

# Node.js Intro

# I/O

- ▶ **I/O:** A communication between CPU and any other process external to the CPU (memory, disk, network).
- ▶ **I/O latency** is defined simply as the time that it takes to complete a single I/O operation.

System Event	Actual Latency	Scaled Latency
One CPU cycle	0.4 ns	1 s
Level 1 cache access	0.9 ns	2 s
Level 2 cache access	2.8 ns	7 s
Level 3 cache access	28 ns	1 min
Main memory access (DDR DIMM)	~100 ns	4 min
Intel® Optane™ DC persistent memory access	~350 ns	15 min
Intel® Optane™ DC SSD I/O	<10 µs	7 hrs
NVMe SSD I/O	~25 µs	17 hrs
SSD I/O	50–150 µs	1.5–4 days
Rotational disk I/O	1–10 ms	1–9 months
Internet call: San Francisco to New York City	65 ms[3]	5 years
Internet call: San Francisco to Hong Kong	141 ms[3]	11 years

# I/O needs to be done differently

---

- ▶ Consider two scenarios in real word:

- ▶ Movie Ticket

- ▶ You are in a queue to get a movie ticket. You cannot get one until everybody in front of you gets one, and the same applies to the people queued behind you.

**Synchronously**

- ▶ Order Food

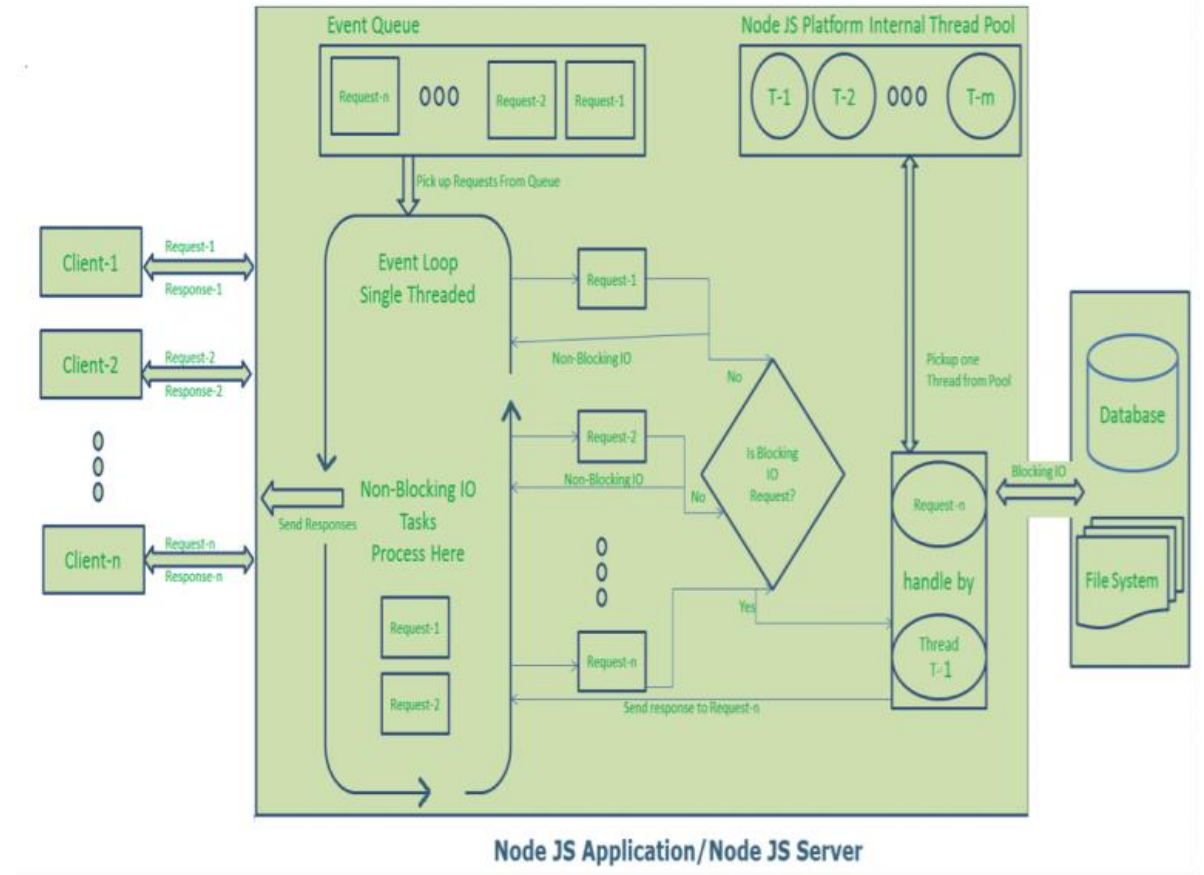
- ▶ You are in a restaurant with many other people. You order your food. Other people can also order their food, they don't have to wait for your food to be cooked and served to you before they can order. In the kitchen restaurant workers are continuously cooking, serving, and taking orders. People will get their food served as soon as it is cooked.

**Asynchronously**

# Node JS Architecture

## - Single Threaded Event Loop Advantages

- ▶ Handling more and more concurrent client's request is very easy.
- ▶ Even though our Node JS Application receives more and more Concurrent client requests, there is no need of creating more and more threads, because of Event loop.
- ▶ Node JS application uses less Threads so that it can utilize only less resources or memory



# Node.js

---

- ▶ JavaScript runtime built on Chrome V8 JavaScript Engine
- ▶ Server-side JavaScript
- ▶ Allows script programs do I/O in JavaScript
- ▶ Event-driven, non-blocking I/O
- ▶ Single Threaded
- ▶ CommonJS module system

# Setting up Node.js

---

- ▶ Go to [nodejs.org](https://nodejs.org) or [download page](#) to download node. After installing Node we will be able to use it using the command line interface.
- ▶ If Node is installed properly, Try this command: **node -v**  

```
H:\courses\MSD\cs477\cs477>node -v  
v16.3.0
```
- ▶ Hit **Ctrl+C** twice or **Ctrl+D** once to quit Node.
- ▶ Github Node.js: <https://github.com/nodejs/nodejs.org>

# Node Versions

---

- ▶ **Current:** Under active development. Code for the Current release is in the branch for its major version number (for example, [v10.x](#)). Node.js releases a new major version every 6 months, allowing for breaking changes. This happens in April and October every year. Releases appearing each October have a support life of 8 months. Releases appearing each April convert to LTS (see below) each October.
- ▶ **LTS:** Releases that receive Long-term Support, with a focus on stability and security. Every even-numbered major version will become an LTS release. LTS releases receive 18 months of *Active LTS* support and a further 12 months of *Maintenance*. LTS release lines have alphabetically-ordered codenames, beginning with v4 Argon. There are no breaking changes or feature additions, except in some special circumstances.

# First Program

---

```
setTimeout(function () { console.log("world"); }, 2000); console.log("hello");
```


hello\_world.js

```
% node hello_world.js
```

```
Hello
```

```
// 2 seconds later...
```

```
World
```

 node.js file name is reserved in Node

Node exits automatically when there is nothing else to do (end of process). Let's change it to never exit, but to keep it in loop!

Node API is not all asynchronous. Some parts of it are synchronous like, for instance, some file operations. Don't worry, they are very well marked: they always end with "Sync". They should only be used when initializing.



# The Server Global Environment

---

- ▶ In Node we run JS on the server so we don't have `window` object. Instead Node provides us with global modules and methods that are automatically created for us (*they aren't part of ECMA specifications*)
  - ▶ `module`
  - ▶ `global` (*The global namespace object*)
  - ▶ `process`
  - ▶ `buffer`
  - ▶ `require`
  - ▶ `setInterval(callback, delay)` and `clearInterval()`
  - ▶ `setTimeout(callback, delay)` and `clearTimeout()`
- ▶ Doc: <https://nodejs.org/api/globals.html>

# Global Scope in Node

---

- ▶ Browser JavaScript by default puts everything into its `window` global scope.
- ▶ Node.js was designed to behave differently with **everything being local by default**. In case we need to set something globally, there is a `global` object that can be accessed by all modules. (not recommended)
- ▶ The document object that represent DOM of the web page is nonexistent in Node.js.

# What's inside Node?

## ▶ V8

- ▶ Google's open source JavaScript engine.
- ▶ Translates your JS code into machine code
- ▶ V8 is written in C++.
- ▶ Read more about how V8 works [here](#).

## ▶ libuv

- ▶ a multi-platform support library with a focus on asynchronous I/O.
- ▶ Asynchronous file and file system operations
- ▶ Thread pool
- ▶ ...

## ▶ **Binding**

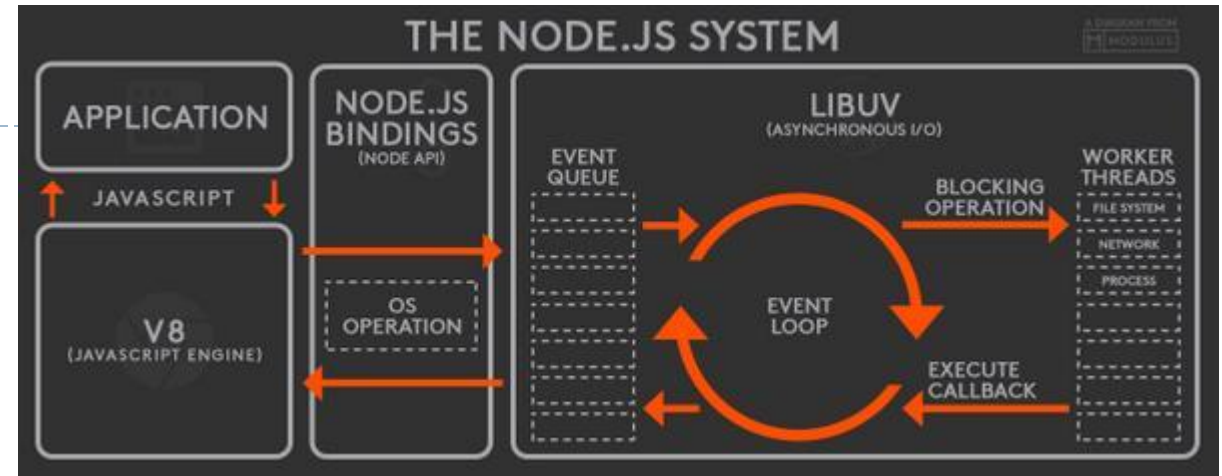
- ▶ A wrapper around a library written in one language and expose the library to codes written in another language so that codes written in different languages can communicate.

## ▶ **Other Low-Level Components**

- ▶ such as [c-ares](#), [http\\_parser](#), [OpenSSL](#), [zlib](#), and etc, mostly written in C/C++.

## ▶ **Application**

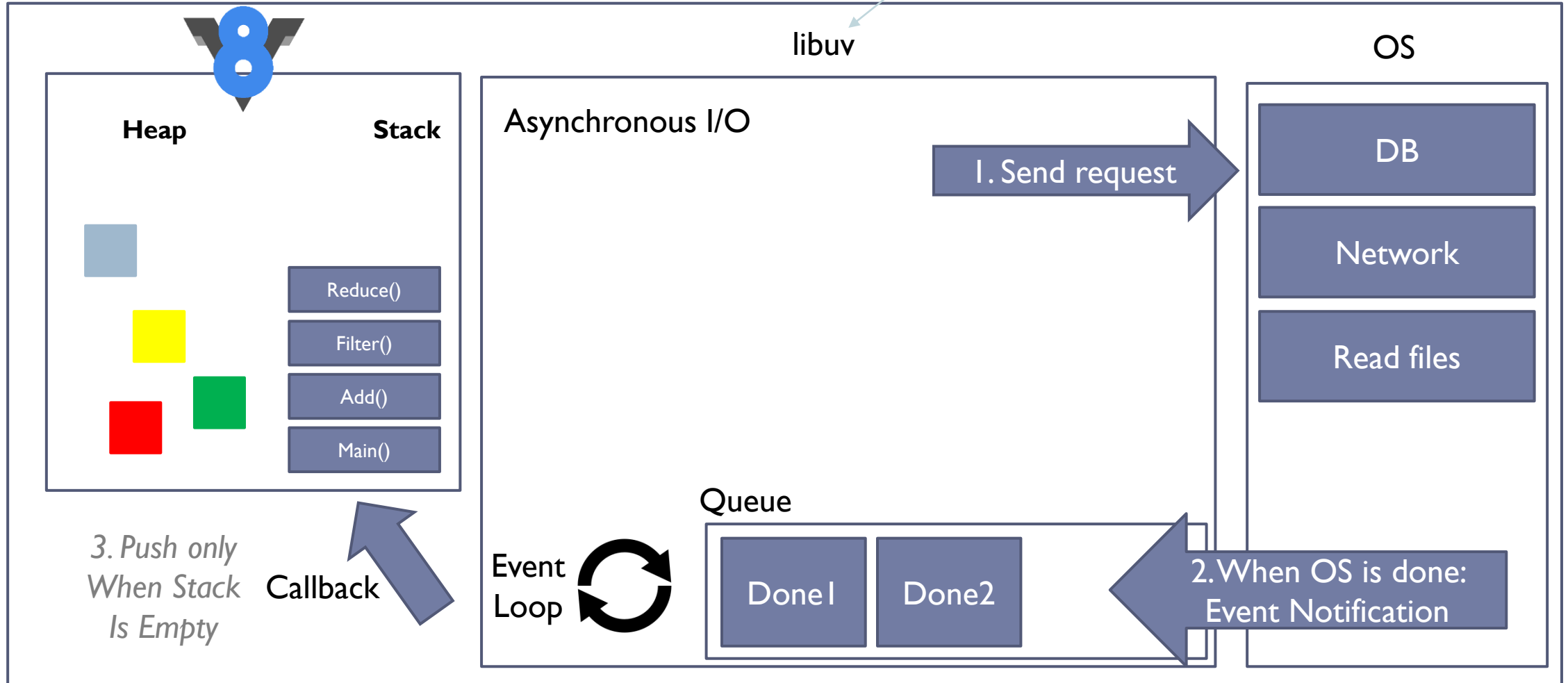
- ▶ here is your code, modules, and Node.js' [built in modules](#), written in JS



# JS on the Server

An abstract non-blocking IO operations (Async) using thread pool

Part of NodeJs



# What's the event loop?

---

The event loop is what allows Node.js to perform non-blocking I/O operations — despite the fact that JavaScript is single-threaded — by offloading operations to the system kernel whenever possible.

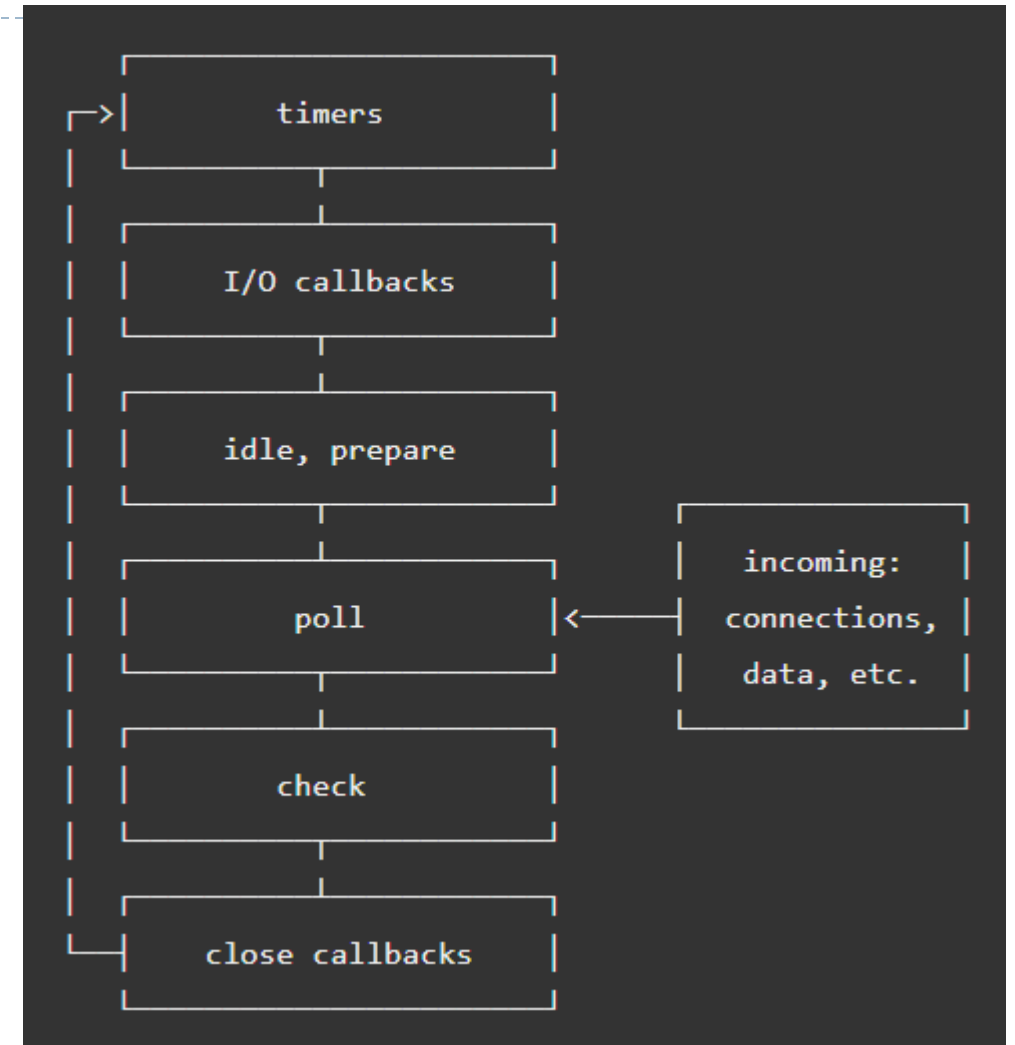
— from node.js doc

A loop that picks events from the event queue and pushes their callbacks into the call stack.

Node.js runs using a **single thread**, at least from a Node.js developer's point of view. Under the hood Node uses many threads through **libuv**.

# Event Loop

- ▶ **timers**: this phase executes callbacks scheduled by `setTimeout()` and `setInterval()`.
- ▶ **pending callbacks**: executes I/O callbacks deferred to the next loop iteration.
- ▶ **idle, prepare**: only used internally.
- ▶ **poll**: retrieve new I/O events; execute I/O related callbacks (almost all with the exception of close callbacks, the ones scheduled by timers, and `setImmediate()`); node will block here when appropriate.
- ▶ **check**: `setImmediate()` callbacks are invoked here.
- ▶ **close** callbacks: some close callbacks, e.g. `socket.on('close', ...)`.



<https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/>

# setTimeout vs setImmediate

---

## ▶ **setTimeout**

- ▶ schedules a callback to run after a specific time, the functions are registered in the **timers phase** of the event loop.

## ▶ **setImmediate**

- ▶ schedules a callback to run at **check phase** of the event loop after IO events' callbacks.

## `process.nextTick(callback)`

---

`process.nextTick()` is not part of the event loop, it adds the callback into the `nextTick` queue. Node processes **all the callbacks** in the `nextTick` queue after the current operation completes and before the event loop continues.

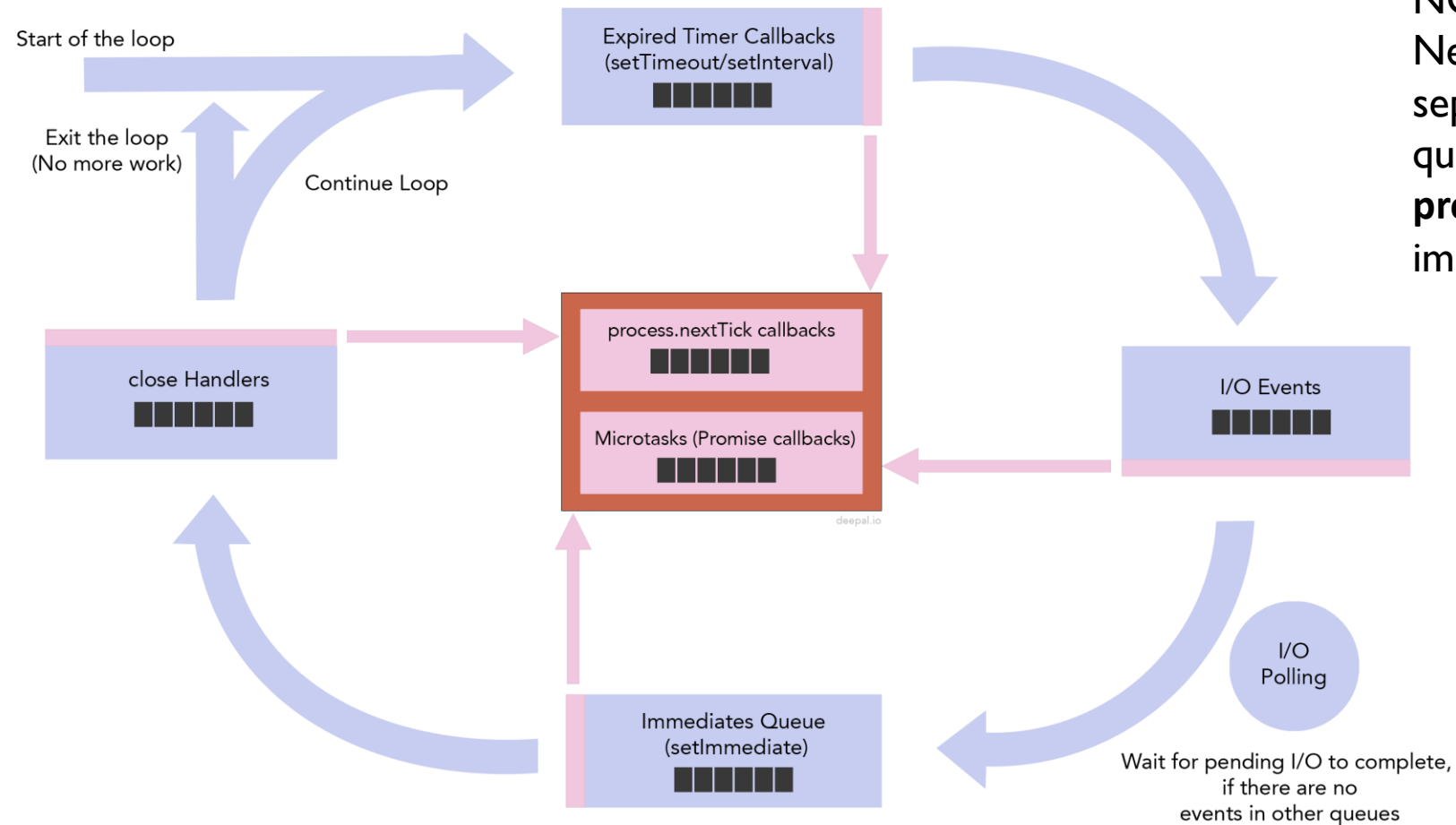
Which means it runs **before** any additional I/O events or timers fire in subsequent ticks of the event loop.

**Note:** the next-tick-queue is completely drained on each pass of the event loop before additional I/O is processed. As a result, recursively setting `nextTick` callbacks will block any I/O from happening, just like a `while(true)` loop.



# process.nextTick(callback)

\* nextTicks and Promise callback queues are processed between each timer and immediate callback in node v11 and above



## NOTE:

Next tick queue is displayed separately from the other four main queues because it is **not natively provided by the libuv**, but implemented in Node.

# process.nextTick(callback)

---

```
function foo() {  
    console.log('foo');  
}  
process.nextTick(foo);  
console.log('bar');
```

Notice that bar will be printed in the console before foo, as we have delayed the invocation of foo() till the next tick of the event loop. We can get the same result by using setTimeout() this way:

```
setTimeout(foo, 0);  
console.log('bar');
```

However, process.nextTick() is not just a simple alias to setTimeout(fn, 0).

What's the difference and why it's more efficient?

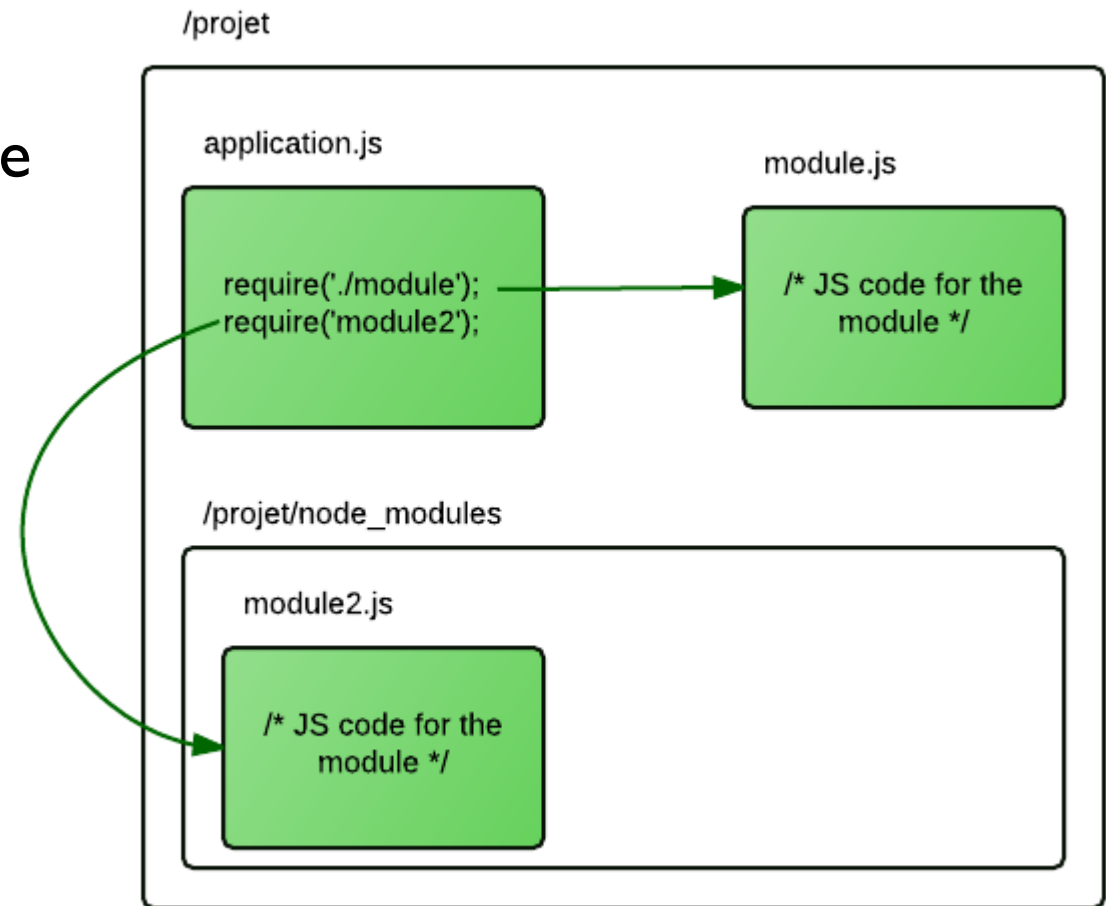
# setTimeout vs setImmediate vs process.nextTick

---

- ▶ `setTimeout(() => { console.log('timeout'); }, 0);`
  - ▶ `setImmediate(() => { console.log('immediate'); });`
  - ▶ `process.nextTick(() => console.log('nexttick'));`
- 
- ▶ What's the output of this code and why?

# Modules in NodeJS

- ▶ Consider modules to be the same as JavaScript libraries.
- ▶ A set of functions you want to include in your application.
- ▶ Node implements **CommonJS** Modules specs.
  - ▶ CommonJS module are defined in normal `.js` files using `module.exports`
  - ▶ In Node.js, each file is treated as a separated module



<https://openclassrooms.com>

# Type of Modules

---

## ▶ Built-in Modules

- ▶ Node.js has a set of built-in modules which you can use without any further installation.
- ▶ [buffer](#), [fs](#), [http](#), [path](#), etc.
- ▶ [Built-in Modules Reference](#)

## ▶ 3<sup>rd</sup> party modules on [www.npmjs.com](http://www.npmjs.com)

## ▶ Your own Modules

- ▶ Simply create a normal JS file it will be a module (*without affecting the rest of other JS files and without messing with the global scope*).

# Include Modules – `require()` function

---

- ▶ The basic functionality of `require` is that it reads a JavaScript file, executes the file, and then proceeds to return the `module.exports` object.
  - ▶ `const path = require('path');`
  - ▶ `const config = require('./config');`
- ▶ Rules:
  - ▶ if the file doesn't start with `"/"` or `"/"`, then it is either considered a **core module** (and the local Node path is checked), or a dependency in the local **node\_modules** folder.
  - ▶ If the file starts with `"/"` it is considered a relative file to the file that called `require`.
  - ▶ If the file starts with `"/"`, it is considered an absolute path.
  - ▶ If the filename passed to `require` is a directory, it will first look for `package.json` in the directory and load the file referenced in the `main` property. Otherwise, it will look for an `index.js`.
  - ▶ **NOTE:** you can omit `.js` and `require` will automatically append it if needed.

# How `require ( ' /path/to/file ' )` works

---

- ▶ Node goes through the following sequence of steps:
  1. Resolve: to find the absolute path of the file
  2. Load: to determine the type of the file content
  3. Wrap: to give the file its private scope
  4. Evaluate: This is what the VM does with the loaded code
  5. Cache: when we require this file again, don't go over all the steps.
  
- ▶ **Note:** Node core modules return immediately (no resolve)

# What's the wrapper?

---

▶ `node -p "require('module').wrapper"`

1. Node will wrap your code into:

```
(function (exports, require, module, __filename, __dirname){  
    exports = module.exports;  
    // this is why can use exports and module objects.. etc in your code  
    // without any problem, because Node is going to initialize these and pass  
    // them as parameters through this wrapper function  
});
```

2. Node will run the function using `.apply()`

3. Node will return the following:

```
return module.exports;
```



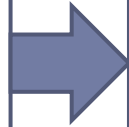
## module.exports

---

- Think about this object (`module.exports`) as return statement.

```
// helloModule.js
let sayHi = function(){
    console.log('hi');
}

module.exports = sayHi;
```

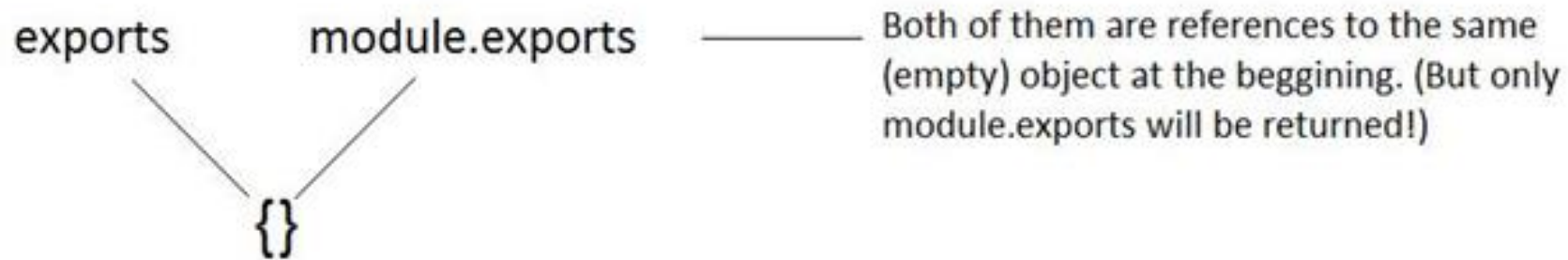


```
// app.js
let hello = require('./helloModule');

hello();
```

## exports vs module.exports

- ▶ **exports object** is a reference to the `module.exports`, that is shorter to type



- ▶ Be careful when using `exports`, a code like this will make it point to another object. At the end, `module.exports` will be returned.



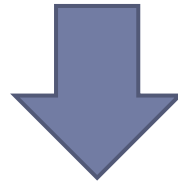
```
exports = function doSomething() {  
  console.log('blah blah');  
}
```

`doSomething()` isn't exported.

# Using Modules – Pattern 1

---

```
// Pattern1 - pattern1.js
module.exports = function () {
  console.log('Josh Edward');
};
```



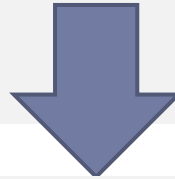
```
// app.js
const getName = require('./pattern1');
getName(); // Josh Edward
```

## Using Modules – Pattern 2

---

```
// Pattern2 - pattern2.js
module.exports.getName = function () {
  console.log('Josh Edward');
};

// OR
exports.getName = function () {
  console.log('Josh Edward');
};
```



```
// app.js
const getName = require('./pattern2').getName;
getName(); // Josh Edward

// OR
const person = require('./pattern2');
person.getName(); // Josh Edward
```

# Using Modules – Pattern 3

```
// Pattern3 - pattern3.js
class Person {
  constructor(name) {
    this.name = name;
  }

  getName() {
    console.log(this.name);
  }
}
module.exports = new Person('Josh Edward');
```

Warning: Not good practice

```
// Pattern3 - cached.js
const personObj2 = require('./pattern3');
// cached
console.log('---inside cached.js ---');
personObj2.getName(); //Emma Smith
```

```
// app.js
const personObj = require('./pattern3');
personObj.getName(); // Josh Edward
personObj.name = 'Emma Smith';
personObj.getName(); //Emma Smith
const cachedObj = require('./cached.js'); // cached in the same module
```

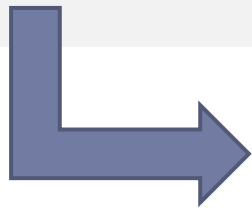
# Using Modules – Pattern 4

---

```
// Pattern - pattern4.js
class Person {
  constructor(name = 'Josh Edward') {
    this.name = name;
  }

  getName() {
    console.log(this.name);
  }
}

module.exports = Person;
```



```
// app.js
const Person = require('./pattern4');
const personObj1 = new Person();
personObj1.getName() // Josh Edward
personObj1.name = 'Emma Smith';
personObj1.getName(); //Emma Smith
```

```
const Person2 = require('./pattern4');
const personObj2 = new Person2();
personObj2.getName(); // Josh Edward
```

# Using Modules – Pattern 5

---

```
// Pattern5 - pattern5.js
const name = 'Josh Edward';
function getName() {
  console.log(name);
}
module.exports = {
  getName: getName // closure
}
```



```
// app.js
const getName = require('./pattern5').getName;
getName(); // Josh Edward
```

# Node core libraries

---

- ▶ Node provides many core libraries

```
const util = require('util'); // We do not use ./ before the file  
name
```

```
const sayHi = util.format("Hi, %s", 'Josh');  
console.log(sayHi); //Hi, Josh
```

- ▶ Read [API documentation](#)



## path module

---

- ▶ The `path` module provides a lot of very useful functionality to access and interact with the file system.

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

```
//Return the directory part of a path:
```

```
console.log(path.dirname('Buffer'));
```

```
console.log(path.dirname('File/example1.js')); // /test/something
```

```
//Joins two or more parts of a path:
```

```
const name = 'joe';
```

```
console.log(path.join('users', name, 'notes.txt'));
```

## fs module

- ▶ The `fs` module provides a lot of very useful functionality to access and interact with the file system.

```
const fs = require('fs');
const path = require('path');
console.log(__dirname); // returns absolute path of current file
const greet = fs.readFileSync(path.join(__dirname, 'greet.txt'), 'utf8');
console.log(greet);
```

```
fs.readFile(path.join(__dirname, 'greet.txt'), 'utf8',
    function(err, data) { console.log(data); });
console.log('Done!');
// Hello
// Done!
// Hello
```

- ▶ Notice the Node Applications Design: any async function accepts a **callback as a last parameter** and the **callback function accepts error as a first parameter** (`null` will be passed if there is no error).

▶ Notice: `data` here is a buffer object. We can convert it with `toString` or add the encoding – `'utf8'`

# Example Read/Write Files

---

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

// Reading from a file:
fs.readFile(path.join(__dirname, 'greet.txt'), { encoding: 'utf8' }, (err, data) => {
  if (err) throw err;
  console.log(data);
});

// Writing to a file:
fs.writeFile('students.txt', 'Hello World!', (err) => {
  if (err) throw err;
  console.log('Done');
});
```

What's the problem with the code above?

# Stream

---

- ▶ Stream is a way to handle reading/writing files, network communications, or any kind of end-to-end information exchange in an efficient way.
- ▶ Why streams?
  - ▶ Memory efficiency: you don't need to load large amounts of data in memory before you are able to process it
  - ▶ Time efficiency: it takes way less time to start processing data, since you can start processing as soon as you have it, rather than waiting till the whole data payload is available
- ▶ The Node.js [stream](#) module provides the foundation upon which all streaming APIs are built. All streams are instances of [EventEmitter](#)

# Different types of streams

---

- ▶ Readable: a stream you can pipe from, but not pipe into (you can receive data, but not send data to it). When you push data into a readable stream, it is buffered, until a consumer starts to read the data. (`fs.createReadStream`)
- ▶ Writable: a stream you can pipe into, but not pipe from (you can send data, but not receive from it). (`fs.createWriteStream`)
- ▶ Duplex: a stream you can both pipe into and pipe from, basically a combination of a Readable and Writable stream. (`net.Socket`)
- ▶ Transform: a Transform stream is similar to a Duplex, but the output is a transform of its input. (`zlib.createGzip`)

# Examples of Readable and Writable streams

---

## Readable Streams

- ▶ HTTP responses, on the client
- ▶ HTTP requests, on the server
- ▶ fs read streams
- ▶ zlib streams
- ▶ crypto streams
- ▶ TCP sockets
- ▶ child process stdout and stderr
- ▶ process.stdin

## Writable Streams

- ▶ HTTP requests, on the client
- ▶ HTTP responses, on the server
- ▶ fs write streams
- ▶ zlib streams
- ▶ crypto streams
- ▶ TCP sockets
- ▶ child process stdin
- ▶ process.stdout, process.stderr

# Stream example

---

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

// Stream will read the file in chunks
// if file size is bigger than the chunk then it will read a chunk and emit a 'data' event.
// Use encoding to convert data to String of hex
// Use highWaterMark to set the size of the chunk. Default is 64 kb

const readable = fs.createReadStream(path.join(__dirname, 'card.jpg'),
  { highWaterMark: 16 * 1024 });

const writable = fs.createWriteStream(path.join(__dirname, 'destinationFile.jpg'));

readable.on('data', function(chunk) {
  console.log(chunk.length);
  writable.write(chunk);
});
```

## Pipes: `src.pipe(dst);`

---

- ▶ To connect two streams, Node provides a method called `pipe()` available on all readable streams. Pipes hide the complexity of listening to the stream events.

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

const readable = fs.createReadStream(path.join(__dirname, 'card.jpg'));
const writable = fs.createWriteStream(path.join(__dirname, 'destinationFile.jpg'));

readable.pipe(writable);

// note that pipe return the destination, this is why you can pipe it again to another
// stream if the destination was readable in this case it has to be DUPLEX because you
// are going to write to it first, then read it and pipe it again to another writable
// stream.
```