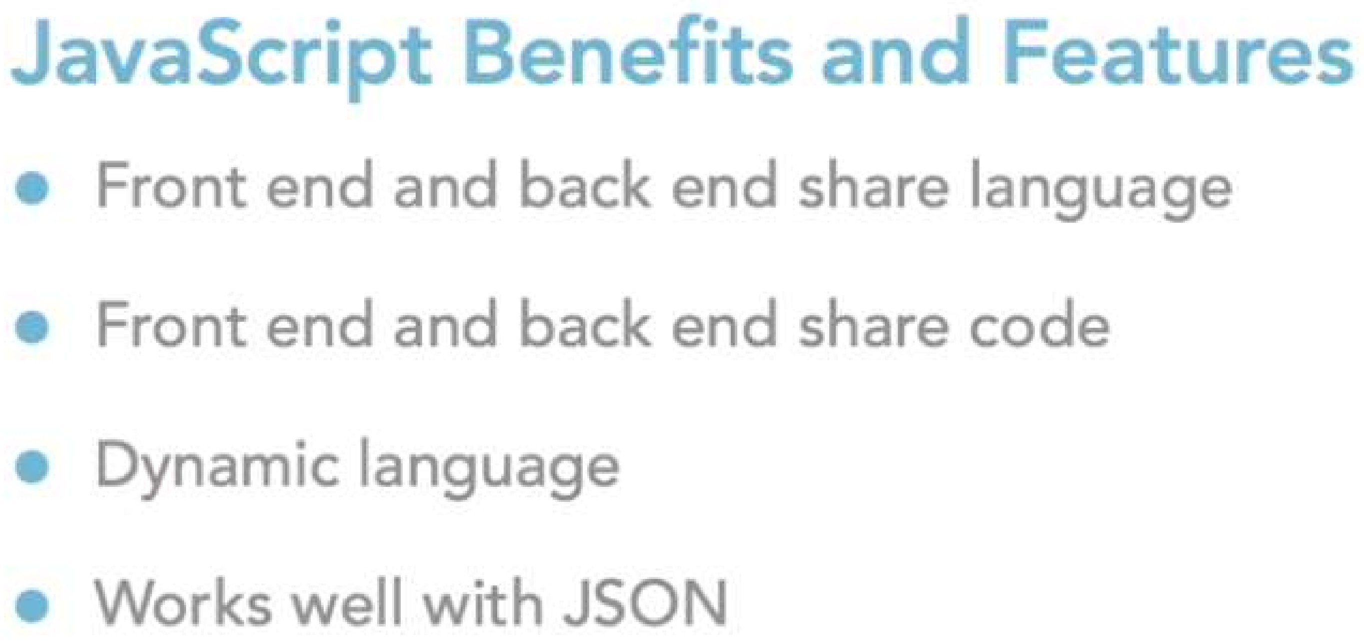
- **Topics**

- What you should know?
- Intro to Node
- NPM
- Node’s Event Loop
- Reading Writing Files
- EventEmitter
- Web frameworks   -like express, body-parser etc
- Building Chat application with node
- Exploring databases - mLab, mongodb
- Error handling and debugging
- Authentication

- JS



JavaScript Benefits and Features

- Front end and back end share language
- Front end and back end share code
- Dynamic language
- Works well with JSON

- In the simplest terms, the tilde matches the most recent minor version (the middle number). ~1.2.3 will match all 1.2.x versions but will miss 1.3.0.
- The caret, on the other hand, is more relaxed. It will update you to the most recent major version (the first number). ^1.2.3 will match any 1.x.x release including 1.3.0, but will hold off on 2.0.0.

- **Getting Started with Node.js – Module I**
- Node.js Background

- "Node.js" is a server-side "JavaScript" platform.
- It was first introduced to the community by its creator, Ryan Dahl, at the "2009 JSConf.eu Conference". Its presentation was received with a standing ovation.
- Since that time, "Node" has continued to evolve, with contributions from the community as well as the project's primary sponsor, cloud computing Company Joyent. "Node" is among the most popular projects on "GitHub", often beating out other heavyweights such as "jQuery" and "Ruby on Rails".

- Node.JS is a runtime environment and library for running JS applications outside the browser
- Node.js is mostly used to run real-time applications and shines through its performance using non-blocking I/O and asynchronous events.
- Node is an open source, server-side, runtime environment.
- Node is not a language but a runtime environment.
- Node is cross platform and uses JS as its language

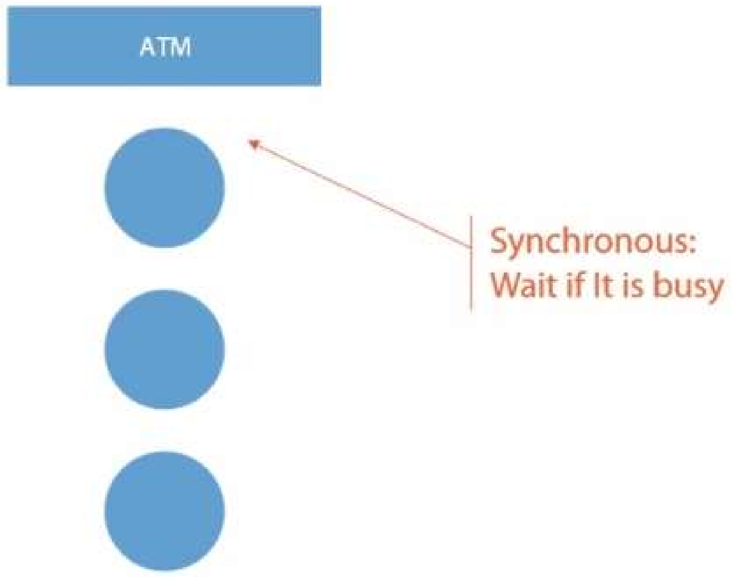
NODE.JS

Server-side JavaScript for Network Applications

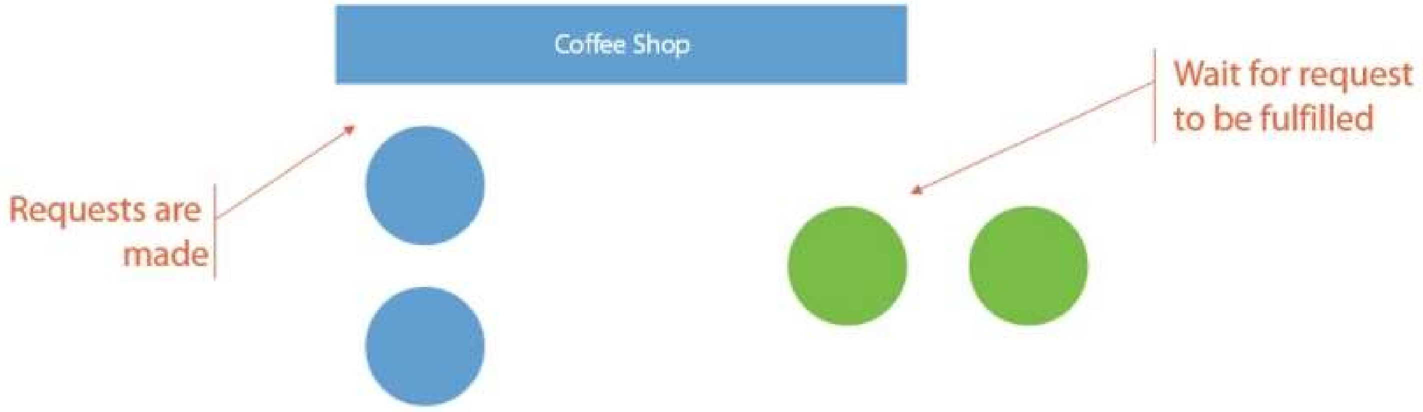
- Supports Modular JavaScript
- Encourages non-blocking (asynchronous) code
- HTTP, Networking, and Web Sockets are 1st class citizens
- Node.js is a low-level, fast platform
  - But wasn't written in JavaScript

- Node.Js and traditional webservers like IIS
- A traditional web server is more like an ATM, in which you have requests coming in and as the request is being fulfilled the person at the front of the line is using the ATM.
- In code you do this synchronous interaction like getting data from the database , do some calculations, rendering a page etc.
- All the above operations are synchronous, so that you are using a thread to serve an individual request until it is done.
- If too many people are lined up to get at the different ATM ports, then other operations simply wait

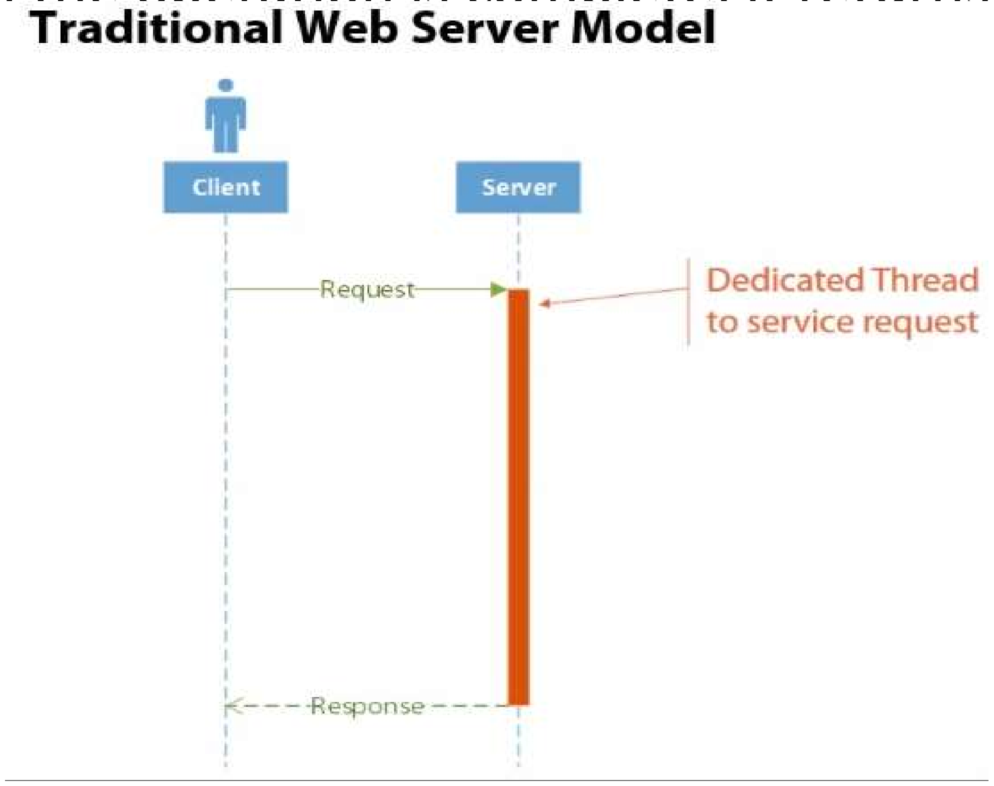
Traditional web servers



- Node.js as a Webserver
- Node is more like a coffee shop, people can come and make a request, but they are started when the request is made.
- Some part of the operation or business logic is executed but for most of the operations its going to use asynchronous callbacks, and hence you have other people waiting for the requests to be fulfilled until you are not consuming the whole thread the entire time the request is made.
- During these spots where non blocking I/O is used for accessing the data from the database, or opening a file, the thread is being freed up for other requests to be handled.
- This is the default behavior in node.js and hence we can achieve some high level of scalability.

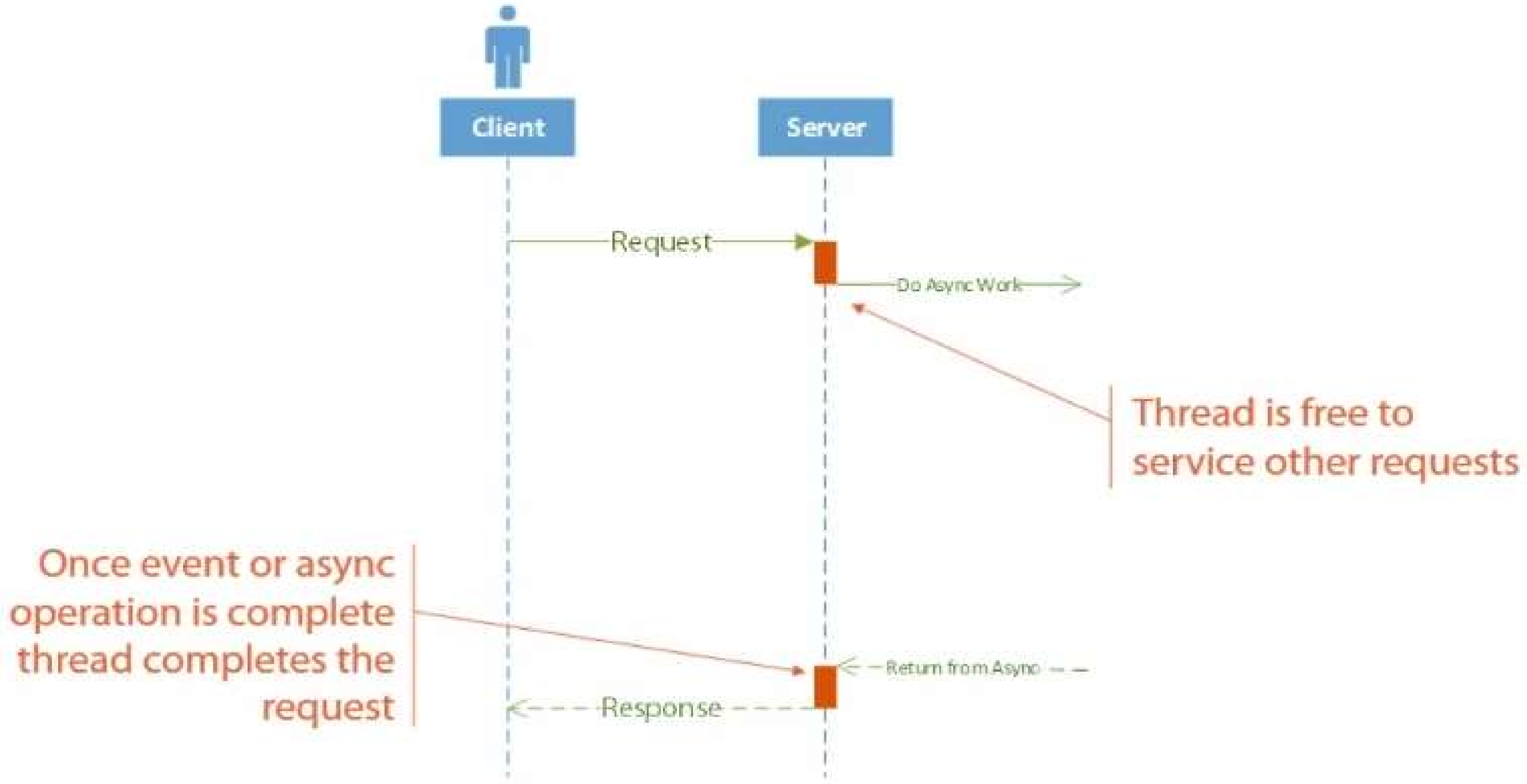

Coffee Shop
Requests are made
Wait for request to be fulfilled

- In traditional Web servers
- In traditional web servers the client makes a request and then while the request is being fulfilled , the server resources is consumed the entire time.
- A request is made and a dedicated thread is used to service that request and when the operation is complete, it returns a response


Traditional Web Server Model
Client
Server
Request
Dedicated Thread to service request
Response

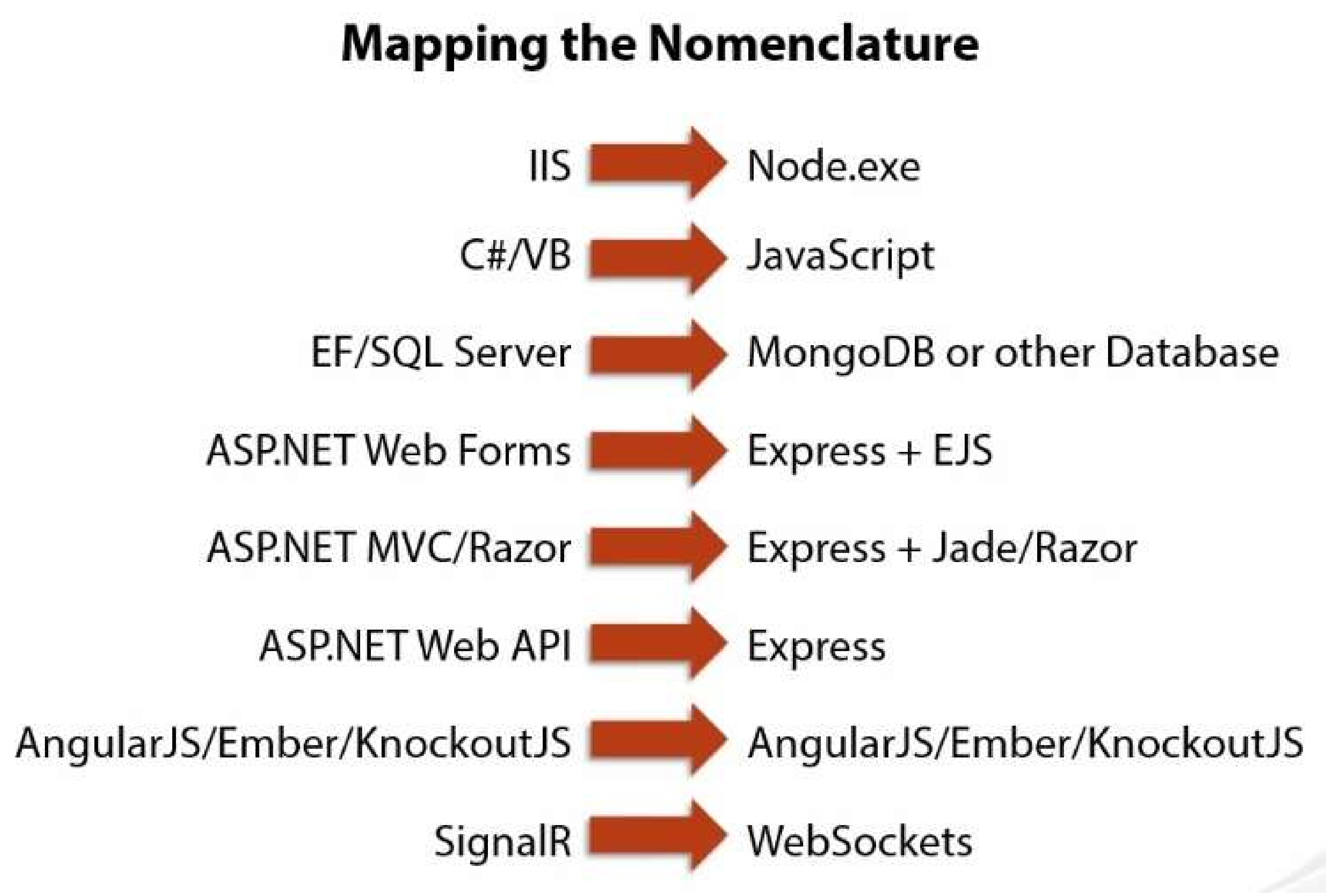
- In node.js or the async of server model, a request is made and that thread is being used while it needs to be actually doing work.
- When it goes to do some async work, when its in wait state , because its opening a file, making a network request, may be getting data from the database, that thread is free to service other requests.
- Once the async operation is complete the event is fired to say that the back half of what needs to be done.
- That is consume that thread for that amount of time and then return the response.
- Hence in this way a single server can handle many more users because the threads are not consumed

Async Web Server Model


Client
Server
Request
Do Async Work
Thread is free to service other requests
Once event or async operation is complete thread completes the request
Return from Async
Response

NODE.JS IS ASYNCHRONOUS BY NATURE

YOU CAN ACCOMPLISH THIS IN ASP.NET BUT IT'S NOT THE DEFAULT BEHAVIOR



Mapping the Nomenclature

IIS	Node.exe
C#/VB	JavaScript
EF/SQL Server	MongoDB or other Database
ASP.NET Web Forms	Express + EJS
ASP.NET MVC/Razor	Express + Jade/Razor
ASP.NET Web API	Express
AngularJS/Ember/KnockoutJS	AngularJS/Ember/KnockoutJS
SignalR	WebSockets

- When to use node
- Node is great for streaming or event-based real-time applications like:
    - Chat applications
    - Game servers
    - Streaming servers
    - Document editing on the cloud
    - Ad servers
- Node is great when you need high level of concurrency but with very little dedicated CPU time
- Great for writing JS everywhere

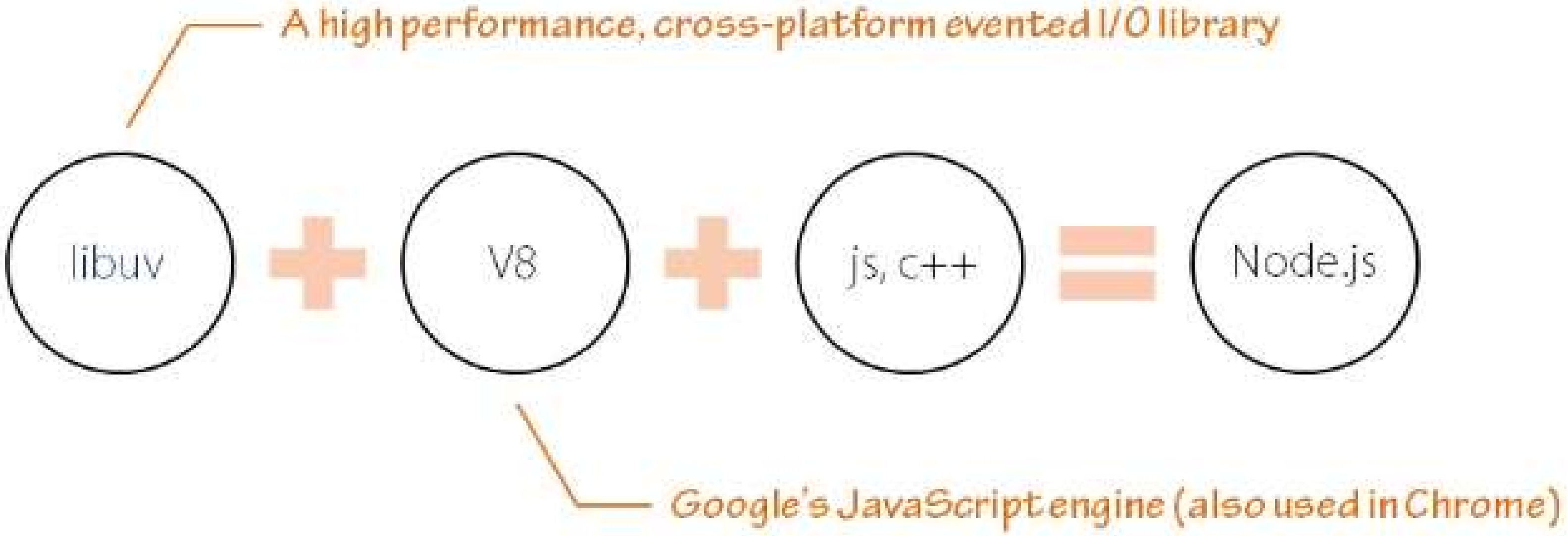
- When node is not the best tool to be used
  - Node is structured to be single threaded.
  - There is an event loop that it runs.
  - So if your application is doing some long running calculations at the backend then you should not use node, because it will be blocking all the other events that are going to come in.
  - If the calculations has started, then the server will not be able to do anything

- Where node is used
  - Microsoft uses node to power up azure mobile services. The backend for mobile services is written in node
  - Cloud9
  - eBay
  - New York times
  - Paypal

- Node Community
  - Introduced in 2009, and node is the third most popular project on github.
  - Over 2 million downloads per month
  - Over 81,000 modules on npm

- At a high level node is comprised of three building blocks.
- 1. **Lib UV** : a high performance , cross-platform evented IO library.
    - This is a new addition to node.
    - It abstracts several UNIX –only library once directly required by the project
    - This was built as a part of porting Node.js to windows environment
- 2. **V8:** This is google’s JS engine, same that is found in chrome browser.
- This is by default leveraged to node
- 3. **Custom C++ and JS code**: This is specifically developed for Node platform itself
- All the above 3 together makes the Node.js platform

Node.js Building Blocks


A high performance, cross-platform evented I/O library
libuv
V8
js, c++
Node.js
Google's JavaScript engine (also used in Chrome)

- Node.JS background

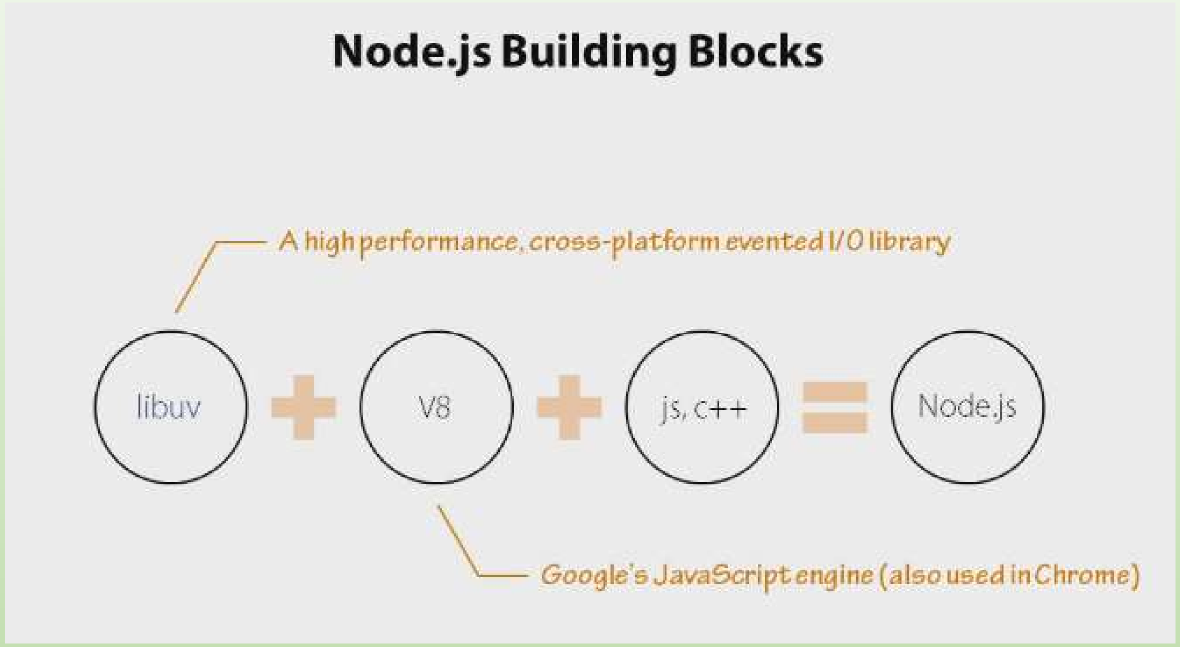
- At a high level, "Node" is comprised of three building blocks.
- The first is "Lib UV", a high performance, cross-platform evented IO library.
- It's a fairly new addition to "Node", and replaces or abstracts several UNIX-only libraries once directly required by the project. "Lib UV" was built as a part of porting "Node.js" to the "Windows" environment.
-

- Node.js Background
- Next is "V8".
- This is Google's "JavaScript" engine, the same engine found in their Chrome web browser.
- The "Node" team makes every effort to leverage "V8" out of the box within "Node".
- This makes it easier for the team to include updated versions of "V8" in each release of "Node", and thereby benefit from Google's continuous innovation of their "JavaScript" engine.

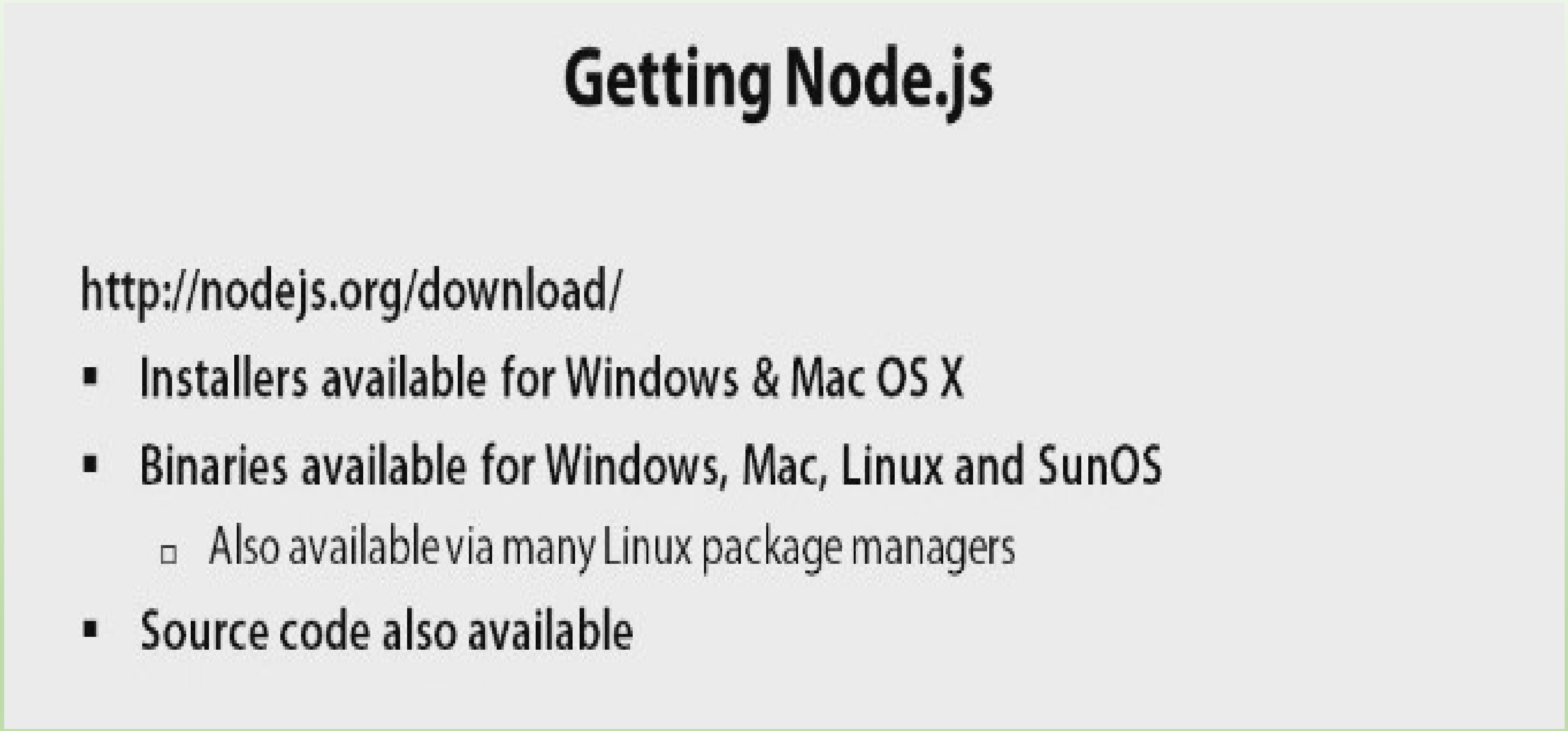
- Node.js Background

- The last component of "Node" is the custom "C++" and "JavaScript" code developed specifically for the "Node" platform itself.
- These three things together make up the "Node.js" platform.

- Node.js Background


Node.js Building Blocks
A high performance, cross-platform evented I/O library
libuv
V8
js, c++
Node.js
Google's JavaScript engine (also used in Chrome)

- Getting node.js

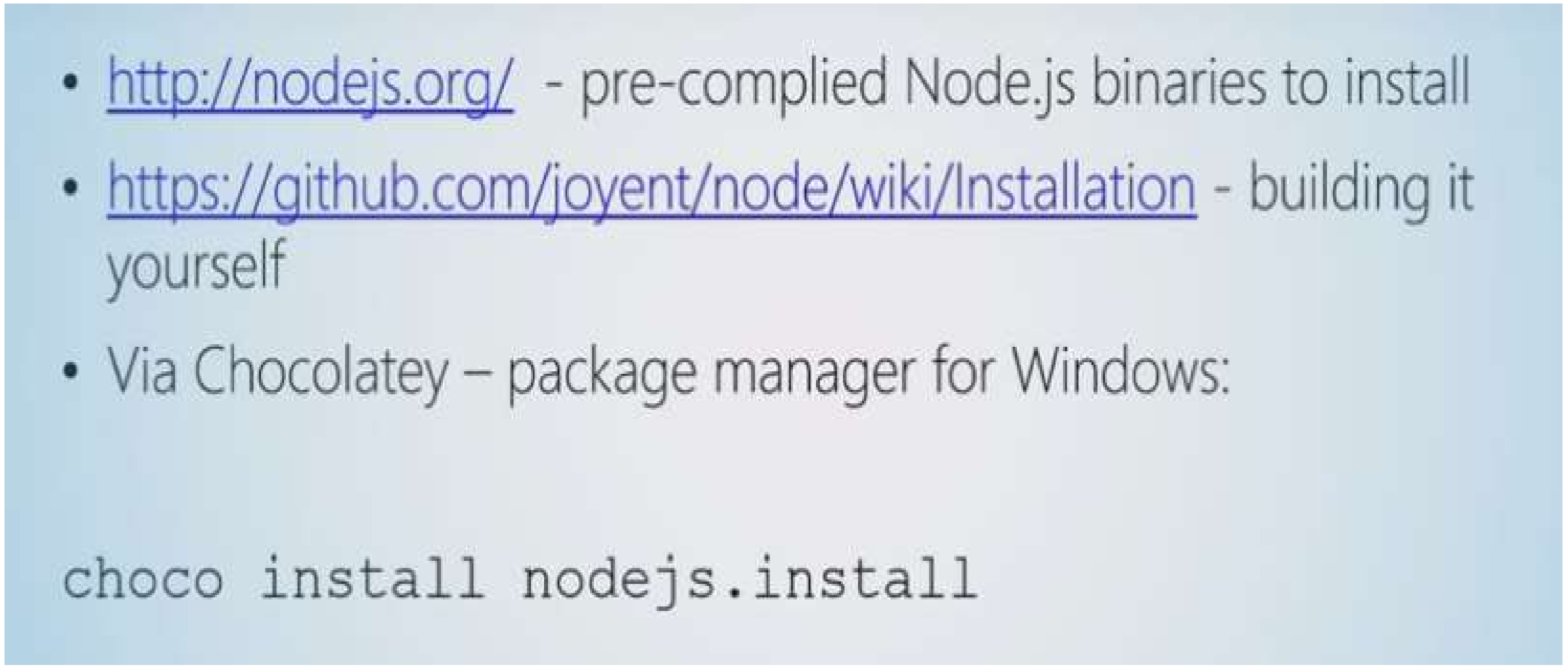


Getting Node.js

http://nodejs.org/download/

- Installers available for Windows & Mac OS X
- Binaries available for Windows, Mac, Linux and SunOS
  - Also available via many Linux package managers
- Source code also available

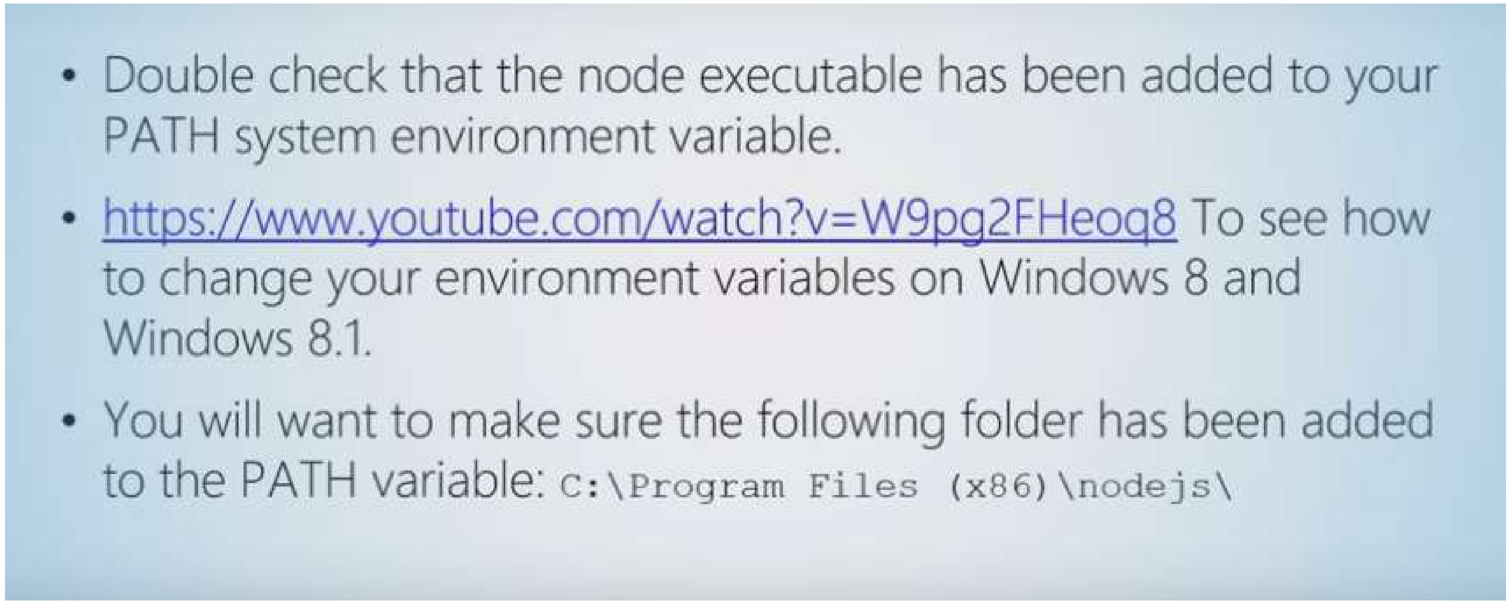
- Setting up the environment



- http://nodejs.org/ - pre-complied Node.js binaries to install
- https://github.com/joyent/node/wiki/Installation - building it yourself
- Via Chocolatey – package manager for Windows:

```
choco install nodejs.install
```

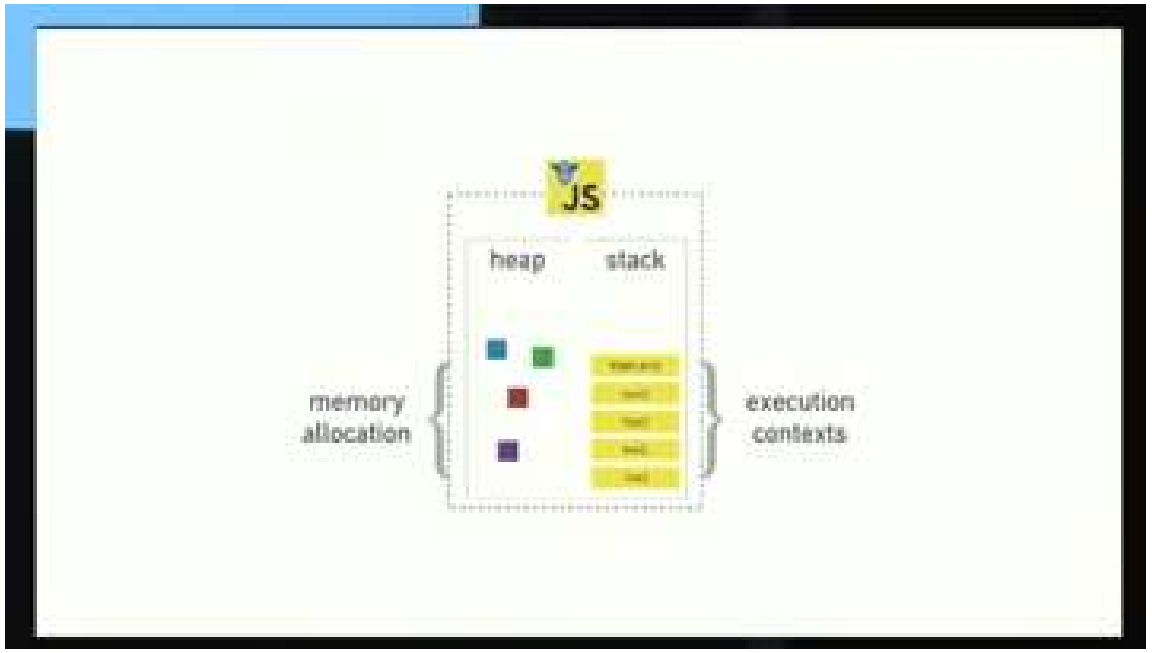
C:\Program Files (x86)\nodejs\;



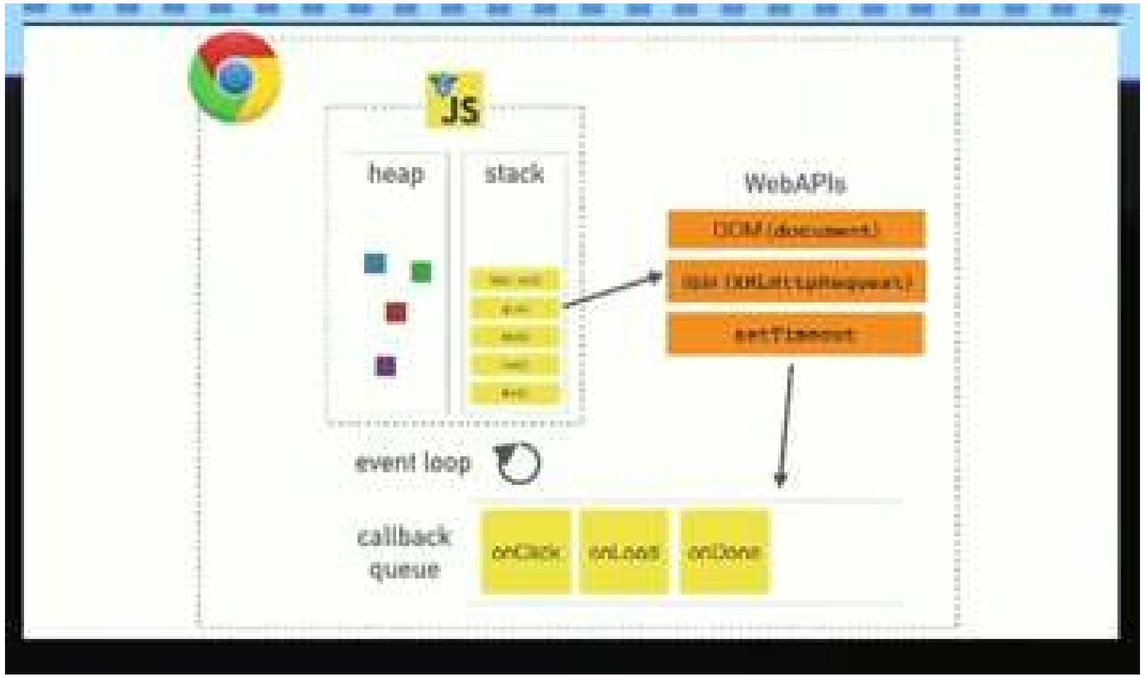
- Double check that the node executable has been added to your PATH system environment variable.
- https://www.youtube.com/watch?v=W9pg2FHeoq8 To see how to change your environment variables on Windows 8 and Windows 8.1.
- You will want to make sure the following folder has been added to the PATH variable: `C:\Program Files (x86)\nodejs\`

- Node’s Event Loop

- The figure shows a simplified view of what JS runtime is.
- The heap where memory allocation happens
- The stack (call stack) where the .
- If V8 is cloned and if we grab things like setTimeOut(),DOM or HTTP request they don’t exists in V8 source.



- Node’s Event Loop



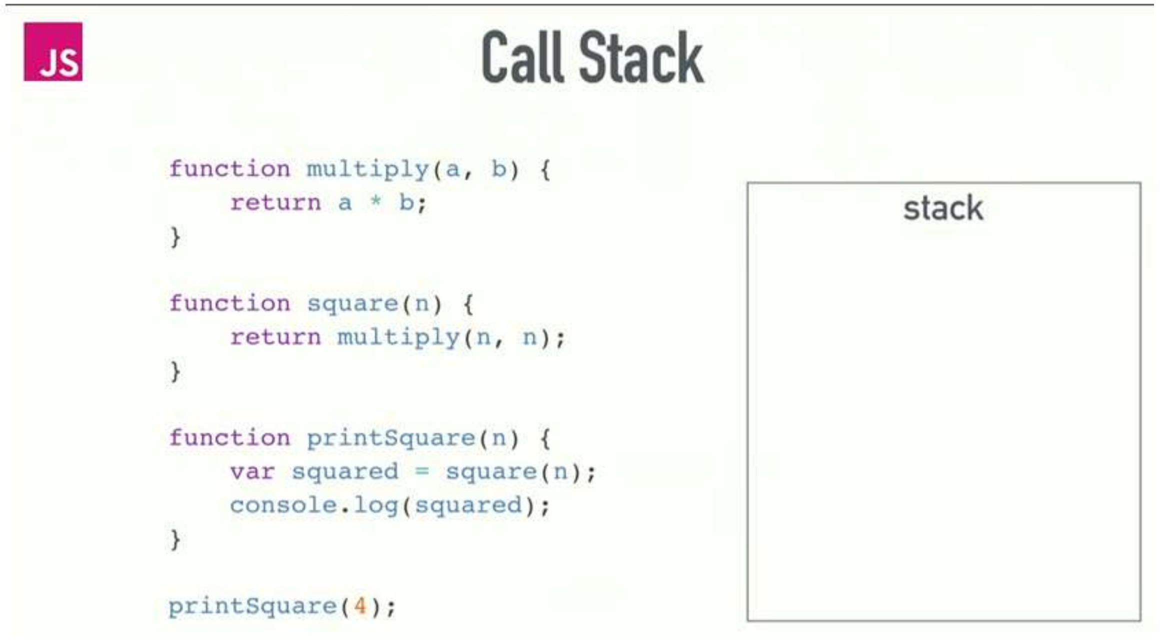
- Node’s Event Loop

the call stack

one thread == one call stack == one thing at a time

- JS is a single threaded programming language.
- This means it has a single call stack
- JS can do only one thing at a time.
- The call stack is part of V8.
- Call stack is a really simple data structure that keeps track of program execution inside the V8.
- In this data structure you can do two things.
  - 1.) add something at the top of it
  - 2) Remove the top most item from it.
- That is how call stack work, like the can of PRINGLES.
- The call stack can execute only one job at a time

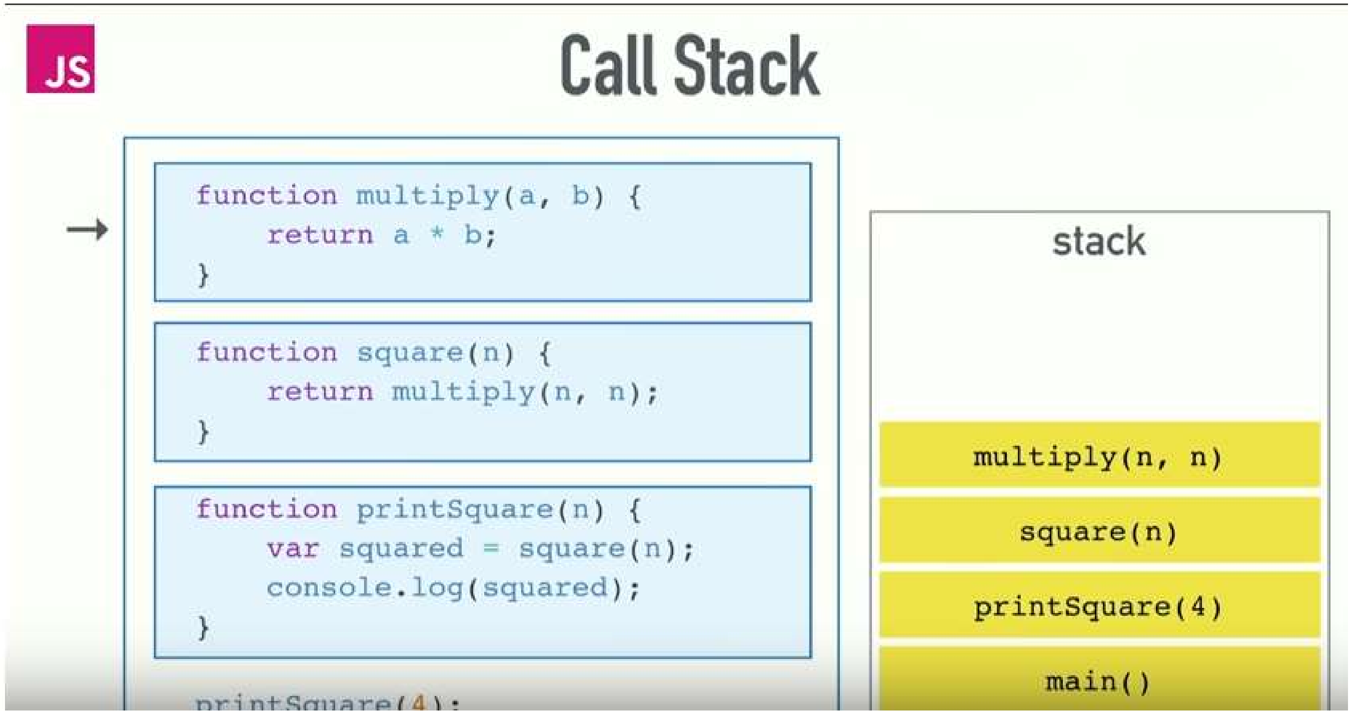
- Node’s Event Loop



```
function multiply(a, b) {
    return a * b;
}

function square(n) {
    return multiply(n, n);
}

function printSquare(n) {
    var squared = square(n);
    console.log(squared);
}

printSquare(4);
```

- Let us visualize what it means by JS being single threaded
- If you execute the adjusant code.
- A call stack is a data structure which records where in the program we are.
- That means when we step into the function we put something on the stack.
- If we return from a function we pop off from the top of the stack.
- So if the code is executed, the main function will be pushed into the stack
- printSquare() function is then pushed on the stack.
- Then the square(n) function is into the stack followed by multiply(a,b)

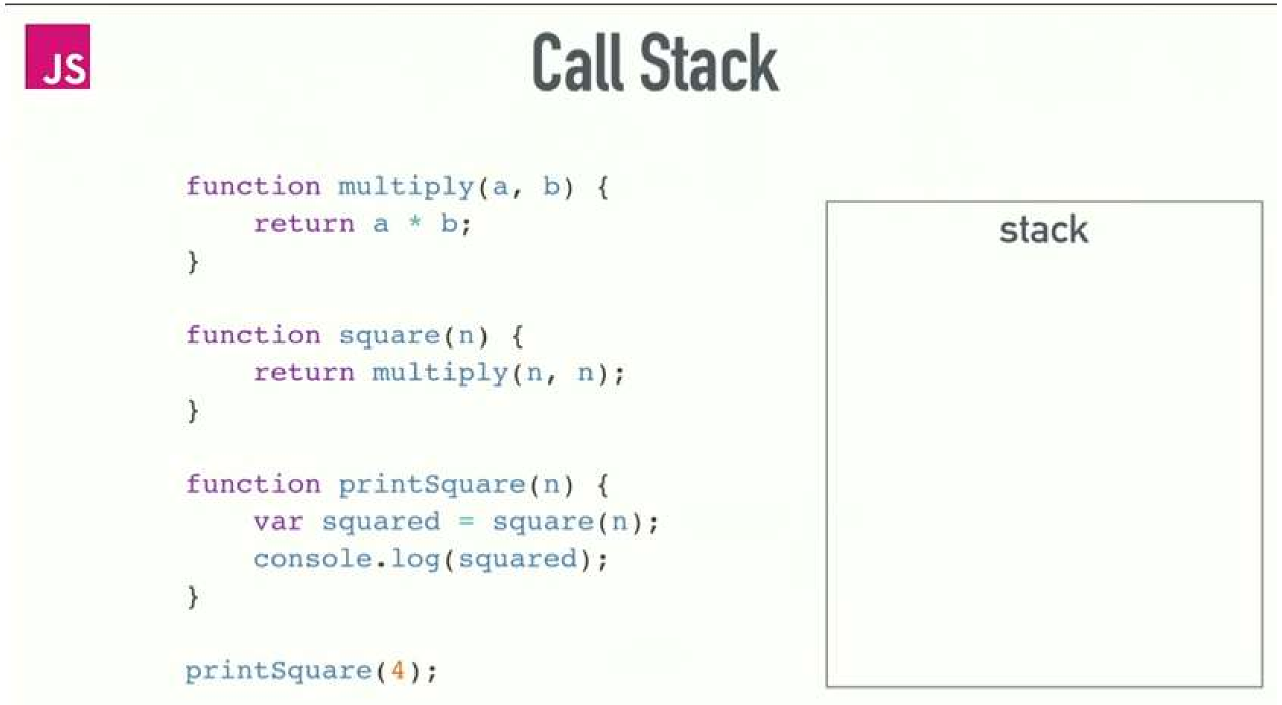
- Node's Event Loop



```
function multiply(a, b) {
    return a * b;
}

function square(n) {
    return multiply(n, n);
}

function printSquare(n) {
    var squared = square(n);
    console.log(squared);
}

printSquare(4);
```

- Functions pushed into the stack in order of execution starting with main()

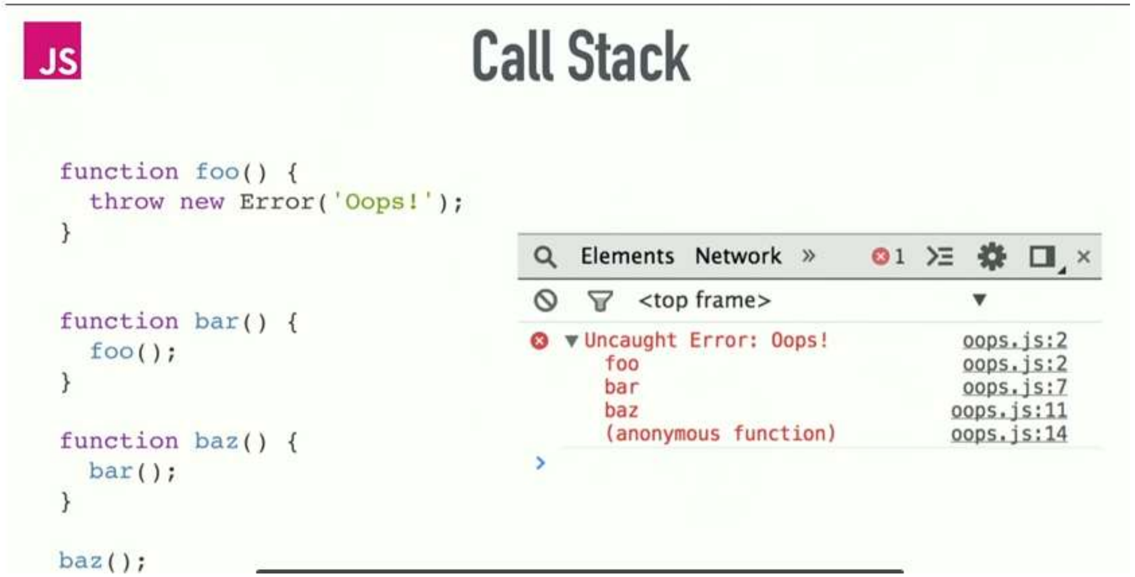
- Node’s Event Loop



```
function multiply(a, b) {
    return a * b;
}

function square(n) {
    return multiply(n, n);
}

function printSquare(n) {
    var squared = square(n);
    console.log(squared);
}

printSquare(4);
```

- Now as the functions are executed they are popped out of the stack in order
- First multiply(a,b) will be popped followed by square(n), then by printSquare(n) and then main().
- This is how we can visualize the call stack

- Node’s Event Loop



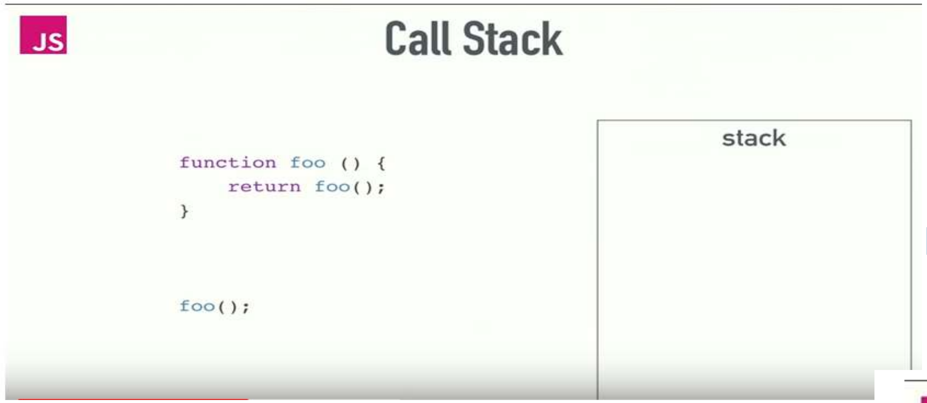
```
function foo() {
  throw new Error('Oops!');
}


function bar() {
  foo();
}

function baz() {
  bar();
}

baz();
```

```
▼ Uncaught Error: Oops!            oops.js:2
    foo                            oops.js:2
    bar                            oops.js:7
    baz                           oops.js:11
    (anonymous function)          oops.js:14
>
```

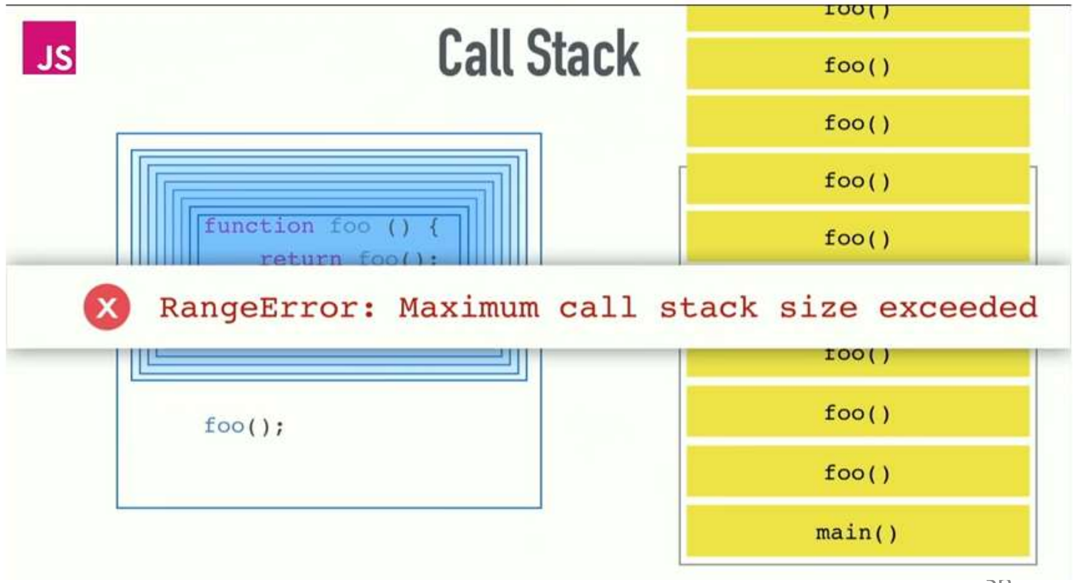
- For example if you have the code like the screen shot baz()→bar()→foo() will be executed in that order and the output will see on the console tab.
- The whole execution call stack will be displayed.

• Node’s Event Loop

- Now the term blowing the stack is as below



```
function foo () {
    return foo();
}

foo();
```

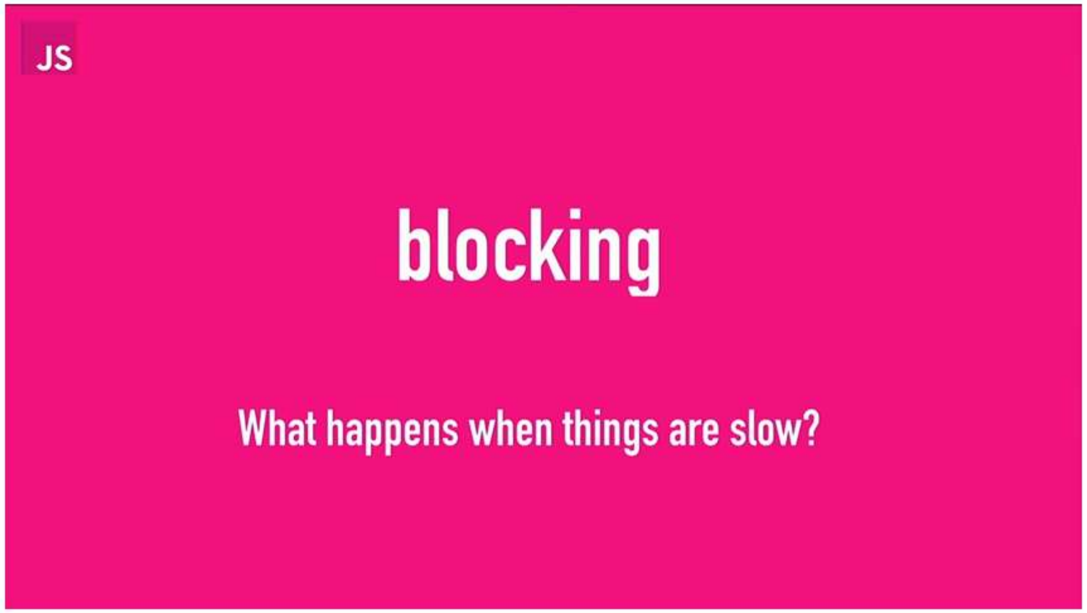


```
function foo () {
    return foo();

foo();
```

```
RangeError: Maximum call stack size exceeded
```

So this is the explanation of call stack

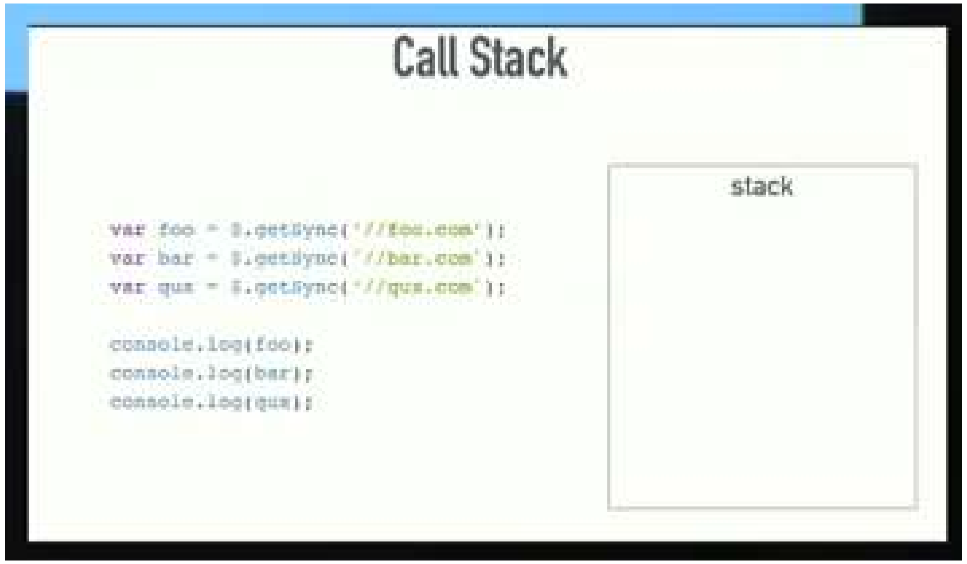
- Node’s Event Loop


JS

blocking

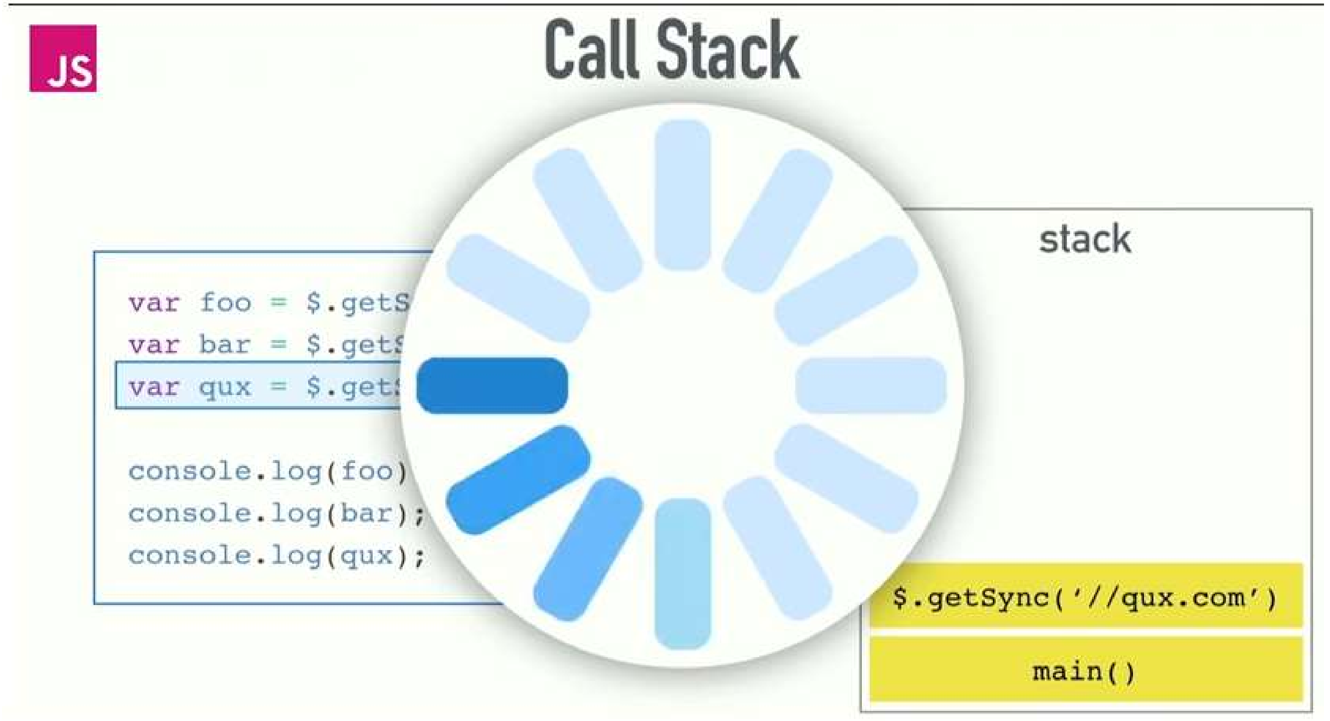
What happens when things are slow?

Blocking means something that is slow
Console.log() is not slow
Doing a while loop is slow, image requests is slow.
So things that are slow and are up in the stack is what blocking means.
Lets take an example of synchronous requests as below


Call Stack

```
var foo = $.getSync('//foo.com');
var bar = $.getSync('//bar.com');
var qux = $.getSync('//qux.com');

console.log(foo);
console.log(bar);
console.log(qux);
```

- Node’s Event Loop



```
var foo = $.getS
var bar = $.getS
var qux = $.getS

console.log(foo)
console.log(bar);
console.log(qux);
```

```
$.getSync('//qux.com')
main()
```

- So for every sync request to execute we need to wait till the previous request is completed.
- So in a programming language that is single threaded this is how we wait till the previous request is completed.
- This problem is because we execute JS code on the browsers.

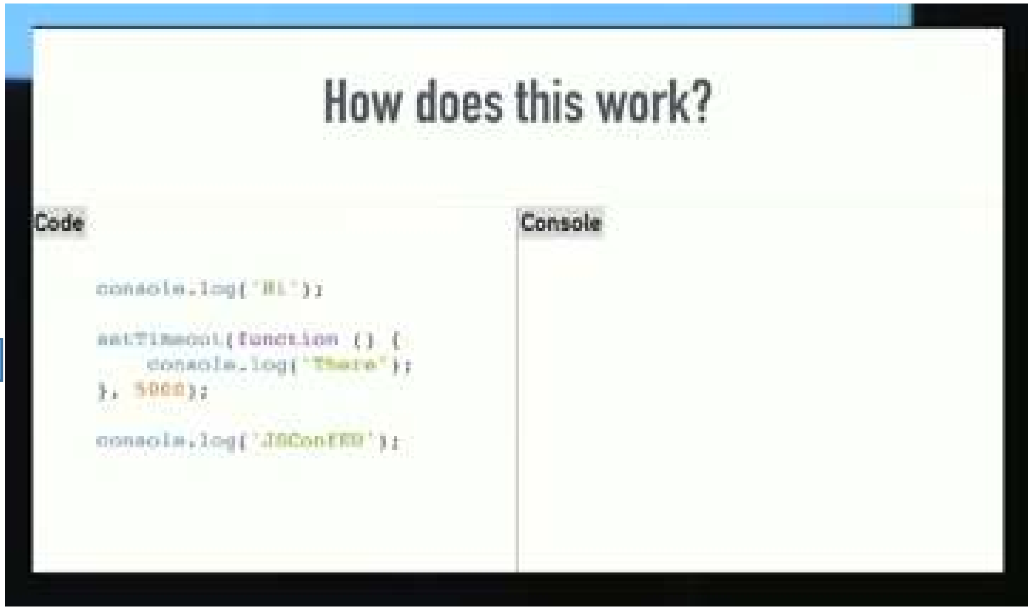
▪ So in real time situation if a lot of long running sync code is executed by the browser, any action done on the UI (like mouse click or button click)will be blocked till the call stack is busy.
▪ To avoid this situation we use asynchronous programming.

- Node’s Event Loop

- To stop the blocking one solution is async callbacks.
- With async callbacks there is no blocking functions in the browser or in node.
- This async callbacks basically means we run some code, give it a callback, and run that later.
- The callback code actually looks as below.

the solution?
asynchronous callbacks

Here’s a function.
Call me maybe?

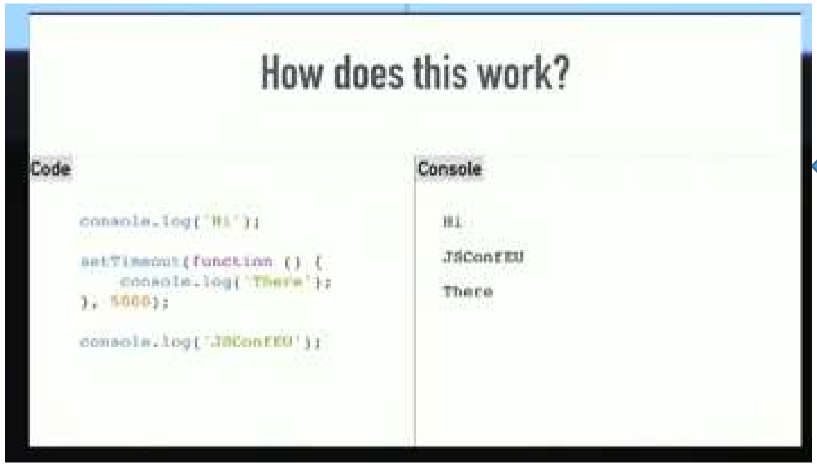


How does this work?

Code

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

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

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

Console



How does this work?

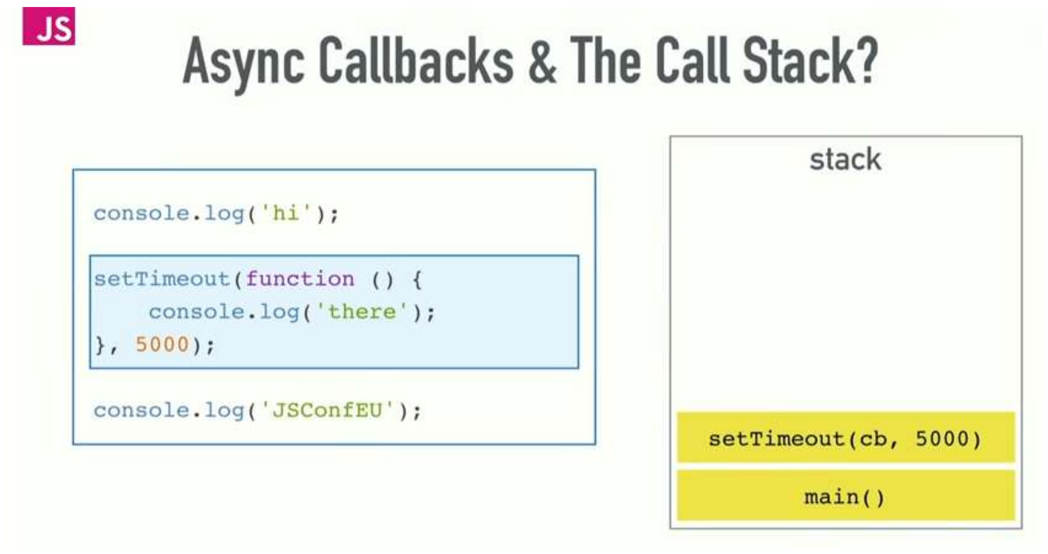
Code

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

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

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

Console

```
Hi
JSConfEU
There
```

- Node’s Event Loop



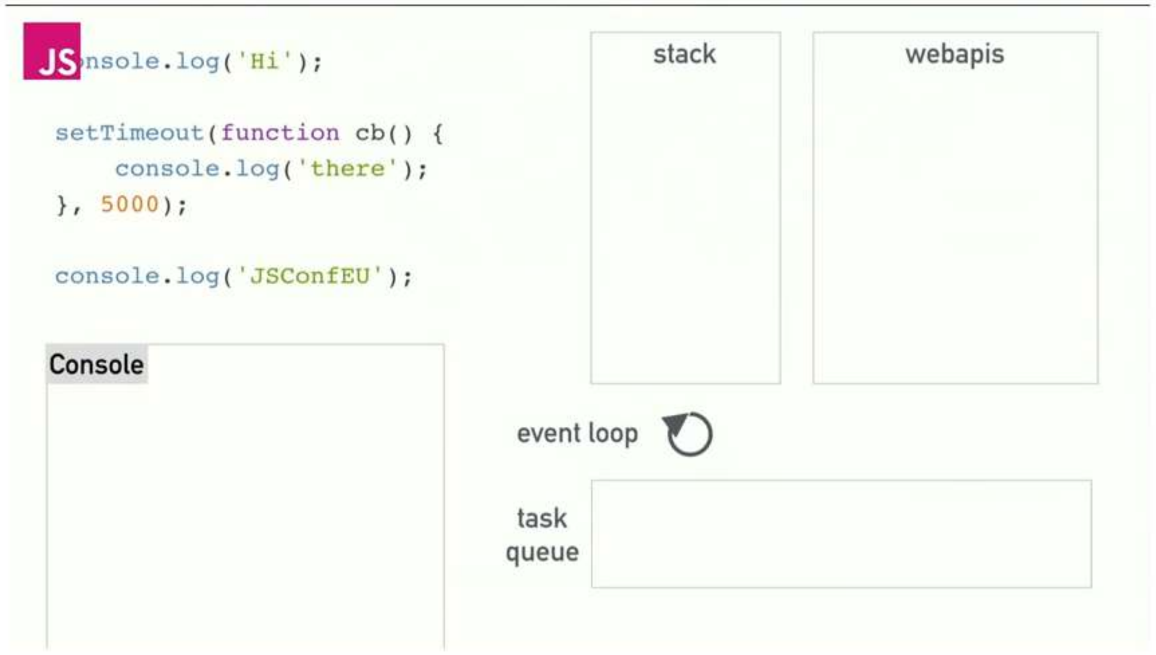
```
console.log('hi');

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

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

- So how does the Async callbacks and the call sack work?
- First the main() method is pushed into the stack
- Next the setTimeOut() function is pushed
- The setTimeOut() function is taking a while to execute and hence it is somehow pushed out of the stack
- Now console.log(‘hi’) will get executed.
- And now five seconds later some how the setTimeOut() function will be pushed back into the stack and will be executed.
- This style of execution with async callbacks happens due **to Concurrency and the Event loop**

- Node’s Event Loop

- As mentioned earlier JS can do only one thing at a time.
- The reason we can do things concurrently is that the browser is more than just the runtime.
- The browsers gives us what we call the API’s(diagram shown in prev slides)

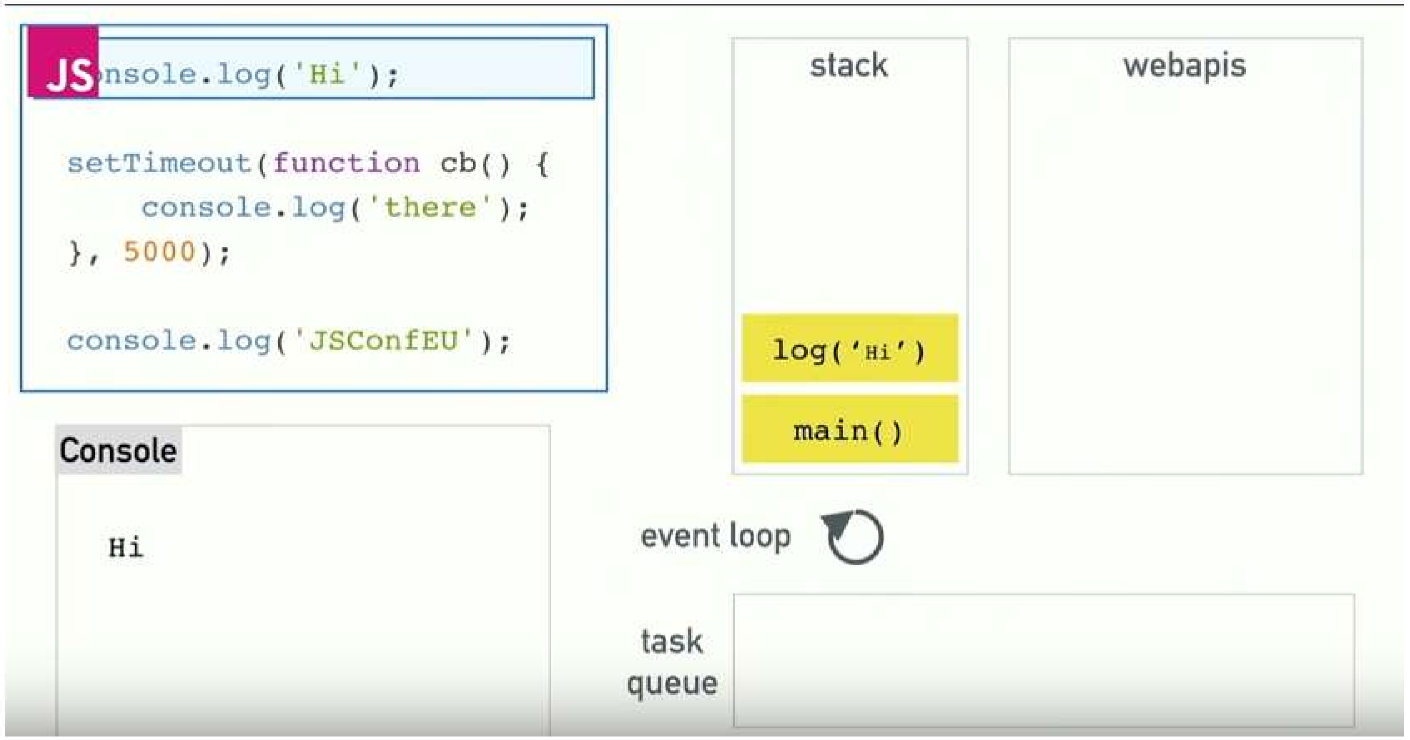
- Event Loop



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

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

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

- So let us see how this code will get executed with the C++ api (when it is node) or with WebApi(when it is browser)
-

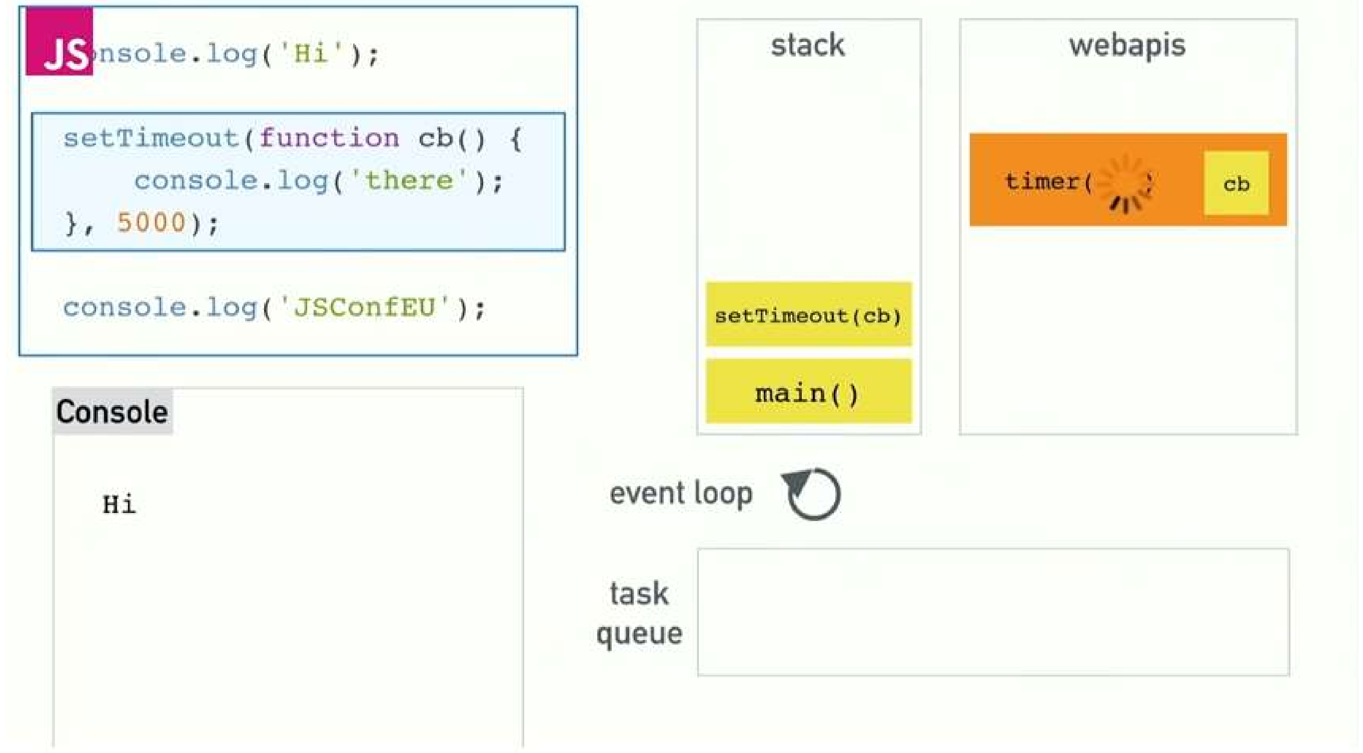
- Event Loop



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

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

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

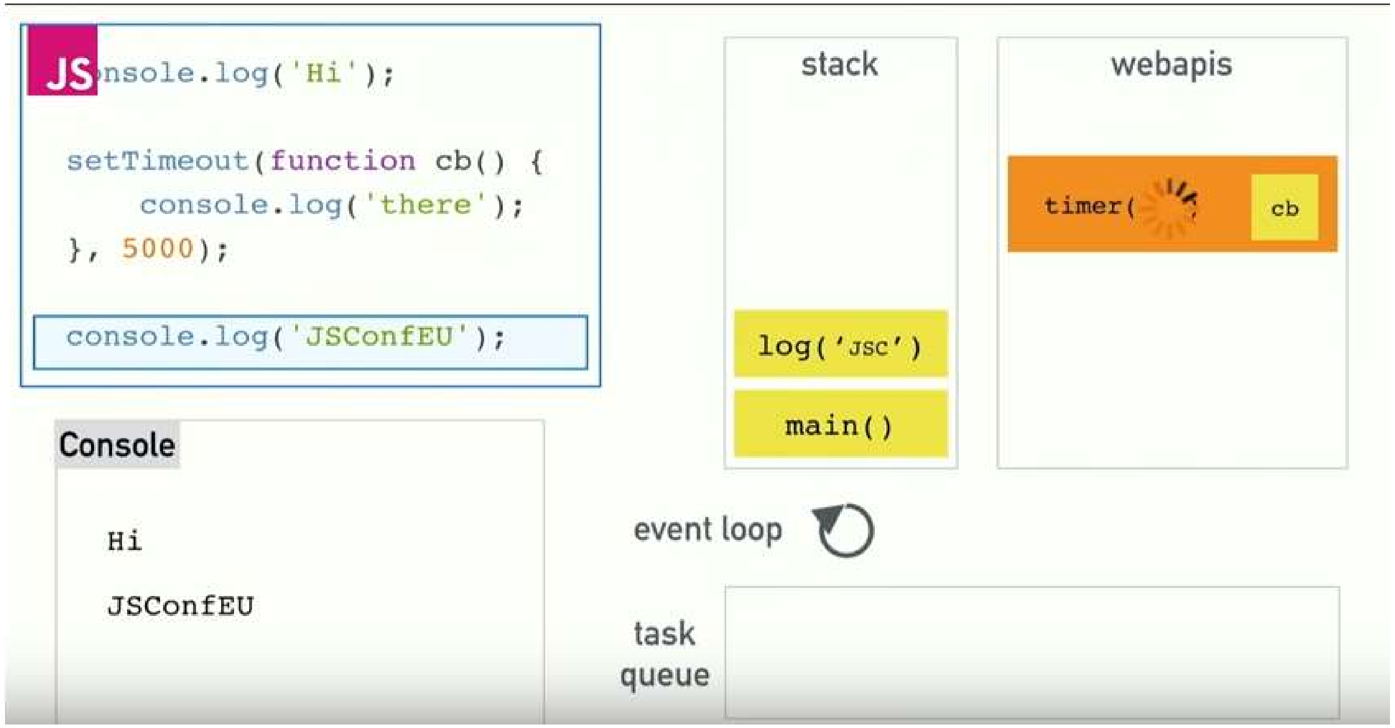
- Step 1:
- Main() method is pushed into the call stack
- Console.log('Hi') gets executed

• Event Loop


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

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

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

- STEP 2:
- setTimeOut is an api provided to us by the browser.
- It does not live in the V8 source
- So the browser is going to take off the timer and handle the count down
- Since setTimeOut callback is not complete it is popped out of the stack

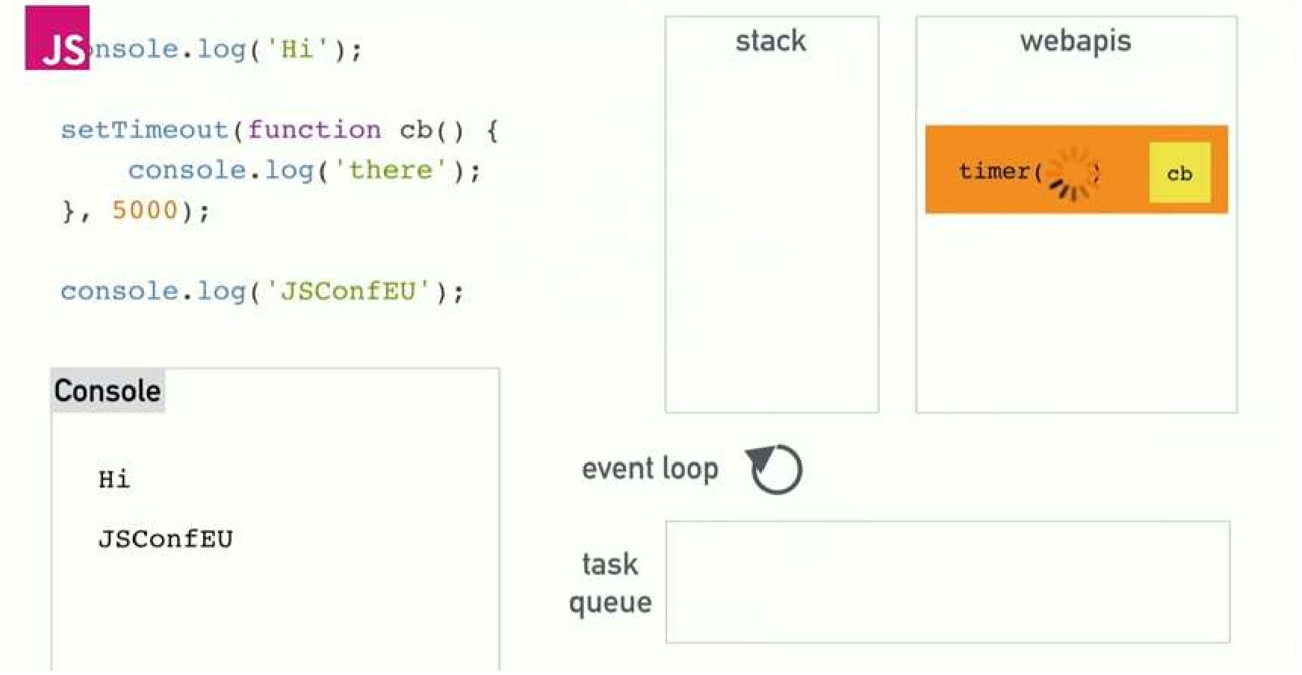
- Event Loop


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

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

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

```
Hi

JSConfEU
```

STEP 3:
While the webapi's is executing the timer the console.log('JSC') is pushed into the call stack.

- Event Loop

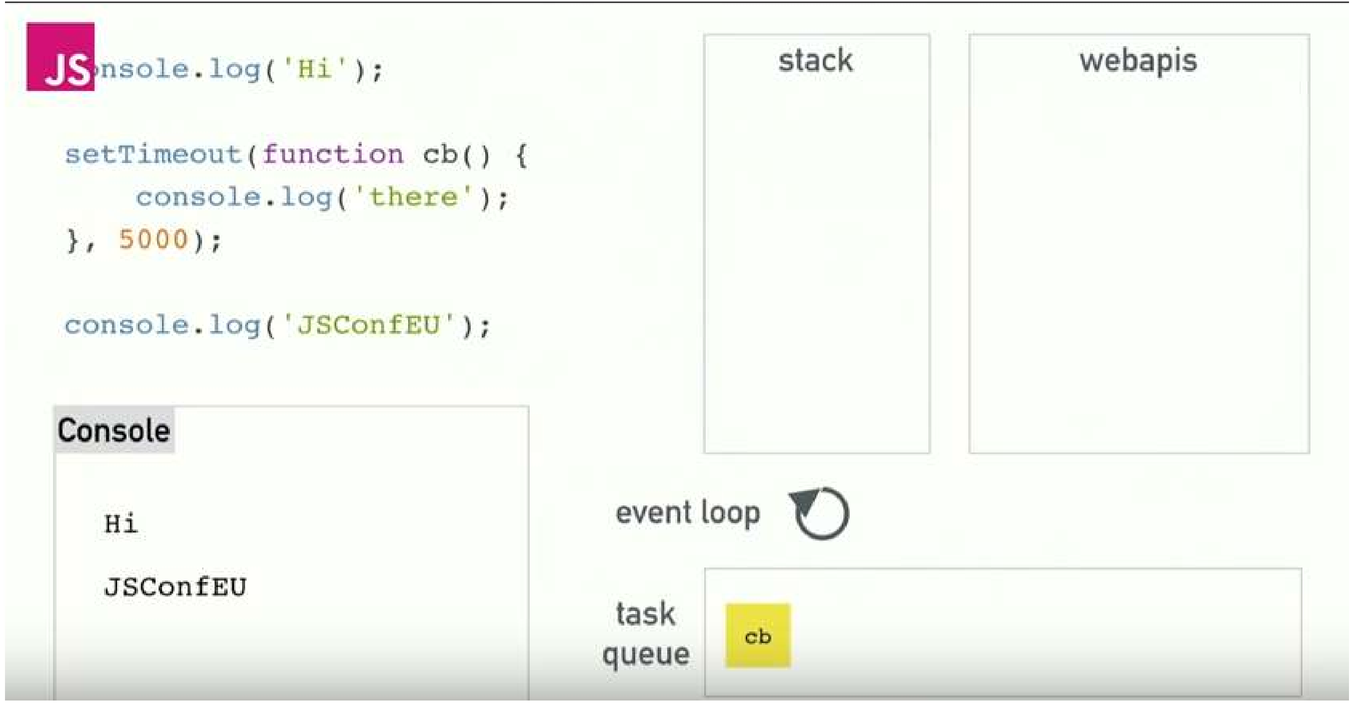


```
console.log('Hi');

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

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

```
Hi

JSConfEU
```

- STEP 4:
- The timer in the webapi is going to complete setTimeOut in 5secs
- The webapi can’t sent the code to the stack and this is where the task queue kicks in.
- So once the timer is complete the webapi pushes the callback to the task queue

- Event Loop

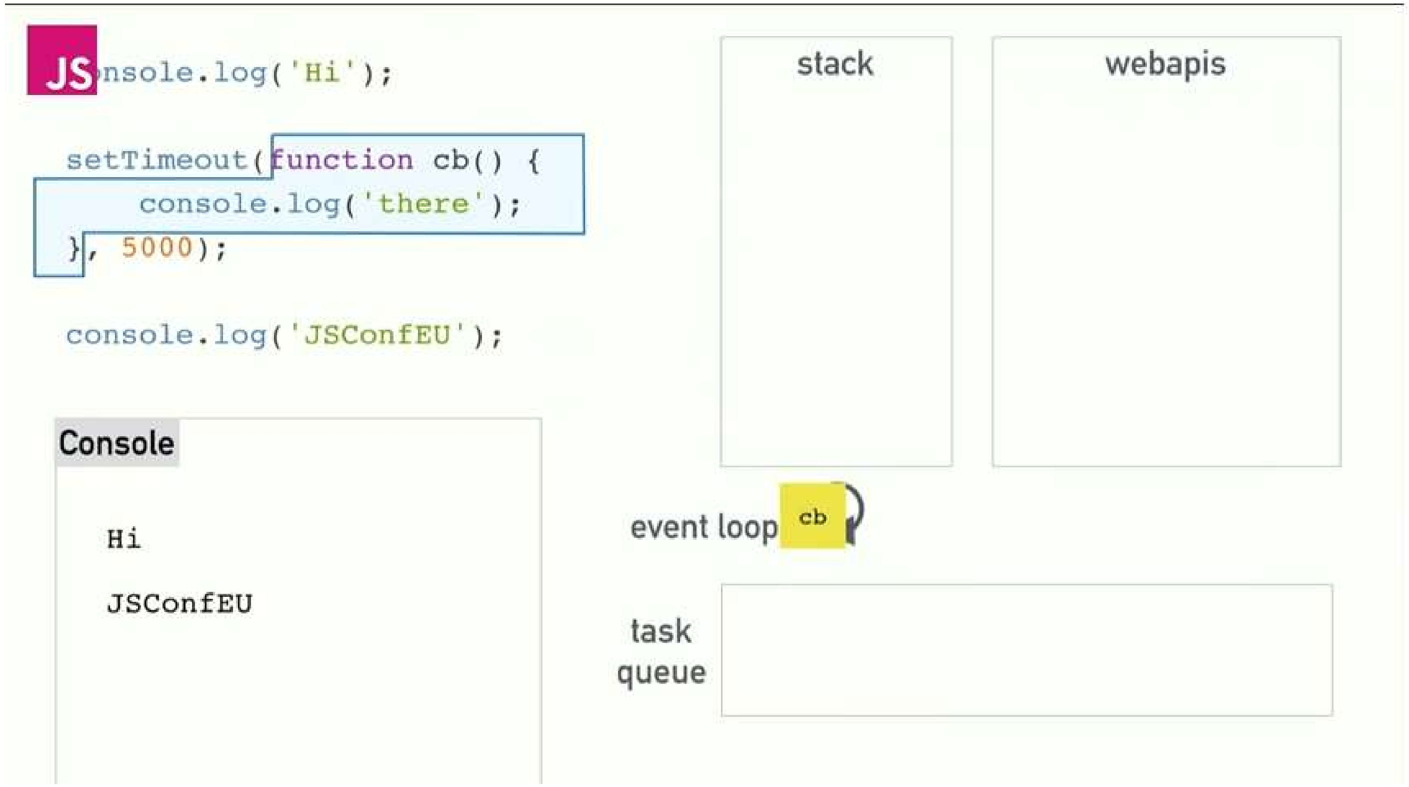


```
console.log('Hi');

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

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

```
Hi

JSConfEU
```

- STEP5:
- Webapi pushes the call back to the task queue.
- And then comes the very simple eventloop with one simple job.
- The job of event loop is to look at the stack and also at the task queue.
- If the stack is empty it takes the first thing on the queue and pushes it to the stack

- Event Loop



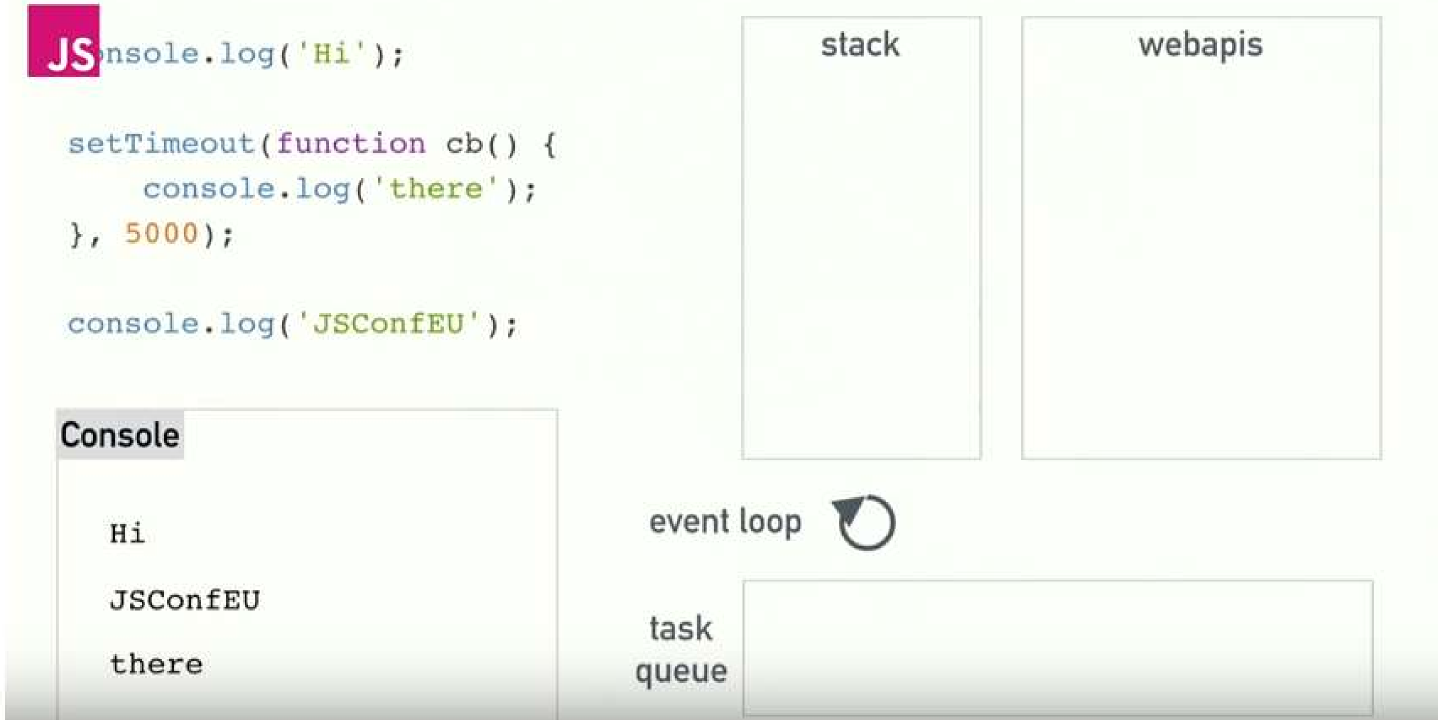
```
console.log('Hi');

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

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

Console

```
Hi

JSConfEU
```

- STEP 6:
- So now the stack is empty and there is cb in the queue hence event loop pushes the call back to the stack.
- Now the cb is back into the V8 and the output is printed on the console.

• Event Loop



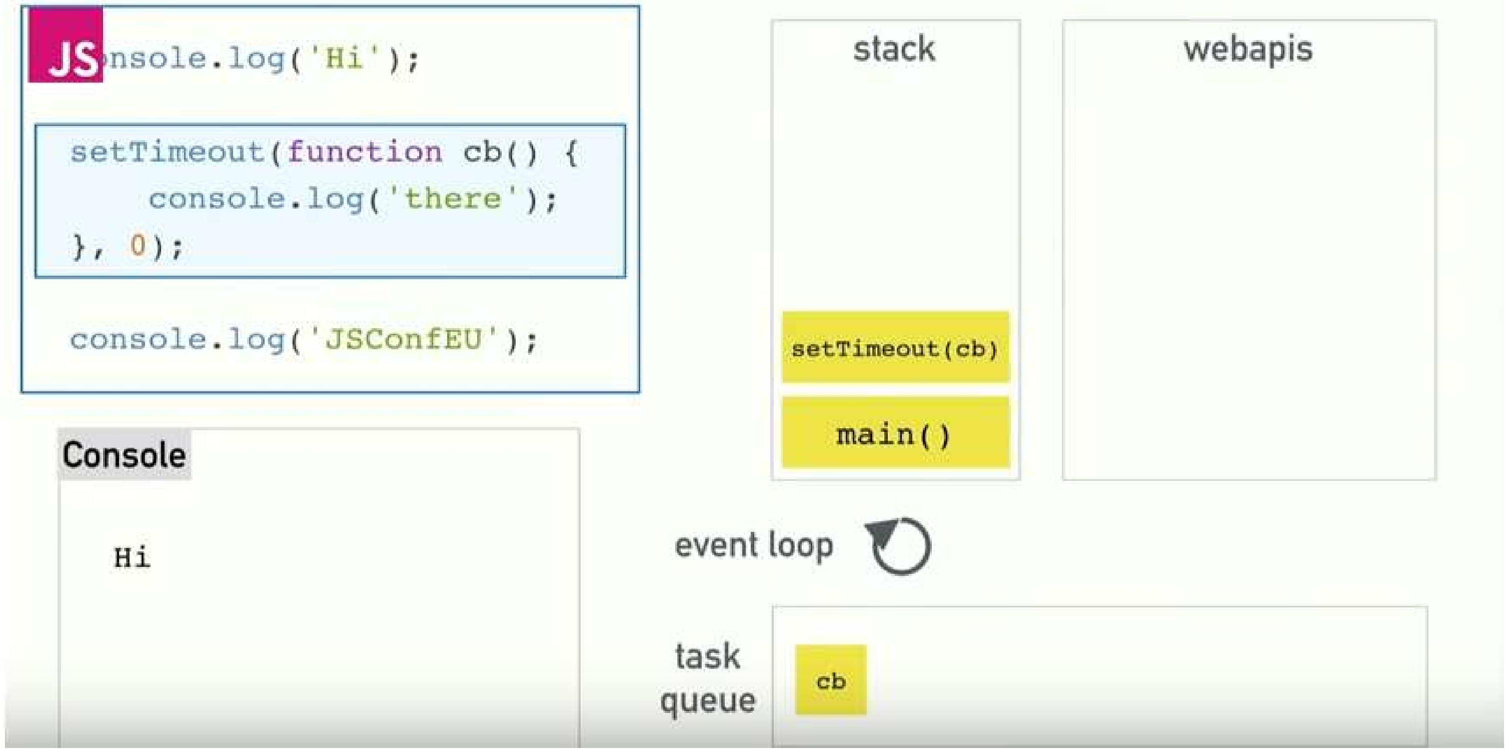
```
console.log('Hi');

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

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

Console

```
Hi

JSConfEU

there
```

- STEP 7:
- The output is shown on the console →''there!"

- Event Loop



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

setTimeout(function cb() {
    console.log('there');
}, 0);

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

- In this example the setTimeOut() cb has a time of 0 milli seconds.
- In such a case the setTimeOut() function still gets executed by the browser apis and immedieately returns to the task queue.
- The event loop will not push the cb to

- Node’s event loop
- One of the key concepts "Node" brings from the browser to "JavaScript" on the server is the "Event Loop".
- In the browser, the "Event Loop" is constantly listening for DOM events, so just key presses or mouse clicks.
- Similarly, Node's "Event Loop" is constantly listening for events on the server side.
- incoming HTTP requests or TCP connections, or they can be timers and other internal events generated by "Node" application itself.
- Additionally, other events may be triggered on the response to a request against an external resource.
- For example, asking "Node" to open a file for reading will fire an event when the file is opened and ready. Sending a message to an external process will fire an event when the message has been sent.
- And making a request of a network resource, such as another web server, will fire an event when the HTTP response is received
- A key point is that each of these are handled as discrete events in "Node

- Node’s Event Loop
- Node provides event loop as part of the language
- With node, there is no call to start the loop
- The loop starts and doesn’t end until the last call back is complete.
- Event loop is run under a single thread therefore sleep() makes everything halt

- Node’s Event Loop
- The events will very likely interleave each other.
- For example, in the diagram, a timer event is received between the request for a file and when the file is ready for reading, and both TCP and HTTP events are received while we're sending a message to an external process.
- "Node" itself doesn't pause and wait for any of these requests to complete.
- It simply continues to react to events as they arrive.

- Node’s Event Loop…

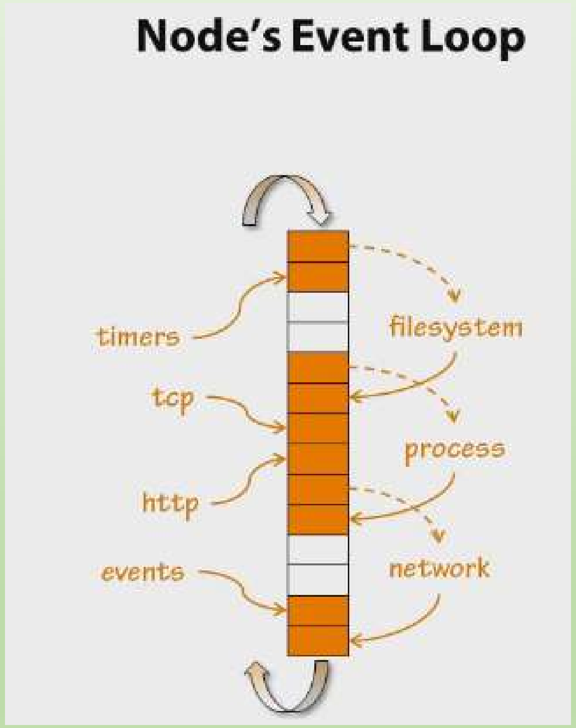
A common example to demonstrate this non-blocking, event-driven approach is a web application that fetches data from a data base.

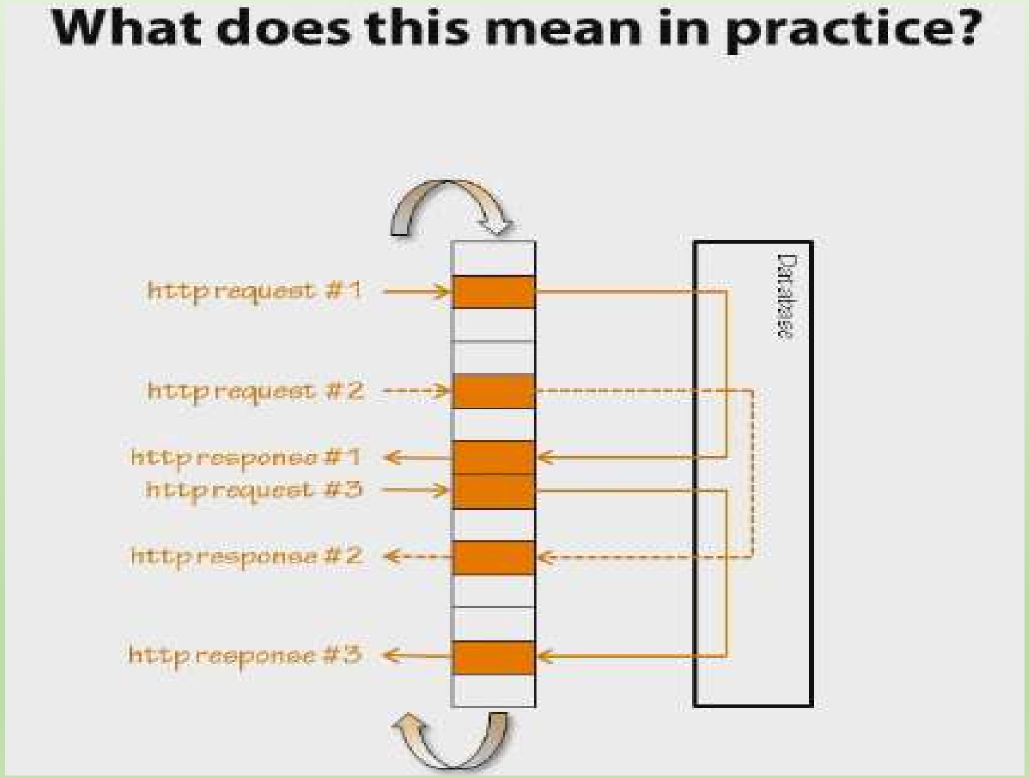
The application raises an event when an HTTP request is received.

This event generates a query to the data base for some information

- Node’s event loop
- Once "Node" receives an event back from the data base that the query is complete, an HTTP response is formulated and sent to the caller.
- While it is waiting for the response from the data base, "Node" is not blocked and is free to handle additional requests.
- This non-blocking approach is fundamental to "Node", and differentiates it from the more traditional, server-side programming model that requires you to manage multiple threads to achieve this type of concurrency.

- Nodes event loop

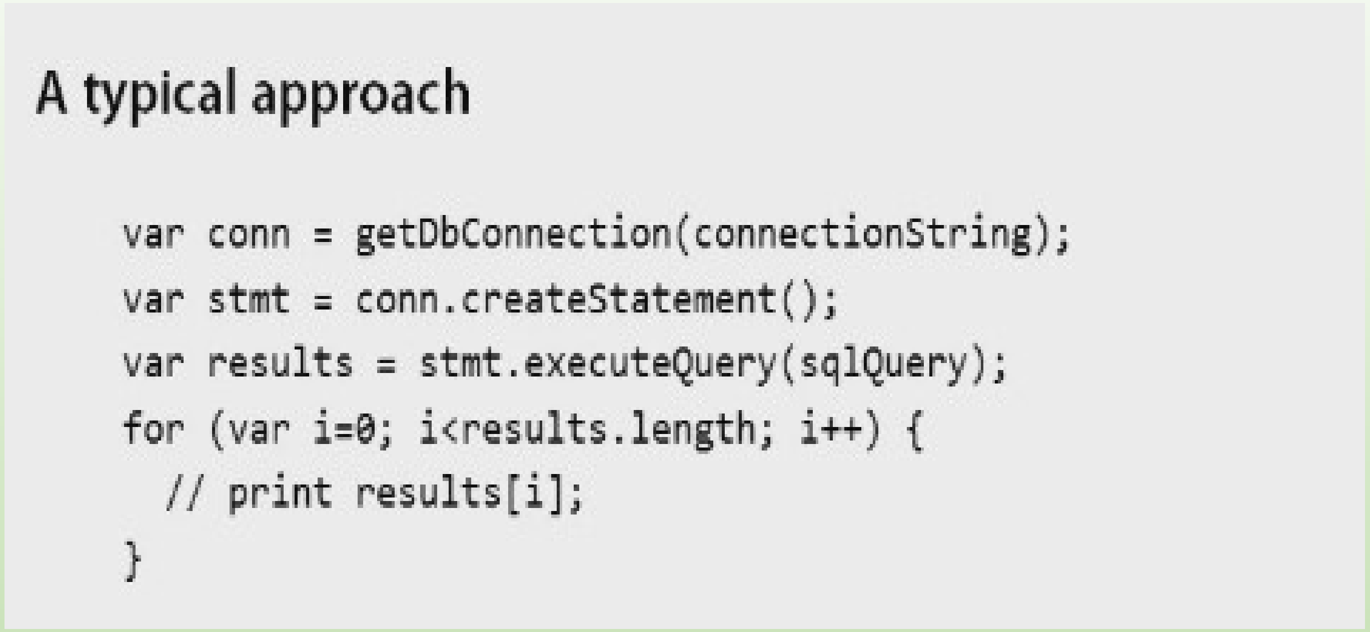




- Event Driven Programming (EDP)
- It is programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key presses) or messages from other programs.
- It's a different way of program flow.
- The flow of the program is defined by the events that are taking place.
- The flow of program is not linear but it is split based on the n/w events or the GUI events.
- If multiple events occur at the same time they the event driven programming knows how to handle that simultaneously.
- The server code is an example of EDP

- Node Conventions for Writing Asynchronous Code
- Writing code that operates in this non-blocking
- , asynchronous environment requires a different way of thinking.
- Here is a typical approach to querying a data base, shown using some JavaScript-
- looking pseudo code. Each function returns a value before the next function is called.

- Each statement builds on the results of the prior one. First,
- we connect to a data base.
- Then we use that connection to create a statement.
- From that statement, we execute a query and get back a set of results. Finally, we iterate over those results.



A typical approach

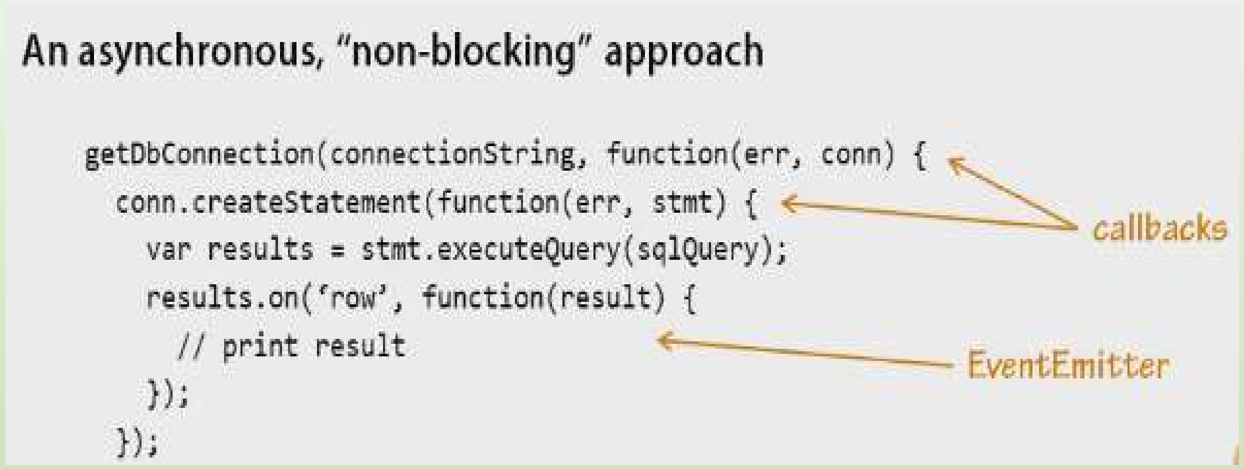
```
var conn = getDbConnection(connectionString);
var stmt = conn.createStatement();
var results = stmt.executeQuery(sqlQuery);
for (var i=0; i<results.length; i++) {
  // print results[i];
}
```

- "Node.js" code that accomplishes this task would look quite different in this case because each function returns almost immediately, before the actual work has been done. We have to find other ways to convey the ordering of our statements.
- If you look at the "Get DB" connection function, you'll see in this case it takes two parameters.
- The first is the same connection string as before, but the second parameter is a function.
- What we're saying, in effect, is, "Get a connection to the data base, and once you have it, call this function and pass it the connection you just created." By crafting the statement this way,
- we've left "Node" free to do other work while it's waiting for the data base connection to be established.
-

- The "Create Statement" function is written similarly, except in this case the only parameter is the function to call once the statement has been created.
- These two functions are examples of using call-backs to write code that will run asynchronously.
- However, lest you think functions with return values have disappeared altogether,
- you'll notice that the "Execute Query" function does indeed return a value. In this case,
- the results object returned from the function does not immediately contain the results of the data base query.
- is a special object called an "Event Emitter", which is capable of emitting events in the future, when each row of the query result becomes available.

- Here we're telling "Node" to invoke a function when each row event is omitted by the results object.
- "Node" has adopted some conventions around its use of call-backs,
- Node’s convention is that the call-back parameter is always the last parameter passed to the asynchronous function.
- Another convention around the use of call-backs is error handling.
- The first value passed to the call-back should always be an error parameter. strictly abiding by this approach will leave you with many, many functions that are only used once in your code. For simple call-backs, or those that are only referenced once,

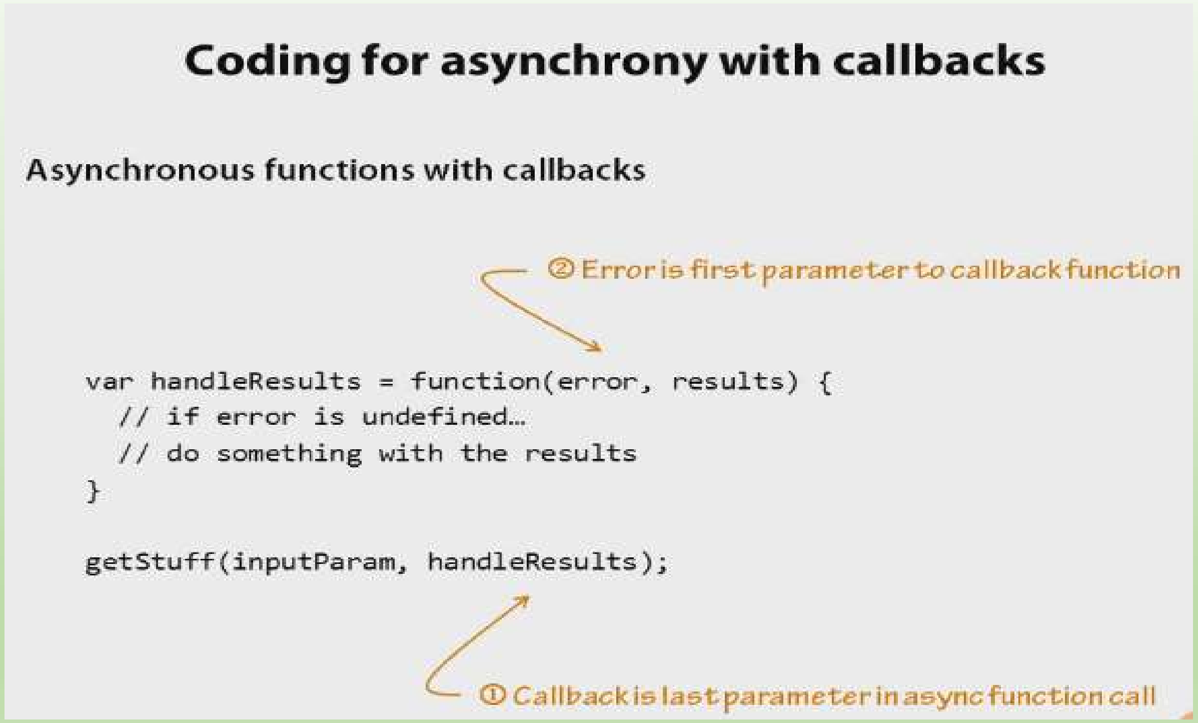
- Let's take a look at some simple examples of writing asynchronous "JavaScript" in "Node.js" using call-backs.



An asynchronous, "non-blocking" approach

```
getDbConnection(connectionString, function(err, conn) {
  conn.createStatement(function(err, stmt) {
    var results = stmt.executeQuery(sqlQuery);
    results.on('row', function(result) {
      // print result
    });
  });
```

The approach below is using a named variable “handleResults” to define the function. This way of creating callbacks will create many functions that are only used once.



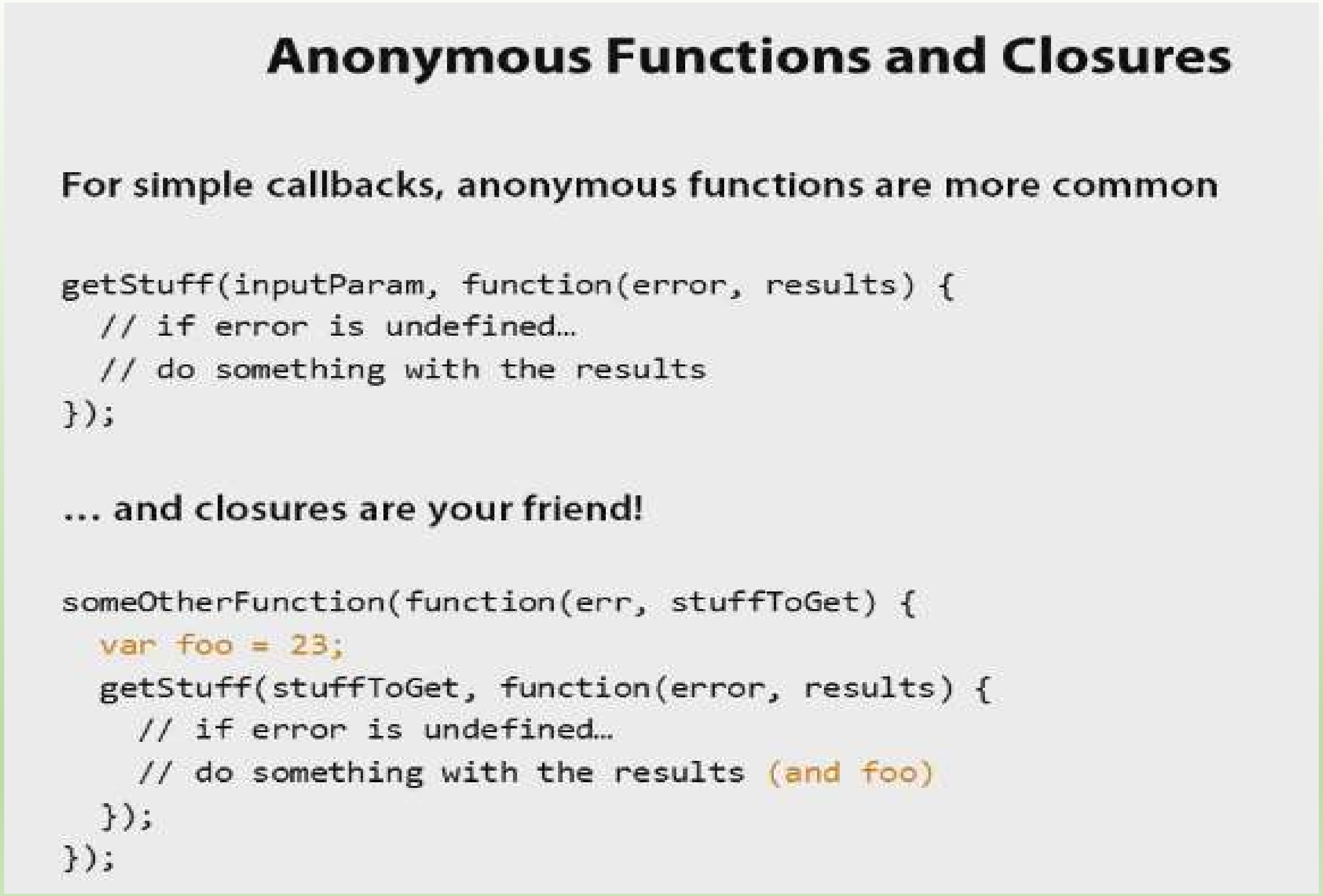
Coding for asynchrony with callbacks

Asynchronous functions with callbacks

```
var handleResults = function(error, results) {
  // if error is undefined…
  // do something with the results
}

getStuff(inputParam, handleResults);
```

② Error is first parameter to callback function
① Callback is last parameter in async function call

The approach below is not using named functions as in the prev slide



Anonymous Functions and Closures

For simple callbacks, anonymous functions are more common

```
getStuff(inputParam, function(error, results) {
  // if error is undefined…
  // do something with the results
});
```

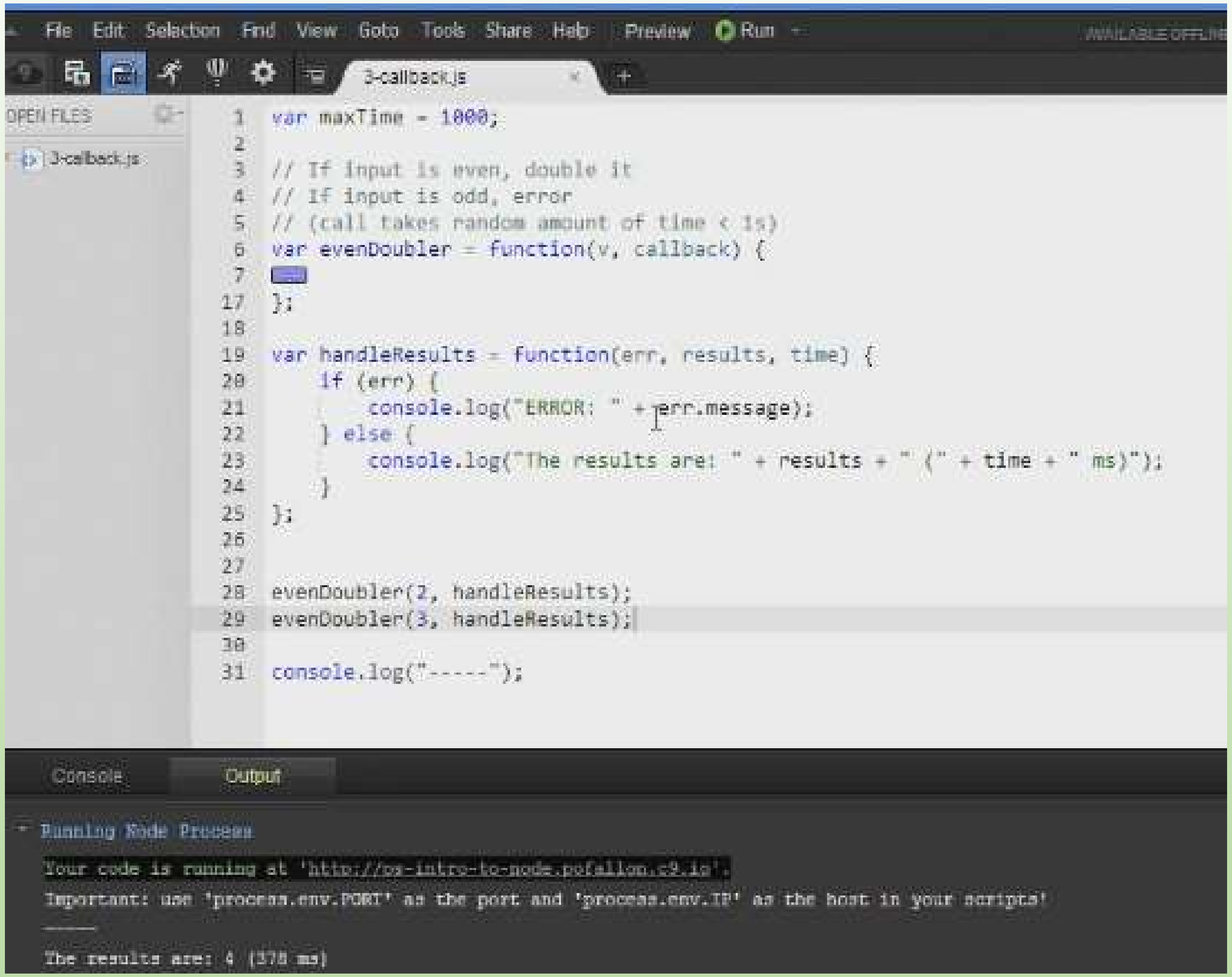
… and closures are your friend!

```
someOtherFunction(function(err, stuffToGet) {
  var foo = 23;
  getStuff(stuffToGet, function(error, results) {
    // if error is undefined…
    // do something with the results (and foo)
  });
});
```

- Asynchronous code for "Node.js" using call-backs.
- To start, we're going to keep it simple.
- And the way that it works is it will double the number you pass in only if the number you pass in is even.
- If you pass it in as even, it will double it;
- If you pass in an odd number, you're going to get an error.
- And each call to "Even Doubler" is going to take a random amount of time, random each time.

- When evenDoubler is invoked I pass in the number that I want to double, and then the call-back to receive the results when the call is done.
- And this is where this random amount of time comes into play. That's when "Handle Results" will be called. And the function "Handle Results" takes three parameters.
- One is the results, which will be the doubled number if there wasn't an error.
- And then third parameter, which is how long did it take this particular invocation of “evenDoubler" to run

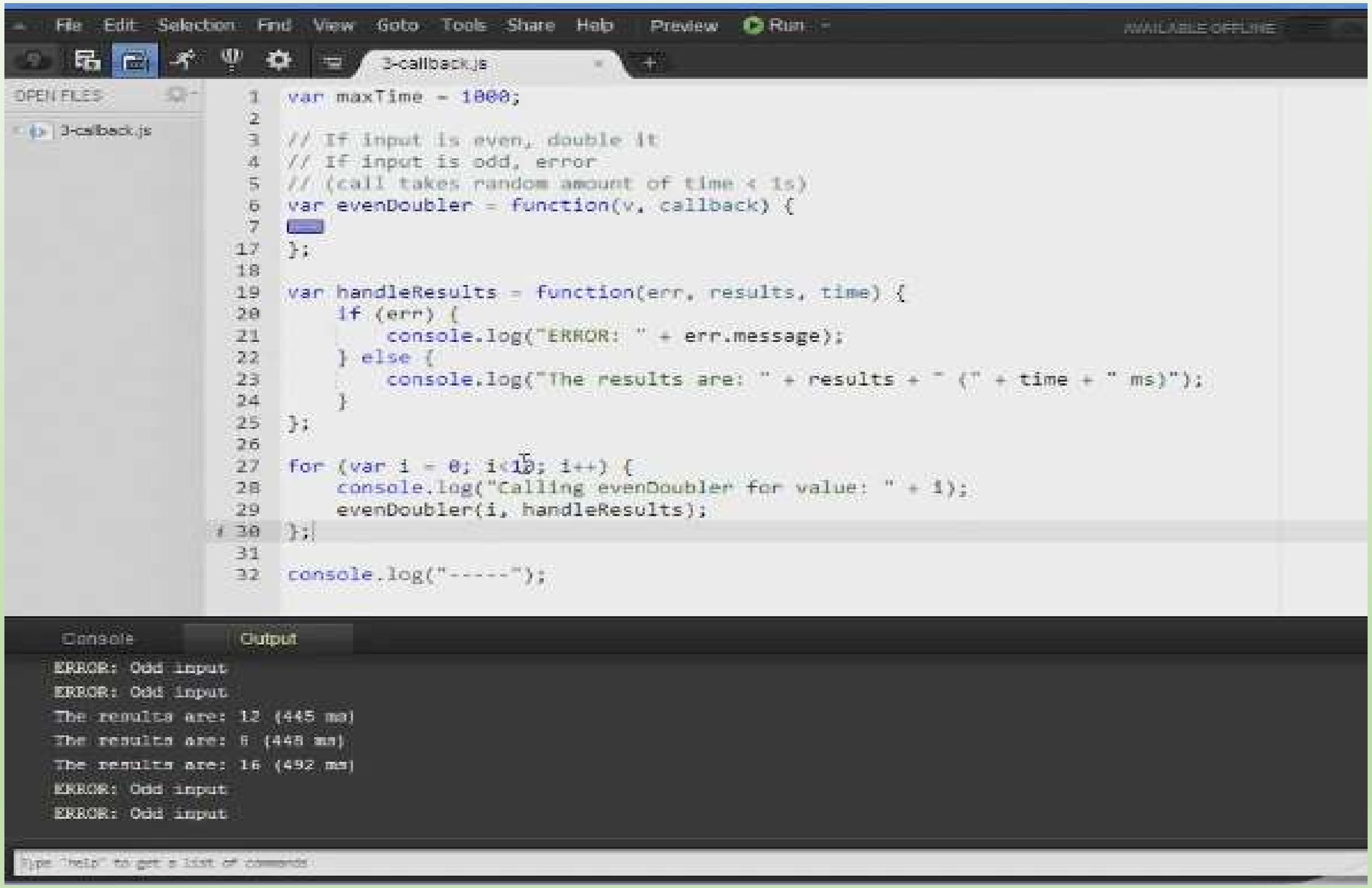
- ? Here we inspect "Error", and if we find an error, we log that to the console either way.
- And then if there's not an error, we tell you what the results are and then how long it took to calculate those results.
- So if we invoke here for an even number We got our dashed line here that printed first. Even though we invoked the function first, the first thing to the console was our dashed line. And that's because the invocation, it was going to take a certain amount



```javascript
var maxTime = 1000;

// If input is even, double it
// If input is odd, error
// (call takes random amount of time < 1s)
var evenDoubler = function(v, callback) {
...
};

var handleResults = function(err, results, time) {
    if (err) {
        console.log("ERROR: " + err.message);
    } else {
        console.log("The results are: " + results + " (" + time + " ms)");
    }
};


evenDoubler(2, handleResults);
evenDoubler(3, handleResults);

console.log("-----");
```

```
Running Node Process
  Your code is running at 'http://ps-intro-to-node.pofallon.c9.io'.
  Important: use 'process.env.PORT' as the port and 'process.env.IP' as the host in your scripts!
  -----
  The results are: 4 (378 ms)
```

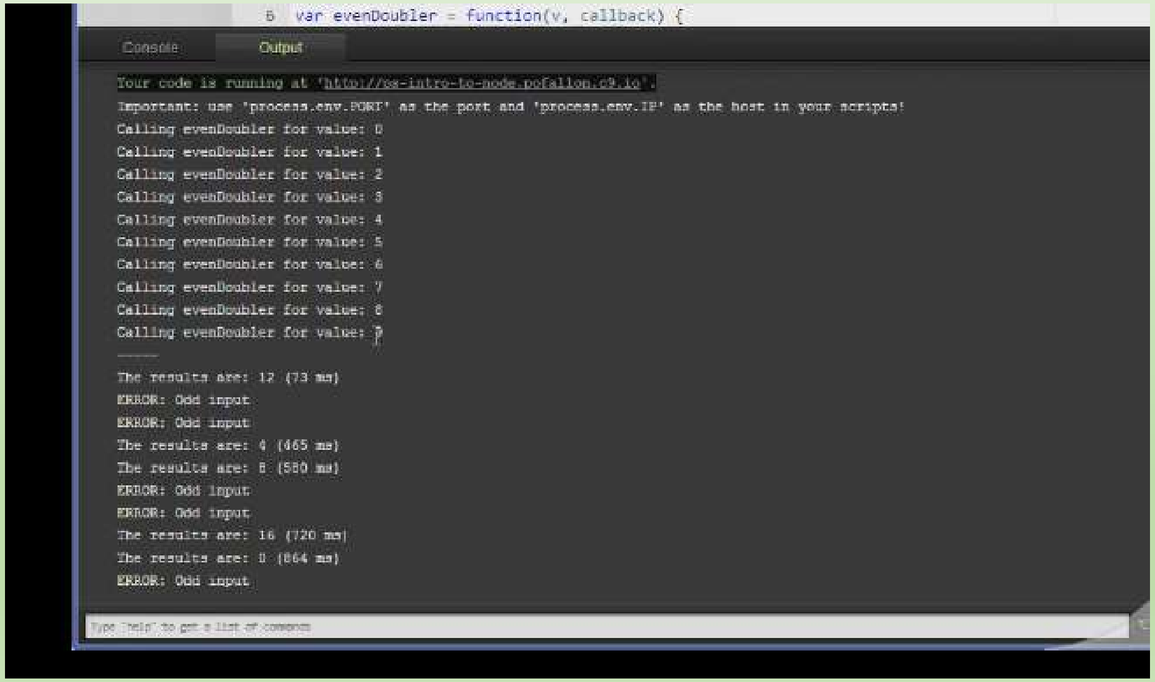
- Using for loop to execute the function multiple times


File Edit Selection Find View Goto Tools Share Help Preview Run
AVAILABLE OFFLINE
3-callback.js
OPEN FILES
3-callback.js
Console Output
Type "help" to get a list of commands

```
var maxTime = 1000;

// If input is even, double it
// If input is odd, error
// (call takes random amount of time < 1s)
var evenDoubler = function(v, callback) {

};

var handleResults = function(err, results, time) {
    if (err) {
        console.log("ERROR: " + err.message);
    } else {
        console.log("The results are: " + results + " (" + time + " ms)");
    }
};

for (var i = 0; i<10; i++) {
    console.log("Calling evenDoubler for value: " + i);
    evenDoubler(i, handleResults);
};

console.log("-----");
```

```
ERROR: Odd input
ERROR: Odd input
The results are: 12 (445 ms)
The results are: 8 (448 ms)
The results are: 16 (492 ms)
ERROR: Odd input
ERROR: Odd input
```

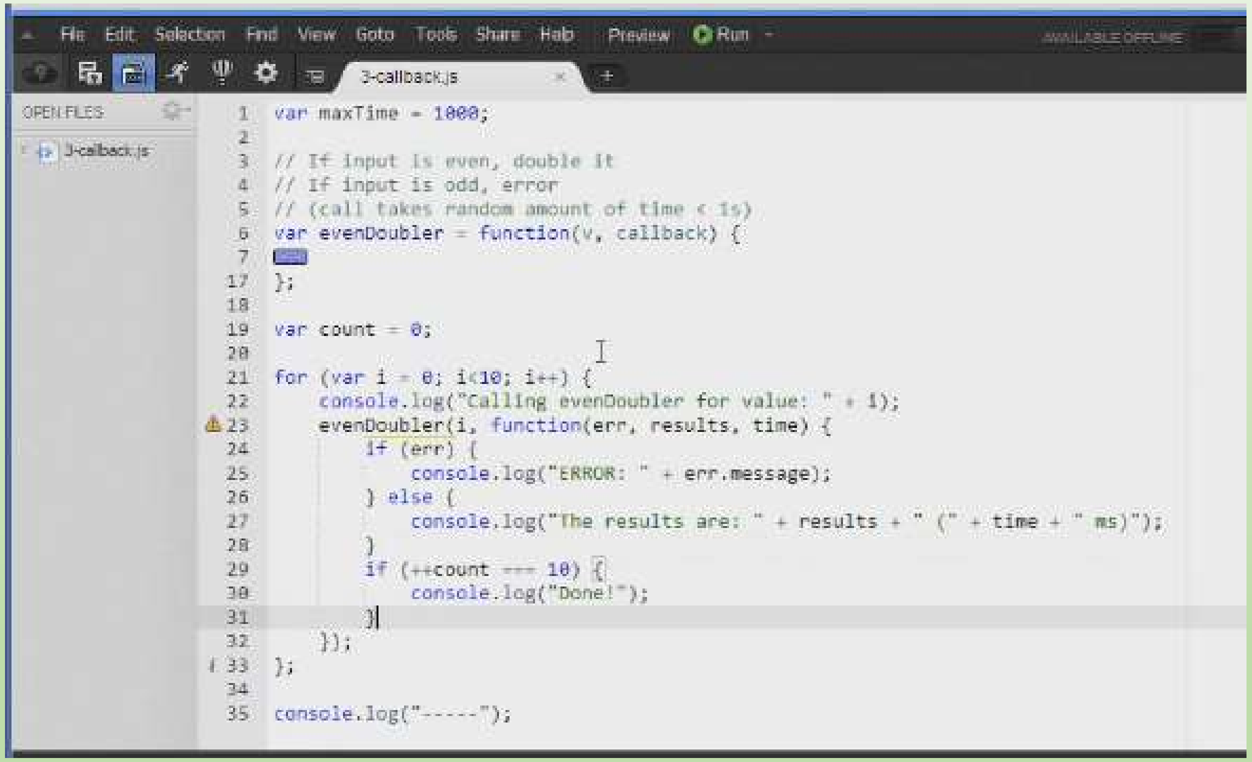
- The detailed output of the previous slide. The for loop is getting executed, then the dashed line, and the results come in very random order, but the timing is in the correct order



```
var evenDoubler = function(v, callback) {
```

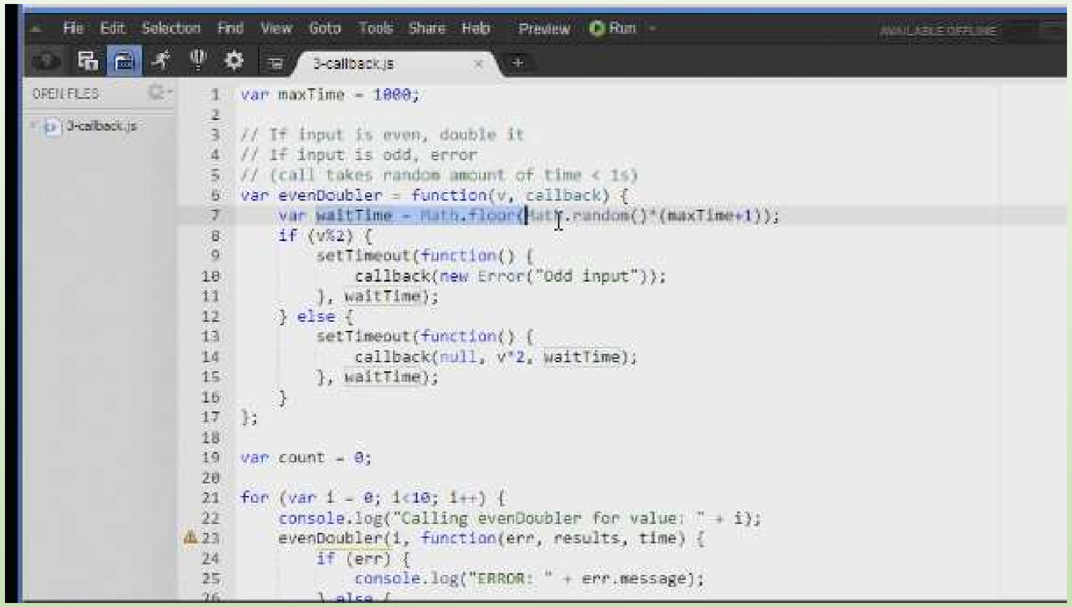
```
Your code is running at 'http://cs-intro-to-node.pofallon.c9.io'.
Important: use 'process.env.PORT' as the port and 'process.env.IP' as the host in your scripts!
Calling evenDoubler for value: 0
Calling evenDoubler for value: 1
Calling evenDoubler for value: 2
Calling evenDoubler for value: 3
Calling evenDoubler for value: 4
Calling evenDoubler for value: 5
Calling evenDoubler for value: 6
Calling evenDoubler for value: 7
Calling evenDoubler for value: 8
Calling evenDoubler for value: 9
-----
The results are: 12 (73 ms)
ERROR: Odd input
ERROR: Odd input
The results are: 4 (465 ms)
The results are: 8 (580 ms)
ERROR: Odd input
ERROR: Odd input
The results are: 16 (720 ms)
The results are: 0 (864 ms)
ERROR: Odd input
```

- Code, after changing the named function to anonymous function
- In the code below, a check is kept on , the number of times the for loop is executed.


File Edit Selection Find View Goto Tools Share Help Preview Run
AVAILABLE OFFLINE
3-callback.js
OPEN FILES
3-callback.js

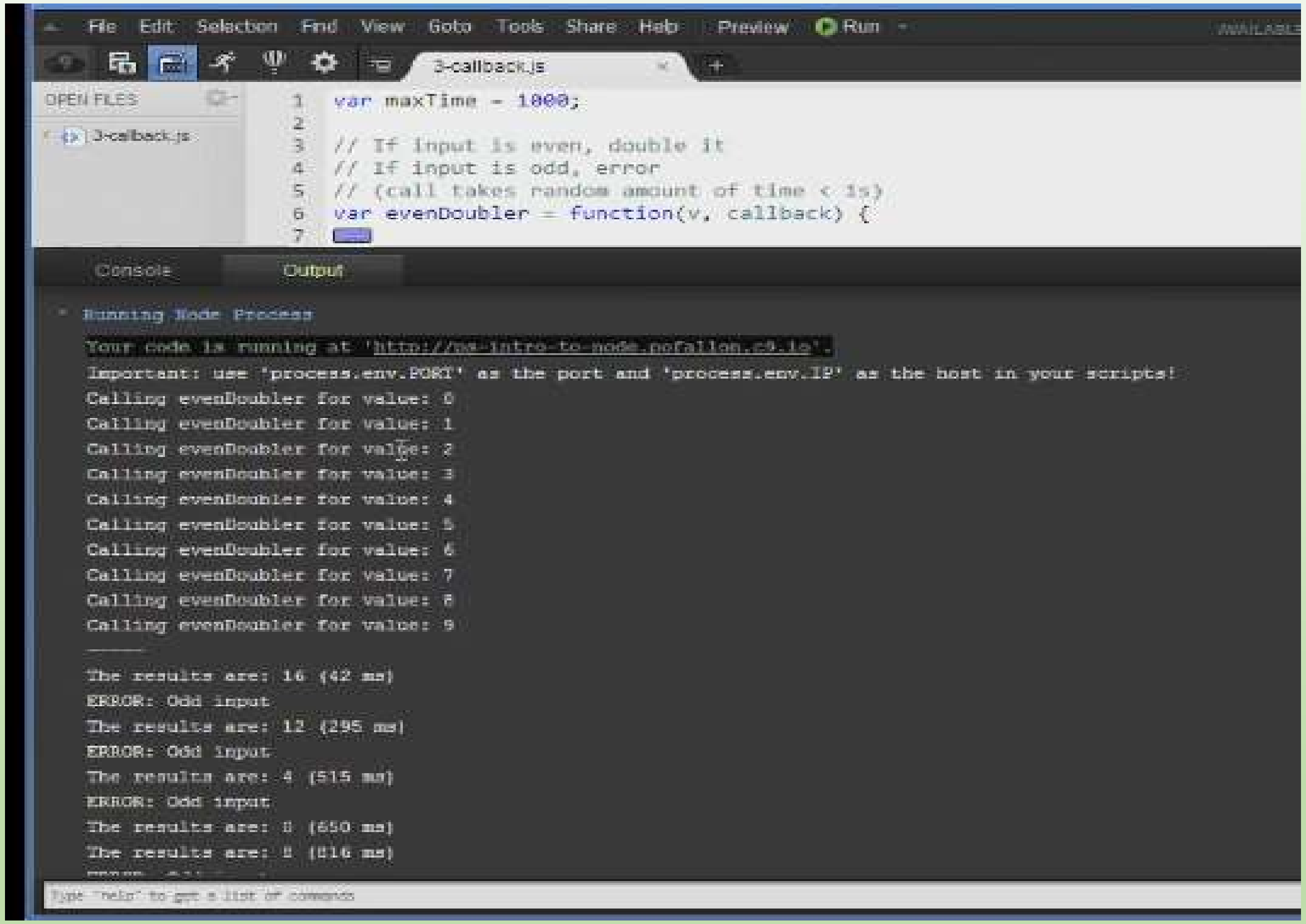
```
var maxTime = 1000;

// If input is even, double it
// If input is odd, error
// (call takes random amount of time < 1s)
var evenDoubler = function(v, callback) {

};

var count = 0;

for (var i = 0; i<10; i++) {
    console.log("Calling evenDoubler for value: " + i);
    evenDoubler(i, function(err, results, time) {
        if (err) {
            console.log("ERROR: " + err.message);
        } else {
            console.log("The results are: " + results + " (" + time + " ms)");
        }
        if (++count === 10) {
            console.log("Done!");
        }
    });
};

console.log("-----");
```

- The evenDoubler method is as below


File Edit Selection Find View Goto Tools Share Help Preview Run
AVAILABLE OFFLINE
3-callback.js
OPEN FILES
3-callback.js

```
var maxTime = 1000;

// If input is even, double it
// If input is odd, error
// (call takes random amount of time < 1s)
var evenDoubler = function(v, callback) {
    var waitTime = Math.floor(Math.random()*(maxTime+1));
    if (v%2) {
        setTimeout(function() {
            callback(new Error("Odd input"));
        }, waitTime);
    } else {
        setTimeout(function() {
            callback(null, v*2, waitTime);
        }, waitTime);
    }
};

var count = 0;

for (var i = 0; i<10; i++) {
    console.log("Calling evenDoubler for value: " + i);
    evenDoubler(i, function(err, results, time) {
        if (err) {
            console.log("ERROR: " + err.message);
        } else {
```

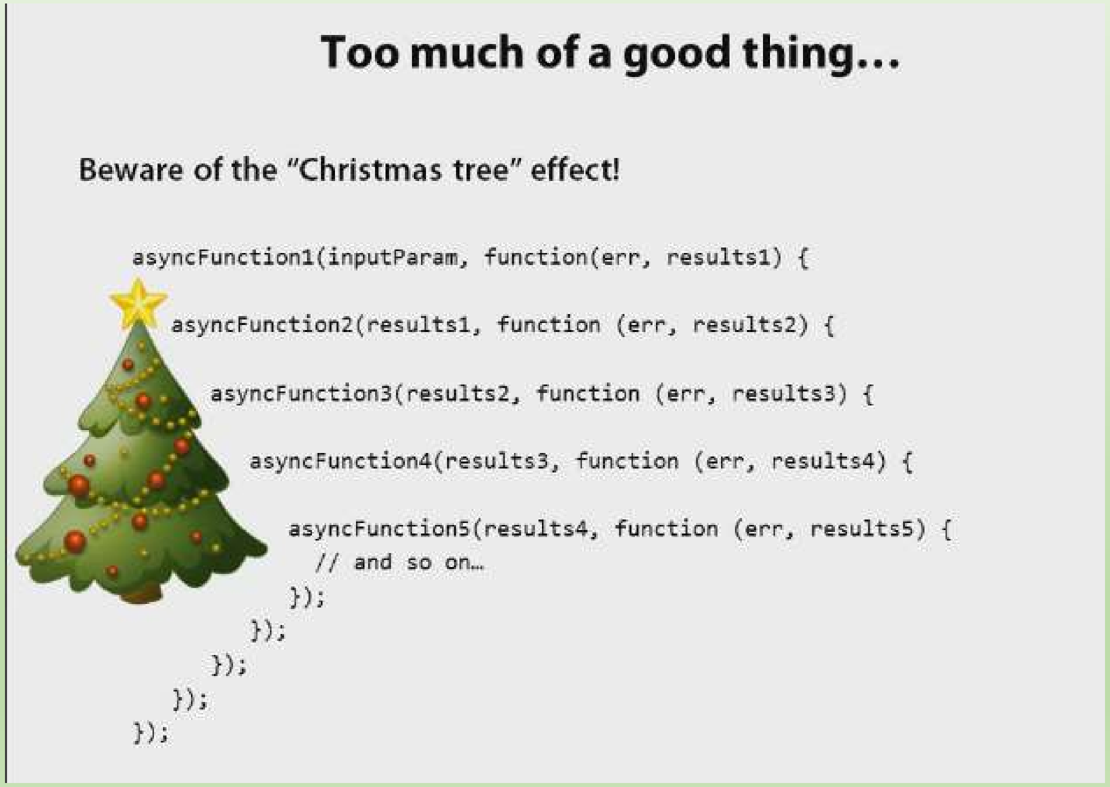
- The output with call back method is


File Edit Selection Find View Goto Tools Share Help Preview Run
3-callback.js
OPEN FILES
3-callback.js
Console
Output

```
var maxTime = 1000;

// If input is even, double it
// If input is odd, error
// (call takes random amount of time < 1s)
var evenDoubler = function(v, callback) {
```

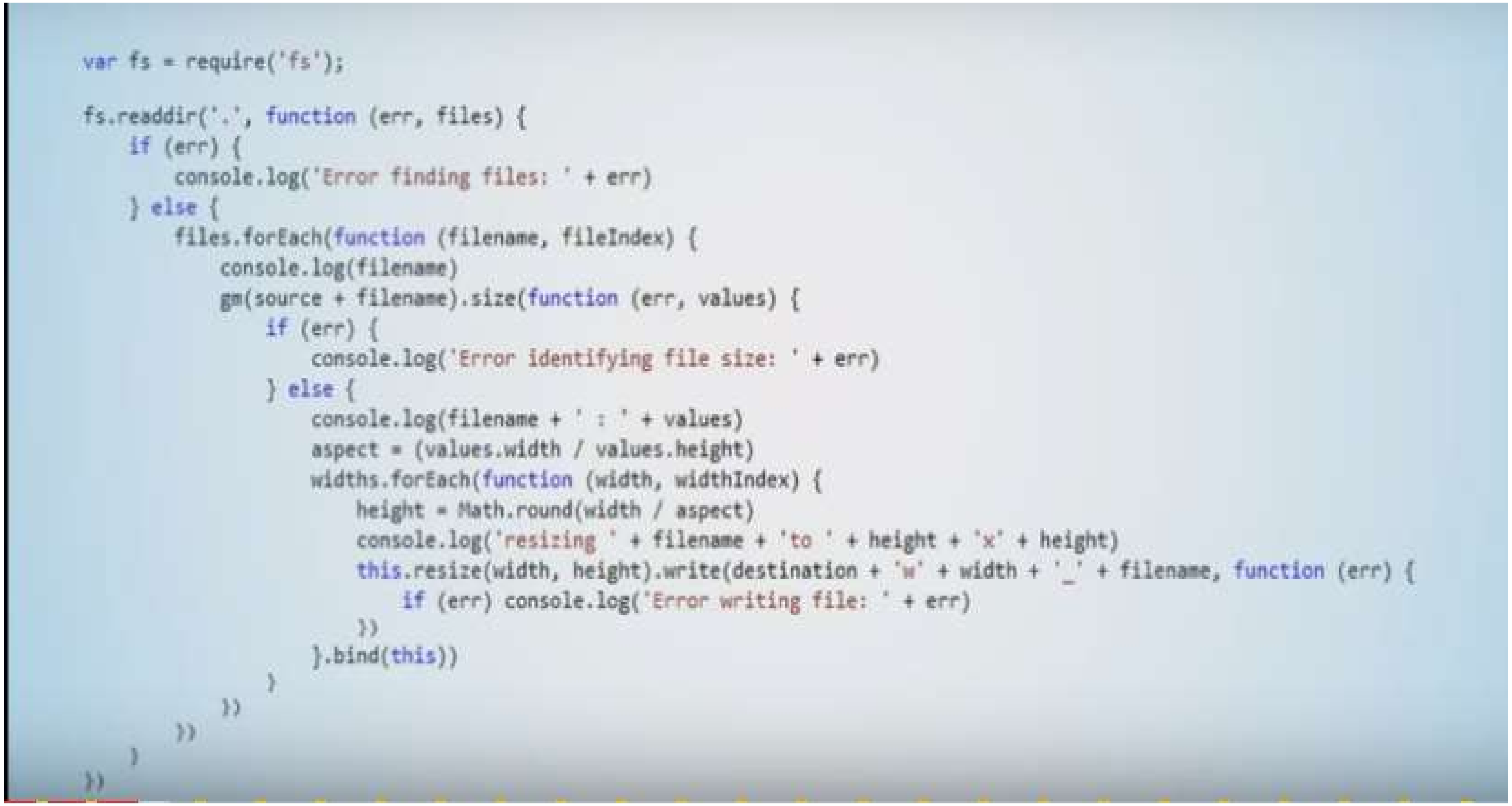
```
Running Node Process
Your code is running at 'http://ns-intro-to-node.pofallon.c9.io'.
Important: use 'process.env.PORT' as the port and 'process.env.IP' as the host in your scripts!
Calling evenDoubler for value: 0
Calling evenDoubler for value: 1
Calling evenDoubler for value: 2
Calling evenDoubler for value: 3
Calling evenDoubler for value: 4
Calling evenDoubler for value: 5
Calling evenDoubler for value: 6
Calling evenDoubler for value: 7
Calling evenDoubler for value: 8
Calling evenDoubler for value: 9
-----
The results are: 16 (42 ms)
ERROR: Odd input
The results are: 12 (295 ms)
ERROR: Odd input
The results are: 4 (515 ms)
ERROR: Odd input
The results are: 0 (650 ms)
The results are: 8 (816 ms)
```

- The Christmas tree problem


Too much of a good thing…

Beware of the "Christmas tree" effect!

```
asyncFunction1(inputParam, function(err, results1) {
  asyncFunction2(results1, function (err, results2) {
    asyncFunction3(results2, function (err, results3) {
      asyncFunction4(results3, function (err, results4) {
        asyncFunction5(results4, function (err, results5) {
          // and so on…
        });
      });
    });
  });
});
```



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

fs.readdir('.', function (err, files) {
    if (err) {
        console.log('Error finding files: ' + err)
    } else {
        files.forEach(function (filename, fileIndex) {
            console.log(filename)
            gm(source + filename).size(function (err, values) {
                if (err) {
                    console.log('Error identifying file size: ' + err)
                } else {
                    console.log(filename + ' : ' + values)
                    aspect = (values.width / values.height)
                    widths.forEach(function (width, widthIndex) {
                        height = Math.round(width / aspect)
                        console.log('resizing ' + filename + 'to ' + height + 'x' + height)
                        this.resize(width, height).write(destination + 'w' + width + '_' + filename, function (err) {
                            if (err) console.log('Error writing file: ' + err)
                        })
                    }.bind(this))
                }
            })
        })
    }
})
```

- The Christmas tree problem

While Using callbacks should be careful not to overdo it. Selecting the right combination of approaches can be crucial in structuring the "Node.js" application.

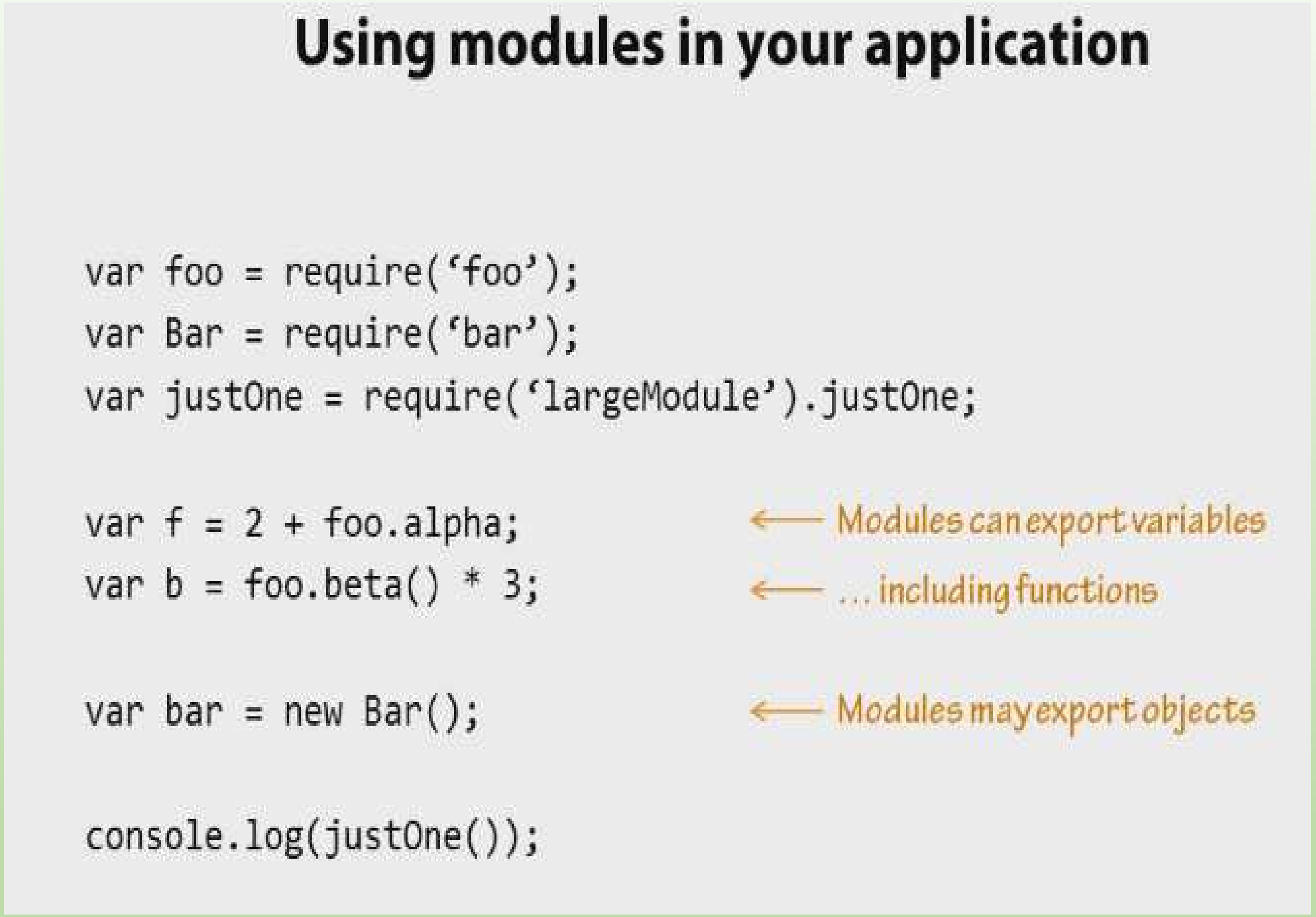
Anonymous functions are very common and very useful, but many novice "Node" developers, who rely on them exclusively, find themselves frustrated with the Christmas tree problem.

Code like this can be difficult to debug and maintain.

This is often cited as a shortcoming of Node's programming model. However, the smart way is to use named functions, as well as modules, event emitters and streams etc.

- Promises
- A function that will return a promise for an object in the future
- Promises can be chained together
- Promises simplify programming of asynchronous systems
- Read more: http://spin.atomicobject.com/2012/03/14/nodejs-and-asynchronous-programming-with-promises/

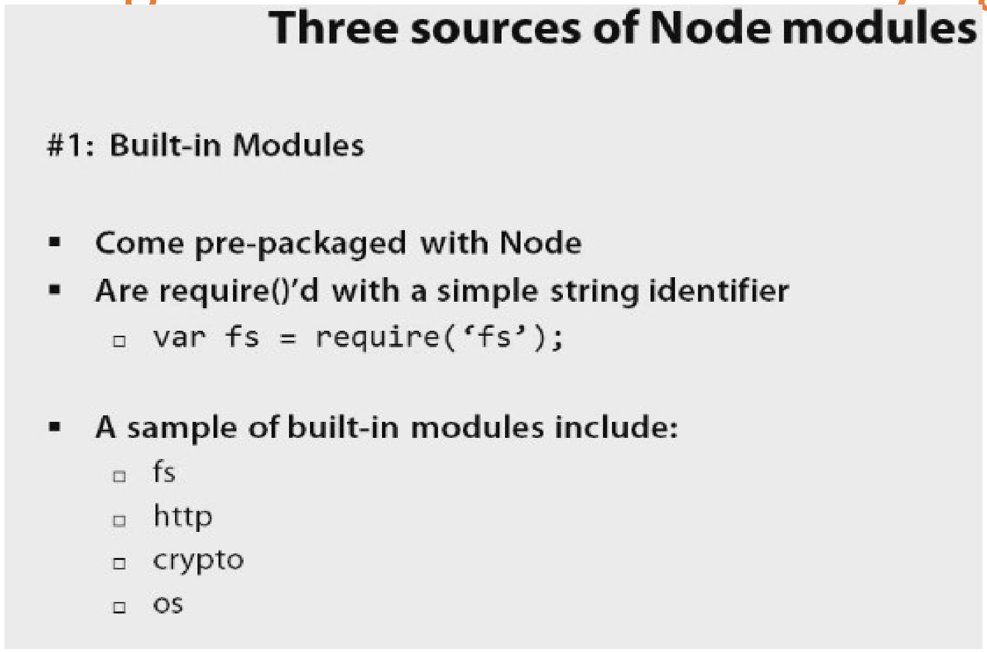
- MODULE 2
- Modules, require and NPM
- Modules are the way to bring external functionality to theNode application.
- The require function loads a module and assigns it to a variable for the application to use.
- Modules make their functionality available by explicitly exporting it for use in other applications.
- A module can export specific variables and these variables can also be functions.
- Sometimes a module may export an object which can be instantiated in the code.
- A module which simply exports a set of variables is often assigned to a camel case variable starting with a lowercase letter.
- One can import just the one function you need by specifying the variable name immediately after the require function call



Using modules in your application

```
var foo = require(‘foo’);
var Bar = require(‘bar’);
var justOne = require(‘largeModule’).justOne;

var f = 2 + foo.alpha;
var b = foo.beta() * 3;

var bar = new Bar();

console.log(justOne());
```

- There are three main sources of modules that can brought into the project with the require function.
- The first is Node's built-in modules.
- http for creating and responding to http requests.
- crypto for performing cryptographic functions.
- OS for accessing attributes of the underlying operating system.

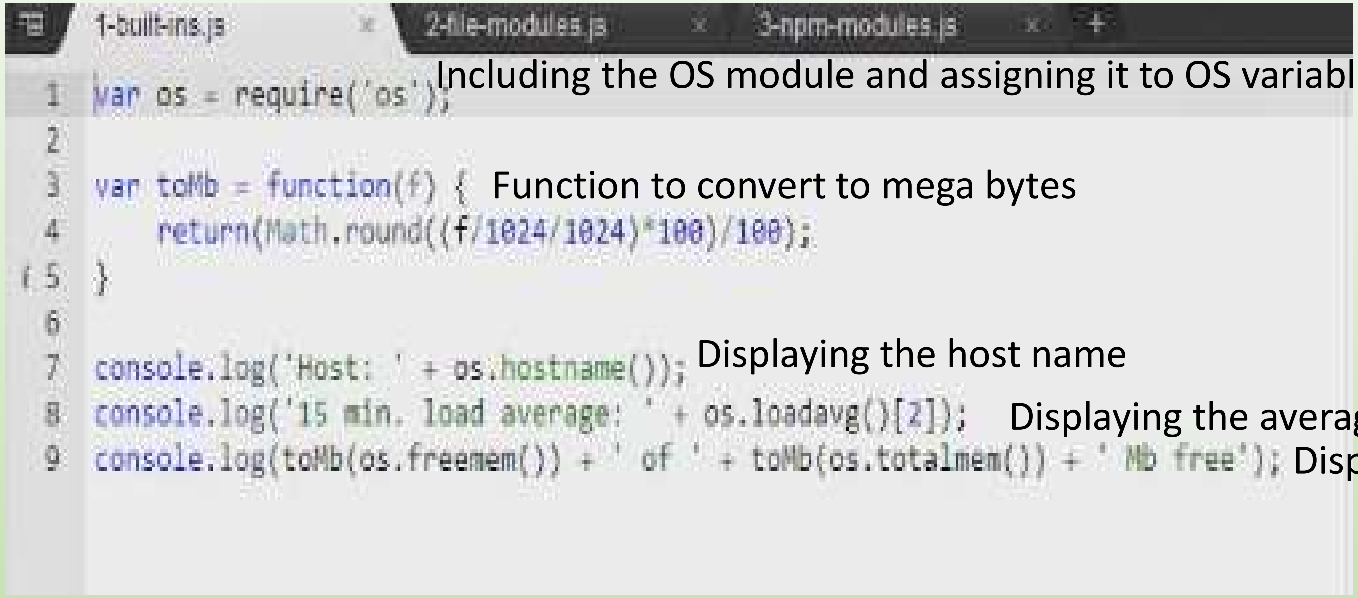


Three sources of Node modules

#1: Built-in Modules

- Come pre-packaged with Node
- Are require()'d with a simple string identifier
  - `var fs = require('fs');`
- A sample of built-in modules include:
  - fs
  - http
  - crypto
  - os

• Accessing Built In modules



```
var os = require('os');
                                                                                                                                   
var toMb = function(f) {
    return(Math.round((f/1024/1024)*100)/100);
}

console.log('Host: ' + os.hostname());
console.log('15 min. load average: ' + os.loadavg()[2]);
console.log(toMb(os.freemem()) + ' of ' + toMb(os.totalmem()) + ' Mb free');
```

Including the OS module and assigning it to OS variable

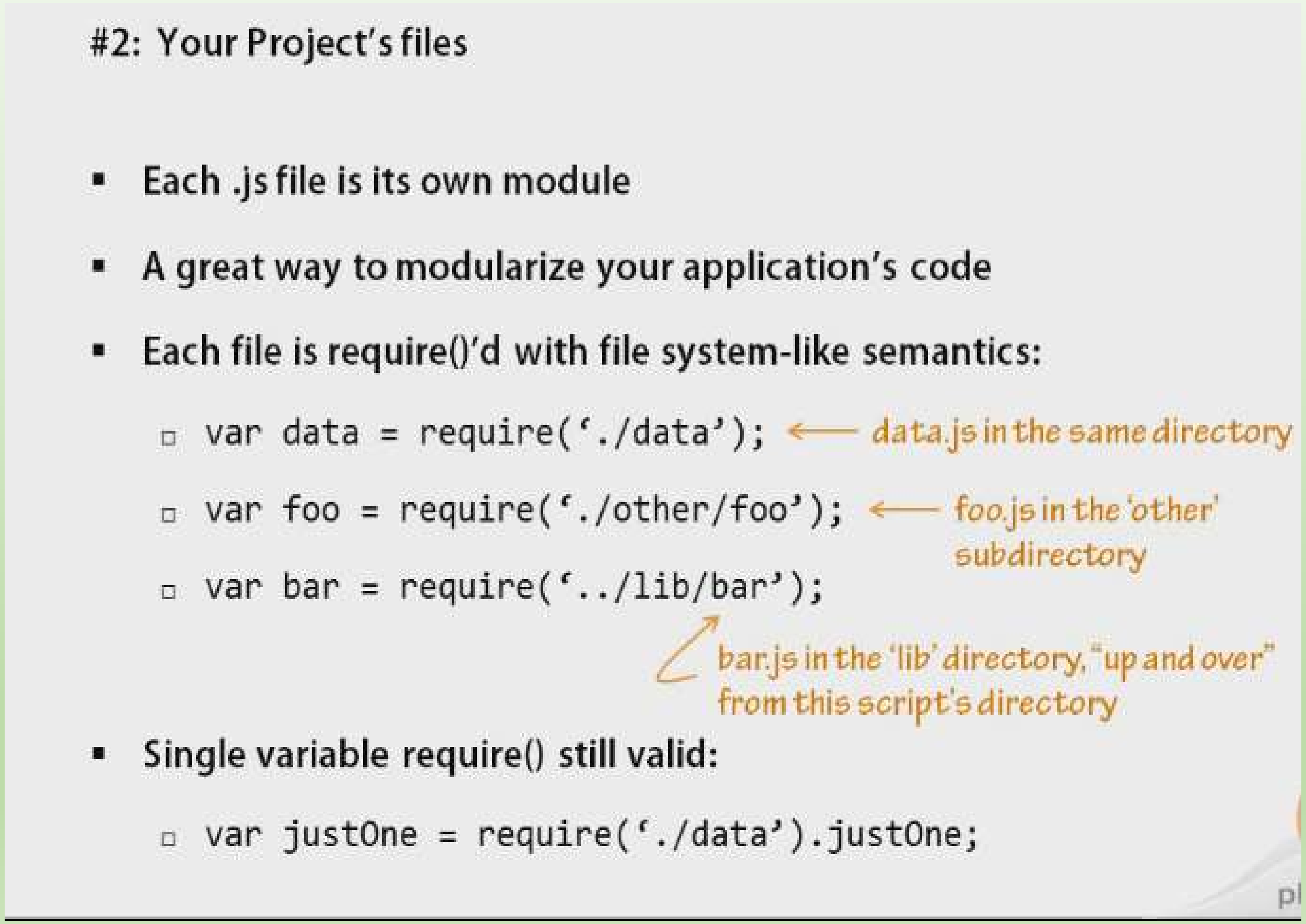
Function to convert to mega bytes

Displaying the host name

Displaying the average load

Displaying the free memory

- Another use of the require function is to access functionality located within other files in the project



#2: Your Project's files

- Each .js file is its own module
- A great way to modularize your application's code
- Each file is require()'d with file system-like semantics:

```
var data = require('./data');          ← data.js in the same directory
var foo = require('./other/foo');      ← foo.js in the 'other' subdirectory
var bar = require('../lib/bar');
```

bar.js in the 'lib' directory, "up and over" from this script's directory

- Single variable require() still valid:

```
var justOne = require('./data').justOne;
```

Another use of the require function is to access functionality located within other files in your project.

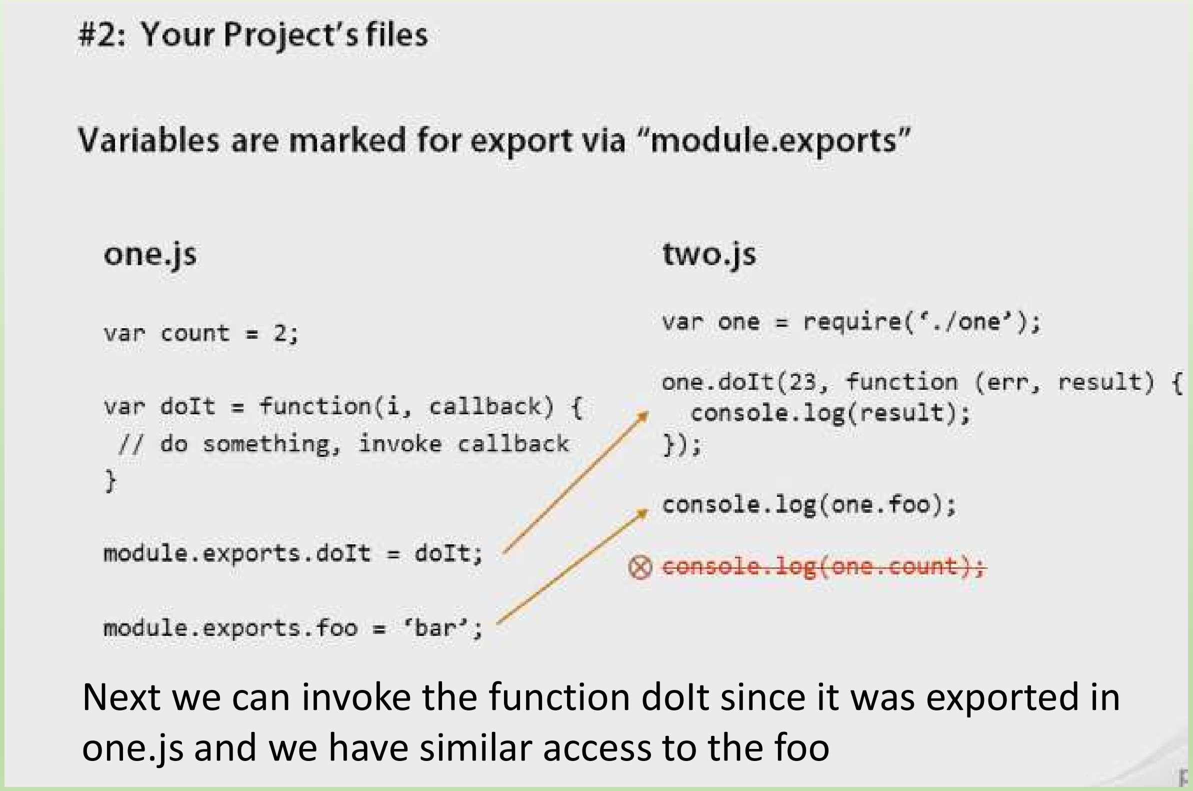
In Node's module system each of your JavaScript files is a module and can expose functionality to be required by other files.

This is a great way to modularize your code, making it easier to develop and maintain.

For example, you can require a file in the same directory, in a subdirectory, or in another navigable directory.

using require to import one JavaScript file into another JavaScript file.

in two.js we require the file one.js using this syntax



#2: Your Project's files

Variables are marked for export via "module.exports"

one.js

```
var count = 2;

var doIt = function(i, callback) {
 // do something, invoke callback
}

module.exports.doIt = doIt;

module.exports.foo = 'bar';
```

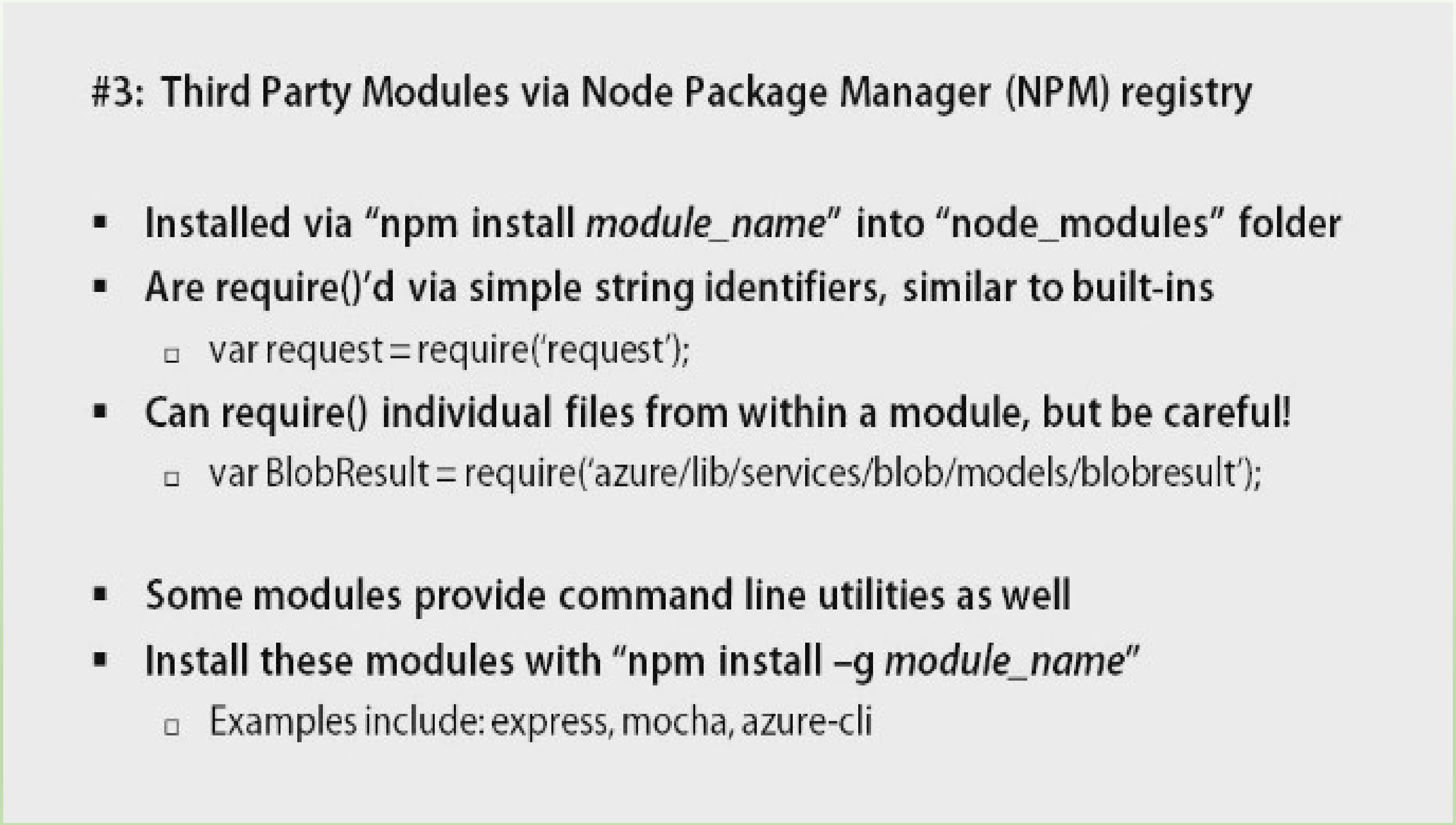
two.js

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

one.doIt(23, function (err, result) {
  console.log(result);
});

console.log(one.foo);

⊗ ~~console.log(one.count);~~
```

the variable foo with the value bar exported.

Next we can invoke the function doIt since it was exported in one.js and we have similar access to the foo

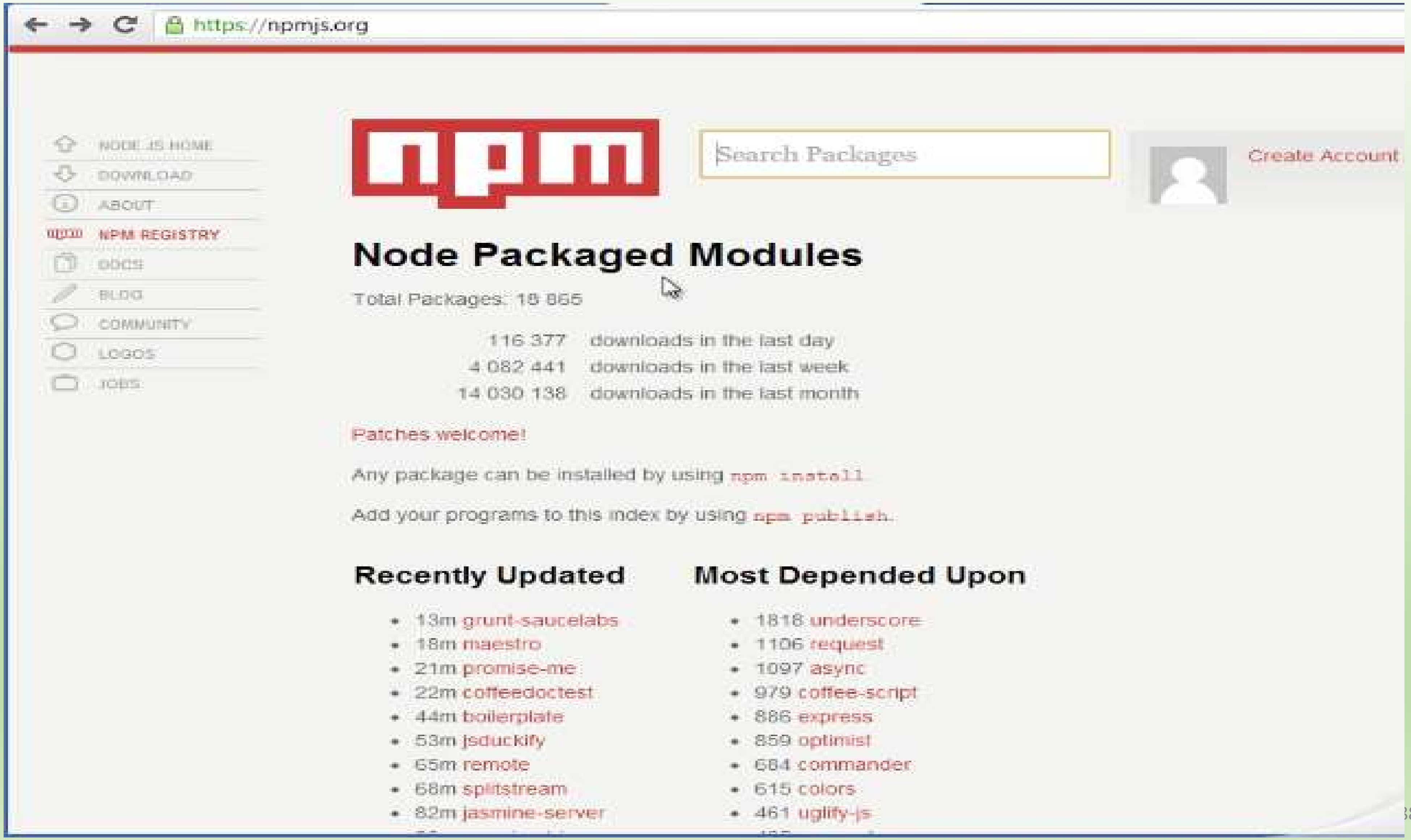
However, since the count variable in one.js was not exported it is not available in two.js



#3: Third Party Modules via Node Package Manager (NPM) registry

- Installed via "npm install *module_name*" into "node_modules" folder
- Are require()'d via simple string identifiers, similar to built-ins
  - var request = require('request');
- Can require() individual files from within a module, but be careful!
  - var BlobResult = require('azure/lib/services/blob/models/blobresult');

- Some modules provide command line utilities as well
- Install these modules with "npm install –g *module_name*"
  - Examples include: express, mocha, azure-cli


https://npmjs.org

Node Packaged Modules

Total Packages: 18 865

```
   116 377   downloads in the last day
 4 082 441   downloads in the last week
14 030 138   downloads in the last month
```

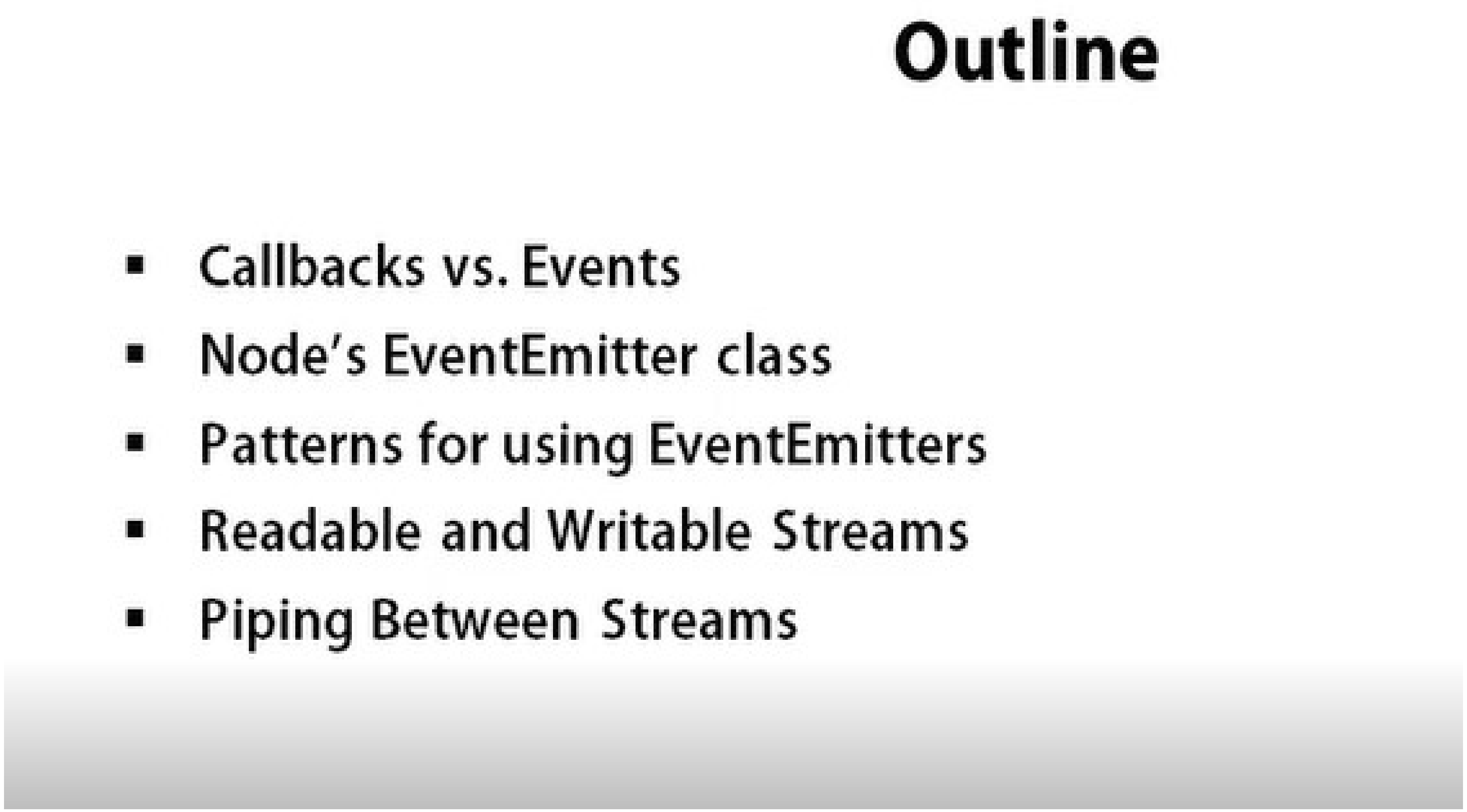
Patches welcome!

Any package can be installed by using `npm install`.

Add your programs to this index by using `npm publish`.

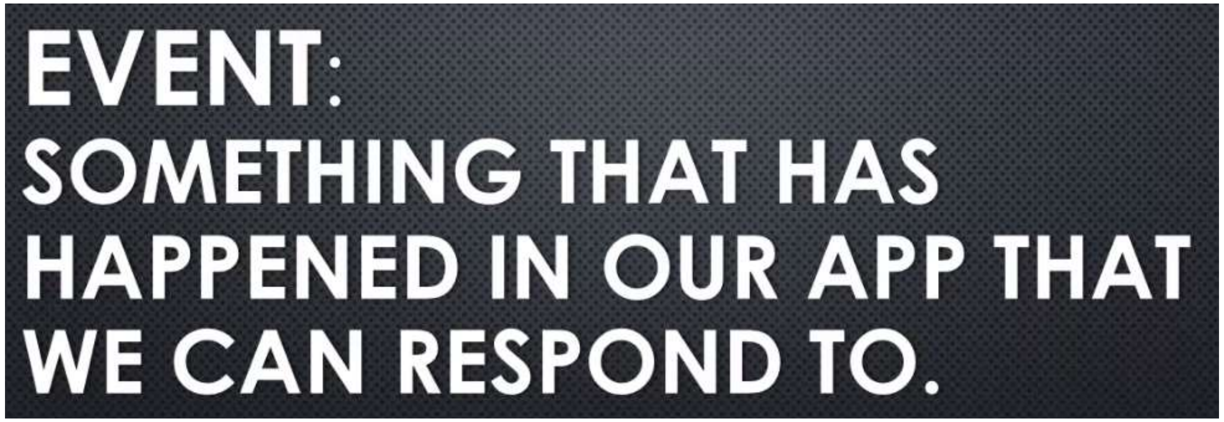
Recently Updated	Most Depended Upon
13m grunt-saucelabs	1818 underscore
18m maestro	1106 request
21m promise-me	1097 async
22m coffeedoctest	979 coffee-script
44m boilerplate	886 express
53m jsduckify	859 optimist
65m remote	684 commander
68m splitstream	615 colors
82m jasmine-server	461 uglify-js

- Events And Streams

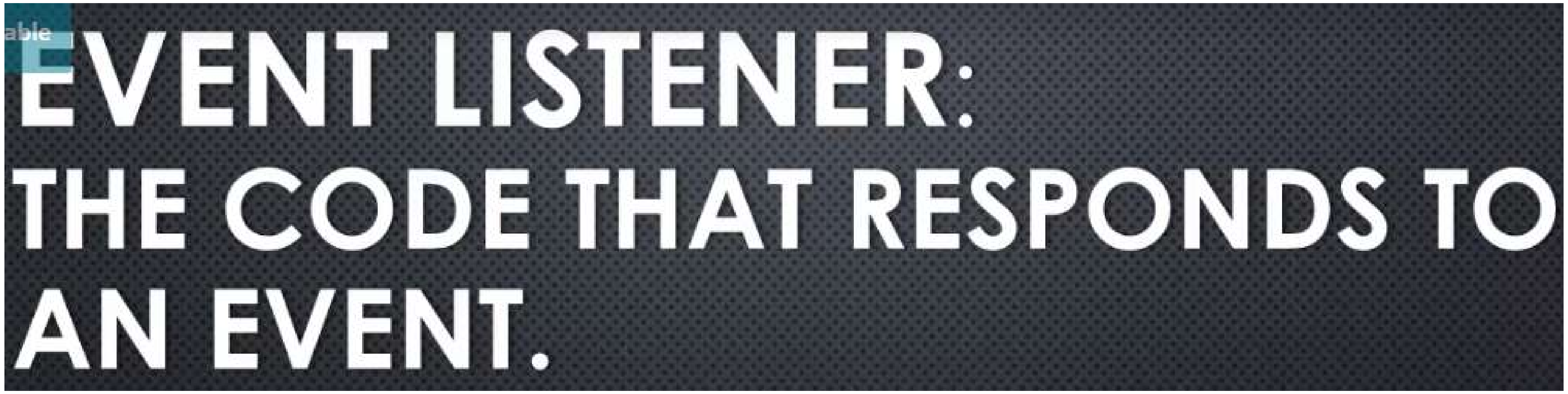


Outline

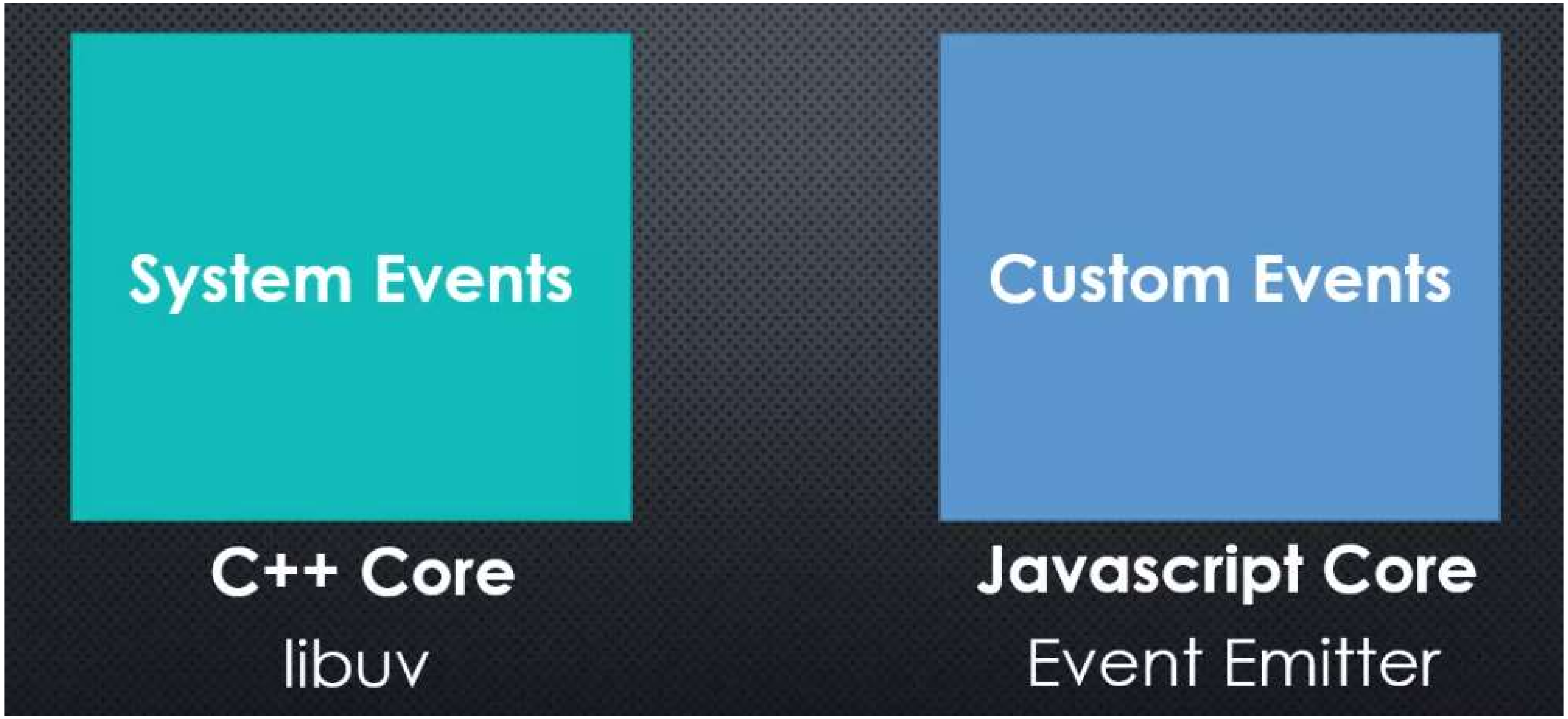
- Callbacks vs. Events
- Node’s EventEmitter class
- Patterns for using EventEmitters
- Readable and Writable Streams
- Piping Between Streams



EVENT:
SOMETHING THAT HAS HAPPENED IN OUR APP THAT WE CAN RESPOND TO.

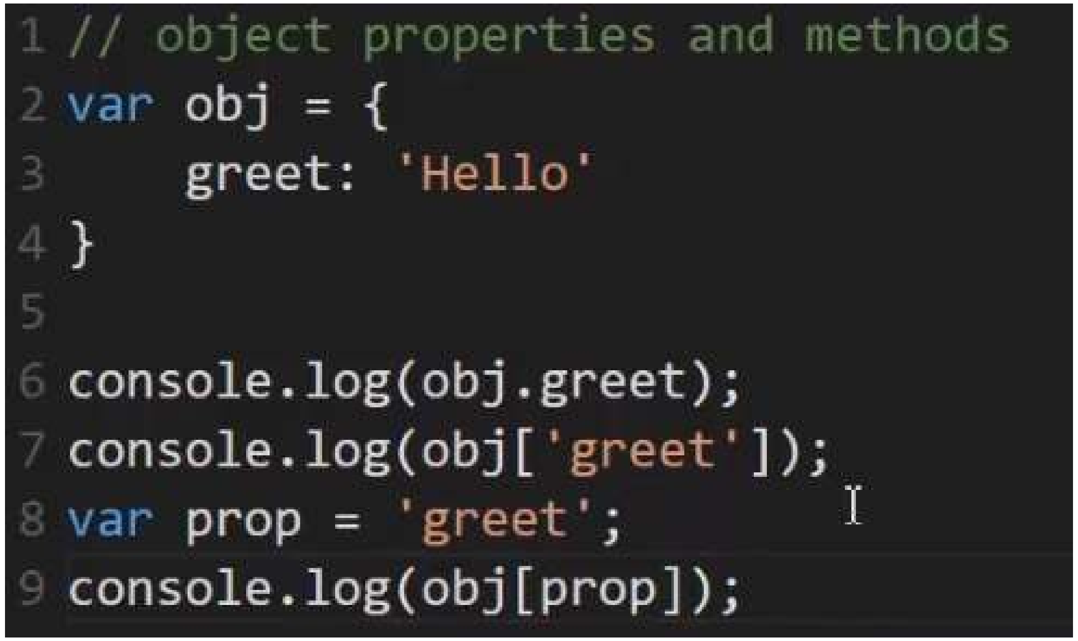


EVENT LISTENER:
THE CODE THAT RESPONDS TO AN EVENT.

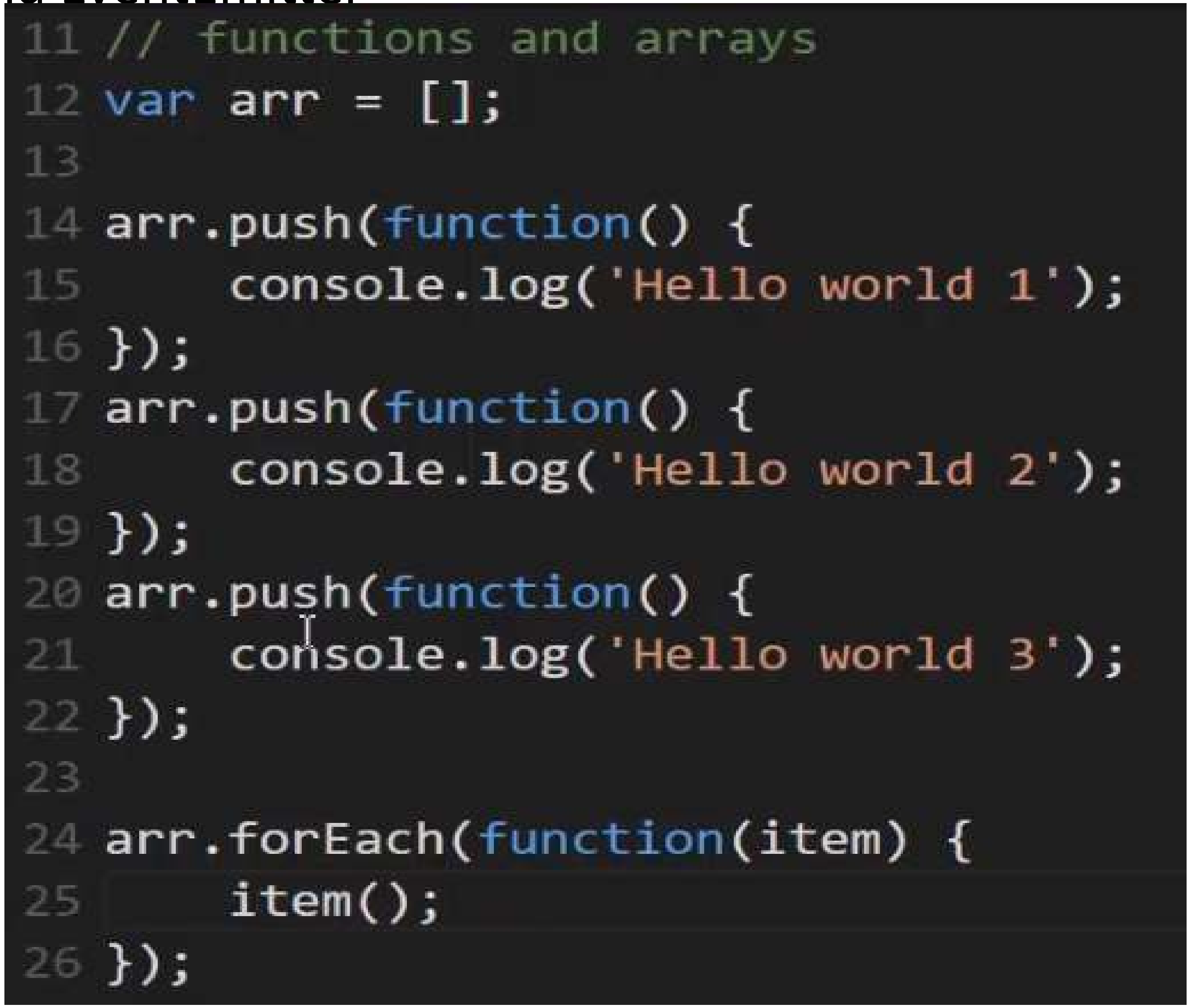

System Events
Custom Events
C++ Core
libuv
Javascript Core
Event Emitter

- Events and EventEmitter

OBJECT PROPERTIES, FIRST CLASS FUNCTIONS, AND ARRAYS

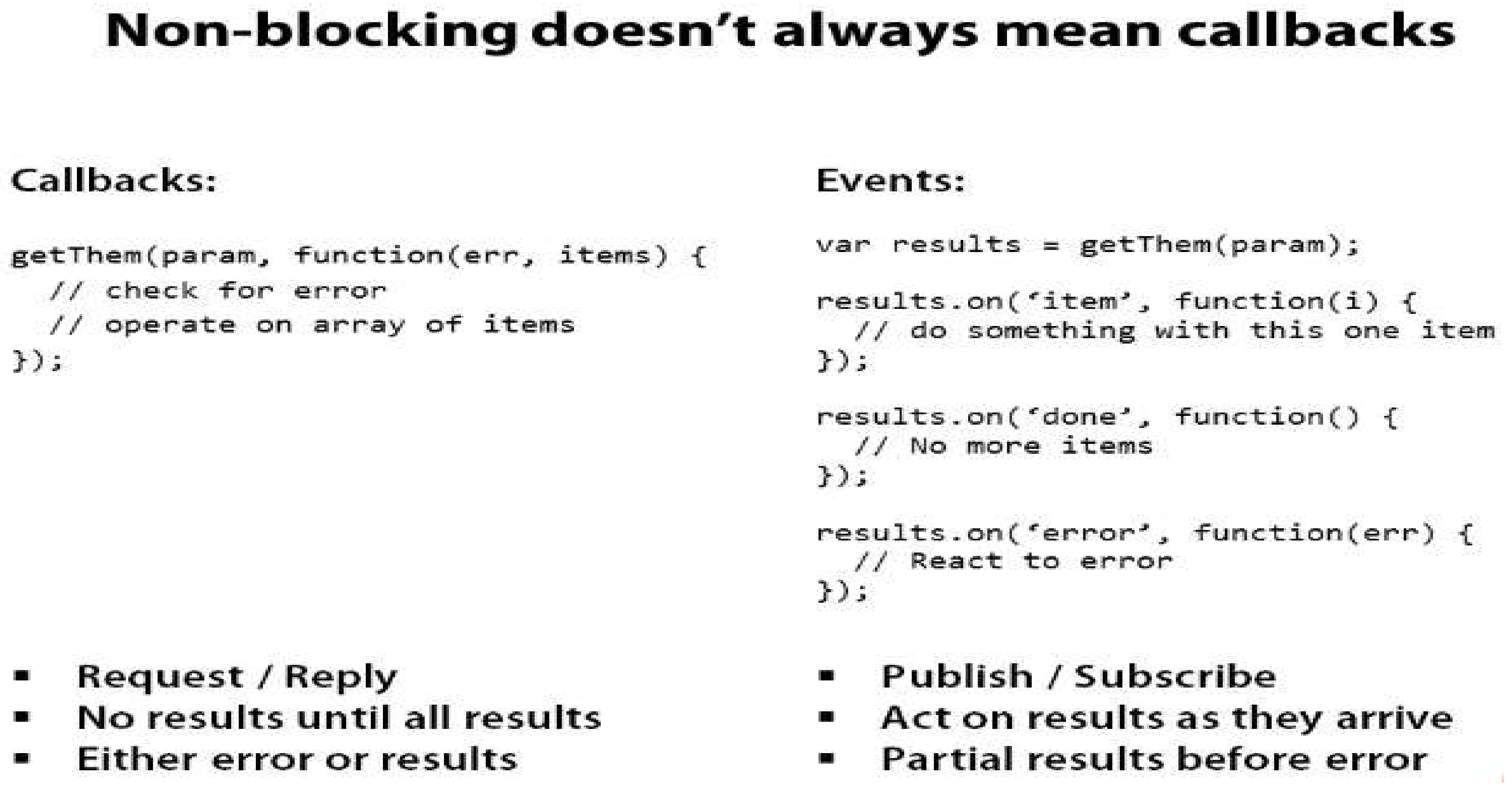

```
// object properties and methods
var obj = {
    greet: 'Hello'
}

console.log(obj.greet);
console.log(obj['greet']);
var prop = 'greet';
console.log(obj[prop]);
```

- Events and EventEmitter



```
// functions and arrays
var arr = [];

arr.push(function() {
    console.log('Hello world 1');
});
arr.push(function() {
    console.log('Hello world 2');
});
arr.push(function() {
    console.log('Hello world 3');
});

arr.forEach(function(item) {
    item();
});
```

- Events and EventEmitter Class
- Callbacks is a way to implement asynchronous non-blocking code.
- Node provides another way to implement it, with EVENTS



Non-blocking doesn't always mean callbacks

Callbacks:

```
getThem(param, function(err, items) {
  // check for error
  // operate on array of items
});
```

- Request / Reply
- No results until all results
- Either error or results

Events:

```
var results = getThem(param);

results.on('item', function(i) {
  // do something with this one item
});

results.on('done', function() {
  // No more items
});

results.on('error', function(err) {
  // React to error
});
```

- Publish / Subscribe
- Act on results as they arrive
- Partial results before error

- In the previous slide
- The getThem function returns a value immediately.
- The value is an instance of the eventEmitter class.
- The results object has an on function where in we are specifying that for each item event execute this function passing in the current item.
- Then on the Done event or when there are no more results invoke this function.
- And if there is an error invoke the on function with error as the first parameter
- So in the events approach the on method can be invoked repeatedly to provide multiple functions to invoke on each event, i.e. subscribing to the events.
- In the evented approach, functions associated with the item event will be invoked for each item.
- This gives to act upon the first item as soon as it arrives and next item and next

- The event emitter contd:
- This means the items are not accumulated in the memory
- In callback scenario it is error or results
- In evented scenario, an error is emitted as a separate event.
- In evented scenario the item and done events do not pass an error as the first value.
- Hence in this evented approach the error can be emitted instead of any item events or after some item events have already been emitted.

- Node’s “EventEmitter ” class
- This class is provided by node as a construct for building event-driven interfaces.
- The code that is subscribing to events, will call the on function of the event emitter instance and specify the event being subscribed to.
- The code that publishes the event will call the Emit function and specify the event being emitted
- The events are strings and can be of any value.
- In the previous example the 3 events we defined were item, done and error.
- An event can be emitted with zero or more arguments.
- The set of events and their arguments constitute an interface or a contract between the subscriber and the publisher

- **The “eventEmitter” class contd….**
- Node uses two common patterns for EventEmitters
- 1. As a return value from a function where an instance of event emitter is created directly and returned from a function.
- 2. The other common pattern is when an object extends EventEmitter and emits events while also providing other functions and values.

Node’s “EventEmitter” class

The publisher:

```
emitter.emit(event, [args]);
```

$\longrightarrow$

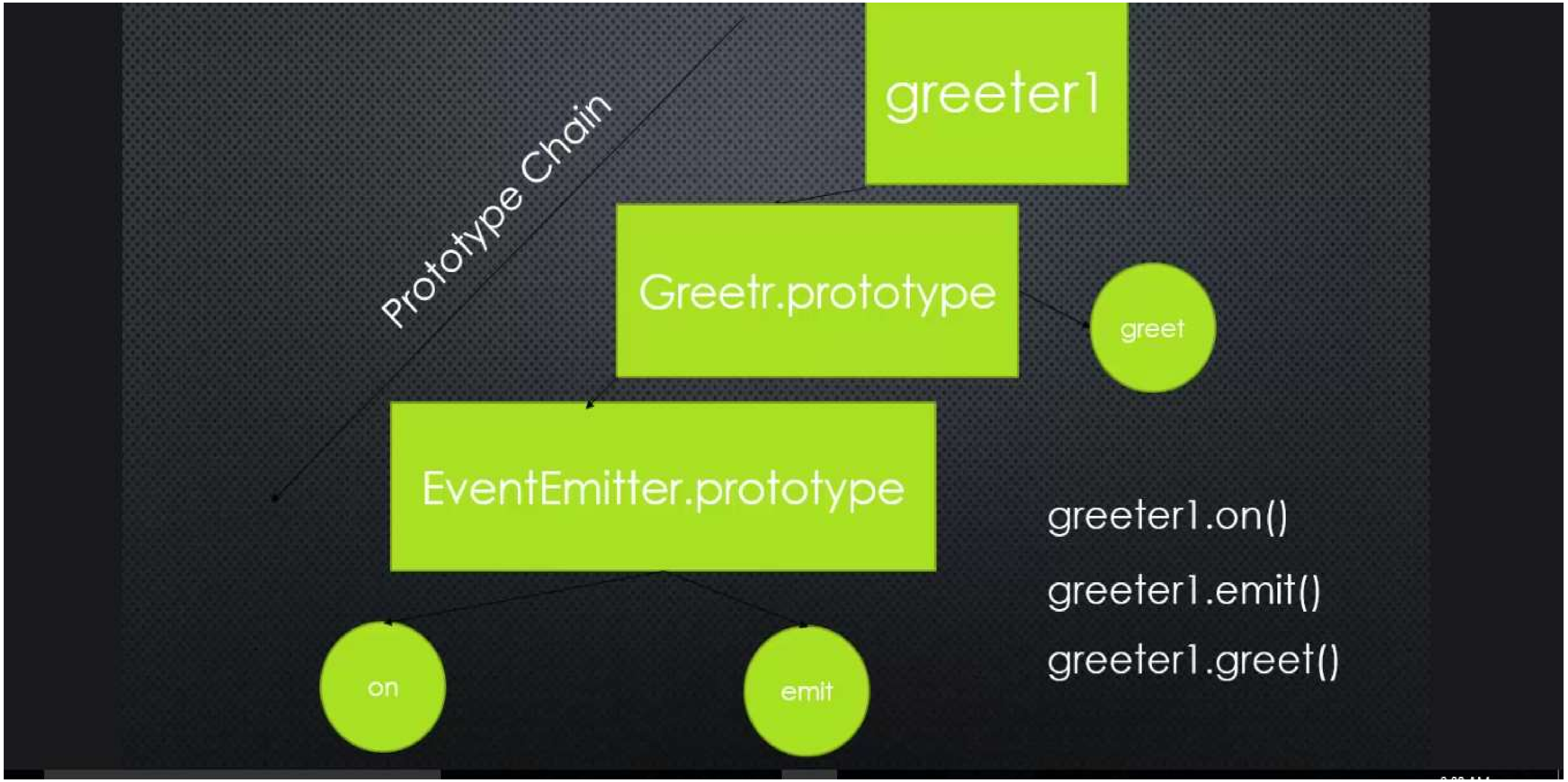
The subscriber:

```
emitter.on(event, listener);
```

- The “event” can be any string
- An event can be emitted with zero or more arguments
- The set of events and their arguments constitute a “interface” exposed to the subscriber by the publisher (emitter).

Two common patterns for EventEmitters:

1. As a return value from a function call (see earlier example)
2. Objects that extend EventEmitter to emit events themselves

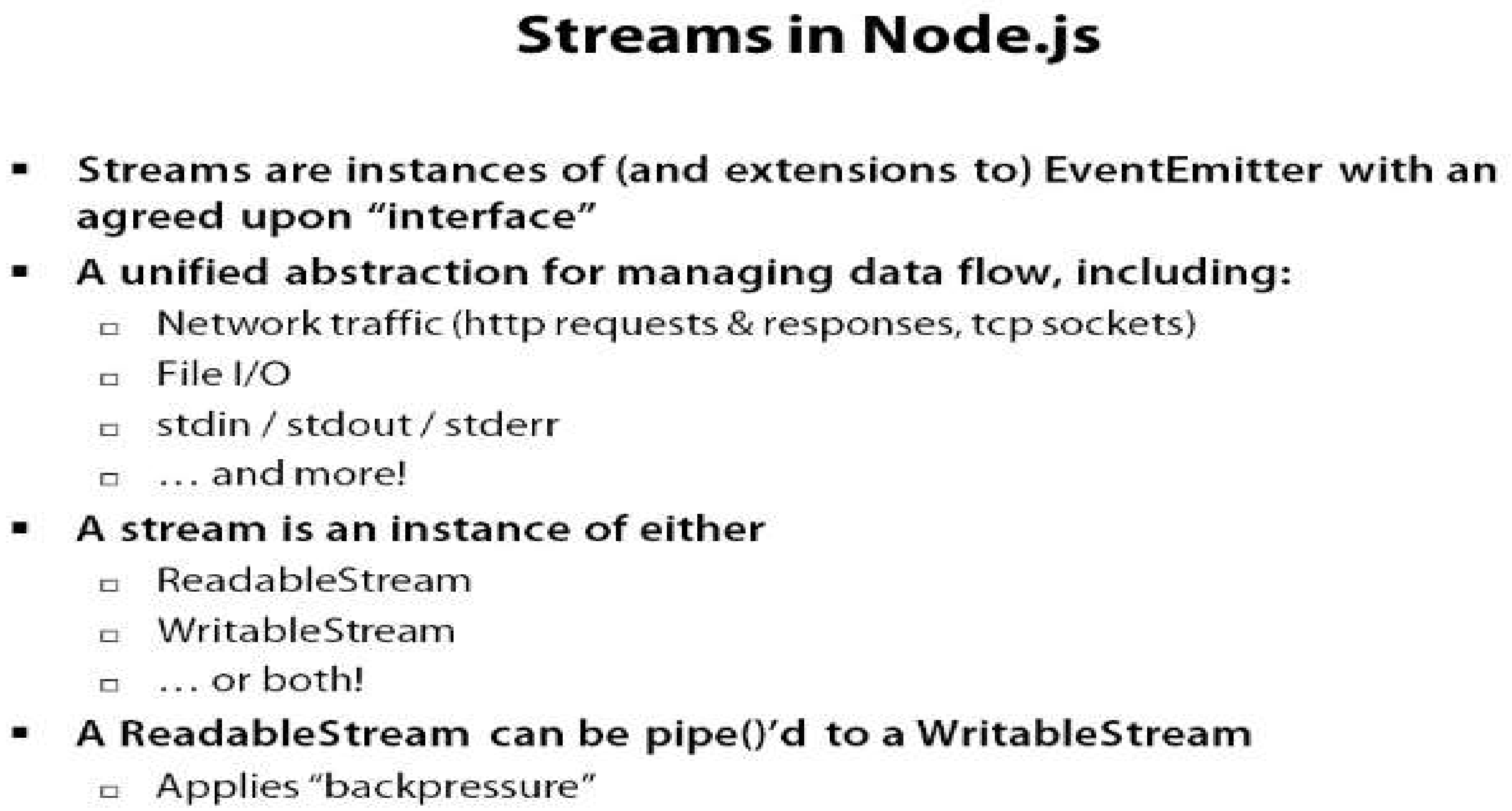


greeter1.on()

greeter1.emit()

greeter1.greet()

- Readable and Writable Streams, The pipe function
- The concept of eventEmitter , node calls it a stream



Streams in Node.js

- Streams are instances of (and extensions to) EventEmitter with an agreed upon "interface"
- A unified abstraction for managing data flow, including:
  - Network traffic (http requests & responses, tcp sockets)
  - File I/O
  - stdin / stdout / stderr
  - … and more!
- A stream is an instance of either
  - ReadableStream
  - WritableStream
  - … or both!
- A ReadableStream can be pipe()'d to a WritableStream
  - Applies "backpressure"

ReadableStream & WritableStream

ReadableStream

- readable [boolean]
- event: 'data'
- event: 'end'
- event: 'error'
- event: 'close'
- pause()
- resume()
- destroy()
- pipe()

WritableStream

- writable [boolean]
- event: 'drain'
- event: 'error'
- event: 'close'
- event: 'pipe'
- write()
- end()
- destroy()
- destroySoon()

Readable Streams

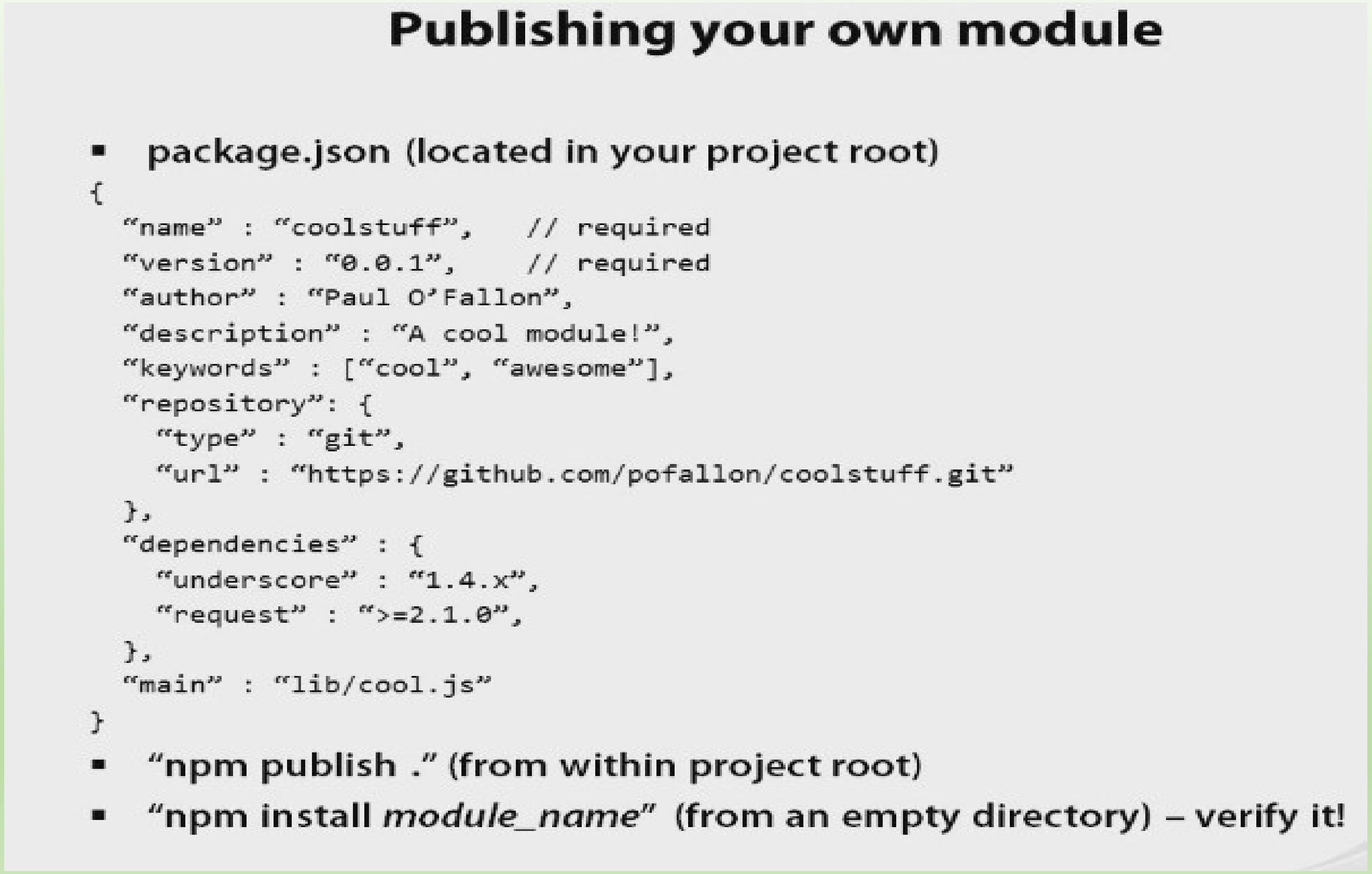
- Streams give us a way to asynchronously handle continuous data flows.
- Understanding how streams work will dramatically improve the way your application handles large data.
- Streams in Node.js are implementations of the underlying abstract extreme interface .
- Process standard input implements a readable stream.
- Whenever a data event is raised, some data is passed to the callback function.
- Streams can be readable, like stdin, writeable like standard output, or duplex, which means they are both readable and writeable.
- Streams can work with binary data or data encoded in a text format like UTF-8.

- Pipi ng Streams.
- The real power of streams comes with the pipe function.
- When the pipe function is invoked on a readable stream, you pass as a parameter the writeable stream you want to pipe to.
- This in turn emits the pipe event on the WritableStream.
- The pipe function then plans the execution of events and functions between the two streams
- When data arrives to the ReadableStream, the data event is emitted and the write function on the writable stream is invoked with this data.
- If the write function returns a false value indicating that no more data should be written, the pause function of the ReadableStream is called to stop the flow of data.

Conclusion

- Callbacks vs. Events
- Node's EventEmitter class
- Patterns for implementing EventEmitters
- Readable and Writable Streams
- Piping Between Streams

- Publishing your own module

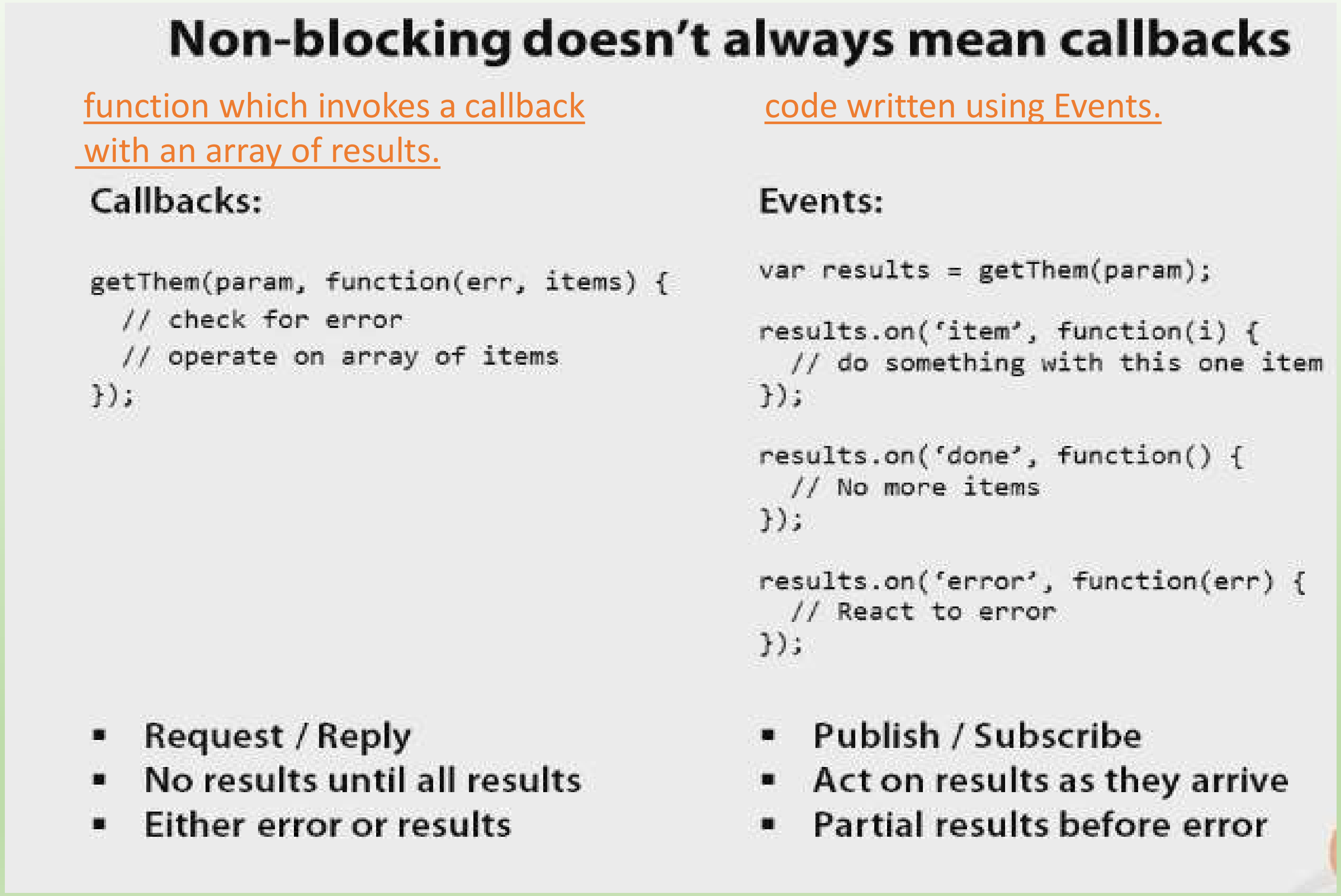


Publishing your own module

- package.json (located in your project root)

```
{
  "name" : "coolstuff",    // required
  "version" : "0.0.1",     // required
  "author" : "Paul O'Fallon",
  "description" : "A cool module!",
  "keywords" : ["cool", "awesome"],
  "repository": {
    "type" : "git",
    "url" : "https://github.com/pofallon/coolstuff.git"
  },
  "dependencies" : {
    "underscore" : "1.4.x",
    "request" : ">=2.1.0",
  },
  "main" : "lib/cool.js"
}
```

- "npm publish ." (from within project root)
- "npm install *module_name*" (from an empty directory) – verify it!

- Events And Streams
- 

Non-blocking doesn't always mean callbacks

function which invokes a callback with an array of results.

Callbacks:

```
getThem(param, function(err, items) {
  // check for error
  // operate on array of items
});
```

- Request / Reply
- No results until all results
- Either error or results

code written using Events.

Events:

```
var results = getThem(param);

results.on('item', function(i) {
  // do something with this one item
});

results.on('done', function() {
  // No more items
});

results.on('error', function(err) {
  // React to error
});
```

- Publish / Subscribe
- Act on results as they arrive
- Partial results before error

<u>**Function which invokes a callback with an array of results.**</u>	<u>**Code written using Events.**</u>
In the callback model , you make a request and provide a function to be called when the request is completed. One request, one reply.	The Event Model is more of a publish/subscribe approach.
In the Callback approach , result is not received until you receive all the results.	The On function can be invoked repeatedly to provide multiple functions to invoke on each event; in essence, subscribing to the events.
the callback will not be invoked until the entire items array is ready.	In the evented example above, functions associated with the item event will be invoked for each item.
If these items arrive slowly, the callback will not be invoked until the last item has arrived.	This gives you the opportunity to act on the first item as soon as it arrives and the second item and so forth.
It also means that the getThem function will be storing the entire list of items in memory while accumulating them prior to invoking the callback with the entire array.	It also means that the getThem function is not accumulating the items in memory.

Module 4

Accessing The Local System

- Introduction, The Process Object
- The **process** object provides a way for the node application to both manage its own process as well as the other processes on the system.
- Process object is available by default in the node application, hence no need to be required
- Process object has a variety of functions and variables including a set of streams for accessing standard in, out and error.
- The process object also provides a series of attributes about the current process such as environment variables, command line arguments,   processId and title etc.
- The process object is an instance of eventEmitter class.
- It emits an exit event when the process is about to exit

The “process” object

- **A collection of Streams**
  - process.stdin
  - process.stdout
  - process.stderr

- **Attributes of the current process**
  - process.env
  - process.argv
  - process.pid
  - process.title
  - process.uptime()
  - process.memoryUsage()
  - process.cwd()

- **Process-related actions**
  - process.abort()
  - process.chdir()
  - process.kill()
  - process.setgid()
  - process.setuid()
  - … etc.

- **An instance of EventEmitter**
  - event: ‘exit’
  - event: ‘uncaughtException’
  - POSIX signal events (‘SIGINT’, etc.)

- <u>Interacting With the File System in Node.JS</u>
- Node uses a built-in module fs to interact with the file system
- Many of the functions provided by the fs module are wrappers around the POSIX functions and are both synchronous and asynchronous.
- The fs module also provides a few stream oriented functions.
- The fs module provides a watch function that watches a file or directory for changes

Interacting with the File System

- **Wrappers around POSIX functions (both async and sync versions)**
  - Functions include:

    rename, truncate, chown, fchown, lchown, chmod, fchmod, lchmod, stat, fstat, lstat, link, symlink, readlink, realpath, unlink, rmdir, mkdir, readdir, close, open, utimes, futimes, fsync, write, read, readFile, writeFile, and appendFile
  - For example: `fs.readdir(path, callback)` and `fs.readdirSync(path)`
- **Stream oriented functions**
  - `fs.createReadStream()` – returns an fs.ReadStream (a ReadableStream)
  - `fs.createWriteStream()` – returns an fs.WriteStream (a WritableStream)
- **Watch a file or directory for changes**
  - `fs.watch()` – returns an fs.FSWatcher (an EventEmitter)
  - 'change' event: the type of change and the filename that changed
  - 'error' event: emitted when an error occurs

- What is Buffer?
- JavaScript has difficulty in dealing with binary data, but when we are interacting with the network and file systems it is required to work with binary data.
- The buffer class provides a raw memory allocation for dealing with binary data directly.
- Buffers can be converted to/from strings by providing an encoding
  - Ascii, utf8(default), utf16le, ucs2, base64, binary, hex
- The support for multiple encodings makes buffers a handy way to convert strings to/from base64

Module 5

Interacting With The Web

- The **HTTP module** will be used for creating web servers, for making requests, for handling responses.
- There are two modules for this. There's the **HTTP module** and the **HTTPS module**.
- Both of these modules are very similar, but HTTPS module is used when we're working with a secure server.
- So that means if we want to create an HTTPS server, we would use the HTTPS module, and then we would have to supply the security certificate.
- With the HTTP module, there's no need to supply a security certificate.

Making web requests in Node

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

Instance of http.ClientRequest (a WritableStream)

```
var req = http.request(options, function(res) {
  // process callback
});
```

Instance of http.ClientResponse (a ReadableStream)

- "options" can be one of the following:
  - A URL string
  - An object specifying values for host, port, method, path, headers, auth, etc.
- The returned ClientRequest can be written/piped to for POST requests
- The ClientResponse object is provided via either callback (shown above) or as a "response" event on the request object.
- http.get() available as a simplified interface for GET requests

Building a Web Server in Node

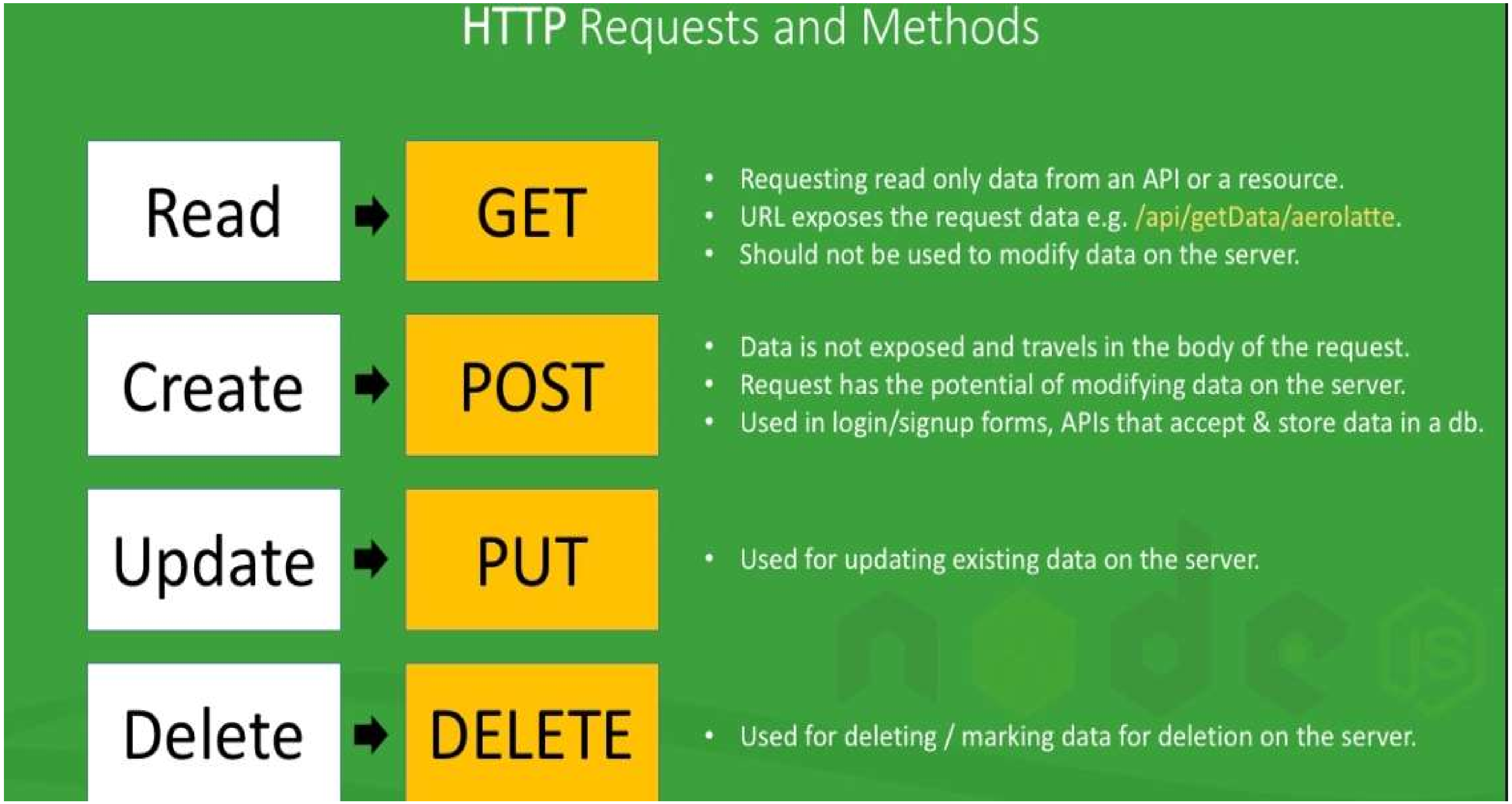
```
var http = require('http');
```

Instance of http.ServerRequest (a ReadableStream)

```
var server = http.createServer(function(req, res) {
  // process request
});
server.listen(port, [host]);
```

Instance of http.ServerResponse (a WritableStream)

- Each request is provided via either callback (shown above) or as a "request" event on the server object
- The ServerRequest can be read from (or piped) for POST uploads
- The ServerResponse can be piped to when returning stream-oriented data in a response
- SSL support is provided by a similar http<u>s</u>.createServer()



HTTP Requests and Methods

Read ➡ GET

- Requesting read only data from an API or a resource.
- URL exposes the request data e.g. /api/getData/aerolatte.
- Should not be used to modify data on the server.

Create ➡ POST

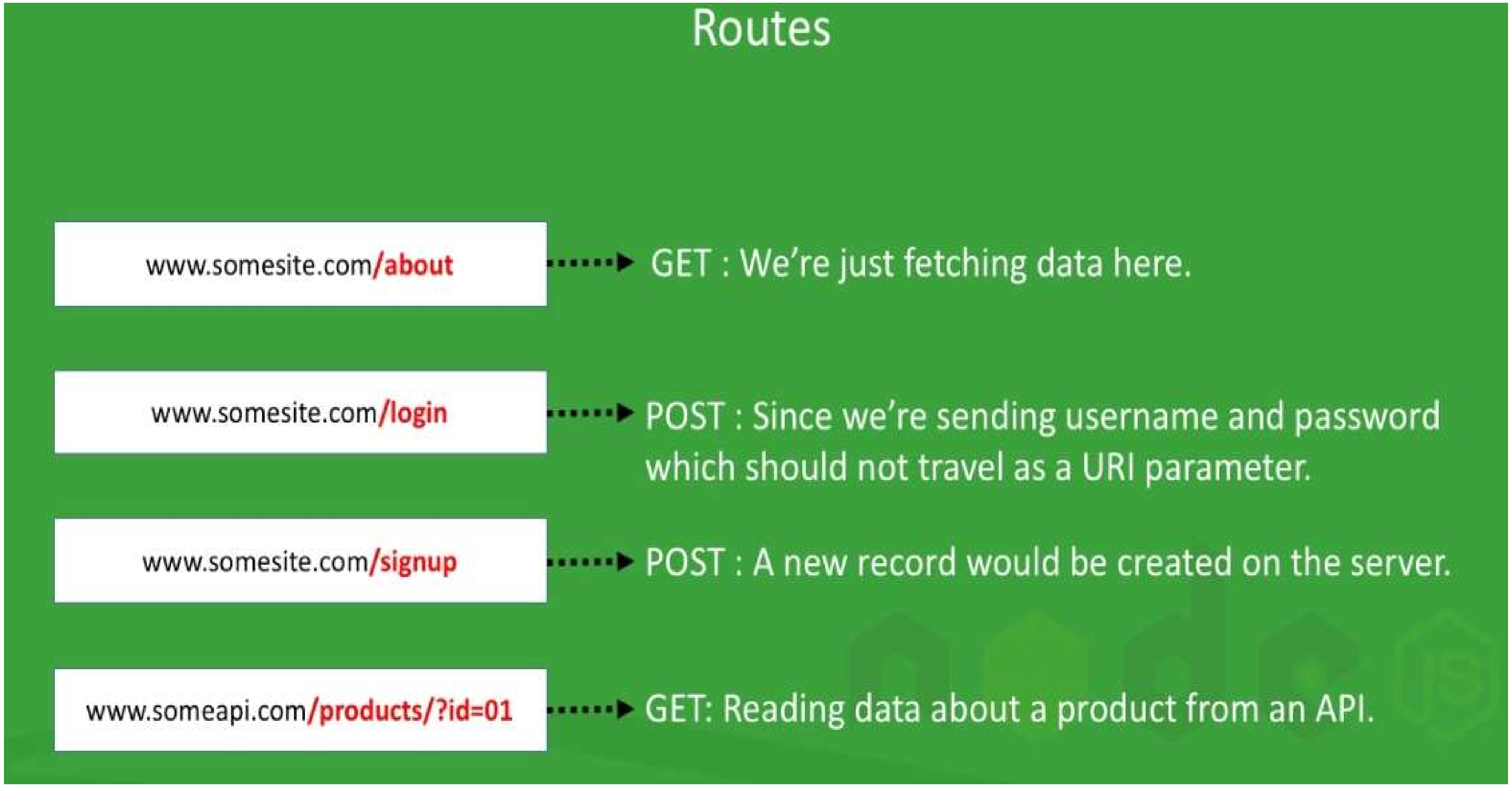
- Data is not exposed and travels in the body of the request.
- Request has the potential of modifying data on the server.
- Used in login/signup forms, APIs that accept & store data in a db.

Update ➡ PUT

- Used for updating existing data on the server.

Delete ➡ DELETE

- Used for deleting / marking data for deletion on the server.



Routes

www.somesite.com**/about**	GET : We're just fetching data here.
www.somesite.com**/login**	POST : Since we're sending username and password which should not travel as a URI parameter.
www.somesite.com**/signup**	POST : A new record would be created on the server.
www.someapi.com**/products/?id=01**	GET: Reading data about a product from an API.

The **req** object

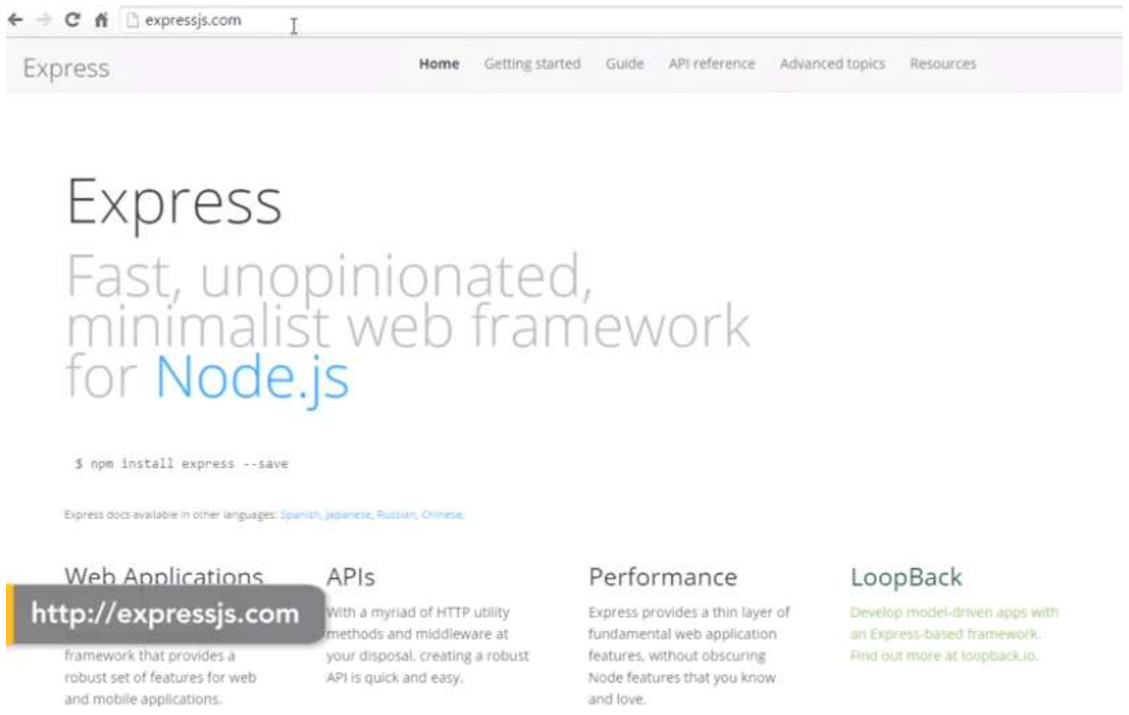
```
http://localhost:3000/api/products?id=021&color=Red&color=Orange&sortBy=price
```

```
req.url        : /api/products?id=021&color=Red&color=Orange&sortBy=price
req.method     : GET
req.headers    : {
                   host: 'localhost:3000’
                   connection: 'keep-alive',
                   'cache-control': 'max-age=0',
                   accept: 'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*,
                   'upgrade-insecure-requests': '1’
                 }
```

- <u>Exploring Web Frameworks</u>
- A framework is an essential supporting structure of a building, vehicle, or object.
- And in software, it's essentially the same thing.
- It is a supporting structure that allows you to build on top of it.
- When it comes to web, and we want to build large APIs, or maybe HTTP servers, we can leverage web frameworks, and there are several options we can look at.

- Exploring Web Frameworks
- Each of these provide us with the structure and components to quickly make serving static files, like traditional websites, easy.
- We can put together a web API to interact in a web app.
- A web API is a service that allows us to get and save data to our server or back end, such as a web API that allows us to maybe create users, serve a list of users etc.
- Now let's take a look at the different options we have for web frameworks for Node.
- We'll be looking at Express, which is a very traditional framework.

• EXPRESS



Express

Fast, unopinionated, minimalist web framework for Node.js

```
$ npm install express --save
```

Express docs available in other languages: Spanish, Japanese, Russian, Chinese.

Web Applications

http://expressjs.com

framework that provides a robust set of features for web and mobile applications.

APIs

With a myriad of HTTP utility methods and middleware at your disposal, creating a robust API is quick and easy.

Performance

Express provides a thin layer of fundamental web application features, without obscuring Node features that you know and love.

LoopBack

Develop model-driven apps with an Express-based framework. Find out more at loopback.io.

Express.js

- Runs within Node.js, and is just on the back end
- Is well-supported and well-documented
- Uses more traditional development features

- Express

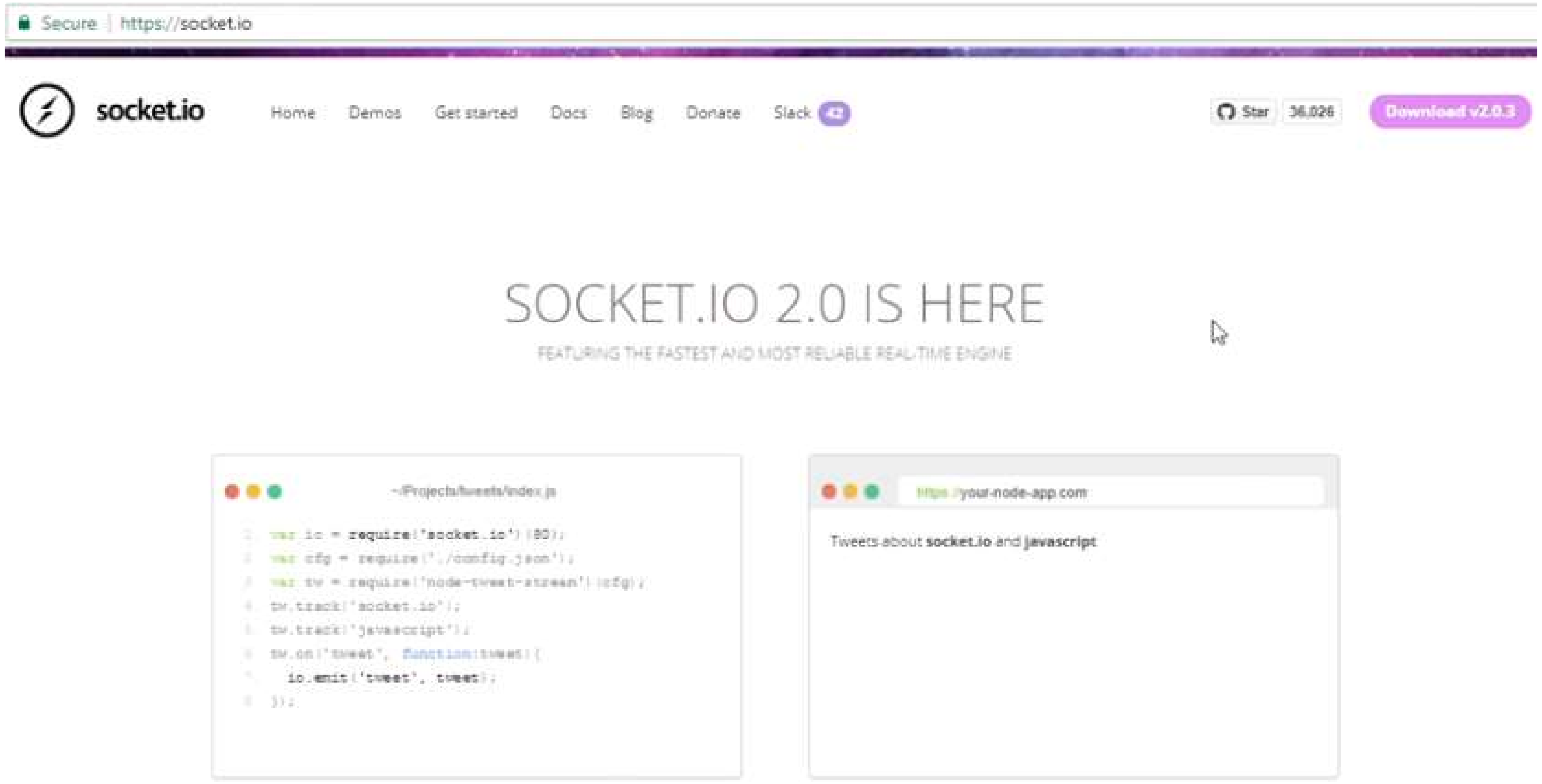
- The express framework provides an abstraction layer above the vanilla http module to make handling web traffic and APIs a little easier.
- There's also tons of middleware available for express (and express-like) frameworks to complete common tasks such as: CORS, XSRF, POST parsing etc.
- The express() function is a top-level function exported by the express module.

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

- Express
- **express.static(root, [options])**
- This is a built-in middleware function in Express.
- It serves static files and is based on serve-static.

- The root argument specifies the root directory from which to serve static assets.
- The function determines the file to serve by combining req.url with the provided root directory.
- When a file is not found, instead of sending a 404 response, it instead calls next() to move on to the next middleware, allowing for stacking and fall-backs.

- Socket IO



SOCKET.IO 2.0 IS HERE

FEATURING THE FASTEST AND MOST RELIABLE REAL-TIME ENGINE

~/Projects/tweets/index.js

```
var io = require('socket.io')(80);
var cfg = require('./config.json');
var tw = require('node-tweet-stream')(cfg);
tw.track('socket.io');
tw.track('javascript');
tw.on('tweet', function(tweet){
  io.emit('tweet', tweet);
});
```

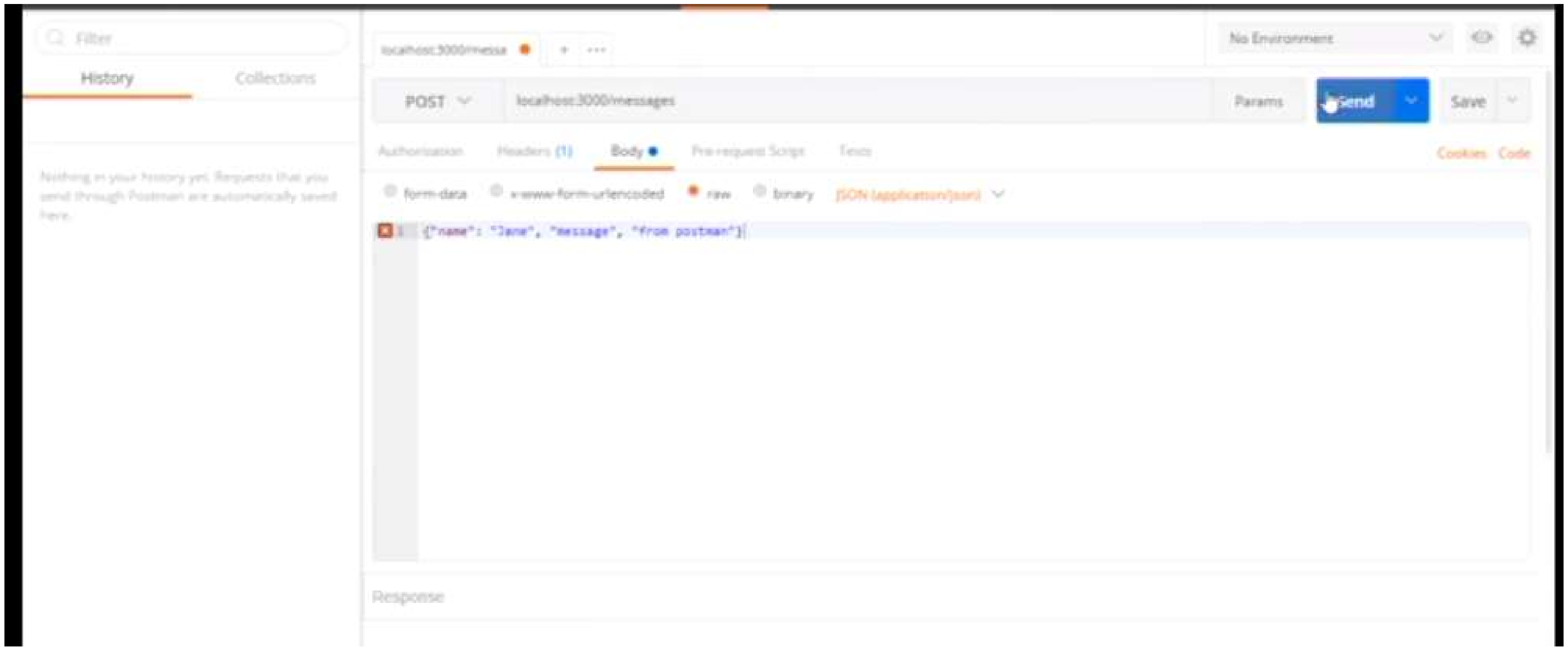
https://your-node-app.com

Tweets about **socket.io** and **javascript**

- Socket.io enables real time, bidirectional, event based communication.
- Express on the other hand allows the client to send a request to the server, but the server cannot send request to the client and so it does not have bidirectional communication.
- Socket.io solves this. That is, we can push notifications from the server to the client when an event happens, as well as other data.
- Socket.io has two parts, a client side library that runs on the browser, and a server side library for node.js.
- Both components have an API that's nearly identical.
- Just like node.js, it is event driven.

- Serving Files
- When creating a web application, we can think of it as having a division between two responsibilities.
- we can either host static content or dynamic content.
- Static content can be things like HTML files for web sites or images, videos, etc.
- Dynamic content, on the other hand, is served through a web API, or sometimes it's used to serve dynamic webpages where the content or the view is composed on the server itself.

- Post Method



POST localhost:3000/messages

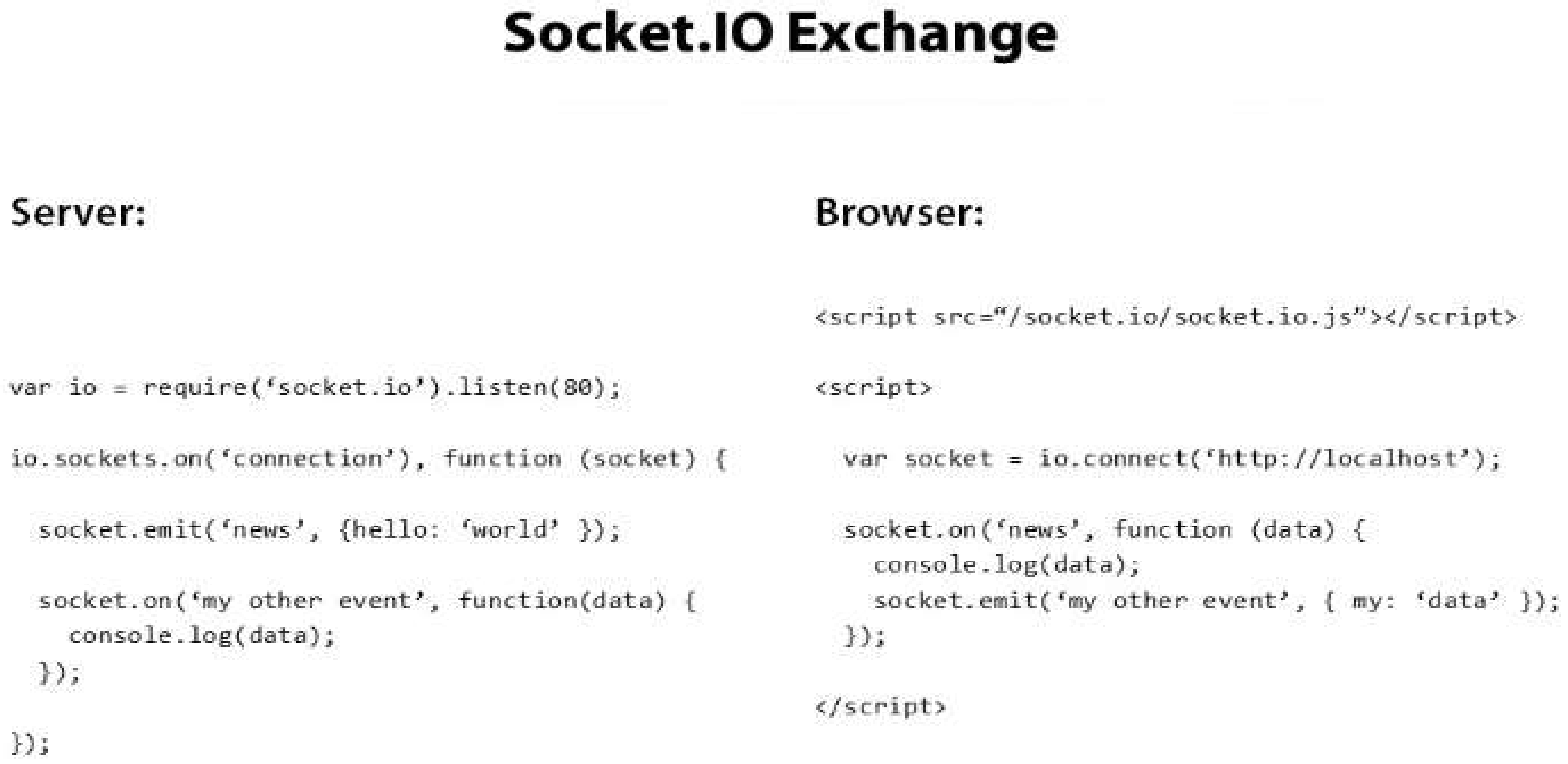
Body: raw, JSON (application/json)

```
{"name": "Jane", "message", "from postman"}
```

- Web Sockets
- - Web Sockets are a wonderful addition to the HTML5 Spec.
- They allow for a true two way connection between the client and the server.
- Web Sockets use their own protocol to send and receive messages from a TCP server.
- Until recently, WebSockets were not part of the step.
- We had no way push information from the server to the browser. The browser had to constantly check the server API by making a GET request to see if the state of the server has changed, and this is called polling.

- Long polling :
- This is the process of checking the server to see if the state has changed.
- A long polling consists of making a request to a server and leaving it open for a longer period of time.
- We let that request time out when information hasn't changed.
- With long polling, if information changes on the server, we can immediately receive a response with the changed information.
- Long polling is just a more efficient way of polling.

- Real Time interaction with socket.io



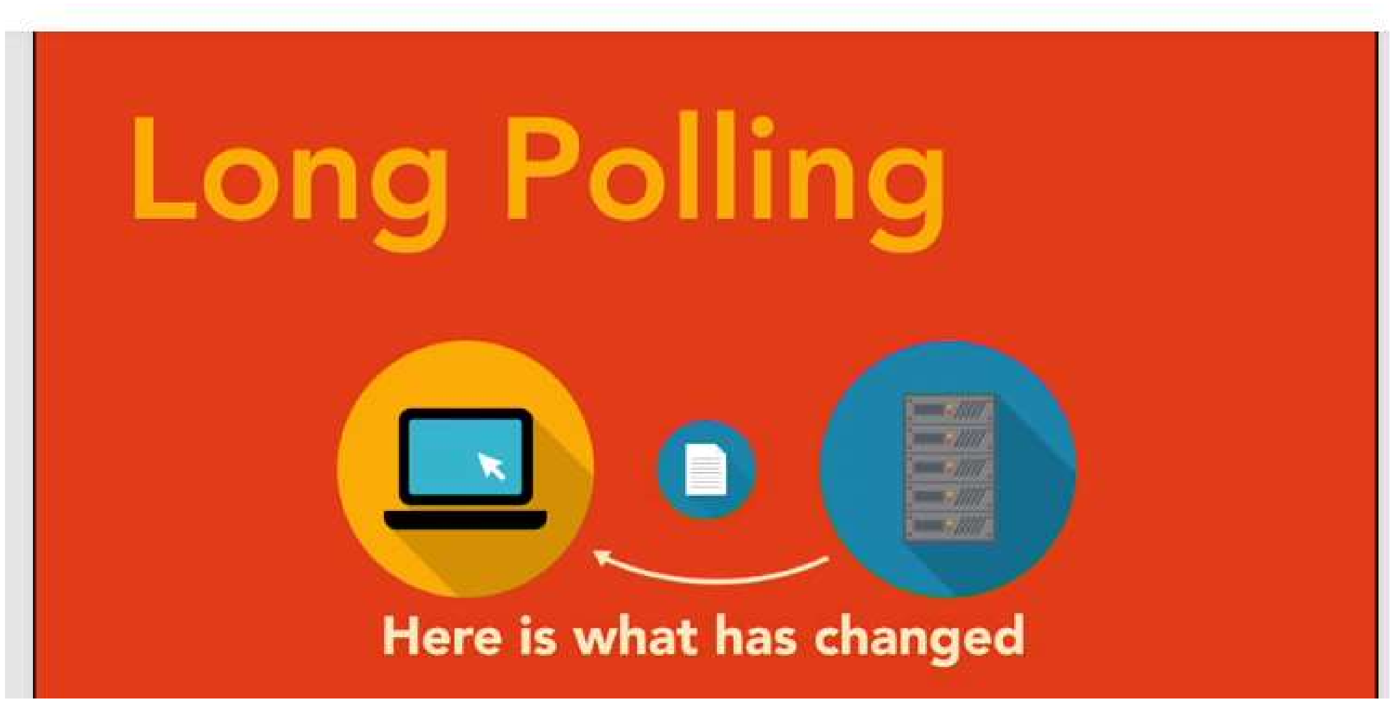
Socket.IO Exchange

Server:

```
var io = require('socket.io').listen(80);

io.sockets.on('connection'), function (socket) {

  socket.emit('news', {hello: 'world' });

  socket.on('my other event', function(data) {
    console.log(data);
  });

});
```

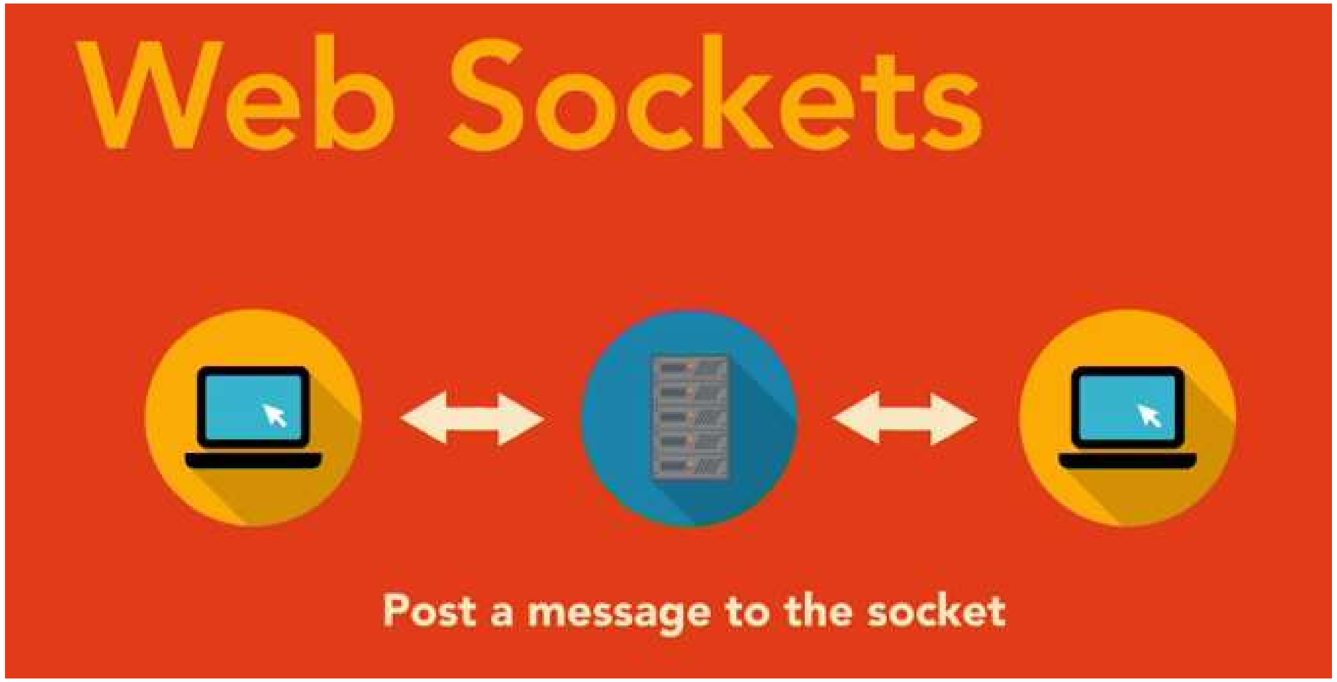
Browser:

```
<script src="/socket.io/socket.io.js"></script>

<script>

  var socket = io.connect('http://localhost');

  socket.on('news', function (data) {
    console.log(data);
    socket.emit('my other event', { my: 'data' });
  });

</script>
```


Long Polling
Here is what has changed

- With Web Sockets we can connect to server and leave the connection open so that we can send and receive data.
- Web Sockets are not just limited to the browser.
- With Web Sockets, clients can connect to the server and leave a two way connection open.
- Through this connection, clients can send data that are easily broadcasted to every open connection.
- With connection always open server is able to push data chain changes to the client using Web Sockets.
- Web Sockets are not just limited to the browser.
- Any client can connect to the server including native applications.
- Now instead of HTTP, Web Sockets use their own protocol.
-

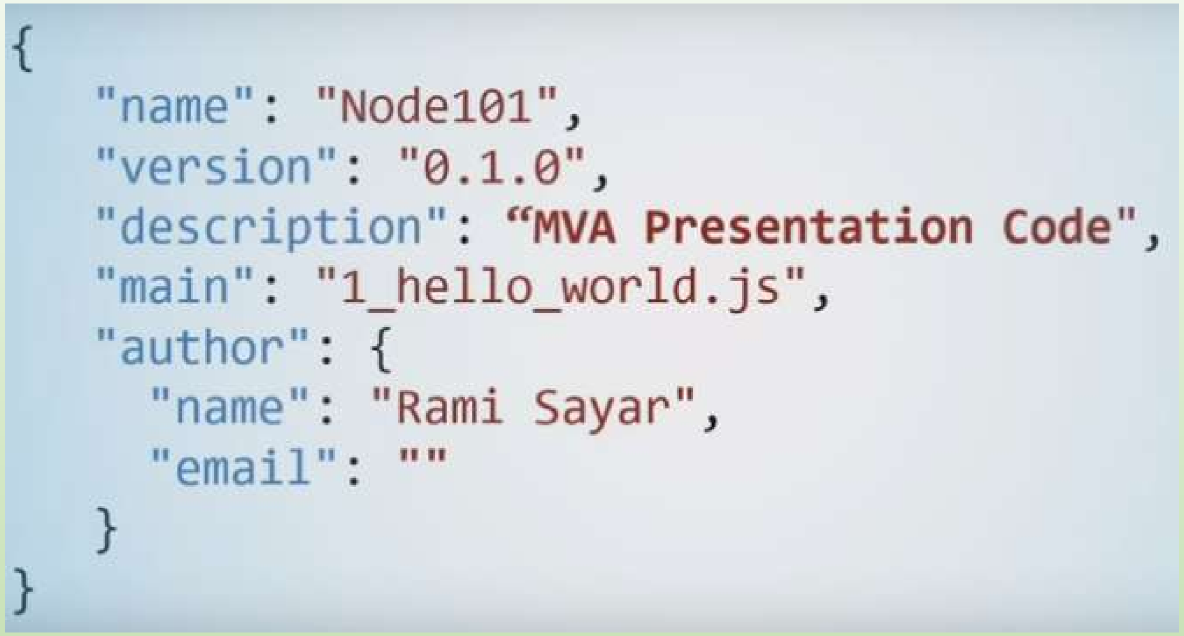
- Setting up a Web Sockets from scratch on your web application can be a little tricky You need a TCP Socket server and a HTTP proxy.
- Fortunately, there are node modules that will help us with building our non Web Sockets.



- NPM (Node Package Manager)
- It is the official package manager for node
- Bundled and installed automatically with the node environment

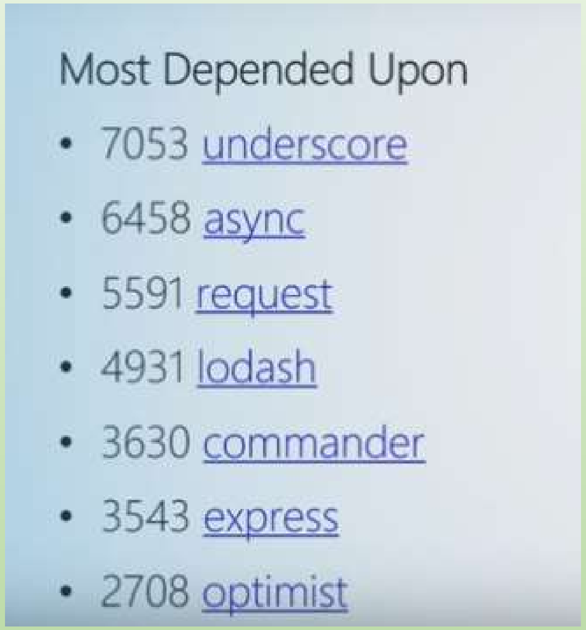
- Frequent usage
- npm install –save package_name [save will add the package name in package.json file]
- npm update

- The package.json file



```
{
   "name": "Node101",
   "version": "0.1.0",
   "description": “MVA Presentation Code",
   "main": "1_hello_world.js",
   "author": {
     "name": "Rami Sayar",
     "email": ""
   }
}
```

Some popular npm modules

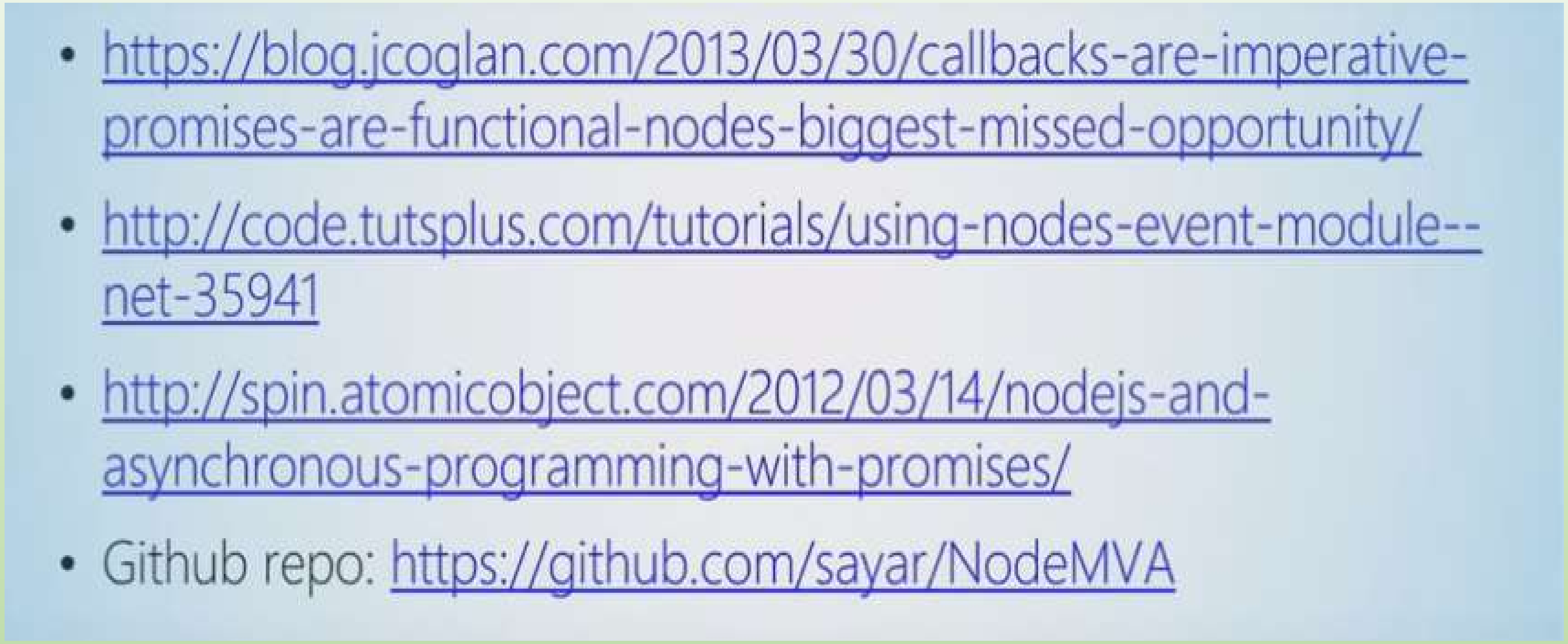


Most Depended Upon

- 7053 underscore
- 6458 async
- 5591 request
- 4931 lodash
- 3630 commander
- 3543 express
- 2708 optimist

- How package.json works
- It reads package.json
- Installs the dependencies in the local node_modules folder
- In global mode, it makes a node module accessible to all
- Can specify dev or optional dependencies

- Resources



- https://blog.jcoglan.com/2013/03/30/callbacks-are-imperative-promises-are-functional-nodes-biggest-missed-opportunity/
- http://code.tutsplus.com/tutorials/using-nodes-event-module--net-35941
- http://spin.atomicobject.com/2012/03/14/nodejs-and-asynchronous-programming-with-promises/
- Github repo: https://github.com/sayar/NodeMVA

- Module 2

- What is express
- Installing & using express
- Demo: creating a rest api
- Templating

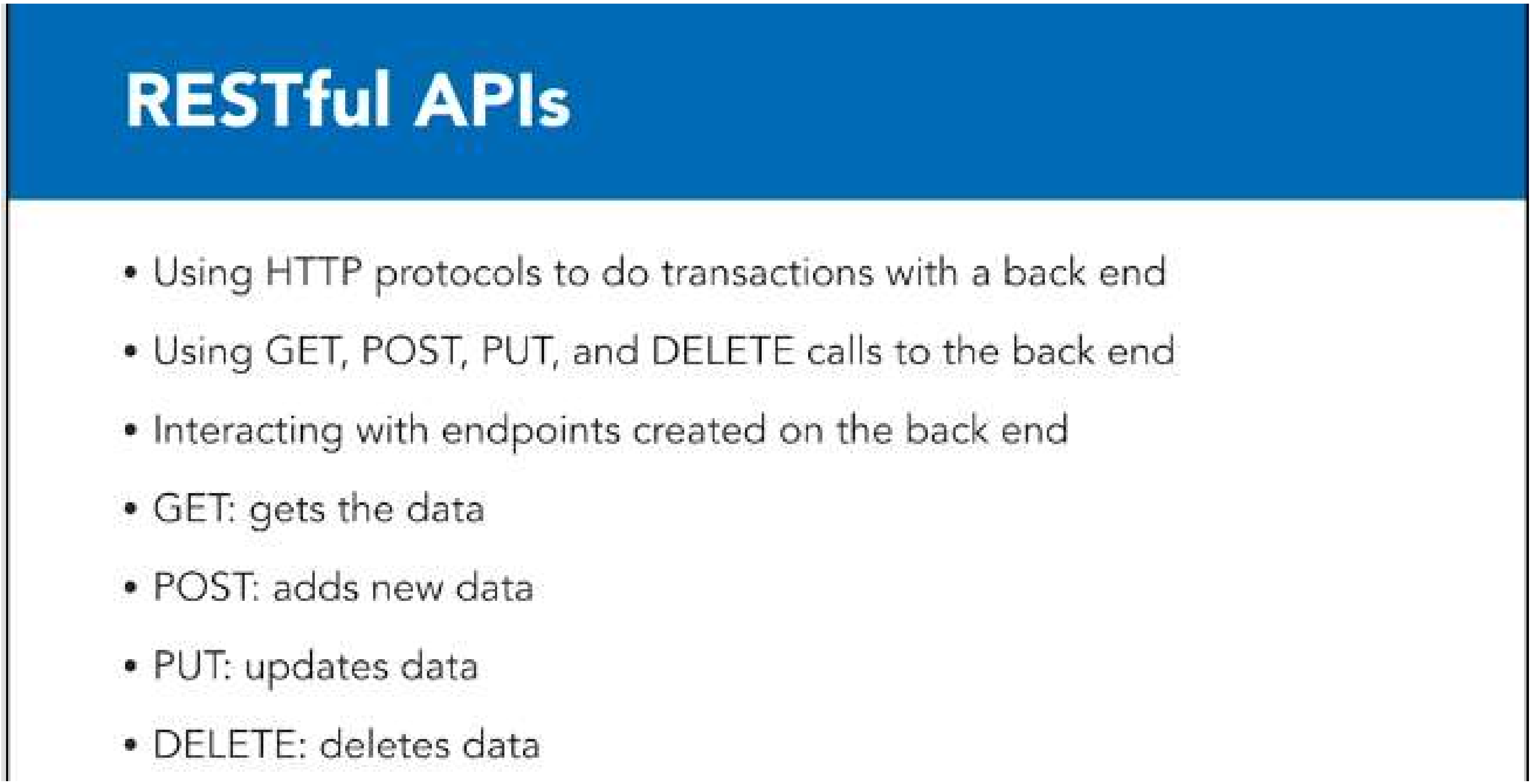
- What is Express
- Express is a web development framework for node.js.
- It is a web piece that will be built inside Node.js
- Express is very light weight minimalist, un-opinionated, open source and flexible node.js web app framework designed to make developing websites, web apps and API’s much easier.
- It structures your web application.
- Express structures your web application so that you can handle multiple url’s, multiple http request whether its GET, PUT, POST or DELETE in a very simple way
- Express is used for node on the server mostly to handle or structure the http calls.
- Express is used strictly on the backend

- Why use Express?
- Express helps you respond to requests with route support so that you can write responses to specific url’s
- Supports multiple templating engines to simplify generating HTML
- Express is simple to use with less lines of code
- Installing Express
  - npm install express
  - npm install jade [jade is a templating language]

- Creating a REST API
  - Explanation of routes
  - A router maps HTTP requests to a callback.
  - HTTP requests can be sent as GET/POST/PUT/DELETE
  - URLs describe targeted
  - Node helps to map a HTTP GET request like:
  - - http://localhost:8888/index
  - To a request handler (callback)
    - App.get('/index',function(req,res){});

- Express Middleware:
- https://expressjs.com/en/guide/using-middleware.html
- Express is a routing and middleware web framework that has minimal functionality of its own:
- An Express application is essentially a series of middleware function calls.
- Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application’s request-response cycle.
- The next middleware function is commonly denoted by a variable named next

- Express Middleware
- Middleware functions can perform the following tasks:
  - Execute any code.
  - Make changes to the request and the response objects.
  - End the request-response cycle.
  - Call the next middleware function in the stack.
- If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function, else the request will be left hanging.



RESTful APIs

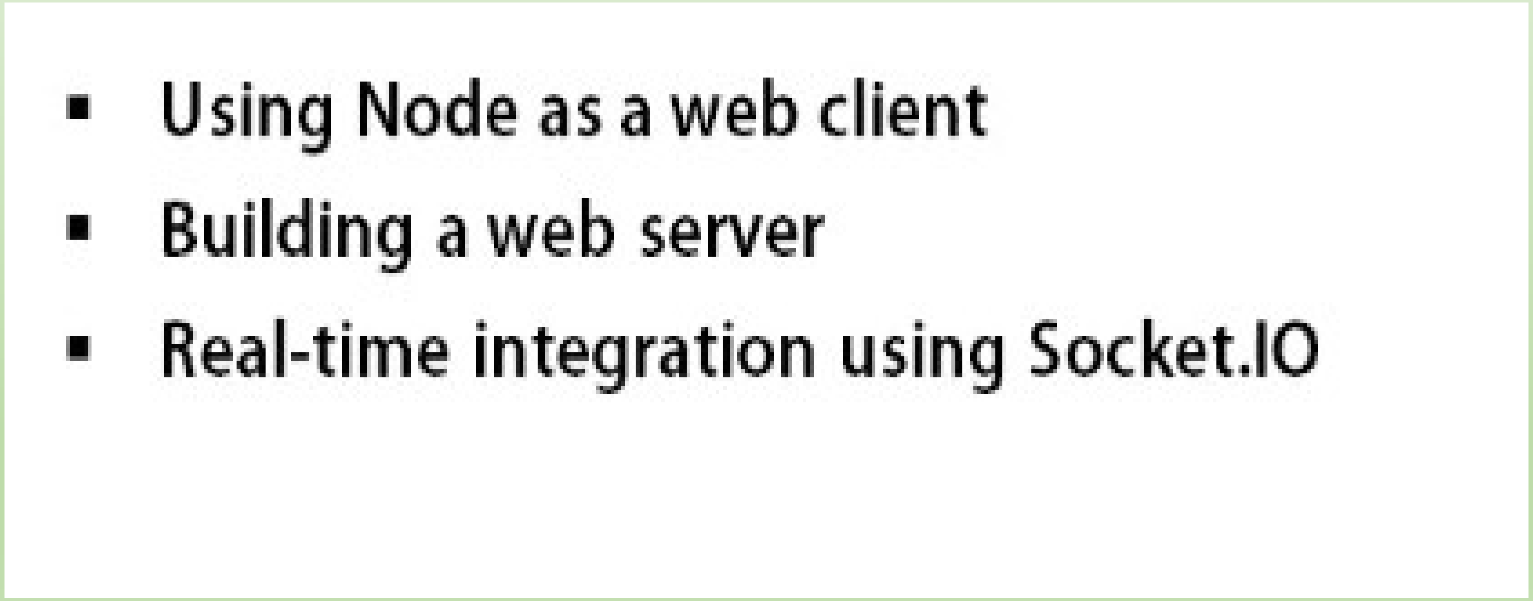
- Using HTTP protocols to do transactions with a back end
- Using GET, POST, PUT, and DELETE calls to the back end
- Interacting with endpoints created on the back end
- GET: gets the data
- POST: adds new data
- PUT: updates data
- DELETE: deletes data

- ExpressJS is a very popular module that adds web server functionality to NodeJS.
- It also makes it easier to build websites and applications.
- It creates a routing mechanism so that your applications can pass along different types of requests easily.
- It also makes it easier to create an API or an application programming interface that can help you deal with data using HTTP verbs like get, put, and delete.
- Express also allows you to plug in other modules into the middle of the framework to perform certain types of tasks.

The urlencoded method within body-parser tells body-parser to extract data from the <form> element and add them to the body property in the request object.
Now, you should be able to see everything in the form field within the req.body object.

- Interacting With the Web

- In this module we will see the following



- Using Node as a web client
- Building a web server
- Real-time integration using Socket.IO

- Making Web Requests in Node
- We have already seen the usage of third party request module to make an http request to a particular website.
- http module also has a request function which takes two parameters:
  - An options parameter and a callback
- The options parameter can be an url string or a more complex object specifying many options about the request being made.
- The request function returns a value which is an instance of client request.
- This is a writable stream and can be written or piped to for http post uploads
- In addition to returning a value, the request function also takes a callback parameter .

- Making web request in Node contd…….
- The callback of request function when invoked, is passed as a single parameter, and instance of client response which represents the results of the http request.
- The client response is a readable stream which can be read from or piped to a writable stream.
- The callback of the request method of http , is an example where Node does not follow the convention, since the first parameter to the callback is not an error.
- If you don’t pass the callback to the request function you can still retrieve the client response object.

Making web requests in Node

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

Instance of http.ClientRequest (a WritableStream)

```
var req = http.request(options, function(res) {
  // process callback
});
```

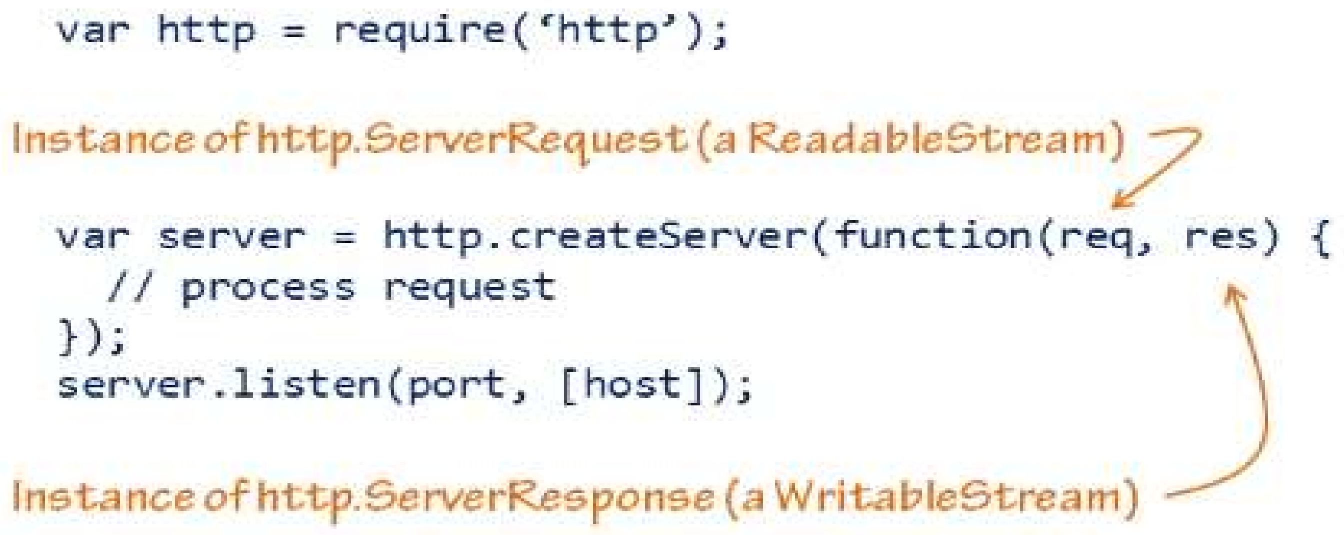
Instance of http.ClientResponse (a ReadableStream)

- **"options" can be one of the following:**
  - A URL string
  - An object specifying values for host, port, method, path, headers, auth, etc.
- **The returned ClientRequest can be written/piped to for POST requests**
- **The ClientResponse object is provided via either callback (shown above) or as a "response" event on the request object.**
- **http.get() available as a simplified interface for GET requests**

- **Building a Web server in Node**
- We have already seen the usage of http module to create a server.
- The createServer function of http takes a single parameter .
- The callback is invoked each time a request is received by the web server.
- If no callback is provided requests can also be received by listening for events on the server object that is returned.
- The server will not begin to accept the HTTP requests until the listen function is called.
- When a request is made to the HTTP server and the callback is invoked, it is passed two parameters (req, res)
- The first is an instance of server requests, which is a readable stream.
- It represents the request being made and for uploads to the server it can be read from

- Building Web Server With Node contd….
- The second parameter passed to the callback on each web request is the server response object which is a writable stream.
- This represents the response sent to the client
- If you are returning stream oriented data, such as a file from disk, you can pipe that stream to a server response writable stream.
- Ports for SSL is done through the HTTPs module which is similar to the createServer function.

Building a Web Server in Node

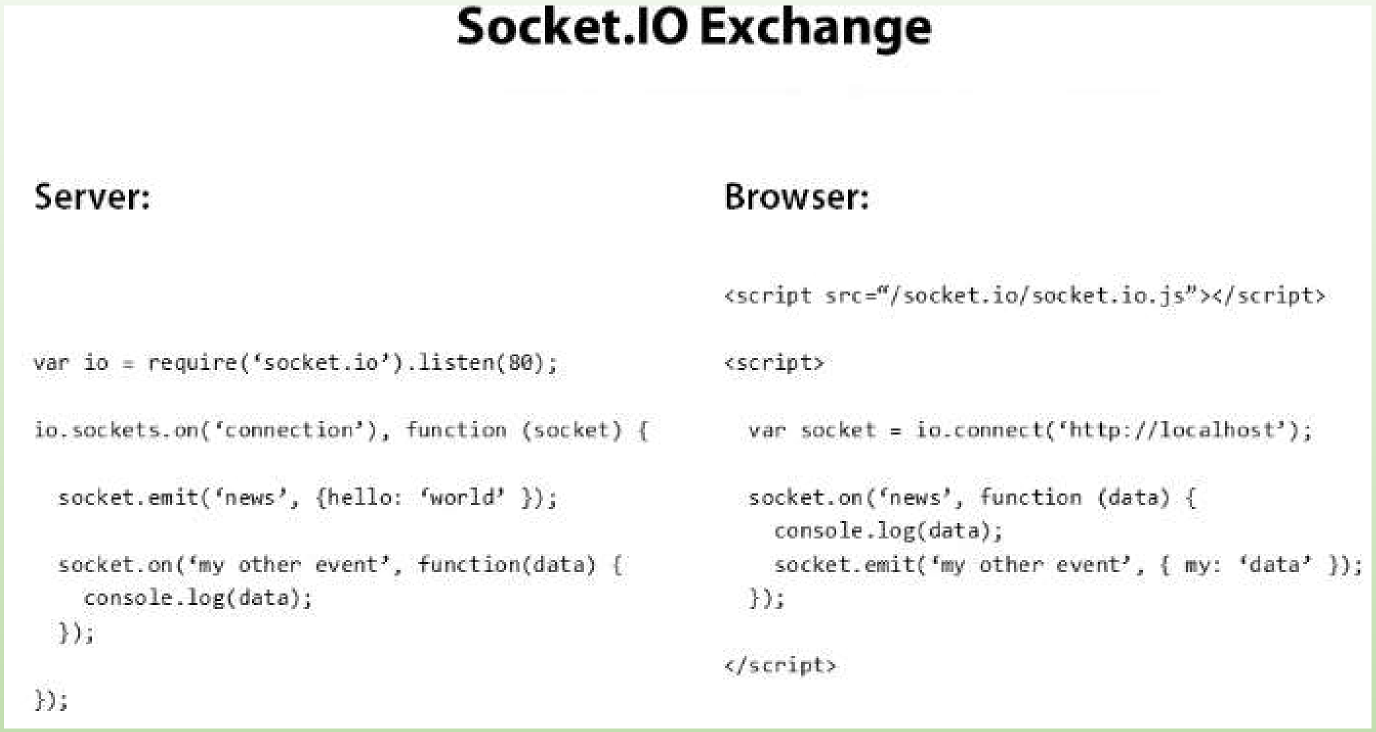


```
var http = require('http');

var server = http.createServer(function(req, res) {
  // process request
});
server.listen(port, [host]);
```

Instance of http.ServerRequest (a ReadableStream)
Instance of http.ServerResponse (a WritableStream)

- Each request is provided via either callback (shown above) or as a "request" event on the server object
- The ServerRequest can be read from (or piped) for POST uploads
- The ServerResponse can be piped to when returning stream-oriented data in a response
- SSL support is provided by a similar https.createServer()

• Real-Time Communication With Socket.IO


Socket.IO Exchange

Server:

```
var io = require('socket.io').listen(80);

io.sockets.on('connection'), function (socket) {

  socket.emit('news', {hello: 'world' });

  socket.on('my other event', function(data) {
    console.log(data);
  });

});
```

Browser:

```
<script src="/socket.io/socket.io.js"></script>

<script>

  var socket = io.connect('http://localhost');

  socket.on('news', function (data) {
    console.log(data);
    socket.emit('my other event', { my: 'data' });
  });

</script>
```

The node.js that runs on the server, starts by requiring the Socket.IO module in invoking the listen function.
It is then using the EventEmitter construct to listen to and emit events

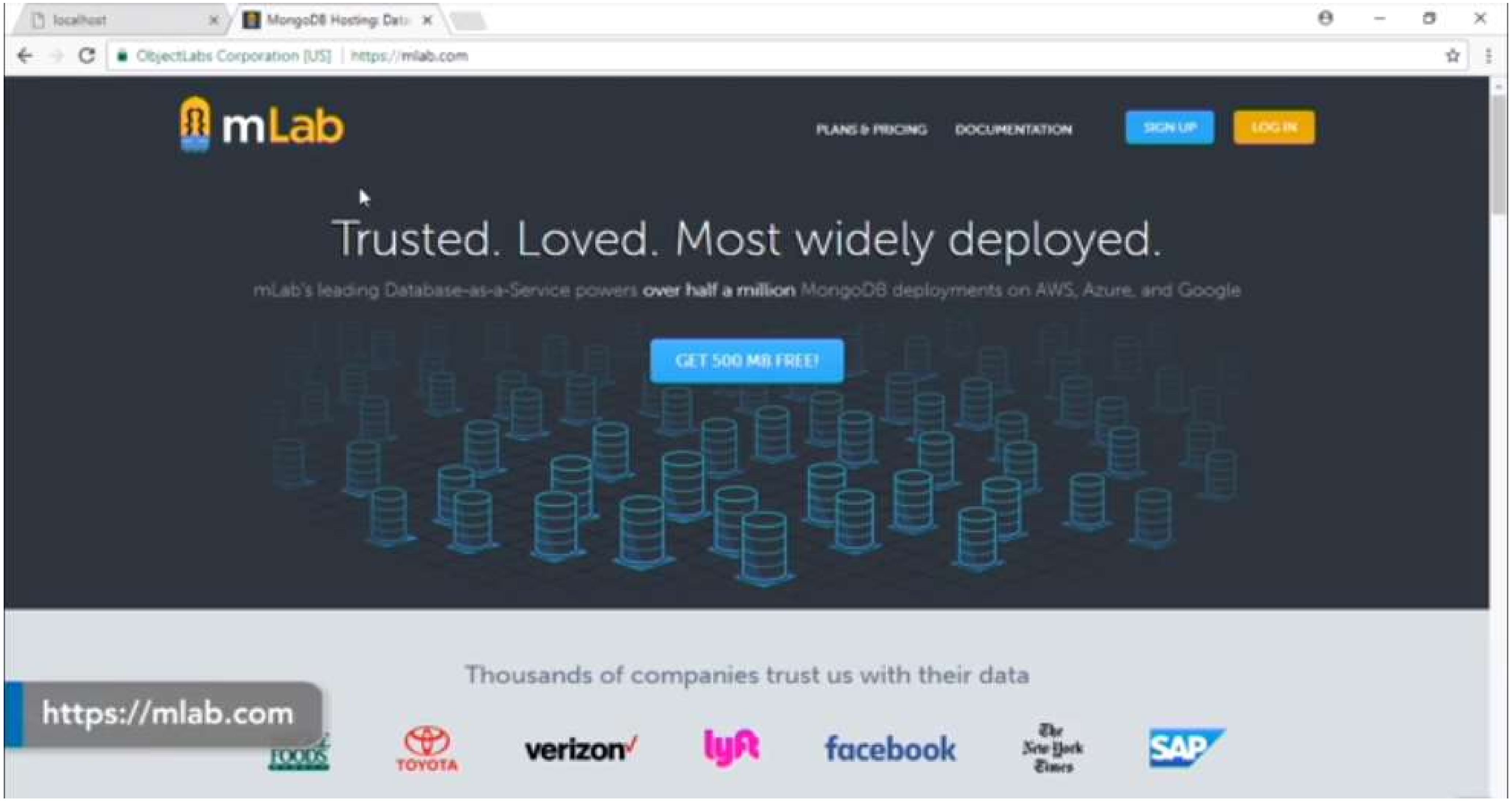
On the browser side, the socket.io JS library is loaded from the server.
There is no special configuration on the server to provide the JS file.
The JS file is handled transparently by the socket.IO node.js module.
Steps:

1. The browser will issue io.connect to eshtablish a connection to node.js server
2. 2. The server receives a connection event and emits a payload –'Hello World'.
3. The browser receives the event and invokes the appropriate function.
4. The browser then emits the event 'myServerData' and provides some data.
5. This is received by the node.js server and the appropriate function is invoked.

- Socket.IO Exchange contd…
- The powerful thing about this concept(prev Slide) is that both, the browser and the server are using the same constructs for emitting and acting on messages being passed back and forth.
- The code on both browser and server looks very similar

- Exploring Databases
- There are many different database technologies, such as MongoDB, MySQL, Veras, and many more.
- Node has packages to work with most of these options, most of these options fall into two categories: Either SQL or NoSQL.
- SQL stands for structured query language.
- The key word here is structured, meaning the data stored inside is structured.
- The database knows and cares about the structure, and the tables that store the data must be designed with that in mind.
- The object classes, which represent the items that will go into those structured tables, must also be designed.
- There is a lot of up-front design work in doing so, but also some performance and query benefits in some cases.
- But, in other cases, there are also performance hits from the overhead of this design.

- Exploring Databases
- NoSQL, on the other hand, is the absence of that structure.
- We just put data into collections without paying much attention to the structure.
- As a result, there's less up-front design work, and possible performance increases in some cases.
- A good analogy from stack overflow is thinking of SQL like an automatic car, where a lot of the manual work is done for you, and some performance improvements, in other cases.
- And NoSQL is like a manual, or standard car.
- Something to keep in mind is that there are some databases that share the features of SQL and NoSQL.

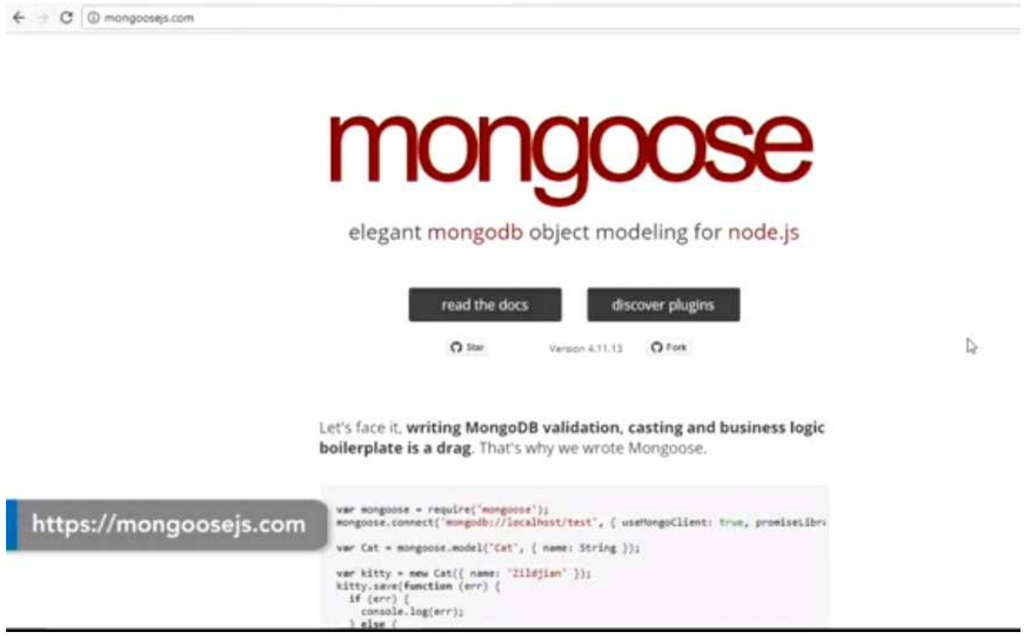


Trusted. Loved. Most widely deployed.

mLab's leading Database-as-a-Service powers **over half a million** MongoDB deployments on AWS, Azure, and Google

Thousands of companies trust us with their data

https://mlab.com



Let's face it, **writing MongoDB validation, casting and business logic boilerplate is a drag**. That's why we wrote Mongoose.

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/test', { useMongoClient: true, promiseLibr

var Cat = mongoose.model('Cat', { name: String });

var kitty = new Cat({ name: 'Zildjian' });
kitty.save(function (err) {
  if (err) {
    console.log(err);
  } else {
```

- Mongoose allows us to elegantly work with our MongoDB database with object schemas.
- Object schemas are JavaScript objects we create that will represent the type of data we will be putting in our database.
- Even though MongoDB is NoSQL, and does not have structure, with Mongoose, there is a bit of structure we must design initially.
- This gives us some benefits of easily interacting with our data through objects, and other things, such as validation.
- Let's begin by installing Mongoose with npm.

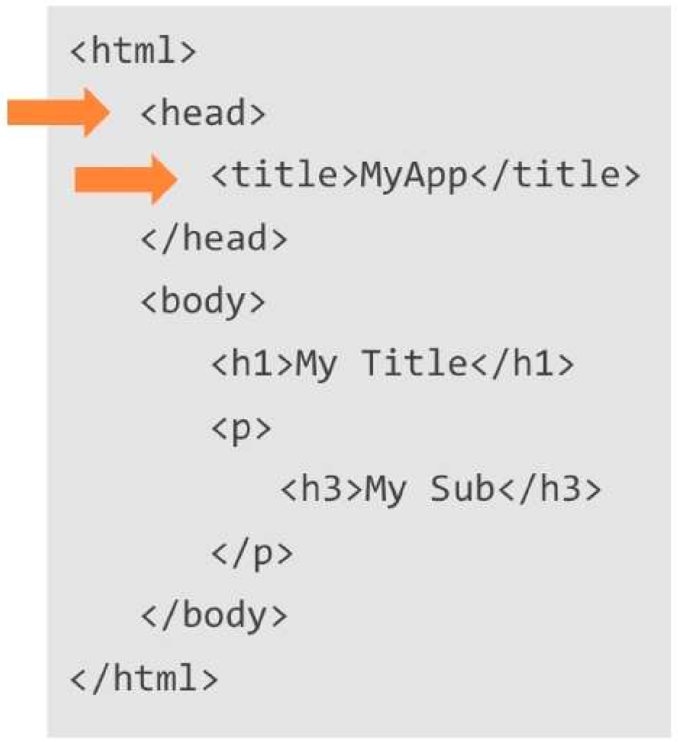
- **Improving Async Code With promises**
- Promises give us another option of how to work with asynchronous code.
- Promises return an object which promise to do some work.
- This object has separate callbacks for success and for failures.
- This lets us work with asynchronous code in a much more synchronous way.
- A really nice feature is that Promises can be combined into dependency chains.
-

- Automation and Deployment
- Web development tends to require a number of static operations that needs to perform to the code base before deployment.
- This requires to pre-process the Less or Sass files into CSS.
- Requires to improve network performance by bundling and compressing client JavaScript into a single file.
- It may even require to run your test to make sure that recent changes in code have not caused regression issues.
- All of the above can be managed by using a tool called **Grunt.**
- Generally Grunt is used in conjunction with npm scripts to automate static processes to the application’s code base.

- Grunt is a command line interface that can be used to run automated processes.
- The Grunt CLI is required to install globally so that we run Grunt anywhere.
- On any project that Grunt is to be used, install Grunt locally.
- In the "dev" dependencies, we save those packages that we need to help us build the application.

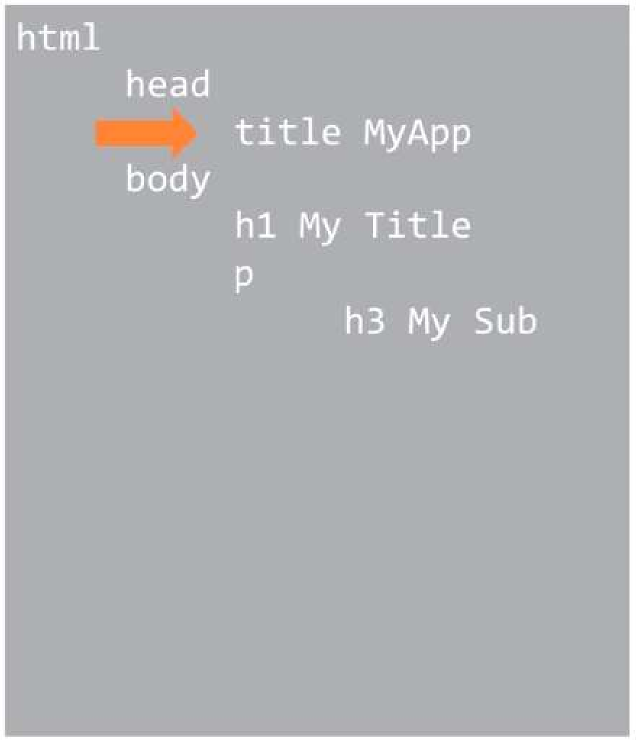
- VIEW ENGINES

- Jade Templating Engine



```
<html>
    <head>
        <title>MyApp</title>
    </head>
    <body>
        <h1>My Title</h1>
        <p>
            <h3>My Sub</h3>
        </p>
    </body>
</html>
```

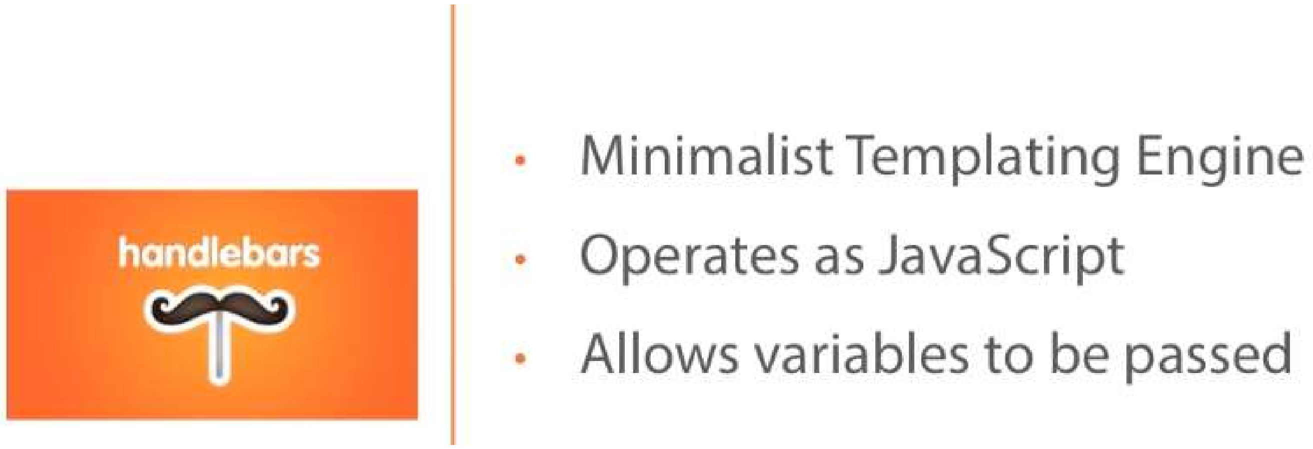
Normal HTML



```
html
    head
        title MyApp
    body
        h1 My Title
        p
            h3 My Sub
```

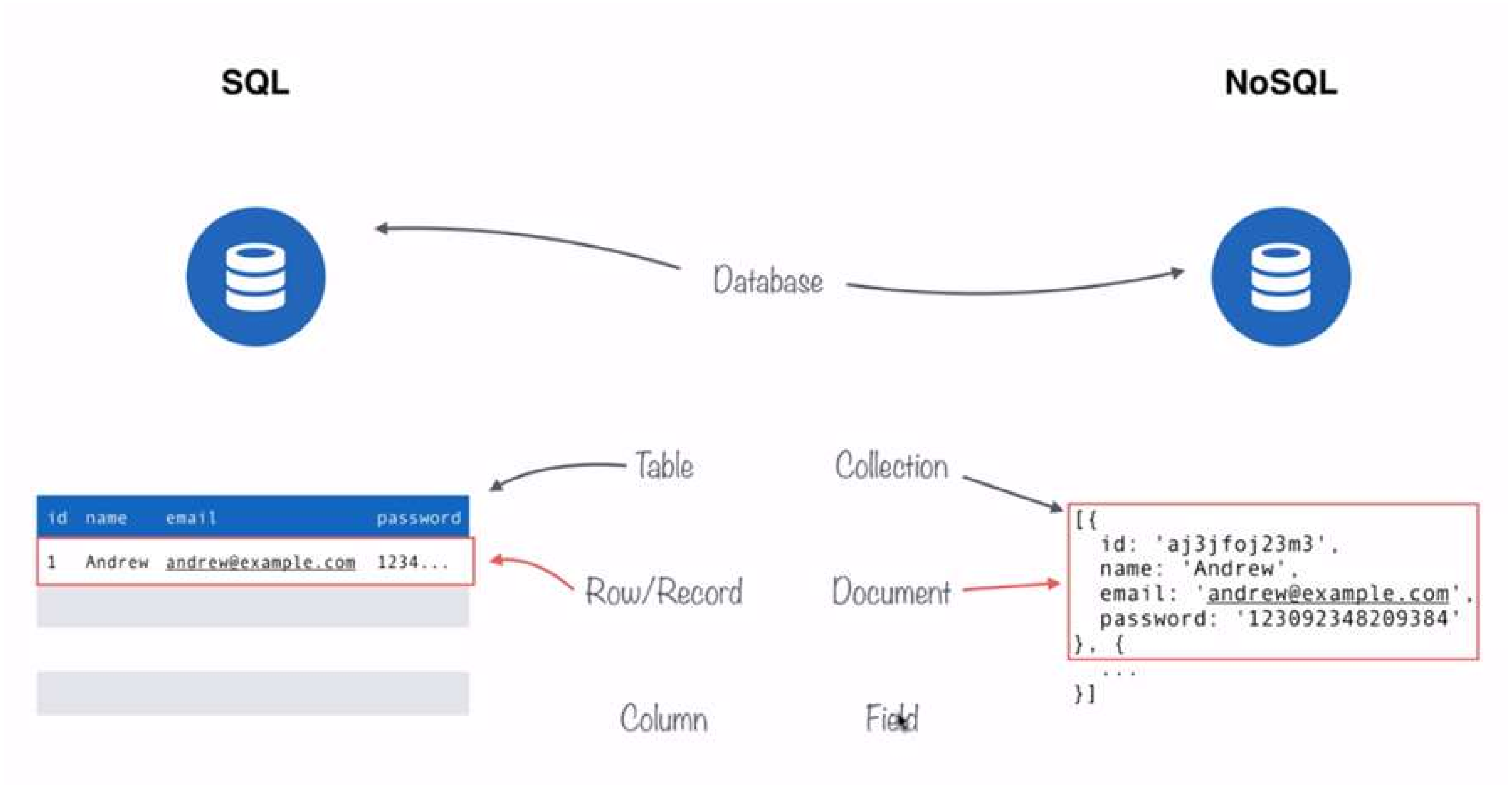
Equivalent JADE Template

- Handlebars



- Minimalist Templating Engine
- Operates as JavaScript
- Allows variables to be passed

- Comparing SQL and NoSQL



id	name	email	password
1	Andrew	andrew@example.com	1234...

```
[{
   id: 'aj3jfoj23m3',
   name: 'Andrew',
   email: 'andrew@example.com',
   password: '123092348209384'
}, {
   ...
}]
```