Node.js - First Application

Before creating an actual "Hello, World!" application using Node.js, let us see the components of a Node.js application. A Node.js application consists of the following three important components −

* **Import required modules** − We use the **require** directive to load Node.js modules.
* **Create server** − A server which will listen to client's requests similar to Apache HTTP Server.
* **Read request and return response** − The server created in an earlier step will read the HTTP request made by the client which can be a browser or a console and return the response.

Creating Node.js Application

Step 1 - Import Required Module

We use the **require** directive to load the http module and store the returned HTTP instance into an http variable as follows −

var http = require("http");

Step 2 - Create Server

We use the created http instance and call **http.createServer()** method to create a server instance and then we bind it at port 8081 using the **listen** method associated with the server instance. Pass it a function with parameters request and response. Write the sample implementation to always return "Hello World".

http.createServer(function (request, response) {

// Send the HTTP header

// HTTP Status: 200 : OK

// Content Type: text/plain

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

// Send the response body as "Hello World"

response.end('Hello World\n');

}).listen(8081);

// Console will print the message

console.log('Server running at http://127.0.0.1:8081/');

The above code is enough to create an HTTP server which listens, i.e., waits for a request over 8081 port on the local machine.

Step 3 - Testing Request & Response

Let's put step 1 and 2 together in a file called **main.js** and start our HTTP server as shown below −

var http = require("http");

http.createServer(function (request, response) {

// Send the HTTP header

// HTTP Status: 200 : OK

// Content Type: text/plain

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

// Send the response body as "Hello World"

response.end('Hello World\n');

}).listen(8081);

// Console will print the message

console.log('Server running at http://127.0.0.1:8081/');

Now execute the main.js to start the server as follows −

$ node main.js

Verify the Output. Server has started.

Server running at http://127.0.0.1:8081/

Make a Request to the Node.js Server

Open http://127.0.0.1:8081/ in any browser and observe the following result.



Congratulations, you have your first HTTP server up and running which is responding to all the HTTP requests at port 8081.

## What is Callback?

Callback is an asynchronous equivalent for a function. A callback function is called at the completion of a given task. Node makes heavy use of callbacks. All the APIs of Node are written in such a way that they support callbacks.

For example, a function to read a file may start reading file and return the control to the execution environment immediately so that the next instruction can be executed. Once file I/O is complete, it will call the callback function while passing the callback function, the content of the file as a parameter. So there is no blocking or wait for File I/O. This makes Node.js highly scalable, as it can process a high number of requests without waiting for any function to return results.

## Blocking Code Example

Create a text file named **input.txt** with the following content −

Tutorials Point is giving self learning content

to teach the world in simple and easy way!!!!!

Create a js file named **main.js** with the following code −

var fs = require("fs");

var data = fs.readFileSync('input.txt');

console.log(data.toString());

console.log("Program Ended");

Now run the main.js to see the result −

$ node main.js

Verify the Output.

Tutorials Point is giving self learning content

to teach the world in simple and easy way!!!!!

Program Ended

## Non-Blocking Code Example

Create a text file named input.txt with the following content.

Tutorials Point is giving self learning content

to teach the world in simple and easy way!!!!!

Update main.js to have the following code −

var fs = require("fs");

fs.readFile('input.txt', function (err, data) {

if (err) return console.error(err);

console.log(data.toString());

});

console.log("Program Ended");

Now run the main.js to see the result −

$ node main.js

Verify the Output.

Program Ended

Tutorials Point is giving self learning content

to teach the world in simple and easy way!!!!!

These two examples explain the concept of blocking and non-blocking calls.

* The first example shows that the program blocks until it reads the file and then only it proceeds to end the program.
* The second example shows that the program does not wait for file reading and proceeds to print "Program Ended" and at the same time, the program without blocking continues reading the file.

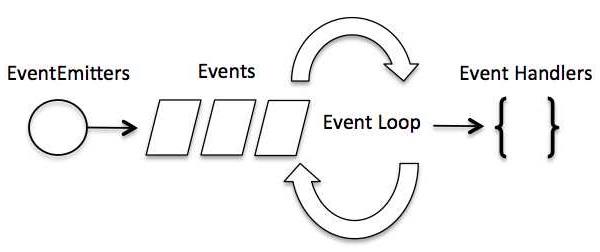
Thus, a blocking program executes very much in sequence. From the programming point of view, it is easier to implement the logic but non-blocking programs do not execute in sequence. In case a program needs to use any data to be processed, it should be kept within the same block to make it sequential execution.

Node.js is a single-threaded application, but it can support concurrency via the concept of **event** and **callbacks**. Every API of Node.js is asynchronous and being single-threaded, they use **async function calls** to maintain concurrency. Node uses observer pattern. Node thread keeps an event loop and whenever a task gets completed, it fires the corresponding event which signals the event-listener function to execute.

## Event-Driven Programming

Node.js uses events heavily and it is also one of the reasons why Node.js is pretty fast compared to other similar technologies. As soon as Node starts its server, it simply initiates its variables, declares functions and then simply waits for the event to occur.

In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function when one of those events is detected.



Although events look quite similar to callbacks, the difference lies in the fact that callback functions are called when an asynchronous function returns its result, whereas event handling works on the observer pattern. The functions that listen to events act as **Observers**. Whenever an event gets fired, its listener function starts executing. Node.js has multiple in-built events available through events module and EventEmitter class which are used to bind events and event-listeners as follows −

// Import events module

var events = require('events');

// Create an eventEmitter object

var eventEmitter = new events.EventEmitter();

Following is the syntax to bind an event handler with an event −

// Bind event and even handler as follows

eventEmitter.on('eventName', eventHandler);

We can fire an event programmatically as follows −

// Fire an event

eventEmitter.emit('eventName');

## Example

Create a js file named main.js with the following code −

// Import events module

var events = require('events');

// Create an eventEmitter object

var eventEmitter = new events.EventEmitter();

// Create an event handler as follows

var connectHandler = function connected() {

console.log('connection succesful.');

// Fire the data\_received event

eventEmitter.emit('data\_received');

}

// Bind the connection event with the handler

eventEmitter.on('connection', connectHandler);

// Bind the data\_received event with the anonymous function

eventEmitter.on('data\_received', function(){

console.log('data received succesfully.');

});

// Fire the connection event

eventEmitter.emit('connection');

console.log("Program Ended.");

Now let's try to run the above program and check its output −

$ node main.js

IT should produce the following result −

connection successful.

data received successfully.

Program Ended.

## How Node Applications Work?

In Node Application, any async function accepts a callback as the last parameter and a callback function accepts an error as the first parameter. Let's revisit the previous example again. Create a text file named input.txt with the following content.

Tutorials Point is giving self learning content

to teach the world in simple and easy way!!!!!

Create a js file named main.js having the following code −

var fs = require("fs");

fs.readFile('input.txt', function (err, data) {

if (err){

console.log(err.stack);

return;

}

console.log(data.toString());

});

console.log("Program Ended");

Here fs.readFile() is a async function whose purpose is to read a file. If an error occurs during the read operation, then the **err object** will contain the corresponding error, else data will contain the contents of the file. **readFile** passes err and data to the callback function after the read operation is complete, which finally prints the content.

Program Ended

Tutorials Point is giving self learning content

to teach the world in simple and easy way!!!!!

Node.js - Event Emitter

Many objects in a Node emit events, for example, a net.Server emits an event each time a peer connects to it, an fs.readStream emits an event when the file is opened. All objects which emit events are the instances of events.EventEmitter.

EventEmitter Class

As we have seen in the previous section, EventEmitter class lies in the events module. It is accessible via the following code −

// Import events module

var events = require('events');

// Create an eventEmitter object

var eventEmitter = new events.EventEmitter();

When an EventEmitter instance faces any error, it emits an 'error' event. When a new listener is added, 'newListener' event is fired and when a listener is removed, 'removeListener' event is fired.

EventEmitter provides multiple properties like **on** and **emit**. **on** property is used to bind a function with the event and **emit** is used to fire an event.

Methods

|  |  |
| --- | --- |
| **S.No.** | **Method & Description** |
| 1 | **addListener(event, listener)**  Adds a listener at the end of the listeners array for the specified event. No checks are made to see if the listener has already been added. Multiple calls passing the same combination of event and listener will result in the listener being added multiple times. Returns emitter, so calls can be chained. |
| 2 | **on(event, listener)**  Adds a listener at the end of the listeners array for the specified event. No checks are made to see if the listener has already been added. Multiple calls passing the same combination of event and listener will result in the listener being added multiple times. Returns emitter, so calls can be chained. |
| 3 | **once(event, listener)**  Adds a one time listener to the event. This listener is invoked only the next time the event is fired, after which it is removed. Returns emitter, so calls can be chained. |
| 4 | **removeListener(event, listener)**  Removes a listener from the listener array for the specified event. **Caution −** It changes the array indices in the listener array behind the listener. removeListener will remove, at most, one instance of a listener from the listener array. If any single listener has been added multiple times to the listener array for the specified event, then removeListener must be called multiple times to remove each instance. Returns emitter, so calls can be chained. |
| 5 | **removeAllListeners([event])**  Removes all listeners, or those of the specified event. It's not a good idea to remove listeners that were added elsewhere in the code, especially when it's on an emitter that you didn't create (e.g. sockets or file streams). Returns emitter, so calls can be chained. |
| 6 | **setMaxListeners(n)**  By default, EventEmitters will print a warning if more than 10 listeners are added for a particular event. This is a useful default which helps finding memory leaks. Obviously not all Emitters should be limited to 10. This function allows that to be increased. Set to zero for unlimited. |
| 7 | **listeners(event)**  Returns an array of listeners for the specified event. |
| 8 | **emit(event, [arg1], [arg2], [...])**  Execute each of the listeners in order with the supplied arguments. Returns true if the event had listeners, false otherwise. |

Class Methods

|  |  |
| --- | --- |
| **S.No.** | **Method & Description** |
| 1 | **listenerCount(emitter, event)**  Returns the number of listeners for a given event. |

Events

|  |  |
| --- | --- |
| **S.No.** | **Events & Description** |
| 1 | **newListener**   * **event** − String: the event name * **listener** − Function: the event handler function   This event is emitted any time a listener is added. When this event is triggered, the listener may not yet have been added to the array of listeners for the event. |
| 2 | **removeListener**   * **event** − String The event name * **listener** − Function The event handler function   This event is emitted any time someone removes a listener. When this event is triggered, the listener may not yet have been removed from the array of listeners for the event. |

Example

Create a js file named main.js with the following Node.js code −

var events = require('events');

var eventEmitter = new events.EventEmitter();

// listener #1

var listner1 = function listner1() {

console.log('listner1 executed.');

}

// listener #2

var listner2 = function listner2() {

console.log('listner2 executed.');

}

// Bind the connection event with the listner1 function

eventEmitter.addListener('connection', listner1);

// Bind the connection event with the listner2 function

eventEmitter.on('connection', listner2);

var eventListeners = require('events').EventEmitter.listenerCount

(eventEmitter,'connection');

console.log(eventListeners + " Listner(s) listening to connection event");

// Fire the connection event

eventEmitter.emit('connection');

// Remove the binding of listner1 function

eventEmitter.removeListener('connection', listner1);

console.log("Listner1 will not listen now.");

// Fire the connection event

eventEmitter.emit('connection');

eventListeners = require('events').EventEmitter.listenerCount(eventEmitter,'connection');

console.log(eventListeners + " Listner(s) listening to connection event");

console.log("Program Ended.");

Now run the main.js to see the result −

$ node main.js

Verify the Output.

2 Listner(s) listening to connection event

listner1 executed.

listner2 executed.

Listner1 will not listen now.

listner2 executed.

1 Listner(s) listening to connection event

Program Ended.

Node.js - Web Module

What is a Web Server?

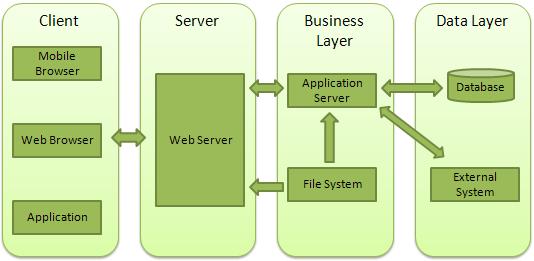
A Web Server is a software application which handles HTTP requests sent by the HTTP client, like web browsers, and returns web pages in response to the clients. Web servers usually deliver html documents along with images, style sheets, and scripts.

Most of the web servers support server-side scripts, using scripting languages or redirecting the task to an application server which retrieves data from a database and performs complex logic and then sends a result to the HTTP client through the Web server.

Apache web server is one of the most commonly used web servers. It is an open source project.

Web Application Architecture

A Web application is usually divided into four layers −



* **Client** − This layer consists of web browsers, mobile browsers or applications which can make HTTP requests to the web server.
* **Server** − This layer has the Web server which can intercept the requests made by the clients and pass them the response.
* **Business** − This layer contains the application server which is utilized by the web server to do the required processing. This layer interacts with the data layer via the database or some external programs.
* **Data** − This layer contains the databases or any other source of data.

Creating a Web Server using Node

Node.js provides an **http** module which can be used to create an HTTP client of a server. Following is the bare minimum structure of the HTTP server which listens at 8081 port.

Create a js file named server.js −

**File: server.js**

var http = require('http');

var fs = require('fs');

var url = require('url');

// Create a server

http.createServer( function (request, response) {

// Parse the request containing file name

var pathname = url.parse(request.url).pathname;

// Print the name of the file for which request is made.

console.log("Request for " + pathname + " received.");

// Read the requested file content from file system

fs.readFile(pathname.substr(1), function (err, data) {

if (err) {

console.log(err);

// HTTP Status: 404 : NOT FOUND

// Content Type: text/plain

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

}else {

//Page found

// HTTP Status: 200 : OK

// Content Type: text/plain

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

// Write the content of the file to response body

response.write(data.toString());

}

// Send the response body

response.end();

});

}).listen(8081);

// Console will print the message

console.log('Server running at http://127.0.0.1:8081/');

Next let's create the following html file named index.htm in the same directory where you created server.js.

**File: index.htm**

<html>

<head>

<title>Sample Page</title>

</head>

<body>

Hello World!

</body>

</html>

Now let us run the server.js to see the result −

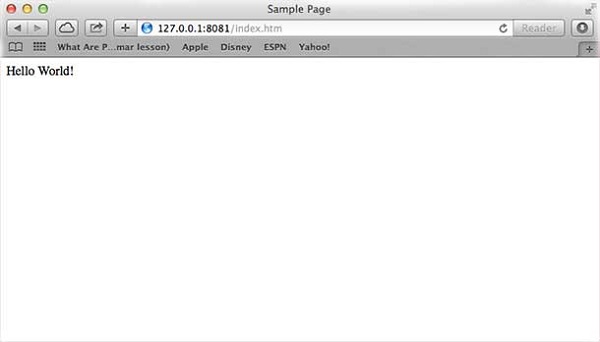
$ node server.js

Verify the Output.

Server running at http://127.0.0.1:8081/

Make a request to Node.js server

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



Verify the Output at server end.

Server running at http://127.0.0.1:8081/

Request for /index.htm received.

Creating Web client using Node

A web client can be created using **http** module. Let's check the following example.

Create a js file named client.js −

**File: client.js**

var http = require('http');

// Options to be used by request

var options = {

host: 'localhost',

port: '8081',

path: '/index.htm'

};

// Callback function is used to deal with response

var callback = function(response){

// Continuously update stream with data

var body = '';

response.on('data', function(data) {

body += data;

});

response.on('end', function() {

// Data received completely.

console.log(body);

});

}

// Make a request to the server

var req = http.request(options, callback);

req.end();

Now run the client.js from a different command terminal other than server.js to see the result −

$ node client.js

Verify the Output.

<html>

<head>

<title>Sample Page</title>

</head>

<body>

Hello World!

</body>

</html>

Verify the Output at server end.

Server running at http://127.0.0.1:8081/

Request for /index.htm received.

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 3000 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.



Request & Response

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

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

// --

})

* [Request Object](https://www.tutorialspoint.com/nodejs/nodejs_request_object.htm) − The request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.
* [Response Object](https://www.tutorialspoint.com/nodejs/nodejs_response_object.htm) − 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 −

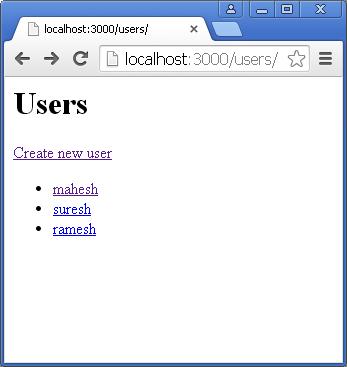
Example app listening at http://0.0.0.0:8081

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



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



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



Serving Static Files

Express provides a built-in middleware **express.static** to serve static files, such as images, CSS, JavaScript, etc.

You simply need to pass the name of the directory where you keep your static assets, to the **express.static** middleware to start serving the files directly. For example, if you keep your images, CSS, and JavaScript files in a directory named public, you can do this −

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

We will keep a few images in **public/images** sub-directory as follows −

node\_modules

server.js

public/

public/images

public/images/logo.png

Let's modify "Hello Word" app to add the functionality to handle static files.

var express = require('express');

var app = express();

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

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

Now open http://127.0.0.1:8081/images/logo.png in any browser and see observe following result.



GET Method

Here is a simple example which passes two values using HTML FORM GET method. We are going to use **process\_get** router inside server.js to handle this input.

<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>

Let's save above code in index.htm and modify server.js to handle home page requests as well as the input sent by the HTML form.

var express = require('express');

var app = express();

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

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

res.sendFile( \_\_dirname + "/" + "index.htm" );

})

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

// Prepare output in JSON format

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(8081, function () {

var host = server.address().address

var port = server.address().port

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

})

Accessing the HTML document using *http://127.0.0.1:8081/index.htm* will generate the following form −

Top of Form

|  |  |
| --- | --- |
| First Name: |  |
| Last Name: |  |
|  | |

Bottom of Form

Now you can enter the First and Last Name and then click submit button to see the result and it should return the following result −

{"first\_name":"John","last\_name":"Paul"}

POST Method

Here is a simple example which passes two values using HTML FORM POST method. We are going to use **process\_get** router inside server.js to handle this input.

<html>

<body>

<form action = "http://127.0.0.1:8081/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>

Let's save the above code in index.htm and modify server.js to handle home page requests as well as the input sent by the HTML form.

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.htm', function (req, res) {

res.sendFile( \_\_dirname + "/" + "index.htm" );

})

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(8081, function () {

var host = server.address().address

var port = server.address().port

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

})

Accessing the HTML document using *http://127.0.0.1:8081/index.htm* will generate the following form −

Top of Form

|  |  |
| --- | --- |
| First Name: |  |
| Last Name: |  |
|  | |

Bottom of Form

Now you can enter the First and Last Name and then click the submit button to see the following result −

{"first\_name":"John","last\_name":"Paul"}

File Upload

The following HTML code creates a file uploader form. This form has method attribute set to **POST** and enctype attribute is set to **multipart/form-data**

<html>

<head>

<title>File Uploading Form</title>

</head>

<body>

<h3>File Upload:</h3>

Select a file to upload: <br />

<form action = "http://127.0.0.1:8081/file\_upload" method = "POST"

enctype = "multipart/form-data">

<input type="file" name="file" size="50" />

<br />

<input type = "submit" value = "Upload File" />

</form>

</body>

</html>

Let's save above code in index.htm and modify server.js to handle home page requests as well as file upload.

var express = require('express');

var app = express();

var fs = require("fs");

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

var multer = require('multer');

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

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

app.use(multer({ dest: '/tmp/'}));

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

res.sendFile( \_\_dirname + "/" + "index.htm" );

})

app.post('/file\_upload', function (req, res) {

console.log(req.files.file.name);

console.log(req.files.file.path);

console.log(req.files.file.type);

var file = \_\_dirname + "/" + req.files.file.name;

fs.readFile( req.files.file.path, function (err, data) {

fs.writeFile(file, data, function (err) {

if( err ){

console.log( err );

}else{

response = {

message:'File uploaded successfully',

filename:req.files.file.name

};

}

console.log( response );

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

});

});

})

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)

})

Accessing the HTML document using *http://127.0.0.1:8081/index.htm* will generate the following form −

**File Upload:**

Select a file to upload:

NOTE: This is just dummy form and would not work, but it must work at your server.

Cookies Management

You can send cookies to a Node.js server which can handle the same using the following middleware option. Following is a simple example to print all the cookies sent by the client.

var express = require('express')

var cookieParser = require('cookie-parser')

var app = express()

app.use(cookieParser())

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

console.log("Cookies: ", req.cookies)

})

app.listen(8081)

Node.js - RESTful API

What is REST architecture?

REST stands for REpresentational State Transfer. REST is web standards based architecture and uses HTTP Protocol. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. REST was first introduced by Roy Fielding in 2000.

A REST Server simply provides access to resources and REST client accesses and modifies the resources using HTTP protocol. Here each resource is identified by URIs/ global IDs. REST uses various representation to represent a resource like text, JSON, XML but JSON is the most popular one.

HTTP methods

Following four HTTP methods are commonly used in REST based architecture.

* **GET** - This is used to provide a read only access to a resource.
* **PUT** - This is used to create a new resource.
* **DELETE** - This is used to remove a resource.
* **POST** - This is used to update a existing resource or create a new resource.

RESTful Web Services

A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., communication between Java and Python, or Windows and Linux applications) is due to the use of open standards.

Web services based on REST Architecture are known as RESTful web services. These webservices uses HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI, Uniform Resource Identifier a service, which provides resource representation such as JSON and set of HTTP Methods.

Creating RESTful for A Library

Consider we have a JSON based database of users having the following users in a file **users.json**:

{

"user1" : {

"name" : "mahesh",

"password" : "password1",

"profession" : "teacher",

"id": 1

},

"user2" : {

"name" : "suresh",

"password" : "password2",

"profession" : "librarian",

"id": 2

},

"user3" : {

"name" : "ramesh",

"password" : "password3",

"profession" : "clerk",

"id": 3

}

}

Based on this information we are going to provide following RESTful APIs.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S. N.** | **URI** | **HTTP Method** | **POST body** | **Result** |
| 1 | listUsers | GET | empty | Show list of all the users. |
| 2 | addUser | POST | JSON String | Add details of new user. |
| 3 | deleteUser | DELETE | JSON String | Delete an existing user. |
| 4 | :id | GET | empty | Show details of a user. |

I'm keeping most of the part of all the examples in the form of hard coding assuming you already know how to pass values from front end using Ajax or simple form data and how to process them using express **Request** object.

List Users

Let's implement our first RESTful API **listUsers** using the following code in a server.js file:

*server.js*

var express = require('express');

var app = express();

var fs = require("fs");

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

fs.readFile( \_\_dirname + "/" + "users.json", 'utf8', function (err, data) {

console.log( data );

res.end( data );

});

})

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)

})

Now try to access defined API using *URL: http://127.0.0.1:8081/listUsers* and *HTTP Method : GET* on local machine using any REST client. This should produce following result:

You can change given IP address when you will put the solution in production environment.

{

"user1" : {

"name" : "mahesh",

"password" : "password1",

"profession" : "teacher",

"id": 1

},

"user2" : {

"name" : "suresh",

"password" : "password2",

"profession" : "librarian",

"id": 2

},

"user3" : {

"name" : "ramesh",

"password" : "password3",

"profession" : "clerk",

"id": 3

}

}

Add User

Following API will show you how to add new user in the list. Following is the detail of the new user:

user = {

"user4" : {

"name" : "mohit",

"password" : "password4",

"profession" : "teacher",

"id": 4

}

}

You can accept the same input in the form of JSON using Ajax call but for teaching point of view, we are making it hard coded here. Following is the **addUser** API to a new user in the database:

*server.js*

var express = require('express');

var app = express();

var fs = require("fs");

var user = {

"user4" : {

"name" : "mohit",

"password" : "password4",

"profession" : "teacher",

"id": 4

}

}

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

// First read existing users.

fs.readFile( \_\_dirname + "/" + "users.json", 'utf8', function (err, data) {

data = JSON.parse( data );

data["user4"] = user["user4"];

console.log( data );

res.end( JSON.stringify(data));

});

})

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)

})

Now try to access defined API using *URL: http://127.0.0.1:8081/addUser* and *HTTP Method : POST* on local machine using any REST client. This should produce following result:

{

"user1":{"name":"mahesh","password":"password1","profession":"teacher","id":1},

"user2":{"name":"suresh","password":"password2","profession":"librarian","id":2},

"user3":{"name":"ramesh","password":"password3","profession":"clerk","id":3},

"user4":{"name":"mohit","password":"password4","profession":"teacher","id":4}

}

Show Detail

Now we will implement an API which will be called using user ID and it will display the detail of the corresponding user.

*server.js*

var express = require('express');

var app = express();

var fs = require("fs");

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

// First read existing users.

fs.readFile( \_\_dirname + "/" + "users.json", 'utf8', function (err, data) {

users = JSON.parse( data );

var user = users["user" + req.params.id]

console.log( user );

res.end( JSON.stringify(user));

});

})

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)

})

Now try to access defined API using *URL: http://127.0.0.1:8081/2* and *HTTP Method : GET* on local machine using any REST client. This should produce following result:

{"name":"suresh","password":"password2","profession":"librarian","id":2}

Delete User

This API is very similar to addUser API where we receive input data through req.body and then based on user ID we delete that user from the database. To keep our program simple we assume we are going to delete user with ID 2.

*server.js*

var express = require('express');

var app = express();

var fs = require("fs");

var id = 2;

app.delete('/deleteUser', function (req, res) {

// First read existing users.

fs.readFile( \_\_dirname + "/" + "users.json", 'utf8', function (err, data) {

data = JSON.parse( data );

delete data["user" + 2];

console.log( data );

res.end( JSON.stringify(data));

});

})

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)

})

Now try to access defined API using *URL: http://127.0.0.1:8081/deleteUser* and *HTTP Method : DELETE* on local machine using any REST client. This should produce following result:

{"user1":{"name":"mahesh","password":"password1","profession":"teacher","id":1},

"user3":{"name":"ramesh","password":"password3","profession":"clerk","id":3}}