What is Node.js?

Node.js is a server side platform built on Google Chrome's JavaScript Engine(V8 Engine). Node.js was developed by Ryan Dahl in 2009 and it's latest version is v0.12.2

Definition of Node.js as put by its official documentation is as follows:

Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications.

Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

Node.js is an open source, cross-platform Runtime Environment for server-side and networking applications.

Node.js applications are written in JavaScript, and can be run within the Node.js runtime on OS X, Microsoft Windows, and Linux.

Node.js also provides a rich library of various JavaScript modules which eases the developement of web application using Node.js to a great extents.

```
Node.js = Runtime Environment + JavaScript Library
```

Features of Node.js

Following are few of the important features which are making Node.js as the first choice of software architects.

- **Asynchronous and Event Driven** All APIs of Node.js library are asynchronous that is non-blocking. It essentially means a Node.js based server never waits for a API to return data. Server moves to next API after calling it and a notification mechanism of Events of Node.js helps server to get response from the previous API call.
- **Very Fast** Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.
- **Single Threaded but highly Scalable** - Node.js uses a single threaded model with event looping. Event mechanism helps server to respond in a non-blocking ways and makes server highly scalable as opposed to traditional servers which create limited threads to handle requests. Node.js uses a single threaded program and same program can services much larger number of requests than traditional server like Apache HTTP Server.
- **No Buffering** - Node.js applications never buffer any data. These applications simply output the data in chunks.

Who Uses Node.js?

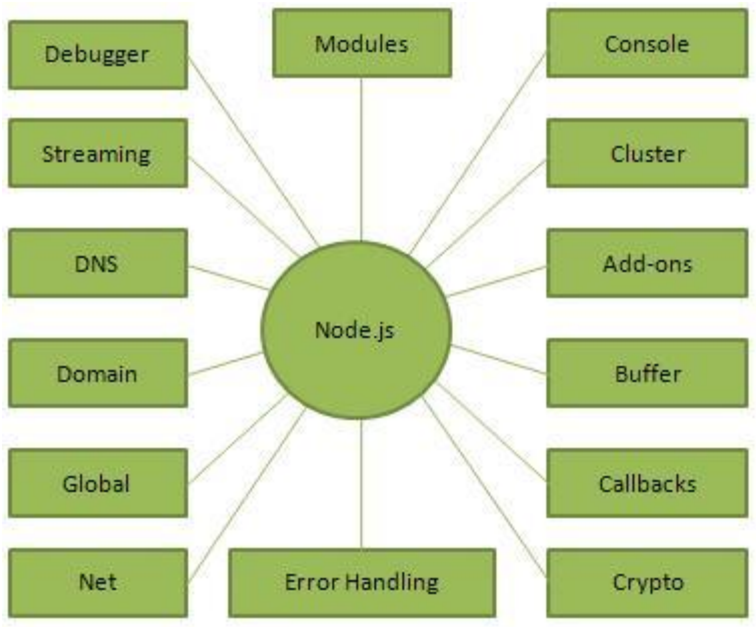
Following is the link on github wiki containing an exhaustive list of projects, application and companies which are using Node.js.

This list include eBay, General Electric, GoDaddy, Microsoft, PayPal, Uber, Wikipins, Yahoo!, Yammer and the list continues.

https://github.com/joyent/node/wiki/projects,-applications,-and-companies-using-node

Concepts

The following diagram depicts some important parts of Node.js which we will discuss in detail in the subsequent sessions.



Where to Use Node.js?

Following are the areas where Node.js is proving itself a perfect technology partner.

- I/O bound Applications
- Data Streaming Applications
- Data Intensive Real-time Applications (DIRT)
- JSON APIs based Applications
- Single Page Applications
- Create an **HTTP server**
- Create a **TCP server** similar to HTTP server
- You can create a **DNS server**.
- You can create a **Static File Server**.
- Node.js can also be used for creating online games, collaboration tools or anything which sends updates to the user in real-time.

Where Not to Use Node.js?

It is not advisable to use Node.js for CPU intensive applications.

Local Environment Setup

If you are still willing to set up your environment for Node.js, you need the following two software's available on your computer:

(a) Text Editor and (b) The Node.js binary installable.

Text Editor

This will be used to type your program. Examples of few editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.

Name and version of text editor can vary on different operating systems.

For example, Notepad will be used on Windows, and vim or vi can be used on windows as well as Linux or UNIX.

The files you create with your editor are called source files and contain program source code. The source files for Node.js programs are typically named with the extension "**.js**".

Before starting your programming, make sure you have one text editor in place and you have enough experience to write a computer program, save it in a file, and finally execute it.

The Node.js Runtime

The source code written in source file is simply javascript.

The Node.js interpreter will be used to interpret and execute your javascript code.

Node.js distribution comes as a binary installable for SunOS , Linux, Mac OS X, and Windows operating systems with the 32-bit (386) and 64-bit (amd64) x86 processor architectures.

Following section guides you on how to install Node.js binary distribution on various OS.

Download Node.js archive

Download latest version of Node.js installable archive file from Node.js Downloads.

You can get the latest Version from nodejs.org website.

OS	Archive name
Windows	node-v0.12.0-x64.msi
Linux	node-v0.12.0-linux-x86.tar.gz
Mac	node-v0.12.0-darwin-x86.tar.gz
SunOS	node-v0.12.0-sunos-x86.tar.gz

Installation on UNIX/Linux/Mac OS X, and SunOS

Based on your OS architecture, download and extract the archive node-v0.12.0-**osname**.tar.gz into /tmp, and then finally move extracted files into /usr/local/nodejs directory.

For example:

```
$ cd /tmp
$ wget http://nodejs.org/dist/v0.12.0/node-v0.12.0-linux-x64.tar.gz
$ tar xvfz node-v0.12.0-linux-x64.tar.gz
$ mkdir -p /usr/local/nodejs
$ mv node-v0.12.0-linux-x64/* /usr/local/nodejs
```

Add /usr/local/nodejs/bin to the PATH environment variable.

OS	Output
Linux	export PATH=$PATH:/usr/local/nodejs/bin
Mac	export PATH=$PATH:/usr/local/nodejs/bin
FreeBSD	export PATH=$PATH:/usr/local/nodejs/bin

Installation on Windows

Use the MSI file and follow the prompts to install the Node.js.

By default, the installer uses the Node.js distribution in C:\Program Files\nodejs.

The installer should set the C:\Program Files\nodejs\bin directory in window's PATH environment variable.

Restart any open command prompts for the change to take effect.

Verify installation: Executing a File

Create a js file named **main.js** on your machine (Windows or Linux) having the following code.

```
/* Hello, World! program in node.js */
console.log("Hello, World!")
```

Now execute main.js file using Node.js interpreter to see the result:

```
$ node main.js
```

A Node.js application consists of following three important parts:

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

Creating Node.js Application

Step 1 - Import required module

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

```
var http = require("http");
```

Step 2: Create Server

At next step we use created http instance and call **http.createServer()** method to create server instance and then we bind it at port 8081 using **listen** method associated with 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/');
```

Above code is enough to create an HTTP server which listens ie. wait for a request over 8081 port on 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 Node.js server

Open http://127.0.0.1:8081/ in any browser and see the below result.

Node.js - npm

Node Package Manager (npm) provides following two main functionalities:

- Online repositories for node.js packages/modules which are searchable on search.nodejs.org
- Command line utility to install Node.js packages, do version management and dependency management of Node.js packages.

npm comes bundled with Node.js installable after v0.6.3 version.

To verify the same, open console and type following command and see the result:

```
$ npm --version
2.7.4
```

If you are running old version of npm then it’s easy to update it to the latest version.

Just use the following command from root:

```
$ sudo npm install npm -g

/usr/bin/npm -> /usr/lib/node_modules/npm/bin/npm-cli.js
npm@2.7.1 /usr/lib/node_modules/npm
```

Installing Modules using npm

There is a simple syntax to install any Node.js module:

```
$ npm install <Module Name>
```

For example, following is the command to install a famous Node.js web framework module called **express**:

```
$ npm install express
```

Now you can use this module in your js file as following:

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

Global vs Local installation

By default, npm installs any dependency in the local mode.

Here local mode refers to the package installation in node_modules directory lying in the folder where Node application is present.

Locally deployed packages are accessible via require() method.

For example when we installed express module, it created node_modules directory in the current directory where it installed express module.

```
$ ls -l
total 0
drwxr-xr-x 3 root root 20 Mar 17 02:23 node_modules
```

Alternatively you can use **npm ls** command to list down all the locally installed modules.

Globally installed packages/dependencies are stored in system directory.

Such dependencies can be used in CLI (Command Line Interface) function of any node.js but can not be imported using require() in Node application directly.

Now Let's try installing express module using global installation.

```
$ npm install express -g
```

This will produce similar result but module will be installed globally.

Here first line tells about the module version and its location where it is getting installed.

```
express@4.12.2 /usr/lib/node_modules/express
├── merge-descriptors@1.0.0
├── utils-merge@1.0.0
├── cookie-signature@1.0.6
├── methods@1.1.1
├── fresh@0.2.4
├── cookie@0.1.2
├── escape-html@1.0.1
├── range-parser@1.0.2
├── content-type@1.0.1
├── finalhandler@0.3.3
├── vary@1.0.0
├── parseurl@1.3.0
├── content-disposition@0.5.0
├── path-to-regexp@0.1.3
├── depd@1.0.0
├── qs@2.3.3
├── on-finished@2.2.0 (ee-first@1.1.0)
├── etag@1.5.1 (crc@3.2.1)
├── debug@2.1.3 (ms@0.7.0)
├── proxy-addr@1.0.7 (forwarded@0.1.0, ipaddr.js@0.1.9)
├── send@0.12.1 (destroy@1.0.3, ms@0.7.0, mime@1.3.4)
├── serve-static@1.9.2 (send@0.12.2)
├── accepts@1.2.5 (negotiator@0.5.1, mime-types@2.0.10)
└── type-is@1.6.1 (media-typer@0.3.0, mime-types@2.0.10)
```

You can use following command to check all the modules installed globally:

```
$ npm ls -g
```

Using package.json

package.json is present in the root directory of any Node application/module and is used to define the properties of a package.

Let's open package.json of express package present in **node_modules/express/**

```
{
  "name": "express",
  "description": "Fast, unopinionated, minimalist web framework",
  "version": "4.11.2",
  "author": {
    "name": "TJ Holowaychuk",
    "email": "tj@vision-media.ca"
  },
  "contributors": [
    {
      "name": "Aaron Heckmann",
      "email": "aaron.heckmann+github@gmail.com"
    },
    {
      "name": "Ciaran Jessup",
      "email": "ciaranj@gmail.com"
    },
    {
      "name": "Douglas Christopher Wilson",
      "email": "doug@somethingdoug.com"
    },
    {
      "name": "Guillermo Rauch",
      "email": "rauchg@gmail.com"
    },
    {
      "name": "Jonathan Ong",
      "email": "me@jongleberry.com"
    },
    {
      "name": "Roman Shtylman",
      "email": "shtylman+expressjs@gmail.com"
    },
    {
      "name": "Young Jae Sim",
      "email": "hanul@hanul.me"
    }
  ],
  "license": "MIT",
  "repository": {
    "type": "git",
    "url": "https://github.com/strongloop/express"
  },
  "homepage": "http://expressjs.com/",
  "keywords": [
    "express",
    "framework",
    "sinatra",
    "web",
    "rest",
    "restful",
    "router",
    "app",
    "api"
  ],
  "dependencies": {
    "accepts": "~1.2.3",
    "content-disposition": "0.5.0",
    "cookie-signature": "1.0.5",
    "debug": "~2.1.1",
    "depd": "~1.0.0",
    "escape-html": "1.0.1",
    "etag": "~1.5.1",
    "finalhandler": "0.3.3",
    "fresh": "0.2.4",
    "media-typer": "0.3.0",
    "methods": "~1.1.1",
    "on-finished": "~2.2.0",
    "parseurl": "~1.3.0",
    "path-to-regexp": "0.1.3",
    "proxy-addr": "~1.0.6",
    "qs": "2.3.3",
    "range-parser": "~1.0.2",
    "send": "0.11.1",
    "serve-static": "~1.8.1",
    "type-is": "~1.5.6",
    "vary": "~1.0.0",
    "cookie": "0.1.2",
    "merge-descriptors": "0.0.2",
    "utils-merge": "1.0.0"
  },
  "devDependencies": {
    "after": "0.8.1",
    "ejs": "2.1.4",
    "istanbul": "0.3.5",
    "marked": "0.3.3",
    "mocha": "~2.1.0",
    "should": "~4.6.2",
    "supertest": "~0.15.0",
    "hjs": "~0.0.6",
    "body-parser": "~1.11.0",
    "connect-redis": "~2.2.0",
    "cookie-parser": "~1.3.3",
    "express-session": "~1.10.2",
    "jade": "~1.9.1",
    "method-override": "~2.3.1",
    "morgan": "~1.5.1",
    "multiparty": "~4.1.1",
    "vhost": "~3.0.0"
  },
  "engines": {
    "node": ">= 0.10.0"
  },
  "files": [
    "LICENSE",
    "History.md",
    "Readme.md",
    "index.js",
    "lib/"
  ],
  "scripts": {
    "test": "mocha --require test/support/env --reporter spec --bail --check-
leaks test/ test/acceptance/",
    "test-cov": "istanbul cover node_modules/mocha/bin/_mocha -- --require
test/support/env --reporter dot --check-leaks test/ test/acceptance/",
    "test-tap": "mocha --require test/support/env --reporter tap --check-
leaks test/ test/acceptance/",
    "test-travis": "istanbul cover node_modules/mocha/bin/_mocha --report
lcovonly -- --require test/support/env --reporter spec --check-leaks test/
test/acceptance/"
  },
  "gitHead": "63ab25579bda70b4927a179b580a9c580b6c7ada",
  "bugs": {
    "url": "https://github.com/strongloop/express/issues"
  },
  "_id": "express@4.11.2",
  "_shasum": "8df3d5a9ac848585f00a0777601823faecd3b148",
  "_from": "express@*",
  "_npmVersion": "1.4.28",
  "_npmUser": {
    "name": "dougwilson",
    "email": "doug@somethingdoug.com"
  },
  "maintainers": [
    {
      "name": "tjholowaychuk",
      "email": "tj@vision-media.ca"
    },
    {
      "name": "jongleberry",
      "email": "jonathanrichardong@gmail.com"
    },
    {
      "name": "shtylman",
      "email": "shtylman@gmail.com"
    },
    {
      "name": "dougwilson",
      "email": "doug@somethingdoug.com"
    },
    {
      "name": "aredridel",
      "email": "aredridel@nbtsc.org"
    },
    {
      "name": "strongloop",
      "email": "callback@strongloop.com"
    },
    {
      "name": "rfeng",
      "email": "enjoyjava@gmail.com"
    }
  ],
  "dist": {
    "shasum": "8df3d5a9ac848585f00a0777601823faecd3b148",
    "tarball": "http://registry.npmjs.org/express/-/express-4.11.2.tgz"
  },
  "directories": {},
  "_resolved": "https://registry.npmjs.org/express/-/express-4.11.2.tgz",
  "readme": "ERROR: No README data found!"
}
```

Attributes of Package.json

- **name** - name of the package
- **version** - version of the package
- **description** - description of the package
- **homepage** - homepage of the package
- **author** - author of the package
- **contributors** - name of the contributors to the package
- **dependencies** - list of dependencies. npm automatically installs all the dependencies mentioned here in the node_module folder of the package.
- **repository** - repository type and url of the package
- **main** - entry point of the package
- **keywords** - keywords

Uninstalling a module

Use following command to uninstall a Node.js module.

```
$ npm uninstall express
```

Once npm uninstall the package, you can verify by looking at the content of /node_modules/ directory or type the following command:

```
$ npm ls
```

Updating a module

Update package.json and change the version of the dependency which to be updated and run the following command.

```
$ npm update express
```

Search a module

Search package name using npm.

```
$ npm search express
```

Node.js - Web Module

What is Web Server?

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.

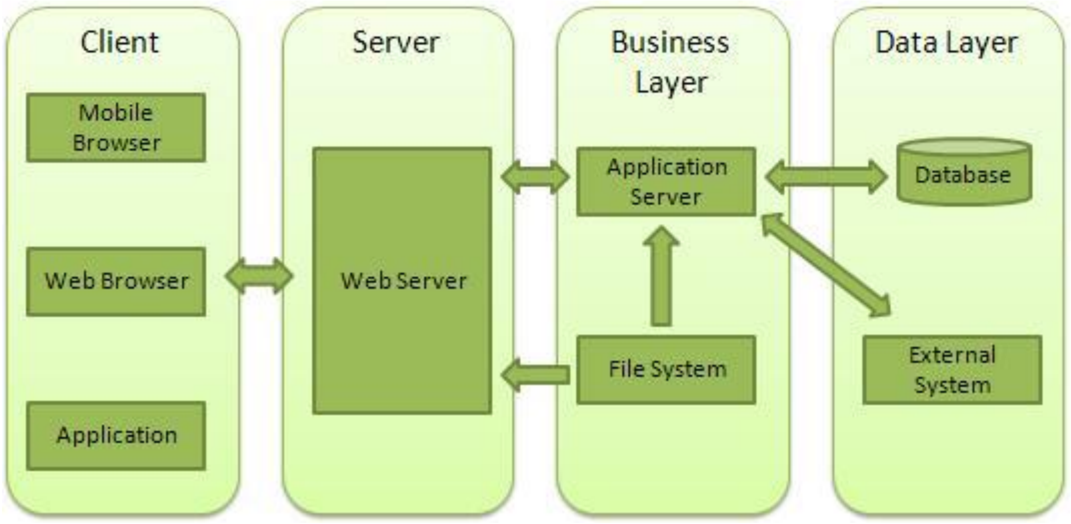
Web servers usually delivers html documents along with images, style sheets and scripts.

Most of the web server support server side scripts using scripting language or redirect to application server which perform the specific task of getting data from database, perform complex logic etc. and then sends a result to the HTTP client through the Web server.

Apache web server is one of the most commonly used web server. 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 request to the web server.

• Server - This layer consists of Web server which can intercepts the request made by clients and pass them the response.

• Business - This layer consists of application server which is utilized by web server to do required processing. This layer interacts with data layer via data base or some external programs.

• Data - This layer consists of databases or any source of data

Creating Web Server using Node

Node.js provides **http** module which can be used to create either HTTP client of server. Following is a bare minimum structure of 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 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/
```

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
```

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 a rapid development of Node based Web applications.

Following are some of the core features of Express framework:

- Allows to set up middleware's to respond to HTTP Requests.
- Defines a routing table which is used to perform different action 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 web application using node terminal.

```
$ npm install express --save
```

Above command saves installation locally in **node_modules** directory and creates a directory express inside node_modules. There are following important modules which you should install 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
```

Request & Response

Express application makes use of a callback function whose parameters are **request** and **response** objects.

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

You can check further detail on both the objects:

- 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 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 add functionality to handle more type 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
```

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

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

Serving Static Files

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

You simply needs 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 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 the below 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 request as well as 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)

})
```

Now accessing HTML document using *http://127.0.0.1:8081/index.htm* will generate following form:

Now you can enter First and Last Name and then click submit button to see the result and it should give result as follows:

```
{"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 above code in index.htm and modify server.js to handle home page request as well as 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)

})
```

Now accessing HTML document using *http://127.0.0.1:8081/index.htm* will generate following form:

Now you can enter First and Last Name and then click submit button to see the result and it should give result as follows:

```
{"first_name":"John","last_name":"Paul"}
```

File Upload

The following HTML code creates a file uploader form.

This form is having 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 request 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)

})
```

Now accessing HTML document using *http://127.0.0.1:8081/index.htm* will generate following form:

Cookies Management

You can send cookies to Node.js server which can handle the 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:

```
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 *http://127.0.0.1:8081/listUsers* on local machine.

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:

```
var express = require('express');
var app = express();
var fs = require("fs");

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

app.get('/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 *http://127.0.0.1:8081/addUsers* on local machine.

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.

```
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)
{
       data = 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 let's call above service using *http://127.0.0.1:8081/2* on local machine. 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.

```
var express = require('express');
var app = express();
var fs = require("fs");

var id = 2;

app.get('/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 let's call above service using *http://127.0.0.1:8081/deleteUser* on local machine.

This should produce following result:

```
{ user1:
   { name: 'mahesh',
     password: 'password1',
     profession: 'teacher',
     id: 1 },
  user3:
   { name: 'ramesh',
     password: 'password3',
     profession: 'clerk',
     id: 3 }
}
```