on top of the actual request handlers. It takes the same parameters as a route handler.

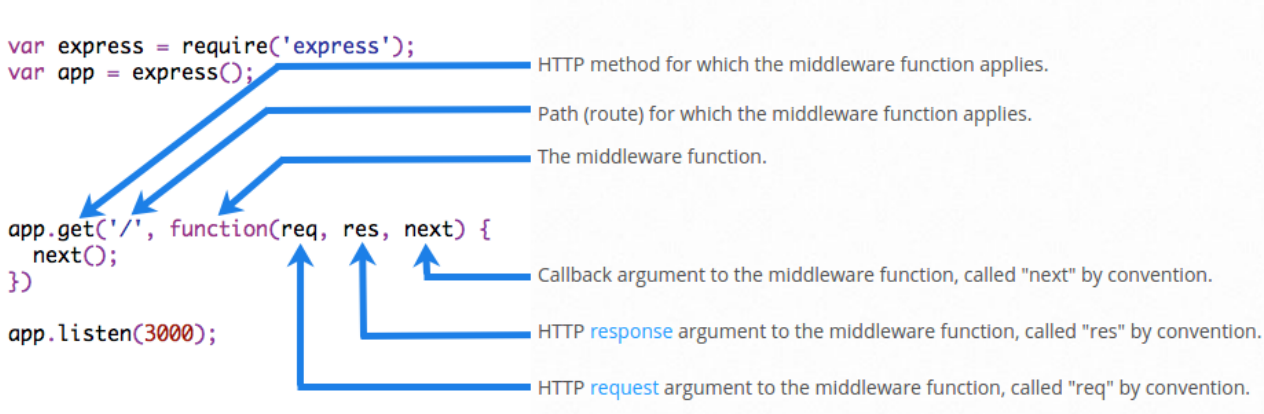
- Middleware functions are functions that have access to the **request object** (request), the **response object** (response), and the **next** middleware function in the application's request-response cycle.

- The next middleware function is commonly denoted by a variable named next.

- Middleware functions can perform the following tasks:
  - Execute any code.
  - Make changes to the request and the response objects.
  - End the request-response cycle.
  - Call the next middleware in the stack.

- If the current middleware function don't want to end the request-response cycle, then it must call next() to pass control to the next middleware function. Otherwise, the request will be left hanging.

- The following figure shows the elements of a middleware function call:



Example

Here is an example of a simple "Hello World" Express application.
The remainder of this article will define and add two middleware functions to the application:
One is called as **myLogger** that prints a simple log message whereas another is called as **requestTime** that displays the timestamp of the HTTP request.

Example 4: myLogger as Middleware function

- Here is a simple example of a middleware function called "myLogger".
- This function just prints "LOGGED" when a request to the app passes through it. The middleware function is assigned to a variable named **myLogger**.
- Example:
  You can download the source code from here(https://github.com/virtualSharif/gettingMEAN2/blob/master/express/examples/example4.js) or write the below code in example4.js

```
1.  var express = require('express');
2.  var app = express();
3.
4.  var myLogger = function (res, req, next){
5.    console.log('logged');
6.    next();
7.  };
8.
9.  app.use(myLogger);
10.
11. app.get('/', function (req, res) {
12.   res.send('Hello Express JS Framework');
13. });
14.
15. app.listen(19991, function () {
16.   console.log('Server is running on http://localhost:19991');
17. });
```

To load the middleware function, call **app.use()**, specifying the middleware function. For example, the above code loads the **myLogger** middleware function before the route to the root path (/).

Note :

- Every time the app receives a request, it prints the message "logged" to the terminal.
- The order of middleware loading is important: middleware functions that are loaded first are also executed first.
- If **myLogger** is loaded after the route to the root path, the request never reaches it and the app doesn't print "logged", because the route handler of the root path terminates the request-response cycle.
- The middleware function myLogger simply prints a message, then passes on the request to the next middleware function in the stack by calling the **next()** function.

For more details on middleware function, please click here (https://expressjs.com/en/guide/using-middleware.html)

Difference between app.use() and app.all()

- **Callback:** app.use() takes only one callback whereas app.all() can take multiple callbacks.
- **Path:** app.use() only see whether url starts with specified path whereas app.all() will match complete path.

Example to understand path difference:

```
app.use( "/product" , mymiddleware);
// will match /product
// will match /product/cool
// will match /product/foo

app.all( "/product" , handler);
// will match /product
// won't match /product/cool   <-- important
// won't match /product/foo    <-- important

app.all( "/product/*" , handler);
// won't match /product        <-- Important
// will match /product/cool
// will match /product/foo
```

 # NEXT() :
next()call inside a middleware invokes the next middleware or route handler depending on whichever is declared next.

Note:

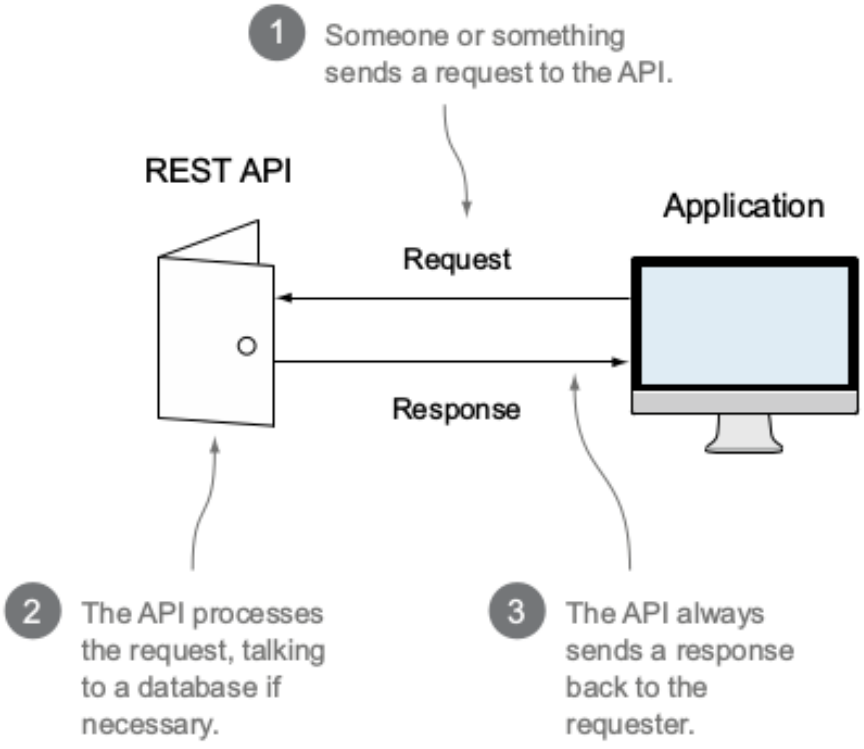
- next() call inside a route handler invokes the next route handler only.
- If there is a middleware next then it's skipped. Therefore middlewares must be declared above all route handlers.

RESTful Web Services

What is REST Architecture

- REST was first introduced by Roy Fielding in 2000.
- REST stands for **RE**presentational **S**tate **T**ransfer. REST is a web standard based architecture that uses HTTP Protocol.
- It revolves around resources where every component is a resource and a resource is accessed by a common interface using HTTP standard methods.
- A REST Server simply provides access to resources and a REST client accesses and modifies the resources using HTTP protocol.
- Here each resource is identified by URIs/ global Ids.

REST uses various representation to represent a resource, for example, text, JSON, XML, but JSON is the most popular one.



A REST API takes incoming HTTP requests, does some processing, and returns HTTP responses.

HTTP Request methods

The following four HTTP methods are commonly used in REST based architecture.

- GET: This is used to provide a read-only access to a resource.
- POST: This is used to create a new resource.
- DELETE: This is used to remove a resource.
- PUT: This is used to update an existing resource.

The rules of REST API

HTTP requests can have different methods that essentially tell the server what type of action to take.

Four basic Request methods used in a REST API

Request method	Use	Response
POST	Create new data in DB	New data object as seen in DB
GET	Read data from the DB	Data object as seen in DB
PUT	Update data in DB	Update data object as seen in DB
DELETE	Delete data from the DB	Null

URL paths and parameters for an API to the Users, all have the same base path, and several have the same userId parameter.

Action	URL	Parameters	Example
Create new User	/users	-	http://localhost:19991/api/users
Find all users	/users	-	http://localhost:19991/api/users
FindOne user	/users	userId	http://localhost:19991/api/users/123
Update user	/users	userId	http://localhost:19991/api/users/123
Delete user	/users	userId	http://localhost:19991/api/users/123

Conclusion:
Request method is used to link the URL to the desired actions, enabling the API to use the same URL for di erent actions.

Action	Method	URL	Parameters
Create new User	POST	/users	-
FindAll Users	GET	/users	-
FindOne user	GET	/users	userId
Update a specific user	PUT	/users	userId
Delete a specific user	DELETE	/users	userId

HTTP Status Code :
Most popular HTTP status code and how they might be used when sending responses to an API request.

Status Code	Name	Use Case
200	OK	A successful GET or PUT request
201	Created	A successful POST request
204	No content	A successful DELETE request
400	Bad request	An unsuccessful GET, POST, or PUT request, due to invalid content
401	Unauthorized	Requesting a restricted URL with incorrect credentials
403	Forbidden	Making a request that isn't allowed
404	Not Found	Unsuccessful request due to an incorrect parameter in the URL
405	Method not allowed	Request Method not allowed for the given URL
409	Conflict	Unsuccesful POST request when another object already exists with the same data
500	Internal Server Error	Problem with your server or the database server

References

https://expressjs.com/

https://expressjs.com/en/4x/api.html

https://github.com/expressjs/express

http://www.restapitutorial.com/lessons/whatisrest.html