



Zacky Pickholz



The Node Way

My First Server - HTTP

```
var http = require('http'); // import the http package

// start a web server listening on port 80
http.createServer(function(req, res) {

    // send http content type and status code
    res.writeHead(200, {'Content-Type': 'text/plain'});

    // send http content type and status code
    res.end('Hola Mundo!');

}).listen(80);

// just a console message
console.log('Server running on port 80');
```

My First Server – TCP Server

```
var net= require('net'); // import the net package

// create the socket server (note we're saving it to a variable this time)
var server = net.createServer( function (socket) {

    // greet user
    socket.write(welcome!\r\n');

    // data ready event - echo it back to the socket
    socket.on('data', function(chunk) {
        socket.write(chunk);
    });

    // client closed the connection - close our end as well
    socket.on('end', socket.end);
});

// start the server – listen on port 1337 on all local interfaces
server.listen(1337, '0.0.0.0');

// just a console message
console.log('Server running on port 1337');
```

What is Node.js

- Open source, cross-platform runtime environment
- Used to easily build fast, scalable server-side and networking applications
- Node.js is written in C++
- Node.js applications are written in JavaScript
 - In fact many of its modules are also written in JS
- Uses an event-driven, non-blocking I/O
 - model that makes it lightweight and efficient
- Highly scalable & high throughput
- Has built-in library to allow apps act as Web server

Current Stable Version

- V6.9.2
- 12-12-2016

Who's Behind Node

- Created by Ryan Dahl sponsored by Joyent
- Initiated in search for good push capabilities
- First published for Linux in 2009
- NPM was introduced in 2011
- Today still managed by Joyent



When Node Should Be Used

- For data-intensive real-time applications running across distributed devices
 - Online games, Chat, Data streaming
 - Stock trading dashboards
 - Leader boards
 - Message pushing
 - Messaging applications
 - AJAX heavy mobile/desktop SPAs (single page applications)

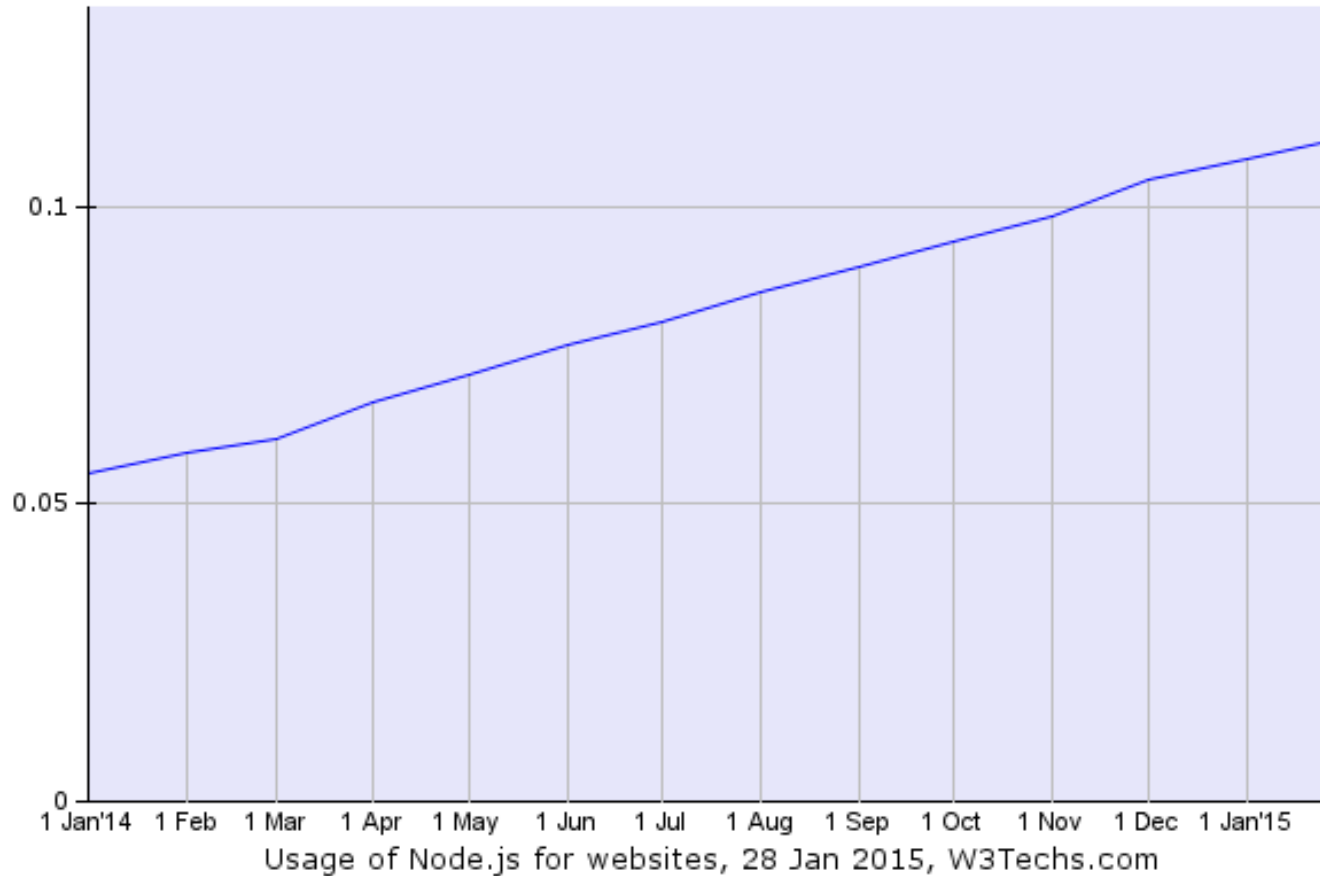
More Reasons Why to Node

- Where we need a persistent server-browser connection
- When end-to-end JavaScript is required
- When we want code reuse across client/server
- When we want a JSON API (Gmail)
- Node.js is great for when you have I/O bound work, not CPU bound

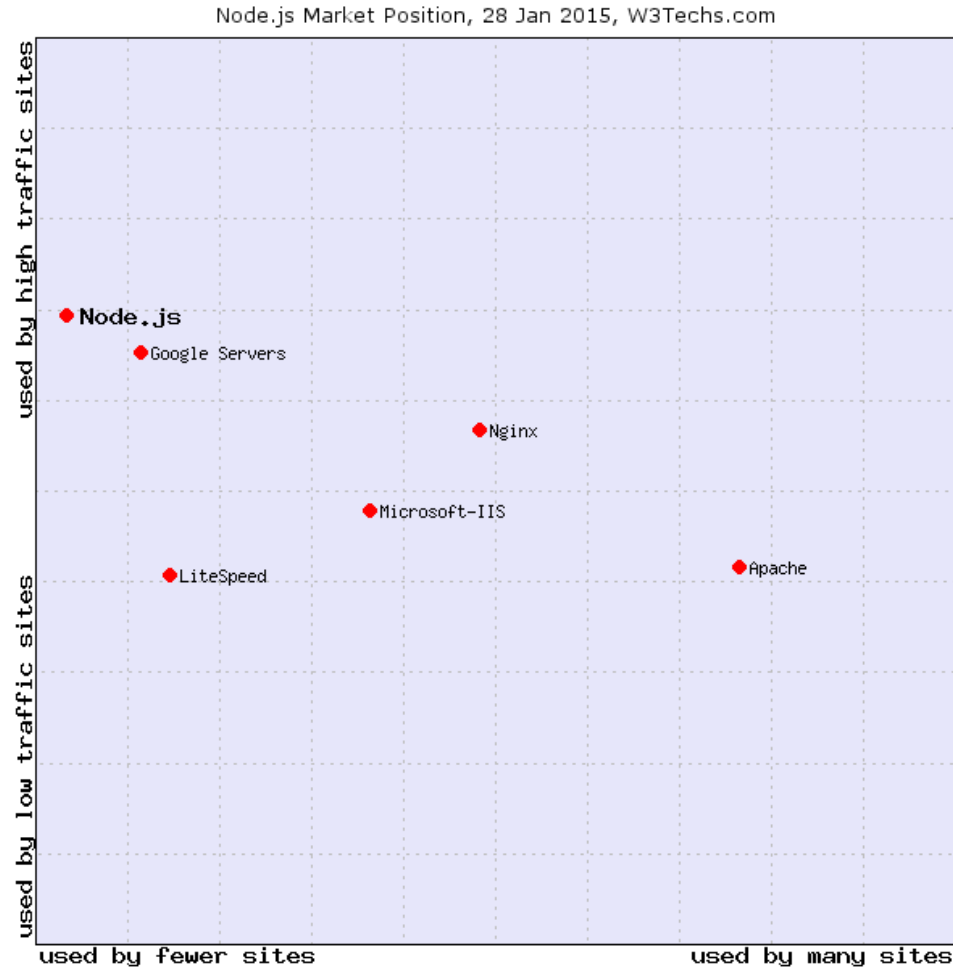
When Node Should NOT Be Used

- Heavy server-side computation/processing
 - CPU intensive operation annuls the throughput benefits of Node
 - Heavy computation should be offloaded to background processes


Historical Trend (%)



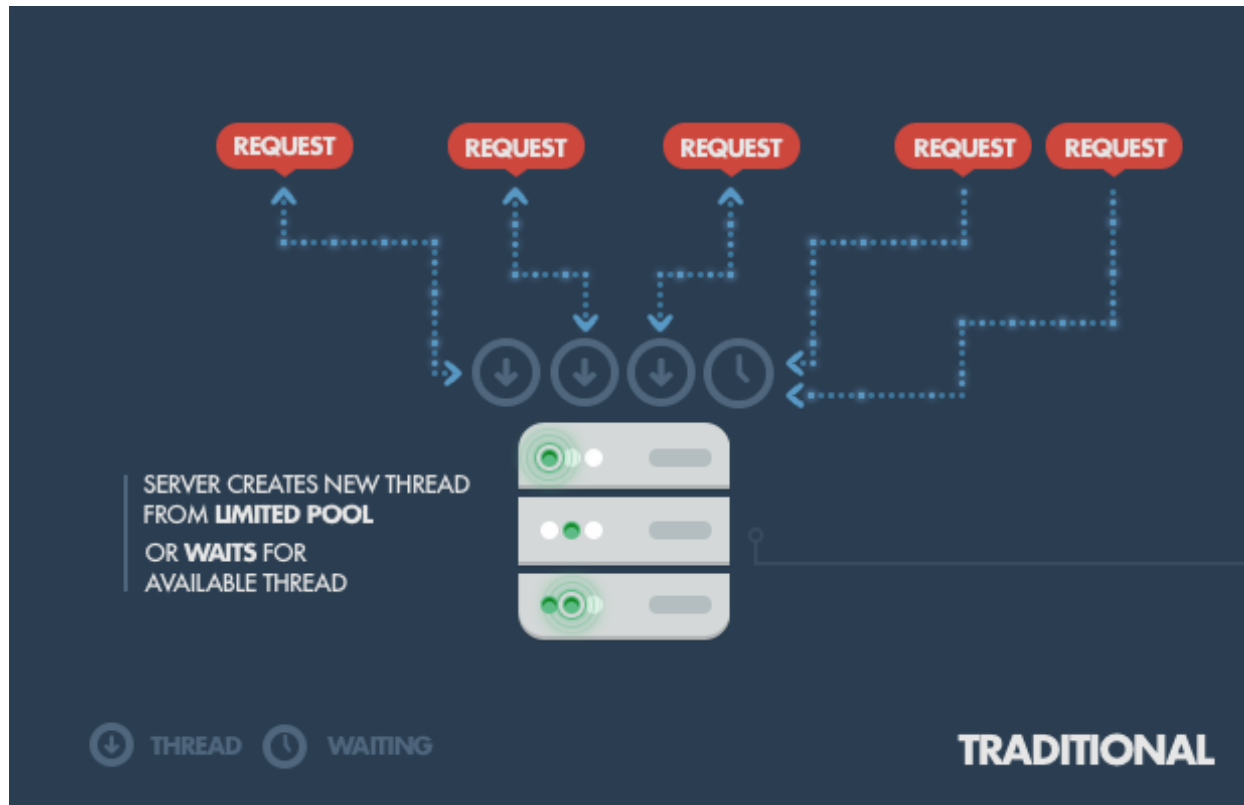
Market Position



The Javascript Engine

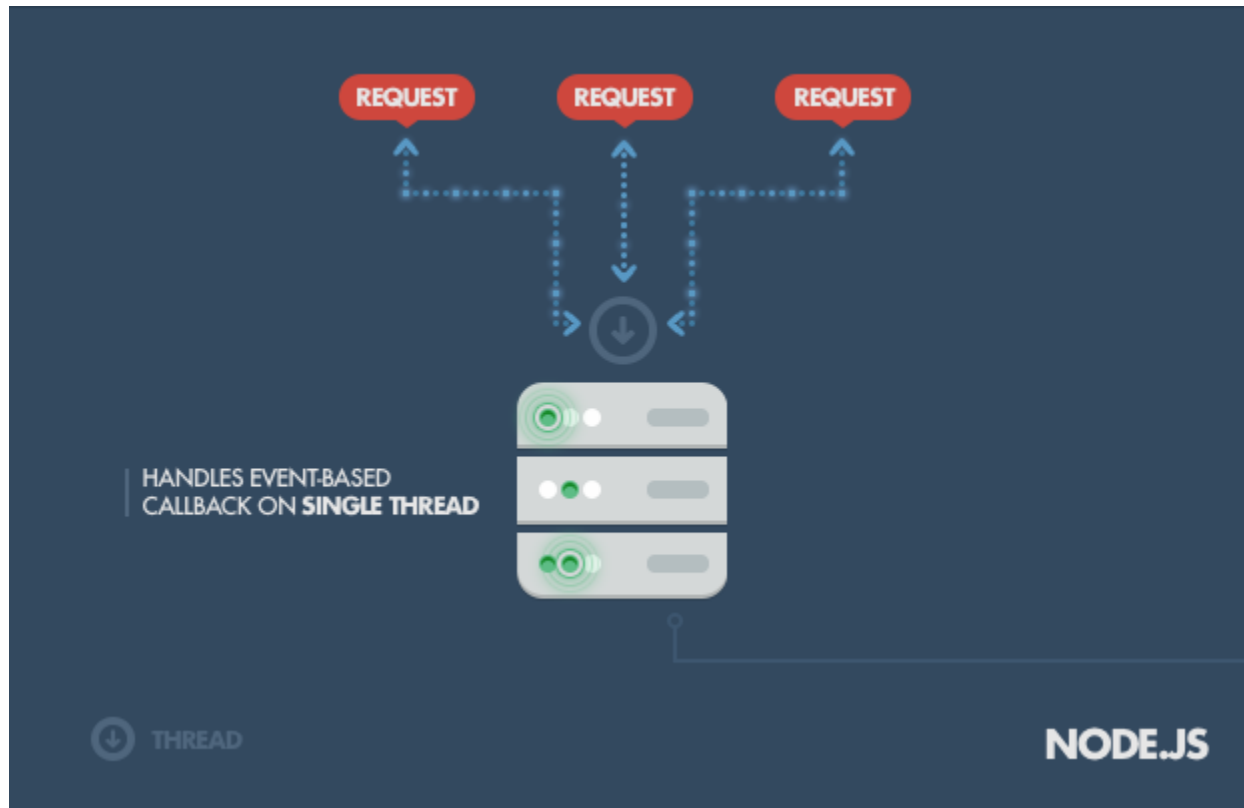
- Built on JavaScript runtime (v8) 
 - Written in C++
 - Developed by Google for Chrome
 - Compiles JS to native machine code before executing it
 - instead of interpreting it and directly executing it from the FS
 - Runs extensive code optimizations
 - Extremely fast and efficient

Traditional Web Servers



Each connection (request) spawns a new thread, taking up system RAM

Node



single-thread, using non-blocking I/O calls, allowing it to support tens of thousands of concurrent connections

What It Means

- Node can keep many connections alive transparently
 - Without having to reject new incoming connections
 - Handling a huge number of simultaneous connections
 - High Throughput → High Scalability
- **Example:** System with 8 GB of RAM
 - With ~2MB/thread → limited to 4K concurrent connections
 - While Node can theoretically reach 1M (POC) *
 - Practically tens of thousands as opposed to thousands on traditional

** This of course does not necessarily mean low response time.*

Non Blocking I/O

- Node's approach: all I/O activities should be non-blocking (async)
 - Because I/O is the most expensive operation (remember L1, L2, RAM, FS, Network?)
- This means that all these:
 - HTTP requests, DB queries, file I/O, other things that require waiting...
- Run independently and do not halt execution

Non Blocking I/O – Cont.

- The runtime handles their operation (transparently)
- And when data is ready an event is placed on the event loop
- Which, when processed, calls your app's callback/closure

The Event Loop

- A fundamental part of Node (and JS)
 - JS needed that to do onclick, onmouseover etc.
- The system JS uses to deal with incoming requests (events)
- A similar concept to context-switching
 - There is only one thing happening at once
- Runs on [LibUv](#)

The Event Loop – Cont.

- I/O calls are the points at which Node.js can switch from one request to another
- At an I/O call, your code saves the callback and returns control to the node.js runtime environment
- The callback will be called later when the data actually is available

The Single Thread Thing

- Node is single threaded (not 100% true, next slide)
 - All clients (in your app) share that thread
- Heavy computation can choke it up
 - Cause problems for all clients
 - Incoming requests blocked while computing
- And also... Exceptions
 - An unhandled exception on one client
 - Bubbling up to the topmost Node.js event loop
 - Will terminate Node.js instance
 - Effectively crashing the program

Seriously? One Thread?

- No. Actually there are more threads
 - But they are transparent
- Your app runs on single thread
- I/O operations on other threads
 - A pool of I/O worker threads receive I/O interrupts and put corresponding events into the queue to be processed by the main thread

Library	
<u>LibUv</u>	Maintains and manages the event loop and the I/O events
<u>LibEio</u>	A library performing input/output asynchronously

Globals *

** Available in all modules*

Name	Usage / Functionality
Global/GLOBAL	GLOBAL._ = require('underscore');
process	process.argv.forEach(function(val, index, array) { ... } Process.abort(); process.on('exit', function(code) { ... } process.on('uncaughtException', function(err) { ... }
console	console.log(...), console.error(...) console.time('task'), console.timeEnd('task')
buffer (class)	Useful for handling octet streams (i.e. TCP apps)
require()	To require (load and use) modules
__filename	console.log(__filename); // absolute path & filename of code being executed – e.g. /opt/myapps/mymodule.js
__dirname	Directory name of currently executing script
module exports	A reference to current module. Used for module.exports A reference to the module.exports that is shorter to type
set/clearTimeout	Run a callback after at least X ms
set/clearInterval,	Repeatedly run a callback every ~X ms

Core Utilities (some)

Name	Usage / Functionality
net	Asynchronous network wrapper for creating servers & clients. Methods: net.createServer(), net.connect(), ... Classes: net.Server, net.Socket
http	Methods: http.createServer(), http.createClient(), ... Classes: http.Server, http.ServerResponse, ...
path	Utilities for handling and transforming file paths path.join('/code', by, 'Z', 'nodejs'); <i>// 'code/by/Z/nodejs'</i>
url	Utilities for URL resolution and parsing url.parse() <i>// returns object with parsed fields (protocol, hostname, path, query, ...)</i> url.format() <i>// take a URL object, return formatted URL string</i>
util	Methods: inherits, format (printf), isArray, isDate, print (synchronous), error, log, inspect (object to string)
crypto	Cryptographic utilities (for security / credentials handling etc.)
fs	File I/O operation wrappers. Methods: read, chown, chmod, watchFile, ...

Basic File Handling [fs]

// the async way

```
var fs = require("fs");  
fs.readFile("foo.txt", "utf8", function(error, data) {  
  console.log(data);  
});
```

// the sync way (bit of a rarity in Node's event driven world)

```
var fs = require("fs");  
var data = fs.readFileSync("foo.txt", "utf8");  
console.log(data);
```


Accessing MySQL [mysql]

```
var mysql = require('mysql');  
var connection = mysql.createConnection({  
  host    : 'localhost',  
  user    : 'me',  
  password : 'super^secret'  
});
```

// note that every method you invoke on a connection is queued and executed in sequence

```
connection.connect(function(err) {  
  if (err) { console.error('error connecting: ' + err.stack); }  
});
```

```
connection.query('SELECT 1 + 1 AS solution', function(err, rows, fields) {  
  if (err) throw err;  
  console.log('The solution is: ', rows[0].solution);  
});
```

```
connection.end(); // terminate gracefully
```

Logging [winston]

```
/******  
* File: loggy_the_log.js  
*****/
```

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

```
var logger = new (winston.Logger)({  
  transports: [  
    new (winston.transports.Console)({ level: 'error' }),  
    new (winston.transports.File)({ filename: 'path/to/all-logs.log' })  
  ],  
  exceptionHandlers: [  
    new (winston.transports.File)({ filename: 'path/to/exceptions.log' })  
  ]  
});
```

```
module.exports = logger; // remember module.exports from few slides back?
```

Logging – Cont.

// console.log is for kids! we're using winston!

```
var loggy = require('./loggy_the_log');
```

// some critical application logging

```
loggy.log('debug', "127.0.0.1 - there's no place like home");
```

// this will go to the console as well

```
loggy.error("127.0.0.1 - there's no place like home");
```

Keeping Node Running

// nohup (no hang-up on SSH exit)

nohup node app.js &

// forever (but what happens after reboot?)

npm install forever -g

forever start app.js

forever stop app.js

// pm2

npm install pm2 -g

pm2 start app.js

pm2 stop app.js

// nodemon

npm install -g nodemon

nodemon app.js

nodemon vs. forever / pm2

- Generally speaking:
 - **nodemon** is for **development**
 - **pm2** and **forever** are for **production**
 - **Recommendation:** pm2 is better for prod

Surviving a Reboot

- With PM2 your life is a breeze

```
pm2 start /var/www/myapp.js
```

```
pm2 startup
```

- To make changes to startup later on do:

```
pm2 save
```

Node Package Manager

- Introduced in 2011
- Automatically installed with Node
- A set of publicly available reusable components, available through easy installation via an online repository, with version and dependency management
- Full list of packages <https://npmjs.org/>

Node Package Manager – Cont.

- Managed by npm, Inc.
- Company founded 2014 by npm's creator Isaac Z. Schlueter
- Run the npm registry as free service
- Build supporting tools for the community
- Moto: *“when everyone else is adding force, we work to reduce friction”*

Example Usage

- `npm install -g forever`
- `npm install -g nodemon`
- `npm init`
- `npm install express --save`

Not Just for Node

- Also used by
 - Browsers
 - Angular
 - Bower
 - Mobile
 - Cordova
 - Javascript

The NPM ecosystem is open for all to publish modules and list them on the NPM repository

Popular Modules

- **express** - web framework for node
- **socket.io** - websockets
- **Jade** - templating engine
- **redis** – redis client library
- **underscore** – utility library
- **forever** – run node continuously
- ... and more (~122K total packages)

Global vs Local Package Installs

	Locally	Globally
Package location	project subfolder node_modules	~/.npm/
Added to PATH env. variable	No	Yes
Need to use package in node application	Yes	No
Need to use package in node all my node applications	Yes	npm link <pkg>
Need to use package in command line (forever, nodemon, node-inspector,...)	No	Yes
Want to use it in shell	No	Yes

NPM Commands – Install & Update

Command	Details
npm install <pkg>	Install a package locally (./node_modules/)
npm install -g <pkg>	Install a package globally (~/.npm/)
Npm uninstall <pkg>	Uninstalling a package installed locally
Npm uninstall -g <pkg>	Uninstalling a package installed globally
Npm install <pkg> --save	Install local package and update package.json
Npm uninstall <pkg> --save	Uninstall local package and update package.json
npm outdated	Check if any package in package.json is old
npm update	Check for outdated packages and install newer versions (limited to versions defined in package.json)
npm update <pkg>	Check and update specific package only (limited to version defined in package.json)

NPM Commands – Cont.

Command	Details
npm init	Interactively create package.json
npm help (or just npm)	Shows list of commands
npm help <cmd>	Shows help on npm command
npm ls	List local packages
npm ls -g	List global packages
npm search <pkg>	Search the repository for a package
npm link <pkg>	Create symbolic link to global package
npm unlink <pkg>	Remove symbolic link to global package
npm publish <folder>	A folder containing a package.json file

package.json - Example

```
{
  "name": "hello-express",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "body-parser": "~1.10.1",
    "cookie-parser": "~1.3.3",
    "debug": "~2.1.1",
    "express": "~4.11.0",
    "jade": "~1.9.0",
    "morgan": "~1.5.1",
    "serve-favicon": "^2.2.0",
    "socket.io": "^1.2.1",
    "underscore": "^1.7.0"
  }
}
```

package.json – Fields

Property	Details
name <i>[required]</i>	Application name. Must be url safe. Preferably short (think using it in a <code>require('my-pkg')</code>)
version <i>[required]</i>	Format MAJOR.MINOR.PATCH (e.g. 2.5.14) Together with the name forms the app's identifier.
description	Helps people discover your package (listed in npm search)
keywords	Array of strings
homepage	Project homepage
bugs	Project's issue tracker url and/or email for reporting bugs { "url" : "...", "email" : "..." }
license	{ "license" : "BSD-3-Clause" } Licenses list here
author, contributors	One person or list of persons (name, email, url)
bin	Executables to add to the PATH variable when installing

package.json – Fields

Property	Details
main	Single entry point to the library <code>{ "name" : "some-library", "main" : "./lib/some-library.js" }</code>
private	If set to true then npm will refuse to publish it. Prevents accidental publication of private repositories.
scripts	Script commands that run the app at various lifecycle times (prestart, start, stop, test, restart, install, postinstall....) Full list here <ul style="list-style-type: none">• npm start: will run prestart, start, poststart• npm test: will run pretest, test, posttest
dependencies	Simple object mapping package names → version range

Dependencies - Versions

```
"dependencies" :  
{  
  "code" : "1.0.0 - 2.9999.9999",  
  "byzed" : ">=1.0.2 <2.1.2",  
  "baz" : ">1.0.2 <=2.3.4",  
  "boo" : "2.0.1", /* Exactly version 2.0.1 */  
  "bar" : "<1.0.0 || >=2.3.1 <2.4.5 || >=2.5.2 <3.0.0",  
  "quack" : "http://asdf.com/asdf.tar.gz",  
  "til" : "~1.2", /* Approximately equivalent to version */  
  "elf" : "~1.2.3",  
  "soup" : "*", /* Any version */  
  "zack" : "2.x", /* Any version starting with 2 */  
  "thr" : "^3.3.0", /* Compatible with version */  
  "lat" : "latest",  
  "dyl" : "file:../dyl", /* Local path */  
  "doink" : "git://github.com/jkupzitz/doink.git"  
}
```

Dependencies – Versions Cont.

- “**Approximately equivalent to version**” (~)
 - Update to most recent **minor** (middle) version
 - ~1.2.3 will match all 1.2.x versions but not 1.3.0
- “**Compatible with version**” (^)
 - Update to most recent **major** (first) version
 - ^1.2.3 will match any 1.x.x including 1.3.0
 - But will hold off on 2.0.0

Versioning Rules

Increment sversion...	When you are ...
MAJOR	Making backward incompatible changes
MINOR	Adding backward compatible functionality
PATCH	Making backward compatible bug fixes

Core Modules

- Node has several modules compiled into the binary
- Nevertheless, *require* is still, well, required
 - `var zlib = require('zlib');`
 - `http = require('http');`
- Core modules always preferentially load
 - `require('http')` will always return the built in HTTP module, even if there is a file by that name

File Modules

- **Absolute path** (starts with /)
 - `require('/home/marco/foo.js')`
- **Relative path** (starts with ./)
 - `require('./circle')` // relative to the caller
- **No path** (core or node module)
 - `require('net')` // core modules
 - `require('code-by-z')` // node_modules folder

File Modules – Cont.

- If exact name not found will try
 - .js, .json, .node ([node addon](#))
 - .node is dynamically linked shared object. provides glue to C and C++ libraries
- Loading from node_modules folders
 - If not native module and not rel/abs path
 - Searches in node_modules folder
 - Up to system root (/node_modules/cbz.js)

Module Caching

- Modules are cached on first load
- By their resolved filename

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

```
// ... then later on ...
```

```
require('awesome'); // return same object (*notes)
```

Notes

- Same object not guaranteed
 - Depends on resolved filename (think absolute path, relative paths)
- If we want a module to execute code multiple times
 - export a function and call it
- If we want another instance - implement a factory method

module.exports

- Exports an object
- Assignment has to be immediate (i.e. in this tick, not in callback)

```
// mymodule.js
var EventEmitter = require('events').EventEmitter;
module.exports = new EventEmitter(); // module.exports object is created by the Module system

// Do some work, and after some time emit
// the 'ready' event from the module itself
setTimeout(function() {
    module.exports.emit('ready');
    console.log('I was required by %s', module.parent.filename); // filename, id, loaded, parent, children
}, 1000);
```

```
// app.js
var mm = require('./mymodule');
mm.on('ready', function() {
    console.log('mymodule a is ready');
});
```

module.exports – Cont.

- **Implicitly** behind the scenes we have something like that for each module

```
var module = { exports : {} };  
var exports = module.exports;
```

require.main

- When Node directly runs a file it sets
 - `require.main = module`
- So can check if we were executed directly
 - **`if (require.main === module)`**
- Also we can obtain the application's entry point from any module
 - `winston.log(require.main.filename);`

require.main – Cont.

```
// mymodule.js
```

```
var EventEmitter = require('events').EventEmitter;  
module.exports = new EventEmitter();
```

```
// Do some work, and after some time emit  
// the 'ready' event from the module itself
```

```
setTimeout(function() {  
    console.log('I was required by %s', module.parent.filename);  
    console.log('main entry point is %s', require.main.filename);  
    module.exports.emit('ready');  
}, 1000);
```



EXPRESS



What is a Web Framework (WAF)

- Software framework designed to support development of
 - dynamic websites
 - Web applications
 - Web services
 - Web resources (i.e. REST API)
- Makes web development easier
- Prevents re-writing common functionality

WAF Aspects

- URL Mapping / Routing
- MVC Architecture
- Template Engine
- Caching
- ORM
- Security
- Scaffolding / Generators
- Web Sockets

WAF Examples

- Django (Py)
- ROR (Ruby)
- Laravel (PHP)
- ASP.NET MVC (C#)
- Play! (Java/Scala)
- CakePHP (PHP)
- Yii (PHP)
- Zend Framework (PHP)
- Sinatra (Ruby)
- and of course... Express (JavaScript)

Express Web Framework

Aspect	
Template Engine	Jade, Mustache, Blade, ...
Routing	Built in
ORM	Waterline, Node ORM2, Sequelize
Security	csurf (CSRF), helmet (HTTP), ... Authentication: passport
Caching	express-view-cache
Controllers	Express.js middleware
Generators	express-generator, Yeoman
Web Sockets	socket.io
Package Manager	npm

Pros & Cons

- Node + Express can be used to create classic web applications
- However this request-response paradigm carrying around rendered HTML is not the typical Node.js use-case
- **Pros**
 - Non CPU bound apps can leverage JS top-to-bottom
 - Easy to setup for simple SPA's
- **Cons**
 - CPU bound apps will block Node.js so threaded platform is better
 - Using Node.js with RDBMS still quite a pain

Installing Express & Generator

- `npm install express --save`
- `npm install express-generator -g`
- `express myapp` // generate app skeleton
- `cd myapp` // our app's directory
- `npm install` // install dependencies
- Update port = 80 (in `/bin/www`)
- `nodemon / npm start`

Generated Directory Structure

```
+-- app.js
+-- bin
|   +-- www
+-- package.json
+-- public
|   +-- images
|   +-- javascripts
|   +-- stylesheets
|       +-- style.css
+-- routes
|   +-- index.js
|   +-- users.js
+-- views
    +-- error.jade
    +-- index.jade
    +-- layout.jade
```

Starting the Server

- npm start
 - or
- nodemon

Routes

// respond with "Hello World!" on the homepage

```
app.get('/', function (req, res) {  
  res.send('Hello World!');  
})
```

// accept DELETE request at /user

```
app.delete('/user', function (req, res) {  
  res.send('Got a DELETE request at /user');  
})
```

// run authentication middleware on all HTTP methods for route /api/

```
app.all('/api', function (req, res, next) {  
  console.log('Accessing protected section ...')  
  next() // pass control to the next handler  
})
```

Routes – Cont.

- Routes work top to bottom
 - First match runs (renders)

Middleware

- An Express application is essentially a series of middleware calls. It can:
 - Execute any code
 - Make changes to the request/response objects
 - End the request-response cycle
 - Call the next middleware in the stack
 - By calling `next()`;

Application Level Middleware

- bound to an instance of express, using `app.use()` and `app.VERB()`

```
// a middleware mounted on /user/:id
// will run for any HTTP method (GET/POST/...)
app.use('/user/:id', function (req, res, next) {
  console.log('Request Type:', req.method);
  next();
});
```

```
// route and its handler function (middleware system)
// which handles GET requests to /user/:id
app.get('/user/:id', function (req, res, next) {
  res.send('USER');
});
```

Router Level Middleware

- loaded using `router.use()` and `router.VERB()`
- Used for dividing to application control areas

```
var app = express();  
var router = express.Router();
```

```
// handler for /user/:id which renders a special page  
router.get('/user/:id', function (req, res, next) {  
  console.log(req.params.id);  
  res.render('special');  
});
```

Third Party Middleware

- Express is a routing and middleware web framework with minimal functionality of its own.
- Functionality to Express apps are added via third-party middleware.

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

```
// use the passport middleware  
// cross system middleware should be loaded before defining routes  
app.use(passport.initialize());  
app.use(passport.session());
```

Error Handling

- Typically used across the whole application, therefore it's best to implement it as a middleware.
- It has the same parameters plus one more, error:

```
app.use(function(err, req, res, next) {  
    // do logging and user-friendly error message display  
    res.send(500);  
})
```

Error Handling – Cont.

- Some error handling alternatives:

```
// redirect to the error page
```

```
app.use(function(err, req, res, next) {  
    res.redirect('/public/500.html');  
})
```

```
// or we can just send an HTTP 500 Bad Request
```

```
app.use(function(err, req, res, next) {  
    res.end(500);  
})
```

```
// or we can pass control to next error middleware/handler
```

```
app.use(function(err, req, res, next) {  
    // log the incident  
    // call next error middleware  
    next(err);  
})
```

Express cookie-parser

- Parse Cookie header and populate req.cookies with an object keyed by the cookie names

```
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(8080)
```

Express body-parser

- Contains key-value pairs of data submitted in the request body

```
var app = require('express')();
var bodyParser = require('body-parser');
var multer = require('multer');

app.use(bodyParser.json()); // for parsing application/json
app.use(bodyParser.urlencoded({ extended: true })); // for parsing
  application/x-www-form-urlencoded
app.use(multer()); // for parsing multipart/form-data

app.post('/', function (req, res) {
  console.log(req.body);
})
app.listen(8080);
```

MORE SOURCES

- [Diving into Node.js – Introduction and Installation](#)
- [Understanding NodeJS](#)
- [Node by Example](#)
- [Let's Make a Web App: NodePad](#)
- <http://www.nodebeginner.org/>