

Full Stack Module-VII

Manual V8.3

ANUDIP FOUNDATION



ICONS AND THEIR MEANING



HINTS:

Get ready for helpful insites on difficult topics and questions.



STUDENTS:

This icon symbolize important instreutions and guides for the students.



TEACHERS/TRAINERS:

This icon symbolize important instreutions and guides for the trainers.

Lesson No	Lesson Name	Practical Duration (Minutes)	Theory Duration (Minutes)	Page No
1	Introduction of ExpressJs	60	60	03
2	Templating Engines	60	60	07
3	Working with Express.js	60	60	14
4	Request/Response in Express.js	120	nil	19

Total Duration: __Hours

Lesson 01: Introduction of ExpressJs (120 minutes)

Objective: After completing this lesson you will be able to learn about : <ul style="list-style-type: none">• Introduction• Features• Installation• Sample example	Materials Required: <ul style="list-style-type: none">• Computer With Windows XP and above• Stable Internet connection
Self- Learning Duration: 60 minutes	Practical Duration: 60 minutes
Total Duration: 120 minutes	

Introduction

ExpressJS 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. It offers a simple API to build websites, web apps and back ends. With ExpressJS, there is no need to worry about low level protocols, processes, etc.

Features

There are definite areas where ExpressJS excels. Here are some of the most impressive features of this framework:

- Rapid Server-Side Programming

Being a Node.js framework, Express.js packages many of the commonly used Node.js features, into functions that can be easily called anywhere on the program. As a result, complex tasks that would otherwise take a Node developer several hundred lines and several hours to program can easily be done by Express JS developers in just a few lines of code and within a few minutes. Express web application development is therefore much quicker than pure Node.js development.

- Routing

Routing allows a web application to preserve web page states through their URLs. These URLs may be shared with other users, and visiting these URLs will take users to the exact page state that was originally shared. Node.js has a routing mechanism, but it's a basic and rudimentary one. Express.js offers a more advanced and efficient routing mechanism that is able to handle highly dynamic URLs.

- Debugging

All developers encounter bugs with every project that can cause entire applications to malfunction, and one of the most critical tasks of developers is identifying the source of these bugs and correcting them in

the quickest possible time. Fortunately, Express.js provides an easy debugging mechanism to allow developers to quickly pinpoint which part of the application causes bugs.

- Templating

Express.js provides a templating engine that allows web pages to have dynamic content by constructing HTML templates on the server side, replacing dynamic content with their proper values, and then sending these to the client side for rendering. In addition to enabling dynamic content, it also takes a significant load from the client side which may have highly variable hardware specifications, and as such, it can make applications more efficient.

- Middleware

Express.js uses middleware to systematically arrange different function calls. A middleware is a chunk or cluster of code that has access to a user's request, the application's response, and the next middleware to be used. With such architecture, it becomes easy for ExpressJS developers to add, remove, or modify various features to and from the application, giving high scalability to the application.

Installation

Assuming you've already installed Node.js, 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.

```
$ 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
```

A sample example

Following is a very basic Express app which starts a server and listens on port 8081 for connection. This app responds with Hello World! for requests to the homepage. For every other path, it will respond with a 404 Not Found.

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

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

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)
})
```

Save the above code in a file named server.js and run it with the following command.

```
$ node server.js
```

You will see the following output –

```
Example app listening at http://0.0.0.0:8081
```

Open <http://127.0.0.1:8081/> in any browser to see the following result:



Practical Session

Instructions: Create the code....using an editor....and save it with HTML extension.

1. Create an ExpressJS program that will display "Hello Human" on the screen

2. Create an ExpressJS program that will display your name on the screen

Note: After saving the code, click the created HTML file to view the output.

Reviewing the chapter

- ExpressJS 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.
- Express.js provides a templating engine that allows web pages to have dynamic content by constructing HTML templates on the server side, replacing dynamic content with their proper values, and then sending these to the client side for rendering.

Testing your skills

1. _____ allows a web application to preserve web page states through their URLs

a) Routing, b) Debugging, c) Templating, d) None

2. Express.js uses _____ to systematically arrange different function calls

a) Router, b) servers, c) middleware, d) None

3. _____ allows web pages to have dynamic content by constructing HTML templates on server side

a) templating engine, b) JSON APIs, c) routers, d) None of the above

4. _____ is a chunk or cluster of code that has access to a user's request

a) server side programming, b) middleware, c) router, d) templating engines

5. Which of these are features of ExpressJS

a) Easy debugging, b) Advanced routing, c) Rapid server side programming, d) All of the above

Lesson 02: Templating Engines (120 minutes)

Objective: After completing this lesson you will be able to learn about : <ul style="list-style-type: none">• Jade Templating Engine• Nested Routers in ExpressJS	Materials Required: <ul style="list-style-type: none">• Computer With Windows XP and above• Stable Internet connection
Self- Learning Duration: 60 minutes	Practical Duration: 60 minutes
Total Duration: 120 minutes	

Jade is a template engine for Node.js. Jade syntax is easy to learn. It uses whitespace and indentation as a part of the syntax.

Install jade into your project using NPM as below.

```
npm install jade
```

Jade template must be written inside .jade file. And all .jade files must be put inside views folder in the root folder of Node.js application.

By default Express.js searches all the views in the views folder under the root folder, which can be set to another folder using views property in express e.g. `app.set('views','MyViews')`.

Have a look at a sample Jade template:

```
doctype html
html
  head
    title Jade Page
  body
    h1 This page is produced by Jade engine
    p some paragraph here..
```

It will produce following html:

```
<!DOCTYPE html>
<html>
<head>
  <title>Jade Page</title>
</head>
<body>
  <h1>This page is produced by Jade engine</h1>
  <p>some paragraph here..</p>
</body>
</html>
```


Using Jade template with Express.js

Express.js can be used with any template engine. Here, we will use different Jade templates to create HTML pages dynamically.

In order to use Jade with Express.js, create sample.jade file inside views folder and write following Jade template in it (*we shall be saving it with the name sample.jade*):

```
doctype html
html
  head
    title Jade Page
  body
    h1 This page is produced by Jade engine
    p some paragraph here..
```

Now, write the following code to render above Jade template using Express.js (*we shall be saving it with the name server.js*):

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

//set view engine
app.set("view engine", "jade")

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

  res.render('sample');

});

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

In the above example, first we imported the express module and then set the view engine using app.set() method.

The set() method sets the "view engine", which is one of the application setting property in Express.js. In the HTTP Get request for home page, it renders sample.jade from the views folder using res.render() method.

Nested Routers in Express.js

Express makes it easy to nest routes in your routers.

Let's say you're building routes for a website www.music.com. Music is organized into albums with multiple tracks. Users can click to see a track list. Then they can select a single track and see a sub-page about that specific track.

```
const express = require('express');
const app = express();
const albumsRouter = require('./routers/albums');

//...

app.use('/albums', albumsRouter); // Forwards any requests to the /albums URI to our albums Router

//...
```

Inside our albums.js file, this is what the code should be:

```
const albums = require('express').Router();

//...

// Our app.use in the last file forwards a request to our albums router.
// So this route actually handles `/albums` because it's the root route when a request to /albums is forwarded to our router.
albums.get('/', function(req, res, next) {
  // res.send() our response here
});

// A route to handle requests to any individual album, identified by an album id
albums.get('/:albumId', function(req, res, next) {
  let myAlbumId = req.params.albumId;
  // get album data from server and res.send() a response here
});

//...

module.exports = albums;
```

Till this part, it is all right!

But then, what happens when we start to add tracks to the routes?

Since tracks are associated with a particular album, our routes will end up looking something like this:

www.music.com/albums/[AlbumIdHere]/tracks/[TrackIdHere]

In the spirit of Express' modular routers, we should have a separate router for tracks. That router isn't part of our top level application logic. We can nest it in our albums router instead.

First let's set up the proper forwarding in our albums router:

```
const albums = require('express').Router();
const tracks = require('./tracks').Router();

//...

// Our root route to /albums
albums.get('/', function(req, res, next) {
  // res.send() our response here
});

// A route to handle requests to any individual album, identified by an album id
albums.get('/:albumId', function(req, res, next) {
  let albumId = req.params.albumId;
  // retrieve album from database using albumId
  // res.send() response with album data
});

// Note, this route represents /albums/:albumId/tracks because our top-level router is already forwarding /albums to our Albums router!
albums.use('/:albumId/tracks', tracks);

//...

module.exports = albums;
```

Now let's make our tracks router in a track.js file.

```
const tracks = require('express').Router();

//...

// The root router for requests to our tracks path
track.get('/', function(req, res, next) {
  let albumId = req.params.albumId; // Our problem line

  // retrieve album's track data and render track list page
});

// The route for handling a request to a specific track
track.get('/:trackId', function(req, res, next) {
  let albumId = req.params.albumId; // <-- How do we get this?
  let trackId = req.params.trackId;

  // retrieve individual track data and render on single track page
});

//...

module.exports = tracks;
```

So, this will work, isn't it?

But then, isn't there something inappropriate with this programming?

To retrieve an album's track data, we'll need to know the album's id using the *req.params.albumId* that we set up in our albums route.

If you use the code above, you'll discover that our *albumId* is undefined. Our *req.params* does not have an *albumId* property inside our tracks router. Our router is broken because we can't figure out which album the user requested when our tracks router takes over.

So, what's the solution?

Let's see!

Express provides an elegant solution. Back in our albums router we can refactor the route that forwards requests to our nested tracks router.

Previously, we had:

```
albums.use('/:albumId/tracks', tracks);
```

Our refactoring will attach our *req.params.albumId* to our req object directly. We'll next() the request, which tells Express to send the request to the next applicable route. But instead of allowing Express to choose the route, we'll include our optional third parameter to *.use()*, the router that we want Express to send the request to. Have a look:

```
albums.use('/:albumId/tracks', function(req, res, next) {  
  req.albumId = req.params.albumId;  
  next()  
}, tracks);
```

We've saved our param on our req object and told Express to send it directly to our tracks router.

Here's what our albums router will look like now:

```
const albums = require('express').Router();  
const tracks = require('./tracks').Router();  
  
//...  
  
albums.get('/', function(req, res, next) {  
  // send our response here  
});  
  
albums.get('/:albumId', function(req, res, next) {  
  let albumId = req.params.albumId;  
  // retrieve album from database using albumId  
  // send response with album data  
});  
  
// The fix for our parameters problem  
albums.use('/:albumId/tracks', function(req, res, next) {  
  req.albumId = req.params.albumId;  
  next()  
}, tracks);  
  
//...  
  
module.exports = albums;
```

Now in tracks we can refactor to access the *req.albumId* property we set. Have a look:

```
const tracks = require('express').Router();  
  
//...  
  
// The root router for requests to our tracks path  
track.get('/', function(req, res, next) {  
  let albumId = req.albumId; // Refactored to access our albumId property  
  
  // retrieve album's track data and render track list page  
});  
  
// The route for handling a request to a specific track  
track.get('/:trackId', function(req, res, next) {  
  let albumId = req.albumId; // Refactored to access our albumId property!  
  let trackId = req.params.trackId;  
  
  // retrieve individual track data and render on single track page  
});  
  
//...  
  
module.exports = tracks;
```

With everything refactored, we can access all the data we need.

Any time that we need to access a parameter from a parent router in its child, we can follow the same procedures:

- Attach the data we need from the parent router to our req object directly.
- next() in our parent router's .use() method.
- Set the optional third parameter in our parent's .use() as the child router that we want the request send to.

Advantages of Template Engine in Node.JS

- Improves developer's productivity
- Improves readability and maintainability
- Faster performance
- Maximizes client side processing
- Single template for multiple pages
- Templates can be accessed from CDN (Content Delivery Network)

Practical Session

Instructions: Create the code....using an editor....and save it with HTML extension.

1. Create a simple template page using Jade

```
doctype html
```

```
html
```

```
head
```

```
title Jade Page
```

```
body
```

```
h1 This page is produced by Jade engine
```

```
p some paragraph here..
```

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

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

```
//set view engine
```

```
app.set("view engine","jade")
```

```
app.get('/', function (req, res) {  
  
  res.render('sample');  
  
});  
  
var server = app.listen(5000, function () {  
  console.log('Node server is running..');  
});
```

2. Create a program using Jade that will display names of three students of your class.

Note: After saving the code, click the created HTML file to view the output.

Reviewing the chapter

In this chapter, we learned about Jade, a template engine and its use with ExpressJR. We also focused at the step by step procedure of nested router application in ExpressJR.

Testing your skills

1. Jade is a _____ for Node.js
 - a) template engine, b) router, c) node, d) none of the above

2. Jade files are stored with extension _____
 - a) .path, b) .html, c) .js, d) .jade

3. Which of these are the features of template engines
 - a) Improves readability and maintainability, b) Faster performance, c) Maximizes client side processing, d) all of the above

4. What does CDN stands for
 - a) Control domain name, b) content delivery network, c) content domain network, d) control delivery network

Lesson 03: Working with Express.js (120 minutes)

Objective: After completing this lesson you will be able to learn about : <ul style="list-style-type: none">• Connect module• Installation• Application steps	Materials Required: <ul style="list-style-type: none">• Computer With Windows XP and above• Stable Internet connection
Self- Learning Duration: 60 minutes	Practical Duration: 60 minutes
Total Duration: 120 minutes	

Connect Module

Connect is a simple framework to glue together various "middleware" to handle requests.

Installing Connect

```
$ npm install connect
```

Creating an app

The main component is a Connect "app". This will store all the middleware added and is, itself, a function.

```
var app = connect();
```

Using middleware

The core of Connect is "using" middleware. Middleware are added as a "stack" where incoming requests will execute each middleware one-by-one until a middleware does not call next() within it.

```
app.use(function middleware1(req, res, next) {  
  // middleware 1  
  next();  
});  
app.use(function middleware2(req, res, next) {  
  // middleware 2  
  next();  
});
```

Mounting middleware

The `.use()` method also takes an optional path string that is matched against the beginning of the incoming request URL. This allows for basic routing.

```
app.use('/foo', function fooMiddleware(req, res, next) {  
  // req.url starts with "/foo"  
  next();  
});  
app.use('/bar', function barMiddleware(req, res, next) {  
  // req.url starts with "/bar"  
  next();  
});
```

Error middleware

There are special cases of "error-handling" middleware. There are middleware where the function takes exactly 4 arguments. When a middleware passes an error to next, the app will proceed to look for the error middleware that was declared after that middleware and invoke it, skipping any error middleware above that middleware and any non-error middleware below.

```
// regular middleware  
app.use(function (req, res, next) {  
  // i had an error  
  next(new Error('boom!'));  
});  
  
// error middleware for errors that occurred in middleware  
// declared before this  
app.use(function onerror(err, req, res, next) {  
  // an error occurred!  
});
```

Creating a server from the app

The last step is to actually use the Connect app in a server. The `.listen()` method is a convenience to start a HTTP server (and is identical to the `http.Server`'s `listen` method in the version of Node.js you are running).

```
var server = app.listen(port);
```

The app itself is really just a function with three arguments, so it can also be handed to `.createServer()` in Node.js.

```
var server = http.createServer(app);
```

Applications Steps

Step	Description
Validate Your Domains	Allows you to verify that you own each of your internal domains. We start with your primary domain, but other domains can also be validated. You can also add subdomains once a domain is validated.
Synchronize Your Directory	Allows us to automatically create each user's Mimecast account. This enables your users to send and receive email and sign into our applications.
Add Users Manually	Allows you to add user accounts that do not currently exist in your directory.
Set Up User Authentication	Allows you to specify the method to be used to authenticate users when they connect to our applications.
Prepare for Inbound Email	Allows you to set up your inbound email flow through us. The process differs depending on whether you have an Office 365 or On Premise / Hybrid Exchange .
Set Up Your Outbound Email	Allows you to set up your outbound email flow through us.
Import Permitted Senders	Allows you to let us know the external addresses that you regularly communicate with. This marks them as permitted senders, thereby bypassing the strict anti-spam screening applied to all inbound emails from unknown external senders.
Set Up Recipient Validation	Ensures only messages destined for valid internal email addresses are accepted. Emails that can't be matched against a valid internal address are rejected.
Modify Your MX Record	Routes your inbound email through us by updating your mail exchanger (MX) records. This designates us as the mail service responsible for accepting email on your behalf.
Secure Your Inbound Email	Looks at securing your firewall to protect you from messages received from a non-Mimecast IP address. The process differs depending on whether you have an Office 365 or On Premise / Hybrid exchange .
Set Up Journaling	Allows you to set up journaling to ensure your Mimecast archive has a full and complete record of all email. This includes your internal email, and copies delivered to members of distribution lists.
Manage Basic Administrators (Optional task)	Allows you to add users to the Basic Administrator role so they can assist with the implementation or general account management in the Administration Console
Manage Forwarding Addresses (Optional task)	Allows you to add 'Relay' mail forwarding rules that route messages external to your organization to a chosen address.
Bypass Anti-Spoofing (Optional task)	Allows you to bypass Anti-Spoofing policies. If you use external services to send email to your internal users, you can add these services as exceptions to your Anti-Spoofing policy.
Set Up Your TLS Policies (Optional task)	Allows you to enforce TLS communication between your mail server and us, as well as between us and specified external domains. This ensures full end to end TLS communication.

Practical Session

Instructions: Create the code....using an editor....and save it with HTML extension.

1. Create a simple middleware function that will display "a new request is received at " a specific date.

```
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);
```

2. Create a simple middleware function that will display "a new request is received by " a specific name.

Note: After saving the code, click the created HTML file to view the output.

Reviewing the chapter

Connect is a simple framework to glue together various "middleware" to handle requests.

There are special cases of "error-handling" middleware. There are middleware where the function takes exactly 4 arguments.

Testing your skills

1. _____ allows you to verify that you own each of your internal domains.
a) Set Up User Authentication, b) Validate Your Domains, c) Set Up Journaling, d) None of the above
2. _____ ensures only messages destined for valid internal email addresses are accepted

a) Bypass Anti-Spoofing, b) Set Up User Authentication, c) Set Up Journaling, d) Set Up Recipient Validation

3. _____ looks at securing your firewall to protect you from messages received from a non-Mimecast IP address

a) Secure Your Inbound Email, b) Import Permitted Senders, c) Prepare for Inbound Email, d) None of the above

4. _____ allows us to automatically create each user's Mimecast account.

a) Set Up User Authentication, b) Set Up Recipient Validation, c) Set Up Journaling, d) Synchronize Your Directory

5. ____ allows you to add 'Relay' mail forwarding rules that route messages external to your organization to a chosen address

a) Bypass Anti-Spoofing, b) Manage Forwarding Addresses, c) Set Up Journaling, d) None

Lesson 04: Request/Response in Express.js (120 minutes)

Objective: After completing this lesson you will be able to learn about :

- Request params
- Response renders
- Middleware

Materials Required:

- Computer With Windows XP and above
- Stable Internet connection

Self- Learning Duration: 120 minutes

Practical Duration: nil

Total Duration: 120 minutes

The Request and Response objects are nothing but parameters to the callback function for Express JS and used exclusively in Express applications.

The express.js Request object basically represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, etc.

The Syntax:

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

The Request Object properties for Express JS

Index	Properties	Description
1.	req.app	This is used to hold a reference to the instance of the express application that is using the middleware.
2.	req.baseUrl	It specifies the URL path on which a router instance was mounted.
3.	req.body	It contains key-value pairs of data submitted in the request body. By default, it is undefined, and is populated when you use body-parsing middleware such as body-parser.
4.	req.cookies	When we use cookie-parser middleware, this property is an object that contains cookies sent by the request.
5.	req.fresh	It specifies that the request is "fresh." It is the opposite of req.stale.
6.	req.hostname	It contains the hostname from the "host" http header.
7.	req.ip	It specifies the remote IP address of the request.
8.	req.ips	When the trust proxy setting is true, this property contains an array of IP addresses specified in the ?x-forwarded-for? request header.
9.	req.originalurl	This property is much like req.url; however, it retains the original request URL, allowing you to rewrite req.url freely for internal routing purposes.
10.	req.params	An object containing properties mapped to the named route ?parameters?. For example, if you have the route /user/:name, then the "name" property is available as req.params.name. This object defaults to {}.
11.	req.path	It contains the path part of the request URL.
12.	req.protocol	The request protocol string, "http" or "https" when requested with TLS.
13.	req.query	An object containing a property for each query string parameter in the route.
14.	req.route	The currently-matched route, a string.
15.	req.secure	A Boolean that is true if a TLS connection is established.
16.	req.signedcookies	When using cookie-parser middleware, this property contains signed cookies sent by the request, unsigned and ready for use.
17.	req.stale	It indicates whether the request is "stale," and is the opposite of req.fresh.
18.	req.subdomains	It represents an array of subdomains in the domain name of the request.
19.	req.xhr	A Boolean value that is true if the request's "x-requested-with" header field is "xmlhttprequest", indicating that the request was issued by a client library such as jQuery

Request object methods

req.accepts (types)

This method is used to check whether the specified content types are acceptable, based on the request's Accept HTTP header field.

req.is(type)

This method returns true if the incoming request's "Content-Type" HTTP header field matches the MIME type specified by the type parameter.

req.get(field)

This method returns the specified HTTP request header field.

req.param(name [, defaultValue])

This method is used to fetch the value of param name when present.

Example of Request Param

```
// ?name=sasha
req.param('name')
// => "sasha"
// POST name=sasha
req.param('name')
// => "sasha"
// /user/sasha for /user/:name
req.param('name')
// => "sasha"
```

Response Object

The Response object (res) specifies the HTTP response which is sent by an Express app when it gets an HTTP request.

Syntax:

res.append(field [, value])

The response Object properties

Index	Properties	Description
1.	res.app	It holds a reference to the instance of the express application that is using the middleware.
2.	res.headersSent	It is a Boolean property that indicates if the app sent HTTP headers for the response.
3.	res.locals	It specifies an object that contains response local variables scoped to the request

The Response Render method

This method renders a view and sends the rendered HTML string to the client.

Syntax:

```
res.render(view [, locals] [, callback])
```

```
// send the rendered view to the client
res.render('index');
// pass a local variable to the view
res.render('user', { name: 'aryan' }, function(err, html) {
  // ...
});
```

Middleware function in Express.js

Middleware can be defined as different types of functions that are invoked by the Express.js routing layer before the final request handler. As the name specified, Middleware appears in the middle between an initial request and final intended route. In stack, middleware functions are always invoked in the order in which they are added.

Middleware is commonly used to perform tasks like body parsing for URL-encoded or JSON requests, cookie parsing for basic cookie handling, or even building JavaScript modules on the fly.

Middleware functions are the functions that access to the request and response object (req, res) in request-response cycle.

Tasks that middleware can perform

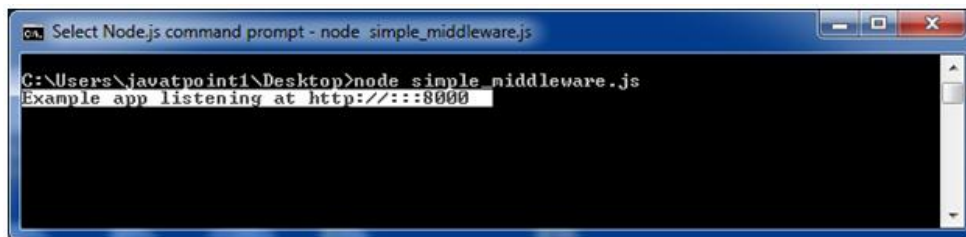
- It can execute any code.
- It can make changes to the request and the response objects.
- It can end the request-response cycle.
- It can call the next middleware function in the stack.

Sample Middleware coding example

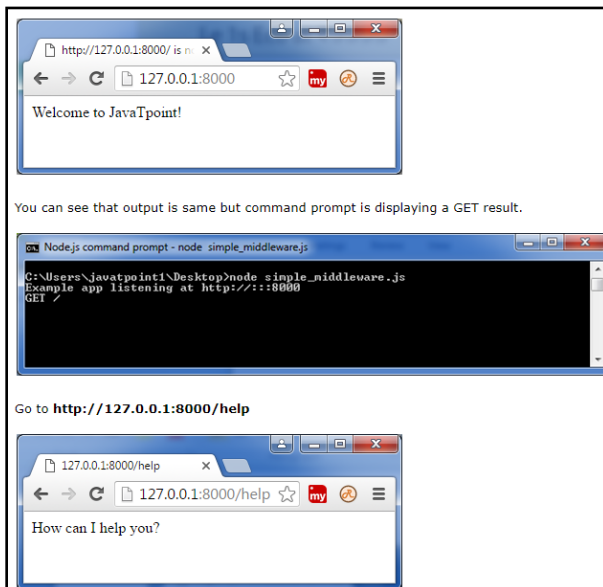
We will be using a middleware for recording every request being received. Let's create the code and save the file with name *simple_middleware.js*

```
var express = require('express');
var app = express();
app.use(function(req, res, next) {
  console.log('%s %s', req.method, req.url);
  next();
});
app.get('/', function(req, res, next) {
  res.send('Welcome to JavaTpoint!');
});
app.get('/help', function(req, res, next) {
  res.send('How can I help you?');
});
var server = app.listen(8000, function () {
  var host = server.address().address
  var port = server.address().port
  console.log("Example app listening at http://%s:%s", host, port)
})
```

When the command prompt is opened, you can observe that the server is listening:




Now, you can see the result generated by server on the local host <http://127.0.0.1:8000>:





```
Node.js command prompt - node simple_middleware.js
C:\Users\javatpoint1\Desktop>node simple_middleware.js
Example app listening at http://:::8000
GET /
GET /help
```

As many times as you reload the page, the command prompt will be updated.



```
Node.js command prompt - node simple_middleware.js
C:\Users\javatpoint1\Desktop>node simple_middleware.js
Example app listening at http://:::8000
GET /
GET /help
GET /help
```

Note: In the above example `next()` middleware is used.

A briefing on the entire example

In the example, a new function is used for invoking with every request via `app.use()`. Middleware is a function, just like route handlers and invoked also in the similar manner. You can add more middlewares above or below using the same API.

Reviewing the chapter

- The Request and Response objects are nothing but parameters to the callback function for Express JS and used exclusive in Express applications.
- The Request object basically represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, etc.
- The Response object specifies the HTTP response which is sent by an Express app when it gets an HTTP request.
- Middleware can be defined as multiple are different types of functions that are invoked by the Express.js routing layer before the final request handler.

Testing your skills

1. Which of the following are tasks performed by middleware:

a) executing any code, b) ending the request-response cycle, c) Both (a) and (b), d) none of the above

2. _____ can be defined as different types of functions that are invoked by the Express.js routing layer

a) Request, b) Response, c) Middleware, d) None of the above

3. _____ is a Boolean that is true if a TLS connection is established

a) req.secure, b) req.route, c) req.xhr, d) req.ips

4. _____ specifies an object that contains response local variables scoped to the request

a) res.app, b) res.locals, c) res.headersSent, d) none

5. _____ method is used to check whether the specified content types are acceptable

a) req.get(field), b) req.is(type), c) req.accepts (types), d) None of the above