Contents

[Node 2](#_Toc411133454)

[NPM 2](#_Toc411133455)

[npm registry 3](#_Toc411133456)

[Require 4](#_Toc411133457)

[Loading a File Module 4](#_Toc411133458)

[Loading a Folder Module 5](#_Toc411133459)

[Loading from the node\_modules Folder 5](#_Toc411133460)

[UNDERSTANDING THE STANDARD CALLBACK PATTERN 5](#_Toc411133461)

[UNDERSTANDING THE EVENT EMITTER PATTERN 6](#_Toc411133462)

[Express Framework 6](#_Toc411133463)

[View Engine 7](#_Toc411133464)

[Jade 8](#_Toc411133465)

[EJS (Embedded Java Script) 8](#_Toc411133466)

[Vash (like mvc razor engine) 9](#_Toc411133467)

[Static Resources 9](#_Toc411133468)

[Bower 9](#_Toc411133469)

[Grunt 9](#_Toc411133470)

[The Request and Response Objects 10](#_Toc411133471)

[The Request Object 10](#_Toc411133472)

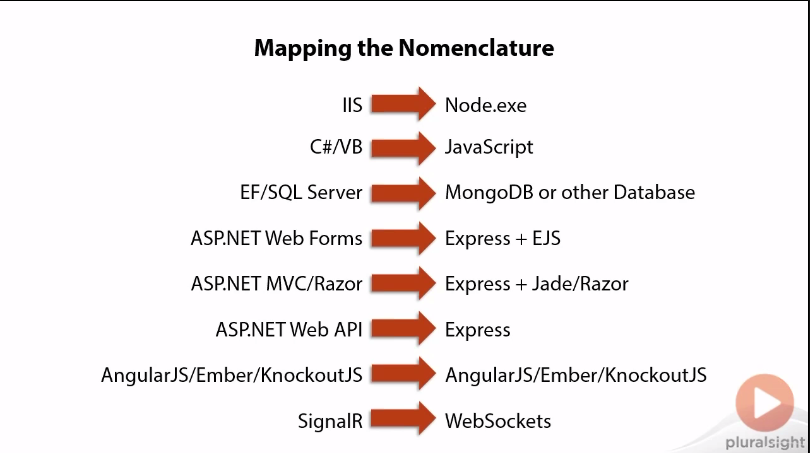
[The Response Object 11](#_Toc411133473)

[Code 13](#_Toc411133474)

# Node

* node --version ( to get Node version )
* npm --version (to get node package manager version)

Node.js is not a web application framework. It's a *platform* for building applications. It may seem like a framework at first glance, though, because much like a framework, it includes a set of tools useful in building web applications (such as modules for writing to the file system or making HTTP requests).

****

# NPM

Npm is the package manager that comes preinstalled with the Node.js server platform. It is used to install node programs from the npm registry. In short, it is a tool that aids a developer as they install and manager third party node programs. Don’t confuse npm with the commonJS require() statement. It is not used to load code, it is used to install code and manage code dependencies from the command line.

Specifically, an npm package, as defined by the npm documentation is:

1. a folder containing a program described by a package.json file
2. a gzipped tarball containing [a]
3. a url that resolves to [b]
4. a <name>@<version> that is published on the registry with [c]
5. a <name>@<tag> that points to [d]
6. a <name> that has a "latest" tag satisfying [e]
7. a git url that, when cloned, results in [a]

Installing npm is simple, given that it is automatically installed when you install Node.js.

$ node --version

$ npm -version

To view the current global settings for npm from the command line run:

$ npm config list -l

## npm registry

Node.js packages which are intended for public use can be published to the npm registry. The point of publishing a package to the registry is so that the package can be used and installed by anyone.

To install the latest version of a package locally, all that is required is to run npm install [package name] from the command line in the directory you want the package installed. This is considered installing a package locally.

When you install a package a couple of things occur.

1. NPM examines the current directory you are calling the npm command from and begins walking up the directory tree structure checking for a folder that contains either a package.json file or a node\_modules folder. When one of these is found, npm will consider that directory, the current directory, for the purpose of running npm commands. If neither are found, then the current directory in which the searching started will be used as the npm current directory.
2. If a ./node\_modules directory is already in the directory determined in step 1, it will place the packages inside of ./node\_modules. If no ./node\_modules directory is found npm will create the directory and then place packages in it.
3. Once packages have been installed npm will attempt to read each package'spackage.json file being installed, and examining it for a list of dependencies. If dependencies are found, npm will install these dependencies in a folder callednode\_modules contained inside of the package's directory being installed.

**Package.json**

{

"name": "PS\_NodejsWebDev\_1",

"version": "0.1.0",

"description": "PS\_NodejsWebDev\_1",

"main": "server.js",

"author": {

"name": "Salman Zafar",

"email": "salman.zafar@gmail.com"

}

}

# Require

var http = require("http"); //library that listen request and respond to them

//creating server

var server = http.createServer(function (request, response) {

console.log(request.url); //logging URl

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

response.write("<html><body><h1>" + request.url + "</h1></body></html>"); //writing back URL

response.end();

});

//listening at port

server.listen(3000);

Better to user some framework to write responses rather to write oneself.

* Npm install express --save (add the dependency in package json)

"dependencies": {

"express": "^4.11.1"

}

Node has several modules compiled into its binary distribution. These are called the core modules, are referred to solely by the module name — not the path — and are preferentially loaded even if a third-party module exists with the same name.

For instance, if you wanted to load and use the http core module, you would do the following:

var http = require('http');

## Loading a File Module

You can also load a non-core module from the fi le system by providing the absolute path like this:

var myModule = require('/home/pedro/my\_modules/my\_module');

Or you can provide a path relative to the current fi le:

var myModule = require('../my\_modules/my\_module');

var myModule2 = require('./lib/my\_module\_2');

Notice here that you can omit the .js fi le termination. When it fails to fi nd such a fi le, Node will look for the path by adding the .js extension. So, if the fi le my\_module.js exists inside the current directory, the two following lines are equivalent:

var myModule = require('./my\_module');

var myModule = require('./my\_module.js');

## Loading a Folder Module

You can use the path for a folder to load a module like this:

var myModule = require('./myModuleDir');

If you do so, Node will search inside that folder. Node will presume this folder is a package and will try to look for a package defi nition. That package defi nition should be a fi le named package.json.

If that folder does not contain a package defi nition fi le named package.json, the package entry point will assume the default value of index.js, and Node will look, in this case, for a fi le under the path ./myModuleDir/index.js.

However, if you place a file named package.json inside the module directory, Node will try to parse that fi le and look for and use the main attribute as a relative path for the entry point. For instance, if your ./myModuleDir/package.json fi le looks something like the following, Node will try to load the fi le with the path./myModuleDir/lib/myModule.js:

## Loading from the node\_modules Folder

If the module name is not relative and is not a core module, Node will try to fi nd it inside the node\_modules folder in the current directory.

For instance, if you do the following, Node will try to look for the file ./node\_modules/myModule.js:

var myModule = require('myModule.js');

If Node fails to find the fi le, it will look inside the parent folder called ../node\_modules/myModule .js. If it fails again, it will try the parent folder and keep descending until it reaches the root or finds the required module.

# UNDERSTANDING THE STANDARD CALLBACK PATTERN

Asynchronous programming does not use function return values to denote that a function is fi nished. Instead it uses the continuation-passing style (CPS):

Continuation-passing style (CPS) is a style of programming in which control is passed explicitly in the form of a continuation. (…)

A function written in continuation-passing style takes as an extra argument an explicit “continuation,” that is, a function of one argument. When the CPS function has computed its result value, it “returns” it by calling the continuation function with this value as the argument.

This is a style in which a function invokes a callback after the operation is complete so that your program can continue. As you will see, JavaScript lends itself to this type of programming. Here is an example in Node that involves loading a fi le into memory:

var fs = require('fs');

fs.readFile('/etc/passwd', function(err, fileContent) {

if (err) {

throw err;

}

console.log('file content', fileContent.toString());

});

Here, you are passing an anonymous inline function as the second argument of the fs.readFile function, and you’re making use of the CPS, because you are continuing the execution of the program inside that function.

As you can see here, the fi rst argument to the callback function is an error object, which will have an instance of the Error class if an error occurs. **This is a common pattern in Node when using CPS**.

# UNDERSTANDING THE EVENT EMITTER PATTERN

When you use an event emitter pattern, two or more objects are involved — the event emitter and one or more event listeners.

An event emitter is an object that — as the name says — emits events. An event listener is a part of the code that binds to the event emitter and listens for certain types of events

var req = http.request(options, function(response) {

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

console.log("some data from the response", data);

});

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

console.log("response ended");

});

});

req.end();

Here, you are looking at some of the steps required to make an HTTP request to a remote HTTP server using the Node http.request API (which is covered later). Line 1 uses the continuation-passing style, passing in an inline function that will be executed once the response is available. The HTTP request API uses the CPS here because the program continues to execute after the http.request function completes.

When complete, the http.request function invokes the callback, passing a response object. This response object is an event emitter and, according to the Node documentation, can emit, among others, the data and end events. You are then registering callback functions that will be invoked every time any of these events happen.

As a rule of thumb, use CPS when you want to regain control after the requested operation completes and use the event emitter pattern when an event can happen multiple times.

# [Express Framework](http://expressjs.com/guide.html#creating-a%20server)

which contains almost everything you need to run a webserver (including cookies, sessions and path routing). Additionally Express supports partials, which take care of your header and footer includes. Express is built on top of [Sencha's Connect](http://www.senchalabs.org/connect/). Cookies and sessions are actually powered by Connect. Express is what simplifies your routing and handles views/partials. So if you don't need all bells and whistles that come with Express you could just go for Connect instead.

//creating express driven web application

var express = require('express'); //adding express dependency

var app = express(); //as its a singleton

app.set(port, process.env.PORT || 3000)

This allows us to override the port by setting an environment value before you start the server.

Note that our custom 404 and 500 pages must be handled slightly differently. Instead of using app.get, it is using app.use. app.use is the method by which Express adds middleware.

You can think of this as a catch-all handler for anything that didn’t get matched by a route. This brings us to a very important point: in Express, the order in which routes and middleware are added is significant. If we put the 404 handler above the routes, the home page and About page would stop working: instead, those URLs would result in a 404

 In this example, the /about/contact and /about/directions handlers will never be matched because the first handler uses a wildcard in its path: /about\*

# View Engine

## Jade

Express has its view engine: Jade

All of the views (html file) while using express needs to be in views folder. Its default hierarchy that followed by express.

//setting up jade => view engine

app.set('view engine','jade');

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

res.render('jade/index', {title: "Express View Engine"});

});

Syntax for jade is HAML.

doctype

html

head

title=title

body

block content

footer This will be footer

## EJS (Embedded Java Script)

Its like webforms

To use Layout in EJS we need to install ejs-locals ( supporting layout)

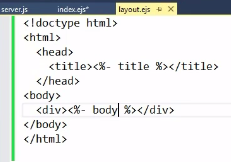
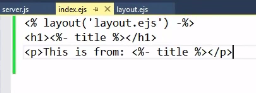
//getting Ejs Locals

var ejsEngine = require('ejs-locals');

app.engine('ejs',ejsEngine); //supprots Master Pages (Layouts)

app.set('view engine', 'ejs'); // ejs view engine like jade

Using Layout and extending it

## Vash (like mvc razor engine)

# Static Resources

//loading static resources from public folder

app.use(express.static(\_\_dirname +'/public'));

## Bower

* npm instal bower --save-dev
* npm install bower –g (installing globally)

This is going to save it in package.json and will save it in Dev dependencies but when it will be on production server it will not used as dependency

# Grunt

* npm install grunt-cli –g

Grunt is a java script task runner.

* npm install grunt-nodemon - -save-dev

What Nodemon do? Basically it restarts node application on any change made in server side js or depend on the options.

To configure it add gruntfile.js for development

//gruntfile.js

module.exports = function (grunt) {

grunt.initConfig({

nodemon: {

script: 'server.js',

options: {

watchedExtensions: ['js']

}

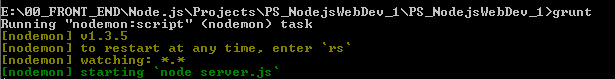
}

});

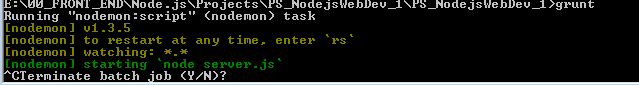
grunt.loadNpmTasks('grunt-nodemon');

grunt.registerTask('default', ['nodemon']);

};

Then go to cmd and root path and use grunt

To terminate this process use ctrl+c



# The Request and Response Objects

* The most common media type for POST bodies is application/x-www-form-urlencoded, which is simply encoded name/value pairs separated by ampersands (essentially the same format as a querystring).
* If the POST needs to support file uploads, the media type is multipart/form-data, which is a morecomplicated format.
* AJAX requests can use application/json for the body.

## The Request Object

The request object starts its life as an instance of http.IncomingMessage, a core Node object. Express adds additional functionality.

Let’s look at the most useful properties and methods of the request object

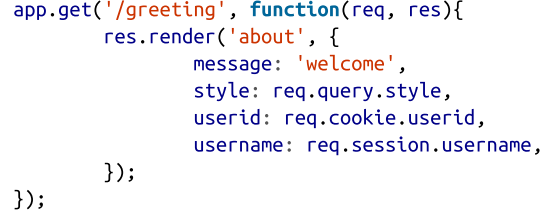
* req.params
  + An array containing the named route parameters.
* req.param(name)
  + Returns the named route parameter, or GET or POST parameters
* req.query
  + An object containing querystring parameters (sometimes called GET parameters) as name/value pairs.
* req.body
  + An object containing POST parameters. It is so named because POST parameters are passed in the body of the REQUEST, not in the URL like query string parameters.
* req.route
  + Information about the currently matched route. Primarily useful for route debugging.
* req.cookies/req.signedCookies
  + Objects containing containing cookie values passed from the client. See Chapter 9.
* req.headers
  + The request headers received from the client.
* req.accepts([types])
  + A convenience method to determine whether the client accepts a given type or types (optional types can be a single MIME type, such as application/json, a comma- delimited list, or an array). This method is of primary interest to those writing public APIs; it is assumed that browsers will always accept HTML by default.
* req.ip
  + The IP address of the client.
* req.path
  + The request path (without protocol, host, port, or querystring).
* req.host
  + A convenience method that returns the hostname reported by the client. This in‐formation can be spoofed and should not be used for security purposes.
* req.xhr
  + A convenience property that returns true if the request originated from an AJAXcall.
* req.protocol
  + The protocol used in making this request (for our purposes, it will either be http or https).
* req.secure
  + A convenience property that returns true if the connection is secure. Equivalent to req.protocol==='https'.
* req.url/req.originalUrl
  + A bit of a misnomer, these properties return the path and querystring (they do not include protocol, host, or port). req.url can be rewritten for internal routing purposes, but req.originalUrl is designed to remain the original request and querystring.
* req.acceptedLanguage
  + sA convenience method that returns an array of the (human) languages the client prefers, in order. This information is parsed from the request header.

## The Response Object

The response object starts its life as an instance of http.ServerResponse, a core Node object

* res.status(code)
  + Sets the HTTP status code. Express defaults to 200 (OK), so you will have to use this method to return a status of 404 (Not Found) or 500 (Server Error), or any other status code you wish to use. For redirects (status codes 301, 302, 303, and 307), there is a method redirect, which is preferable.
* res.set(name, value)
  + Sets a response header. This is not something you will normally be doing manually.
* res.cookie(name, value, [options]), res.clearCookie(name, [options])
  + Sets or clears cookies that will be stored on the client. This requires some middle‐ware support
* res.redirect([status], url)
  + Redirects the browser. The default redirect code is 302 (Found). In general, you should minimize redirection unless you are permanently moving a page, in which case you should use the code 301 (Moved Permanently).
* res.send(body), res.send(status, body)
  + Sends a response to the client, with an optional status code. Express defaults to a content type of text/html, so if you want to change it to text/plain (for example), you’ll have to call res.set('Content-Type', 'text/plain\') before calling
* res.send.
  + If body is an object or an array, the response is sent as JSON instead (with the content type being set appropriately), though if you want to send JSON, I recommend doing so explicitly by calling res.json instead.
* res.json(json), res.json(status, json)
  + Sends JSON to the client with an optional status code.
* res.jsonp(json), res.jsonp(status, json)
  + Sends JSONP to the client with an optional status code.
* res.type(type)
  + A convenience method to set the Content-Type header. Essentially equivalent to res.set('Content-Type', type), except that it will also attempt to map file extensions to an Internet media type if you provide a string without a slash in it. For example, res.type('txt') will result in a Content-Type of text/plain. There are areas where this functionality could be useful (for example, automatically serving disparate multimedia files), but in general, you should avoid it in favor of explicitly setting the correct Internet media type.
* res.format(object)
  + This method allows you to send different content depending on the Accept request header. This is of primary use in APIs, a very simple example: res.format({'text/plain': 'hi there', 'text/html': '<b>hi there</b>'}).
* res.attachment([filename]), res.download(path, [filename], [callback])
  + Both of these methods set a response header called Content-Disposition to at tachment; this will prompt the browser to download the content instead of displaying it in a browser. You may specify filename as a hint to the browser. With res.download, you can specify the file to download, whereas res.attachment justsets the header; you still have to send content to the client.
* res.sendFile(path, [options], [callback])
  + This method will read a file specified by path and send its contents to the client. There should be little need for this method; it’s easier to use the static middleware, and put files you want available to the client in the public directory. However, if you want to have a different resource served from the same URL depending on some condition, this method could come in handy.
* res.links(links)
  + Sets the Links response header. This is a specialized header that has little use in most applications.
* res.locals, res.render(view, [locals], callback)
  + res.locals is an object containing default context for rendering views. res.render will render a view using the configured templating engine (the locals parameter to res.render shouldn’t be confused with res.locals: it will override the context in res.locals, but context not overridden will still be available). Note that res.render will default to a response code of 200; use res.status to specify a different response code.

Passing a context to a view, including querystring, cookie, and session values



# Form Handling

## Encoding

When the form is submitted (either by the browser or via AJAX), it must be encoded somehow. If you don’t explicitly specify an encoding, it defaults to application/x-www-form-urlencoded (this is just a lengthy media type for “URL encoded”). This is a basic, easy-to-use encoding that’s supported by Express out of the box.

If you need to upload files, things get more complicated. There’s no easy way to send files using URL encoding, so you’re forced to use the multipart/form-data encoding type, which is and is not handled directly by Express (actually, Express still supports this encoding, but it will be removed in the next version of Express, and its use is not recommended:).

## Form Handling with Express

If you’re using GET for your form handling, your fields will be available on the req.query object. For example, if you have an HTML input field with a name attribute of email, its value will be passed to the handler as req.query.email. There’s really not much more that needs to be said about this approach: it’s just that simple.

If you’re using POST, you’ll have to link in middleware to parse the URL-encoded body. First, install the body-parser middleware (npm install --save body-parser), then link it in:

app.use(require('body-parser')());

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

console.log('Form (from querystring): ' + req.query.form);

console.log('CSRF token (from hidden form field): ' + req.body.\_csrf);

console.log('Name (from visible form field): ' + req.body.name);

console.log('Email (from visible form field): ' + req.body.email);

res.redirect(303, '/thank-you');

});

## 

## Handling AJAX Forms

Express provides us with a couple of convenience properties, req.xhr and req.accepts. req.xhr will be true if the request is an AJAX request (XHR is short for XML HTTP Request, which is what AJAX relies on). req.accepts will try to determine the most appropriate response type to return. In our case, req.accepts('json,html') is asking if the best format to return is JSON or HTML: this is inferred from the Accepts HTTP header, which is an ordered list of acceptable response types provided by the browser.

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

if(req.xhr || req.accepts('json,html')==='json'){

// if there were an error, we would send { error: 'error description' }

res.send({ success: true });

} else {

// if there were an error, we would redirect to an error page

res.redirect(303, '/thank-you');

}

});

# Cookies in Express

Before you start setting and accessing cookies in your app, you need to include the cookie-parser middleware. First, npm install --save cookie-parser, then:

app.use(require('cookie-parser')(credentials.cookieSecret));

Once you’ve done this, you can set a cookie or a signed cookie anywhere you have access to a request object:

res.cookie('monster', 'nom nom');

res.cookie('signed\_monster', 'nom nom', { signed: true });

To retrieve the value of a cookie (if any) sent from the client, just access the cookie or signedCookie properties of the request object:

var monster = req.cookies.monster;

var signedMonster = req.signedCookies.monster;

# Sessions

install express-session (npm install --save express-session); then, after linking in the cookie parser, link in express-session:

app.use(require('cookie-parser')(credentials.cookieSecret));

app.use(require('express-session')());

The express-session middleware accepts a configuration object with the following options:

* key
  + The name of the cookie that will store the unique session identifier. Defaults to connect.sid.
* store
  + An instance of a session store. Defaults to an instance of MemoryStore, which is fine for our current purposes.
* cookie
  + Cookie settings for the session cookie (path, domain, secure, etc.). Regular cookie defaults apply.

Once you’ve set up sessions, using them couldn’t be simpler: just use properties of the request object’s session variable:

req.session.userName = 'Anonymous';

var colorScheme = req.session.colorScheme || 'dark';

# Code

Creating server manually

//creating server (manualy handling req and resp and writing back)

var server = http.createServer(function (request, response) {

console.log(request.url); //logging URl

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

response.write("<html><body><h1>" + request.url + "</h1></body></html>"); //writing back URL

response.end();

});

//express supports http verbs

// we dont need to write or use end. send method manages itself

//handling root url

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

//res.send("<html><body><h1>" + req.url + "</h1></body></html>");

res.render('index', { title: "Express View Engine" });

});

Setting up view engine and support for Layout

//setting up jade => view engine

app.set('view engine','jade');

//getting Ejs Locals

var ejsEngine = require('ejs-locals');

app.engine('ejs',ejsEngine); //supprots Master Pages (Layouts)

app.set('view engine', 'ejs'); // ejs view engine like jade

Writing API

//for writing api's we can manage as we want

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

res.set('content-type', 'application/json'); //setting header if needed

res.send({ name: 'Salman', company: 'Ascertia' }); //sending json back

});