**Express.js Tutorial**

Express.js tutorial provides basic and advanced concepts of Express.js. Our Express.js tutorial is designed for beginners and professionals both.

Express.js is a web framework for Node.js. It is a fast, robust and asynchronous in nature.

Our Express.js tutorial includes all topics of Express.js such as Express.js installation on windows and linux, request object, response object, get method, post method, cookie management, scaffolding, file upload, template etc.

**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 Node.js 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 render 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!');

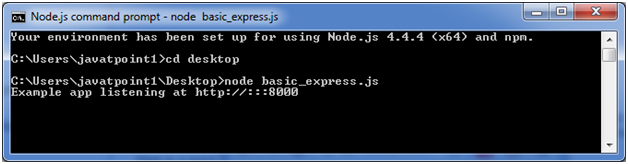
});

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

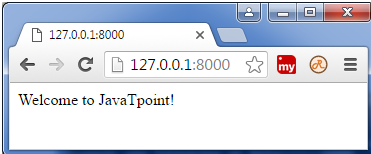
var host = server.address().address;

var port = server.address().port;

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

}); 

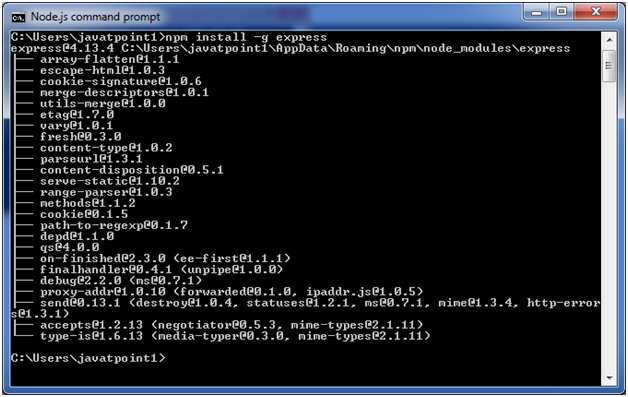
**Output:**

****

**Install Express.js**

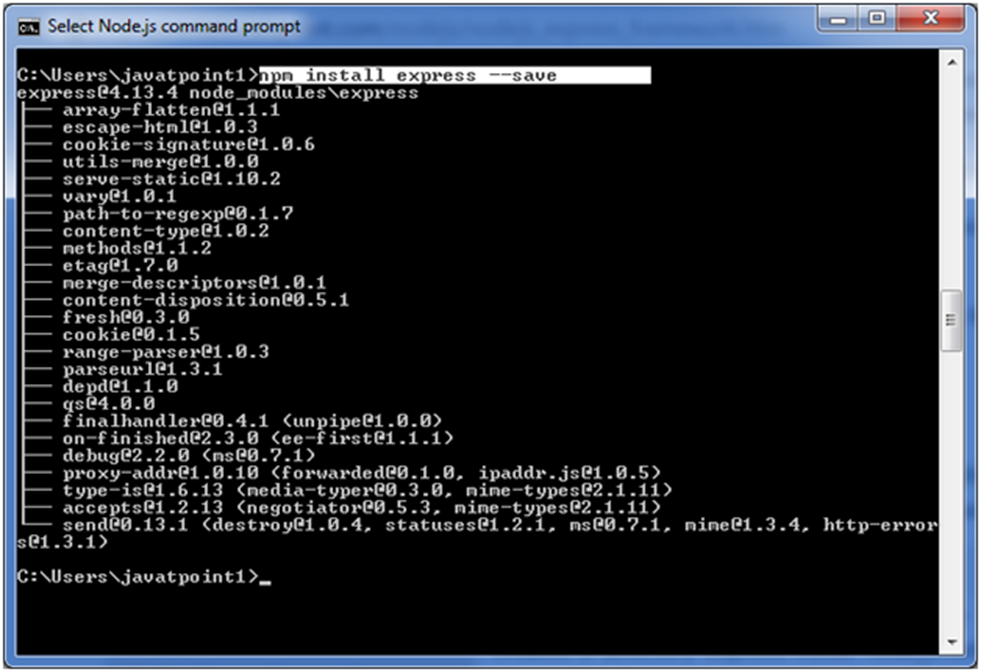
Firstly, you have to install the express framework globally to create web application using Node terminal. Use the following command to install express framework globally.

npm install -g express



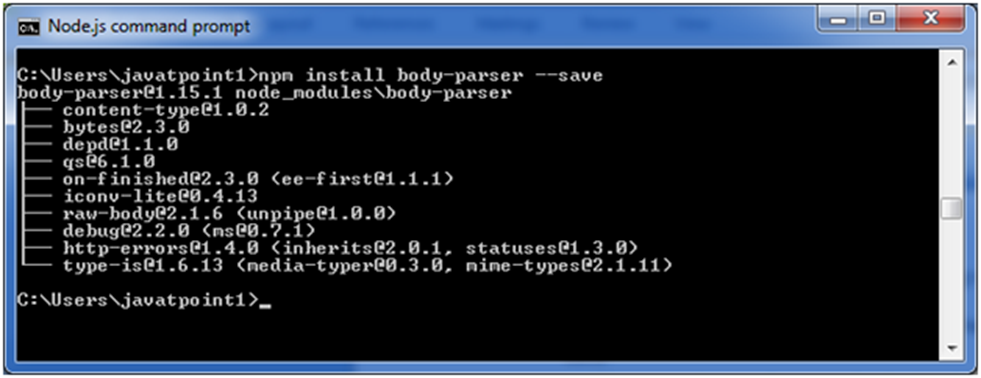
**Installing Express**

Use the following command to install express:

npm install express --save 

The above command install express in node\_module directory and create a directory named express inside the node\_module. You should install some other important modules along with express. Following is the list:

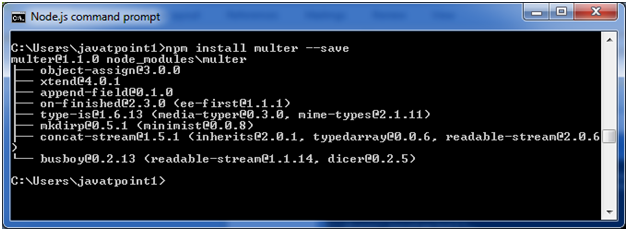
* **body-parser:** This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data.
* **cookie-parser:** It is used to parse Cookie header and populate req.cookies with an object keyed by the cookie names.
* **multer:** This is a node.js middleware for handling multipart/form-data.

npm install body-parser --save 

npm install cookie-parser --save



npm install multer --save



**Express.js App Example**

Let's 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');

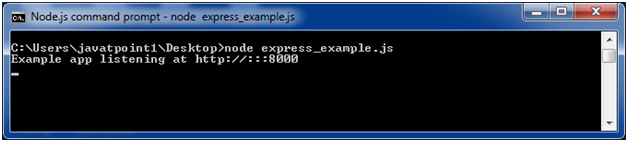
})

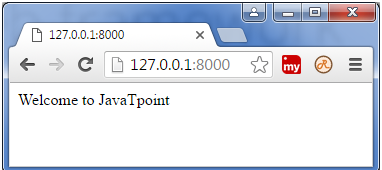
var server = app.listen(8000, function () {

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.

**Syntax:**

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

// --

})

**Express.js Request Object Properties**

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

|  |  |  |
| --- | --- | --- |
| 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**

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.redirect() 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.

|  |  |  |
| --- | --- | --- |
| 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 |

**Response Object Methods**

Following are some methods:

**Response Append method**

**Syntax:**

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**

**Syntax:**

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**

**Syntax:**

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

This method is used to set a cookie name to value. The value can be a 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**

**Syntax:**

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**

**Syntax:**

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**

**Syntax:**

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

This method is used to end the response process.

**Example:**

res.end();

res.status(404).end();

**Response Format method**

**Syntax:**

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**

**Syntax:**

res.get(field)

This method provides HTTP response header specified by field.

**Example:**

res.get('Content-Type');

**Response JSON method:**

**Syntax:**

res.json([body])

This method returns the response in JSON format.

**Example:**

res.json(null)

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

Response JSONP method

**Syntax:**

res.jsonp([body])

This method returns response in JSON format with JSONP support.

**Examples:**

res.jsonp(null)

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

**Response Links method**

**Syntax:**

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**

**Syntax:**

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**

**Syntax:**

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**

**Syntax:**

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**

**Syntax:**

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**

**Syntax:**

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**

**Syntax:**

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**

**Syntax:**

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**

**Syntax:**

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 into 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 requests 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.html**

<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));

})

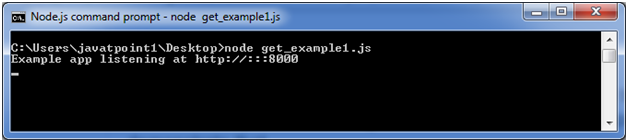
var server = app.listen(8000, function () {

var host = server.address().address

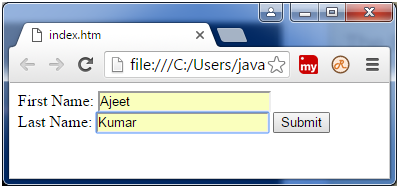
var port = server.address().port

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

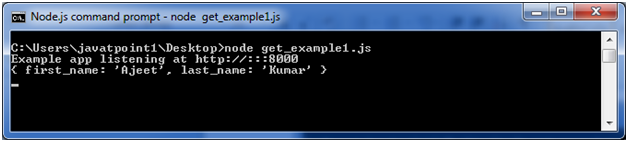
})

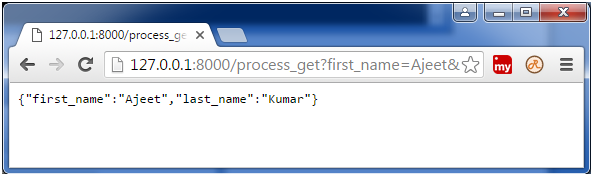


Open the page index.html and fill the entries:



Now, you get the data in JSON format.





**Express.js 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"/> <br/>

Last Name: <input type="text" name="last\_name"/><br/>

<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['first\_name']+'</p><p>Lastname: '+req.query['last\_name']+'</p>');

})

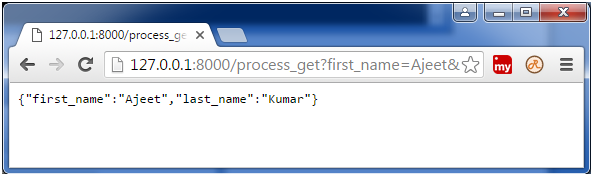
var server = app.listen(8000, function () {

var host = server.address().address

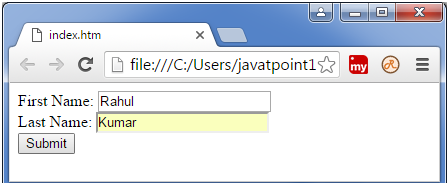
var port = server.address().port

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

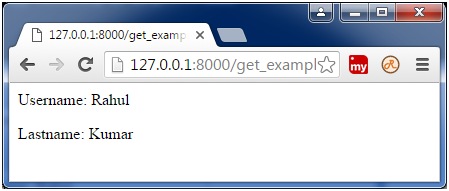
})



Open the page index.html and fill the entries:



Output:



**Express.js GET Method Example 3**

**File:index.html**

<!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" placeholder="Write about yourself">

</textarea>

</td></tr>

<tr><td colspan="2"><input type="submit" value="register"/></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>AboutYou: '+req.query['aboutyou']+'</p>');

})

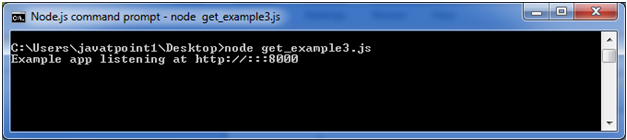
var server = app.listen(8000, function () {

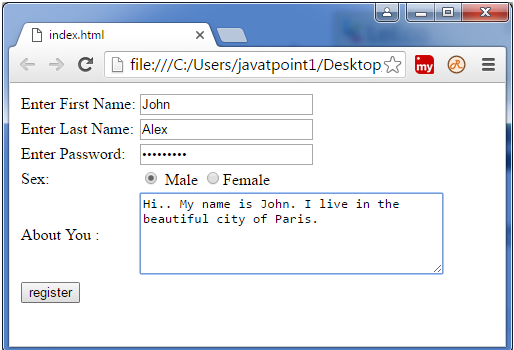
var host = server.address().address

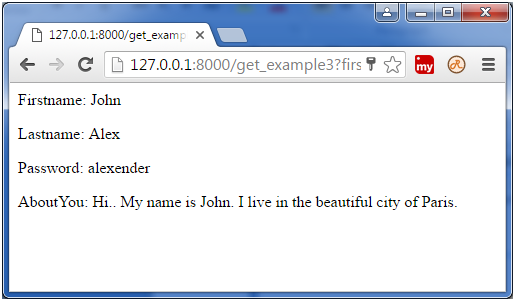
var port = server.address().port

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

})







**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 it 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.

**Example1:**

**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));

})

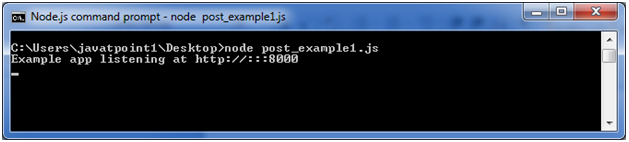
var server = app.listen(8000, function () {

var host = server.address().address

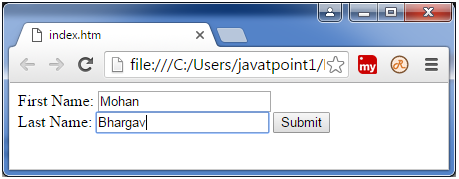
var port = server.address().port

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

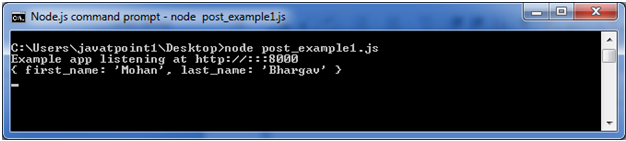
})

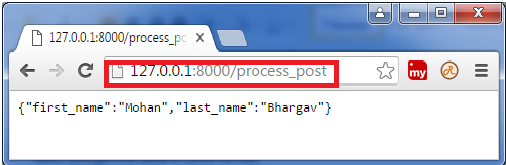


Open the page index.html and fill the entries:



Now, you get the data in JSON format.



Note: In the above picture, you can see that entries are not visible in the URL bar unlike GET method.

**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 to 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 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 request for /enrolled\_student");

res.send('I am an enrolled student.');

})

// This responds a GET request 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.');

})

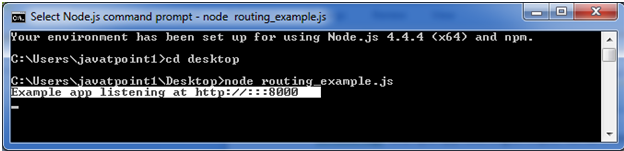
var server = app.listen(8000, function () {

var host = server.address().address

var port = server.address().port

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

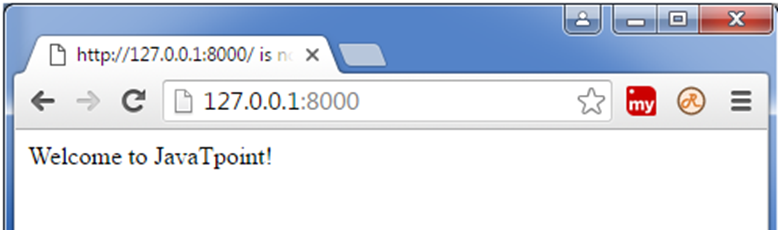
})



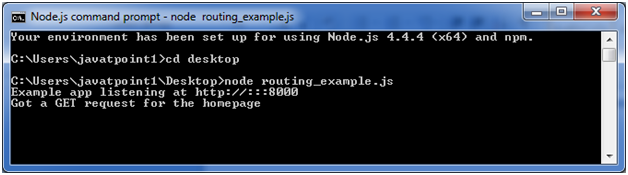
You see that server is listening.

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

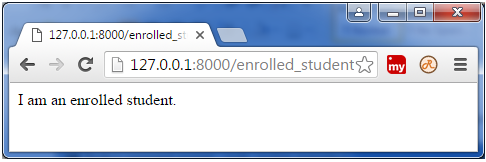
**Output:**

This is the homepage of the example app.

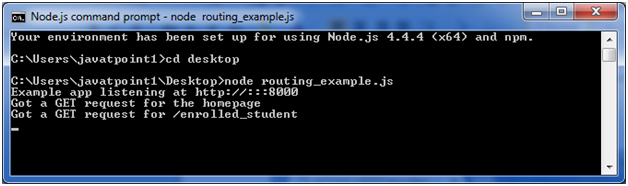
**Note:** The Command Prompt will be updated after one successful response.



You can see the different pages by changing routes**.** [**http://127.0.0.1:8000/enrolled\_student**](http://127.0.0.1:8000/enrolled_student)

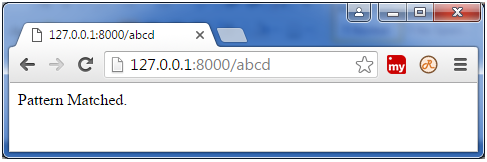


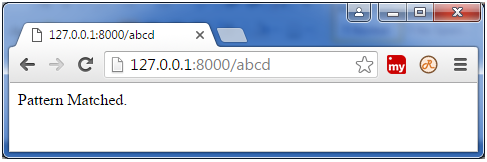
**Updated command prompt:**



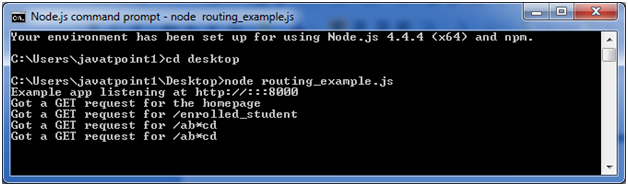
This can read the pattern like abcd, abxcd, ab123cd, and so on.

Next route **http://127.0.0.1:8000/abcd**



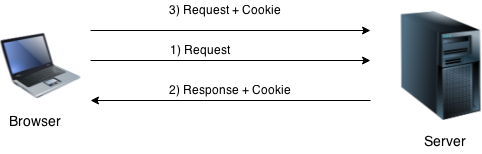
Next route [**http://127.0.0.1:8000/ab12345cd**](http://127.0.0.1:8000/ab12345cd)****

**Updated command prompt:**

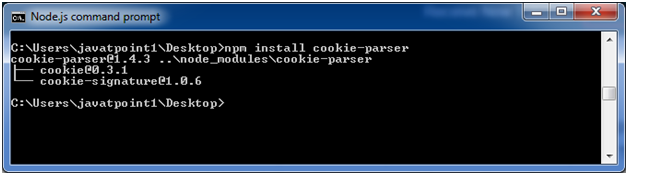
****

**Express.js 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 browses 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 cookie 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');

var cookieParser = 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');

var cookieParser = 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('/cookieget', function(req, res) {

res.status(200).send(req.cookies);

});

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

res.status(200).send('Welcome to JavaTpoint!');

});

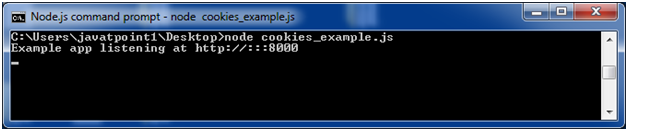
var server = app.listen(8000, function () {

var host = server.address().address;

var port = server.address().port;

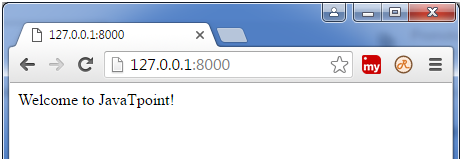
console.log('Example app listening at http://%s:%s', host, port);

});



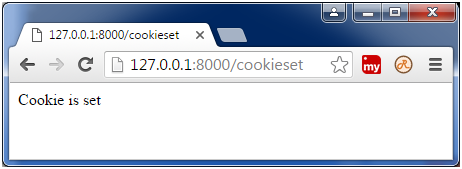
Output:

Open the page **http://127.0.0.1:8000/** on your browser:



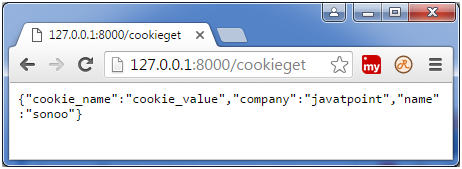
Set cookie:

Now open **http://127.0.0.1:8000/cookieset** to set the cookie:



Get cookie:

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



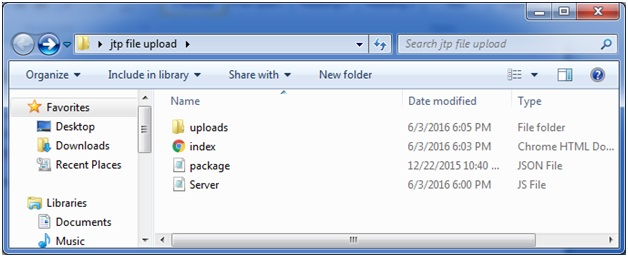
**Express.js File Upload**

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.

Let's 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 following files:



**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"

},

"devDependencies": {

"should": "~7.1.0",

"mocha": "~2.3.3",

"supertest": "~1.1.0"

}

}

**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" enctype="multipart/form-data" action="/uploadjavatpoint" method="post">

<input type="file" name="myfile" /><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('/uploadjavatpoint',function(req,res){

upload(req,res,function(err) {

if(err) {

return res.end("Error uploading file.");

}

res.end("File is uploaded successfully!");

});

});

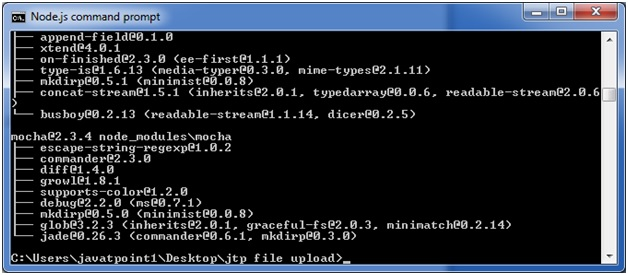
app.listen(2000,function(){

console.log("Server is running on port 2000");

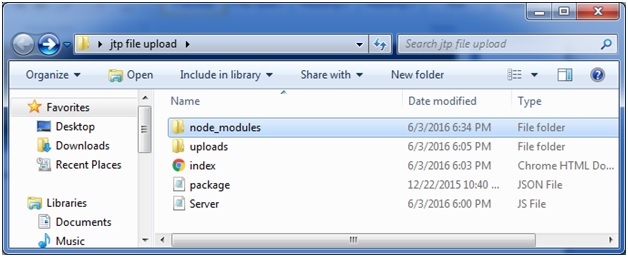
});

To install the package.json, **execute the following code:**

npm install

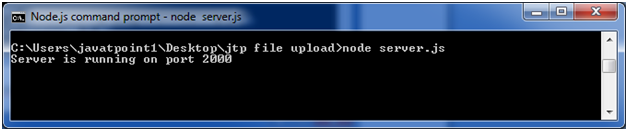


It will create a new folder "node\_modules" inside the "jtp file upload" folder.



Dependencies are installed. Now, run the server:

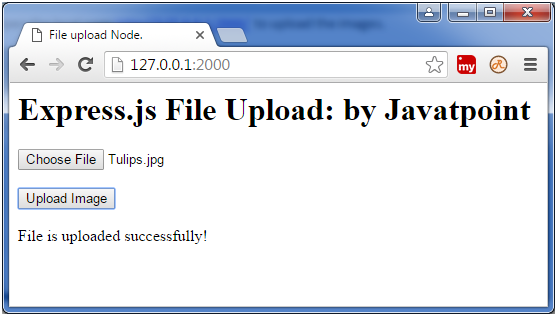
node server.js



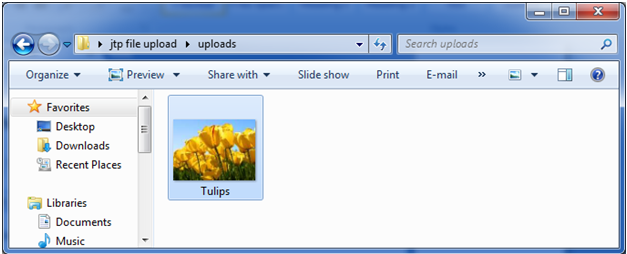
Open the local page http://127.0.0.1:2000/ to upload the images.



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



Here, you see that file is uploaded successfully. You can see the 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?');

});

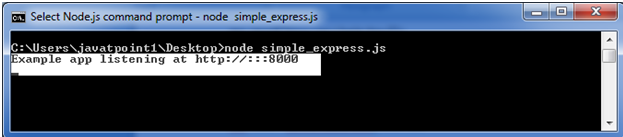
var server = app.listen(8000, function () {

var host = server.address().address

var port = server.address().port

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

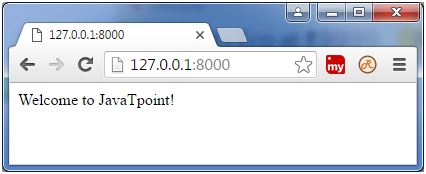
})



You see that server is listening.

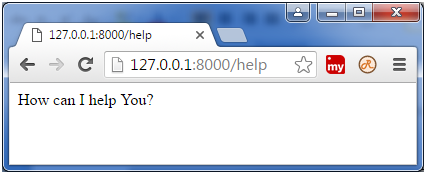
Now, you can see the result generated by server on the local host **http://127.0.0.1:8000**

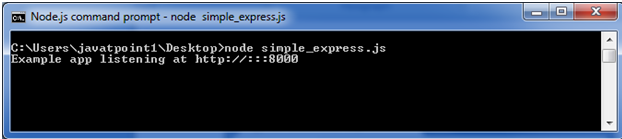
**Output:**



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

Output:





**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 a get a request then you can use a middleware.

**See this example:**

**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 JavaTpoint!');

});

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

res.send('How can I help you?');

});

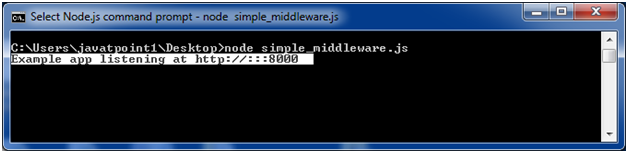
var server = app.listen(8000, function () {

var host = server.address().address

var port = server.address().port

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

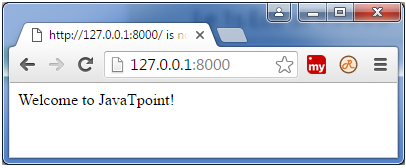
})



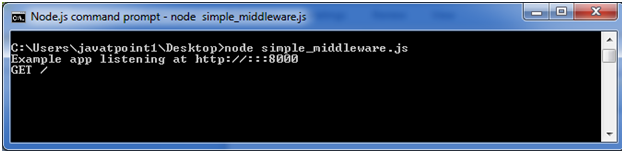
You see that server is listening.

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

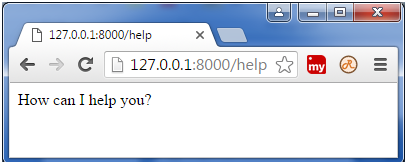
Output:

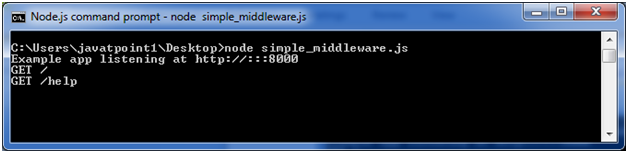


You can see that output is same but command prompt is displaying a GET result.

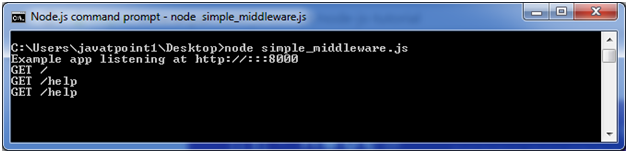


Go to **http://127.0.0.1:8000/help**





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



**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,OutSystems Platform, Express Framework, Play framework, Django, MonoRail, Brail, Symfony, Laravel, CodeIgniter, Yii, CakePHP, 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**

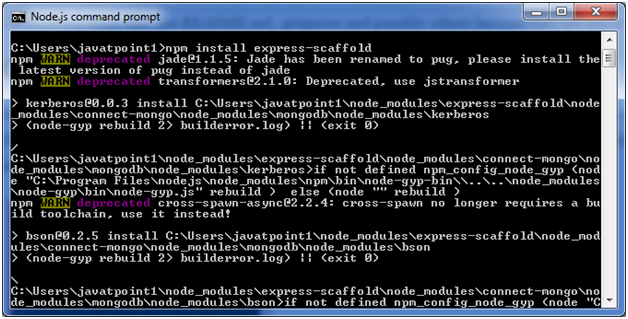
An Express.js scaffold supports candy and more web projects based on Node.js.

**Install scaffold**

Execute the following command to install scaffold.

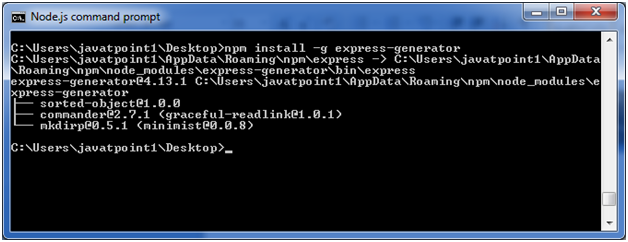
npm install express-scaffold

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



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!');

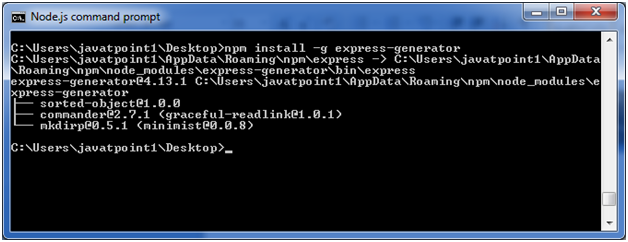
});

app.listen(8000, function () {

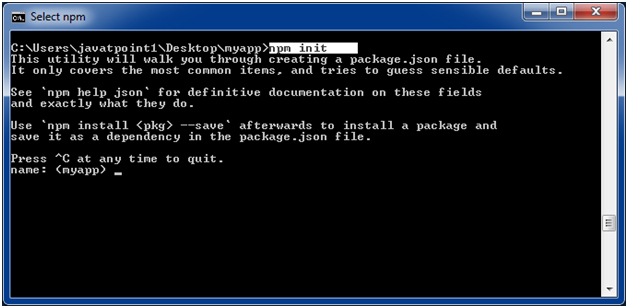
console.log('Example app listening on port 8000!');

});

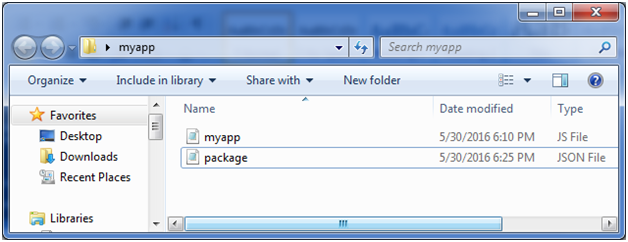
Open Node.js command prompt, go to myapp and run npm init command (In my case, I have created myapp folder on desktop)



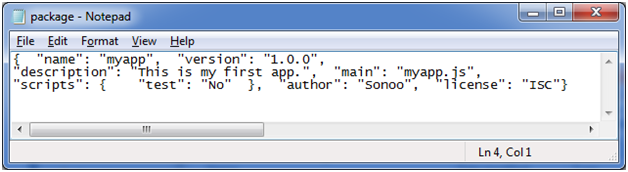
Fill the entries and press enter.



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



Output:



**Express.js Template Engine**

**What is a template engine**

A template engine facilitates you to use static template files in your applications. At runtime, it replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client. So this approach is preferred to design HTML pages easily.

Following is a list of some popular template engines that work with Express.js:

* Pug (formerly known as jade)
* mustache
* dust
* atpl
* eco
* ect
* ejs
* haml
* haml-coffee
* handlebars
* hogan
* jazz
* jqtpl
* JUST
* liquor
* QEJS
* swig
* templayed
* toffee
* underscore
* walrus
* whiskers

In the above template engines, pug (formerly known as jade) and mustache seems to be most popular choice. Pug is similar to Haml which uses whitespace. According to the template-benchmark, pug is 2x slower than Handlebars, EJS, Underscore. ECT seems to be the fastest. Many programmers like mustache template engine mostly because it is one of the simplest and versatile template engines.

**Using template engines with Express**

Template engine makes you able to use static template files 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).

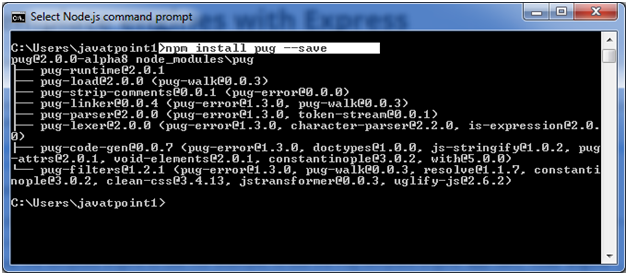
**Pug Template Engine**

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 aesy 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 in a simple way and produces the output in HTML. See how it renders HTML:

**Simple input:**

doctype html

html

head

title A simple pug example

body

h1 This page is produced by pug template engine

p 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 in it:

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..');

});