Learning Outcome

Able to develop the real time scenarios based on Node JS applications

# Create Web App using express

## Node.js - Express Framework

### Express Overview

Express is a minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications. It facilitates the rapid development of Node based Web applications. Following are some of the core features of Express framework −

* Allows to set up middlewares to respond to HTTP Requests.
* Defines a routing table which is used to perform different actions based on HTTP Method and URL.
* Allows to dynamically render HTML Pages based on passing arguments to templates.

### Installing Express

Firstly, install the Express framework globally using NPM so that it can be used to create a web application using node terminal.

**$ npm install express --save**

The above command saves the installation locally in the node\_modules directory and creates a directory express inside node\_modules. You should install the following important modules along with express −

* body-parser − This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data.
* cookie-parser − 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**

### Hello world Example

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

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

**var app = express();**

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

**res.send('Hello World');**

**})**

**var server = app.listen(8081, function () {**

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

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

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

**})**

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

**$ node server.js**

You will see the following output −

Example app listening at http://0.0.0.0:8081

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



Image : Output

Reference: <https://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm>

### Request & Response

Express application uses a callback function whose parameters are request and response objects.

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

**// --**

**})**

Request Object − The request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.

Response Object − The response object represents the HTTP response that an Express app sends when it gets an HTTP request.

You can print req and res objects which provide a lot of information related to HTTP request and response including cookies, sessions, URL, etc.

### Basic Routing

We have seen a basic application which serves HTTP request for the homepage. Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).

We will extend our Hello World program to handle more types of HTTP requests.

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

**var app = express();**

**// This responds with "Hello World" on the homepage**

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

**console.log("Got a GET request for the homepage");**

**res.send('Hello GET');**

**})**

**// This responds a POST request for the homepage**

**app.post('/', function (req, res) {**

**console.log("Got a POST request for the homepage");**

**res.send('Hello POST');**

**})**

**// This responds a DELETE request for the /del\_user page.**

**app.delete('/del\_user', function (req, res) {**

**console.log("Got a DELETE request for /del\_user");**

**res.send('Hello DELETE');**

**})**

**// This responds a GET request for the /list\_user page.**

**app.get('/list\_user', function (req, res) {**

**console.log("Got a GET request for /list\_user");**

**res.send('Page Listing');**

**})**

**// 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('Page Pattern Match');**

**})**

**var server = app.listen(8081, function () {**

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

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

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

**})**

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

**$ node server.js**

You will see the following output −

Example app listening at http://0.0.0.0:8081

Now you can try different requests at http://127.0.0.1:8081 to see the output generated by server.js. Following are a few screens shots showing different responses for different URLs.

Screen showing again <http://127.0.0.1:8081/list_user>



Image : Output

Reference: <https://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm>

Screen showing again <http://127.0.0.1:8081/abcd>

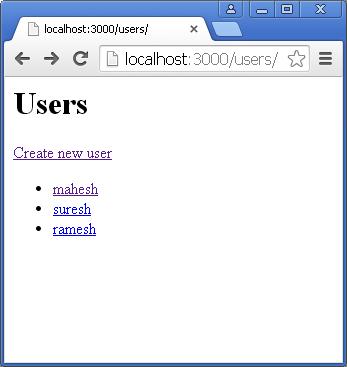


Image : Output

Reference: <https://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm>

Screen showing again <http://127.0.0.1:8081/abcdefg>



Image : Output

Reference: <https://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm>

# Introduction to Middleware Concepts

## ExpressJS – Middleware

Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application’s request-response cycle. These functions are used to modify req and res objects for tasks like parsing request bodies, adding response headers, etc.

Here is a simple example of a middleware function in action −

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

**var app = express();**

**//Simple request time logger**

**app.use(function(req, res, next){**

**console.log("A new request received at " + Date.now());**

**//This function call is very important. It tells that more processing is**

**//required for the current request and is in the next middleware**

**function route handler.**

**next();**

**});**

**app.listen(3000);**

The above middleware is called for every request on the server. So after every request, we will get the following message in the console −

**A new request received at 1467267512545**

To restrict it to a specific route (and all its subroutes), provide that route as the first argument of app.use(). For Example,

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

**var app = express();**

**//Middleware function to log request protocol**

**app.use('/things', function(req, res, next){**

**console.log("A request for things received at " + Date.now());**

**next();**

**});**

**// Route handler that sends the response**

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

**res.send('Things');**

**});**

**app.listen(3000);**

Now whenever you request any subroute of '/things', only then it will log the time.

Order of Middleware Calls

One of the most important things about middleware in Express is the order in which they are written/included in your file; the order in which they are executed, given that the route matches also needs to be considered.

For example, in the following code snippet, the first function executes first, then the route handler and then the end function. This example summarizes how to use middleware before and after route handler; also how a route handler can be used as a middleware itself.

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

**var app = express();**

**//First middleware before response is sent**

**app.use(function(req, res, next){**

**console.log("Start");**

**next();**

**});**

**//Route handler**

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

**res.send("Middle");**

**next();**

**});**

**app.use('/', function(req, res){**

**console.log('End');**

**});**

**app.listen(3000);**

When we visit '/' after running this code, we receive the response as Middle and on our console −

**Start**

**End**

The following diagram summarizes what we have learnt about middleware –

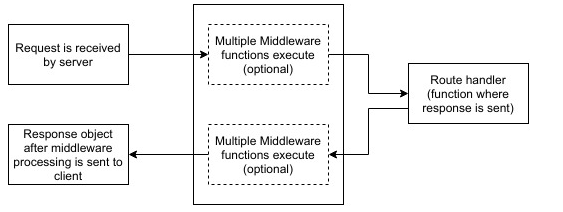


Image : Express Middleware

Reference: <https://www.tutorialspoint.com/expressjs/expressjs_middleware.htm>

### Third Party Middleware

A list of Third party middleware for Express is available here. Following are some of the most commonly used middleware; we will also learn how to use/mount these −

**body-parser**

This is used to parse the body of requests which have payloads attached to them. To mount body parser, we need to install it using npm install --save body-parser and to mount it, include the following lines in your index.js −

**var bodyParser = require('body-parser');**

**//To parse URL encoded data**

**app.use(bodyParser.urlencoded({ extended: false }))**

**//To parse json data**

**app.use(bodyParser.json())**

**cookie-parser**

It parses Cookie header and populate req.cookies with an object keyed by cookie names. To mount cookie parser, we need to install it using npm install --save cookie-parser and to mount it, include the following lines in your index.js −

**var cookieParser = require('cookie-parser');**

**app.use(cookieParser())**

# Express Routing

## ExpressJS – Routing

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

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

**app.method(path, handler)**

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

Path is the route at which the request will run.

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

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

**var app = express();**

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

**res.send("Hello World!");**

**});**

**app.listen(3000);**

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

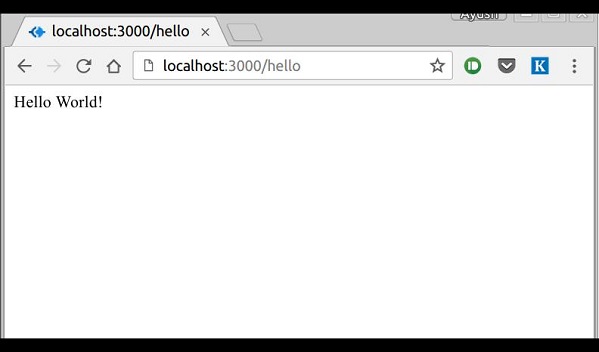


Image : Output

Reference: <https://www.tutorialspoint.com/expressjs/expressjs_routing.htm>

### Routers

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

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

**var router = express.Router();**

**router.get('/', function(req, res){**

**res.send('GET route on things.');**

**});**

**router.post('/', function(req, res){**

**res.send('POST route on things.');**

**});**

**//export this router to use in our index.js**

**module.exports = router;**

**Now to use this router in our index.js, type in the following before the app.listen function call.**

**var express = require('Express');**

**var app = express();**

**var things = require('./things.js');**

**//both index.js and things.js should be in same directory**

**app.use('/things', things);**

**app.listen(3000);**

The app.use function call on route '/things' attaches the things router with this route. Now whatever requests our app gets at the '/things', will be handled by our things.js router. The '/' route in things.js is actually a subroute of '/things'. Visit localhost:3000/things/ and you will see the following output.

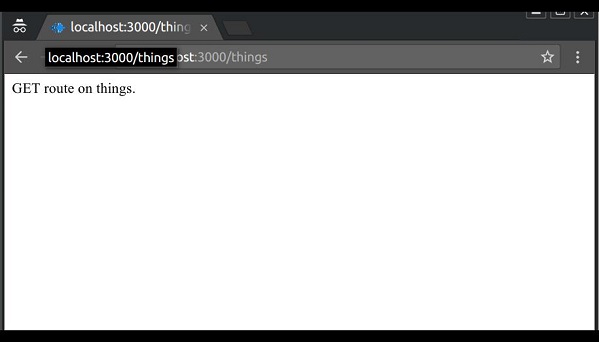


Image : Output

Reference: <https://www.tutorialspoint.com/expressjs/expressjs_routing.htm>

Routers are very helpful in separating concerns and keep relevant portions of our code together. They help in building maintainable code. You should define your routes relating to an entity in a single file and include it using the above method in your index.js file.

# Express JS Template Engine

## ExpressJS – Templating

Pug is a templating engine for Express. Templating engines are used to remove the cluttering of our server code with HTML, concatenating strings wildly to existing HTML templates. Pug is a very powerful templating engine which has a variety of features including filters, includes, inheritance, interpolation, etc. There is a lot of ground to cover on this.

To use Pug with Express, we need to install it,

**npm install --save pug**

Now that Pug is installed, set it as the templating engine for your app. You don't need to 'require' it. Add the following code to your index.js file.

**app.set('view engine', 'pug');**

**app.set('views','./views');**

Now create a new directory called views. Inside that create a file called first\_view.pug, and enter the following data in it.

**doctype html**

**html**

**head**

**title = "Hello Pug"**

**body**

**p.greetings#people Hello World!**

To run this page, add the following route to your app −

**app.get('/first\_template', function(req, res){**

**res.render('first\_view');**

**});**

You will get the output as − Hello World! Pug converts this very simple looking markup to html. We don’t need to keep track of closing our tags, no need to use class and id keywords, rather use '.' and '#' to define them. The above code first gets converted to −

**<!DOCTYPE html>**

**<html>**

**<head>**

**<title>Hello Pug</title>**

**</head>**

**<body>**

**<p class = "greetings" id = "people">Hello World!</p>**

**</body>**

**</html>**

Pug is capable of doing much more than simplifying HTML markup.

### Important Features of Pug

Let us now explore a few important features of Pug.

**Simple Tags**

Tags are nested according to their indentation. Like in the above example, <title> was indented within the <head> tag, so it was inside it. But the <body> tag was on the same indentation, so it was a sibling of the <head> tag.

We don’t need to close tags, as soon as Pug encounters the next tag on same or outer indentation level, it closes the tag for us.

To put text inside of a tag, we have 3 methods −

* **Space seperated**

h1 Welcome to Pug

* **Piped text**

div

| To insert multiline text,

| You can use the pipe operator.

* **Block of text**

div.

But that gets tedious if you have a lot of text.

You can use "." at the end of tag to denote block of text.

To put tags inside this block, simply enter tag in a new line and

indent it accordingly.

### Comments

Pug uses the same syntax as JavaScript(//) for creating comments. These comments are converted to the html comments(<!--comment-->). For example,

**//This is a Pug comment**

This comment gets converted to the following.

**<!--This is a Pug comment-->**

### Attributes

To define attributes, we use a comma separated list of attributes, in parenthesis. Class and ID attributes have special representations. The following line of code covers defining attributes, classes and id for a given html tag.

**div.container.column.main#division(width = "100", height = "100")**

This line of code, gets converted to the following. −

**<div class = "container column main" id = "division" width = "100" height = "100"></div>**

### Passing Values to Templates

When we render a Pug template, we can actually pass it a value from our route handler, which we can then use in our template. Create a new route handler with the following.

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

**var app = express();**

**app.get('/dynamic\_view', function(req, res){**

**res.render('dynamic', {**

**name: "Wikipedia",**

**url:"https://www.wikipedia.org"**

**});**

**});**

**app.listen(3000);**

And create a new view file in views directory, called dynamic.pug, with the following code −

**html**

**head**

**title=name**

**body**

**h1=name**

**a(href = url) URL**

Open localhost:3000/dynamic\_view in your browser;

We can also use these passed variables within text. To insert passed variables in between text of a tag, we use #{variableName} syntax. For example, in the above example, if we wanted to put Greetings from TutorialsPoint, then we could have done the following.

**html**

**head**

**title = name**

**body**

**h1 Greetings from #{name}**

**a(href = url) URL**

This method of using values is called interpolation.

### Conditionals

We can use conditional statements and looping constructs as well.

Consider the following −

If a User is logged in, the page should display "Hi, User" and if not, then the "Login/Sign Up" link. To achieve this, we can define a simple template like −

**html**

**head**

**title Simple template**

**body**

**if(user)**

**h1 Hi, #{user.name}**

**else**

**a(href = "/sign\_up") Sign Up**

When we render this using our routes, we can pass an object as in the following program −

**res.render('/dynamic',{**

**user: {name: "Ayush", age: "20"}**

**});**

You will receive a message − Hi, Ayush. But if we don’t pass any object or pass one with no user key, then we will get a signup link.

### Include and Components

Pug provides a very intuitive way to create components for a web page. For example, if you see a news website, the header with logo and categories is always fixed. Instead of copying that to every view we create, we can use the include feature. Following example shows how we can use this feature −

Create 3 views with the following code −

HEADER.PUG

**div.header.**

**I'm the header for this website.**

CONTENT.PUG

**html**

**head**

**title Simple template**

**body**

**include ./header.pug**

**h3 I'm the main content**

**include ./footer.pug**

FOOTER.PUG

**div.footer.**

**I'm the footer for this website.**

Create a route for this as follows −

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

**var app = express();**

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

**res.render('content');**

**});**

**app.listen(3000);**

Go to localhost:3000/components, you will receive the following output –

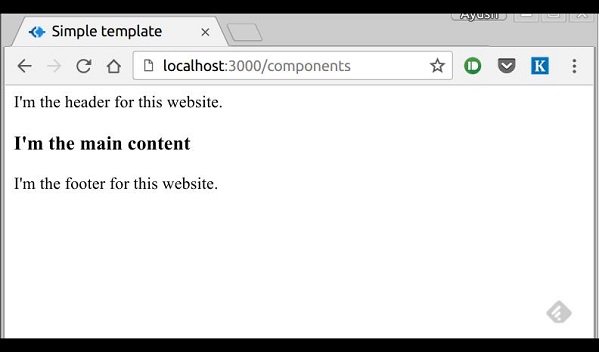


Image : Output

Reference: <https://www.tutorialspoint.com/expressjs/expressjs_templating.htm>

include can also be used to include plaintext, css and JavaScript.

# Handling Query string parameter

## ExpressJS - URL Building

We can now define routes, but those are static or fixed. To use the dynamic routes, we SHOULD provide different types of routes. Using dynamic routes allows us to pass parameters and process based on them.

Here is an example of a dynamic route −

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

**var app = express();**

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

**res.send('The id you specified is ' + req.params.id);**

**});**

**app.listen(3000);**

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

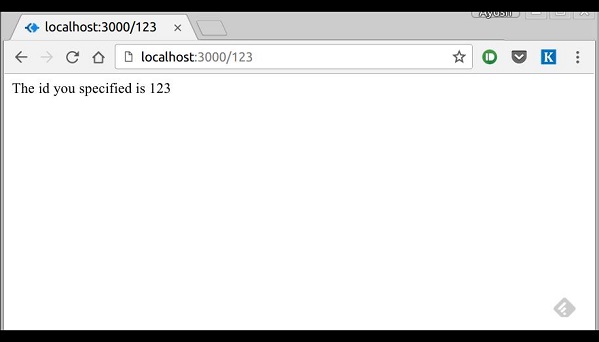


Image : Output

Reference: <https://www.tutorialspoint.com/expressjs/expressjs_url_building.htm>

You can replace '123' in the URL with anything else and the change will reflect in the response.

# Cookies parser & body parser

## ExpressJS – Cookies

Cookies are simple, small files/data that are sent to client with a server request and stored on the client side. Every time the user loads the website back, this cookie is sent with the request. This helps us keep track of the user’s actions.

The following are the numerous uses of the HTTP Cookies −

* Session management
* Personalization(Recommendation systems)
* User tracking

To use cookies with Express, we need the cookie-parser middleware. To install it, use the following code −

**npm install --save cookie-parser**

Now to use cookies with Express, we will require the cookie-parser. cookie-parser is a middleware which parses cookies attached to the client request object. To use it, we will require it in our index.js file; this can be used the same way as we use other middleware. Here, we will use the following code.

**var cookieParser = require('cookie-parser');**

**app.use(cookieParser());**

cookie-parser parses Cookie header and populates req.cookies with an object keyed by the cookie names. To set a new cookie, let us define a new route in your Express app like −

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

**var app = express();**

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

**res.cookie('name', 'express').send('cookie set'); //Sets name = express**

**});**

**app.listen(3000);**

To check if your cookie is set or not, just go to your browser, fire up the console, and enter −

**console.log(document.cookie);**

You will get the output like (you may have more cookies set maybe due to extensions in your browser) −

**"name = express"**

The browser also sends back cookies every time it queries the server. To view cookies from your server, on the server console in a route, add the following code to that route.

**console.log('Cookies: ', req.cookies);**

Next time you send a request to this route, you will receive the following output.

**Cookies: { name: 'express' }**

### Adding Cookies with Expiration Time

You can add cookies that expire. To add a cookie that expires, just pass an object with property 'expire' set to the time when you want it to expire. For example,

**//Expires after 360000 ms from the time it is set.**

**res.cookie(name, 'value', {expire: 360000 + Date.now()});**

Another way to set expiration time is using 'maxAge' property. Using this property, we can provide relative time instead of absolute time. Following is an example of this method.

**//This cookie also expires after 360000 ms from the time it is set.**

**res.cookie(name, 'value', {maxAge: 360000});**

### Deleting Existing Cookies

To delete a cookie, use the clearCookie function. For example, if you need to clear a cookie named foo, use the following code.

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

**var app = express();**

**app.get('/clear\_cookie\_foo', function(req, res){**

**res.clearCookie('foo');**

**res.send('cookie foo cleared');**

**});**

**app.listen(3000);**

## Body-parser middleware in Node.js

Body-parser is the Node.js body parsing middleware. It is responsible for parsing the incoming request bodies in a middleware before you handle it.

Installation of body-parser module:

You can visit the link to Install body-parser module. You can install this package by using this command.

**npm install body-parser**

After installing body-parser you can check your body-parser version in command prompt using the command.

**npm --version body-parser**

After that, you can just create a folder and add a file, for example, index.js. To run this file you need to run the following command.

**node index.js**

Filename: SampleForm.ejs

**<!DOCTYPE html>**

**<html>**

**<head>**

**<title>Body-Parser Module Demo</title>**

**</head>**

**<body>**

**<h1>Demo Form</h1>**

**<form action="saveData" method="POST">**

**<pre>**

**Enter your Email : <input type="text"**

**name="email"> <br>**

**<input type="submit" value="Submit Form">**

**</pre>**

**</form>**

**</body>**

**</html>**

Filename: index.js

**const bodyparser = require('body-parser')**

**const express = require("express")**

**const path = require('path')**

**const app = express()**

**var PORT = process.env.port || 3000**

**// View Engine Setup**

**app.set("views", path.join(\_\_dirname))**

**app.set("view engine", "ejs")**

**// Body-parser middleware**

**app.use(bodyparser.urlencoded({extended:false}))**

**app.use(bodyparser.json())**

**app.get("/", function(req, res){**

**res.render("SampleForm")**

**});**

**app.post('/saveData', (req, res) => {**

**console.log("Using Body-parser: ", req.body.email)**

**})**

**app.listen(PORT, function(error){**

**if (error) throw error**

**console.log("Server created Successfully on PORT", PORT)**

**})**

Steps to run the program:

1. Make sure you have installed ‘view engine’ like I have used “ejs” and also installed express and body-parser module using following commands:

**npm install express**

**npm install ejs**

**npm install body-parser**

1. Run index.js file using below command:

**node index.js**

1. Now Open browser and type the below URL and you will see the Demo Form as shown below:

**http://localhost:3000/**

Now submit the form

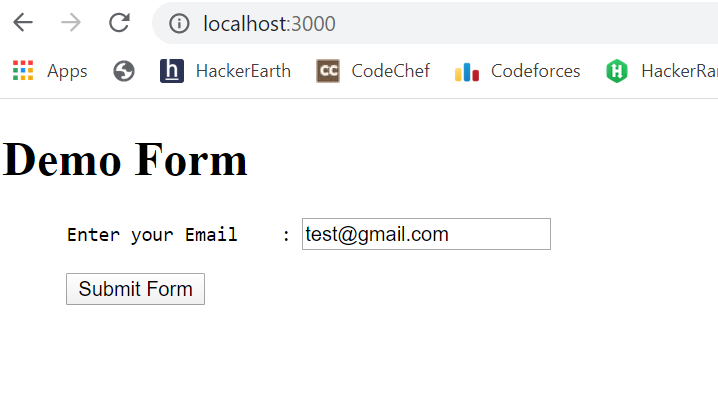


Image : Output

Reference: <https://www.geeksforgeeks.org/body-parser-middleware-in-node-js/>

1. Now submit the form

# Session Handling

## ExpressJS – Sessions

HTTP is stateless; in order to associate a request to any other request, you need a way to store user data between HTTP requests. Cookies and URL parameters are both suitable ways to transport data between the client and the server. But they are both readable and on the client side. Sessions solve exactly this problem. You assign the client an ID and it makes all further requests using that ID. Information associated with the client is stored on the server linked to this ID.

We will need the Express-session, so install it using the following code.

**npm install --save express-session**

We will put the session and cookie-parser middleware in place. In this example, we will use the default store for storing sessions, i.e., MemoryStore. Never use this in production environments. The session middleware handles all things for us, i.e., creating the session, setting the session cookie and creating the session object in req object.

Whenever we make a request from the same client again, we will have their session information stored with us (given that the server was not restarted). We can add more properties to the session object. In the following example, we will create a view counter for a client.

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

**var cookieParser = require('cookie-parser');**

**var session = require('express-session');**

**var app = express();**

**app.use(cookieParser());**

**app.use(session({secret: "Shh, its a secret!"}));**

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

**if(req.session.page\_views){**

**req.session.page\_views++;**

**res.send("You visited this page " + req.session.page\_views + " times");**

**} else {**

**req.session.page\_views = 1;**

**res.send("Welcome to this page for the first time!");**

**}**

**});**

**app.listen(3000);**

What the above code does is, when a user visits the site, it creates a new session for the user and assigns them a cookie. Next time the user comes, the cookie is checked and the page\_view session variable is updated accordingly.

Now if you run the app and go to localhost:3000, the following output will be displayed.

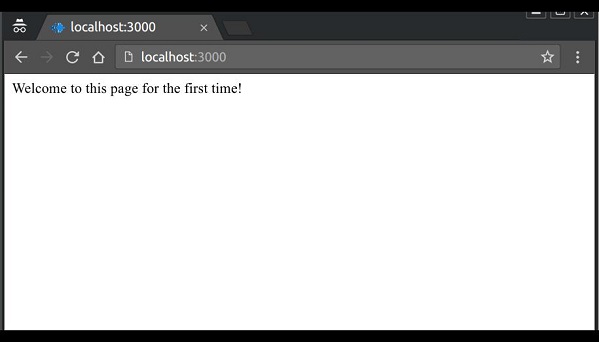


Image : Output

Reference: <https://www.tutorialspoint.com/expressjs/expressjs_sessions.htm>

If you revisit the page, the page counter will increase. The page in the following screenshot was refreshed 42 times.

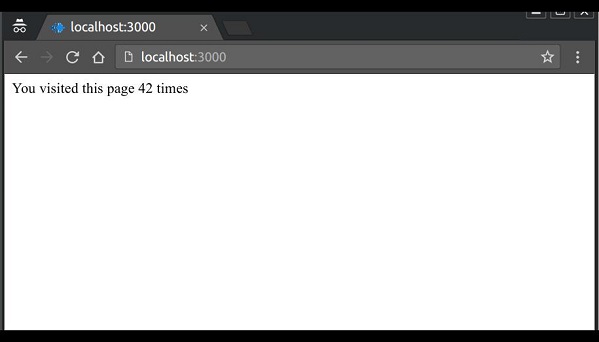


Image : Output

Reference: <https://www.tutorialspoint.com/expressjs/expressjs_sessions.htm>

# Express-mailer

## Installation

Works with Express 3.x.x

**$ npm install express-mailer**

### Usage

Express Mailer extends your express application

**// project/app.js**

**var app = require('express')(),**

**mailer = require('express-mailer');**

**mailer.extend(app, {**

**from: 'no-reply@example.com',**

**host: 'smtp.gmail.com', // hostname**

**secureConnection: true, // use SSL**

**port: 465, // port for secure SMTP**

**transportMethod: 'SMTP', // default is SMTP. Accepts anything that nodemailer accepts**

**auth: {**

**user: 'gmail.user@gmail.com',**

**pass: 'userpass'**

**}**

**});**

### Sending an email

You can send an email by calling app.mailer.send(template, locals, callback). To send an email using the template above you could write:

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

**app.mailer.send('email', {**

**to: 'example@example.com', // REQUIRED. This can be a comma delimited string just like a normal email to field.**

**subject: 'Test Email', // REQUIRED.**

**otherProperty: 'Other Property' // All additional properties are also passed to the template as local variables.**

**}, function (err) {**

**if (err) {**

**// handle error**

**console.log(err);**

**res.send('There was an error sending the email');**

**return;**

**}**

**res.send('Email Sent');**

**});**

**});**

You can also send an email by calling mailer on an applications response object: res.mailer.send(template, options, callback)

# Database operation with MySQL

## Setup Express js for Node.js MySQL

To set up express, we will use the express-generator. You can generate an express js app without any view engine for this Node.js MySQL tutorial with the following command:

**npx express-generator --no-view --git nodejs-mysql**

To quickly check the output execute the following:

**cd nodejs-mysql && npm install && DEBUG=nodejs-mysql:\* npm start**

### MySQL Installation

**$ npm install mysql**

Example

**const mysql = require('mysql')**

**const connection = mysql.createConnection({**

**host: 'localhost',**

**user: 'dbuser',**

**password: 's3kreee7',**

**database: 'my\_db'**

**})**

**connection.connect()**

**connection.query('SELECT 1 + 1 AS solution', (err, rows, fields) => {**

**if (err) throw err**

**console.log('The solution is: ', rows[0].solution)**

**})**

**connection.end()**

References

1. <https://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm>
2. <https://www.tutorialspoint.com/expressjs/expressjs_middleware.htm>
3. <https://www.tutorialspoint.com/expressjs/expressjs_routing.htm>
4. <https://www.tutorialspoint.com/expressjs/expressjs_templating.htm>
5. <https://www.tutorialspoint.com/expressjs/expressjs_url_building.htm>
6. <https://www.tutorialspoint.com/expressjs/expressjs_cookies.htm>
7. <https://www.geeksforgeeks.org/body-parser-middleware-in-node-js/>
8. <https://www.tutorialspoint.com/expressjs/expressjs_sessions.htm>
9. <https://www.npmjs.com/package/express-mailer>
10. <https://expressjs.com/en/guide/database-integration.html#mysql>
11. <https://geshan.com.np/blog/2020/11/nodejs-mysql-tutorial/>