Web Application Development

Produced by

David Drohan (ddrohan@wit.ie)

Department of Computing & Mathematics
Waterford Institute of Technology

http://www.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

PART 1

SERVER SIDE JAVASCRIPT

Outline

1. **Introduction** – What Node is all about
2. **Events** – Nodes Event-Driven, Non-Blocking I/O model
3. **Node Modules** – The Building Blocks of Node
4. **Express** – A Framework for Node
5. **REST** – The Architectural Style of the Web
6. **API Design** – Exposing Application Functionality
7. **REST in Express** – Leveraging URLs, URI's and HTTP
8. **Demo** – Labs in action

Outline

1. Introduction – What Node is all about
2. Events – Nodes Event-Driven, Non-Blocking I/O model
3. Node Modules – The Building Blocks of Node
4. Express – A Framework for Node
5. **REST** – The Architectural Style of the Web
6. **API Design** – Exposing Application Functionality
7. **REST in Express** – Leveraging URLs, URI's and HTTP
8. **Demo** – Labs in action

Introduction

WHAT NODE IS ALL ABOUT

Background

V8 is an open source JavaScript engine developed by Google. Its written in C/C++ and is used in Google Chrome Browser (and is fast!)

Node.js runs on V8.

It was created by **Ryan Dahl** in 2009.

Finally out of Beta phase.

- Latest LTS (Long Term Support) Version is v**6.11.3 (see next Slide)**
- Latest stable current version is v**8.5.0** (nodejs.org) **(as at 22/09/17)**

Is **Open Source**. It runs well on Linux systems, can also run on Windows systems.

It comes with a built-in **HTTP server library**

It has lots of libraries and tools available; even has its own **package manager (npm)**

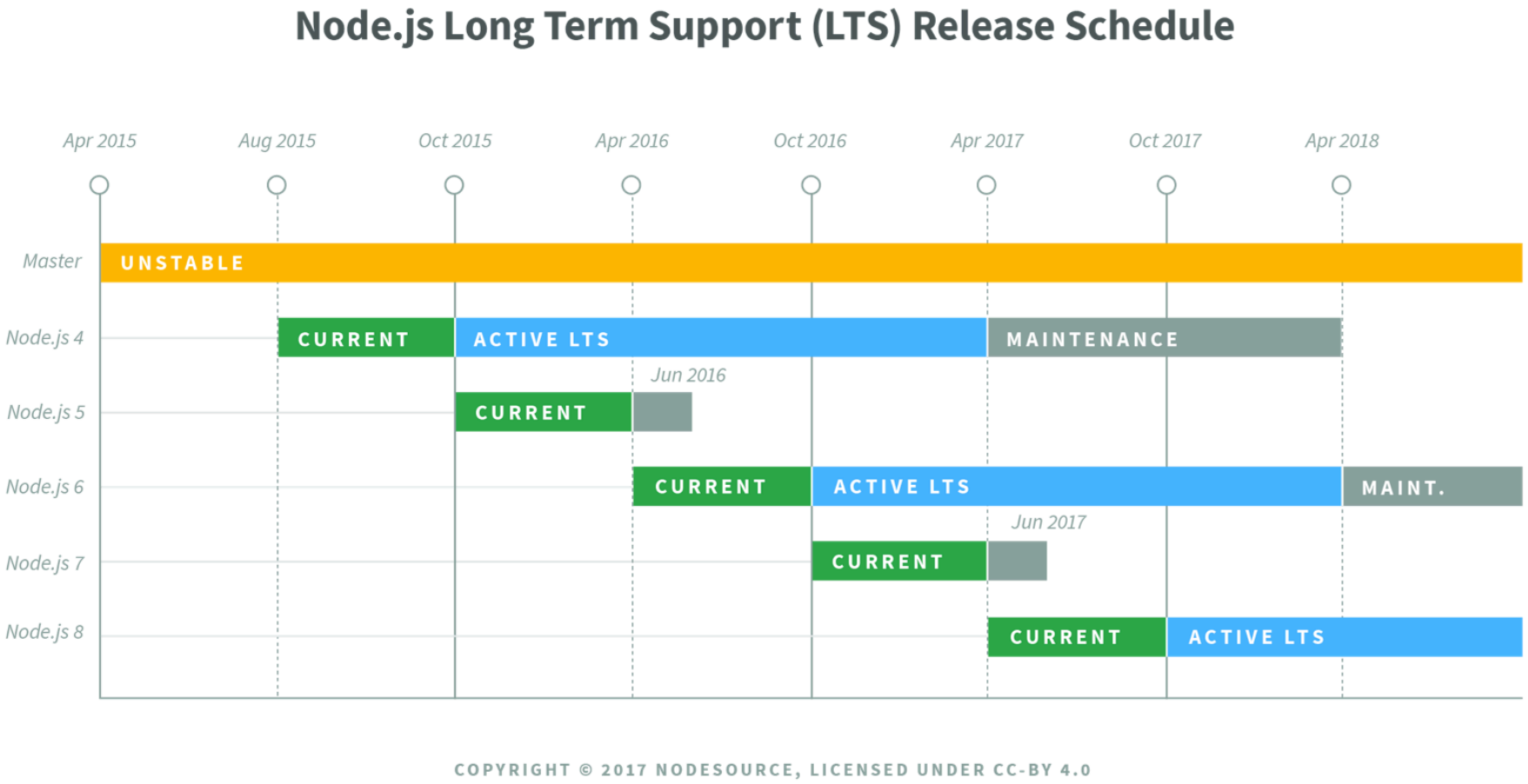
Node.js Release Working Group

Release schedule[1]

Release	LTS Status	Codename	Active LTS Start	Maintenance Start	Maintenance End
v0.10.x	**End-of-Life**	-	-	2015-10-01	2016-10-31
v0.12.x	**End-of-Life**	-	-	2016-04-01	2016-12-31
4.x	**Maintenance**	Argon	2015-10-01	2017-04-01	April 2018
5.x	No LTS				
6.x	**Active**	Boron	2016-10-18	April 2018	April 2019
7.x	No LTS				
8.x	**Pending**	Carbon	October 2017	April 2019	December 2019[2]
9.x	No LTS				
10.x	**Pending**	Pending	October 2018	April 2020	April 2021

- [1]: All scheduled dates are subject to change by the Node.js Release working group or Node.js Core Technical Committee.
- [2]: The 8.x *Maintenance* LTS cycle is currently scheduled to expire early on December 31, 2019 to align with the scheduled End-of-Life of OpenSSL-1.0.2. Note that this schedule *may* change if the version of OpenSSL is upgraded to 1.1.x before 8.x enters the *Active* LTS cycle.

Node.js Release Working Group



Introduction: Basic

In simple words Node.js is '**server-side JavaScript**'.

In not-so-simple words Node.js is a **high-performance** network applications framework, well optimized for high concurrent environments.

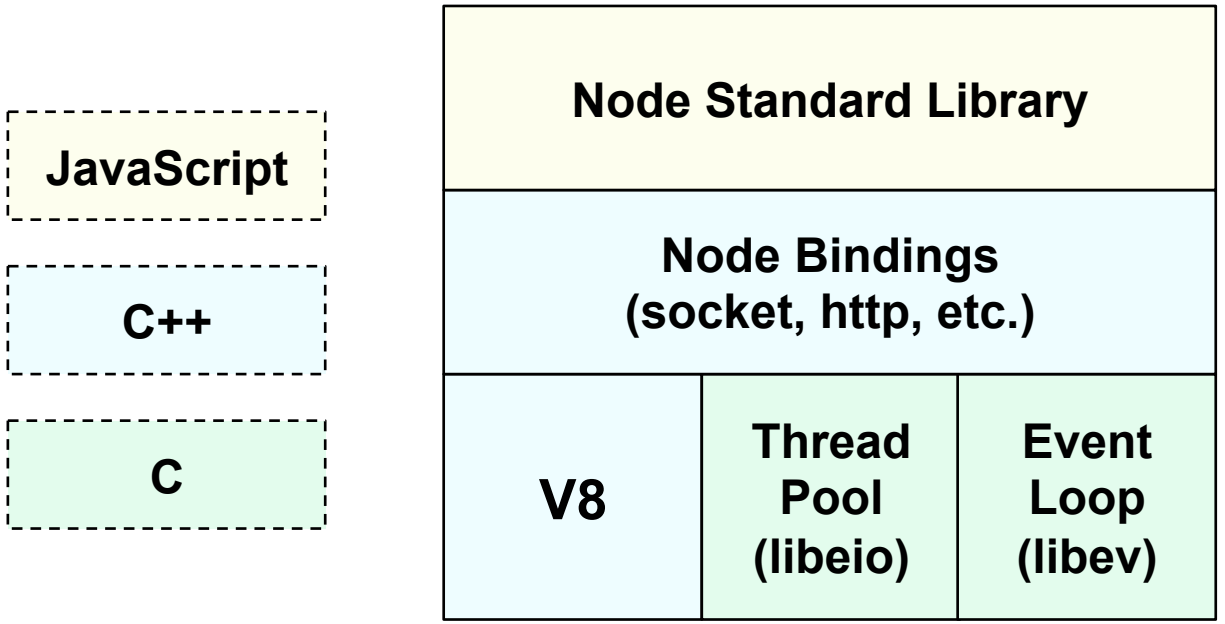
It's a **command line** tool.

In 'Node.js' , '.js' doesn't mean that its solely written in JavaScript. It is **40%** JS and **60%** C++. (next slide)

From the official site:

'Node's goal is to provide an easy way to build scalable network programs' - (from nodejs.org!)

Introduction: Node Architecture



Introduction: Advanced (& Confusing)

Node.js uses an **event-driven**, **non-blocking I/O** model, which makes it lightweight. (again, from nodejs.org!)

It makes use of **event-loops** via JavaScript's **callback** functionality to implement the **non-blocking I/O**.

Programs for Node.js are written in JavaScript but not in the same JavaScript we are use to. There is **no DOM implementation** provided by Node.js, i.e. you **can not** do this:

```
var element = document.getElementById("elementId");
```

Everything inside Node.js runs in a **single-thread** (which must **never block**!).

If your program needs to wait for something (e.g., a response from some server you contacted), it **must provide a callback function**

NODE IS DEPLOYED BY BIG BRANDS

Big brands are using Node to power their business

Manufacturing

BMW
GE
General Motors
Johnson Controls
SIEMENS

Financial

ADP
citigroup
Goldman Sachs
PayPal
WELLS FARGO

eCommerce

amazon.com
BEST BUY
ebay
TARGET
Zappos.com

Media

beatsMUSIC
CONDÉ NAST
DOWJONES
The New York Times
SONY

Technology

salesforce.com
box
intel
YAHOO!

Local Company

And more recently….

MetLife
MasterCard
airbnb
DEPARTMENT OF THE NAVY · UNITED STATES OF AMERICA
nearForm
travelocity
STAPLES
PEARSON
BARNES&NOBLE
npr
Morgan Stanley
HBO
Go Daddy.com
NETFLIX

When to use Node.js?

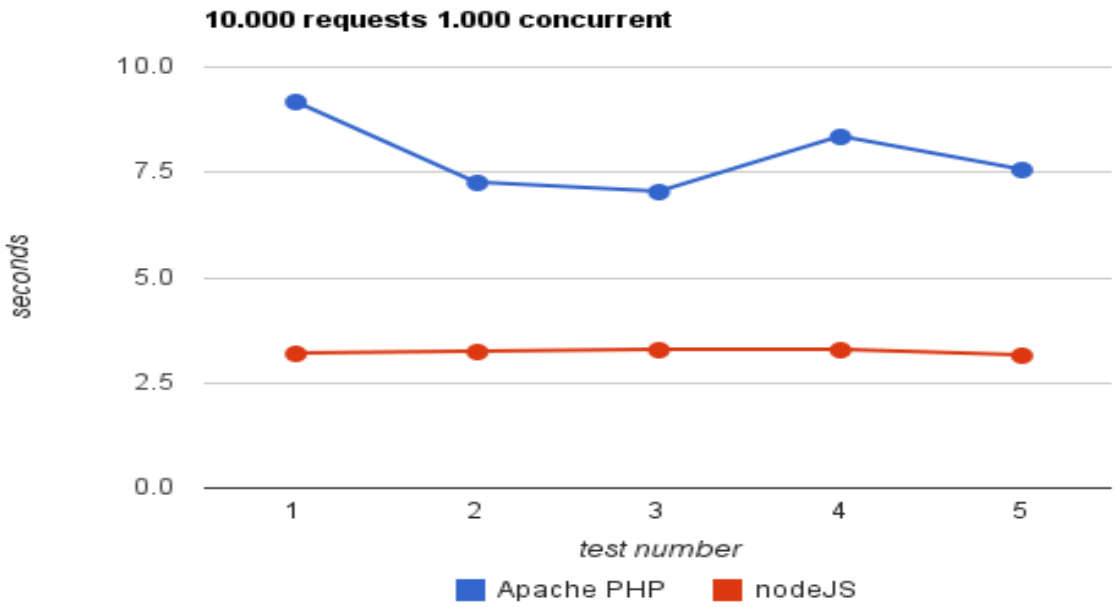
Node.js is good for creating streaming based real-time services, web chat applications, static file servers etc.

If you need high level concurrency and not worried about CPU-cycles.

If you are great at writing JavaScript code because then you can use the same language at both the places: *server-side and client-side*.

More can be found at: http://stackoverflow.com/questions/5062614/how-to-decide-when-to-use-nodejs

Some Node.js benchmarks

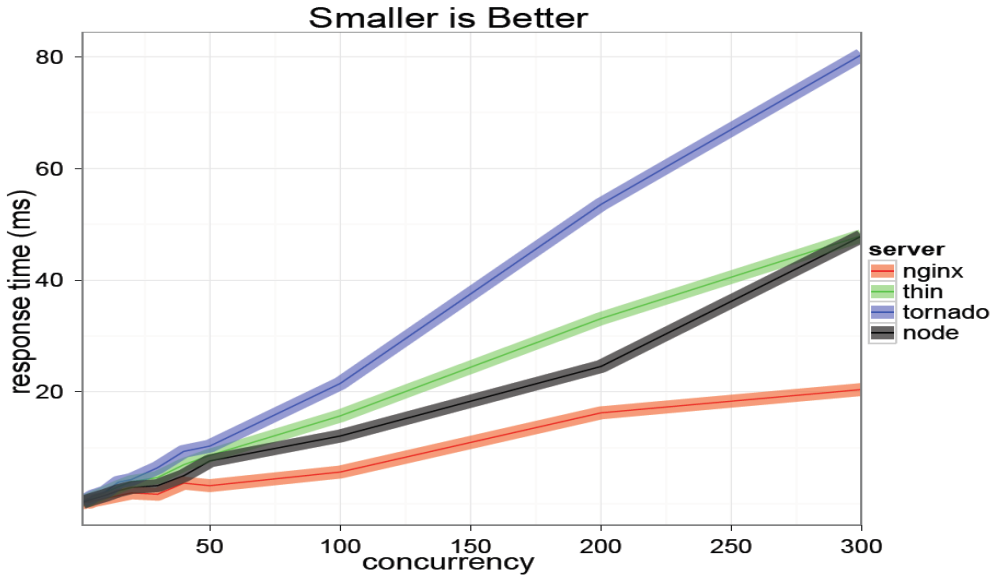


Taken from:
http://code.google.com/p/node-js-vs-apache-php-benchmark/wiki/Tests

A benchmark between Apache+PHP and node.js, shows the response time for 1000 concurrent connections making 10,000 requests each, for 5 tests.

Taken from: http://nodejs.org/jsconf2010.pdf

The benchmark shows the response time in milli-secs for 4 evented servers.



When to not use Node.js

When you are doing heavy and CPU intensive calculations on server side, because event-loops are CPU hungry.

Node.js API is finally out of beta, but it will keep on changing from one revision to another and there is a very little backward compatibility. A lot of the packages are also unstable. Therefore is not production ready just yet.

Node.js is a no match for enterprise level application frameworks like Spring(java), Django(python), Symfony(php) etc. Applications written on such platforms are meant to be highly user interactive and involve complex business logic.

Read further on disadvantages of Node.js on Quora:
http://www.quora.com/What-are-the-disadvantages-of-using-Node-js

Events

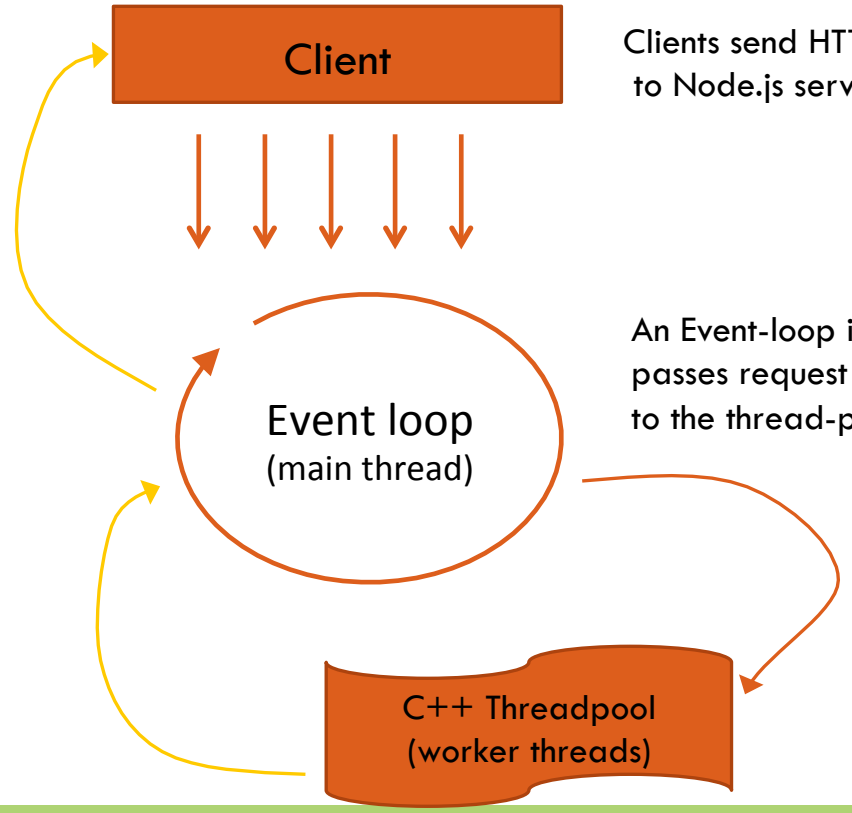
NODES EVENT-DRIVEN, NON-BLOCKING I/O MODEL

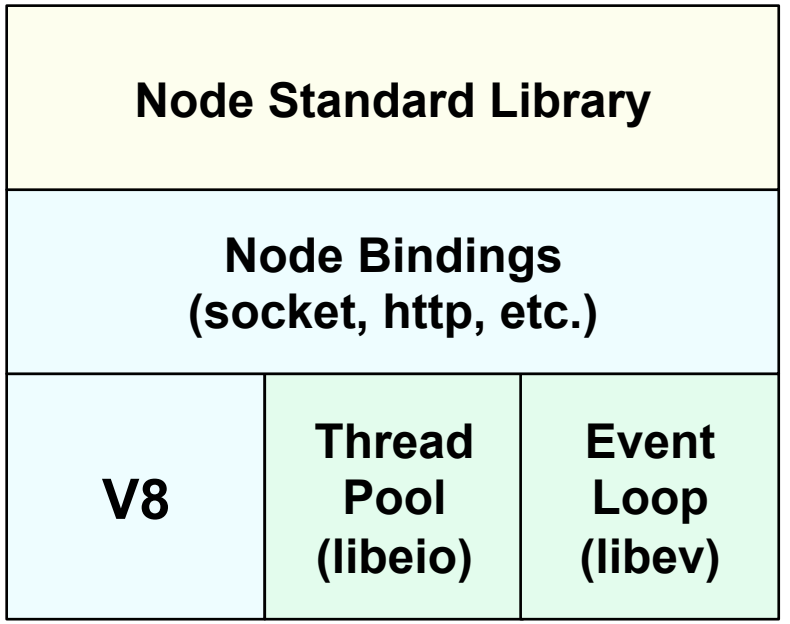
Some Theory: Events

- Generally, input/output (I/O) is slow.
  - Reading/writing to data store, probably across a network.
- Calculations in cpu are fast.
  - 2+2=4
- Most time in programs is spent waiting for I/O to complete.
  - In applications with lots of concurrent users (e.g. web servers), you can't just stop everything and wait for I/O to complete.
- Solutions to deal with this are:
  - Blocking code with multiple threads of execution (e.g. Apache, IIS Servers)
  - Non-blocking, event-based code in single thread (e.g. NGINX, Node.js Servers)

Some Theory: Event-loops

Event-loops are the core of event-driven programming, almost all the UI programs use event-loops to track the user event, for example: Clicks, Ajax Requests etc.





Some Theory: Event-loops

Warning! Be careful to keep CPU intensive operations off the event loop.

Some Theory: Non-Blocking I/O

Traditional I/O

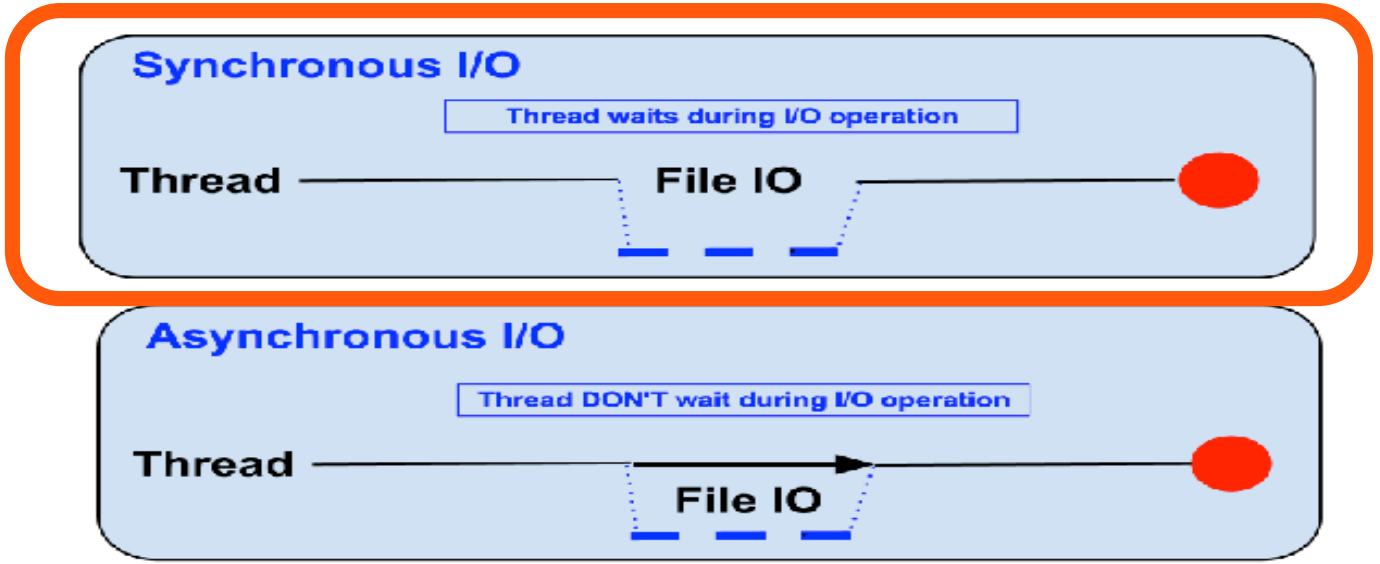
```
var result = db.query("select x from table_Y");
doSomethingWith(result); //wait for result!
doSomethingWithOutResult(); //execution is blocked!
```

Non-traditional, Non-blocking I/O

```
db.query("select x from table_Y",function (result){
  doSomethingWith(result); //wait for result!
});
doSomethingWithOutResult(); //executes without any delay!
```

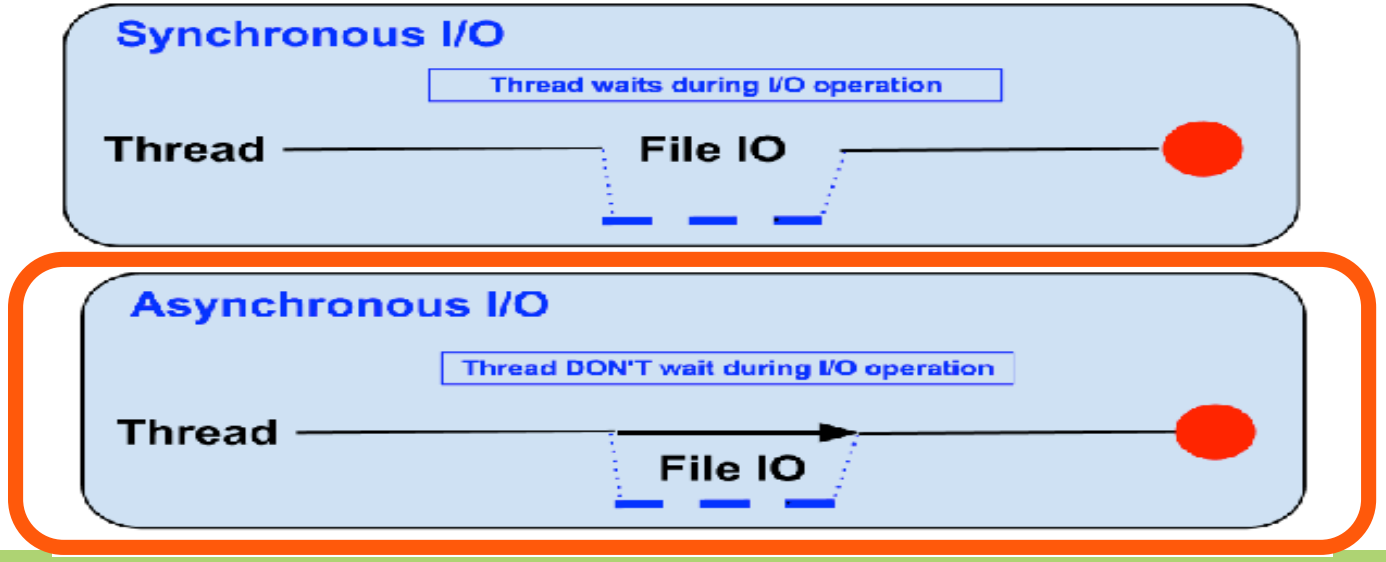
Blocking (Traditional)

- Traditional code waits for input before proceeding (Synchronous)
- The thread on a server "blocks" on I/O and resumes when it returns.



Non-blocking (Node)

- Node.js code runs in a Non-blocking (Asynchronous), event-based Javascript thread
  - No overhead associated with threads
  - Good for high concurrency (i.e. lots of client requests at the same time)



Blocking vs. Non-blocking

- Threads consume resources
  - Memory on stack
  - Processing time for context switching etc.
- No thread management on single threaded apps
  - Just execute “callbacks” when event occurs
  - Callbacks are usually in the form of anonamous functions.

Blocking | I/O Model

Example: ways in which a server can process orders from customers

- The server serves one customer at a time.
- As each customer is deciding on their order, the server **sits and waits**.
- When the customer decides on an order, the server processes their order and moves on to the next customer.

Blocking | I/O Model

Blocking | I/O Model

Pseudocode:

```
order1 = db.query("SELECT * FROM
menu WHERE preference = most")

order1.process

order2.process
```

Blocking | I/O Model

The more customers you want to serve at once, the more cashier lines you'll need.

Cashier lines ~ threads in computing

Multi-threaded processing

Parallel code execution

Multiple CPUs run at a time, utilizing shared resources (memory)

Non-Blocking | I/O Model

- Node loops through the customers and polls them to determine which ones are ready to order.
- During a function's queue, Node can listen to another event.
- When the other customer is finally ready to order, he'll issue a *callback*.
- Asynchronous **callbacks**: *"come back to me when I'm finished"*
  - function called at the completion of a given task.

Non-Blocking | I/O Model

Node code

```
console.log('Hello');

setTimeout(function () {
  console.log('World');
}, 5000);

console.log('Bye');

// Outputs:
// Hello
// Bye
// World
```

Allows for high concurrency

Non-Blocking | I/O Model

***Every* function in Node is non-blocking**

Single-threaded

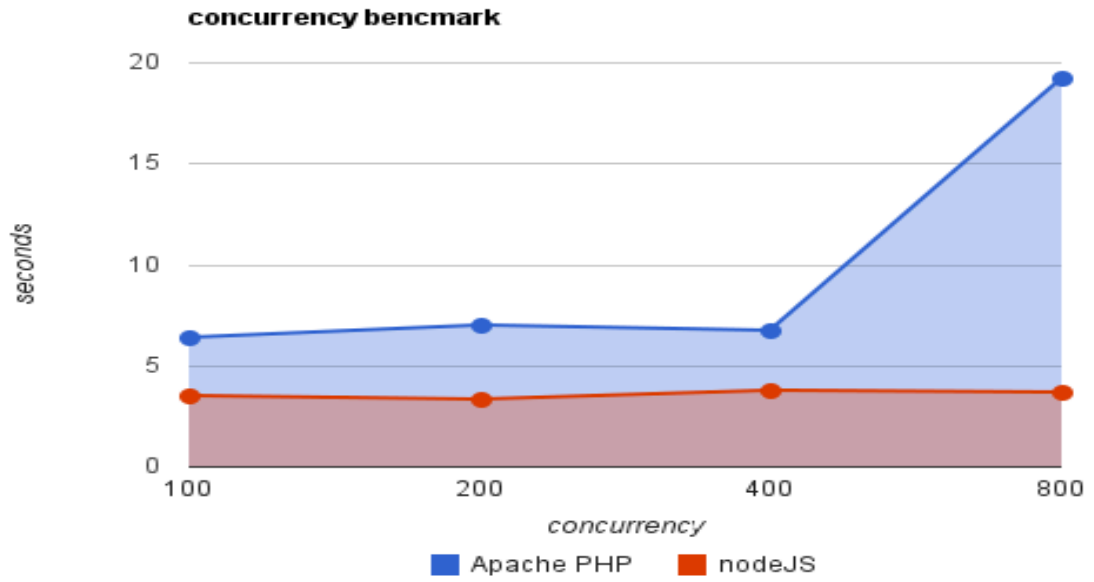
No parallel code execution

Single CPU

Non-Blocking | I/O Model

concurrency bencmark

concurrency	100	200	400	800
Apache PHP	6.337	6.955	6.723	19.232
nodeJS	3.461	3.284	3.721	3.689



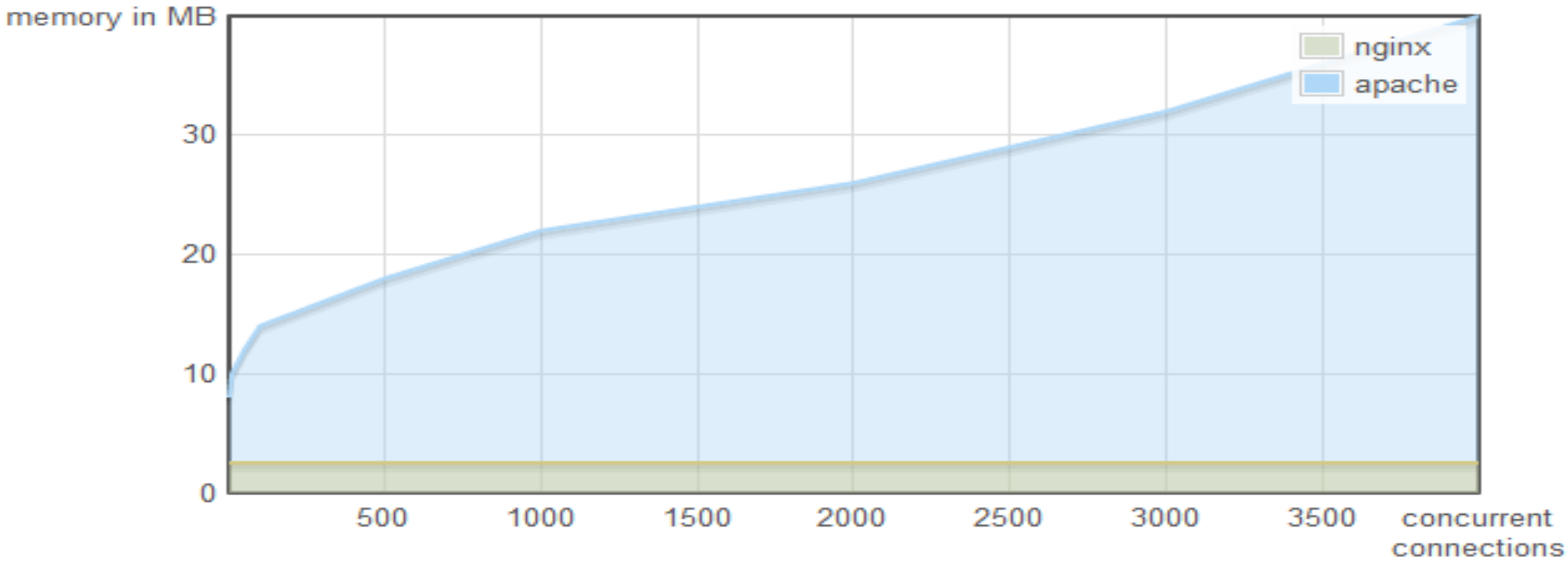
Node is great for applications with high concurrency

(Concurrency = number of concurrent clients or users)

Non-Blocking | I/O Model

nginx: non-blocking I/O

apache: blocking I/O



Callbacks

In a *synchronous* program, you would write something along the lines of:

```
function processData () {
  var data = fetchData ();
  data += 1;
  return data;
}
```

This works just fine and is very typical in other development environments.

However, if *fetchData* takes a long time to load the data, then this causes the whole program to **'block'** - otherwise known as sitting still and waiting - until it loads the data.

Node.js, being an *asynchronous* platform, doesn't wait around for things like file I/O to finish - Node.js uses **callbacks**.

https://docs.nodejitsu.com/articles/getting-started/control-flow/what-are-callbacks

Callbacks

If Google's V8 Engine is the heart of your Node.js application, then callbacks are its veins.

They enable a balanced, non-blocking flow of asynchronous control across modules and applications.

> But for callbacks to work at scale you need a common, reliable protocol.
>
> The "*error-first*" callback (also known as an "errorback", "errback", or "node-style callback") was introduced to solve this problem, and has since become the standard for Node.js callbacks.

A callback is basically a function called at the completion of a given task; this prevents any blocking, and allows other code to be run in the meantime.

https://docs.nodejitsu.com/articles/getting-started/control-flow/what-are-callbacks & http://thenodeway.io/posts/understanding-error-first-callbacks/

Defining an Error-First Callback

There's really only two rules for defining an error-first callback:

The first argument of the callback is reserved for an error object. If an error occurred, it will be returned by the first err argument.

The second argument of the callback is reserved for any successful response data. If no error occurred, err will be set to null and any successful data will be returned in the second argument.

```
fs.readFile('/foo.txt', function(err, data) {
  // TODO: Error Handling Still Needed!
  console.log(data);
});
```

Defining an Error-First Callback

fs.readFile() takes in a file path to read from, and calls your callback once it has finished.

If all goes well, the file contents are returned in the data argument.

But if something goes wrong (the file doesn't exist, permission is denied, etc) the first err argument will be populated with an error object containing information about the problem.

Its up to you, the callback creator, to properly handle this error. You can throw an error if you want your entire application to shutdown. Or if you're in the middle of some asynchronous flow you can propagate that error out to the next callback. The choice depends on both the situation and the desired behavior.

Defining an Error-First Callback

```
fs.readFile('/foo.txt', function(err, data) {
  // If an error occurred, handle it (throw, propagate, etc)
  if(err) {
    console.log('Unknown Error');
    return;
  }
  // Otherwise, log the file contents
  console.log(data);
});
```

Callbacks

The node.js way to deal with the previous example we saw would look a bit more like this:

```
function processData (callback) {
  fetchData(function (err, data) {
    if (err) {
      console.log("An error has occured. Abort everything!");
      callback(err);
    }
    data += 1;
    callback(data);
  });
}
```

https://docs.nodejitsu.com/articles/getting-started/control-flow/what-are-callbacks

Callbacks

At first glance, it may look unnecessarily complicated, but callbacks are *the foundation of Node.js.*

Callbacks give you an interface with which to say, "and when you're done doing that, do all this." This allows you to have as many IO operations as your OS can handle happening at the same time.

For example, in a web server with hundreds or thousands of pending requests with multiple blocking queries, performing the blocking queries asynchronously gives you the ability to be able to continue working and not just sit still and wait until the blocking operations come back.

This is a major improvement.

https://docs.nodejitsu.com/articles/getting-started/control-flow/what-are-callbacks

https://www.quora.com/Whats-the-difference-between-a-promise-and-a-callback-in-Javascript

Callbacks & Promises *

If you've done any serious work in JavaScript, you have probably had to face callbacks, nested inside of callbacks, nested inside of callbacks. This is especially true of code written in node.js, since every form of i/o, such as file reads, database reads and writes is asynchronous, and most code needs more than a single i/o call. You may end up with code that looks something like this:

```
1  function isUserTooYoung(id, callback) {
2      openDatabase(function(db) {
3          getCollection(db, 'users', function(col) {
4              find(col, {'id': id},function(result) {
5                  result.filter(function(user) {
6                      callback(user.age < cutoffAge)
7                  })
8              })
9          })
10     })
11 }
```

Pretty difficult to follow. And it can get much worse. In our current node.js codebase we sometimes do as many as ten sequential, asynchronous calls. That would be a lot of nesting. Thankfully, there's a much better way: *promises*.

https://www.quora.com/Whats-the-difference-between-a-promise-and-a-callback-in-Javascript

Callbacks & Promises *

A promise is a proxy for a value not necessarily known at its creation time. With promises, rather than an asynchronous call accepting a callback, it instead returns a promise. The calling code can then wait until that promise is fulfilled before executing the next step. To do so, the promise has a method named **then**, which accepts a function that will be invoked when the promise has been fulfilled. As an example, the following is the above code rewritten using promises:

When *then* invokes the specified function, that function receives as a parameter the resolved value of the promise. So, for example, when getCollection is called, a handle to the database will be passed to it.

```
1  function isUserTooYoung(id) {
2      return openDatabase()
3          .then(getCollection)
4          .then(find.bind(null, {'id': id}))
5          .then(function(user) {
6              return user.age < cutoffAge;
7          });
8  }
```

Node Modules

THE BUILDING BLOCKS OF NODE

Node.js Ecosystem

Node.js relies heavily on **modules.**

Creating a module is easy, just put your JavaScript code in a separate js file and include it in your code by using the keyword **require**, like:

```
var modulex = require('./modulex');
```

Libraries in Node.js are called packages and they can be installed by typing

```
npm install "package_name"; //installs in current folder
//package should be available in npm registry @ nmpjs.org
```

NPM downloads and installs modules, placing them into a **node_modules** folder in your current folder.

NPM

- Common npm commands:
  - **npm init** initialize a package.json file
  - **npm install <package name> -g** install a package, if –g option is given package will be installed globally, **--save** and **--save-dev** will add package to your dependencies
  - **npm install** install packages listed in package.json
  - **npm ls –g** listed local packages (without –g) or global packages (with –g)
  - **npm update <package name>** update a package

Creating your own Node Modules

donations.js

```
var donations = require('../models/donations');
var express = require('express');
var router = express.Router();

function getByValue(arr, id) {...}

router.findAll = function(req, res) {
    // Return a JSON representation of our list
    res.json(donations);
}

router.findOne = function(req, res) {...}

router.addDonation = function(req, res) {...}

router.deleteDonation = function(req, res) {...}

router.incrementUpvotes = function(req, res) {...}

module.exports = router;
```

```
var routes = require('./routes/index');
var donations = require('./routes/donations');

var app = express();
```

app.js

```
//Our Custom Routes
app.get('/donations', donations.findAll);
app.get('/donations/:id', donations.findOne);
app.post('/donations', donations.addDonation);
app.put('/donations/:id/votes', donations.incrementUpvotes);
app.delete('/donations/:id', donations.deleteDonation);
```

app.js

Defines what 'require' returns

The require search

- Require searches for modules based on path specified:

```
var myMod = require('./myModule'); //current dir
var myMod = require('../myModule'); //parent dir
var myMod = require('../modules/myModule');
```

- Just providing the module name will search in node_modules folder

```
var myMod = require('myModule');
```

Express

A FRAMEWORK FOR NODE

What is Express?

Express is a minimal and flexible framework for writing web applications in Node.js

- Built-in handling of HTTP requests
- You can tell it to 'route' requests for certain URLs to a function you specify
  - Example: When /login is requested, call function handleLogin()
- These functions are given objects that represent the request and the response, not unlike Servlets
- Supports parameter handling, sessions, cookies, JSON parsing, and many other features
- API reference: http://expressjs.com/api.html

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

app.get('/', function(req, res) {
  res.send('hello world');
});

module.exports = app;
```

We'll cover this (and more) in detail in the next Section (Part 2)

Using NodeJS

NodeJS is *just* a JavaScript interpreter.

It comes with a package manager called npm

- Install packages like this: `npm install <package_name>`
- This will install it in the current folder.
- To install globally, do `npm install -g <package_name>`

To use Node as a webserver, you must write an application that responds to web requests.

Node has a library (HTTP) for doing this, but it's easier to use a framework, like *Express*

To access a library, use the `require()` function

Using Express

Express is just a package for Node

- Create a new web application with `var app = express();`
- Respond to requests like `app.get('/user', function(req, res){`
- Look at parameters through the `req` object
  - `req.params` for query parameters
  - `req.body` for post fields
  - `req.files` for files
- Send responses through the `res` object
  - `res.send("Hi mom!")`
- Start the application with
  - `app.listen(<port>)`

Great Resources

Official Tutorial – https://nodejs.org/documentation/tutorials/

Official API – https://nodejs.org/api/

Developer Guide – https://nodejs.org/documentation

Video Tutorials – http://nodetuts.com

Video Introduction – https://www.youtube.com/watch?v=FqMIyTH9wSg

YouTube Channel – https://www.youtube.com/channel/UCvhIsEIBIfWSn_Fod8FuuGg

Articles, explanations, tutorials – https://nodejs.org/community/

Questions?