**NodeJS**

Node.js is an open-source, cross-platform JavaScript run-time environment for executing JavaScript code on server-side.

**Installation:**

**Linux:**

Open Terminal and type the following:

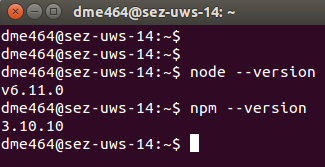
sudo apt-get update

curl -sL https://deb.nodesource.com/setup\_6.x | sudo -E bash

sudo apt-get install nodejs

node --version

npm --version



**Important Note:-**

All the functionality explained in the document is already done and is available to download from GitHub(https://github.com/tysonpaul89/nodeJS\_express\_tutorial).

Please download/clone the file on a desired folder and run the following command to run the project

npm install

node test.js

**Express:**

Express is a minimal and flexible Node.js web application framework.

Before installing express we need create a project directory, so in Terminal create a directory.

mkdir test

cd test

Now we need create a package.json file.

A package.json - file holds various metadata relevant to the project. This file is used to give information to npm that allows it to identify the project as well as handle the project's dependencies.

Open Terminal and type the command below to create a package.json file.

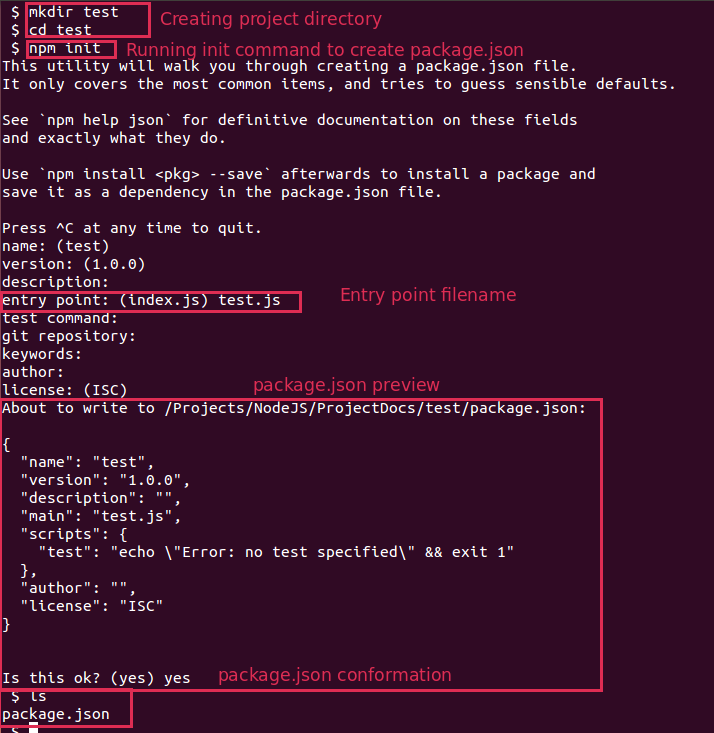
npm init

Now npm will ask for some information like name, version, description, entry point, etc. We will skip all other fields except ‘entry point’ field. In the entry point type test.js.

Note that, all these fields are optional so its ok to skip the fields.

At the end npm will show a preview of our package.json and ask confirm to create the file. Type ‘yes’ and press return key to create the package.json file.

Now type ls command in terminal to verify if the package.json file is created.



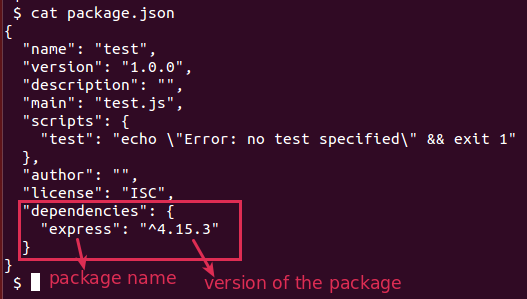
Now that we created the package.json file, we can now install express framework.

Type the following command

npm install express --save

This will create the ‘node\_modules’ directory in the current directory(if one doesn't exist yet), and will download the express package to that directory.

Now open package.json file and see if express listed in the ‘dependencies’.



**Note:-**

* Npm will install all it’s packages in the ‘node\_modules’ directory.
* The ‘-- save’ command at the end of the npm install command is used to save the package details in package.json file. All the packages installed with ‘--save’ command will be listed under ‘dependencies’ property in the package.json file.

**Useful Commands:**

|  |  |  |
| --- | --- | --- |
| **Command Syntax** | **Example** | **Description** |
| node <script>.js | node app.js | To start a node application. |
| Ctr + C | - | To stop node application. |
| npm init | - | To create a package.json file to manage dependency packages of the project. |
| npm start | - | To start the node application using the command saved in ‘script.start’ property of the package.json file |
| npm install <package> | npm install express | To install a package. |
| npm install <package> --save | npm install express --save | To install a package and to save the package details as a production dependency in package.json file. |
| npm install <package> --save-dev | npm install nodemon --save-dev | To install a package and to save the package details as a development dependency in package.json file. |
| npm install -g <package> | npm install -g standard | To install a package globally rather than locally.  All the normal npm install syntax operation will only install package locally and wouldn’t available in other project but globally installed packages will be available to all projects. |
| npm uninstall <package> --save | npm uninstall express --save | To uninstall a package and to remove the package details from package.json file. |
| npm uninstall <package> --save-dev | npm uninstall nodemon --save-dev | To uninstall a package and to remove the package details from package.json file. |
| npm uninstall -g <package> | npm uninstall -g standard | To uninstall a package globally |

**Working:**

1. **Intro to ES6 features**

EcmaScript is the "official" name for JavaScript and ES6 is short-hand for EcmaScript 6. ie, ES6 is the 6th version of JavaScript.

Following are the some of the new ES6 features that we will encounter in NodeJS:

**let and const keywords:**

let and const are newly introduced in ES6. They mainly used for block scoping.

Declaring variables with let or const within a code block restricts their visibility to that block only. This is called block scoping.

A variable declared with let keyword will only available inside the block. example

example 1:

for (var i=0; i < 5; i++) {}

console.log('i= ' + i) // i= 5

for (let i=0; i < 5; i++) {}

console.log('i= ' + i) // Uncaught ReferenceError: i is not defined

const allows us to declare constant variables. const does not indicate that the assigned value will be

constant, but that the binding with the value is constant.

const person = {}

person.name = 'joe' // "joe"

person = null // Uncaught TypeError: Assignment to constant variable.

**Arrow functions:**

An arrow function expression has a shorter syntax than a function expression.

Example1:

const numbers = [2, 6, 7, 8, 1];

const even = numbers.filter(function(x) {

return x%2 === 0;

});

const numbers = [2, 6, 7, 8, 1];

const even = numbers.filter(x => x%2 === 0);

1. **Creating Web Server**

NodeJS doesn’t need a server application to serve pages, instead it can create its own server.

Lets create a webserver to say ‘hello world’. Open the project folder and paste the following lines

// Gets http system module

*const* http = require('http')

// Creates a web server and listens on port 4000

http.createServer((*req*, *res*) *=>* {

// Sets status code and header

res.writeHead(200, {'content-type': 'text/plain'})

// Writes the content

res.end('Hello world')

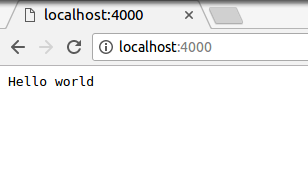
}).listen(4000)

To run the code open terminal in the project folder and type

node test.js

Now this url http://localhost:4000/ in a browser and you should see page like below.

**Note:-** To stop the server press Ctrl + C



1. **Routing using express**

**Simple routing:**

Stop the server if running and paste the code below in test.js

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

http.createServer(*function* (*req*, *res*) {

// normalize url by removing querystring, optional

// trailing slash, and making it lowercase

*var* path = req.url.replace(/\/?(?:\?.\*)?$/, '').toLowerCase();

switch (path) {

case '':

res.writeHead(200, { 'Content-Type': 'text/html' });

res.end('<h1>Homepage</h1>');

break;

case '/about':

res.writeHead(200, { 'Content-Type': 'text/html' });

res.end('<h1>About</h1>');

break;

default:

res.writeHead(404, { 'Content-Type': 'text/html' });

res.end('<h1>Not Found</h1>');

break;

}

}).listen(4000);

*console*.log('Server started on localhost:4000; press Ctrl-C to terminate....');

Now open the following links:

http://localhost:4000/ will serve homepage

http://localhost:4000/about will serve about page

http://localhost:4000/foo will serve not found page with 400 status

Here, req.url property will gets the full url. Then with the using a regular expression we will filter the url to get the only the path(eg: /about) of the url. This path then matched to get correct route.

**Routing Using Express:**

The above example using node will seems simple but its get complex when we want a pass data with the url. Also setting header and content type for each request break the DRY coding principles. This is where the express come into the picture. Now replace the existing code with the following code and run server.

// getting express module

*const* express = require('express')

*const* app = express()

*const* port = 4000 // port

// Home page route

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

res.send('<h1>Homepage</h1>')

})

// About page route

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

res.send('<h1>About</h1>')

})

// Hello page route

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

// Gets name from query string

*let* name = req.params.name

res.send('<h1>Hello ' + name + '</h1>')

})

// Error page route

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

// passing a custom error to error handler route

next(new Error('testing error'))

})

// Error handler route

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

// shows error stack in console

*console*.error(err.stack)

res.status(500)

// shows error message

res.send('<h1>Server Error</h1> <br/> Error Message: <b>' +

err.message + '</b>'

)

})

// 404 page route

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

res.status(404);

res.send('<h1>Not Found</h1>')

})

// Creates server and listens to the port

app.listen(port, *function* () {

*console*.log('Express started on http://localhost:' +

port + '; press Ctrl-C to terminate.')

})

Now open the following links:

http://localhost:4000/ will serve homepage

http://localhost:4000/about will serve about page

http://localhost:4000/foo will serve not found page

http://localhost:4000/hello/Joe will serve hello page

http://localhost:4000/error will serve error handler page

Notice that we didn’t filter the url, content-type header and status code; all this will be handled by the express framework.

**Express Routing explained:**

app.get() function is used to handle HTTP GET request.

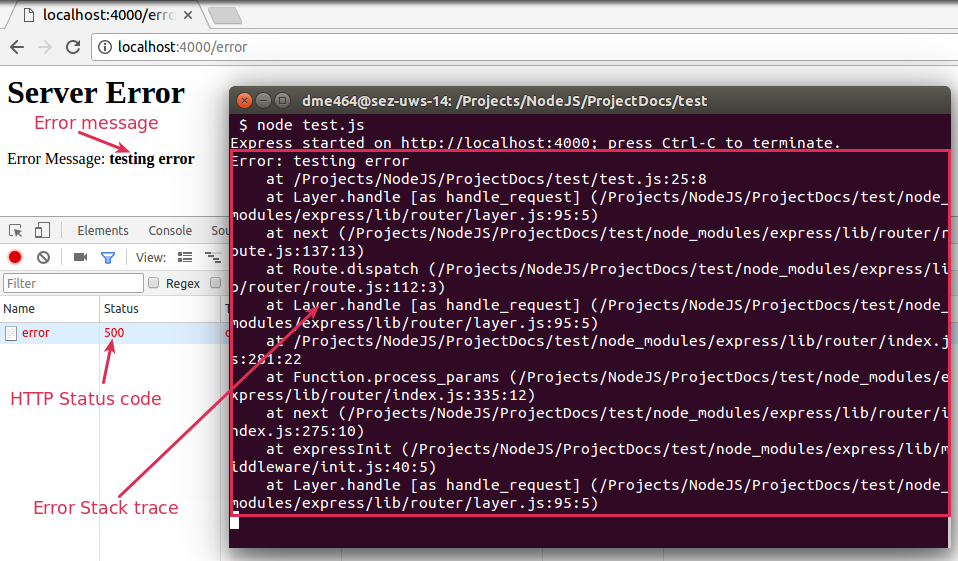
The first argument to get() function is path and second argument is the a callback function.

When a url is called in the browser, then express will get that request and search for the function that matches the path.

If the express finds the match, then then the callback function of the corresponding get() function is executed and if express couldn't find the match, then last 404 route will be executed.

The callback function of the get() function have must have three arguments (req and res are required and next is optional). The three arguments are req - which is the request object, res - which is the response object and next - which is a callback function.

In example url http://localhost:4000/error , an error object is passed to the ‘next’ callback function. The express will call get this error object in the ‘Error handler route’ which is the second last route.



The error handler route will then console stack to the terminal and then displays the error message in the error page. This is the common way of handling error in nodeJS.

In example url http://localhost:4000/hello/Joe ‘Joe’ is a query string data passed with url. In the app.get () function the path string ‘:name’ indicates that ‘name’ part is query string. We can access query strings using ‘req.params’ property

**Important Note:-**

The order of the routes are important. The 404 page route should be place at the end.

Always pass the error to the error handler as an error object using the ‘next’ function. When we pass error as an error object, the error stack will contain the line number from where it was called.

1. **Templating using ejs package**

EJS is a simple templating language that combines data and a template to produce HTML.

**Installation:**

npm install ejs --save

To use ejs we must first set the view path and view engine in express. Then by calling ‘res.render()’ function will render the page. The following code will demonstrate the basic working of ejs.

Open test.js and paste the following code.

// getting express module

*const* express = require('express')

*const* app = express()

*const* port = 4000 // port

// Setting view directory

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

// Setting view engine

app.set('view engine', 'ejs')

// Home page route

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

res.send('<h1>Homepage</h1>')

})

// About page route

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

res.send('<h1>About</h1>')

})

// Hello page route

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

// Gets name from query string

*let* name = req.params.name

// Renders view using ejs template

res.render('hello', { name: name })

})

// Error page route

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

// passing a custom error to error handler route

next(new Error('testing error'))

})

// 404 page route

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

res.status(404)

res.send('<h1>Not Found</h1>')

})

// Error handler route

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

// shows error stack in console

*console*.error(err.stack)

res.status(500)

// shows error message

res.send('<h1>Server Error</h1> <br/> Error Message: <b>' + err.message +

'</b>')

})

app.listen(port, *function* () {

*console*.log('Express started on http://localhost:' +

port + '; press Ctrl-C to terminate.')

})

Now create a directory named ‘views’, then in that directory create new file and name it hello.ejs.

In hello.ejs paste the following code:

<% if (name) {%>

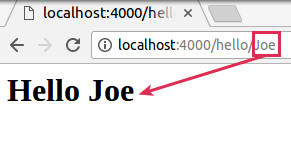
<h1>Hello <%= name %></h1>

<% } else { %>

<h1>Hello</h1>

<% } %>

Now start the server and open this url http://localhost:4000/hello/Joe , you will see the following message.



In test.js ‘res.render()’ function has two arguments, first is the name of the view page and second is the data object. The properties set in the data object will be accessible in view page.

The view file is should be save as .ejs extension, otherwise an error will be thrown.

‘<%= %>’ is the ejs tag to outputs the value into the template and ‘<% %>’ is the tag, for control-flow.

Ref:

https://expressjs.com/en/guide/using-template-engines.html

[https://www.npmjs.com/package/ejs#usage](https://www.npmjs.com/package/ejs" \l "usage)

1. **Serving Static Content and Template Layouts**

**Serving Static Content:**

First create a directory in the root called ‘public’ and put all static contents like CSS, JS, images, etc in this directory.

Then add following line in the ‘test.js’ file. Now we can load the files that are in ‘public’ directory like http://localhost:3000/css/style.css

Here ‘path’ module is used to get the full path of the directory. Path module will make sure that path given matches operating system path styles.

Read more about serving static content from http://expressjs.com/en/starter/static-files.html

// Gets the path middleware package

const path = require('path')

// Setting directory to serve static content

app.use(express.static(path.join(\_\_dirname, 'public')))

**Template Layouts using EJS:**

EJS does not specifically support layouting, but layouts can be implemented by including headers and footers.

Open ‘views’ directory and create a new directly called ‘layouts’ and in that directory create two ejs files names ‘header.ejs’ and ‘footer.ejs’

Add the following following codes and open http://localhost:4000/user/add link:

/views/layouts/header.ejs

<!DOCTYPE html>

<html>

<head>

<title>NodeJS tutorial</title>

<link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" rel="stylesheet">

<link href="/css/style.css" rel="stylesheet">

</head>

<body>

<div id="wrapper">

<div class="page-header"><h2><%= pageTitle %></h2></div>

<div class="page-container">

/views/layouts/footer.ejs

</div>

</div>

<script

src="https://code.jquery.com/jquery-3.2.1.min.js"

integrity="sha256-hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4="

crossorigin="anonymous">

</script>

<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js">

</script>

</body>

</html>

Now in the view file add the follow code:

/views/add.ejs

<!-- Header -->

<%- include('./layouts/header') %>

<!-- /Header -->

<form enctype="multipart/form-data" method="post" action="/user/add">

<div class="form-group">

<a href="/user/list" class="btn btn-warning">Go Back</a>

</div>

<div class="form-group">

<labe for="name">Name</labe>

<input type="text" id="name" name="name" class="form-control">

</div>

<div class="form-group">

<labe for="age">Age</labe>

<input type="text" id="age" name="age" class="form-control">

</div>

<div class="form-group">

<labe for="dob">Date of Birth</labe>

<input type="date" id="dob" name="dob" class="form-control">

</div>

<div class="form-group">

<labe for="image">Image</labe>

<input type="file" id="image" name="image" class="form-control">

</div>

<div class="form-group">

<input type="submit" id="submit" name="submit" class="btn btn-sucess">

</div>

</form>

<!-- Footer -->

<%- include('./layouts/footer') %>

<!-- /Footer -->

/test.js

// User Add form route

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

req.app.locals.pageTitle = 'Add User'

res.render('add')

})

1. **Database connection using sequelize package**

Sequelize is a ORM for Nodejs. It supports the dialects PostgreSQL, MySQL, SQLite and MSSQL and features solid transaction support, relations, read replication and more.

**Installation:**

npm install mysql2 --save

npm install sequelize --save

mysql2 package is the Mysql driver package for node and sequelize package is the ORM.

Lets create a model using Sequalize:

// Getting ORM package

const Sequelize = require('sequelize')

// Sequelize package database configuration

const ormObj = new Sequelize(

'test', // database name

'root', // username

'pass', { // password

dialect: 'mysql', // Specifying DBMS

logging: false // Disables console logging queries

}

)

// User Model definition

const User = ormObj.define('user', { // Table name

id: { // Filed name

type: Sequelize.INTEGER, // Field type for INT

primaryKey: true,

autoIncrement: true

},

name: {

type: Sequelize.STRING,

allowNull: false

},

age: {

type: Sequelize.INTEGER,

allowNull: true

},

dob: {

type: Sequelize.DATEONLY,

allowNull: true

},

doj: {

type: Sequelize.DATE,

allowNull: false

},

profile: {

type: Sequelize.STRING,

allowNull: true

}

}, {

freezeTableName: true, // To user original name of table

timestamps: false // Prevents sequelize from creating default columns

})

At first we create a connection object using ‘Sequelize’ constructor. Then, we use ‘define’ function to create a mappings between a model and a table. The define function accepts three parameters; model name, attribute object and an option object. The model name is the name of the table and attribute object contains the column data.

Now to we can use ‘User’ object do the CRUD operation. Following code is and example of a CRUD operation using ‘Sequelize’:

// Create Operation

User.create({

name: 'test',

age: 20,

doj: Sequelize.fn('NOW') // current timestamp

}).then(insRes => console.log(JSON.stringify(insRes))).catch(err => {

if (err) throw new Error(err)

})

// Read Operation

User.findAll({

attributes: ['id', 'name', 'age', 'profile']

}).then(results => {

console.log(results)

}).catch(err => {

if (err) throw new Error(err)

})

// Update operation

User.update({ // value object

name: 'tys'

}, { // Option object

where: {

id: 3

}

}).then(updRes => console.log(updRes)).catch(err => {

if (err) throw new Error(err)

})

// Delete operation

User.destroy({

where: {id: 2}

}).then(dat => console.log(dat)).catch(err => {

if (err) throw new Error(err)

})

1. **CURD operation**

NodeJS by default can parse query strings using ‘req.query’ property. But it wouldn’t parse POST body. So to parse url encoded and multipart/form form data we need to install the following packages.

**Installation:**

npm install body-parser --save

npm install express-fileupload --save

body-parser middleware used to parse request bodies of POST data from a form and express-fileupload middleware is used to parse files from the multipart forms.

**Note:-** body-parser middleware wouldn't parse data from multipart forms, so we need to use express-fileupload middleware to parse multipart forms.

Please download the project code from GitHub using this link: https://github.com/tysonpaul89/nodeJS\_express\_tutorial and see the ‘CRUD Routes’ section.

Here there is two function pointing to same route app.get(‘/user/add’) and app.post(‘/user/add’), this is because express framework when finding route will also checks the HTTP method of the request. So all the HTTP GET request will go to the app.get() function and HTTP POST request will go to the app.post() function.

In express req.query is used to get the query string data of the url, req.body is used to get the post data and req.param is used to get the data passed in the url

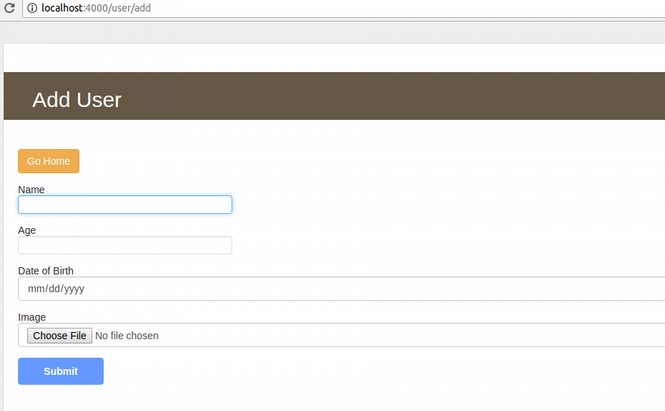
Example:

In this url http://localhost:4000/?test=ok case, the output of the req.query will be { test: 'ok' }

Also in this update user route url http://localhost:4000/user/add/1 case, the output of the req.param will be { userId: '1' }

I. Create Operation

Link: http://localhost:4000/user/add



The above link will match this to app.get('/user/add') route and the /views/add.ejs file will be rendered.

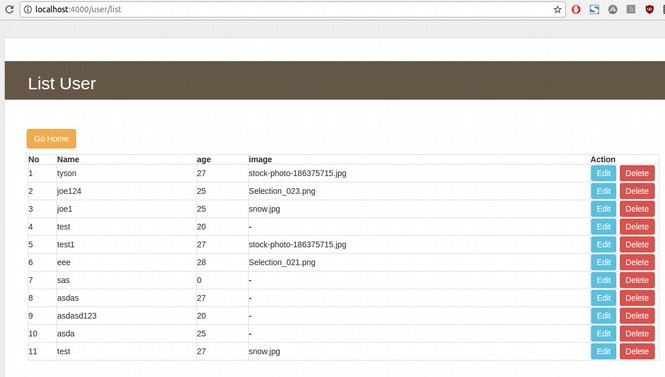
When the form is submitted then express will match app.post('/user/add') route.

The form data will be available in req.body and image data will be available in req.files objects.

res.redirect() is the function in nodejs to redirect user to a another page.

II. Read Operation

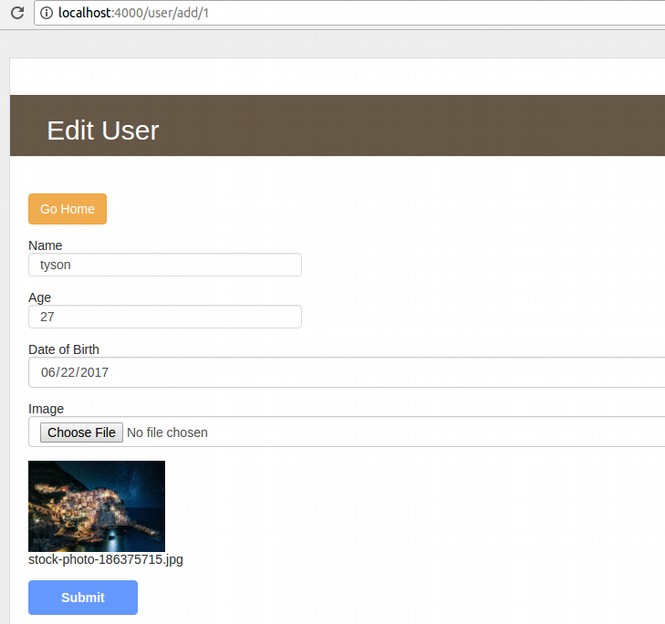
Link: http://localhost:4000/user/list



The route app.get('/user/list') is used to list the users.

III. Update Operation

Link: http://localhost:4000/user/add/1



Here, app.get('/user/add/:userId') function will render the page and form will be submitted to app.post('/user/edit') route.

In the app.get('/user/add/:userId') route passed, used id can be accessible in req.params.userId property.

IV. Delete Operation

Link: http://localhost:4000/user/delete/13

Here, app.get('/user/delete/:userId') will be matched and he user id will be available in the req.params.userId property.

1. **Validation using validate.js**

Validate.js package that can be used in front-end and back-end to validate forms data.

The syntax for validator constraints(rules) is as follows

{

<attribute>: {

<validator name>: <validator options>

}

}

Note:- In the test.js the validate.js is used in app.post('/user/add') route to validate form data.

Consider the following example:

const constraints = {

name: {

presence: true,

length: {

maximum: 10,

message: 'More than 10 character is not allowed in the Name field'

}

},

age: {

presence: true,

numericality: {

onlyInteger: true,

greaterThanOrEqualTo: 18,

lessThanOrEqualTo: 60

}

},

dob: { // Date of birth rule

presence: false,

format: {

pattern: '\\d{4}-\\d{2}-\\d{2}',

message: '^Please enter a valid date of birth'

}

}

}

const formData = {"name":"", "age":"", "dob":"", "submit":"Submit"}

let errors = validate(formData, constraints)

Output of errors is:

{

"name":[

"Name can't be blank"

],

"age":[

"Age can't be blank",

"Age is not a number"

],

"dob":[

"Please enter a valid date of birth"

]

}

Here presence validator is used in required fields. It will produce an error, if the data is empty, undefined and whitespaces strings. If user provides empty data, then a message “Name can't be blank”.

length validator is used to check the length of a string. The option maximum is used to limit the character length to 10. If the user enters a string of length more than 10 then the error will have message given in the message option.

The numericality validator will only allow numbers. The options greaterThanOrEqualTo and lessThanOrEqualTo is used to limit the input.

The format validator will validate a value against a regular expression.

Ref: https://validatejs.org/

1. **Realtime time updates using socket.io package**

Socket.IO is composed of two parts:

* A server that integrates with (or mounts on) the Node.JS HTTP Server: **socket.io**
* A client library that loads on the browser side: **socket.io-client**

During development, **socket.io** serves the client automatically for us, so we need to install just one package

**Installation**

npm install --save socket.io

In the entry script **test.js,**  require the following HTTP module and socket.io

var http = require('http').Server(app)

var io = require('socket.io')(http)

Here what we did is, we have initialized the instance of socket.io module by passing the HTTP server object.

Then add the following code in the same **test.js**

Here we are listening on the **connection** event for incoming sockets, and will log it to the console.

io.on('connection', function(socket){

console.log('a user connected');

});

Add the following code in the before **</body>** tag in the view file. In our case add the same in the **footer.ejs**

Here **socket.io-client** will be automatically connect to the io

<script src="/socket.io/socket.io.js"></script>

<script>

var socket = io();

</script>

After that, if we restart or refresh any page we can see the message – “a user connected” in the terminal

Each socket also fires a special **disconnect** event. Try the disconnect event by adding the following code below

io.on('connection', function(socket){

console.log('a user connected');

socket.on('disconnect', function(){

console.log('user disconnected')

})

})

**Emitting events**

The main idea behind Socket.IO is that you can send and receive any events you want, with any data you want. Any objects that can be encoded as JSON will do, and binary data is supported too

Let us try a simple message sending.

Firstly create the following samples inputs

1. Text box for writing message,
2. Text box for writing user’s name,
3. Text box area for displaying the chat message
4. Send button in your view page.

<textarea rows="4" cols="50" class="text-area" id="text-area"></textarea><br> <input type="text" class="chat-user" id="chat-user" placeholder="Enter Username"></input> <input type="text" class="text-chat" id="text-chat" placeholder="Enter message"></input> <input type="button" value="Send" class="send-chat" id="send-chat"></input>

Then add the following script code in the before the **</body>** part.

<script src="/socket.io/socket.io.js"></script>

<script>

$(function () {

var socket = io();

$('#send-chat').click(function(){

var user = $('#chat-user').val()

socket.emit('chat message', $('#text-chat').val(), user);

$('#text-chat').val('');

return false;

});

socket.on('display message',function (msg, user) {

console.log('Message : ' + msg)

$('#text-area').append( user + ' says : ' + msg + '\n');

})

});

</script>

Here we are taking the value of user’s name and the message from the text box and passed to an **chat** **message (event name)** and then we will clear the value of the text box message after successful click of the Send Button

And similarly we paste the code in the **test.js**

io.on('connection', function (socket) {

socket.on('chat message', function (msg, user) {

io.emit('display message', msg, user)

})

})

Here we are listening the event **chat message** by using **on** method and we will get the message and user details and same details are send to a new event **display message** via emit method**.**

From the above code which we have written in **</body>** part, there we are listening the new event **display message** and will append the details to the text area box.