

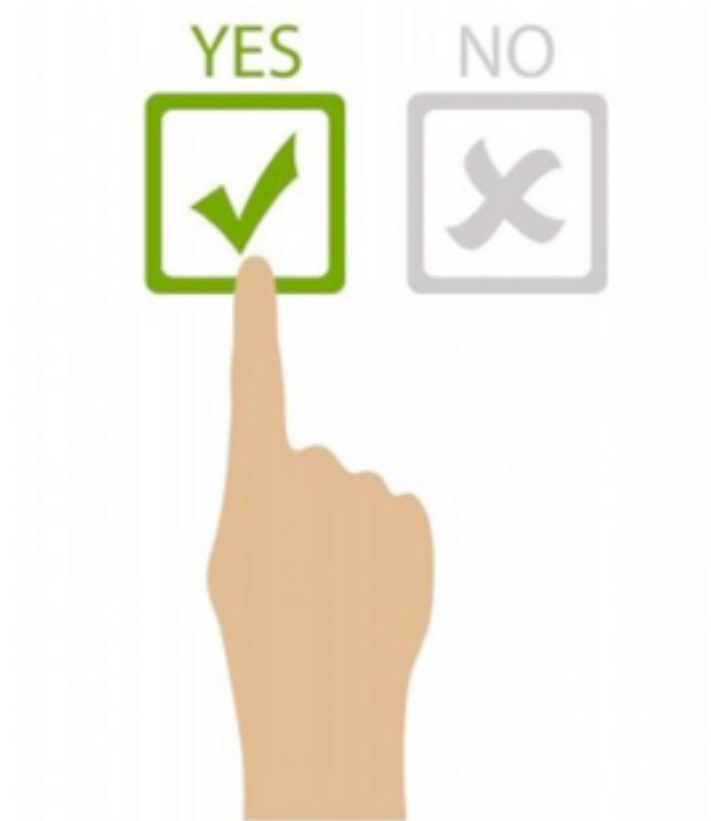
# JavaScript & Node.js

---

Olivier Liechti & Simon Oulevay  
COMEM Web Services 2016

heig-vd

Haute Ecole d'Ingénierie et de Gestion  
du Canton de Vaud



# Quick Poll

- What is your current perception of JavaScript?
- **Scope**
  - It's a "toy" language for creating animations on web pages, but I would not use it for anything "serious".
  - It's a **very powerful language**. It is essential on the client side, but it is also really interesting on the server side.

- What is your current perception of JavaScript?
- **Personal taste**
  - I hate it.
  - I am not a big fan.
  - It's kind of interesting.
  - I love it.
  - I don't care.

- What is your current perception of JavaScript?
- **Relationship to Java**
  - It's Java, with a few syntactic differences.
  - It has nothing to do with Java, except for some common syntax.

- **What is your current perception of JavaScript?**
- **Current knowledge**
  - **Novice:** I may have hacked a few scripts on web pages, but mostly by copy-pasting examples and without fully understanding the language (what is a prototype?).
  - **Intermediate:** I have used JavaScript quite a bit. I can describe the object-oriented model, I understand what a constructor is and how it works. I have quite a bit of experience with JQuery and other libraries. I am always working with a debugger.
  - **Expert:** closures and modules have no secret for me, I have read "JavaScript: the good parts". I have designed my development workflow with yeoman, grunt, bower and a few other tools. I know who Paul Irish is.



heig-vd

Haute Ecole d'Ingénierie et de Gestion  
du Canton de Vaud

# **Douglas Crockford: JavaScript: The Good Parts**

[https://www.youtube.com/watch?v=\\_DKkVvOt6dk](https://www.youtube.com/watch?v=_DKkVvOt6dk)

# JavaScript is built on some very good ideas and a few very bad ones.

**JavaScript is an important language** because it is the language of the web browser. Its association with the browser makes it one of the most popular programming languages in the world. **At the same time, it is one of the most despised programming languages in the world.** [...]

Most people in that situation **don't even bother to learn JavaScript first**, and then they are surprised when JavaScript turns out to have significant differences from the some other language they would rather be using, and that those differences matter.

The amazing thing about JavaScript is that it is possible to get work done with it without knowing much about the language, or even knowing much about programming. It is a language with enormous expressive power. It is even better when you know what you're doing. **Programming is difficult business. It should never be undertaken in ignorance.**

heig-vd

Haute Ecole d'Ingénierie et de Gestion  
du Canton de Vaud

*Unearthing the Excellence in JavaScript*



## JavaScript: The Good Parts

SPD  
O'REILLY\*

YAHOO! PRESS

Douglas Crockford



# JavaScript is important. That wasn't always so, but it's true now.

heig-vd

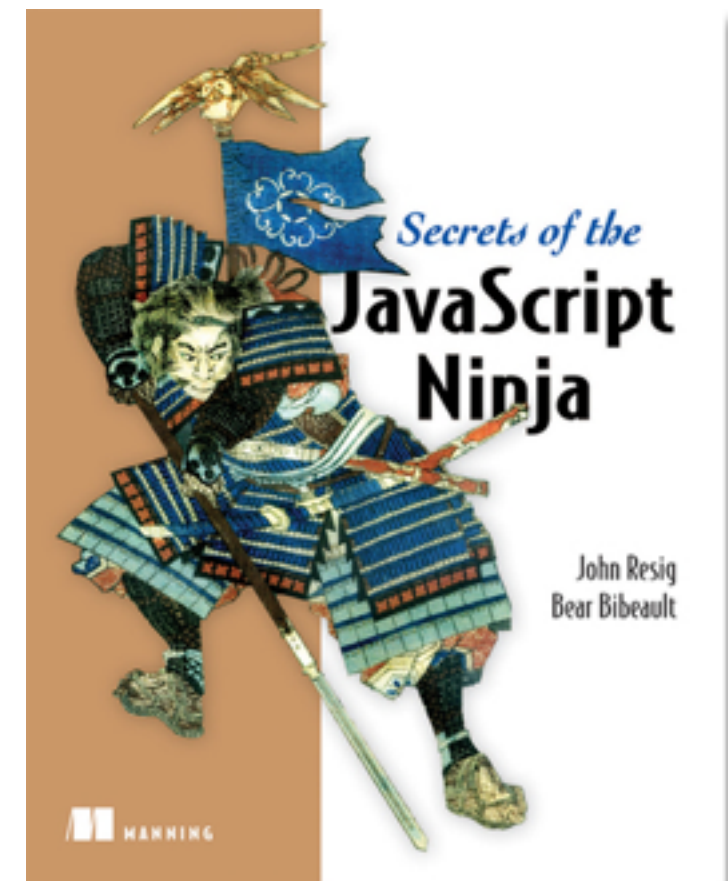
Haute Ecole d'Ingénierie et de Gestion  
du Canton de Vaud

**Web applications are expected to give users a rich user interface experience**, and without JavaScript, you might as well just be showing pictures of kittens. More than ever, web developers need to have a sound grasp of the language that brings life to web applications.

And like orange juice and breakfast, **JavaScript isn't just for browsers anymore**. The language has knocked down the walls of the browser and is being **used on the server** in engines such as Rhino and V8, and in frameworks like Node.js.

Although this book is primarily focused on JavaScript for web applications, the fundamentals of the language presented in part 2 of this book are applicable across the board.

With more and more developers using JavaScript, it's now more important than ever that they **grasp its fundamentals**, so that they can become true ninjas of the language.



# JavaScript 101

# Rule #1

## JavaScript defines 6 types

```
var aNumber = 3.12;
var aBoolean = true;
var aString = "HEIG-VD";
var anObject = {
  aProperty: null
};

// t is true for all of these:
var t;
t = typeof aNumber === "number";
t = typeof aBoolean === "boolean";
t = typeof aString === "string";
t = typeof anObject === "object";
t = typeof anObject.aProperty === "object";
t = typeof anObject.foo === "undefined";
```

- The 6 types are:
  - number
  - boolean
  - string
  - object
  - undefined
  - null
- null is a type, but `typeof null === object`.

# Rule #2

## JavaScript is a dynamic language

```
var myVariable = "aString";  
typeof myVariable; // "string"  
  
myVariable = 3.12;  
typeof myVariable; // "number"  
  
myVariable = true;  
typeof myVariable; // "boolean"  
  
myVariable = {  
  aProperty: "aValue"  
};  
typeof myVariable; // "object"
```

- When you declare a **variable**, you don't specify a type.
- The type can **change** over time.

# Rule #3

There are 2 scopes for variables:  
the (evil) global scope and the function scope

```
var aVariableInGlobalScope;

function myFunction() {
    var aVariableInFunctionScope;
    anotherVariableInGlobalScope;
}

function myFunction2() {
    for (i = 0; i < 10; i++) {
        // i is in global scope!
    }
    for (var j = 0; j < 10; j++) {
        // j is in function scope!
    }
}
```

- A variable declared within a function is **not accessible** outside this function.
- Unless using **strict mode**, it is not mandatory to declare variables (beware of typos...)
- Two scripts loaded from the same HTML page share the same global scope (beware of **conflicts**...).
- There is **no block scope**.

# Rule #4

## JavaScript supports first-class functions

```
function multiplyByTwo(n) {  
    return n * 2;  
}  
  
multiplyByTwo(3); // 6  
  
var hello = function(name) {  
    console.log("Hello " + name + "!");  
};  
  
hello("World"); // "Hello World!"  
  
function applyToArray(array, func) {  
    for (int i = 0; i < array.length; i++) {  
        array[i] = func(array[i]);  
    }  
}  
  
var a = [ 1, 2, 3 ];  
applyToArray(a, multiplyByTwo);  
console.log(a); // [ 2, 4, 6 ]
```

- New functions can be defined at **runtime**.
- Functions can be **stored** in data structures.
- Functions can be passed as **arguments** to other functions.

# Rule #5

## Objects are dynamic bags of properties

```
// let's create an object
var person = {
  firstName: 'olivier',
  lastName: 'liechti'
};

// we can dynamically add properties
person.gender = 'male';
person['zip'] = 1446;

// and delete them
delete person.zip;

for (var key in person) {
  console.log(key + " : " + person[key]);
};
```

- There are different ways to **access properties** of an object.
- JavaScript is **dynamic**: it is possible to **add** and **remove** properties to an object at any time.
- Every object has a different list of properties (**no class**).

# Rule #6

## Arrays are objects

```
var fruits = ["apple", "pear"];

fruits.push("banana");
typeof fruits; // "object"

for (var i = 0; i < fruits.length; i++) {
    console.log("fruits[" + i + "] = " + fruits[i]);
}

var transformedFruits = fruits.map( function(fruit) {
    return fruit.toUpperCase();
});

transformedFruits.forEach( function(fruit) {
    console.log(fruit);
});
```



# Rule #7

The language has no support for classes

There are 3 ways to create objects

heig-vd

Haute Ecole d'Ingénierie et de Gestion  
du Canton de Vaud

```
// create an object with a literal  
var person = {  
  firstName: "olivier",  
  lastName: "liechti"  
};
```

```
// create an object with a prototype  
var child = Object.create(person);
```

```
// create an object with a constructor  
var child = new Person("olivier", "liechti");
```

- **class** is a reserved word in JavaScript, but it is not used in the current version of the language (reserved for the future).
- A **constructor** is function like any other (uppercase is a coding convention).
- It is the use of the **new** keyword that triggers the object creation process.

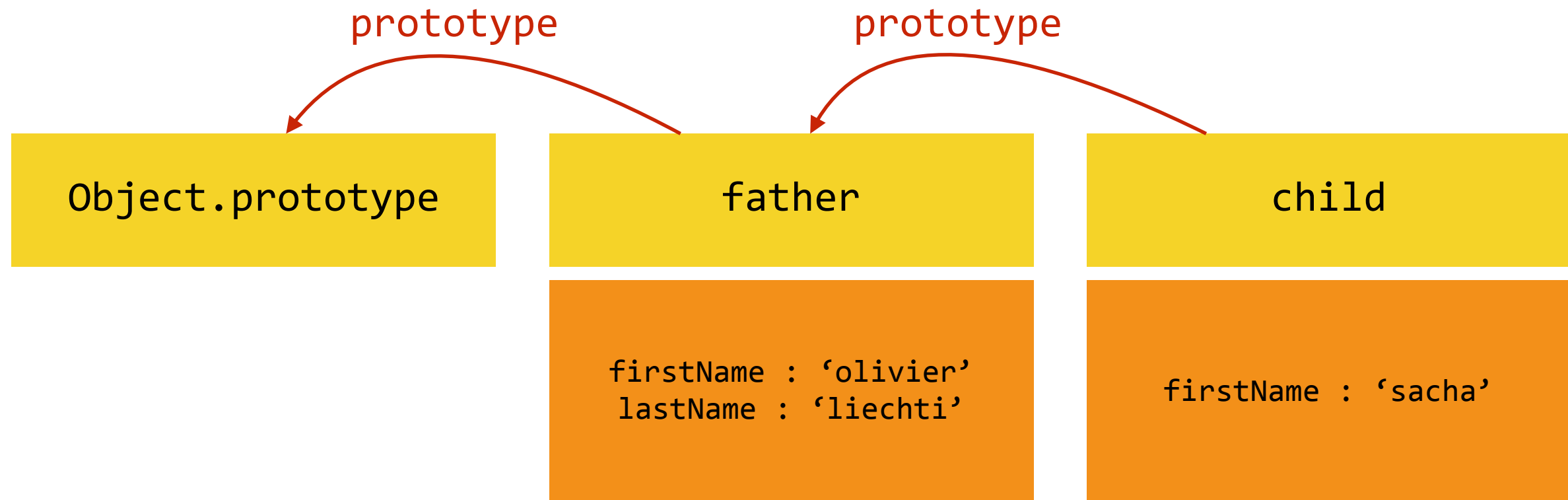
# Rule #8

## Every object inherits from a prototype object

```
var person = {  
  firstName: "olivier",  
  lastName: "liechti"  
};  
// person's prototype is Object.prototype  
  
var father = {};  
var child = Object.create(father);  
// child's prototype is father  
  
function Person(fn, ln) {  
  this.firstName = fn;  
  this.lastName = ln;  
}  
var john = new Person("John", "Doe");  
// john's prototype is Person.prototype
```

# Rule #5

## Every object inherits from a prototype object



```
console.log(child.lastName);  
// prints 'liechti' on the  
console
```

- Every object inherits from a prototype object. **It inherits and can override its properties**, including its methods.
- Objects created with object literals inherit from **Object.prototype**.
- When you access the property of an object, JavaScript **looks up the prototype chain** until it finds an ancestor that has a value for this property.

# Rule #6

## With patterns, it is possible to implement class-like data structures

```
function Person(fn, ln) {  
  var privateVar;  
  this.firstName = fn;  
  this.lastName = ln;  
  this.badGreet = function() {  
    console.log("Hi " + this.firstName);  
  };  
};  
  
Person.prototype.greet = function() {  
  console.log("Hey " + this.firstName);  
};  
  
var p1 = new Person("olivier", "liechti");  
  
p1.badGreet(); // "Hi olivier"  
p1.greet();    // "Hey olivier"
```

- **badGreet** is a property that will be replicated for every object created with the Person constructor:
  - poor memory management
  - not possible to alter behavior of all instances at once
- **greet** is a property that will be shared by all instances (because it will be looked up along the object inheritance chain).
- **privateVar** is not accessible outside of the constructor.
- **firstName** is publicly accessible (no encapsulation).

# JavaScript resources

---

- **A re-introduction to JavaScript**

- [https://developer.mozilla.org/en-US/docs/Web/JavaScript/A\\_re-introduction\\_to\\_JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript)

- **Inheritance and the prototype chain**

- [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance\\_and\\_the\\_prototype\\_chain](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance_and_the_prototype_chain)

- **Introduction to Object-Oriented JavaScript**

- [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction\\_to\\_Object-Oriented\\_JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript)

JavaScript Objects in Detail

javascriptissexy.com/javascript-objects-in-detail/

JavaScript.is (Sexy)

Learn everything about modern web application development with JavaScript and HTML5.

Sponsored By:

Grammar & Writing for Creators

Improve Quickly and Significantly  
Write Powerful & Eloquent Prose

Bloggers · Writers · Programmers · Designers · Other Professionals

RICHARD OR STANLEY

Follow on Twitter

Recent Posts

Beautiful JavaScript: Easily Create

JavaScript Objects in Detail

January 27 Last Year

JavaScript's core—most often used and most fundamental—data type is the Object data type. JavaScript has one complex data type, the Object data type, and it has five simple data types: Number, String, Boolean, Undefined, and Null. Note that these simple (primitive) data types are immutable, they cannot be changed, while objects are mutable.

What is an Object

An object is an unordered list of primitive data (and sometimes reference data types) types that are stored as name-value pairs. Each item in the list is called a property (functions are called methods) and each property name has to be unique and can be a string or a number.

Here is a simple object:

```
var myFirstObject = {firstName: "Richard", favoriteAuthor: "Conrad"};
```

To reiterate: Think of an object as a list that contains items and each item (a property) in the list is stored by a name-value pair. The property names in the example above are firstName and favoriteAuthor. And the values for each are "Richard" and "Conrad."

Show Modern Dev Ad



# Node.js



“Node.js is a **platform** built on Chrome's JavaScript runtime for easily building **fast, scalable network applications.** *not only!*

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.**”





heig-vd

Haute Ecole d'Ingénierie et de Gestion  
du Canton de Vaud

## Trends

### Topics

Subscribe



**Node.js**

Software

**Ruby on Rails**

Web Framework

+ Add term

Beta: Measuring search interest in *topics* is a beta feature which quickly provides accurate measurements of overall search interest. To measure search interest for a specific *query*, select the "search term" option. ?

### Interest over time ?

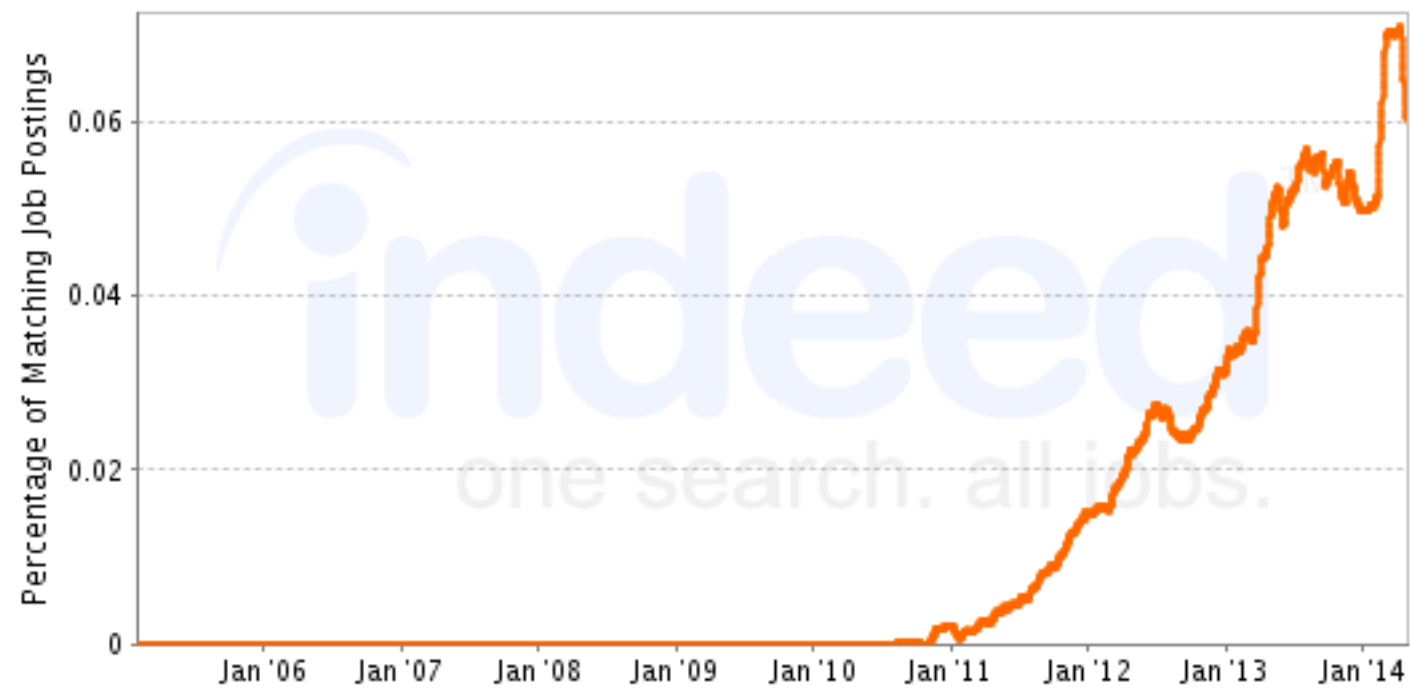
☐ News headlines ?

☐ Forecast ?



Job Trends from Indeed.com

node js



heig-vd

Haute Ecole d'Ingénierie et de Gestion  
du Canton de Vaud

Job Trends from Indeed.com

ruby on rails





# Single-threaded      Asynchronous

“I am not leaving until you are done...”

vs

“Wake me up when you have result”



# Let's look at a **synchronous** example

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

/**
 * Simple function to test the synchronous readFileSync function of Node.js
 * @param {string} filename - the file to read
 */
function testSyncRead(filename) {
  console.log("I am about to read a file: " + filename);
  var data = fs.readFileSync(filename);
  console.log("I have read " + data.length + " bytes (synchronously).");
  console.log("I am done.");
}

// We get the file name from the argument passed on the command line
var filename = process.argv[2];

console.log("\nTesting the synchronous call");
testSyncRead(filename);
```

We use a standard **Node module** for accessing the file system

**fs.readFileSync** is synchronous: it blocks the main thread until the data is available.

```
$ node sample2.js medium.txt
```

```
Testing the synchronous call
I am about to read a file: medium.txt
I have read 1024 bytes (synchronously).
I am done.
```



*Synchronous functions are easier to use, but they have **severe** performance implications!!*



# Let's look at an **asynchronous** example

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

/**
 * Simple function to test the asynchronous readFile function of Node.js
 * @param {string} filename - the file to read
 */
function testAsyncRead(filename) {
  console.log("I am about to read a file: " + filename);

  fs.readFile(filename, function (err, data) {
    console.log("Nodes just told me that I have read the file.");
  });

  console.log("I am done. Am I?");
}

// We get the file name from the argument passed on the command line
var filename = process.argv[2];

console.log("\nTesting the asynchronous call");
testAsyncRead(filename);
```

**fs.readFile** is asynchronous:  
it does not block the main  
thread until the data is  
available.

We must provide a  
**callback function**, which  
Node.js will invoke when the  
data is available.

**Problems** can happen  
when an (asynchronous)  
function is called.



Node.js developers **have to** learn  
the asynchronous programming  
style.

```
$ node sample2.js medium.txt
```

```
Testing the asynchronous call
I am about to read a file: medium.txt
I am done. Am I?
Nodes just told me that I have read the file.
```



# Node.js **callback** convention

When calling an asynchronous operation, the convention in Node.js is to use a **callback function** that will be called with **2 arguments**. If an error occurs, **the first argument** will be an error describing the problem. If the operation succeeds, **the second argument** will be the result. You will only receive one or the other, not both.

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

fs.readFile(filename, function (err, data) {

  if (err) {
    // handle the error
    console.warn("An error occurred: " + err.message);
    return;
  }

  // do something with the data
  process(data)
});
```

The callback function with its two arguments.

Before doing anything with the data, **check** if there is an error.

If there was an error, handle it and **stop there**. There is no data available.

Otherwise, if there was no error, you may use the **data**.

# Node.js v0.10.32 Manual & Documentation

heig-vd

Haute Ecole d'Ingénierie et de Gestion  
du Canton de Vaud

- Assertion Testing

- Buffer

- C/C++ Addons

- Child Processes

- Cluster

- Console

- Crypto

- Debugger

- DNS

- Domain

- Events

- File System

- Globals

- HTTP

- HTTPS

- Modules

- Net

- OS

- Path

- Process

- Punycode

- Query Strings

- Readline

- REPL

- Stream

- String Decoder

- Timers

- TLS/SSL

- TTY

- UDP/Datagram

- URL

- Utilities

- VM

- ZLIB





## Let's look at a **another example**

```
/*global require */  
var http = require("http");  
  
/**  
 * This function starts a http daemon on port 9000. It also  
 * registers a callback handler, to handle incoming HTTP  
 * requests (a simple message is sent back to clients).  
 */  
function runHttpServer() {  
  var server = http.createServer();  
  
  server.on("request", function (req, res) {  
    console.log("A request has arrived: URL=" + req.url);  
    res.writeHead(200, {  
      'Content-Type': 'text/plain'  
    });  
    res.end('Hello World\n');  
  });  
  
  console.log("Starting http server...");  
  server.listen(9000);  
}  
  
runHttpServer();
```

We use a standard **Node module** that takes care of the HTTP protocol.

Node can provide us with a **ready-to-use** server.

We can attach **event handlers** to the server. Node will notify us asynchronously, and give us access to the request and response.

We can **send back** data to the client.

We have wired everything, let's **welcome** clients!



HTTP Node.js v0.10.32 Ma x

nodejs.org/api/http.html#http\_event\_request

NPM REGISTRY

DOCS

BLOG

COMMUNITY

LOGOS

JOB

@nodejs

# Node.js v0.10.32 Manual & Documentation

Index | View on single page | View as JSON

## Table of Contents

- [HTTP](#)
  - [http.STATUS\\_CODES](#)
  - [http.createServer\(\[requestListener\]\)](#)
  - [http.createClient\(\[port\], \[host\]\)](#)
  - Class: [http.Server](#)
    - [Event: 'request'](#)
    - [Event: 'connection'](#)
    - [Event: 'close'](#)
    - [Event: 'checkContinue'](#)
    - [Event: 'connect'](#)
    - [Event: 'upgrade'](#)
    - [Event: 'clientError'](#)
    - [server.listen\(port, \[hostname\], \[backlog\], \[callback\]\)](#)
    - [server.listen\(path, \[callback\]\)](#)
    - [server.listen\(handle, \[callback\]\)](#)
    - [server.close\(\[callback\]\)](#)
    - [server.maxHeadersCount](#)
    - [server.setTimeout\(msecs, callback\)](#)
    - [server.timeout](#)
  - Class: [http.ServerResponse](#)
    - [Event: 'close'](#)
    - [Event: 'finish'](#)
    - [response.writeContinue\(\)](#)

These are the events that are **emitted** by the class. You can write callbacks and **react** to these events.



How does Node.js use an **event loop** to offer an asynchronous programming model?

```
on('request', function(req, res) { // my code});
```

```
on('data', function(data) { // my code});
```

**Callback functions** that **you** have written and registered

'request' **event**

'request' **event**

'data' **event**

'request' **event**

Queue of events that have been **emitted**

## Event Loop

get the next event in the queue; invoke the registered callbacks in sequence; delegate I/O operations to the Node platform



All the code that **you** write runs on a **single thread**

The **long-running tasks** (I/Os) are executed by Node in parallel; Node emits events to report progress (which triggers your callbacks).

Another pattern is to provide a callback to node when invoking an asynchronous function.

# Node.js resources

---

- **Understanding the Node.js Event Loop**
  - <http://strongloop.com/strongblog/node-js-event-loop/>
- **Mixu's Node book: What is Node.js? (chapter 2)**
  - <http://book.mixu.net/node/ch2.html>
- **Node.js Explained, video**
  - <http://kunkle.org/talks/>



# Node.js package manager



What about **package management**?



Not reinventing  
the wheel...

... building on the  
shoulders of  
giants.





## What is **npm**?

*"**npm** is the **package manager** for the Node JavaScript platform. It puts **modules** in place so that node can find them, and manages **dependency** conflicts intelligently.*

*It is extremely configurable to support a wide variety of use cases. Most commonly, it is used to **publish**, **discover**, **install**, and **develop** node programs."*

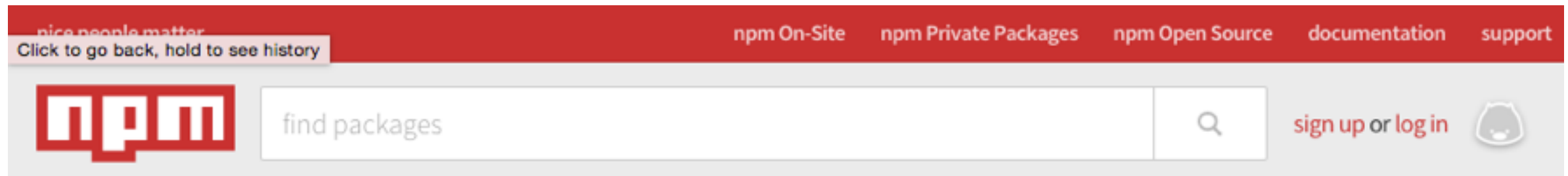


<https://docs.npmjs.com/getting-started/what-is-npm>





npm is a set of command line tools that work together with the **node registry**



npm is the package manager for **javascript**.



243,005  
total packages



59,432,329  
downloads in the last day



856,457,025  
downloads in the last week



3,487,903,280  
downloads in the last month

packages people 'npm install' a lot



**browserify**  
browser-side require() the node way  
13.0.0 published a month ago by feros

express

**express**  
Fast, unopinionated, minimalist we...  
4.13.4 published a month ago by doug...



**pm2**  
Production process manager for No...  
1.0.0 published 2 months ago by tknew



**grunt-cli**  
The grunt command line interface.  
0.1.13 published 2 years ago by tkellen



**npm**  
a package manager for JavaScript  
3.7.1 published 3 weeks ago by iarna



**karma**  
Spectacular Test Runner for JavaSc...  
0.13.19 published 2 months ago by dig...

<https://www.npmjs.org>



npm is a set of command line tools that work together with the **node registry**

nice people matter    npm On-Site    npm Private Packages    npm Open Source    documentation    support

**npm**    express          sign up or log in   

*Packages*

**express**  
Fast, unopinionated, minimalist web framework

**express-session**  
Simple session middleware for Express

**express-handlebars**  
A Handlebars view engine for Express which doesn't suck.

**express-validator**  
Express middleware for the validator module.

**express-jwt**  
JWT authentication middleware.

*Search Suggestions*

**243,005**  
total packages

**3,487,903,280**  
downloads in the last month

<https://www.npmjs.org>





npm is a set of command line tools that work together with the **node registry**

The screenshot shows the npm website interface. At the top is a red navigation bar with links: 'nightly panic munchies', 'npm On-Site', 'npm Private Packages', 'npm Open Source', 'documentation', and 'support'. Below this is a search bar with the 'npm' logo and the text 'find packages'. To the right of the search bar are links for 'sign up or log in' and a user profile icon. The main content area features the 'express' package page. It includes the package name 'express' with a 'public' tag, a description 'Fast, unopinionated, minimalist web framework', and a large 'express' logo. Below the logo are several status badges: 'npm v4.13.4', 'downloads 5M/month', 'linux passing', 'windows passing', and 'coverage 100%'. A code block shows a sample Express.js application setup. On the right side, there is a section for 'npm Orgs' with a description and a 'Learn more' link. Below that, there is a section for the latest release '4.13.4' and a link to the GitHub repository 'github.com/expressjs/express'. At the bottom right, there is a 'MIT' license logo.

nightly panic munchies npm On-Site npm Private Packages npm Open Source documentation support

npm find packages sign up or log in

express public  
Fast, unopinionated, minimalist web framework

express

npm v4.13.4 downloads 5M/month linux passing windows passing coverage 100%

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

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

app.listen(3000)
```

npm Orgs  
Everything's better in groups. Manage developer teams with varying permissions and multiple projects. [Learn more »](#)

npm install express  
[how? learn more](#)

[dougwilson](#) published a month ago

4.13.4 is the latest of 273 releases

[github.com/expressjs/express](https://github.com/expressjs/express)

MIT

<https://www.npmjs.org>



npm provides many **commands**

admin — bash — 120x30

Last login: Wed Sep 23 19:59:09 on console

\$ npm help

Usage: npm <command>

where <command> is one of:

access, add-user, adduser, apihelp, author, bin, bugs, c,  
cache, completion, config, ddp, dedupe, deprecate, dist-tag,  
dist-tags, docs, edit, explore, faq, find, find-dupes, get,  
help, help-search, home, i, info, **init**, **install**, issues, la,  
link, list, **login**, ls, outdated, owner, pack,  
prefix, prune, **publish**, rb, rebuild, remove, repo,  
restart, rm, root, run-script, s, se, search, set, show,  
shrinkwrap, star, stars, start, stop, tag, test, tst, un,  
uninstall, unlink, unpublish, unstar, **update**, v,  
verison, version, view, whoami

npm <cmd> -h      quick help on <cmd>  
npm -l            display full usage info  
npm faq           commonly asked questions  
npm help <term>   search for help on <term>  
npm help npm      involved overview

Specify configs in the ini-formatted file:

/Users/admin/.npmrc

or on the command line via: npm <command> --key value

Config info can be viewed via: npm help config

npm@2.5.1 /usr/local/lib/node\_modules/npm

\$

init  
install  
publish  
update



## npm can **install packages**

- You can **install packages** with **npm install [options] <name>**:
  - **globally** (this is the case for tools and CLI utilities used across projects):
    - You need to use the **-g** flag if you want to do that (and often **sudo** on Unix systems).
  - **locally to a project** (this is the case for libs that your code depends on):
    - In this case, you first use **npm init** which will create a **package.json** file. This file lists the packages you use and at what versions.
    - Then you can use npm install with the **--save** flag, which will install the latest version of the package you want and save its version to the package file.
    - The modules are stored in a (often large) directory named **node\_modules**, which you typically add to your **.gitignore** file.



# How do I **reuse** code?

m1.js

```
module.exports.a = "b";

module.exports.hello = function() {
  console.log("Hello!");
};
```

m2.js

```
module.exports = function(name) {
  console.log("Hello " + name + "!");
};
```

myScript.js

```
var module1 = require("./m1");

console.log(module1.a);
module1.hello();

var module2 = require("./m2");

module2("World");
```

```
$> node myScript.js
b
Hello!
Hello World!
```

Run it!



How do I reuse code through **npm**?



KEEP  
CALM  
AND  
SAY  
HELLO

Someone wrote **myHello**.  
I want to use it in **myScript**.  
How?



How do I **publish** myHelloModule as an npm **package**?

**myHello**/hello.js

```
module.exports.hello = function() {  
  console.log("Hello!");  
};
```

**myHello**/package.json

```
{  
  "name": "myHello",  
  "version": "1.0.0",  
  "description": "My awesome module.",  
  "main": "./hello.js",  
  "repository": {  
    "type": "git",  
    "url": "https://github.com/jdoe/hello"  
  },  
  "author": "John Doe <john.doe@example.com>",  
  "license": "MIT"  
}
```

Define the package:

Publish it!

```
$> cd  
myHelloModule  
$> npm publish
```



How do I **install** myHelloModule and **use** it?

Add the package as a **dependency** to your own package:

myScript/package.json

```
{
  "name": "myApp",
  "version": "1.0.0",
  "description": "My awesome app.",
  "main": "script.js",
  "dependencies": {
    "myHello": "1.0.0"
  }
}
```

**Install** dependencies:

```
$> npm install
```

**Require** and use the dependency:

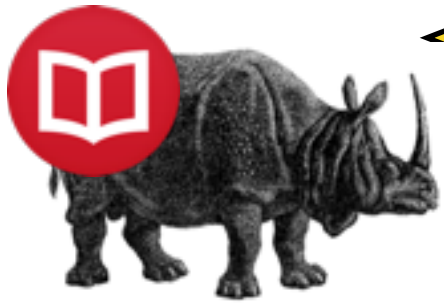
myScript/script.js

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

myHello.hello();
```

Enjoy:

```
$> node script.js
Hello!
```



How do I use npm if I'm lazy?

Install the latest version of a new module and automatically save it to your **package.json** file:

```
$> npm install --save myHello
```

(Note: Your package.json file must already exist.)



Command line usage

<https://www.npmjs.org/doc/cli/npm.html>

package.json format

<https://docs.npmjs.com/files/package.json>