

---

# Table of Contents

Introduction	1.1
About the author	1.2
Setting up Node	1.3
Core API Basics	1.4
Buffers	1.4.1
Event Emitters	1.4.2
HTTP	1.4.3
Streams	1.4.4
Timers	1.4.5
Cryptography	1.4.6
Networking	1.4.7
ASYNC IO	1.4.8
Logging	1.5
Async IO	1.6
Event loop	1.7
Template Engines	1.8
Promises	1.9
NPM	1.10
Linting	1.11
Testing	1.12
Tools for unit testing	1.12.1
E2E testing	1.12.2
Performance	1.13
ECMAScript 2015	1.14
Maps and Sets	1.14.1

# Professional Node.JS development

## Introduction:

Node.js might be the most overwhelming single piece of software in the current JavaScript universe.

Node.JS is an open source project.

Node.JS is based on the JavaScript language and Node.JS applications are written with JavaScript language.

It is a cross platform environment and has runtime for windows, mac OS X, and Linux.

Node.JS philosophy is that its core should be kept as small as possible.

The original author of Node JS is Ryan Dahl who originally wrote Node JS in 2009.

He demonstrated his projects at the inaugural European [JSConf](#) on November 8, 2009. Dahl was inspired to create Node.js from Flickr progress bar widget.

He wanted to prevent first frame from flickering when animation starts and that point was his inspiration moment...

His demonstration at the JSConf combined with the new speed race Google's V8(JavaScript engine, an event loop and a low-level I/O API) were the trigger for the emerging technology.

The official documentation of Node.JS describes it as:

\*"Node.js is a platform built on Chrome's JavaScript run time for easily building fast and 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 provides an event-driven architecture and a non-blocking I/O API designed to optimize an application's throughput and it scales well for real-time Web applications. (More on that later)

### Why Node.js?

Node.js goal is to provide an easy way to build scalable network programs with asynchronous processing model.

The main benefits are:

1. Single thread Asynchronous I/O framework

2. Core in c++ on top of V8.Implemented on top of the v8 js engine (chromium and chrome engine)
3. Rest of it is written in javascript
4. Files io and db are non blocking
5. Very good for all the network related stuff
6. It can handle thousands of concurrent connections with minimal overhead(CPU/Memory) on a single process
7. Has a very small surface area backed by an enormous modules and extensions available from the large community supporting it.
8. Amazing performance even handling a big number of users

### **What makes Node.JS so fast?**

Node.JS contains the new V8 engine.

The JS VM v8 engine of chrome is a super-fast with smart performance optimizations VM that was designed & created by the Danish Lars Bak (a virtual machine master/ [genius](#)).

The V8 engine reference the [libuv](#) library that is based on asynchronous I/O. It was mainly developed for use by Node.js

The initial release supported only Linux.

Later on the NPM package manager was integrated with the release of NODE.JS.

It helped the community to publish and share open source Node.JS libraries that brought more functionality into the ecosystem.

Image of Ryan Dahl - NODE.JS creator



Ryan Dahl video talk [demonstration](#) at JSConf Nov 2009.

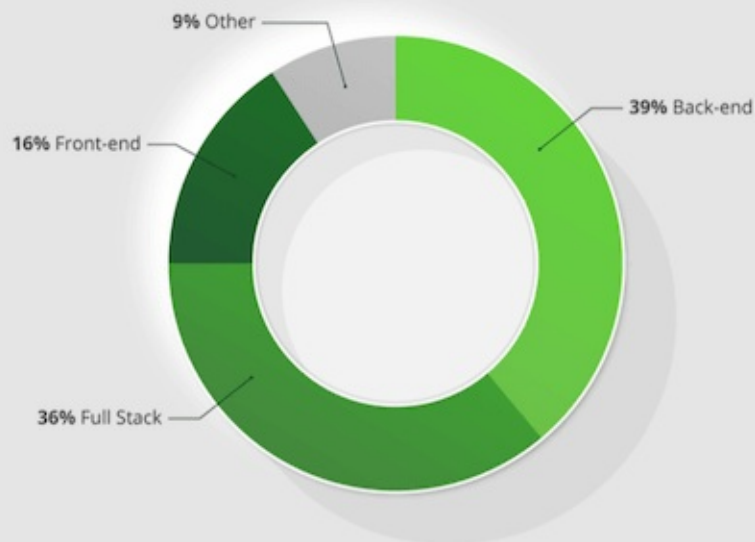
## For What exactly Node.JS is used these days?

Developers mainly use Node.js on the back-end, but it is popular as a full-stack and front-end solution as well.

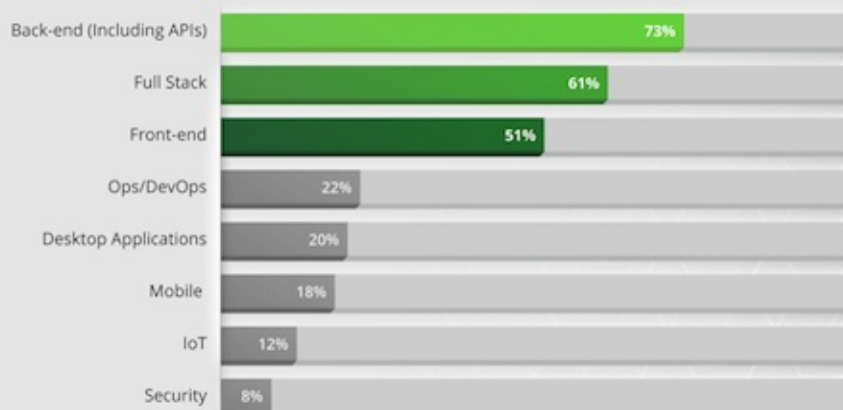
## USAGE APPLICATION

## How organizations use Node.js

### DEVELOPMENT PRIMARY FOCUS



### DEVELOPMENT NET FOCUS



This is no surprise since node.js strengths are that it is a full stack language which enable you to write projects both for the frontend and for the backend.

# **Which kind of applications are built today with node.js?**

according to the node js foundation survey:

## USAGE APPLICATION

## Node.js usage spans development type

TYPES OF DEVELOPMENT WORK SPEND TIME ON WITH NODE.JS *top mentions*

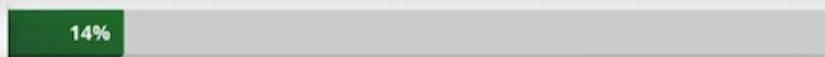
## Web Apps



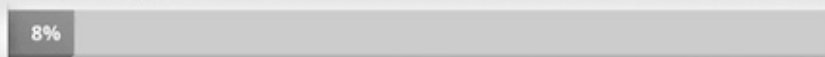
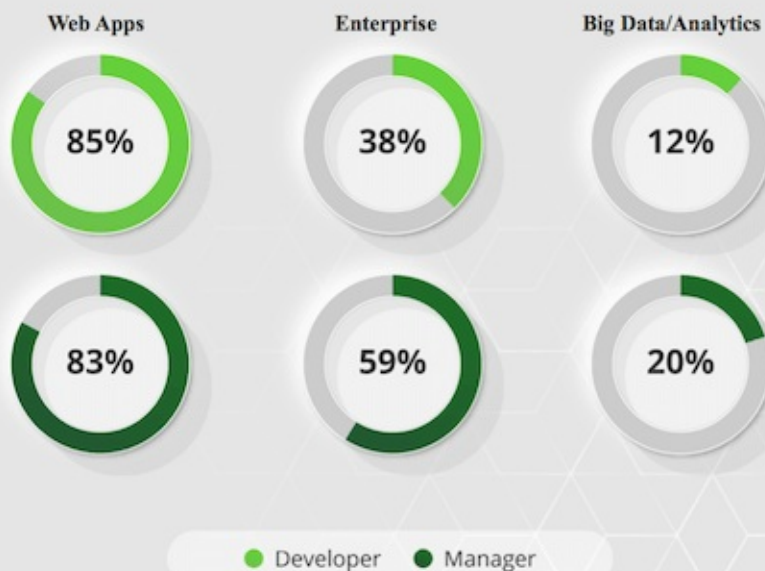
## Enterprise



## Big Data/Analytics



## Embedded System

DEVELOPMENT WORK BY TITLE *top mentions*





## About the author

[Tal Avissar](#) is a writer, speaker, open-web evangelist whose passionate about all things like JavaScript frameworks, Node.JS Expert & React.JS, Micro-services architecture, Kafka, Scala enthusiast, and open sorcerer.

You can contact me at talaviss at gmail without the space between names.

Currently working in the Tel Aviv silicon roundabout area doing mostly backend development.

To my wife and children: Carmella Eliya and Ori Avissar.

To my wonderful and devoted wife, Carmella, for her unending support and encouragement; without you this would not have been possible; and to my loving parents for always believing in me.

Node.js is an open source JavaScript runtime environment for easily building server-side and networking applications.

The platform runs on

- Linux, OS X,
- FreeBSD,
- MacOS
- and Windows.

### Install with Setup file

In order to install node you need , you can download a setup executable from [the official site](#)

Now it's the time to test Node.

To see if Node is installed, open the Windows Command Prompt, Powershell or a similar command line tool, and type `node -v` .

### Install with NVM

You can also install the node version by using nvm.

nvm is the node version manager that can install multiple version of the node runtime.

To install or update nvm, you can use the [install script](#) using cURL:

```
curl -o- https://raw.githubusercontent.com/creationix
vm/v0.33.2/install.sh |bash
```

or you can use Wget

```
wget -qO- https://raw.githubusercontent.com/creationix
vm/v0.33.2/install.sh |bash
```

## Test it!

Make sure you have Node and NPM installed by running simple commands to see what version of each is installed and to run a simple test program:

- **Test Node.** To see if Node is installed, open the Windows Command Prompt, Powershell or a similar command line tool, and type `node -v` . This should print a version number, so you'll see something like this `v0.10.35` .
- **Test NPM.** To see if NPM is installed, type `npm -v` in Terminal. This should print NPM's version number so you'll see something like this `1.4.28` .
- **Create a test file and run it.** A simple way to test that node.js works is to create a

JavaScript file: name it `hello.js` , and just add the code `console.log('Node is installed!');` . To run the code simply open your command line program, navigate to the folder where you save the file and type `node hello.js` . This will start Node and run the code in the `hello.js` file. You should see the output `Node is installed!`

as explained also in order to verify the installation that nvm installed correctly issue the following command:

```
command -v nvm
```

which should output 'nvm' if the installation was successful. Please note that `which nvm` will not work, since `nvm` is a sourced shell function, not an executable binary.

Node.js applications can be run at the command line, but you can also on run them as a service,

so that they will automatically restart on reboot or failure, and can safely be used in a production environment.

# Core API Basics

# Buffers

In the past the JavaScript language had no mechanism for reading or manipulating streams of binary data. The Buffer class was introduced as part of the Node.JS language API, because JavaScript does not handle straight binary data very well.

Since Node.JS is a server technology and has to deal with streams (TCP/UDP) and reading and writing to files and network.

So the buffer was the solution to the old way of working with encoded strings.

The buffer in Node.JS is a way of handling raw binary data. It is done with Raw memory allocation outside the V8 heap inside the Global namespace object. The buffer object is global.

An example of how to allocate simple buffer:

```
var buff = new Buffer('some arbitrary string');
```

The buffer can be created with different string encodings: 'ascii', 'utf8', 'ucs2', 'base64', 'binary', 'hex'. (later on explanation on each one of these).

You can also init buffer from an array:

```
var buffer = new Buffer([2,3,4]);
```

Once allocated the buffer cannot be resized.

A buffer can be initialized to certain size like:

```
var buffer = new Buffer(4096);
```

A buffer can be sliced to smaller buffer as follows:

```
var buffer = new Buffer("this is my first buffer");
```

```
var partSliced = buffer.slice(12,5);
```

# Event Emitters

In node.js an event can be described simply as a string with a corresponding callback. An event can be "emitted" (or in other words, the corresponding callback be called) multiple times or you can choose to only listen for the first time it is emitted.

Event emitter as it sounds is just something that triggers an event to which anyone can listen.

Different libraries offer different implementations and for different purposes, but the basic idea is to provide a framework for issuing events and subscribing to them.

Here is an Example from jQuery:

```
// Subscribe to event.

$('#foo').bind('click',function()
{
    alert("Click!");
});
// Emit event.

$('#foo').trigger('click');
```

Here is an example of code snippet that explains how events are emitted in node:

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

example_emitter.on("test", function () { console.log("test"); });

example_emitter.on("print", function (message) { console.log(message); });

example_emitter.emit("test");

example_emitter.emit("print", "message");

example_emitter.emit("unhandled");
```

And here an example of doing it from the REPL:

```
> var example_emitter = new (require('events').EventEmitter);

{}

> example_emitter.on("test", function () { console.log("test"); });

{ _events: { test: [Function] } }

> example_emitter.on("print", function (message) { console.log(message); });

{ _events: { test: [Function], print: [Function] } }
```

```
> example_emitter.emit("test");
```

```
test //console.log'd
```

```
true //return value
```

```
> example_emitter.emit("print", "message");
```

```
message //console.log'd
```

```
true //return value
```

```
> example_emitter.emit("unhandled");
```

```
false //return value
```

# HTTP

The purpose of this guide is to give you understanding of how the core http API with Node works.

Express and hapi frameworks use underneath the builtin http module.

With this module you can create an http server and much more.

This is the way you create the basic http server code:

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

var server = http.createServer(function(request, response) {
  // http server implementation
});``

The callback function that that is passed to the httpserver is called ebery time there
is a request from the client.

Http headers can be manipulates and retrieved from headers object:
```

```
var headers = request.headers;
var userAgent = headers['user-agent'];``
```

The http properties can be also retrieved as follows:

```
var url = request.url;
var httpmethod = request.method;``

1. In order to perform a simple get with the http module :
```

```
var http = require('http');
var options = {
  host: 'www.stackoverflow.com',
  path: '/index.html'
};

var req = http.get(options, function(res) {
  console.log('STATUS: ' + res.statusCode);
  console.log('HEADERS: ' + JSON.stringify(res.headers));
```



```
// Buffer the body entirely for processing as a whole.
var bodyChunks = [];
res.on('data', function(chunk) {
  // You can process streamed parts here...
  bodyChunks.push(chunk);
}).on('end', function() {
  var body = Buffer.concat(bodyChunks);
  console.log('BODY: ' + body);
  // ...and/or process the entire body here.
})
});
//because it implements event emitter it if on error is defined
req.on('error', function(e) {
  console.log('ERROR: ' + e.message);
});``
```

This get method calls the **req.end()** automatically

Another way is to use the general `http.request(options, callback)` function which allows you to specify the request method and other request details.

You need to specify the different attributes for the options object.  
options can be an object or a string.

```
...
var postData = querystring.stringify({
  'msg' : 'some arbitrary token'
});

var options = {
  hostname: 'www.facebook.com',
  port: 80,
  path: '/token',
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded',
    'Content-Length': postData.length
  }
};

var req = http.request(options, (res) => {
  console.log( STATUS: ${res.statusCode} );
  console.log( HEADERS: ${JSON.stringify(res.headers)} );
});
```

```
res.setEncoding('utf8');
res.on('data', (chunk) => {
  console.log( BODY: ${chunk} );
});
res.on('end', () => {
  console.log('No more data in response.')
})
});

req.on('error', (e) => {
  console.log( problem with request: ${e.message} );
});

// write data to request body
req.write(postData);
req.end();``
```

NodeJS supports `http.request` as a standard module:

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

var options = {
  host: 'example.com',
  port: 80,
  path: '/foo.html'
};

http.get(options, function(res){
  res.on('data', function(chunk){
    //do something with chunk
  });
}).on("error", function(e){
  console.log("Got error: " + e.message);
});
```

Of course we can also combine node-http-proxy and express. node-http-proxy will support a proxy inside node.js web server via RoutingProxy (see the example called Proxy requests within another http server).

# Streams

Node.js is asynchronous and event driven in nature. As a result, it's very good at handling I/O bound tasks. If you are working on an app that performs I/O operations, you can take advantage of the streams available in Node.js. So, let's explore Streams in detail and understand how they can simplify I/O.

What are exacty streams?

Streams are collections of data—just like arrays or strings. The difference is that streams might not be available all at once, and they don't have to fit in memory. This makes streams powerful when working with large amounts of data, or data that's coming from an external source one chunk at a time.

There are couple of operations that can be performed with streams:

There are couple of types of streams: Readable, writable, duplex.

## Readable Streams

1. Reading from streams
2. Setting Encoding
3. **Piping**
4. **Chaining**

Assume that you have an archive and want to decompress it. There are a many ways to achieve this. But the easiest and cleanest way is to use piping and chaining. Have a look at the following snippet:

```
var fs = require('fs');  
  
var zlib = require('zlib');  
  
fs.createReadStream('input.txt.gz')  
  .pipe(zlib.createGunzip())  
  .pipe(fs.createWriteStream('output.txt'));
```

## Writable Streams



# Timers

## 1. `setInterval(callback, delay, [arg], [...])`

```
setInterval(function(){
  console.log('test');
}, 60 * 60 * 1000);
```

2. To schedule the repeated execution of `callback` every `delay` milliseconds. Returns a `intervalId`

for possible use with `clearInterval()`

Optionally you can also pass arguments to the callback. **`setTimeout(callback, delay, [arg], [...])`**

To schedule execution of a one-time `callback` after `delay` milliseconds.

This function Returns a `timeoutId` for possible use with `clearTimeout()` `callback`. It is important to note that your callback will probably not be called in exactly `delay` milliseconds - Node.js makes no guarantees about the exact timing of when the callback will fire, nor of the ordering things will fire in. The callback will be called as close as possible to the time specified.

## 3. `clearImmediate(immediateObject)`

this method Prevents a timeout from triggering. it basically clear the `immediateObject` reference passed to it.

# Using The timer module:

The `timer` module exposes a global API for scheduling functions to be called at some future period of time. Because the timer functions are globals, there is no need to call

```
require('timers')
```

to use the API.

A timer in Node.js is an internal construct that calls a given function after a certain period of time. When a timer's function is called varies depending on which method was used to create the timer and what other work the Node.js event loop is doing.



# Cryptography

The crypto module is mostly useful as a tool for implementing cryptographic protocols such as TLS and https.

Nodejs offers great support for cryptography.

The crypto module is a wrapper above the OpenSSL cryptographic functions. (HMAC, Cyphers, ...)

```
require("crypto")
  .createHash("md5")
  .update("This is some crypt string in node!")
  .digest("hex");
```

The `crypto` module provides cryptographic functionality that includes a set of wrappers for OpenSSL's hash, HMAC, cipher, decipher, sign and verify functions.

How to determine if crypto support is unavailable

```
let crypto;

try {

  crypto = require('crypto');

} catch (err) {

  console.log('crypto support is disabled!');

}
```

The `crypto` module provides the `Certificate` class for working with SPKAC data.

Instances of the `Certificate` class can be created using the `new` keyword or by calling `crypto.Certificate()` as a function:

Example on how to decrypt/encrypt **text** using the crypto module



```
var crypto = require('crypto'),
    algorithm = 'aes-256-ctr',
    password = 'd6F3Efeq';

function encrypt(text){
  var cipher = crypto.createCipher(algorithm,password)
  var crypted = cipher.update(text,'utf8','hex')
  crypted += cipher.final('hex');
  return crypted;
}

function decrypt(text){
  var decipher = crypto.createDecipher(algorithm,password)
  var dec = decipher.update(text,'hex','utf8')
  dec += decipher.final('utf8');
  return dec;
}

var hw = encrypt("hello world")

console.log(decrypt(hw));
```

An example on how to decrypt/encrypt **buffers**:

```
var crypto = require('crypto'),
    algorithm = 'aes-256-ctr',
    password = 'd6F3Efeq';

function encrypt(buffer){
  var cipher = crypto.createCipher(algorithm,password)
  var crypted = Buffer.concat([cipher.update(buffer),cipher.final()]);
  return crypted;
}

function decrypt(buffer){
  var decipher = crypto.createDecipher(algorithm,password)
  var dec = Buffer.concat([decipher.update(buffer) , decipher.final()]);
  return dec;
}

var hw = encrypt(new Buffer("hello world", "utf8"))
// outputs hello world
console.log(decrypt(hw).toString('utf8'));
```

# Networking examples with node.js

## A simple Telnet Chat

```
var sockets = [];  
var nicks = 1;  
  
var s = net.Server(function(socket) {  
    sockets.push(socket);  
    socket.nickname = nicks++;  
    socket.write('Welcome to telnet-chat!\n');  
  
    socket.on('data', function(d) {  
        for (var i=0; i  
<  
sockets.length; i++) {  
            sockets[i].write(socket.nickname+":\t "+d);  
        }  
    });  
    socket.on('end', function() {  
        var i = sockets.indexOf(socket);  
        sockets.splice(i, 1);  
    });  
}).listen(8000);
```

## Working with TCP

### A simple tcp based chat server

```
// Load the TCP Library
```

```
net = require('net');

// Keep track of the chat clients
var clients = [];

// Start a TCP Server
net.createServer(function (socket) {

  // Identify this client
  socket.name = socket.remoteAddress + ":" + socket.remotePort

  // Put this new client in the list
  clients.push(socket);

  // Send a nice welcome message and announce
  socket.write("Welcome " + socket.name + "\n");
  broadcast(socket.name + " joined the chat\n", socket);

  // Handle incoming messages from clients.
  socket.on('data', function (data) {
    broadcast(socket.name + "
>
" + data, socket);
  });

  // Remove the client from the list when it leaves
  socket.on('end', function () {
    clients.splice(clients.indexOf(socket), 1);
    broadcast(socket.name + " left the chat.\n");
  });

  // Send a message to all clients
  function broadcast(message, sender) {
    clients.forEach(function (client) {
      // Don't want to send it to sender
      if (client === sender) return;
      client.write(message);
    });
    // Log it to the server output too
    process.stdout.write(message)
  }

}).listen(5000);

// Put a friendly message on the terminal of the server.
console.log("Chat server running at port 5000\n");
```

# Asynchronous I/O

Asynchronous I/O, or non-blocking I/O, is a form of input/output processing that permits other processing to continue before the transmission has finished.

What this means is, if a process wants to do a `read()` or `write()`, in a synchronous call, the process would have to wait until the hardware finishes the physical I/O so that it can be informed of the success/failure of the I/O operation.

On asynchronous mode, once the process issues a read/write I/O asynchronously, the system calls is returned immediately once the I/O has been passed down to the hardware or queued in the OS/VM. Thus the execution of the process isn't blocked (hence why it's called non-blocking I/O) since it doesn't need to wait for the result from the system call, it will receive the result later.

[Asynchronous I/O](#) (from Wikipedia)

## Overview of Blocking vs Non-Blocking

This overview covers the difference between **blocking** and **non-blocking** calls in Node.js.

This overview will refer to the event loop and `libuv`

`libuv` is a multi-platform support library with a focus on asynchronous I/O. It was primarily developed for use by Node.js, but it's also used by Luvit, Julia, pyuv, and others. In case you find errors in this documentation you can help by sending pull requests!

Readers are assumed to have a basic understanding of the JavaScript language and Node.js callback pattern.

"I/O" refers primarily to interaction with the system's disk and network supported by `libuv`.

## Blocking

**Blocking** is when the execution of additional JavaScript in the Node.js process must wait until a non-JavaScript operation completes. This happens because the event loop is unable to continue running JavaScript while a **blocking** operation is occurring.

In Node.js, JavaScript that exhibits poor performance due to being CPU intensive rather than waiting on a non-JavaScript operation, such as I/O, isn't typically referred to as **blocking**. Synchronous methods in the Node.js standard library that use `libuv` are the most

commonly used **blocking** operations. Native modules may also have **blocking** methods.

All of the I/O methods in the Node.js standard library provide asynchronous versions, which are **non-blocking**, and accept callback functions. Some methods also have **blocking** counterparts, which have names that end with `Sync` .

# Logging with node

There are many libraries available for logging in the node ecosystem. The main ones in use are log4js and the winston libraries

## Using Log4JS library

The npm module for this library appears [here](#)

```
//requiring the log4js library
var log4js = require('log4js');

//console log is loaded by default, so you won't normally need to do this
//log4js.loadAppender('console');

log4js.loadAppender('file');

//log4js.addAppender(log4js.appenders.console());

log4js.addAppender(log4js.appenders.file('logs/cheese.log'), 'cheese');

var logger = log4js.getLogger('cheese'); //get a reference to a named instance

//setting the level of information trace, debug, info, warn, error or fatal
logger.setLevel('ERROR');
```

The library is used heavily in the industry.

It is needed to configure the log4js with the correct appenders and other metadata for the logging framework:

```
configure('./src/config/log4js-config.json');

const logger = getLogger("app");
```

Here is an example for log configuration

```
{
  "appenders": [
    {
      "type": "console",
      "layout": {
        "type": "pattern",
        "pattern": "[%d] [%p] %c {%x{ln}} - %m",
        "tokens": {
          "ln" : "loggerFunction()"
        }
      }
    },
    {
      "type": "dateFile",
      "filename": "log/access.log",
      "pattern": "-yyyy-MM-dd",
      "alwaysIncludePattern": false,
      "category": "http",
      "layout": {
        "type" : "pattern",
        "pattern": "[%d] [%p] %c {%x{ln}} - %m",
        "tokens": {
          "ln" : "loggerFunction()"
        }
      }
    }
  ],
}
```

There are builtin appenders that can be configured for the log4js which can be found [here](#).

Out of the box it supports the following features:

1. colored console logging
2. replacement of node's console.log functions (optional)
3. file appender, with log rolling based on file size
4. SMTP appender
5. GELF appender
6. hook.io appender
7. Loggly appender
8. Logstash UDP appender
9. multiprocess appender (useful when you've got worker processes)
10. logger for connect/express servers

11. configurable log message layout/patterns
12. different log levels for different log categories (make some parts of your app log as DEBUG, others only ERRORS, etc.)

## Using Winston library

A multi-transport async logging library for node.js.

Winston is one of the most popular Node.js logging frameworks

First you need to require the module:

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

The default logger is accessible through the winston module directly.

Usage example:

```
``
```

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

```
winston.log('info', 'Node. JS logging');
```

```
winston.info('this is some important info');
```

```
winston.level = 'debug';
```

```
winston.log('debug', 'debug messages are logged now');``
```



# Event loop

Event loop is a construction that is responsible for dispatching events in a program that almost always operates asynchronously with the message originator. When you call an I/O operation, NodeJS stores the callback assigned with that operation and continue processing other events. Callback will be triggered when all needed data is collected.

Here is more advanced definition of the event loop:

The event loop, message dispatcher, message loop, message pump, or run loop is a programming construct that waits for and dispatches events or messages in a program. It works by making a request to some internal or external “event provider” (which generally blocks the request until an event has arrived), and then it calls the relevant event handler(“dispatches the event”). The event-loop may be used in conjunction with a reactor, if the event provider follows the file interface, which can be selected or ‘polled’ (the Unix system call, not actual polling). The event loop almost always operates asynchronously with the message originator.

Here is a simple illustration that explains how event loop works in NodeJS.



## NodeJS Event Loop

When a request is received by web-server it goes to the event loop. Event loop registers operation in a thread pool with assigned callback. Callback will be triggered when processing request is done. Your callback also can do other intensive operations like

querying the database, but it does so the same way—registers operation in a thread pool with assigned callback and so on...

But what about code execution and its speed? Next, we are going to talk about virtual machine that executes JavaScript code—V8.

If you want to know more about how the v8 does with that loop you can read much more details about it [in the following article](#)

The V8 and Lars Bak, the lead developer of V8:

# Template engines:

A template engine enables you to use static template files in your application. At runtime, the template engine replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client. This approach makes it easier to design an HTML page.

Some popular template engines that work with Express are [Pug](#), [Mustache](#), and [EJS](#). The [Express application generator](#) uses [Jade](#) as its default, but it also supports several others.

To render template files, set the following [application setting properties](#), set in `app.js` in the default app created by the generator:

- `views` , the directory where the template files are located. Eg: `app.set('views', './views')` . This defaults to the `views` directory in the application root directory.
- `view engine` , the template engine to use. For example, to use the Pug template engine: `app.set('view engine', 'pug')` .

Then install the corresponding template engine npm package; for example to install Pug:

```
$ npm install pug --save
```

After the view engine is set, you don't have to specify the engine or load the template engine module in your app; Express loads the module internally, as shown below (for the above example).

```
app.set('view engine', 'pug')
```

Create a Pug template file named `index.pug` in the `views` directory, with the following content:

```
html
  head
    title= title
  body
    h1= message
```

Then create a route to render the `index.pug` file. If the `view engine` property is not set, you must specify the extension of the `view` file. Otherwise, you can omit it.

```
app.get('/',function(req,res){
  res.render('index',{title:'Hey',message:'Hello there!'}})
)
```

When you make a request to the home page, the `index.pug` file will be rendered as HTML.

Note: The view engine cache does not cache the contents of the template's output, only the underlying template itself. The view is still re-rendered with every request even with the cache is on.

To learn more about how template engines work in Express, see: [“Developing template engines for Express”](#).

## Dust

Dust.js comes from LinkedIn — a fully asynchronous Javascript templating system/engine for the browser and server. Dust, while not completely logic-less, does involve a lot less logic than your average templating system. With Dust you're moving all your logical parts of the code towards a simple data model, at which point you're able to execute functions within that model and call it forth by using the template system itself, which then grants you full control over how your templates react in different situations.

## doT

doT.js is small, efficient, fast and lightweight templating engine that supports itself (no dependancies), and works great with Node.js and native Browser integration.

## Handlebars

Handlebars is a close successor to Mustache, and both can actually be used at the same time, with the ability to swap out tags where necessary. The only difference is that Handlebars is more focused on helping developers to create semantic templates, without having to involve all the confusion and time consumption. You can easily [try out Handlebars yourself](#) (there's also an option to try Mustache on the same page) and see for yourself whether this is the type of templating engine you're looking for.

## EJS

The last of the most popular JavaScript template engines on our list is going to be Embedded JavaScript Templates (EJS) — a lightweight solution towards creating HTML markup with simple JavaScript code. Worry not about organizing your stuff in the right manner, it's just straight JavaScript all the way. Fast code execution, ease of debugging makes this the perfect templating engine for those who want to do HTML work with their favorite language, presumably JavaScript.

## Underscore

Underscore, another highly reputable templating engine, is an external JavaScript library that enables developers to take advantage of functional helpers that keep the code base intact. Underscore solves the problem of having to open your code editor and not knowing where to start. Underscore provides over 100 functions that support both your favorite workaday functional helpers: map, filter, invoke — as well as more specialized goodies: function binding, javascript templating, creating quick indexes, deep equality testing, and so on.

## Pug

When people say Python is like writing in English, they underestimate the magnitude of that statement when it comes to Pug syntax programming. The Pug template engine (for Node.js) is literally enabling developers to write code that looks like paragraphs straight out of a book. Not only does this improve the overall code productivity, it can help to streamline the work on a project that consists of multiple team members.

## Choosing the Best Templating Engine for JavaScript

When choosing the right templating engine for our projects, we should take into consideration the exact type of work we are looking to do, and how much of the project is actually going to have to be templated, and what kind of solution would work out for you individually in both long-term and short-term.

# Promises

The first thing to know about promises is that they are an abstraction for asynchronous programming.

Promises are a very important pattern when developing in node.JS and have vast usage.

Promises provide a alternative way for writing asynchronous code with raw callbacks.

Promise are an abstraction and a pattern to solve calling asynchronous code in a more controllable manner.

Promises provide us with a cleaner and more robust way of handling async code.

It also reverts the [IOC](#) pattern and returns the control to the caller itself.

So this pattern behavior of promises help us to manage the async code as if it was sync code.

This chapter is an introduction to ECMAScript 6 Promise API in particular.

The default pattern of promise is:

```
``function readFileAsync() {  
  
  return new Promise(  
    function (resolve, reject) {  
  
      //...implementation goes here  
      resolve(value); // resolving the promise  
      reject(error); // rejecting the promise  
    });  
}
```

The above function can be used as follows:

```
readFileAsync()  
.then(value => { / success / })  
.catch(error => { / failure / });
```

Promises have different states:

1. Pending - The initial state of a promise.
2. Fulfilled - The state of a promise representing a successful operation.
3. Rejected - The state of a promise representing a failed operation.

Promises have a then method, which you can use to get the eventual return value (fulfillment) or thrown exception (rejection).

There are couple of 3rd party libraiaes module that are vastly used in the community:

## Bluebird:

[Bluebird](#) provides promisification on steroids.

Installation goes like:

```
npm install bluebird
```

When creating a promise:

```
new Promise(function (ok, err) { doSomething(function () { if (success) { ok(); } else { err(); } }); })
```

From consumer prespective

```
promise
  .then(okFn, errFn)
  .catch(errFn)``
```

When dealing with an array of promises:

```
var promises = [
  promiseDoSomething(), promiseDoOther(), ...
]

// succeeds when all the above promises succeed
Promise.all(promises)
  .then(function (results) {
  });

// succeeds when one of the promises finishes first
Promise.race(promises)
  .then(function (result) {
  });``
```

You can create from an object api a promisification of all its exposed library methods. For example:

So Now you can use the fs module as if it was designed to use by bluebird promises from the beginning =>

```
var fs = require("fs");
Promise.promisifyAll(fs);

fs.readFileAsync("file.js", "utf8").then(...)`
```

Once you've done the above code you just need to add the "Async"-suffix to method calls and start using the usual promise interface (instead of the callback interface).

Usually this should be done on the .prototype (calling the promisifyAll) when requiring the library's classes.

For example:

```
Promise.promisifyAll(require("mysql/lib/Connection").prototype);`
```

### **Bluebird promise monitoring:**

This is a very nice feature in bluebird which enables us to hook on the lifecycle events of promises in the bluebird library:

- "promiseCreated" - when created through the constructor.
- "promiseChained" - when created through chaining (e.g. .then).
- "promiseFulfilled" - when a promise is fulfilled.
- "promiseRejected" - when a promise is rejected.
- "promiseResolved" - when a promise adopts another's state.
- "promiseCancelled" - when a promise is cancelled.

In order to enable this promise you need to manually call `Promise.config` with `monitoring: true`.

### **canceling a promise:**

With the bluebird library it is also possible to cancel a library:



```
var Promise = require('bluebird');

var parentPromise = Promise.resolve(42)
  .then((val1) => {
    console.log(val1);
    return val1 + 1;
  })
  .then((val2) => {
    console.log(val2);
    return val2 + 1;
  })
  .cancellable()//from here =>now you can cancel the promise
  .then((val3) => {
    const randomValue = parseInt(Math.random() * 10)
    console.log('random value:' + randomValue);
    return randomValue;
  })
  .then((randomValue) => {
    console.log(randomValue);
    if (randomValue <= 5) {
      console.log('cancelling the promise');
      return parentPromise.cancel('value cannot be lower then five');
    }
    console.log('continuing the promise');
    return someRndValue + 1;
  })
  .then((val4) => {
    //this code will not be reached if someRndValue<5
    console.log(val4);
    return val4;
  })
  .catch(Promise.CancellationError, function (err) {
    //this code will be reached only if someone calls without reason : parentPromi
se.cancel();
    console.log('CancellationError:' + err);
  })
  .catch((err) => {

    console.log(err.message);
  });
```

## Node.js 8: `util.promisify()`

Node.js 8 has a new utility function: `util.promisify()` .

It converts a callback-based function to a Promise-based one. This contribution is mainly by [Benjamin](#)

## For example the following code :

If you hand the path of a file to the following script, it prints its contents.

```
const {promisify} = require('util');

const fs = require('fs');

const readFileAsync = promisify(fs.readFile); // (A)

const filePath = process.argv[2];

readFileAsync(filePath, {encoding: 'utf8'})

.then((text) => {

  console.log('CONTENT:', text);

})

.catch((err) => {

  console.log('ERROR:', err);

});
```

## The Q library

A tool for creating and composing asynchronous promises in JavaScript

The Q library created by kris kowal and is one of the main libraries.

To promisify a callback with the Q library:

```
Q.fcall(promisedStep1) .then(promisedStep2) .then(promisedStep3) .then(promisedStep4)
.then(function (value4) { // Do something with value4 }) .catch(function (error) { //
Handle any error from all above steps }) .done();
```

An important article on how we are missing the point about promises can be found [here](#).

# NPM

NPM is the largest ecosystem of open source libraries in the world.

NPM stands for Node.js' package manager.

Make sure you have Node and NPM installed by running simple commands to see what version of each is installed and to run a simple test program.

## install NPM on windows

1. Download the Windows installer from the [Node.js® web site.](<https://nodejs.org/en/>)
2. Run the installer (the .msi file you downloaded in the previous step.)
3. Follow the prompts in the installer (Accept the license agreement, click the NEXT button a bunch of times and accept the default installation settings).

installer

1. Restart your computer. You won't be able to run Node.js® until you restart your computer.

## Command line interface (aka CLI)

The npm command-line tool is bundled with Node.js.

npm also has a command line client that allows developers to install and publish your packages.

```
npm --version
```

```
2.14.12
```

The second important command is npm init:

```
$ npm init
package name: (project)
version: (1.0.0)
description: Demo of package.json
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
```

Press Enter to accept the defaults, then type yes to confirm. This will create a package.json file at the root of the project.

```
{
  "name": "project",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Tip: You can also issue the command for default initialization of all the defaults with

This way to generate a package.json file use `npm init --y`

NPM contains different APIs which will shown here:

You can use the following command in order to list down all the locally installed modules

```
npm ls
```

Search for packages in the repository

```
npm search mocha
```

You can install modules with NPM locally or globally.

When you install modules globally they are installed in a system directory.

## Where can you find all the packages?

<https://npmjs.org>

## installing packages with npm

NPM comes along with Node.js

to install a packages simply run:

- `npm install package-name`
- `npm install tarball file`
- `npm install tarball url`

If you want to install a package globally you can issue:

- `npm install -g package-name`

Note: global installation is against best practices. because when you will deploy your app with CI to different servers these globally tools will not be there and will cause dependencies issues that are ahrd to overcome. the preferred way is to install this dependencies as local or if they are tools then you can put them in the bin folder.

Global installation makes the package available globally irrespective to the directory you installed it from.

Is you want to install as development dependency issue (shortcuts)

- `npm i -D`

Is you want to install as a normal dependency issue (shortcuts)

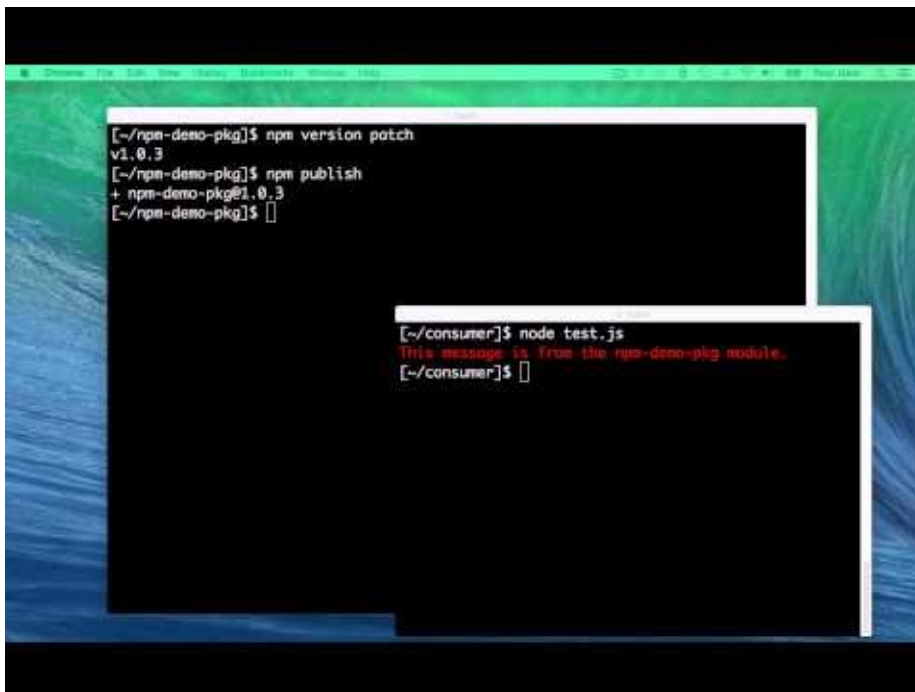
- `npm i -S`

## package.json file

Undersatnding the pacage.json:

- It is a valid JSON object
- name and version fields are required, the combination makes a unique identifier for the package
- There are Some used fields in package.json
  - description
  - keywords
  - homepage

- bugs
  - license
  - author & contributors
  - main
- the dependencies section is where you know which packages are inside this project build
  - The script section is where you define your script commands (start, preinstall)
  - The bin field is the folder where your binaries of this project exist
1. this file also serves as documentation for what packages your project depends on
  2. It allows you to specify the versions of a package for your project according to semantic versioning rules](<https://docs.npmjs.com/getting-started/semantic-versioning>). take a look in this video:



```
[~/npm-demo-pkg]$ npm version patch
v1.0.3
[~/npm-demo-pkg]$ npm publish
+ npm-demo-pkg@1.0.3
[~/npm-demo-pkg]$

[~/consumer]$ node test.js
This message is from the npm-demo-pkg module.
[~/consumer]$
```

[Video link](#)

# Linting

## Why linting?

Linting is the process of running a program that will analyse code for potential errors.

Linting will run through your source code to find

- formatting discrepancy
- non-adherence to coding standards and conventions
- pinpointing possible logical errors in your program

Linting will help you to catch bugs and to enforce your styling code guidelines in the project.

Linting is the process of checking the source code for Programmatic as well as Stylistic errors. This is most helpful in identifying some common and uncommon mistakes that are made during coding.

A `Lint` or a `Linter` is a program that supports linting (verifying code quality). They are available for most languages like JavaScript, CSS, HTML, Python, etc..

Some of the useful linters are [JSLint](#), [CSSLint](#), [JSHint](#), [Pylint](#)

There are couple of available/useful linters that can be used to perform linting with node.js

## Eslint:

While ESLint is designed to be run on the command line, it's possible to use ESLint programmatically through the Node.js API. The purpose of the Node.js API is to allow plugin and tool authors to use the ESLint functionality directly, without going through the command line interface.

### Installing ESLint

You can install ESLint globally by running the command

```
npm install -g eslint
```

Next you need to configure your eslint with running a setup that creates eslint configuration file running the following command:

**eslint --init**

```
testdemo eslint --init
How would you like to configure ESLint? Answer questions about your style
Are you using ECMAScript 6 features? Yes
Are you using ES6 modules? Yes
Where will your code run? Browser
Do you use CommonJS? Yes
Do you use JSX? Yes
Do you use React? Yes
What style of indentation do you use? Tabs
What quotes do you use for strings? Single
What line endings do you use? Unix
Do you require semicolons? Yes
What format do you want your config file to be in? JavaScript

Successfully created .eslintrc.js file in /Users/user/dev/testdemo
ESLint was installed locally. We recommend using this local copy instead of your
globally-installed copy.
testdemo
```

To be continued ...



# Unit Testing

In [computer programming](#), **unit testing** is a [software testing](#) method by which individual units of [source code](#), sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use.

Unit testing is, roughly speaking, testing bits of your code in isolation with test code.

The immediate advantages that come to mind are:

- Running the tests becomes automate-able and repeatable
- You get a safety net for your existing code
- You can easily refactor your code and cover the different unit functionality
- You can test at a much more granular level than point-and-click testing via a GUI

Note that if your test code writes to a file, opens a database connection or does something over the network, it's more appropriately categorized as an integration test. Integration tests are a good thing, but should not be confused with unit tests. Unit test code should be short, sweet and quick to execute.

Another way to look at unit testing is that you write the tests first. This is known as Test-Driven Development (TDD for short). TDD brings additional advantages:

- You don't write speculative "I might need this in the future" code -- just enough to make the tests pass
- The code you've written is always covered by tests
- By writing the test first, you're forced into thinking about how you want to call the code, which usually improves the design of the code in the long run.

# Tools for TDD

## Mocha

Using Mocha for use as a framework for test driven development of your Node.JS apps.

Mocha is a feature-rich JavaScript test framework running on Node.js and the browser, making asynchronous testing simple and fun.

Agile methods today are very common among software projects. TDD is one of the main agile development techniques.

Installing Mocha:

```
$ npm install -g mocha
```

## TDD

Test-driven development (TDD) is a software development process that based upon repetition of a very short development cycle.

The main idea behind TDD is to:

1. Add and define the test.
2. Implement the internal test logic.
3. Check and validate that the test either passes or fails
4. Write some code
5. Run tests
6. Refactor the code
7. Repeat

## Assersion libraries

1. [Assert](#) built in module in Node
2. [Expect.JS](#)
3. [Chai Assert](#)

```
expect({ foo: 'baz' }).to.have.property('foo') .and.not.equal('bar');
```



## End 2 End testing

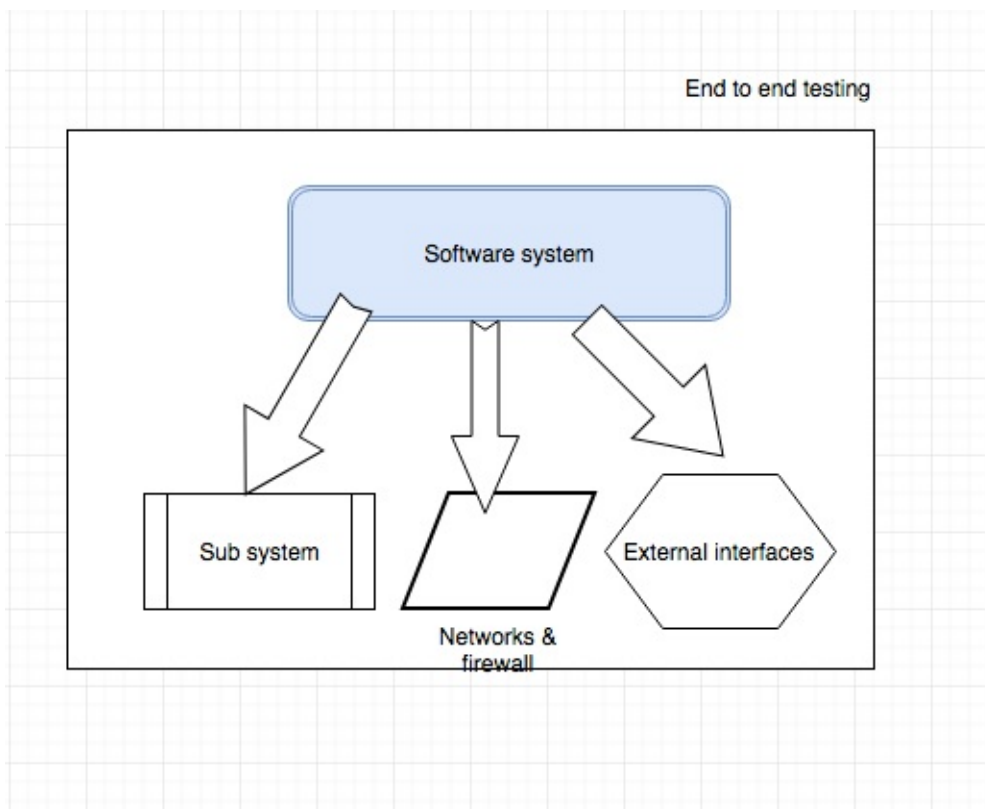
**End to end testing** is a methodology used to **test** whether the flow of an application is performing as designed from start to finish. The purpose of carrying out **end to end tests** is to identify system dependencies and to ensure that the right information is passed between various system components and systems.

End-to-end testing is a technique used to test whether the flow of an application right from start to finish is behaving as expected.

End to End Testing is usually executed after functional and system testing.

There are many frameworks that can help the developer to perform e2 testing.

When it comes to data testing we can use production like data for the e2e tests by copying real data (from real environment) in order to perform as close as possible tests to real life scenarios



## Why do we need End to End Testing ?

Since modern application are complex and tend to depend on many external integrations and dependencies we want to perform full scenario tests as the user interacts with the system. Sometimes we are also interacting with sub system or legacy external systems.

### The well known solutions for e2e tests are:

These tools helps you perform Browser automated testing done easy.

**Nightwatch.js** — 5975 stars [Site](#) [GitHub](#)

Nightwatch.js is an easy to use *Node.js \_based End-to-End (E2E) testing solution for browser based apps and websites. It uses the powerful [ W3C WebDriver API]* (<https://www.w3.org/TR/webdriver/>) to perform commands and assertions on DOM elements.

**CasperJS** — 6399 stars [Site](#) [GitHub](#)

**Protractor** — 6351 stars [Site](#) [GitHub](#)

**TestCafe** — 2106 stars [Site](#) [GitHub](#)

**CodeceptJS** — 1040 stars [Site](#) [GitHub](#)

## Node.JS and Performance

Under high load (high concurrency), Node.js maintains high throughput and low latency

Three factors underpin Node's high performance:

1. The V8 JavaScript engine, upon which Node.js is based, is highly optimized for performance by Google, who continues to invest heavily in advancing V8's performance
2. JavaScript/Node.js events are lightweight, while threads are heavyweight. Consequently, Node.js is inherently more performant under load than monolithic designs (this article offers a nice recap)
3. Node.js is container-ready, which simplifies the move to cloud and microservices architectures (Richard Rodger, author of The Tao of Microservices, explains)

Some Customers giants using node js Performance facts:

1. Groupon Node.js implementation reduced page load times by 50%



- 2.
3. PayPal Node.js App doubled the number of requests per second and reduced response time by 35% versus previous Java version



1. GoDaddy rolled out global site rebrand in [1 hour](#)



---

4.

1. Netflix has moved monolithic Java architecture to node.JS Netflix improved performance and reduced infrastructure costs:
2. Reduced startup time from 40 mins to sub 1 minute
3. Reduced the number of EC2 instances on Node compared with the legacy Java stack by 75%, while serving the same number of subscribers at lower latencies

# NETFLIX

There are sometimes problems that you can't diagnose expediently, or issues such as memory leaks.

So what can we do when we encounter these issues?

## **Do Post-Mortem analysis:**

1. Capture a core dump of running process at any time in production to capture all of the state the process and then reboot it.
2. there are tools that can be used like: mdb\_v8, lnode and IBM's IDDE
3. this offers to Node.js engineers several advantages including:

```
\* Allows service to stay up while developers investigate the problem \ (high availability\)\n\n\* Allows several developers to investigate the same problem not at the same time \ (collaboration\)\n\n\* Allows developers to investigate issues at any time that is convenient for them
```

For further details you can see the [Node.JS Core Post-Mortem Working Group](#) and [best practices from Netflix](#) maintaining high performance in large scale production Node.js applications

Note: there are some problems when trying to do post mortem with [promises](#)



# ECMAScript 2015/ES6 and node js

Ecmascript (or ES) is a trademarked scripting-language specification standardized by Ecma International in ECMA-262 and ISO/IEC 16262. It was created to standardize JavaScript, so as to foster multiple independent implementations.

The other name for ecmascript 2015 is ES6

Node.js is built against modern versions of [V8](#).

By keeping up-to-date with the latest releases of this engine, we ensure new features from the [JavaScript ECMA-262 specification](#) are brought to Node.js developers in a timely manner, as well as continued performance and stability improvements.

All ECMAScript 2015 (ES6) features are split into three groups for **shipping**, **staged**, and **in progress** features:

1. All **shipping** features, which V8 considers stable, are turned **on by default on Node.js** and do **NOT** require any kind of runtime flag.
2. **Staged** features, which are almost-completed features that are not considered stable by the V8 team, require a runtime flag: `--harmony`
3. **In progress** features can be activated individually by their respective harmony flag, although this is highly discouraged unless for testing purposes. Note: these flags are exposed by V8 and will potentially change without any deprecation notice.

## ES6 includes the following new features:

### Arrows

Arrows are a function shorthand using the `=>` syntax.

They are syntactically similar to the related feature in C#, Java 8 and CoffeeScript. They support both statement block bodies as well as expression bodies which return the value of the expression. Unlike functions, arrows share the same lexical `this` as their surrounding code.

```
// Here is an example for an Expression bodies
```

```
var odds = evens.map(v => v + 1);
```

```
var nums = evens.map((v, i) => v + i);
```

```
var pairs = evens.map(v => ({even: v, odd: v + 1}));

// Here is an example for a Statement bodies

nums.forEach(v => {
  if (v % 5 === 0)
    fives.push(v);
});

// Here is an example for aLexical this

var bob = {
```

```
  `_name: "Bob",`
  `_friends: [],`
  `printFriends() {`
    `this._friends.forEach(f =>`
      `console.log(this._name + " knows " + f));`
  `}`
}
```

```
}
```

More info: [MDN Arrow Functions](#)

## classes

ES6 classes are a just syntactic sugar over the prototype-based OO pattern.

Having a single convenient declarative form makes class patterns easier to use, and encourages interoperability. Classes support prototype-based inheritance, super calls, instance and static methods and constructors.

```
class Person {
  constructor(height, age) {
    this.height = height;
    this.age = age;
  }
  get chestsize() {
    return this.calcChestSize();
  }
}
```

```
calcChestSize() {  
  return this.height * 0.23;  
}  
}
```

```
const men = new Person(170, 28);
```

```
console.log(men.chestsize);
```

- enhanced object literals
- template strings
- destructuring
- default + rest + spread
- let + const
- iterators + for..of
- generators
- unicode
- modules
- module loaders
- map + set + weakmap + weakset
- proxies
- symbols
- subclassable built-ins
- promises
- math + number + string + array + object APIs
- binary and octal literals
- reflect api

## tail calls

Calls in tail-position are guaranteed to not grow the stack unboundedly.

The stack will not explode and thus Makes recursive algorithms safe in the face of unbounded inputs.

To read more about [what is a tail recursion](#) and what we are achieving here

# Maps and Sets

This feature helps us to map one to one objects in JavaScript.

This has been long missing from JavaScript and enables to create simple object to object maps with  $O(1)$  access time.

The Map object is a simple key/value map.

It is possible to use it as follows:

```
var someMap = new Map();
var keyString = "mykey",
    keyObj = {},
    keyFunc = function () {};

//performing setters
someMap.set(keyString, "some string");
someMap.set(keyObj, "another string");
someMap.set(keyFunc, "yet another");
```

So everything that is type of object can be a key in the map (string, object, function).

The map also supports the Symbol.iterator which helps to iterate on the map with for..of as follows:

```
for (var v of someMap)
{
    console.log(v);
}
```