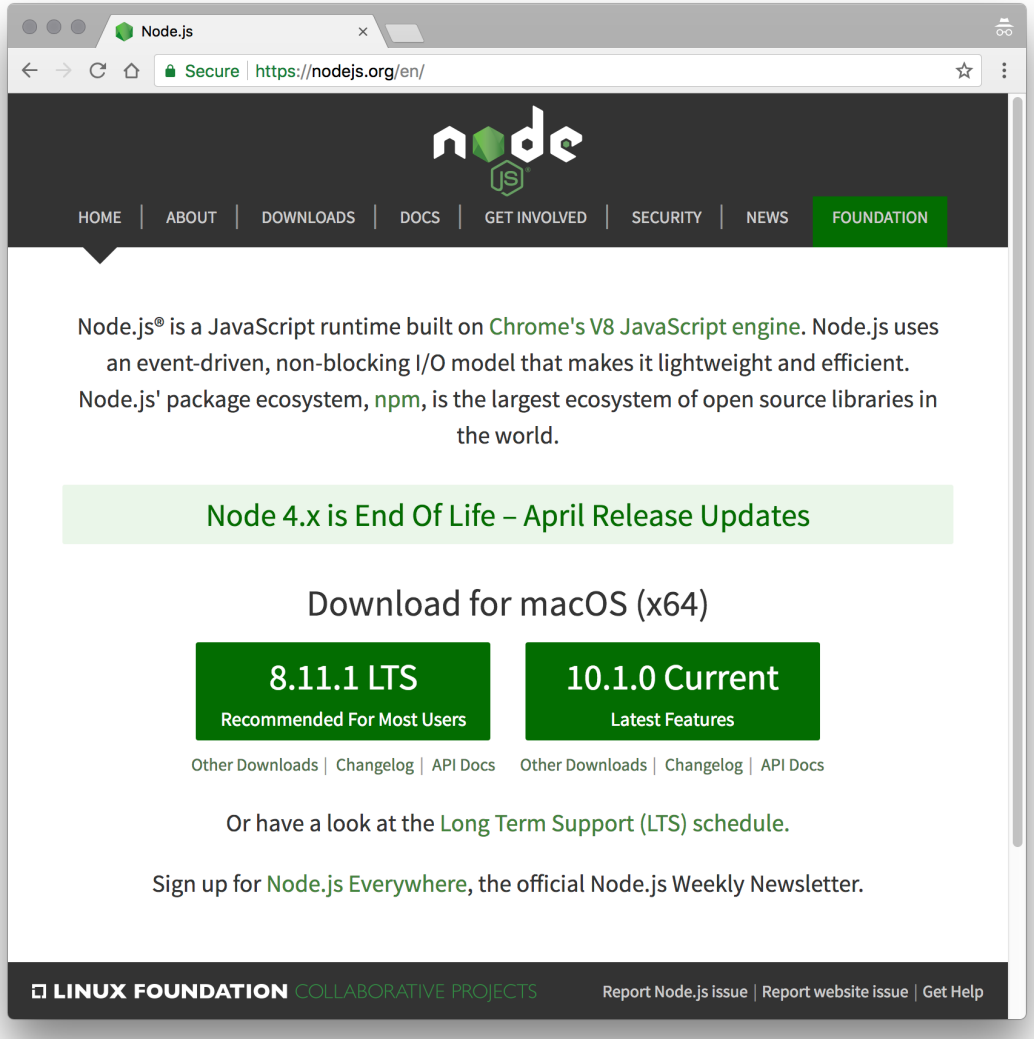
# Node Fundamentals - NodeSummit 2018

# Introduction to Node.js

# Learning Objectives

- Learn what Node.js *is* and *is not*
- Understand the costs of I/O and the advantages of non-blocking I/O via event-driven programming
- Understand how JavaScript is a good fit for event-driven programming
- Understand the Node.js event loop and how to avoid blocking it
- Understand that Node.js is minimal and that its user-land is rich and diverse



# 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 is Using Node.js?

# Who is Using Node.js?

*Who isn't using Node.js?*

# Why Does Node.js Exist?

# Latency: Memory vs I/O

| Class | Operation | Time cost |
|---|---|---|
| Memory | L1 cache reference: | 1 ns |
| | L2 cache reference: | 4 ns |
| | Main memory reference: | 100 ns |
| I/O | 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 is Expensive

# Blocking vs. Non-Blocking

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

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

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

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

# Waiting for I/O is a Waste

While waiting for I/O, your program is unable to do any other work

This is a **huge** waste of resources!

# I/O is usually hidden

On typical programming platforms, I/O operations are obscured amongst non-I/O operations

```
// Java: Spot the I/O

System.out.println("Reading file...");
BufferedReader br = new BufferedReader(new FileReader("in.txt"));

try {
  StringBuilder sb = new StringBuilder();
  String line;

  while ((line = br.readLine()) != null)
    sb.append(line + "\n");
  System.out.print(sb.toString());
} finally {
  br.close();
}

System.out.println("Finished reading file!");
```

# Treat I/O as a Special Case

Performing I/O is not the same as executing business logic

We should not treat them the same

# Some Solutions

# 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

# Why 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) {
  let 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:

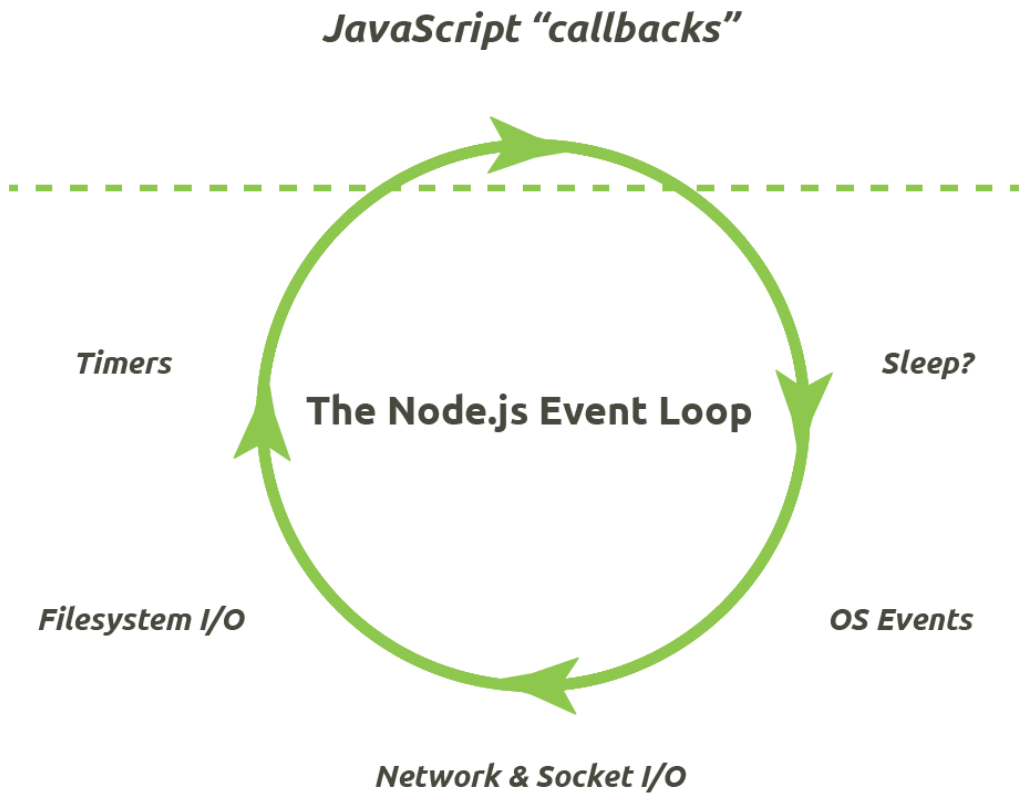
```
function calculateSum (list) {
  let sum = 0

  function addItem (item) {
    sum += item
  }

  list.forEach(addItem)

  return function () { // return a function that can print the result
    console.log(`sum: ${sum}`)
  }
}

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

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

printSum()
```

A function will capture its containing scope for as long as 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)
- File system I/O delegated to worker threads (4 threads by default)
- Glued together with an "event loop"

# The Node.js Event Loop



# The Node.js Event Loop

## Application start-up

```
// startup phase, generally synchronous

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


// execution phase, generally asynchronous

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

- The built-in `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 onReadFile (err, data) {
  console.log(data.split('\n').length, 'lines of data');
})
```

```
// 2.
setTimeout(() => console.log('Waited 10s'), 10000);
```

```
// 3.
let i = 0;
const timer = setInterval(() => {
  console.log(`tick ${i}`)
  if (++i === 60)
    clearInterval(timer)
}, 1000)
```

# Blocking the Event Loop

## Two Things to Always Keep in Mind

1. Node.js runs JavaScript in a single thread.
2. Each JavaScript function runs until completion.

# Everything is Asynchronous

## Except for *Your Code*

# Completely Blocking the Event Loop

It's possible...

```
let cycle = true
let 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

let points = 100000000
let inside = 0

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

// the event loop is blocked until the calculations are done

console.log(`π ≈ ${(inside / points) * 4}`)
```

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

# Sync vs. Async

## Synchronous

While we wait, **nothing else** can happen in the program.

```
const bigdata = fs.readFileSync('bigdata', 'utf8')
```

# Sync vs. Async

## Synchronous

While we wait, **nothing else** can happen in the program.

```
const bigdata = fs.readFileSync('bigdata', 'utf8')
```

## Asynchronous

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

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

# 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 Philosophy

## Node.js is Minimal

Complexity is in user-land

# Node.js Philosophy

## Node.js is Minimal

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

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

This allows Node.js core to evolve independently

# Node.js Philosophy

## Node.js is Minimal

> *A language standard library is where modules go to die*

More power and flexibility to the programmer

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

# The Node.js Async Model

# Learning Objectives

- Discuss what "Asynchronous" means in Node.js
- Why did Node.js choose the async model?
- Walk through some asynchronous Node.js code

# libuv and V8

libuv orchestrates I/O and timers

- File system
- Network
- Timers
- DNS

V8 is the JavaScript engine that executes your code

# libuv and V8

- libuv uses a threadpool to coordinate work but not *do* work
- V8 employs a single thread to do all computations and work with data
- Running in a single thread allows V8 to provide isolation and prevent the need for thread locks

# libuv and V8

- Node.js tracks `Handles`—JavaScripts functions plus context—and attaches them to events
- `libuv` coordinates the events and triggers the `Handles` to be executed in V8 when ready
- When there are no more `Handles` left, there is nothing left to execute in V8, and Node.js will exit

# Why Async?

Most modern applications spend most of their time waiting for I/O

- Reading files
- HTTP APIs
- Database queries

CPUs are orders of magnitude faster than I/O

**Web applications are usually just routing data from A to B**

# Why Evented

For typical web application workloads, the Evented model is more efficient than thread pools:

- Reduces thread CPU overhead
- Reduces thread lock contention
- Better work scheduling

# Why Evented

With an Evented Asynchronous model:

- Waiting is basically free
- Waiting is done in parallel
- Blocked time is reserved for things that can't be done in parallel

# Node.js Async In Action

**timer.js** is a trivial Node.js application that does a single async task based on a timer

```
'use strict'

const later = function later () {
  console.log('Hello (a little later)')
}

console.log('Hello!')
setTimeout(later, 100)
```

*Let's walk through what happens under the hood...*

# timer.js Walkthrough

```
/*┌──────────────┬──────────────┐*/
/*|    libuv     |      V8      |*/
/*├──────────────┼──────────────┤*/
/*|   STARTUP    |   STARTUP    |*/
/*|     [ ]      |    <busy>    |*/  const later = /* ... */
/*|     [ ]      |    <busy>    |*/  console.log("Hello!")
/*|  [timer] <-+- <schedule>  |*/  setTimeout(later, 100)
/*|  [timer]     |               |*/
/*|  [timer]     |               |*/
/* ····· ~100ms later ····· */
/*|  [timer]     |               |*/
/*|  [timer]  -+-> <resume>   |*/  function later () {
/*|     [ ]      |    <busy>    |*/    console.log("Hello (a little later)")
/*|     [ ]      |    <busy>    |*/  }
/*|     [ ]      |               |*/
/*└──────────────┴──────────────┘*/

/* No Handles and V8 is done—application exits */
```

# Running JavaScript is Blocking

Any pending JavaScript event handling must wait for prior JavaScript work to finish with V8 before it can do its work

This means long-running JavaScript will delay pending work

**blocked.js:**

```
'use strict'

const start = Date.now()

setTimeout(function A () {
  let count = 0
  while (++count < 1e9) { Math.random() }
}, 100)

setTimeout(function B () {
  console.log('elapsed: %sms', Date.now() - start)
}, 105)
```

# blocked.js Walkthrough

```
/*┌──────────────────────────┬──────────────┐*/
/*|    libuv                 |      V8      |*/
/*├──────────────────────────┼──────────────┤*/
/*|   STARTUP                |   STARTUP    |*/
/*|   [ ]                    |   <busy>     |*/  const start = Date.now()
/*|   [timer:A]          <-+- <schedule>   |*/  setTimeout(A, 100)
/*|   [timer:A, timer:B] <-+- <schedule>   |*/  setTimeout(B, 105)
/*|   [timer:A, timer:B]     |              |*/
/* ............. ~100ms later ............. */
/*|   [timer:A, timer:B]  -+-> <resume A> |*/  function A () {
/*|   [timer:B]              |   <busy>     |*/    let count = 0
/*|   [timer:B]              |   <busy>     |*/    while (++count < 1e9)
/*|   [timer:B]              |   <busy>     |*/      Math.random()
/*|   [timer:B]              |   <busy>     |*/      Math.random() // looping...
/*|   [timer:B]              |   <busy>     |*/      Math.random()
/* .......... ~105ms from start .......... */  // would have resumed B
/*|   [timer:B]              |   <busy>     |*/      Math.random()
/*|   [timer:B]              |   <busy>     |*/      Math.random()
```

# blocked.js Walkthrough (cont.)

```
/* ····· ~1e9 million randoms later ····· */
/*|   [timer:B]              |   <busy>     |*/      Math.random()
/*|   [timer:B]              |   <busy>     |*/  }
/*|   [timer:B]              |              |*/
/*|   [timer:B]            -+-> <resume B> |*/
/*|   [ ]                    |   <busy>     |*/  function B () {
/*|   [ ]                    |   <busy>     |*/    console.log("...")
/*|   [ ]                    |   <busy>     |*/  }
/*|   [ ]                    |              |*/
/*└──────────────────────────┴──────────────┘*/

/* No Handles and V8 is done—application exits */
```

# Avoid Blocking Operations

This is why we avoid using **Synchronous** I/O mechanisms

A synchronous, blocking I/O wait means other events are blocked from V8 even though, in reality, it is just waiting

# JavaScript is Blocking Part 2

In the worst-case scenario, we will **never** resume waiting events because we never leave V8

V8 does not yield until it is finished with the JavaScript it is currently executing

**unreachable.js:**

```
'use strict'

const later = () => {
  console.log('Hello?')
}

// Schedule `later` for 10 milliseconds from start
setTimeout(later, 10)

// Calculate random numbers until the end of time
while (true) {
  Math.random()
}
```

# unreachable.js Walkthrough

The timer is blocked from accessing V8 because V8 is busy, and V8 will be busy forever. The timer never resumes and the application never finishes

```
/*┌──────────────────────┬──────────────┐*/
/*|    libuv             |      V8      |*/
/*├──────────────────────┼──────────────┤*/
/*|   STARTUP            |   STARTUP    |*/
/*|   [ ]                |   <busy>     |*/  const later = /* ... */
/*|   [timer]         <-+- <schedule>   |*/  setTimeout(later, 10)
/*|   [timer]            |              |*/  while (true) {
/*|   [timer]            |   <busy>     |*/    Math.random()
/*|   [timer]            |   <busy>     |*/    Math.random() // looping...
/*|   [timer]            |   <busy>     |*/    Math.random()
/* ......... ~10ms from start .......... */  // would have resumed `later`
/*|   [timer]            |   <busy>     |*/    Math.random()
/*|   [timer]            |   <busy>     |*/    Math.random() // and so on...
/* ...... later is never triggered ..... */
```

# A "Real" Example

**server.js** is a simple HTTP server in Node.js that serves up itself whenever it is read

```
'use strict'

const http = require('http')
const fs = require('fs')

const server = http.createServer(function handler (req, res) {
  fs.readFile(__filename, function callback (err, content) {
    if (err) {
      res.statusCode = 500
      return res.end(err.message)
    }
    res.end(content)
  })
})

server.listen(8000)
```

# server.js Walkthrough

```
/*┌──────────────────┬──────────────────┐*/
/*|     libuv        |       V8         |*/
/*├──────────────────┼──────────────────┤*/
/*|    STARTUP       |    STARTUP       |*/
/*|    [ ]           |    <busy>        |*/  const http = require("http")
/*|    [ ]           |    <busy>        |*/  const http = require("fs")
/*|    [ ]           |    <busy>        |*/  const server = http.createServer(/* ... */)
/*|    [socket]    <-+- <register>     |*/  server.listen(8000)
/*|    [socket]      |                  |*/
/*|    [socket]      |                  |*/
/* · socket is now listening/waiting · */
```

```
/* ............ a client connects! ............. */
/*|  [socket, client]     -+> <resume handler>  |*/ function handler (req, res) {
/*|  [socket, client, fs] <+- <register>        |*/   fs.readFile(/* ... */)
/*|  [socket, client, fs]  |  <busy>            |*/ }
/*|  [socket, client, fs]  |                    |*/
/*|  [socket, client, fs]  |                    |*/
/*|  [socket, client, fs]  |                    |*/
/* ............. reading the file ............. */
/*|  [socket, client, fs]  |                    |*/
/*|  [socket, client, fs]  |                    |*/
/*|  [socket, client]     -+> <resume callback> |*/ function callback (err, content) {
/*|  [socket, client]      |  <busy>            |*/   if (err) { /* branch */ }
/*|  [socket, client]     <+- <write>           |*/   res.end(content)
/*|  [socket, client]      |  <busy>            |*/ }
/*|  [socket, client]      |                    |*/
/* ··· client socket is done, gets torn down ··· */
/* ··· server socket still listening/waiting ··· */
```

# Summary

- Node.js use the Evented model
- The Evented I/O model allows:
  - waiting to be done in parallel (non blocking)
  - safe coordination of blocking tasks
  - efficient use of resources
- libuv orchestrates I/O and timers
- When there are no more async `Handles` left, your program is done
- V8 can only do one thing at a time
- Spend as little time in V8 as possible

# First Steps

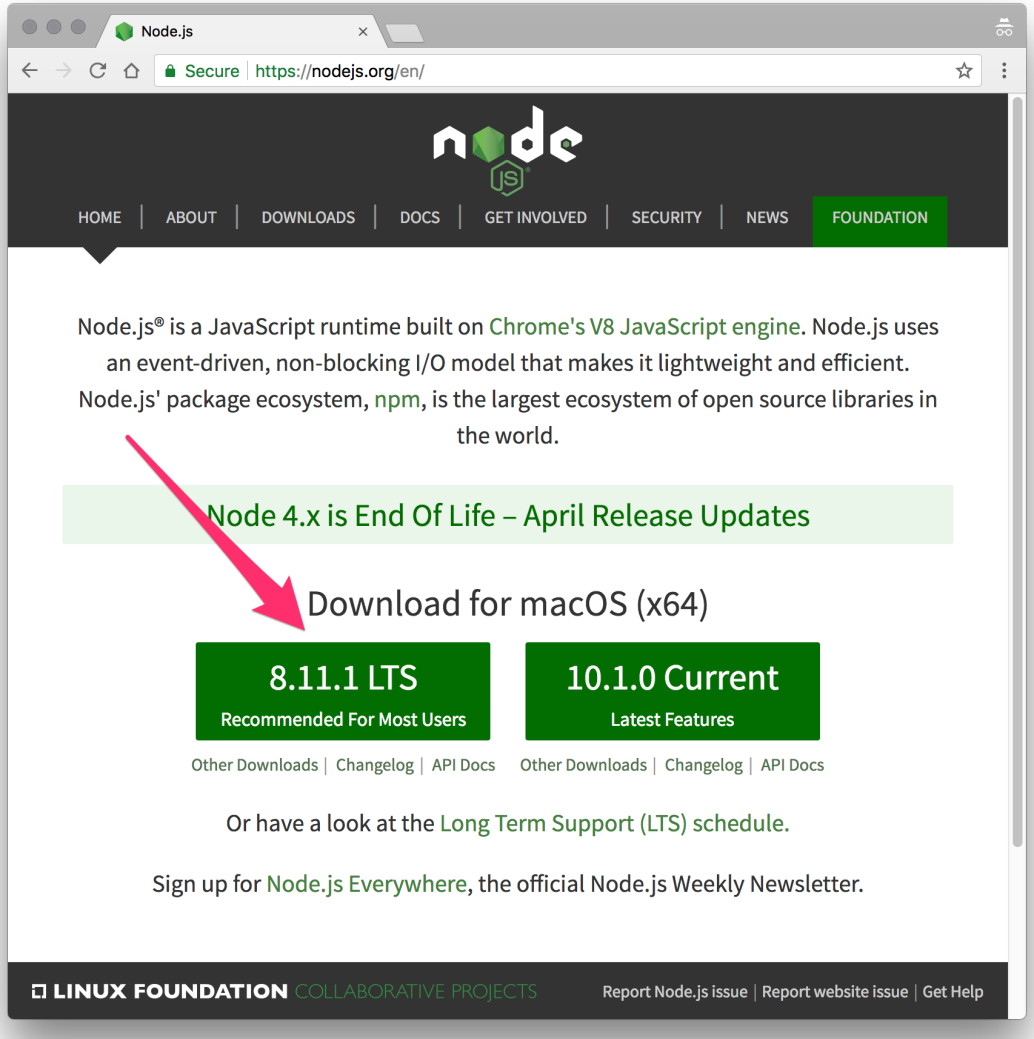
# Learning Objectives

This unit is designed to give a first taste of Node.js

- Learn how to install Node.js
- Learn how to manage installs with `nvm`
- Learn how to use the Node.js REPL
- Learn how to execute `*.js` files with node
- Build and instantiate a simple HTTP server
- Discover Node.js references and learning resources

# Download and Install

Install Node.js by going to https://nodejs.org and downloading the latest LTS version:



# Sanity Check

The `node` command should be available (cmd.exe/xterm/terminal):

```
$ node -v
v8.11.1
```

# Install Node on Linux

NodeSource offers Node.js Linux Binary Distributions https://github.com/nodesource/distributions

## Debian and Ubuntu based distributions

```
# Using Ubuntu
$ curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -
$ sudo apt-get install -y nodejs

# Using Debian, as root
$ curl -sL https://deb.nodesource.com/setup_8.x | bash -
$ apt-get install -y nodejs
```

## Enterprise Linux based distributions

```
$ curl -sL https://rpm.nodesource.com/setup_8.x | bash -
$ sudo yum -y install nodejs
$ sudo yum install gcc-c++ make # Optional
```

# Manage Installs with `nvm`

`nvm` stands for Node Version Manager and can be used on macOS and Linux.
https://github.com/creationix/nvm

To install `nvm` in your system you can run:

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.11/install.sh | bash
```

To install a Node.js version

```
$ nvm install 8
Now using node v8.11.1 (npm v5.6.0)
```

To switch between versions

```
$ nvm use 6
Now using node v6.12.3 (npm v5.6.0)
```

*There are alternatives for Windows like* `nvmw` *and* `nvm-windows`

# Node REPL

Calling `node` with no arguments starts the Node.js REPL (Read-Eval-Print Loop):

```
$ node
>
```

# Node REPL

You can type JavaScript directly into the REPL and it will execute it and print the returned value:

```
$ node
> 3
3
> true
true
> 'Hey'
'Hey'
```

# Node REPL

You can type JavaScript directly into the REPL and it will execute it and print the returned value:

```
> console.log('Hey!')
Hey!
undefined
```

# Executing JavaScript Files

### 01_hello_world.js:

```
'use strict'

console.log('hello world!')
```

Execute JavaScript files from the command-line (cmd.exe/xterm/terminal)

```
$ node 01_hello_world.js
hello world!
$
```

# Hello Function World

## 02_hello_world_function.js:

```
'use strict'

function helloWorld () {
  console.log('hello world!')
}

function messenger (printMessageFunction) {
  console.log('delivering message')
  printMessageFunction()
}

helloWorld() // says 'hello world' right away

messenger(helloWorld) // says 'hello world' via a messenger

setTimeout(helloWorld, 1000) // aynchronous - say 'hello world' later

console.log('waiting for async hello world')
```

```
hello world!
delivering message
hello world!
```

```
waiting for async hello world
hello world!
```

# Hello HTTP World

**03_hello_world_server.js:** our first HTTP server

```
'use strict'

const http = require('http')
const port = parseInt(process.argv[2], 10) || 8080
const server = http.createServer()

function handleRequest (req, res) {
  res.writeHead(200, { 'content-type': 'text/plain' })
  res.write('Hello ')
  res.write('World!')
  res.end()
}

server.on('request', handleRequest)

server.listen(port)
console.log('Listening on port %d', port)
```

# Hello HTTP World

**04_hello_world_server.js:** compact version

```
'use strict'

const port = parseInt(process.argv[2], 10) || 8080

require('http').createServer((req, res) => {
  res.writeHead(200, {'content-type': 'text/plain'})
  res.end('Hello World!')
}).listen(port)

console.log('Listening on port %d', port)
```

- Avoids creating single use temporary variables (`http`, `server`)
- Uses the `http.createServer()` request handler
- Responds in one call (`res.end(...)`)

# References

## Code Repository

https://github.com/nodejs/node

# References

## API Docs

https://nodejs.org/api/

# References

## Dash *(macOS Only)*

Offline documentation app

https://kapeli.com/dash

# References

## Node.js Help

Official Node.js help repository

https://github.com/nodejs/help

# References

## Mailing List

https://groups.google.com/group/nodejs

# References

## IRC Channel

Hostname: irc.freenode.net

Channel: #Node.js

# References

## Gitter Room

https://gitter.im/nodejs/node

# References

## Node Slack Community

http://www.nodeslackers.com/

# Modules and Packages

# Learning Objectives

- Learn how to use `require()` to load modules
- Understand core and local modules
- Learn how to create local modules
- Understand the basics of npm
- Learn how to install packages with npm
- Learn how to create packages

# Require

- `require()` is a global function in Node.js
- Loads external modules or packages
- Similar to `import` or `include` statements from other platforms
- **Requiring a module does not modify the environment**

```
const util = require('util')
```

# Core Modules

Node.js comes with several core modules, such as: `'http'`, `'net'`, `'assert'`, `'dgram'`, `'path'`, `'url'`, `'util'` and others

To use a core module pass its name to the `require()` function:

```
const http = require('http')
const fs = require('fs')
```

# Local Modules

`require()` can be called with a file path

Paths can be absolute or relative to the current file:

```
const circle = require('./lib/shapes/circle.js')
const myUtil = require('../util.js')
```

# Anatomy of a Module

A module is a file

A module is a JavaScript file

A module is a JavaScript file that exports a value

# Anatomy of a Module

- Calling `require()` executes the body of a module synchronously
- The value of `module.exports` is the value returned when requiring the module
- A module can export *any* JavaScript value

```
module.exports = function getPI () {
  return Math.PI;
}
```

# Anatomy of a Module

An approximation of how Node.js loads a module internally:

```
// after it determined what module, require and filename are
const dirname = path.dirname(filename)

module.exports = {} // start with an empty exports object

function initModule (exports, require, module, __filename, __dirname) {

  // contents of our module
  module.exports = function getPI () {
    return Math.PI
  };

}

initModule(module.exports, require, module, filename, dirname)

return module.exports
```

- `__filename` - path to the current file
- `__dirname` - path to the current file's directory

# Anatomy of a Module

A module should be a **unit of logic** such as:

```
'use strict'

const pi = Math.PI

function diameter (radius) {
  return 2 * radius
}

function circumference (radius) {
  return pi * diameter(radius)
}

function area (radius) {
  return pi * Math.pow(radius, 2)
}
```

# Anatomy of a Module

... and can have private code

**01_circle.js:**

```
'use strict'

const pi = Math.PI

module.exports = function createCircle (radius) {
  function diameter () {
    return 2 * radius
  }

  function circumference () {
    return pi * diameter()
  }

  function area () {
    return pi * Math.pow(radius, 2)
  }

  return {
    area,
    circumference
```

# Using Modules

# From the REPL

```
$ node
> _
```

```
> const createCircle = require('./01_circle.js');
undefined
> const circle = createCircle(10);
undefined
> circle;
{ area: [Function: area],
  circumference: [Function: circumference] }
> circle.area();
314.1592653589793
> circle.circumference();
62.83185307179586
```

You can also omit the `.js`:

```
> const createCircle = require('./01_circle');
```

# Module Caching

## The module body is executed exactly once

```
$ node
> const request = require('request');
> request.hello = 'world';
> console.log(require('request').hello);
world
undefined
```

Subsequent requires to the same module return exactly the same value

# Command Line npm

A Unix-like tool

```
$ npm search request
$ npm doc request
$ npm install request
$ npm remove request
```

# npm

npm is your gateway to the Node.js ecosystem

# Installing Packages from npm

npm is bundled with Node.js

Example: *install the "request" module*

```
$ npm install request
```

# Installing Packages from npm

```
$ npm ls
/Users/user/project
└─┬ request@2.85.0
  ├── aws-sign2@0.7.0
  ├── aws4@1.7.0
  ├── caseless@0.12.0
  ├─┬ combined-stream@1.0.6
  │ └── delayed-stream@1.0.0
  ├── extend@3.0.1
  ├── forever-agent@0.6.1
  ...
  ├─┬ tough-cookie@2.3.4
  │ └── punycode@1.4.1
  ├─┬ tunnel-agent@0.6.0
  │ └── safe-buffer@5.1.1 deduped
  └── uuid@3.2.1
```

# Installing Packages from npm

npm installs packages into a *node_modules* folder in the current directory:

```
$ ls node_modules
ajv                     dashdash                hoek                    punycod
asn1                    delayed-stream          http-signature          qs
assert-plus             ecc-jsbn                is-typedarray           request
...
co                      getpass                 mime-db                 tweetna
combined-stream         har-schema              mime-types              uuid
core-util-is            har-validator           oauth-sign              verror
cryptiles               hawk                    performance-now
```

# Installing Packages from npm

Back in the REPL:

```
$ node
> const request = require('request');
```

Module search paths for module names (not a file path)

1. core module
2. `node_modules` folder in current directory
3. `node_modules` folder in parent directories

# Creating Our Own Packages

## package.json - the package manifest

### package.json:

```
{
  "name": "modules-src",
  "version": "0.0.1",
  "description": "Modules unit source code",
  "dependencies": {
    "request": "*",
    "async": "*"
  }
}
```

# Creating Our Own Packages

Now, you can tell npm to resolve those dependencies locally:

```
$ npm install
```

this will ensure all the dependencies in your *package.json* are installed

```
$ node
> const async = require('async');
```

# Summary

- Modules separate program logic into discrete, manageable units
- Packages groups modules into distributable bundles
- Modules can be loaded from Node.js core, packages or locally
- npm is the gateway to the Node.js ecosystem

# Callbacks

# Learning Objectives

- Understand what callbacks are and are used for
- Understand how callbacks are used for flow-control

# What Are Callbacks?

Regular JavaScript functions that are called once a unit of work is complete

```
function timerCallback () {
  console.log('tick')
}

setTimeout(timerCallback, 1000)
```

# Node.js Callbacks

```
someAsyncFunction('arg1', 2, function (err, result) {
  // handle error
  // if no error, process result
})
```

- Callback function supplied as the last argument
- Error object supplied as the first argument to the callback

# Node.js Callbacks

Naming and extracting callbacks is good practice:

```
function asyncCallback (err, result) {
  // handle error
  // if no error, process result
}

someAsyncFunction('arg1', 2, asyncCallback)
```

# Read a File

## 01_read_file.js

```
'use strict'

const fs = require('fs')

function onReadFile (err, contents) {
  if (err) {
    console.error('File read error:', err)
  } else {
    console.log(contents)
  }
}

fs.readFile(__filename, 'utf8', onReadFile)
```

# Resolve a Hostname

## 02_dns_lookup.js

```
'use strict'

const dns = require('dns')
const hostname = process.argv[2] || 'google.com'

dns.lookup(hostname, function onDnsLookup (err, address, family) {
  if (err) {
    console.error('error looking up', err.message)
  } else {
    console.log('%s resolved to %s (IPv%s)', hostname, address, family)
  }
})
```

# Flow-Control

## Asynchronous functions can be run in serial or in parallel

# Serial Execution

Perform I/O operations, each depending on the previous:

## 03_serial.js

```
'use strict'

const fs = require('fs')
const maxFileSize = parseInt(process.argv[2], 10) || 400

fs.stat(__filename, function onStat (err, stats) {
  if (err) return console.error(err)

  if (stats.size < maxFileSize) {
    fs.readFile(__filename, 'utf8', function onReadFile (err, contents) {
      if (err) return console.error(err)

      console.log(contents)
    })
  } else {
    console.log('Not reading the file, it is too big')
  }
})
```

# Serial Execution

Run with:

```
$ node 03_serial.js 1000
```

and

```
$ node 03_serial.js 100
```

# Serial Execution: Flattened

## 04_serial_flat.js

```
'use strict'

const fs = require('fs')
const maxFileSize = parseInt(process.argv[2], 10) || 400

function onStat (err, stats) {
  if (err) return console.error(err)

  if (stats.size < maxFileSize) {
    fs.readFile(__filename, 'utf8', onReadFile)
  } else {
    console.log('Not reading the file, it is too big')
  }
}

function onReadFile (err, contents) {
  if (err) return console.error(err)

  console.log(contents)
}
```

Minimizes the indentation level

# Parallel Execution

Perform multiple I/O operations, at the same time:

## 05_parallel.js

```
'use strict'

const http = require('http')
const zipCodes = [ '90210', '94107', '10007', '33109' ]
let count = 0
let endpoint = 'http://api.openweathermap.org/data/2.5/weather'
let apiKey = '91ee1b726e850a52750c87205ec0e2a7'

zipCodes.forEach(zip => {
  const url = `${endpoint}?zip=${zip},us&APPID=${apiKey}&units=imperial`
  http.get(url, res => {
    let responseBody = ''

    // Collect response body
    res.on('data', d => {
      responseBody += d.toString()
    })

    res.once('end', () => {
      const obj = JSON.parse(responseBody.replace('///', ''))
```

# Summary

- A callback is a JavaScript function
- Callbacks are only called once (in general)
- A callback is the last argument in an async call
- Error object is the first callback argument

# Understanding Promises

# Learning Objectives

- Understand what Promises are
- Define terminology
- Chaining Promises
- Error Handling
- Anti Patterns with Promises
- Flow Control with Promises
- Understand async / await

# Promise Concepts

- Promises are for managing state
- Used for async operations

# Promise States

- **fullfilled**: The action relating to the promise succeeded
- **rejected**: The action relating to the promise failed
- **pending**: Hasn't fulfilled or rejected yet
- **settled**: Has been fufilled or rejected

# Promises have been around for a while

- Q
- when
- WinJS
- RSVP.js

Not a new concept...

- jQuery had the Deferred object which was similar but not Promise/A+ compliant.
- Node.js had them prior to 0.2
- Other language may refer to them as futurables

# Creating a Promise

```
new Promise((resolve, reject) => {
    // do a thing, possibly async, then....
    if ( /* everything is fine */ ) {
        resolve("success")
    } else {
        reject(new Error("broken"))
    }
})
```

# Using Promises

```
promise.then(result => {
  console.log(result) // "success"
}, err => {
  console.log(err) // Error: "broken"
})
```

# Then

As we saw `then()` takes two arguments, a callback for a success case and another for the failure case.

## You can also use `catch()`

```
promise.then(result => {
  console.log(result) // "success"
}).catch(err => {
  console.log(err) // Error: "broken"
})
```

# Catch

There is nothing special about `catch()`, it's just sugar for `then(undefined fn)`, but it's more readable.

Promise rejections skip forward to the next `then()` with a rejection callback (or `catch()`).

With `then(fn1, fn2)`, `fn1` or `fn2` will be called, never both. But with `then(fn1).catch(fn2)`, both will be called if `fn1` rejects, as they're separate steps in the chain.

# Exceptions

```
const jsonPromise = new Promise((resolve, reject) => {
  // JSON.parse throws an error if you feed it some
  // invalid JSON, so this implicitly rejects:
  resolve(JSON.parse("This ain't JSON"))
});

jsonPromise.then(data => {
  // This never happens:
  console.log("It worked!", data)
}).catch(err => {
  // Instead, this happens:
  console.log("It failed!", err)
})
```

THe same goes for errors thrown in `then()`

# Error handling in practice

```
getJSON('story.json')
  .then(story => getJSON(story.chapterUrls[0]))
  .then(chapter1 => addHtmlToPage(chapter1.html))
  .catch(err => addTextToPage("Failed to show chapter"))
  .then(() => {
    document.querySelector('.spinner').style.display = 'none'
  })
```

# Log Unhandled Rejected Promises

## In the Browser

```
window.addEventListener('unhandledrejection', event => {
  // Can prevent error output on the console:
  event.preventDefault()

  // Send error to log server
  log('Reason: ' + event.reason)
});
```

### In Node.js

```
process.on('unhandledRejection', reason => {
  console.log('Reason: ' + reason)
})
```

# Anti Pattern

```
function getHotDog () {
    return new Promise((resolve, reject) => {
        getBun().then(bun => {
            addSausage(bun).then(bunWithSausage => {
                addSauce(bunWithSausage).then(hotdog => {
                    resolve(hotdog)
                })
            })
        })
    })
}
```

# Refactoring

All the underlying function calls return Promises, so there is no need to create `new Promise` but rather return directly:

```
function getHotDog () {
    return getBun().then(bun => {
        return addSausage(bun).then(bunWithSausage => {
            return addSauce(bunWithSausage).then(hotdog => {
                return hotdog
            })
        })
    })
}
```

# Refactoring

Because we can pass the output of one function to another, we can chain them.

```
function getHotDog () {
    return getBun()
      .then(bun => addSausage(bun))
      .then(bunWithSausage =>addSauce(bunWithSausage))
      .then(hotdog => hotdog)
}
```

# Refactoring

Since the functions are simple we can compact it further

```
function getHotDog () {
    return getBun()
    .then(addSausage)
    .then(addSauce)
}
```

# Avoid Side Effects

```
firstAsyncTask().then(result => {
    secondAsyncTask(result)
}).then(() => {
    thirdAsyncTask()v
})
```

# Avoid Side Effects

The `secondAsyncTask()` Promise is very likely not resolved by the time `.then()` handler executes. Always return one of the following from `then` handlers.

- a new Promise - which would solve our case above
- a synchronous value or `undefined`
- or, throw an Error

# Executing Promises in Series

```
// Promise returning functions to execute
function doFirstThing () { return Promise.resolve(1) }
function doSecondThing (res) { return Promise.resolve(res + 1) }
function doThirdThing (res) { return Promise.resolve(res + 2) }
function lastThing (res) { console.log("result:", res) }

const fnlist = [ doFirstThing, doSecondThing, doThirdThing, lastThing ]

// Execute a list of Promise return functions in series
function pseries(list) {
  const p = Promise.resolve()
  return list.reduce((pacc, fn) => {
    return pacc = pacc.then(fn)
  }, p)
}

pseries(fnlist)
// result: 4
```

# Executing in Parallel

```
Promise.all([ firstTask, secondTask, thirdTask ])
  .then(results => { // array of results
  })
  .catch(err => {

  })
```

# Promise Race

```
var promise1 = new Promise(function(resolve, reject) {
    setTimeout(resolve, 500, 'one');
});

var promise2 = new Promise(function(resolve, reject) {
    setTimeout(resolve, 100, 'two');
});

Promise.race([promise1, promise2]).then(function(value) {
  console.log(value);
  // Both resolve, but promise2 is faster
});
// expected output: "two"
```

# Promisify

- Part of Node.js carbon (v8)

```
const util = require('util');
const fs = require('fs');

const stat = util.promisify(fs.stat)
stat('.')
  .then(stats => {
  // Do something with `stats`
  })
  .catch(error => {
  // Handle the error.
  })
```

# async / await

- Yet another way to write asynchronous code
- async / await is actually built on top of promises
- async / await is, like promises are non blocking
- async / await makes async code look and behave more like sync code.

# Syntax

## Promises

```
const makeRequest = () =>
  getJSON()
    .then(data => {
      console.log(data)
      return 'done'
    })

makeRequest()
```

## async / await

```
const makeRequest = async () => {
  console.log(await getJSON())
  return 'done'
}

makeRequest()
```

# Subtle behavior

- `async` is defined before the function
- The `await` keyword can only be used inside functions defined with `async`
- Any `async` function returns a `Promise` implicitly

```
// this will not work in top level
// await makeRequest()

// this will work
makeRequest().then(result => {
  // do something
})
```

# Error Handling Benefits

```
const makeRequest = async () => {
  try {
    // this parse may fail
    const data = JSON.parse(await getJSON())
    console.log(data)
  } catch (err) {
    console.log(err)
  }
}
```

# Promises Readability

```
const makeRequest = () => {
  return getJSON()
    .then(data => {
      if (data.needsAnotherRequest) {
        return makeAnotherRequest(data)
          .then(moreData => {
            console.log(moreData)
            return moreData
          })
      } else {
        console.log(data)
        return data
      }
    })
}
```

# Much Better

```
const makeRequest = async () => {
  const data = await getJSON()
  if (data.needsAnotherRequest) {
    const moreData = await makeAnotherRequest(data);
    console.log(moreData)
    return moreData
  } else {
    console.log(data)
    return data
  }
}
```

# Error Stack

The error stack returned from a promise chain gives no clue of where the error happened. Even worse, it's misleading; the only function name it contains is callAPromise which is totally innocent of this error

```
const makeRequest = () => {
  return callAPromise()
    .then(() => callAPromise())
    .then(() => callAPromise())
    .then(() => callAPromise())
    .then(() => callAPromise())
    .then(() => {
      throw new Error('oops')
    })
}

makeRequest()
  .catch(err => {
    console.log(err)
    // output
    // Error: oops at callAPromise.then.then.then.then.then (index.js:8:13)
  })
```

# Error Stack

```
const makeRequest = async () => {
  await callAPromise()
  await callAPromise()
  await callAPromise()
  await callAPromise()
  await callAPromise()
  throw new Error('oops')
}

makeRequest()
  .catch(err => {
    console.log(err)
    // output
    // Error: oops at makeRequest (index.js:7:9)
  })
```

# EventEmitter

# Learning Objectives

- Understand what an EventEmitter is and where it should be used
- Learn how to emit events
- Learn how to add event listeners
- Learn how to remove event listeners

# What Are EventEmitters?

- An implementation of the publish / subscribe pattern
- A mechanism for communicating state changes

```
const { EventEmitter } = require('events')

const emitter = new EventEmitter()
```

# Built-in EventEmitters

EventEmitters are used extensively throughout Node core:

- Streams
- Net / HTTP
- Filesystem
- Domains
- ... and more!

# Adding Listeners

Add listeners using the `on()` function:

### 01_on.js

```
'use strict'

const { EventEmitter } = require('events')
const emitter = new EventEmitter()

// register interest in 'ping' events
emitter.on('ping', function firstPingListener () {
  console.log('First listener: ping')
})

// another 'ping' event listener
emitter.on('ping', function secondPinglistener () {
  console.log('Second listener: ping')
})

// trigger a 'ping' event
emitter.emit('ping')
emitter.emit('ping')
```

# Adding One-Shot Listeners

## 02_once.js

```
'use strict'

const { EventEmitter } = require('events')
const emitter = new EventEmitter()

emitter.once('ping', function pingListener () {
  console.log('Received ping') // only called once
})

emitter.emit('ping')
emitter.emit('ping')
```

# Removing a Listener

```
// remove a single listener
emitter.removeListener('ping', firstPingListener)

// remove all listeners
emitter.removeAllListeners('ping')
```

# EventEmitter in Node.js Core

## **03_http_server.js:** HTTP server

```
'use strict'

const http = require('http')
const server = http.createServer()
const port = parseInt(process.argv[2], 10) || 9000 // default to port 9000
let closing = false

server.on('request', function onRequest (req, res) {
  res.setHeader('Connection', 'close') // turn off keep-alive
  res.end('Hello World!')
  if (!closing) {
    closing = true
    server.close()
  }
})

server.once('listening', () => console.log('Server listening on port %d', port))
server.once('close', () => console.log('Server is closing. Bye bye!'))

server.listen(port)
```

# Sending Messages

Emit alternating `'tic'` and `'tac'` events every second

### 04_tic_tac_emitter.js:

```
'use strict'

const { EventEmitter } = require('events')
const emitter = new EventEmitter()

let tick = true
let count = 0

setInterval(() => {
  count++
  emitter.emit(tick ? 'tic' : 'tac', count)
  tick = !tick
}, 1000)

emitter.on('tic', count => console.log('[%d] TIC', count))
emitter.on('tac', count => console.log('[%d] TAC', count))
```

# When to Use EventEmitter

- When you need to communicate state changes
- When a callback can be called more than once
- When an action can result in more than one *type* of non-error reaction

# Summary

- EventEmitter is a core Node.js construct
- An event has an event name, identified by a string
- Add event listener: `on(name, callback)`
- Add an one time listener: `once(name, callback)`
- Remove a listener: `removeListener(name, fn)`
- Remove all listeners: `removeAllListeners(name)`
- Emit an event: `emit(name [, data, ...])`

# Labs: EventEmitter

# Labs: EventEmitter

Extend the "tic tac" example found in the EventEmitter unit
(filename: *04_tic_tac_emitter.js*) such that:

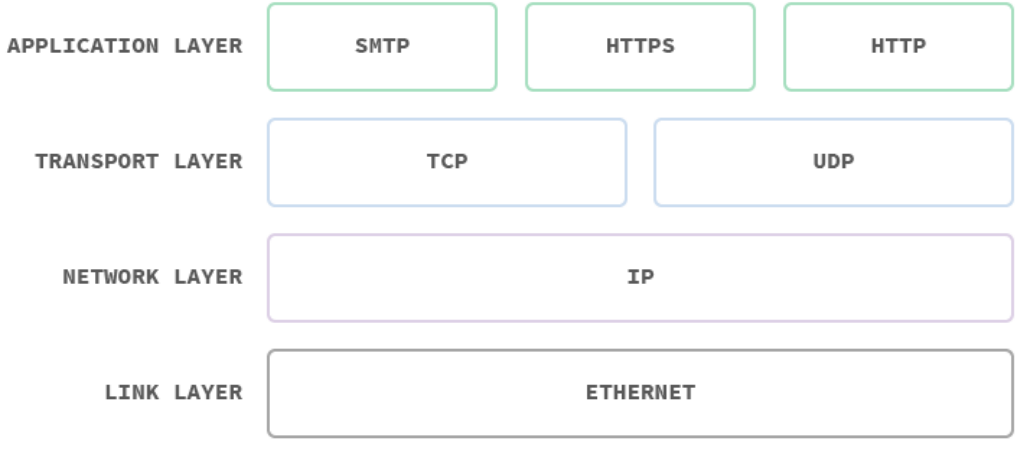
- There is a third event, `'tock'` that is emitted
- The event to emit is chosen at random (use `Math.random()` to choose between the three events)
- The event listeners keep count of how many of *their* event type have been emitted and print this along with the total number of events.

# Networking Basics

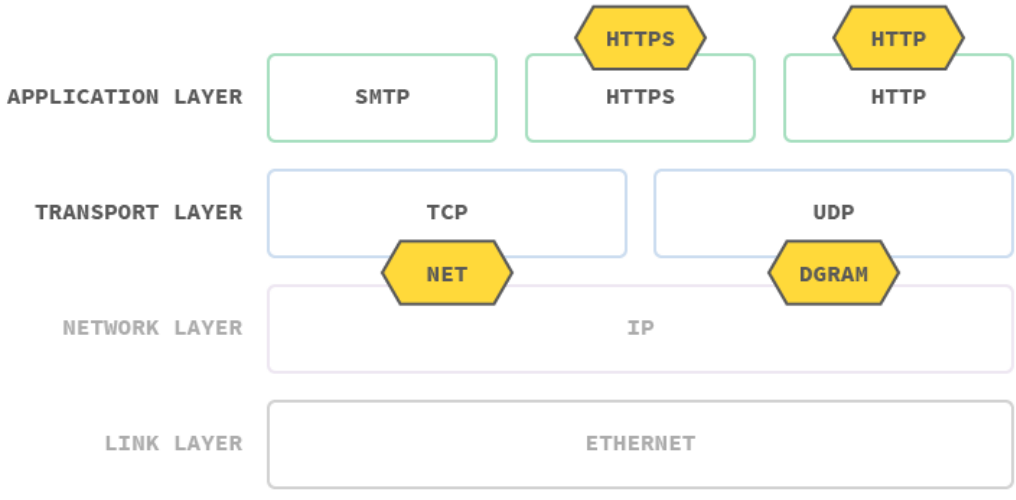
# Learning Objectives

- Understand the basics of TCP, HTTP and HTTPS
- Learn how Node.js exposes each of these via built-in modules

# Network Stack



# Network Stack



# TCP

**T**ransmission **C**ontrol **P**rotocol

- Connection oriented
- Guarantees lossless and ordered transmission of data
- Implemented by Node.js core `'net'` module: http://nodejs.org/api/net.html

# TCP

# TCP Server

## 01_tcp_server.js:

```
'use strict'

const net = require('net')
const server = net.createServer()
const PORT = 8000

server
  .on('connection', onConnection)
  .on('listening', onListening)
  .listen(PORT)

function onConnection (conn) {
  conn.write('You are in a huge cave\r\n')
  conn.pipe(conn)
}

function onListening () {
  console.log('TCP server listening on port', PORT)
}
```

# TCP Server

## 01_tcp_server.js:

```
'use strict'

const net = require('net')
const server = net.createServer()
const PORT = 8000

server
  .on('connection', onConnection)
  .on('listening', onListening)
  .listen(PORT)

function onConnection (conn) {
  conn.write('You are in a huge cave\r\n')
  conn.pipe(conn)
}

function onListening () {
  console.log('TCP server listening on port', PORT)
}
```

Using the Node.js `'net'` module to create a TCP server

# TCP Server

## 01_tcp_server.js:

```
'use strict'

const net = require('net')
const server = net.createServer()
const PORT = 8000

server
  .on('connection', onConnection)
  .on('listening', onListening)
  .listen(PORT)

function onConnection (conn) {
  conn.write('You are in a huge cave\r\n')
  conn.pipe(conn)
}

function onListening () {
  console.log('TCP server listening on port', PORT)
}
```

- Registering callbacks with server and listening on given `PORT`
- `onConnection` invoked for each new client connection
- `onListening` only invoked once

# TCP Server

### 01_tcp_server.js:

```
'use strict'

const net = require('net')
const server = net.createServer()
const PORT = 8000

server
  .on('connection', onConnection)
  .on('listening', onListening)
  .listen(PORT)

function onConnection (conn) {
  conn.write('You are in a huge cave\r\n')
  conn.pipe(conn)
}

function onListening () {
  console.log('TCP server listening on port', PORT)
}
```

- Send invitation message to client by writing to TCP socket connection

# TCP Server

### 01_tcp_server.js:

```
'use strict'

const net = require('net')
const server = net.createServer()
const PORT = 8000

server
  .on('connection', onConnection)
  .on('listening', onListening)
  .listen(PORT)

function onConnection (conn) {
  conn.write('You are in a huge cave\r\n')
  conn.pipe(conn)
}

function onListening () {
  console.log('TCP server listening on port', PORT)
}
```

- Piping all data sent to server back to client causes all client messages to be echoed back to the client, like an "echo server"

# TCP Client

## 02_tcp_client.js:

```
'use strict'

const PORT = 8000
const net = require('net')
const client = net.connect(PORT)

client.on('data', onData)

function onData (data) {
  process.stdout.write('server: ' + data.toString())
  setTimeout(respond, 1000)
}

function respond () {
  const msg = 'Describe cave\r\n'
  process.stdout.write('client: ' + msg)
  client.write(msg)
}
```

# TCP Client

## 02_tcp_client.js:

```
'use strict'

const PORT = 8000
const net = require('net')
const client = net.connect(PORT)

client.on('data', onData)

function onData (data) {
  process.stdout.write('server: ' + data.toString())
  setTimeout(respond, 1000)
}

function respond () {
  const msg = 'Describe cave\r\n'
  process.stdout.write('client: ' + msg)
  client.write(msg)
}
```

# TCP Client

## 02_tcp_client.js:

```
'use strict'

const PORT = 8000
const net = require('net')
const client = net.connect(PORT)

client.on('data', onData)

function onData (data) {
  process.stdout.write('server: ' + data.toString())
  setTimeout(respond, 1000)
}

function respond () {
  const msg = 'Describe cave\r\n'
  process.stdout.write('client: ' + msg)
  client.write(msg)
}
```

- Registering the `onData` callback which will be invoked every time the server sends a message

# TCP Client

## 02_tcp_client.js:

```
'use strict'

const PORT = 8000
const net = require('net')
const client = net.connect(PORT)

client.on('data', onData)

function onData (data) {
  process.stdout.write('server: ' + data.toString())
  setTimeout(respond, 1000)
}

function respond () {
  const msg = 'Describe cave\r\n'
  process.stdout.write('client: ' + msg)
  client.write(msg)
}
```

- Logging server message and scheduling response

# TCP Client

## 02_tcp_client.js:

```
'use strict'

const PORT = 8000
const net = require('net')
const client = net.connect(PORT)

client.on('data', onData)

function onData (data) {
  process.stdout.write('server: ' + data.toString())
  setTimeout(respond, 1000)
}

function respond () {
  const msg = 'Describe cave\r\n'
  process.stdout.write('client: ' + msg)
  client.write(msg)
}
```

- Logging the response we are about to send and sending it by *writing* to the TCP socket

# TCP Connections

## End TCP Client Connection

```
client.end()
```

- Terminates the *client* part of a connection
- You may still get data events on the *server*

# TCP Connections

## Close TCP Server

```
server.close()
```

- The *server* accepts no more *client* connections, but keeps existing ones

# HTTP

- **Application layer** protocol
- Request / response based
- Sits on top of a **transport layer** protocol, like TCP or UDP

# HTTP Request & Response


- Abstractions of the TCP socket
- **Request**:
  - `http.IncomingMessage` is a *readable stream*
  - Represents the part of the socket that is *readable* to the server and *writable* by the client

# HTTP Request & Response

- Abstractions of the TCP socket

- **Request**:
  - `http.IncomingMessage` is a *readable stream*
  - Represents the part of the socket that is *readable* to the server and *writable* by the client

- **Response**:
  - `http.ServerResponse` is a *writable stream*
  - Represents the part of the socket that is *writable* by the server and *readable* to the client

# HTTP Server

## 03_http_server.js:

```
'use strict'

const http = require('http')
const server = http.createServer()
const PORT = 8000

server
  .on('request', onRequest)
  .on('listening', onListening)
  .listen(PORT)

function onRequest (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' })
  res.end('You are still in a huge cave\r\n')
}

function onListening () {
  console.log('HTTP server listening on port', PORT)
}
```

Is very similar to ...

# TCP Server

## 01_tcp_server.js:

```
'use strict'

const net = require('net')
const server = net.createServer()
const PORT = 8000

server
  .on('connection', onConnection)
  .on('listening', onListening)
  .listen(PORT)

function onConnection (conn) {
  conn.write('You are in a huge cave\r\n')
  conn.pipe(conn)
}

function onListening () {
  console.log('TCP server listening on port', PORT)
}
```

# HTTP Server

## 03_http_server.js:

```
'use strict'

const http = require('http')
const server = http.createServer()
const PORT = 8000

server
  .on('request', onRequest)
  .on('listening', onListening)
  .listen(PORT)

function onRequest (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' })
  res.end('You are still in a huge cave\r\n')
}

function onListening () {
  console.log('HTTP server listening on port', PORT)
}
```

Using the Node.js `'http'` module to create an HTTP server

# HTTP Server

## 03_http_server.js:

```
'use strict'

const http = require('http')
const server = http.createServer()
const PORT = 8000

server
  .on('request', onRequest)
  .on('listening', onListening)
  .listen(PORT)

function onRequest (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' })
  res.end('You are still in a huge cave\r\n')
}

function onListening () {
  console.log('HTTP server listening on port', PORT)
}
```

- Registering callbacks with server and listening on given `PORT`
- `onRequest` invoked for *each* client request, `onListening` invoked only once
- Connections are handled for us; an `onConnection` handler is not needed

# HTTP Server

## 03_http_server.js:

```
'use strict'

const http = require('http')
const server = http.createServer()
const PORT = 8000

server
  .on('request', onRequest)
  .on('listening', onListening)
  .listen(PORT)

function onRequest (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' })
  res.end('You are still in a huge cave\r\n')
}

function onListening () {
  console.log('HTTP server listening on port', PORT)
}
```

# HTTPS

- **Application layer** request / response based protocol for secure communications
- Layers the HTTP protocol on top of *TLS/SSL*
- The `'https'` module exposes a very similar API to `'http'` module

# HTTPS

## 04_https_server.js:

```
'use strict'

const PORT = 8000
const https = require('https')
const fs = require('fs')
const server = https.createServer({
  pfx: fs.readFileSync('some_cert.pfx')
})

server
  .on('request', onRequest)
  .on('listening', onListening)
  .listen(PORT)

function onRequest (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' })
  res.end('You are still in a huge, but secure, cave\r\n')
}

function onListening () {
```

# HTTP

## 03_http_server.js:

```
'use strict'

const http = require('http')
const server = http.createServer()
const PORT = 8000

server
  .on('request', onRequest)
  .on('listening', onListening)
  .listen(PORT)

function onRequest (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' })
  res.end('You are still in a huge cave\r\n')
}

function onListening () {
  console.log('HTTP server listening on port', PORT)
}
```

# Summary

- The `'net'` module provides an interface to the TCP layer by exposing an API for servers and clients
- TCP sockets are *duplex* streams
- The `'http'` module implements the HTTP protocol
- The HTTP API `request` and `response` objects abstract the underlying socket, allowing us to set headers and writing data to the stream
- The `'https'` module implements the HTTPS protocol with very similar API to the `'http'` module

# Streams Basics

# Learning Objectives

- Understand what Node.js Streams are
- Understand why they are important in an asynchronous environment and how they help with scalability
- Understand the difference between Readable, Writable, and Duplex Streams
- Learn how Node.js core uses Streams in its APIs
- Understand Node.js Stream piping, flow-control, and backpressure

# What Are Streams?

## Streams produce, consume and transform data

*"Data"* can be:

- Buffers (default)
- Strings
- or plain objects (in "object mode")

# What Are Streams?

Streams can take many forms:

- Files
- TCP connections
- HTTP messages
- Database Records

# Why Streams?

## A Pluggable Abstraction

A useful modular way to construct data pipelines

# Why Streams?

## Scalability

- Operate on data by the chunk
- Work around memory constraints
- Allow work to yield to the event loop

# Streams as EventEmitters

Streams are EventEmitters

### 01_stream_events.js:

```
'use strict'

const https = require('https')
const url = 'https://en.wikipedia.org/wiki/List_of_buzzwords'

https.get(url, res => {
  let byteCount = 0

  res.on('data', d => {
    console.log('Received %d bytes', d.length)
    byteCount += d.length
  })

  res.on('end', () => {
    console.log('Ended with total %d bytes', byteCount)
  })
})
```

# Streams as EventEmitters

Often we need to buffer the result of a Stream before we can continue

**02_stream_collect.js:** A basic "collect" pattern for receiving data

```
'use strict'

const https = require('https')
const url = 'https://en.wikipedia.org/wiki/List_of_buzzwords'

https.get(url, res => {
  let page = '' // concatenate entire page into a single variable

  res.setEncoding('utf8')

  res.on('data', d => {
    console.log('Received %d bytes', d.length)
    page += d.toString() // `d` is a Buffer, convert to a String and concat
  })

  res.on('end', () => {
    console.log('Got entire page:')
    console.log(page)
  })
})
```

# Stream Types

The Node.js core `'stream'` module contains a set of base Stream abstractions

- **`stream.Readable`**: a source of flowing data
- **`stream.Writable`**: a destination for flowing data
- **`stream.Duplex`**: a Stream that can be both read from and written to
- **`stream.Transform`**: a Duplex Stream that transforms data as it flows through
- **`stream.PassThrough`**: a no-op implementation of `stream.Transform`

# Stream Types: Readable

Found throughout Node.js core where there is a data *source*:

- `fs.createReadStream()`
- `http.request(options, function (response) { /* response is a Readable */ })`
- `http.createServer(function (request, response) { /* request is a Readable */ })`
- ... many others

# Stream Types: Writable

Found throughout Node.js core where data has a *destination*:

- `fs.createWriteStream()`
- `http.createServer(function (request, response) { /* response is a Writable */ })`
- ... many others

# Connecting Streams Together: pipe()

Readables can be *piped* to Writables

**05_pipe_fs.js:** Copy a file using Streams

```
/// create streams
const source = fs.createReadStream(__filename)
const destination = fs.createWriteStream(__filename + '.copy')

// copy source file to destination
source.pipe(destination)
```

# Connecting Streams Together: pipe()

**06_pipe_http.js:** Connecting an HTTP request Readable to a file-system Writable

```
/// create a Writable
const targetStream = fs.createWriteStream(targetFile)

https.get('https://nodejs.org/static/images/logo.svg', res => {
  // res is a Readable
  res.pipe(targetStream)
    .on('error', err => console.error(err))
})
```

# Stream Types: Duplex

Duplex Streams implement both Readable and Writable behavior

An example of Duplex Streams Node.js core is TCP network sockets:

**04_duplex_net.js**

```
'use strict'

const net = require('net')

net.createServer(socket => {
  // pipe the Readable side to the Writable side of the same socket
  // to create an echo server
  socket.pipe(socket)
}).listen(1337)
```

# Stream Behavior: Ending

```
// will end the target Stream when the source Stream ends
source.pipe(target);

// doesn't end target Stream when the source ends
source.pipe(target, { end: false });
```

Useful for multi-pipe situations, e.g. concatenating multiple files into one

# Stream Behavior: *Backpressure*

*Backpressure* is what happens when a Writable stream cannot keep up with the data being supplied to it.

It is an *advisory* signal that tells the upstream source to back-off, it is up to the implementation to handle the signal.

# Stream Behavior: *Backpressure*

Example: Sending a large file over the network

**07_tcp_server.js:** A server to receive data

```
/// create a network server
net.createServer(socket => {
  // each time we get a connection, write the data to a new file
  const outFile = `out${i++}.dat`

  socket.pipe(fs.createWriteStream(outFile))
    .on('error', err => console.error(err))
    .on('finish', () => {
      console.log('Finished writing to %s', outFile)
    })
}).listen(1337)
```

# Stream Behavior: *Backpressure*

Example: Sending a large file over the network

**08_tcp_client.js:** A client to send data

```
/// connect to server
const socket = net.connect({ port: 1337 })
socket.on('error', err => console.error(err))

const ws = fs.createReadStream(__filename + 'pep')
ws.on('error', err => console.error(err))
  .on('finish', () => {
    console.log('Finished sending file')
  })

ws.pipe(socket)
```

# Stream Behavior: *Backpressure*

Backpressure is applied:

- On the client when the server doesn't receive as fast as the client can send
- On the server when the filesystm doesn't write as fast as the network receives

Backpressure allows *very large* files to be sent without significant memory being used on either client or server, regardless of network and disk speed

# Stream Behavior: Errors

Like any other EventEmitter, streams can emit `'error'` events that must be handled or they will `throw` an `'unhandledException'`

**08_tcp_client.js:** `'error'` must be handled on all Stream instances

```
'use strict'

const net = require('net')
const fs = require('fs')

/// connect to server
const socket = net.connect({ port: 1337 })
socket.on('error', err => console.error(err))

const ws = fs.createReadStream(__filename + 'pep')
ws.on('error', err => console.error(err))
  .on('finish', () => {
    console.log('Finished sending file')
  })

ws.pipe(socket)
///
```

# Summary

- Streams produce, consume and transform data
- Streams process `Buffer` and `String` objects by default
- Streams are `EventEmitter` objects and the `'error'` event should be handled
- The `'stream'` module contains a set of base objects that can be extended for custom implementations
- Readable Streams produce or read data from a source
- Writable Streams consume or write data to a destination
- Duplex Streams implement perform both Readable and Writable functionality
- `pipe()` is used to connect Streams together to make data flow
- Backpressure is how Node.js maintains efficient memory usage with Streams while dealing with data sources and destinations of varying speed

# File System

# Learning Objectives

- Learn how to manipulate paths with `path` module
- Learn how to read and write files using three levels of abstraction:
  - Low-level POSIX-like methods
  - Buffering methods for convenience
  - Streaming methods for performance
- Understand the costs of using synchronous file system operations
- Using standard I/O streams

# Manipulating Paths

The Node.js core `'path'` module is used for manipulating and resolving paths

- Utility methods
- Safe and cross-platform
- Does not perform I/O

# The 'path' Module

```
const path = require('path')
```

# Path Operations

## 01_path_operations.js

```
'use strict'

// Run these in the REPL

path.join('/foo', 'bar', 'baz/asdf', 'quux', '..')
// '/foo/bar/baz/asdf'

path.normalize('/foo/bar//baz/asdf/quux/..')
// '/foo/bar/baz/asdf'

path.resolve('/foo/bar', './baz')
// '/foo/bar/baz'

path.resolve('wwwroot', 'static_files/png/', '../gif/image.gif')
// '/Users/tmpvar/wwwroot/static_files/gif/image.gif'

path.relative('/path/to/test', '/path/to/foo')
// '../foo'
```

# Path Component Operations

## 02_path_component_operations.js

```
'use strict'

// Run these in the REPL

path.dirname('/foo/bar/baz/asdf/quux.txt')
// '/foo/bar/baz/asdf'

path.basename('/foo/bar/baz/asdf/quux.txt')
// 'quux.txt'

path.basename('/foo/bar/baz/asdf/quux.txt', '.txt')
// 'quux'

path.extname('/foo/bar/baz/asdf/')
// ''

path.extname('/foo/bar/baz/asdf/quux.txt')
// '.txt'
```

# File system I/O

The Node.js core `'fs'` module is used for querying and manipulating files and directories

Contains three levels of abstraction:

1. Near-direct translation of the raw UNIX / POSIX APIs
2. Buffering read and write operations for convenience
3. Streaming read and write operations for performance

# The 'fs' Module

Provides APIs for low and high-level access to the underlying file system

```
const fs = require('fs')

console.log(Object.keys(fs))
```

# Utility Methods For the File System

`'fs'` contains many utility methods for querying and manipulating the file system

## 03_stat.js

```
'use strict'

const fs = require('fs')
const file = process.argv[2] || __filename

fs.stat(file, function onStat (err, stats) {
  if (err) return console.error(err)

  console.log(stats, stats.isFile(), stats.isDirectory())
})
```

# File Descriptors

File descriptors provide access to an index on the operating system's file descriptor table

Regular file descriptors are handles to files and directories on disk

Special file descriptors exist for standard out, error and in and other special system resources

# Low-level: Reading Files With fs.read()

**04_fs_read.js:** random read access using low-level APIs

```
'use strict'

const fs = require('fs')
const file = process.argv[2] || __filename

fs.open(file, 'r', function onOpen (err, fd) {
  if (err) return console.error(err)

  const buffer = Buffer.alloc(1024)
  const length = buffer.length
  const filePosition = parseInt(process.argv[3], 10) || 100

  fs.read(fd, buffer, 0, length, filePosition, function onRead (err, bytes) {
    if (err) return console.error(err)

    console.log('Read %d bytes', bytes)
    if (bytes > 0) console.log(buffer.slice(0, bytes).toString())
  })
})
```

# Low-level: Reading Files With fs.read()

```
$ node 04_fs_read.js 04_fs_read.js 400
```

```
$ node 04_fs_read.js 04_fs_read.js 350
```

# Low-level: Writing Files With fs.write()

**05_fs_write.js:** create and append to a file using low-level APIs

```
'use strict'

const fs = require('fs')
const path = require('path')

fs.open(path.join(__dirname, 'out.txt'), 'a', function onOpen (err, fd) {
  if (err) return console.error(err)

  const buffer = Buffer.from('Writing this string', 'utf8')
  const length = buffer.length

  fs.write(fd, buffer, 0, length, null, function onWrite (err, written) {
    if (err) return console.error(err)

    console.log(`Wrote ${written} bytes`)
  })
})
```

# Buffering: Reading Files With fs.readFile()

**06_fs_readfile.js:** read an entire file, buffering the content until ready

```
'use strict'

const fs = require('fs')
const file = process.argv[2] || __filename

fs.readFile(file, function onReadFile (err, bytes) {
  if (err) return console.error(err)

  console.log(bytes.toString())
})
```

*Can also return plain Strings by supplying an encoding argument* `'utf8'`

# Buffering: Writing Files With fs.writeFile()

**07_fs_writefile.js:** append a buffer to a file

```
'use strict'

const fs = require('fs')
const path = require('path')
const file = path.join(__dirname, 'out.txt')
const buffer = Buffer.from('Writing this string', 'utf8')

fs.writeFile(file, buffer, { flag: 'a' }, function onWriteFile (err) {
  if (err) return console.error(err)

  console.log('Wrote %d bytes', +buffer.length)
})
```

*Can also write plain Strings*

# Streaming: Reading Files With Streams

**08_fs_readstream.js:** create ReadableStream for a file

```
'use strict'

const fs = require('fs')
const file = process.argv[2] || __filename

fs.createReadStream(file).pipe(process.stdout)
```

# Streaming: Writing Files With Streams

**09_fs_pipe.js:** stream a file to another location on the file system

```
'use strict'

const fs = require('fs')
const path = require('path')
const src = process.argv[2] || __filename
const dest = process.argv[3] || path.join(__dirname, 'copy.js')

fs.createReadStream(src).pipe(fs.createWriteStream(dest))
```

# Synchronous I/O

Most `'fs'` methods are also exposed as a `*Sync()` alternative for synchronous I/O

```
// asynchronous I/O
fs.readFile(__filename, function onReadFile (err, data) {
  if (err) return console.log(err)
  console.log(data.toString())
});

// synchronous I/O
try {
  const data = fs.readFileSync(__filename)
  console.log(data.toString())
} catch (err) {
  return console.log(err)
}
```

# Synchronous I/O

Synchronous operations are performed on the JavaScript thread and will **block** normal operation

Error handling by `throw` instead of callbacks

Should be avoided unless absolutely necessary

# Standard I/O

Node.js exposes the standard I/O file descriptors as follows:

- `process.stdin`: Standard input, file descriptor 0
- `process.stdout`: Standard output, file descriptor 1
- `process.stderr`: Standard error, file descriptor 2

# Writing to Standard I/O

```
process.stdout.write('hello world\n')
process.stderr.write('error!\n')
```

# Standard I/O Streams

You can treat the Node.js standard I/O objects just like regular file streams:

- `process.stdin`:
  ReadableStream
- `process.stdout`:
  WritableStream
- `process.stderr`:
  WritableStream

**10_stdin.js:** echo standard input back to standard output

```
'use strict'

process.stdin.pipe(process.stdout)

process.stdin.on('end', () => {
  process.stdout.write('\ndone\n')
})
```

# Summary

- Node.js provides a rich set of tools for working with the file system
- The `'path'` module contains non-I/O file path utilities
- The `'fs'` module exposes utility methods, low-level POSIX-like methods, buffering convenience methods and a Streams API
- Most `'fs'` methods have an corresponding `fs.*Sync()` method

http://nodejs.org/api/fs.html

# Labs: File System

# Labs: File System

Write a program that is self-replicating!

Your program file should be called 'replicator0.js' and each time it is run, a new file is created that contains an identical copy of the program. The new file should be called 'replicator*X*.js' where *X* is the next *available* integer in the series of replications

You should be able to execute any of the 'replicator*X*.js' files and a new file will be created, continuing the sequence of *X*

- You should use `fs.createReadStream()` and `fs.createWriteStream()` to perform the actual replication
- The `__filename` variable and `fs.exists()` will come in handy

# Child Processes

# Learning Objectives

- Understand how Node.js exposes process forking
- Learn how to create new processes from Node.js
- Learn difference between `exec()`, `fork()` and `spawn()`
- Learn how to configure child process execution

# The *child_process* Module

The *child_process* module is your interface to creating, communicating and managing the lifecycle of new processes

```
require('child_process')
```

# exec()

`exec()` will execute any command as if you had typed it into your terminal

```
const { exec } = require('child_process')

exec(command, [options], callback)
```

# exec() Example

First, let's run a command in the terminal:

```
$ cat *.js does_not_exist | wc -l
cat: does_not_exist: No such file or directory
       8
```

# exec() Example

The equivalent using *child_process*:

## 01_exec.js:

```
'use strict'

const { exec } = require('child_process')

const command = 'cat *.js does_not_exist | wc -l'

exec(command, function onExec (err, stdout, stderr) {
  if (err) throw err

  console.log('Stdout: ' + stdout.toString())
  console.log('Stderr: ' + stderr.toString())
})
```

# exec() Example

Executing `01_exec.js`:

```
$ node 01_exec.js
Stdout:        68

Stderr: cat: does_not_exist: No such file or directory
```

# exec() Caveats

`exec()` is very convenient for executing simple commands, but it has some caveats you should be aware of

# exec() Only Returns on Exit or Timeout

`exec()` will only fire its callback when the spawned process exits, or a timeout is reached

This rules out `exec()` for use with long-running or streaming processes

# exec() Timeout

You *can* adjust the timeout length by passing the `timeout` option:

## 02_exec_timeout.js:

```
'use strict'

const { exec } = require('child_process')

exec('ping google.com', { timeout: 1000 }, function onExec (err, stdout, stderr) {
  console.log('Error:\n', err)
  console.log('Stdout:\n', stdout)
  console.log('Stderr:\n', stderr)
})
```

# exec() Output is Streamed into Fixed-Size Buffers

If your command's output exceeds this size, `exec()` will fail with an error and the output will be truncated

# exec() Output is Streamed into Fixed-Size Buffers

You can adjust the size of these buffers by passing in `maxBuffer`:

### 03_exec_maxbuffer.js:

```
'use strict'

const { exec } = require('child_process')

exec('ping google.com', { maxBuffer: 1 }, function onExec (err, stdout, stderr) {
  console.log('Error:\n', err)
  console.log('Stdout:\n', stdout)
  console.log('Stderr:\n', stderr)
})
```

# exec() Caveat

Don't use `exec()` for long-lived commands or anything which could potentially output a lot of data

# spawn()

`spawn()` is similar to `exec()`, except it provides no buffering—you're immediately returned streams corresponding to the stdio of the new process

`spawn()` provides the lowest-level access available for spawning child processes

```
const { spawn } = require('child_process')

spawn(command, [args], [options])
```

# spawn() Example

Tailing a file with `spawn()`:

## 04_tail.js:

```
'use strict'

const { spawn } = require('child_process')
const fs = require('fs')
const path = require('path')

const fileName = path.join(__dirname, 'somefile.txt')
const out = fs.createWriteStream(fileName)

// write a line to the file every second
setInterval(() => {
  out.write((new Date()).toString() + ' - Hey there!\n')
}, 1000)

/// tail
const tail = spawn('tail', [ '-f', fileName ])

tail.stdout.on('data', d => {
  console.log('[tail.stdout]: ' + d)
})
```

# spawn() vs. exec()

`exec()` is a convenience wrapper around `spawn()` for the common use-case of running a command then processing its output

Use `spawn()` to interact directly with the child's stdio streams

# fork()

Like `spawn()` but specifically for creating **new Node.js processes**

`fork()` opens a special data channel for inter-process communication

```
const { fork } = require('child_process')

fork(modulePath, [args], [options]);
```

**Note**: this is not the same as the system level `fork()` on UNIX

# fork() Example

## 05_parent.js:

```
'use strict'

const { fork } = require('child_process')
const path = require('path')

const child = fork(path.join(__dirname, '06_child.js'), {
  env: {
    NODE_ENV: 'production'
  }
})

let messageCount = 0
child.on('message', message => {
  messageCount++
  console.log('[parent] got message from child', message)
  child.send('I got your message #' + messageCount + ', thanks!')
})
```

Send messages to the child using `child.send('my message')`

Receive messages from the child using `child.on('message', fn)`

# fork() Example

## 06_child.js:

```
'use strict'

console.log(process.env)

process.on('message', m => {
  console.log('[child] got this message from parent:', m)
})

setInterval(() => {
  process.send({
    a: 'Here is something for you',
    b: true
  })
}, 1000)
```

Send messages to the parent using `process.send('my message')`

Recieve events from the parent using `process.on('message', fn)`

# fork() Example

Run it from the command line:

```
$ node 03_parent.js
[parent] got message from child { a: 'Here is something for you', b: true }
[child] got this message from parent: I got your message #1, thanks!
[parent] got message from child { a: 'Here is something for you', b: true }
[child] got this message from parent: I got your message #2, thanks!
[parent] got message from child { a: 'Here is something for you', b: true }
[child] got this message from parent: I got your message #3, thanks!
[parent] got message from child { a: 'Here is something for you', b: true }
[child] got this message from parent: I got your message #4, thanks!
[parent] got message from child { a: 'Here is something for you', b: true }
[child] got this message from parent: I got your message #5, thanks!
...
```

You will see messages being exchanged between the parent and child processes

# Configuring a Child Process

When creating a child process with any of the aforementioned methods,
you can specify the environment it executes in, including:

- Current working directory
- Environment variables
- Custom stdio streams
- User and group to execute
  under

You can also specify whether the process is 'detached',
i.e. does not die when the parent dies

# Summary

- *child_process* utilities allow you to control the execution of other programs
- Know when to use `exec()` vs `spawn()`
- `fork()` sets up a special communication channel between node processes
- You can configure the environment a child processes runs in

# Labs: Child Process

# Labs: Child Process

In the *Introduction to Node.js* unit in day 1, you were shown a simple method for estimating the value of π using a Monte Carlo method. Unfortunately this method is CPU intensive and will block the main JavaScript thread from performing any other work.

```
'use strict'

const points = 100000000;
let inside = 0;

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

console.log(`π ≈ ${(inside / points) * 4}`)
```

Your task is to distribute this work across child-processes, leaving your main process' event loop unblocked.

*(continued on next page)*

# Labs: Child Process (cont.)

Use the `fork()` method in the 'child_process' module to distribute this calculation over multiple processes. Use the `cpus()` method in the 'os' module to determine how many CPUs your computer has and spawn *at least* that many processes: `require('os').cpus().length`.

Each child-process should plot at least `100000000` points (more if you have a particularly powerful computer!) and come up with its own estimation for π. When the child process has an estimation, it should use the child_process messaging passing functionality to pass the number back to the main process.

When the main process has all of the estimations from the child processes, take an average of those values and print out the average as your new estimation for π.

# Debugging Node.js

# Learning Objectives

- Understand the difficulties for debugging imposed by an asynchronous environment
- Learn about some of the solutions natively available in Node.js for debugging
- Learn about popular external tools and development environments with debugging facilities

# Challenges Debugging Asynchronous Applications

## Stack traces are truncated

The stack is *blown away* when crossing the asynchronous boundary

# Challenges Debugging Asynchronous Applications

Solutions?

- **Node.js Domains** - currently deprecated
- **Async Hooks** - currently in Node.js but not documented
- **llnode** - advanced debugger for Node.js
- Userland options
  - **longjohn** - adds overhead
  - **trace** - async stacktraces, adds overhead

# Built-in Node.js Debugger

## Flags for `node`

- `--inspect` - open a port and listen for remote debuggers (Used along with Chrome Developer Tools)
- `--inspect-brk` - same as `--inspect` but breaks on the first line

## JavaScript / V8 built-ins

- `debugger` - acts as a breakpoint

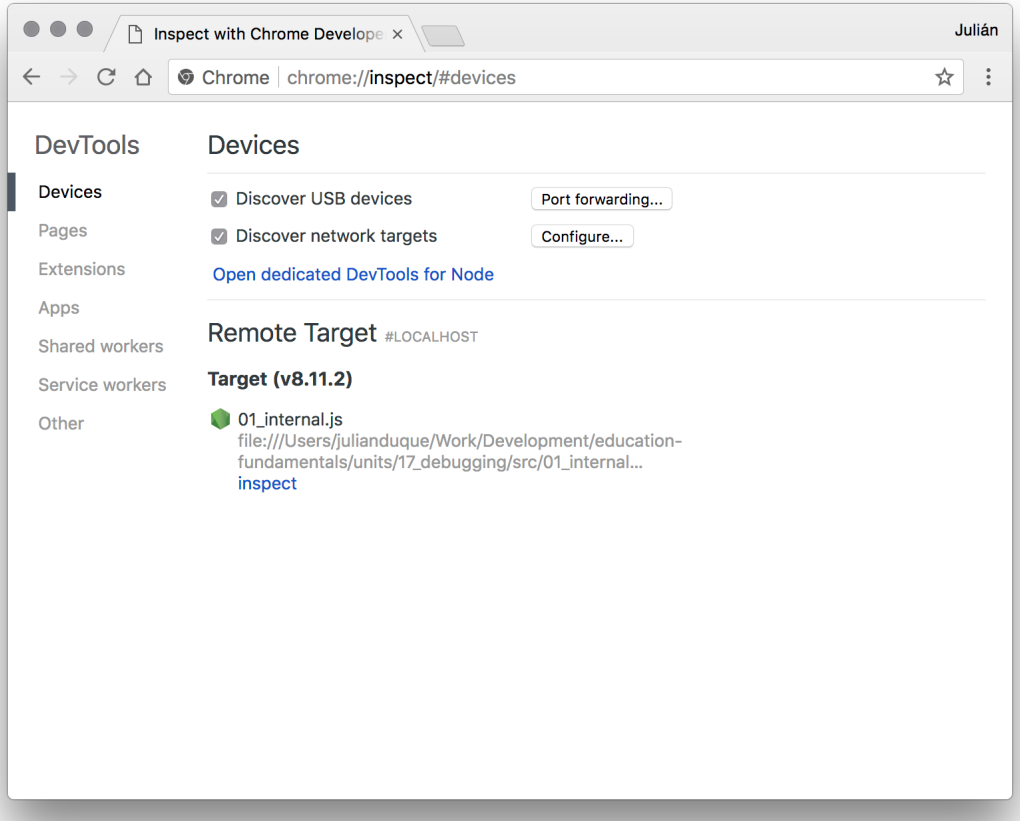
# Built-in Node.js Debugger

Start process in debug mode

```
$ node --inspect-brk 01_internal.js
Debugger listening on ws://127.0.0.1:9229/97663784-4f4a-4abe-8fba-0ea3245d1c4b
For help see https://nodejs.org/en/docs/inspector
```

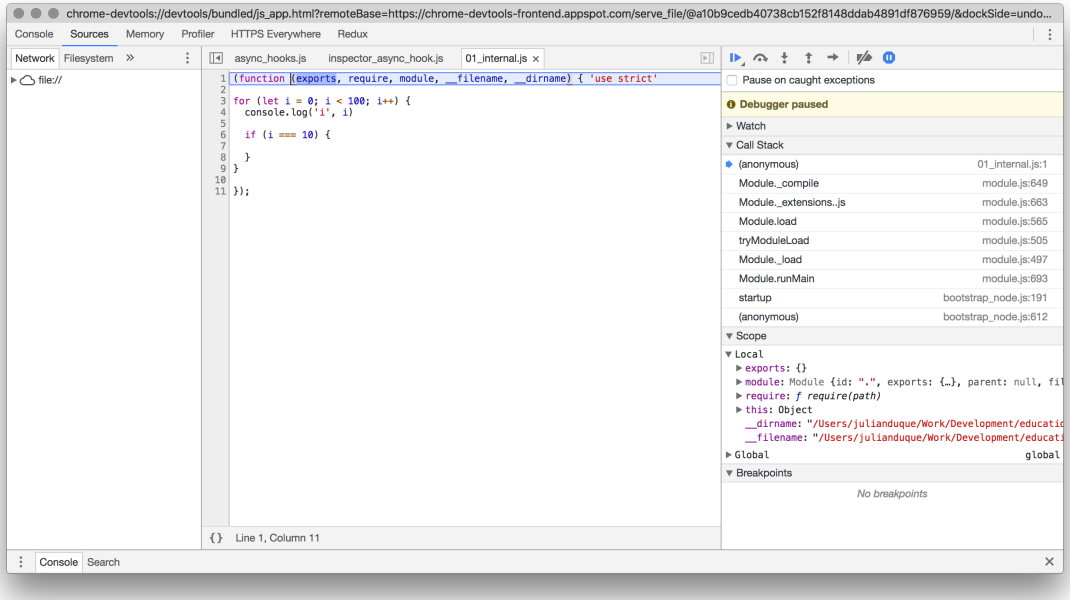
# Built-in Node.js Debugger

Open Google Chrome and load `chrome://inspect`



# Built-in Node.js Debugger

Click on `inspect` and it will open Chrome Developer Tools



# Built-in Node.js Debugger

## Signal the process

Start long running process

```
$ node 02_interval.js
```

In another terminal, find the PID

```
$ ps | grep 02_interval
```

Send a `SIGUSR1` signal to the process

```
$ kill -SIGUSR1 <PID>
```

In the first terminal you should see

```
Debugger listening on ws://127.0.0.1:9229/5f4dce41-0ee9-49f6-8fc5-06387ae6bef8
For help see https://nodejs.org/en/docs/inspector
```

# node-inspector

- A remote debugging interface for Node.js
- Runs in Chrome
- Based on Blink Developer Tools (WebKit Web Inspector)
- Good for Node.js v4.x or older versions

# Using node-inspector

Install

```
$ npm install -g node-inspector
```

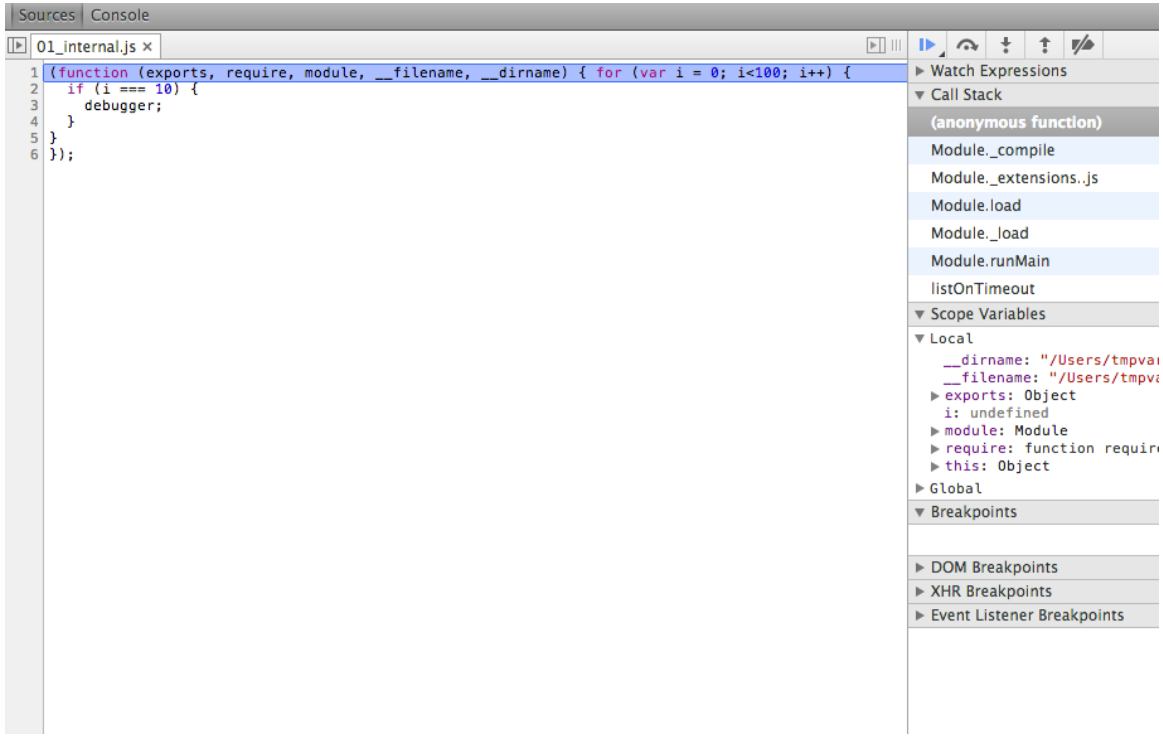
Launch `node-inspector`

```
$ node-inspector
Node Inspector v0.12.8
Visit http://127.0.0.1:8080/debug?port=5858 to start debugging.
```

Launch the file to be debugged

```
$ node --debug-brk 01_internal.js
```

# Open In Chrome




```
http://127.0.0.1:8080/debug?port=5858
```

# Visual Studio code

# Installing Visual Studio code

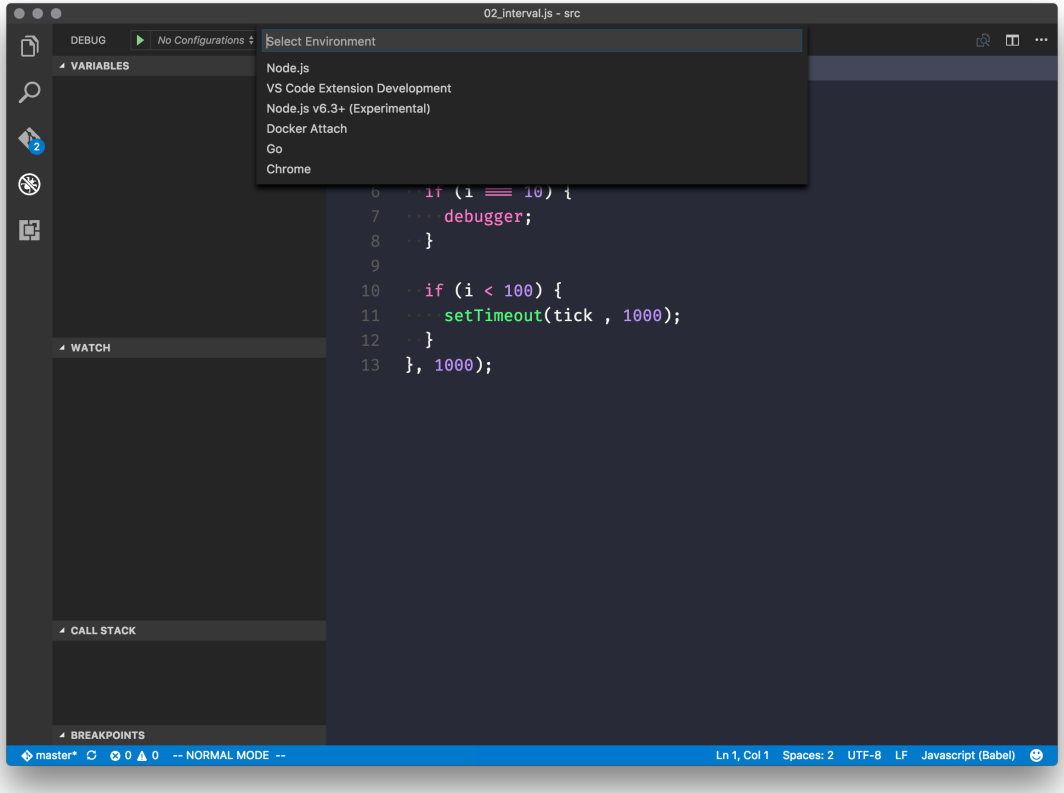
Download the installer approariate to your platform:

http://code.visualstudio.com/

Open and follow the instructions to install

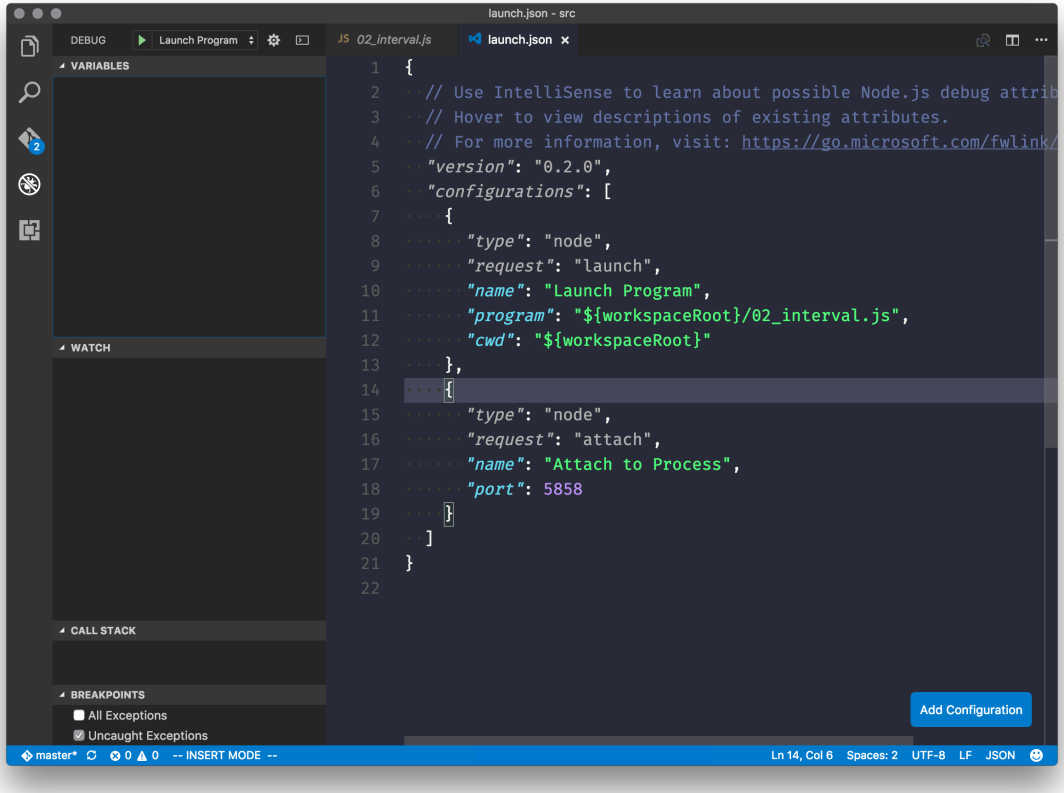
# Debugging with Visual Studio Code

- Go to Debug view and select the Node.js environment for debugging



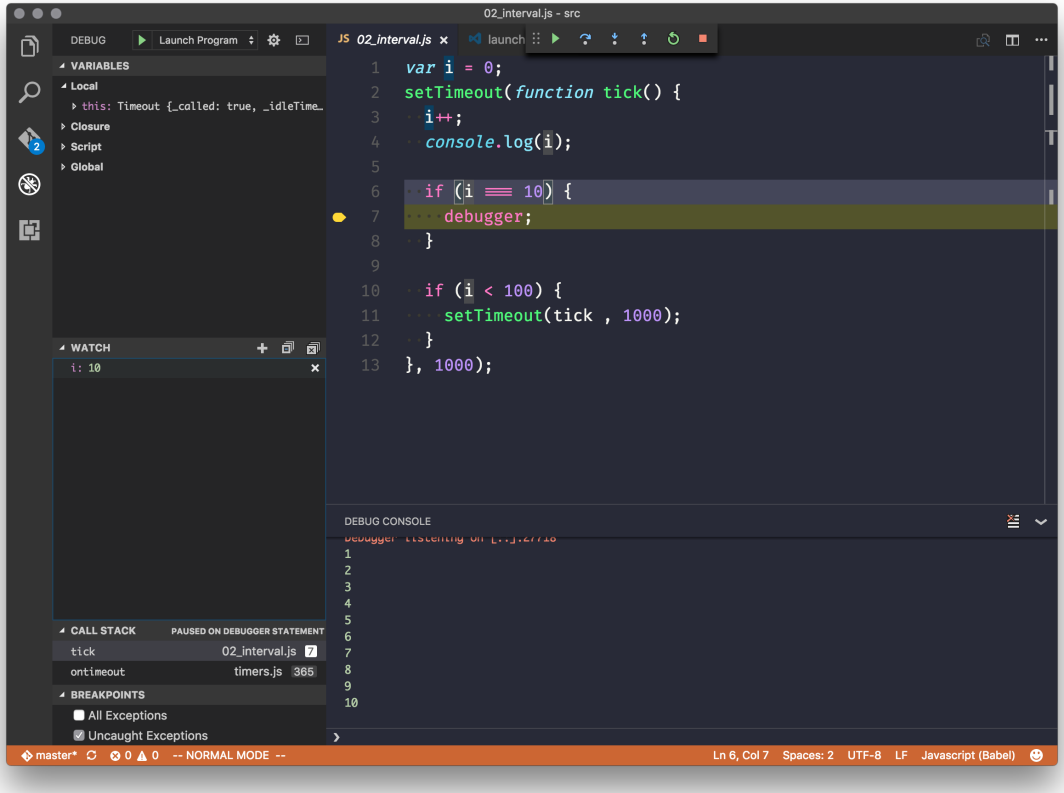
# Debugging with Visual Studio Code

Set the main file of your application to debug in `launch.json` file

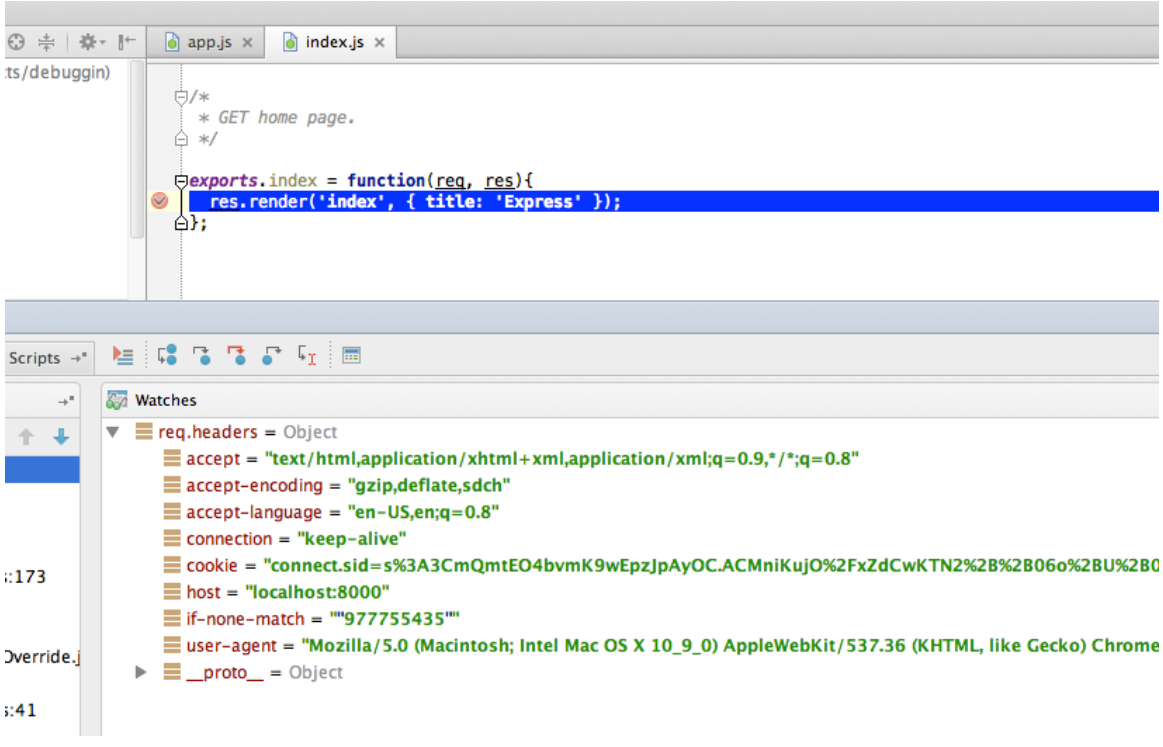


# Debugging with Visual Studio Code

## Run the debugger



# WebStorm



# Installing WebStorm

Download the installer appropriate to your platform:

http://www.jetbrains.com/webstorm/download/

Open and follow the instructions to install

# Debugging with WebStorm

- Set up a project
- Set a breakpoint by clicking left of the line number
- Click the little "bug" icon or press Control+D

# Other GUI Debuggers

- Brackets with Theseus
- Cloud9 IDE (c9.io)
- Eclipse with Google Chrome Developer Tools

# Log based Debugging

Use the `debug` third party module to perform log based debugging

```
$ npm install debug
```

Use it in your code, `debug` returns a function than can instantiate a logger based on a namespace.

```
const log = require('debug')('namespace')

log('This is a debug message')
```

Enable debugging by setting the `DEBUG` environment variable with the namespace

```
$ DEBUG=namespace node app.js
```

If you want to log all the namespaces you can pass an asterisk `*`, or list the namespaces separated by comma

```
$ DEBUG=server,db node app.js
```

# Summary

- Debugging in an asynchronous environment is a non-trivial exercise
- Use the right tool for the job
- Get comfortable with the available tools