

COMPLETE NOTES ON

Express.JS

COPYRIGHTED BY:-

INSTAGRAM:- @codes.learning

YOUTUBE:- Point of Programming

TELEGRAM:- codeslearning TG

WRITTEN BY:-

Mayuri Khavche (Software Engineer)

INDEX

| Sr. No | TOPIC | Page NO |
|-----------|------------------------|------------|
| 1. | Express.js Tutorial | 1 |
| 2. | Install Express.js | 2 |
| 3. | Express.js Request | 6 |
| 4. | Express.js Response | 10 |
| 5. | Express.js Get | 14 |
| 6. | Express.js Post | 24 |
| 7. | Express.js Routing | 27 |
| 8. | Express.js Cookies | 29 |
| 9. | Express.js File Upload | 32 |
| 10. | Express.js MiddleWare | 37 |
| 11. | Express.js Scaffolding | 42 |
| 12. | Express.js Template | 46 |

Express.js

What is Express.js

Express is a fast, assertive, essential and moderate Web framework of Node.js. You can assume express as a layer built on the top of the Nodejs that helps manage a server and routes. It provides a robust set of features to develop Web and mobile applications.

Let's see some of the core features of Express framework:

- It can be used to design Single-page, multi-page, and hybrid Web applications.
- It allows to setup middlewares to respond to HTTP Requests.
- It defines a routing table which is used to perform different actions based on HTTP method and URL.
- It allows to dynamically reader HTML pages based on passing arguments to templates.

Why USE Express

- Ultra fast I/O
- Asynchronous and single threaded
- MVC like structure
- Robust API makes routing easy

How does Express look like

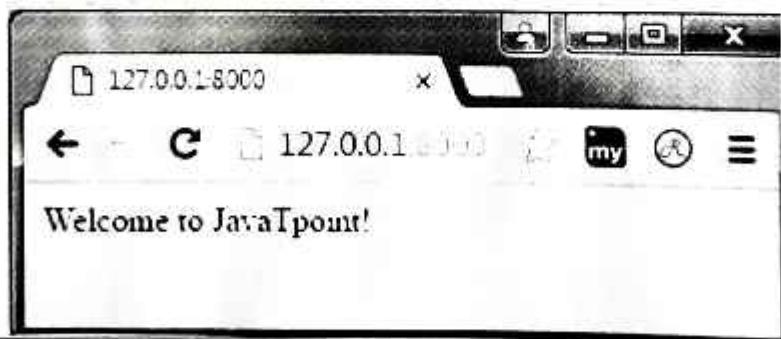
Let's see a basic Express.js app.

File: basic_express.js

```
var express = require('express');
var app = express();
app.get('/', function(req,res){
    res.send('Welcome to JavaTpoint');
});
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);
});
```

The screenshot shows a terminal window titled "Node.js command prompt - node basic_express.js". The terminal displays the following text:
Your environment has been set up for using Node.js 4.4.4 (v64) and npm.
C:\Users\javatpoint\Desktop>cd desktop
C:\Users\javatpoint\Desktop>node basic_express.js
Example app listening at http://:::8000

OUTPUT



Install Express.js

Firstly, you have to install the express framework globally to create Web applications use Node terminal. Use the following command to install express framework globally.

```
npm install -g express
```

```

Node.js command prompt
C:\Users\Aman\Documents\Node.js\Express>express
express@4.13.1 node modules\express
├── accepts@1.3.4
├── array-flatten@1.0.1
├── caseless@1.0.3
├── content-disposition@0.2.3
├── content-type@1.0.4
├── cookie-signature@1.0.6
├── debug@2.6.9
├── encodeurl@1.0.2
└── fresh@0.4.4

  +-- body-parser@1.13.1
  +-- content-type@1.0.2
  +-- cookie-parser@1.4.3
  +-- debug@2.6.9
  +-- fresh@0.4.4
  +-- json-utf8@0.1.0
  +-- method-override@1.0.3
  +-- qs@3.2.6
  +-- type-encoding@1.1.0
  +-- urlencoded@1.0.2
  +-- verify-expired@0.1.0
  +-- www-type@2.1.11

C:\Users\Aman\Documents\Node.js\Express>

```

Installing Express

Use the following Command to install express:

`npm install express --save`

```

Select Node.js command prompt
C:\Users\Aman\Documents\Node.js\Express>npm install express --save
express@4.13.1 node modules\express
├── accepts@1.3.4
├── array-flatten@1.0.1
├── caseless@1.0.3
├── content-disposition@0.2.3
├── content-type@1.0.4
├── cookie-signature@1.0.6
├── debug@2.6.9
├── encodeurl@1.0.2
└── fresh@0.4.4

  +-- body-parser@1.13.1
  +-- content-type@1.0.2
  +-- cookie-parser@1.4.3
  +-- debug@2.6.9
  +-- fresh@0.4.4
  +-- json-utf8@0.1.0
  +-- method-override@1.0.3
  +-- qs@3.2.6
  +-- type-encoding@1.1.0
  +-- urlencoded@1.0.2
  +-- verify-expired@0.1.0
  +-- www-type@2.1.11

C:\Users\Aman\Documents\Node.js\Express>

```

The above command install express in node module director, and Create a directory named express inside the node-module. You should install some other important modules along with the express. Following is the list:

- body-parser:

This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data.

- Cookie-parser:

It is used to parse Cookie header and populate req.cookies with an object keyed by the cookie names.

- multer:

This is a node.js middleware for handling multipart/form data.

`npm install body-parser -- save`

```
C:\Users\javatpoint1>npm install body-parser -- save
body-parser@1.19.1 node modules\body-parser
  | content-type@1.1.2
  | http@0.13.0
  | dep@1.1.0
  | qs@6.5.0
  | on-finished@0.1.0 (ee-first@1.1.0)
  | iconv-lite@0.4.13
  | raw-body@2.1.6 (unpipe@1.0.0)
  | debug@2.2.0 (ms@0.7.1)
  | http-errors@1.4.0 (inherits@0.8.1, statuses@1.3.0,
  | type-is@1.6.13 (media-type@0.3.0, mime-type@2.1.1))
```

`npm install cookie-parser -- save`

```
C:\Users\javatpoint1>npm install cookie-parser -- save
cookie-parser@1.4.2 node modules\cookie-parser
  | cookie-signature@1.0.6
  | cookie@0.2.4
```

`npm install multer -- save`

```
C:\Users\javatpoint1>npm install multer -- save
multer@1.1.0 node modules\multer
  | object-assign@3.0.0
  | xtend@4.0.1
  | append-field@0.1.0
  | on-finished@0.3.0 (ee-first@1.1.0)
  | type-is@1.6.13 (media-type@0.3.0, mime-type@2.1.1))
  | mkdirp@0.5.4 (minimist@0.0.8)
  | concat-stream@1.5.1 (inherits@0.8.1, typedarray@0.0.6, readable-stream@2.0.6)
  | busboy@0.2.13 (readable-stream@1.1.14, dicer@0.2.5)
```

Express.js App Example

Lets take a Simple Express app example which Starts a server and listen on a local port. It only responds to homepage For every other path, it will respond with a 404 Not Found error.

File: express_example.js

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

```
Var app = express();
```

```
App.get('/', function(req, res) {
```

```
    res.send('Welcome to JavaTpoint');
```

```
}
```

```
Var server = app.listen(8000, function() {
```

```
Var host = Server.address().address
```

```
Var port = Server.address().port
```

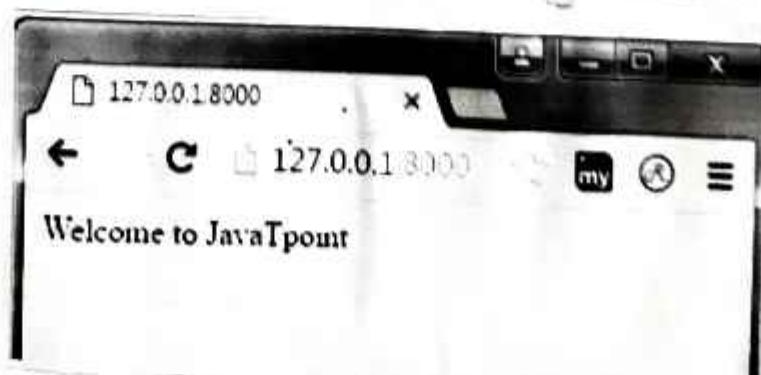
```
Console.log("Example app listening at http://%s.%s,"
```

```
host, port)
```

```
)
```



Open <http://127.0.0.1:8000> in your browser to see the result.



Express.js Request Object

Express.js Request and Response objects are the parameters of the callback function which is used in Express applications.

The express.js request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.

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

Express.js Request Object Properties

The following table specifies some of the properties associated with request object.

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 properties 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 address 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 contain 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 represent an array of subdomains 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

Following is a list of some generally used 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.

Examples:

```
req.accepts('html');
//=>? html?
```

```
req.accepts('text/html');
//=>? Text/html?
```

`req.get(field)`

This method returns the specified HTTP request header field.

Examples:

```
req.get('Content-Type');
//=> "text/plain"
```

```
req.get('content-type');
//=> "text/plain"
```

```
req.get('Something');
//=> undefined
```

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.

Examples:

```
// With Content - Type: text/html; charset = utf-8  
req.is('html');  
req.is('text/html');  
req.is('text/*');  
//=> true
```

req.param(name[, defaultValue])

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

Examples:

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

Express.js Response Object

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

What it does

- It sends response back to the Client browser.
- It facilitates you to put new cookies value and that will write to the client browser (under cross domain rule).
- Once you res.send() or res.direct() or res.render(), you cannot do it again, otherwise, there will be uncaught error.

Response Object Properties

Let's see some properties of response object:

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.

Response Object Methods

Following are some methods:

Response Append method

res.append(Field[, Value])

This method appends the specified value to the HTTP response header field. That means if the specified value is not appropriate then this method redress that.

Examples:

```
res.append('Link', [<http://localhost/>, <http://localhost:3000/>]);
```

```
res.append('Warning', '199 Miscellaneous Warning');
```

Response Attachment method

```
res.attachment([filename])
```

This method facilitates you to send a file as an attachment in the HTTP response.

Examples:

```
res.attachment('path/to/js-pic.png');
```

Response Cookie method

```
res.cookie(name, value[, options])
```

This method is used to set a cookie name of value. The value can be string or object converted to JSON.

Examples:

```
res.cookie('name', 'Aryan', {domain: 'xyz.com', path: '/admin', secure: true});
```

```
res.cookie('Section', {Names: [Aryan, Sushil, Priyanka]});
```

```
res.cookie('Cart', {Items: [1, 2, 3]}, {maxAge: 900000});
```

Response clearcookie method

```
res.clearCookie(name[, options])
```

As the name specifies, the clearCookie method is used to clear the cookie specified by name.

Examples:

To set a cookie

```
res.cookie('name', 'Aryan', {path: '/admin'});
```

To clear a cookie

```
res.clearCookie('name', {path: '/admin'});
```

Response Download method

```
res.download(path[, filename][, fn])
```

this method transfers the file at path as an "attachment" and enforces the browser to prompt user for download.

Example:

```
res.download('/report-12345.pdf');
```

Response End method

```
res.end([data][,encoding])
```

This method is used to end the response process

Example:

```
res.end();
```

```
res.status(404).end();
```

Response Format method

```
res.format(object)
```

This method performs content negotiation on the Accept HTTP header on the request object. When present,

Example:

```
res.format({  
  'text/plain': function() {  
    res.send('hey');  
  },  
  'text/html': function() {  
    res.send(  
      'hey');  
  },  
  'application/json': function() {  
    res.send({ message: 'hey' });  
  },  
  'default': function() {  
    // log the request and respond with 406  
    res.status(406).send('Not Acceptable');  
  }  
});
```

Response Get method

res.get(field)

This method provides HTTP response header specified by field.

Example:

```
res.get('Content-Type');
```

Response JSON method:

res.json([body])

This method returns the response in JSON format.

Example:

`res.json(null)`

`res.json({ name: 'ajeet' })`

Response JSONP method

`res.jsonp([body])`

This method returns response in JSON format with JSONP support.

Examples:

`res.jsonp(null)`

`res.jsonp({ name: 'ajeet' })`

Response Links method

`res.links(links)`

This method populates the response's Link HTTP header field by joining the links provided as properties of the parameter.

Examples:

`res.links({`

`next: 'http://api.rnd.com/users?page=5',`

`last : 'http://api.rnd.com/users?page=10';`

`});`

Response Location method

`res.location(path)`

This method is used to set the response location HTTP header field based on the specified path parameter.

Examples:

`res.location('http://xyz.com');`

Response Redirect method

`res.redirect([status], path)`

This method redirects to the URL derived from the specified path, with specified HTTP status.

Examples:

`res.redirect('http://example.com');`

Response Render method

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

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

Examples:

// 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){`

`//...`

`});`

Response Send method

`res.send([body])`

This method is used to send HTTP response.

Examples:

```
res.send(new Buffer('Whoop'));
res.send({some:'json'});
res.send(
  ....
  some html
);
```

Response sendFile method

res.sendFile(path[options][fn])

This method is used to transfer the file at the given path. It sets the Content-Type response HTTP header field based on the filename's extension.

Examples:

```
res.sendFile(fileName, options, function(err){
  //...
});
```

Response Set method

res.set(field,value)

This method is used to set the response of HTTP header field to value.

Examples:

```
res.set('Content-Type', 'text/plain');
res.set({
  'Content-Type': 'text/plain',
  'Content-Length': '123',
})
```

Response Status method

res.status(code)

This method sets an HTTP status for the response.

Examples:

`res.status(403).end();`

`res.status(400).Send('Bad Request');`

Response Type method

res.type(type)

This method sets the content-type HTTP header to the MIME type.

Examples:

`res.type('html');` //=> 'text/html'

`res.type('html');` //=> 'text/html'

`res.type('json');` //=> 'application/json'

`res.type('application/json');` //=> 'application/json'

`res.type('png');` //=> image/png :

Express.js GET Request

GET and POST both are two common HTTP requests used for building REST API's. GET requests are used to send only limited amount of data because data is sent in the header while POST requests are used to send large amount of data because data is sent in the body.

Express.js facilitates you to handle GET and

POST request Using the instance of express.

Express.js GET Method Example 1

Fetch data in JSON format

Get method facilitates you to send only limited amount of data because data is sent in the header. It is not secure because data is visible in URL bar.

Let's take an example to demonstrate GET method

File : index.htm

```
<html>
<body>
<form action = "http://127.0.0.1:8081/process_get" method = "GET">
First Name : <input type = "text" name = "first_name"><br>
Last Name : <input type = "text" name = "last_name">
<input type = "submit" value = "Submit">
</form>
</body>
</html>
```

file : get example1.js

```
Var express = require('express');
Var app = express();
app.use(express.static('public'));
app.get('/index.html', function(req, res) {
  res.sendFile(__dirname + "/" + "index.html");
})
app.get('/process_get', function(req, res) {
  response = {
    first_name : req.query.first_name,
```

```
last_name: req.query.last_name
});
```

```
console.log(response);
```

```
res.end(JSON.stringify(response));
```

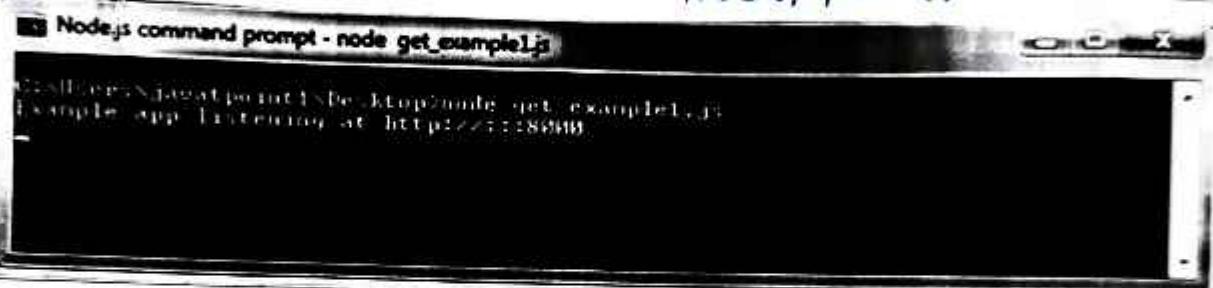
3)

```
Var Server = app.listen(8000, function() {
```

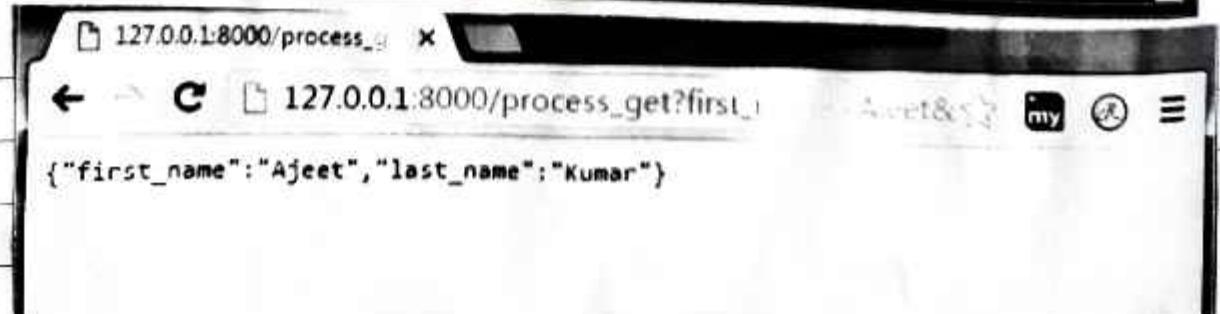
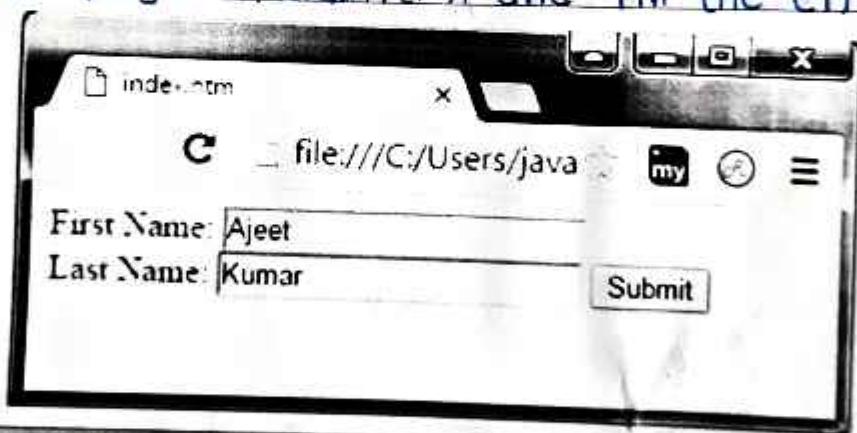
```
Var host = Server.address().address
```

```
Var port = Server.address().port
```

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



Open the page index.html and fill the entries



Express.js GET Method Example 2

Fetch data in paragraph format

File: index.html

```
<html>
```

```
<body>
```

```
<form action="http://127.0.0.1:8000/get_example2"
      method="GET">
```

First name: <input type="text" name="first_name" />

Last name: <input type="text" name="last_name" />


```
<input type="Submit" value="submit" />
```

```
</form>
```

```
</body>
```

```
</html>
```

File: get_example2.js

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

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

```
app.get('/get_example2', function(req, res) {
```

```
    res.send('<p> Username:' + req.query.username + '
```

```
    <p> Lastname: ' + req.query.last_name + '</p>');
```

```
}
```

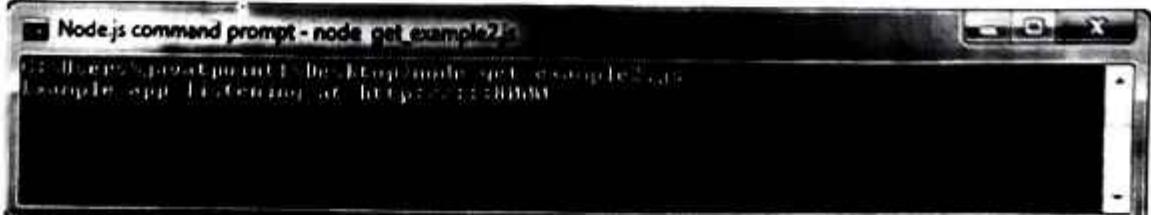
```
var server = app.listen(8000, function() {
```

```
    var host = server.address().address
```

```
    var port = server.address().port
```

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

```
})
```



Open the page index.htm and fill the entries:

index.htm

C file:///C:/Users/javatpoint1

First Name: Rahul
Last Name: Kumar
Submit

Output:

127.0.0.1:8000/get_exampl

← C 127.0.0.1:8000/get_exampl

Username: Rahul
Lastname: Kumar

Express.js GET Method Example 3

File: index.htm

```
<!DOCTYPE html>
<html>
<body>
<form action = "http://127.0.0.1:8000/get_example3">
<table>
<tr><td> Enter First Name:</td> <td> <input type = "text" name = "firstname"/> <td> </tr>
<tr><td> Enter Last Name:</td> <td> <input type = "text" name = "lastname"/> <td> </tr>
<tr><td> Enter Password:</td> <td> <input type = "password" name = "password"/> <td> </tr>
<tr><td> Sex:</td> <td>
<input type = "radio" name = "sex" value = "male"> Male
<input type = "radio" name = "sex" value = "female"> Female
</td> </tr>
<tr><td> About You:</td> <td>
```

```

<textarea rows="5" cols="40" name="aboutyou"
    place holder="Write about yourself">
</textarea>
</td> </tr>
<tr><td colspan="2"><input type="Submit"
    value="register"/><br/></td></tr>
</table>
</form>
</body>
</html>

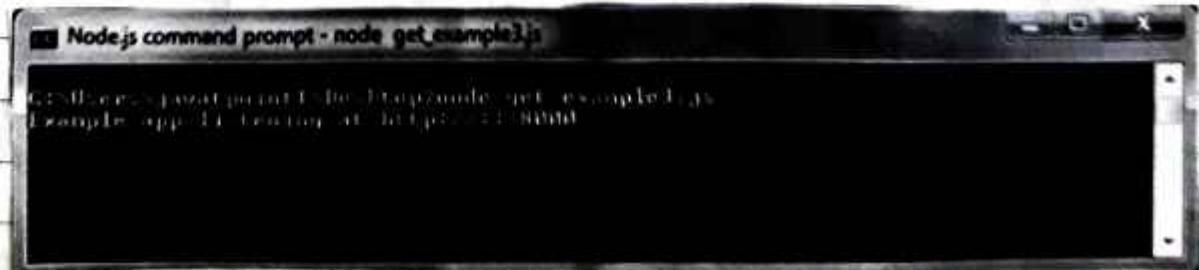
```

File : get_example3.js

```

var express = require('express');
var app = express();
app.get('/get_example3' function(req, res) {
    res.send('<p>Firstname:' + req.query['firstname'] + '</p>
<p>Lastname:' + req.query['lastname'] + '</p> <p>password
:' + req.query['password'] + '<p>
<p>About You : ' + req.query['aboutyou'] + '</p>');
}
)
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)
}
)

```



Open the page index.html and fill the entries:

A screenshot of a web browser window titled "index.html". The URL bar shows "file:///C:/Users/javatpoint1/Desktop". The page contains a form with fields for First Name (John), Last Name (Alex), Password (redacted), and Sex (Male selected). Below the form is a text area with the message: "Hi... My name is John. I live in the beautiful city of Paris." A "register" button is at the bottom.

A screenshot of a terminal window titled "127.0.0.1:8000 get_error". The URL bar shows "127.0.0.1". The terminal output shows the raw data sent in a POST request:

```
Firstname John
Lastname Alex
Password alexander
AboutYou Hi. My name is John. I live in the beautiful city of Paris.
```

Express.js POST Request

GET and POST both are two common HTTP requests used for building REST API's. POST requests are used to send large amount of data.

Express.js facilitates you to handle GET and POST requests using the instance of express.

Express.js POST Method

Post method facilitates you to send large amount of data because data is send in the body. Post method is secure because data is not visible in URL bar but is not used as popularly as GET method. On the other hand GET method is more efficient and used more than POST.

Let's take an example to demonstrate POST method

Example - 1

Fetch data in JSON format

File: Index.html

```
<html>
```

```
<body>
```

```
<form action = "http://127.0.0.1:8000/process_post"
      method = "POST">
```

```
First Name: <input type = "text" name = "first_name"><br>
```

```
Last Name: <input type = "text" name = "last_name">
```

```
<input type = "Submit" value = "Submit">
```

```
</form>
```

```
</body>
```

```
</html>
```

File: post_example1.js

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

```
Var app = express();
```

```
Var bodyParser = require('body-parser');
```

```
//Create application/x-www-form-urlencoded parser
```

```
Var urlencodedParser = bodyParser.urlencoded({extended: false})
```

```
app.use(express.static('public'));
```

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

```
    res.sendFile(__dirname + "/" + "index.html");
```

```
}
```

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

```
//Prepare output in JSON format
```

```
response = {
```

```
    first_name: req.body.first_name,
```

```
    last_name: req.body.last_name
```

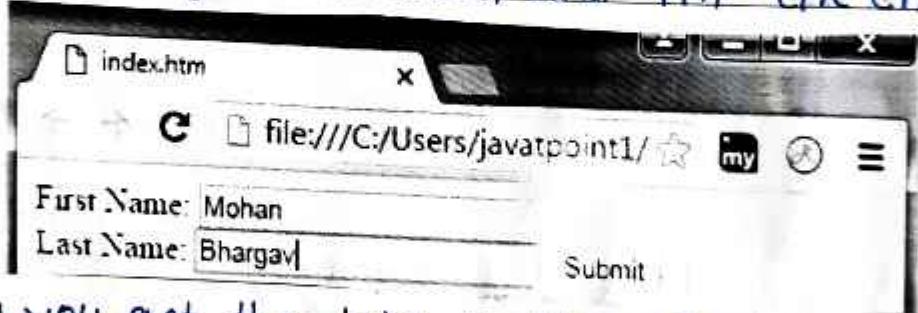
```
};
```

```
    console.log(response);
    res.end(JSON.stringify(response));
}
```

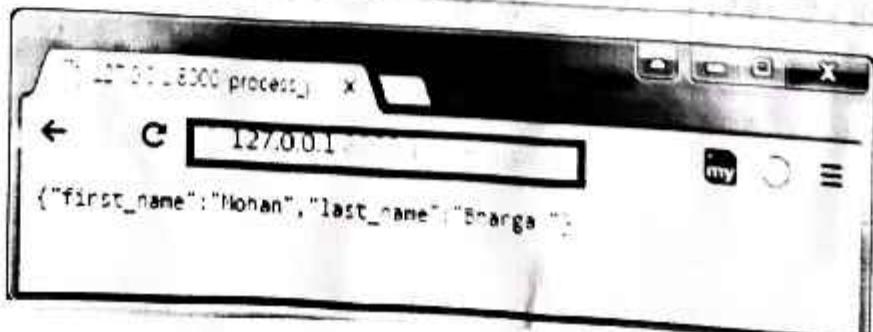
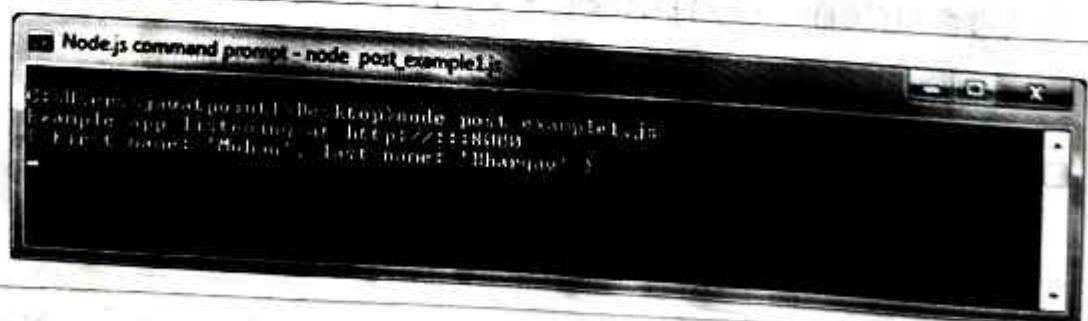
```
var Server = app.listen(8000, function() {
  var host = Server.address().address;
  var port = Server.address().port;
  console.log("Example app listening at http://%s:%s", host, port);
})
```



Open the page index.html and fill the entries:



Now, you get the data in JSON format.



Express.js Routing

Routing is made from the word route. It is used to determine the specific behavior of an application. It specifies how an application responds a client request to a particular route, URI or path and a specific HTTP request method (GET, POST etc.). It can handle different types of HTTP requests.

Let's take an example to see basic routing.

File: routing_example.js

```

var express = require('express');
var express = require('express');
var app = express();
app.get('/', function(req, res) {
    console.log("Got a GET request for the homepage");
    res.send('Welcome to JavaTpoint!');
})
app.post('/', function(req, res) {
    console.log("Got a POST request for the homepage");
    res.send('I am Impossible');
})
app.delete('/del_student', function(req, res) {
    console.log("Got a DELETE request for /del_student");
    res.send('I am deleted!');
})
app.get('/enrolled_student', function(req, res) {
    console.log("Got a GET requests for /enrolled_student");
    res.send('I am an enrolled student.');
})
// This responds a GET requests for abcd, abxcd, ab123cd, and so on
app.get('/ab*cd', function(req, res) {
    console.log("GOT a GET request for /ab*cd");
})

```

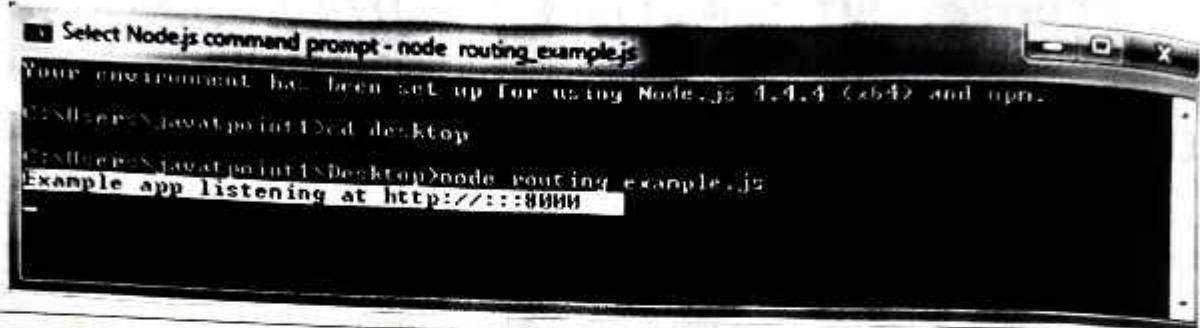
res.send('Pattern Matched');
 3)

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);
 3)

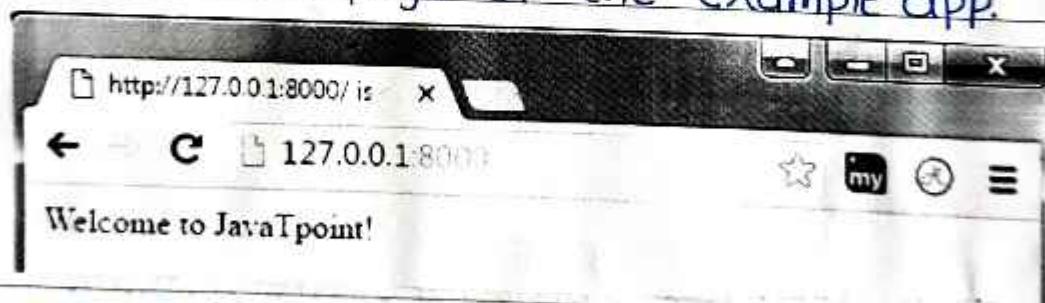


You see that server is listening.

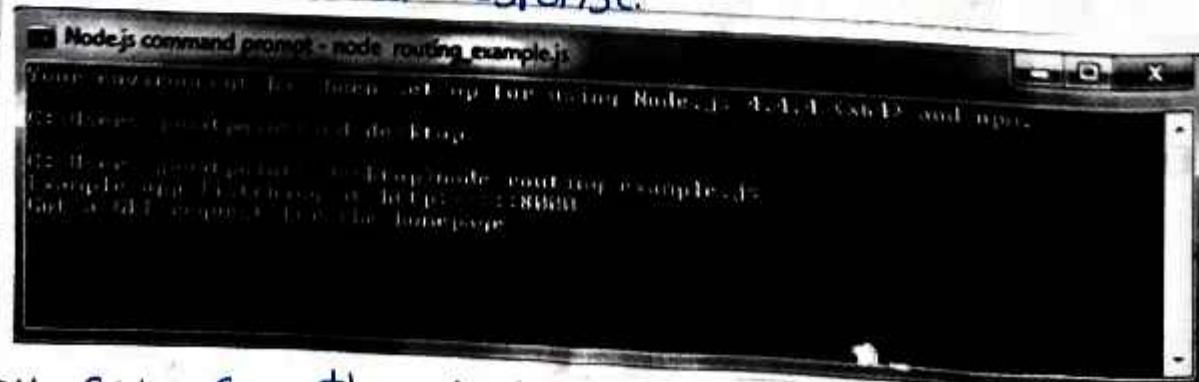
Now, you can see the result generated by server
on the local host <http://127.0.0.1:8000>

Output:

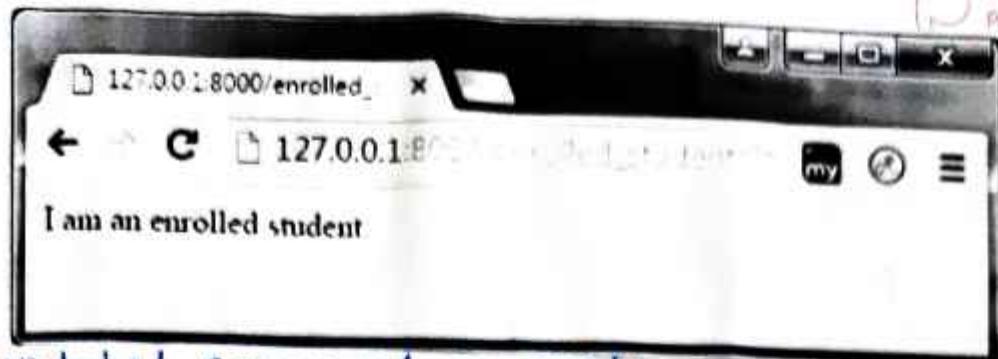
This is the home page of the example app.



Note: The Command Prompt Will be updated after
one sucessful response.



You can see the different pages by changing routes
http://127.0.0.1:8000/enrolled_student



updated command prompt:

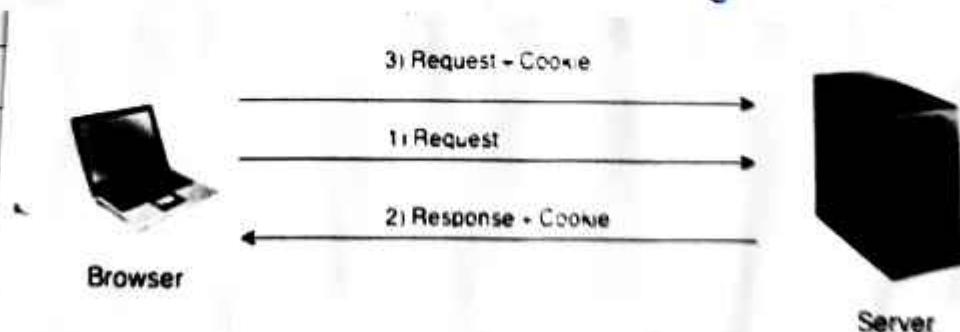
```
Node.js command prompt - node routing_example.js
Your environment has been set up for using Node.js 1.1.1 (v61) and npm
Version 1.1.1 (v61) on your local desktop.

C:\Users\guptapratim\Desktop\node-routing-example.js
Example app listening at http://127.0.0.1:8000
Get a GET request for the homepage
Get a GET request for /enrolled_student
```

Expressjs Cookies Management

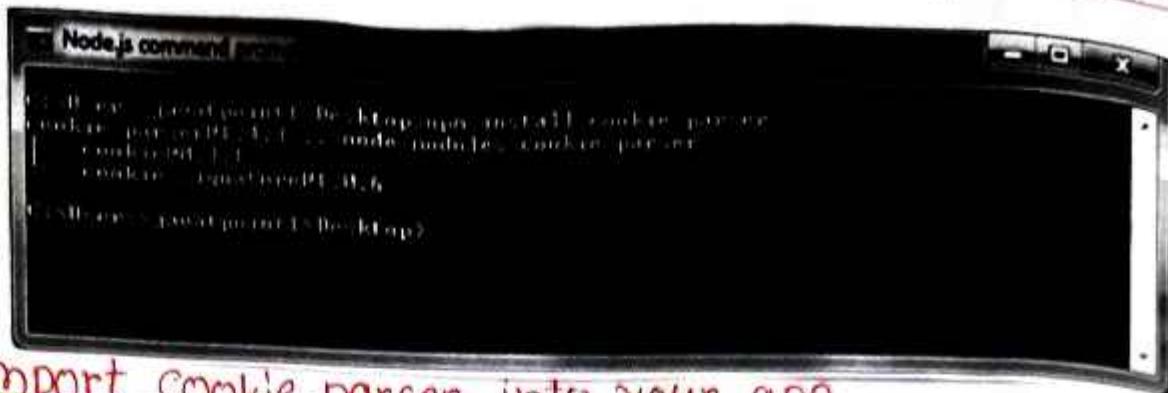
What are Cookies

Cookies are small piece of information i.e. sent from a Website and stored in user's Web browser when user browser that Website. Every time the user loads that Website back, the browser sends that stored data back to Website or Server, to recognize user.



Install cookie

You have to acquire Cookies abilities in Express.js so, install Cookie parser middleware through npm by using the following Command:



Import cookie-parser into your app.

```
Var express = require('express');
VarCookieParser = require ('cookie-parser');
Var app = express();
App.use(cookieParser());
```

Define a route:

Cookie-parser parses cookie header and populate req.cookies with an object keyed by the cookie names.
 Let's define a new route in your express app like set a new cookie:

```
App.get('/cookie', function(req, res) {
  res.cookie('cookie_name', 'cookie_value').send('Cookie is set');
});

App.get('/', function(req, res) {
  console.log("Cookies: ", req.cookies);
});
```

Browser sends back that cookie to the server, every time when it requests that website.

Express.js Cookies Example

File: Cookies_example.js

```
Var express = require('express');
VarCookieParser = require ('cookie-parser');
Var app = express();
App.use(cookieParser());
```

```

app.get('/cookieset', function(req, res) {
  res.cookie('cookie_name', 'cookie_value');
  res.cookie('Company', 'javatpoint');
  res.cookie('Name', 'sonoo');
  res.status(200).send('cookie is set');
});

app.get('/', function(req, res) {
  res.status(200).send('Welcome to JavaTpoint');
});

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);
});

```

```

Node.js command prompt - node cookies_example.js
C:\Users\sonoo\Desktop>node cookies_example.js
Example app listening at http://:::8000

```

Output:

Open the page <http://127.0.0.1:8000> on your browser:

Set cookie:

Now open <http://127.0.0.1:8000/cookieset> to see the set cookie:



Get Cookie:

Now open <http://127.0.0.1:8000/cookieget> to get the cookie:

```
{"cookie_name": "cookie_blue", "company": "jastpoint", "name": "sonoo"}
```

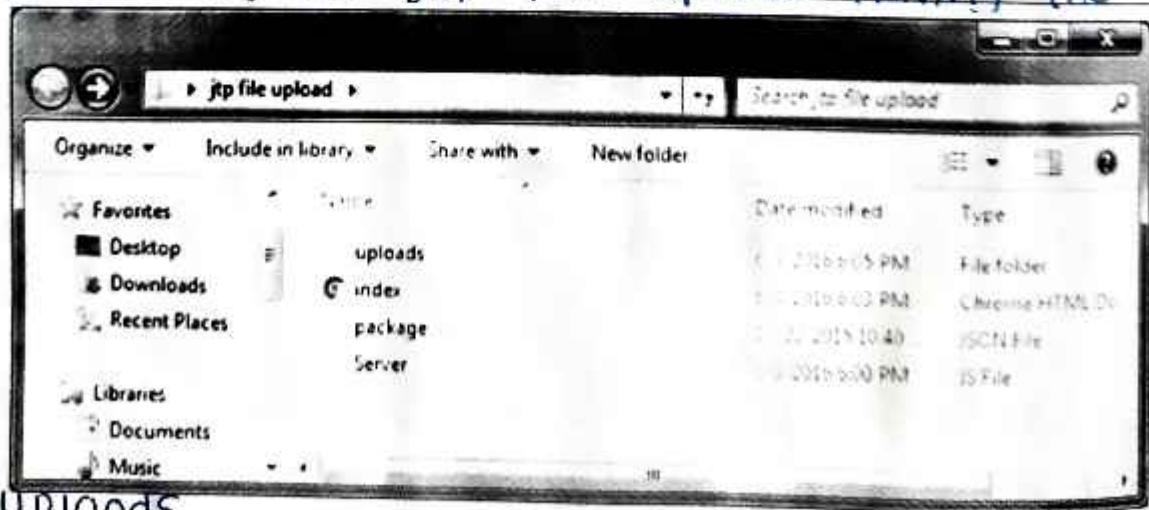
Express.js File Upload

In Express.js, file upload is slightly difficult because of its asynchronous nature and networking approach.

It can be done by using middleware to handle multipart/form data. There are many middleware that can be used like multer, connect, body-parser etc.

Take take an example to demonstrate file upload in Node.js. Here, we are using the middleware 'multer'.

Create a folder "jtp file upload" having the



uploads.

It is an empty folder i.e. created to store the uploaded images.

Package:

It is JSON file, having the following data:

File: package.json

{

```
"name": "file_upload",
"version": "0.0.1",
"dependencies": {
    "express": "4.13.3",
    "multer": "1.1.0"
}
```

3.

```
"devDependencies": {
    "Should": "~7.1.0",
    "mocha": "~2.3.3",
    "Supertest": "~1.1.0"
}
```

3

File: index.html

```
<html>
  <head>
    <title>File upload in Node.js by JavaTpoint</title>
    <script src = "http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/
      jquery.min.js"></script>
    <script src = "http://cdnjs.cloudflare.com/ajax/libs/jquery.
      form/3.51/jquery.form.min.js"></script>
  <Script>
    $(document).ready(function() {
      $('#uploadForm').submit(function() {
        $('#status').empty().text("File is uploading... ");
        $(this).ajaxSubmit({
          error: function(xhr) {

```

```
    status('Error:' + xhr.status);
},
success: function(response) {
    console.log(response)
    $('#status').empty().text(response);
}
});
return false;
});
});
</script>
</head>
<body>
<h1>Express.js File upload by JavaTpoint </h1>
<form id="uploadform"
data action="/uploadjavatpoint" method="post">
<input type="file" name="mylife"/><br/><br/>
<input type="Submit" value="Upload Image"
name="Submit"><br/><br/>
<span id="status"></span>
</form>
</body>
</html>
```

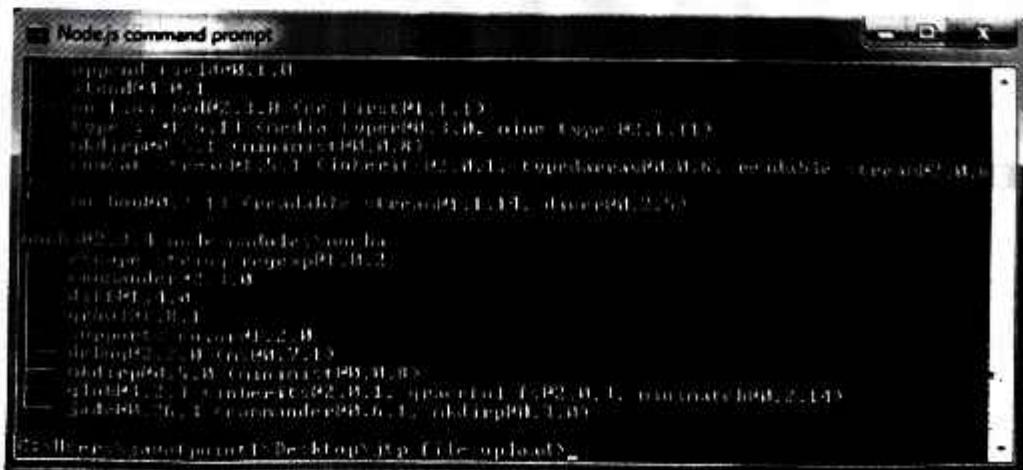
File: server.js

```
var express = require("express");
var multer = require('multer');
var app = express();
var storage = multer.diskStorage({
  destination: function(req, file, callback) {
    callback(null, './uploads');
  }
});
```

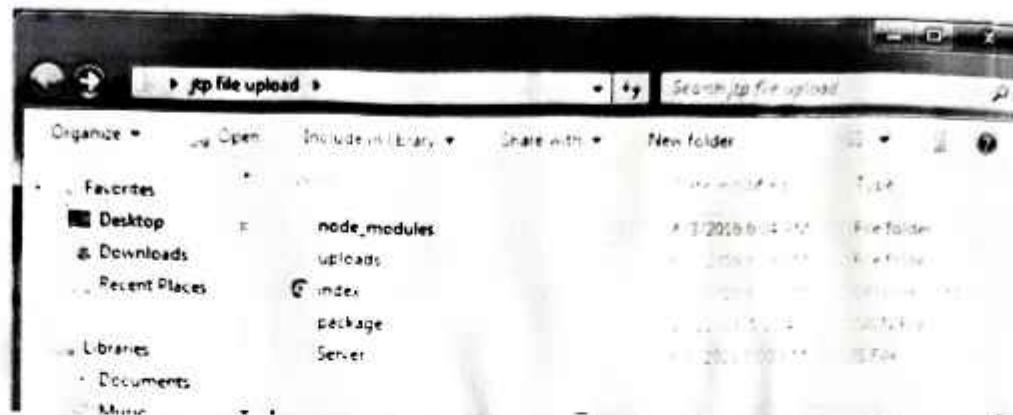
```
filename: function(req, file, callback) {
    callback(null, file.originalname);
}
});

var upload = multer({storage:Storage}).single('myfile');
app.get('/', function(req, res) {
    res.sendFile(__dirname + '/index.html');
});
app.post('/uploadingjavatpoint', function(req, res) {
    upload(req, res, function(err) {
        if(err) {
            return res.end("Error uploading file");
        }
        res.send("File is uploaded sucessfully!");
    });
});
app.listen(2000, function() {
    console.log('Server is running on port 2000');
});
```

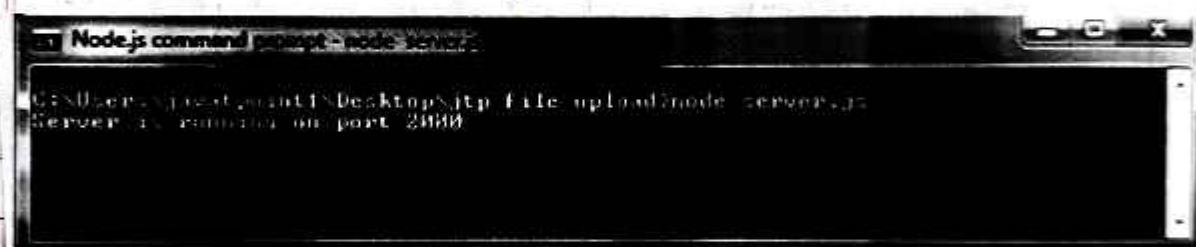
To install the package.json, execute the following code:
`npm install`



It will create a new folder "node_modules" inside the "Jtp file upload" folder.



Dependencies are installed. Now, run the server: node server.js



Open the local page <http://127.0.0.1:2000/> to upload the images.



Select an image to upload and click on "Upload Image" button.



Here, you see that file is uploaded successfully. You can see uploaded file in the 'Uploads' folder.



Express.js Middleware

Express.js Middleware are 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.

What Is a Middleware function

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

A middleware function can perform the following tasks:

- 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.

Express.js Middleware

Following is a list of possibly used middleware in Express.js app:

- Application-level middleware
- Router-level middleware
- Error-handling middleware
- Built-in-middleware
- Third-party middleware

Let's take an example to understand what middleware is and how it works.

Let's take the most basic Express.js app:

File: simple_express.js

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

```
Var app = express();
```

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

```
    res.send('Welcome to JavaTpoint');
```

});

```
app.get('/help', function(req, res) {
  res.send('How can I help you?');
```

});

```
var server = app.listen(8000, function() {
```

```
  var host = server.address().address;
```

```
  var port = server.address().port;
```

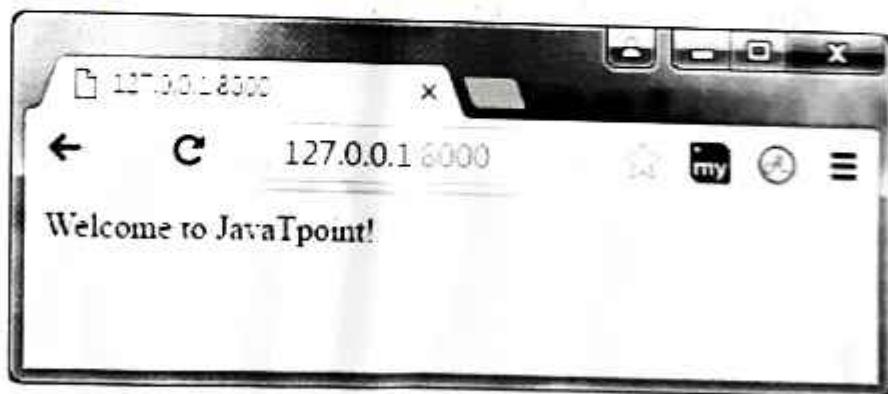
```
  console.log("Example app listening at http://%s:%s", host, port);
});
```

```
Select Node.js command prompt - node simple_express.js
D:\JavaTpoint\Desktop>node simple_express.js
Example app listening at http://:::8000
```

You see that server is listening.

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

Output:



Let's see the next page: <http://127.0.0.1:8000/help>

Output:

```
Select Node.js command prompt - node simple_express.js
D:\JavaTpoint\Desktop>node simple_express.js
Example app listening at http://:::8000
```

Node.js command prompt - node simple_middleware.js

```
Condition: > curl -v http://localhost:8000/simple_middleware.js
Example app listening at: http://127.0.0.1:8000
GET /
GET /help
GET /help
```

Note: You see that the Command prompt is not changed. Means, it is not showing any record of the GET request although a GET request is processed in the `http://127.0.0.1:8000/help` page.

Use of Express.js Middleware

If you want to record every time you get a request then you can use a middleware.

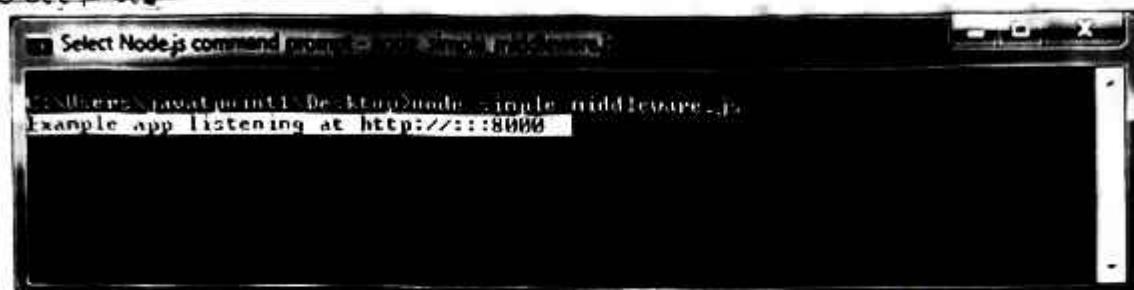
File: Simple_middleware.js

```
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 Java Tpoint');
})
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")
})
```

You see that server is listening.

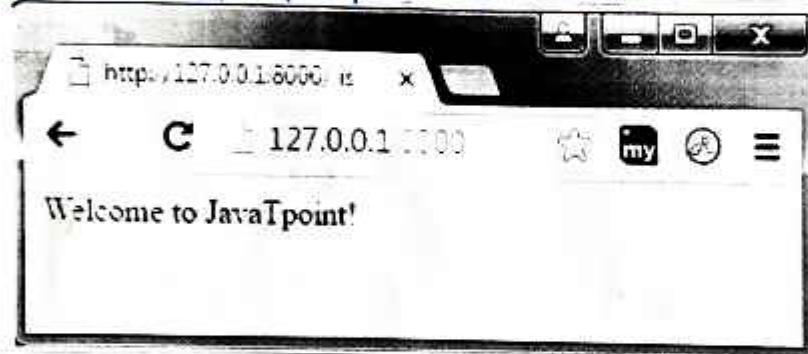
Now, you can see the result generated by server on the local host <http://127.0.0.1:8000>

Output:

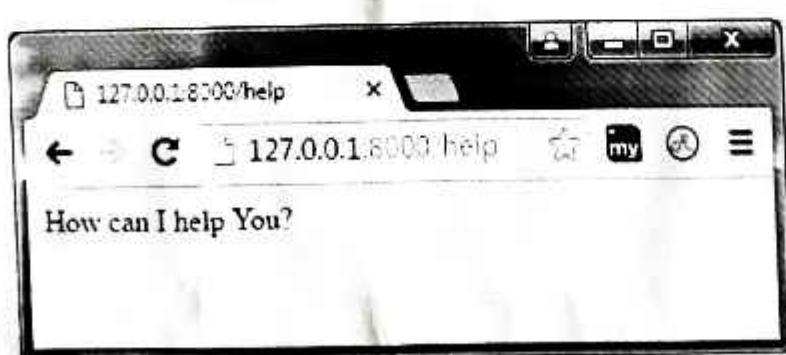


```
Select Node.js command
C:\Users\JavaTpoint\Desktop>node simple_middleware.js
Example app listening at http://127.0.0.1:8000
```

You can see that output is same but command prompt is displaying a GET result.



Go to <http://127.0.0.1:8000/help>




```
Node.js command prompt - node simple_middleware
C:\Users\JavaTpoint\Desktop>node simple_middleware.js
Example app listening at http://127.0.0.1:8000
GET /
GET /help
```

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

A screenshot of a Windows command prompt window titled "Node.js command prompt - node simple_middleware.js". The window shows the command "node simple_middleware.js" being run, followed by the output: "Example app listening at: http://0.0.0.0:8000". Below this, a GET request is shown: "GET /". The window has standard Windows-style borders and a title bar.

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

Middleware example explanation

- In the above middleware example a new function is used to invoke 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.

Express.js Scaffolding

What is scaffolding

Scaffolding is a technique that is supported by some MVC frameworks.

It is mainly supported by the following frameworks: Ruby On Rails, OutSystem Platform, Express Framework, Play Framework, Django, monorail, Brail, Symfony, Laravel, CodeIgniter, Yii, Cake PHP, Phalcon PHP, Model-Glue, PRADO, Grails, Catalyst, Seam Framework, Spring Roo, ASP.NET etc.

Scaffolding facilitates the programmers to specify

how the application data may be used. This specification is used by the frameworks with predefined code templates, to generate the final code that the application can use for CRUD operations (create, read, update and delete database entries).

Express.js scaffold

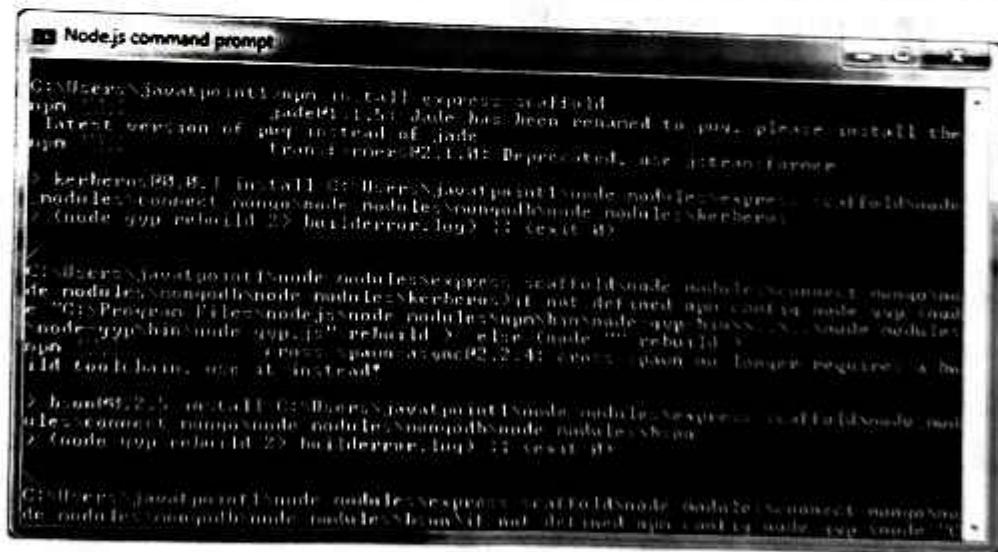
An Express.js Scaffold Supports Candy and more Web projects based on Node.js.

Install Scaffold

Execute the following command to install scaffold.

npm install express - scaffold

It will take a few seconds and the screen will look like this:



After this step, execute the following command to install express generator:

npm install -g express-generator

```

Node.js command prompt
C:\Users\javatpoint\Desktop\myapp> npm init -y express --generator
Creating a new app at C:\Users\javatpoint\Desktop\myapp
This will guide you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.
See "npm help json" for definitive documentation on these fields,
and run "npm help config" for details on how to modify built-in config.

Press ^C at any time to quit.
name: (myapp)_

```

Now, you can use express to scaffold a web-app.

Let's take an example:

First create a directory named myapp. Create a file named app.js in the myapp directory having the following code:

```

var express = require('express');
var app = express();
app.get('/', function(req, res){
    res.send('Welcome to JavaTpoint');
});
app.listen(8000, function(){
    console.log('Example app listening on port 8000!');
});

```

Open Nodejs Command prompt, go to myapp and run npm init command (In my case, I have created myapp folder on desktop)

```

Select name
C:\Users\javatpoint\Desktop\myapp> npm init
This will guide you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.
See "npm help json" for definitive documentation on these fields,
and run "npm help config" for details on how to modify built-in config.

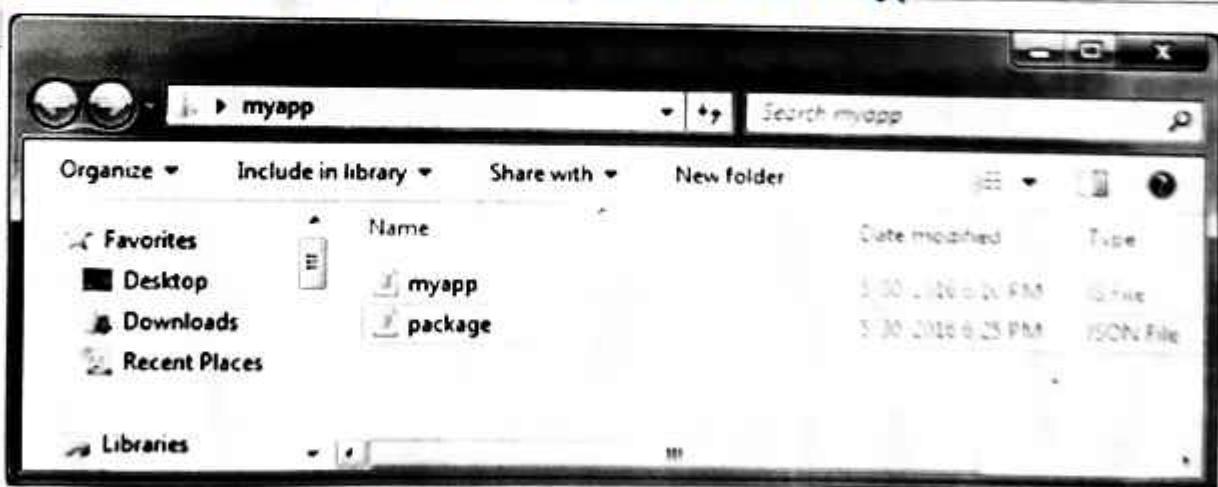
Press ^C at any time to quit.
name: (myapp)_

```

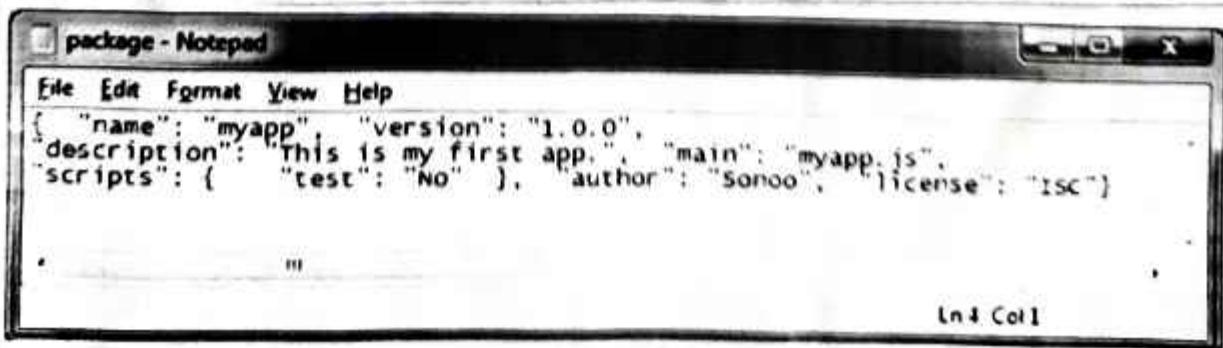
Fill the entries and press enter.



It will create a package.json file in myapp folder and the data is shown in JSON format.



Output:



Using template engines with express

Template engine makes you able to use static template file in your application. To render template files you have to set the following application setting properties:

- Views:

It specifies a directory where the template files are located.

for example: `app.set('views', './views');`

- View engine:

It specifies the template engine that you use. For example, to use the Pug template engine: `app.set('view engine', 'pug');`

Let's take a template engine pug (formerly known as jade).

Pug Template Engine

Let's learn how to use pug template engine in Node.js application using Express.js. Pug is a template engine for Node.js. Pug uses Whitespaces and indentation as the part of the syntax. Its syntax is easy to learn.

Install pug

Execute the following command to install pug template engine:

```
npm install pug -- save
```

Pug template must be written inside .pug file and all .pug files must be put inside views folder in the root folder of Node.js application.

Note: By default Express.js searches all the views in the views folder under the root folder. You can also set to another folder using views property in express. For example: app.set('views', 'myviews')

The pug template engine takes the input a simple way and produce the output in HTML. See how it renders HTML:

Simple input:

doctype html

head

title A simple pug example

body

This page is produced by pug template engine.
Some paragraph here..

Output produced by pug template:

```
<!DOCTYPE html>
<html>
  <head>
    <title>A simple pug example</title>
  </head>
  <body>
    <h1>This page is produced by pug template engine
    </h1>
    <p>Some paragraph here..</p>
  </body>
</html>
```

Express.js can be used with any template engine.
Let's take an example to deploy how pug template creates HTML page dynamically.

See this example :

Create a file named index.pug file inside views folder and write the following pug template init:

doctype html

html

head

title a simple pug example

body

h1 this page is produced by pug template
 engine

p Some paragraph here...

File: Server.js

```
Var express = require('express');
Var app = express();
// set view engine
app.set("view engine", "pug")
app.get('/', function(req, res) {
    res.render('view.pug', index);
    res.render('index')
});
Var server = app.listen(5000, function() {
    console.log('Node server is running..');
});
```