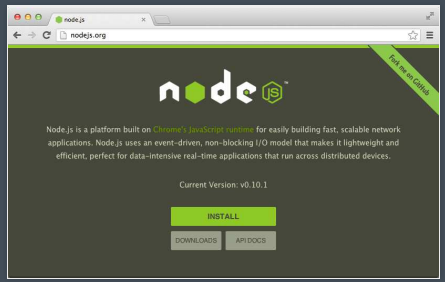
the NODE FIRM

INTRO TO NODE.JS


node.js
nodejs.org
Fork me on GitHub
Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, 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.
Current Version: v0.10.1
INSTALL
DOWNLOADS
API DOCS

Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications.

NODE.JS IS NOT

...a language

...a web framework

WHO'S USING NODE?

IT'S GETTING TO THE POINT WHERE IT'S EASIER TO LIST WHO'S NOT USING NODE


GROUPON
MasterCard
Cloudup
Joyent
ebay
lift
apigee
PayPal
tm
YAHOO!
Walmart
&yet
Telefónica
f5
Microsoft
yammer
Zappos
.com
the web's most popular shoe store!
New Relic.
LinkedIn

WHY DOES NODE EXIST?

MEMORY VS. I/O LATENCY

- L1 cache reference: 1 ns
- L2 cache reference: 4 ns
- Main memory reference: 100 ns

- SSD random-read: 16,000 ns
- Round-trip in same datacenter: 500,000 ns

- Physical disk seek: 4,000,000 ns
- Round-trip from US to EU: 150,000,000 ns

I/O has a much bigger latency than accessing local memory

I/O is expensive

BLOCKING VS. NON-BLOCKING

On typical programming platforms, performing I/O is a blocking operation

```
console.log('Fetching article...');

var result = query("SELECT * FROM articles WHERE id = 1");

console.log('Here is the result:', result);
```

While waiting for I/O, your program is unable to do any other work
This is a **huge** waste of resources!

Conclusion:
Performing I/O is not the same as executing business logic
We should not treat them the same

SOME SOLUTIONS TO SOLVING THIS PROBLEM

PROCESSES

One process per connection
Like Apache Web Server version 1

THREADS

One thread per connection

Like Apache Web Server version 2

EVENT-DRIVEN, NON-BLOCKING PROGRAMMING

Don't block on I/O

```
function handleResult(result) {
  console.log('Here is the result:', result);
}

select('SELECT * FROM articles WHERE id = 1', handleResult);

console.log('Fetching article...');
```

Continue executing code

Listen for an event (result is ready)

When the event is triggered, perform operations on the result

JAVASCRIPT

JavaScript is well suited to the event-driven programming model

FUNCTIONS ARE FIRST-CLASS OBJECTS

- Treat functions like any other object:
  - Pass them as arguments
  - Return them from other functions
  - Store them as variables for later use

FUNCTIONS ARE CLOSURES

Inside a function, the code can access variables on parent scopes

```
function calculateSum (list) {
  var sum = 0;

  function addItem(item) {
    sum += item;
  }

  list.forEach(addItem);

  console.log('sum: ' + sum);
}

calculateSum([ 2, 5, 10, 42, 67, 78, 89, 120 ]);
```

Notice that `addItem` can access and change the value of a variable declared outside its scope

FUNCTIONS ARE CLOSURES

That even works if the outer scope has returned:

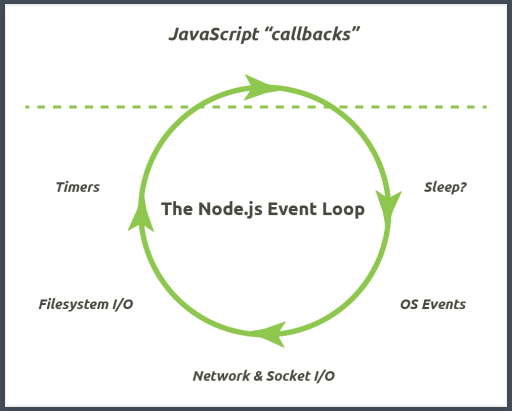
```
function calculateSum (list) {
  var sum = 0;

  function addItem(item) {
    sum += item;
  }

  list.forEach(addItem);

  return function() { // return a function that can print the result
    console.log('sum: ' + sum);
  }
}

var printSum = calculateSum([ 2, 5, 10, 42, 67, 78, 89, 120 ]);

// printSum is a function returned by calculateSum but it still has acce
// scope within its parent function

printSum();
```

A function will capture its containing scope for as long as it we hold a reference to that function

HOW NODE.JS PERFORMS NON-BLOCKING I/O

- Kernel-level non-blocking socket I/O using `epoll` or `select` system calls (depending on the platform)
- Filesystem I/O delegated to worker threads (4 threads by default)
- Glued together with an "event loop"

THE NODE.JS EVENT LOOP



Exit when there are no more pending events

THE NODE.JS EVENT LOOP

Application start-up

```
// startup phase, generally synchronous

var fs = require('fs');
var transformer = require('./transformer.js')
var writer = require('./writer.js')


// execution phase, generally asynchronous

fs.createReadStream('bigdata.txt')
  .pipe(transformer())
  .pipe(writer());
```

- The in-built `require()` method is synchronous but is normally only used on application start-up
- A running Node.js application uses asynchronous I/O for performance

THE NODE.JS EVENT LOOP

When does it stop?

```
// 1.
fs.readFile('bigdata.txt', 'utf8', function (err, data) {
  console.log(data.split('\n').length, 'lines of data');
});

// 2.
setTimeout(function () {
  console.log('Waited 10s');
}, 10000);

// 3.
var i = 0;
var timer = setInterval(function () {
  console.log('tick', i);
  if (++i === 60)
    clearInterval(timer);
}, 1000);
```

THE NODE.JS EVENT LOOP

When does it stop?

```
// 1.
fs.readFile('bigdata.txt', 'utf8', function (err, data) {
  console.log(data.split('\n').length, 'lines of data');
});

// 2.
setTimeout(function () {
  console.log('Waited 10s');
}, 10000);

// 3.
var i = 0;
var timer = setInterval(function () {
  console.log('tick', i);
  if (++i === 60)
    clearInterval(timer);
}, 1000);
```

- **1** will not finish until the event loop has dispatched the data handler callback
- **2** will not finish until the 10s timer has elapsed and the event loop dispatches the timer callback
- **3** will cause the event loop to dispatch the timer callback once every second for 60 seconds and then finish

BLOCKING THE EVENT LOOP

Two things to always keep in mind:

1. Node.js runs JavaScript in a single thread
2. Each callback runs until it returns

COMPLETELY BLOCKING THE EVENT LOOP

```
var cycle = true;
var counter = 0;

function firstCallback() {
  while(cycle) {
    counter ++;
  }
}

setTimeout(firstCallback, 100);

function secondCallback() {
  cycle = false;
}

setTimeout(secondCallback, 200);
```

TEMPORARILY BLOCKING THE EVENT LOOP

Big iterations and complex calculations will temporarily prevent the process from handling other events

```
// some Monte Carlo madness
// estimating π by plotting *many* random points on a plane

var points = 100000000;
var inside = 0;

for (var i = 0; i < points; i++) {
  if (Math.pow(Math.random(), 2) + Math.pow(Math.random(), 2) <= 1)
    inside++;
}

console.log('π ≈', (inside / points) * 4);
```

TEMPORARILY BLOCKING THE EVENT LOOP

Many I/O operations can be performed synchronously, but doing so will occupy the main thread and prevent events from being processed

Synchronous:

```
var bigdata = fs.readFileSync('bigdata', 'utf8');
```

while we wait, **nothing else** can happen in the program

Asynchronous:

```
fs.readFile('bigdata', 'utf8', function (err, bigdata) {
  // ...
});
```

while we wait, the event loop is free to process any other unrelated events

RULES OF THUMB

- Choose asynchronous I/O over synchronous I/O
- Opt for parallel I/O wherever possible
- Don't hog the JavaScript thread with long-running calculations and iterations

NODE.JS PHILOSOPHY

NODE.JS IS MINIMAL

Complexity is in user-land

NODE.JS IS MINIMAL

Instead of providing a big framework, Node provides the minimum viable library for doing I/O

All the rest is built on top of this in user-land

This allows Node core to evolve independently

NODE.JS IS MINIMAL

A language standard library is where modules go to die

More power and flexibility to the programmer: functionality is specialized
Pick and choose from a huge variety existing third-party modules

SUMMARY

- The advantages of non-blocking I/O
- JavaScript is a great fit for event-driven programming
- Don't block the event loop
- Node.js is minimal. User-land is rich and diverse