

Node.js Intro & Module

Rujuan Xing

Front end and Back end

- ▶ **Front end / Client-side**

- ▶ HTML, CSS and Javascript
- ▶ Asynchronous request handling and AJAX

- ▶ **Back end / Server-side**

- ▶ Node.js, PHP, Python, Ruby, Perl
- ▶ Compiled languages like C#, Java or Go
- ▶ Various technologies and approaches

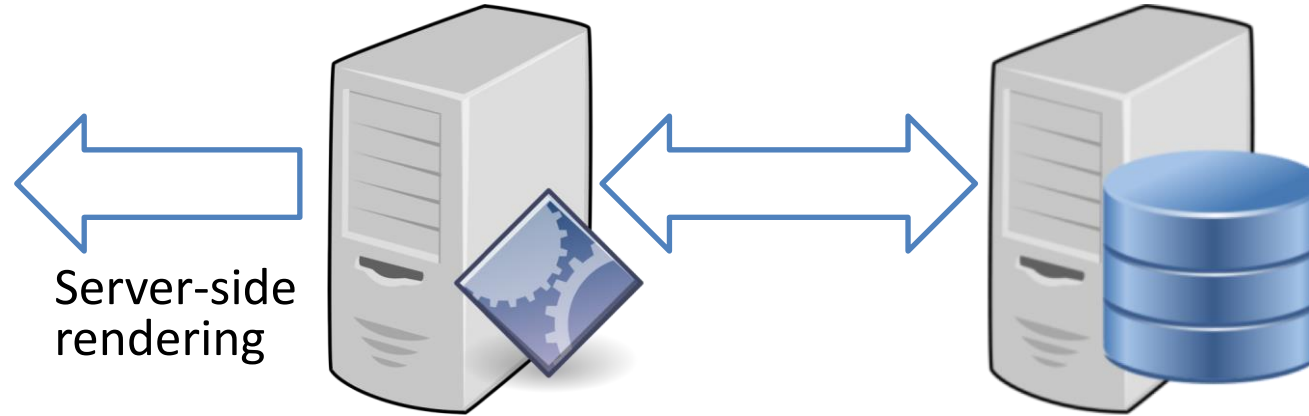
- ▶ https://en.wikipedia.org/wiki/Front_and_back_ends

Traditional Web Development

HTML, CSS, JS

Ruby, Python, Java, C++, PHP

DBMS



Presentation layer

Business Logic layer

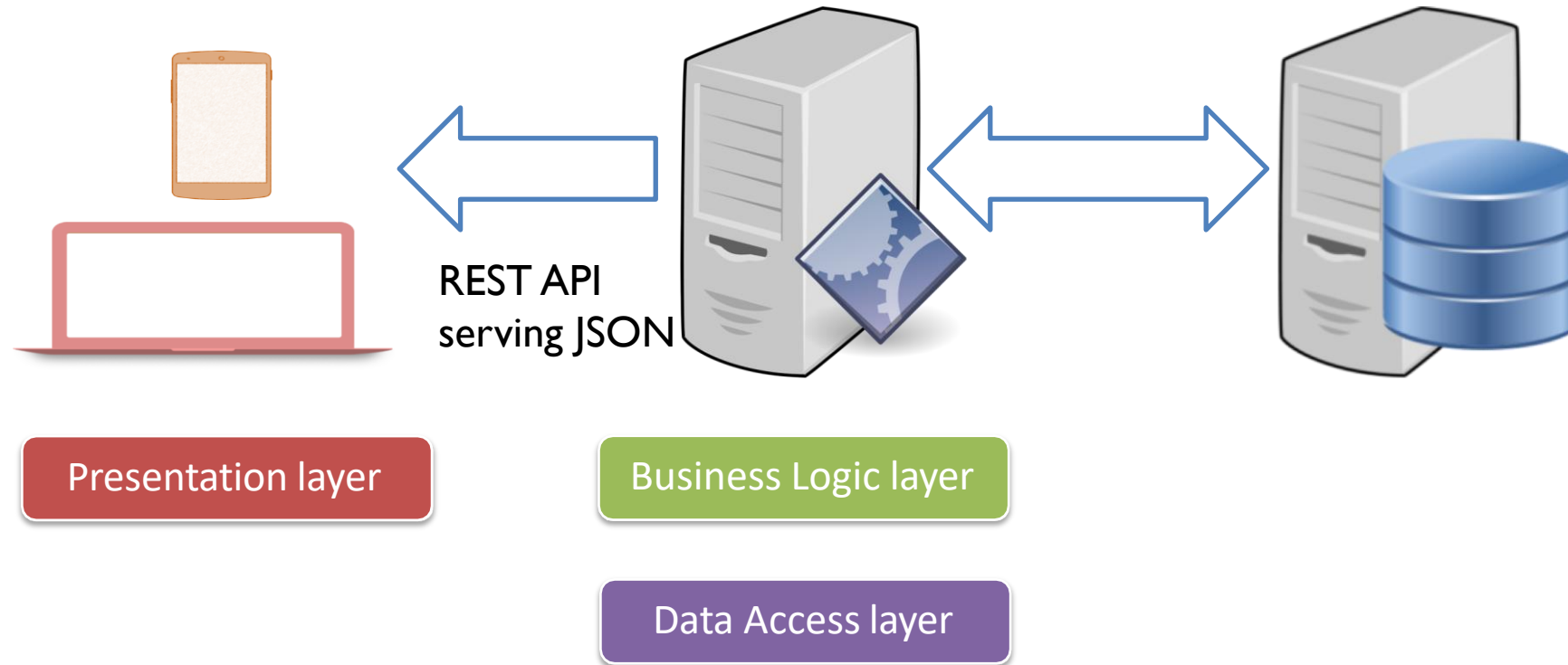
Data Access layer

Full Stack JavaScript Development

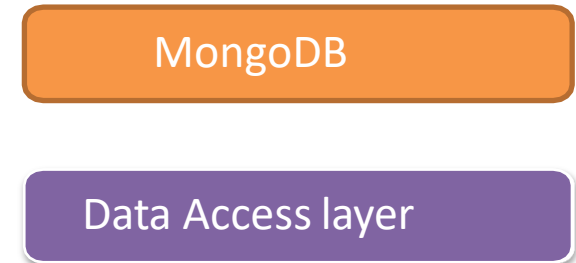
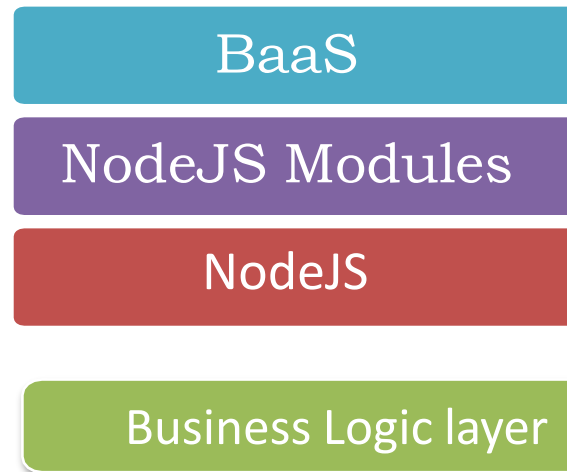
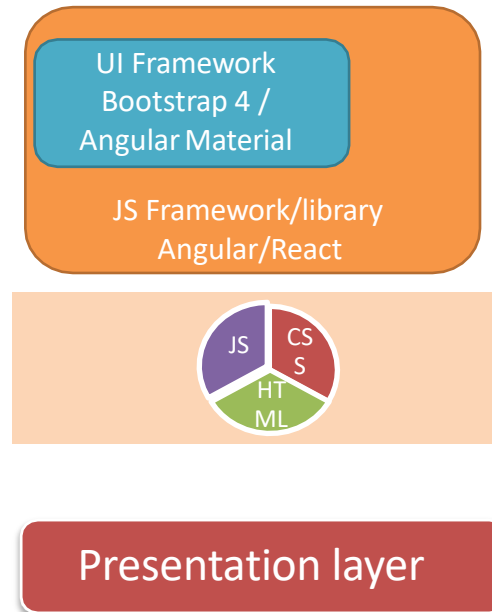
Single page Apps
using JavaScript frameworks/libraries
like Angular or React

NodeJS and
NodeJS modules

MongoDB
JSON documents



Full Stack Web Development



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

Blocking vs non-blocking?

```
const add = (a,b)=>{  
  for(let i=0; i<9e27; i++){  
    console.log(a+b);  
  }  
}
```

```
console.log('start');  
const A = add(1,2);  
const B = add(2,3);  
const C = add(3,4);  
console.log('end');
```

Blocking methods execute **synchronously**

```
const add = (a,b)=>{  
  setTimeout(()=>{  
    for(let i=0; i<9e27; i++){  
      console.log(a+b);  
    }, 5000);  
}  
console.log('start');  
const A = add(1,2);  
const B = add(2,3);  
const C = add(3,4);  
console.log('end');
```

non-blocking methods execute **asynchronously**

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
- ▶ Focused on Performance

Setting up Node.js

- ▶ Go to 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
```

- ▶ 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 webpage 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**

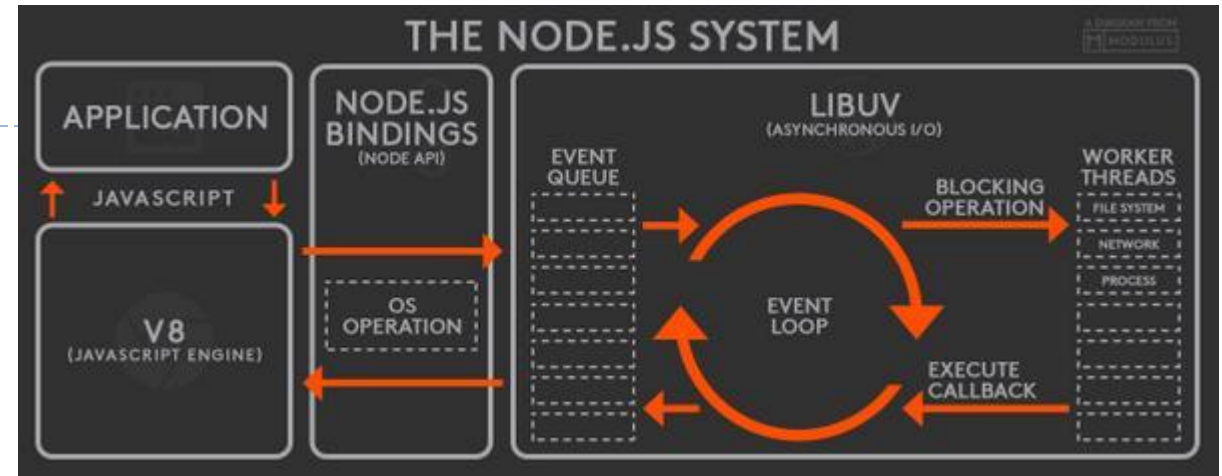
- ▶ A wrapper around a library written in one language and expose the library to codes written in another language so that codes written in different languages can communicate.

▶ **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



Asynchronous code execution

- ▶ libuv helps handle asynchronous operations in Node.js.
 - ▶ For async operations like handling a network request, libuv relies on the operating system primitives.
 - ▶ For async operations like reading a file that has no native OS support, libuv relies on its thread pool to ensure that the main thread is not blocked.

```
const fs = require('fs');  
  
console.log('first');  
fs.readFile('hello.txt', () => console.log('second'));  
console.log('third');
```


What's the event loop?

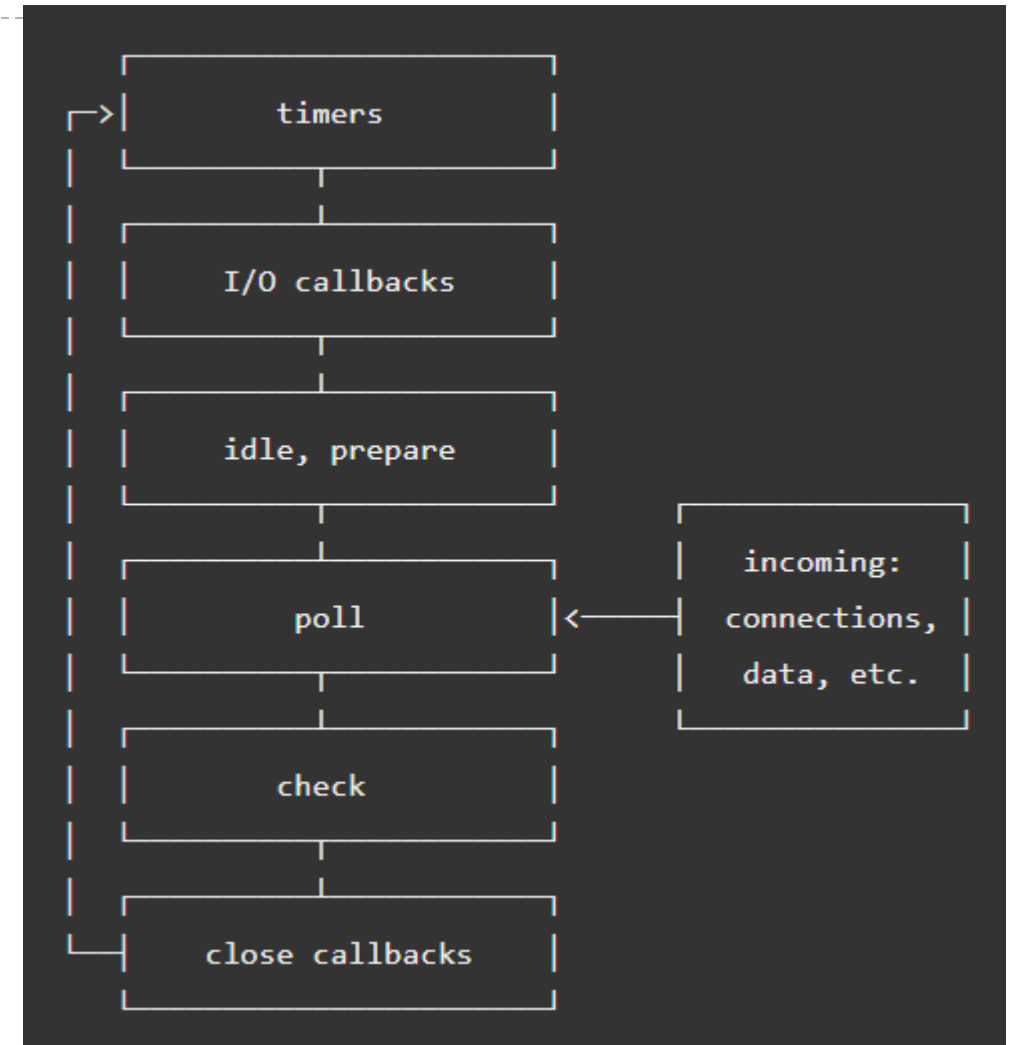
The event loop is what allows Node.js to perform non-blocking I/O operations — despite the fact that JavaScript is single-threaded — by offloading operations to the system kernel whenever possible.

— from node.js doc

Technically, the event loop is just a C program. But, you can think of it as a design pattern that orchestrates or coordinates the execution of synchronous and asynchronous code in Node.js.

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/>

`process.nextTick(callback)`

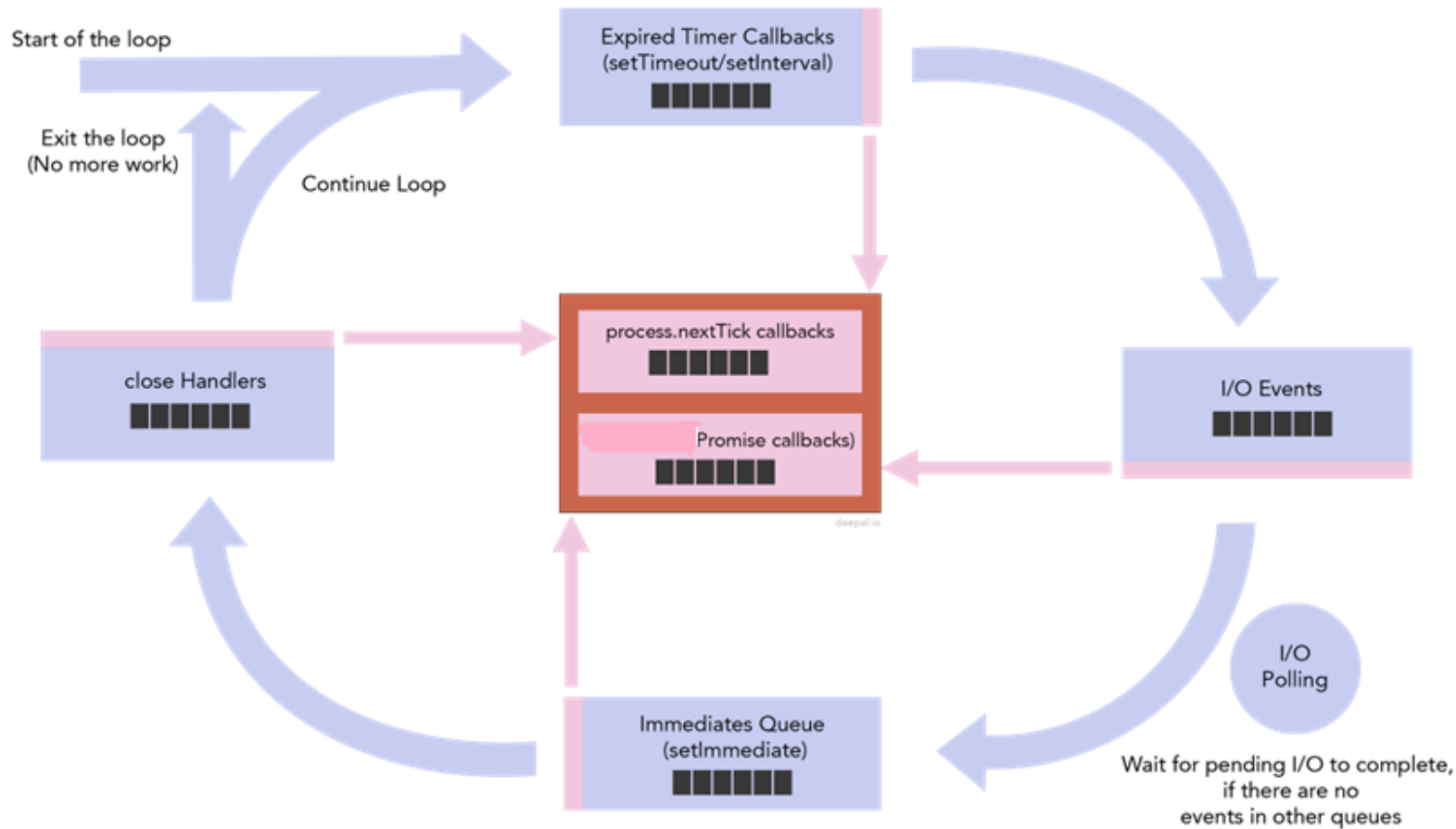
`process.nextTick()` is not part of the event loop, it adds the callback into the `nextTick` queue. Node processes **all the callbacks** in the `nextTick` queue after the current operation completes and before the event loop continues.

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.

nextTick and Promise Queues in Node.js Event Loop

```
new Promise(resolve => resolve('Hi'))  
  .then(() => console.log("this is Promise.resolve 1"));  
process.nextTick(() => console.log("this is process.nextTick 1"));
```

- ▶ When the call stack executes line 1, it queues the callback function in the `Promise` queue.
- ▶ When the call stack executes line 2, it queues the callback function in the `nextTick` queue.
- ▶ There is no more user-written code to execute after line 2.
- ▶ Control enters the event loop, where the `nextTick` queue gets priority over the `promise` queue (it's how the Node.js runtime works).
- ▶ The event loop executes the `nextTick` queue callback function and then the `promise` queue callback function.
- ▶ The console shows "this is process.nextTick 1", and then "this is Promise.resolve 1".

All callbacks in the `nextTick` queue are executed before callbacks in the `promise` queue.

The Timer Queue in Node.js Event Loop

```
setTimeout(() => console.log("this is setTimeout 1"), 0);
setTimeout(() => {
  console.log("this is setTimeout 2");
  process.nextTick(() =>
    console.log("this is inner nextTick inside setTimeout")
  );
}, 0);
setTimeout(() => console.log("this is setTimeout 3"), 0);

new Promise(resolve => resolve('Hi')).then(() => console.log("this is Promise.resolve 1"));
new Promise(resolve => resolve('Hi')).then(() => console.log("this is Promise.resolve 2"));

process.nextTick(() => console.log("this is process.nextTick 1"));
process.nextTick(() => console.log("this is process.nextTick 2"));
```

Callbacks in the Microtask Queues are executed before callbacks in the Timer Queue.

Callbacks in microtask queues are executed in between the execution of callbacks in the timer queue.

I/O Queue in the Node.js Event Loop

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

fs.readFile('hello.txt', () => console.log('this is readFile 1'));
new Promise(resolve => resolve('Hi')).then(() => console.log("this is Promise.resolve 1"));
process.nextTick(() => console.log("this is process.nextTick 1"));
```

Callbacks in the microtask queue are executed before callbacks in the I/O queue.

I/O Queue in the Node.js Event Loop Cont.

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

fs.readFile('hello.txt', () => console.log('this is readFile 1'));
setTimeout(() => console.log("this is setTimeout 1"), 0);
```

The anomaly is due to how a minimum delay is set for timers. In [the C++ code for the DOMTimer](#), we come across a very interesting piece of code. The interval in milliseconds is calculated, but the calculation is capped at a maximum of 1 millisecond or the user-passed interval multiplied by 1 millisecond.

This means that if we pass in 0 milliseconds, the interval is set to $\max(1, 0)$, which is 1. This will result in `setTimeout` with a 1 millisecond delay. It seems that Node.js follows a similar implementation. When you set a 0 millisecond delay, it is overwritten to a 1 millisecond delay.

When running `setTimeout()` with a delay of 0ms and an I/O async method, the order of execution can never be guaranteed.

I/O Polling in the Node.js Event Loop

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

fs.readFile('hello.txt', () => console.log('this is readFile 1'));
new Promise(resolve => resolve('Hi')).then(() => console.log("this is Promise.resolve 1"));
process.nextTick(() => console.log("this is process.nextTick 1"));
setTimeout(() => console.log("this is setTimeout 1"), 0);
setImmediate(() => console.log("this is setImmediate 1"));

for (let i = 0; i < 2000000000; i++) {}
```

the `readFile()` callback is not queued up at the same time as other callbacks. The event loop has to poll to check if I/O operations are complete, and it only queues up completed operation callbacks. This means that when the control enters the I/O queue for the first time, the queue is still empty.

The control then proceeds to the polling part of the event loop, where it checks with `readFile()` if the task has been completed. `readFile()` confirms that it has, and the event loop now adds the associated callback function to the I/O queue. However, the execution has already moved past the I/O queue, and the callback has to wait for its turn to be executed.

The control then proceeds to the check queue, where it finds one callback. It logs `"setImmediate 1"` to the console and then starts a new iteration because there is nothing else left to process in the current iteration of the event loop.

I/O events are polled and callback functions are added to the I/O queue only after the I/O is complete

Check Queue in the Node.js Event Loop

```
const fs = require("fs");

fs.readFile('hello.txt', () => {
  console.log("this is readFile 1");
  setImmediate(() => console.log("this is setImmediate 1"));
  process.nextTick(() =>
    console.log("this is inner process.nextTick inside readFile")
  );
  Promise.resolve().then(() =>
    console.log("this is inner Promise.resolve inside readFile")
  );
});

process.nextTick(() => console.log("this is process.nextTick 1"));
Promise.resolve().then(() => console.log("this is Promise.resolve 1"));
setTimeout(() => console.log("this is setTimeout 1"), 0);

for (let i = 0; i < 2000000000; i++) {}
```

Check queue callbacks are executed after microtask queues callbacks, timer queue callbacks and I/O queue callbacks are executed.

Microtask queues callbacks are executed after I/O queue callbacks and before check queue callbacks.

Close Queue in the Node.js Event Loop

```
const fs = require("fs");

const readableStream = fs.createReadStream('hello.txt');
readableStream.close();

readableStream.on("close", () => {
  console.log("this is from readableStream close event callback");
});
setImmediate(() => console.log("this is setImmediate 1"));
setTimeout(() => console.log("this is setTimeout 1"), 0);
Promise.resolve().then(() => console.log("this is Promise.resolve 1"));
process.nextTick(() => console.log("this is process.nextTick 1"));
```

Close queue callbacks are executed after all other queue callbacks in a given iteration of the event loop.

Summary

The event loop is a C program that coordinates the execution of synchronous and asynchronous code in Node.js. It manages six different queues: `nextTick`, `promise`, `timer`, `I/O`, `check`, and `close`.

To add a task to the `nextTick` queue, we use the `process.nextTick()` method. To add a task to the `promise` queue, we resolve or reject a promise. To add a task to the `timer` queue, we use `setTimeout()` or `setInterval()`.

To add a task to the `I/O` queue, we execute an async method. To add a task to the `check` queue, we use the `setImmediate()` function. Finally, to add a task to the `close` queue, we attach `close` event listeners.

The order of execution follows the same order listed here. However, it's important to note that the `nextTick` and `promise` queues are executed in between each queue and also in between each callback execution in the `timer` and `check` queues.



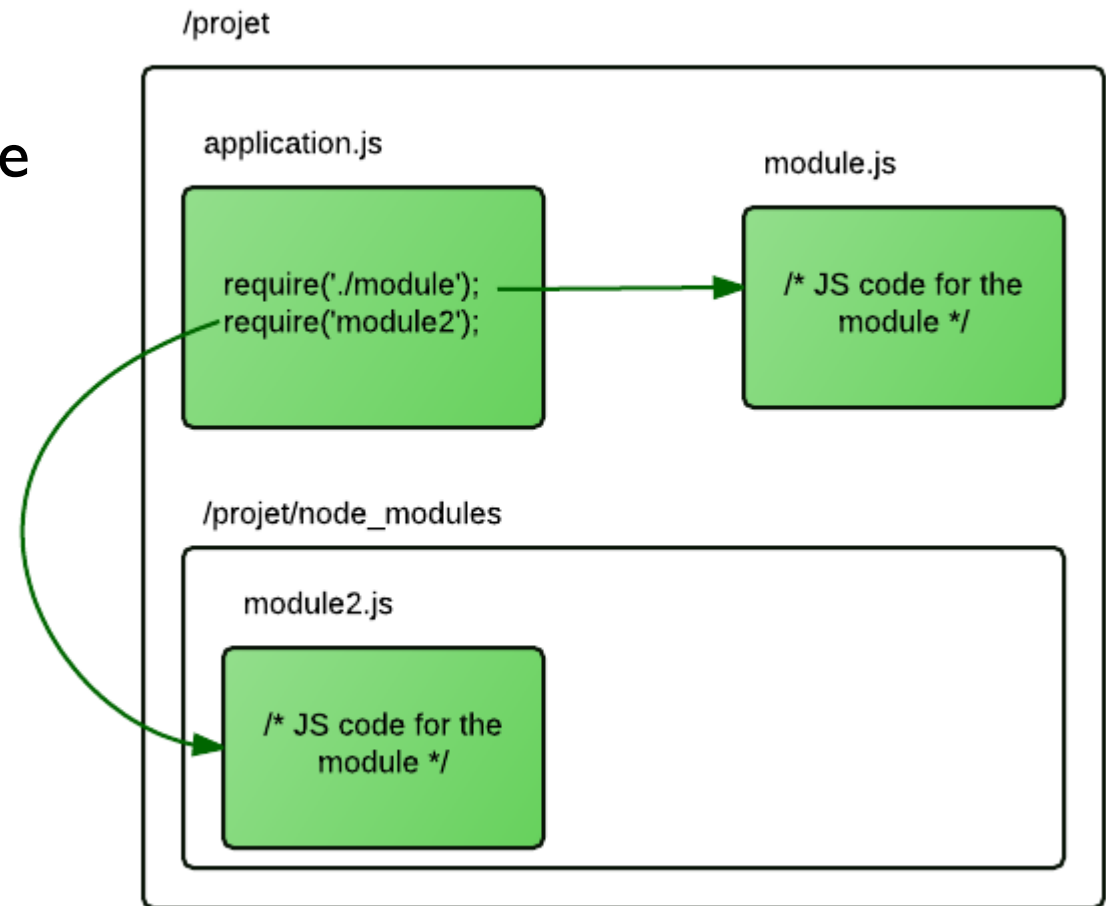
Module Intro



Rujuan Xing

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](#)

▶ 3rd party modules on 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:

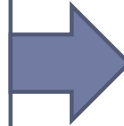
```
return module.exports;
```

module.exports

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

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

module.exports = sayHi;
```

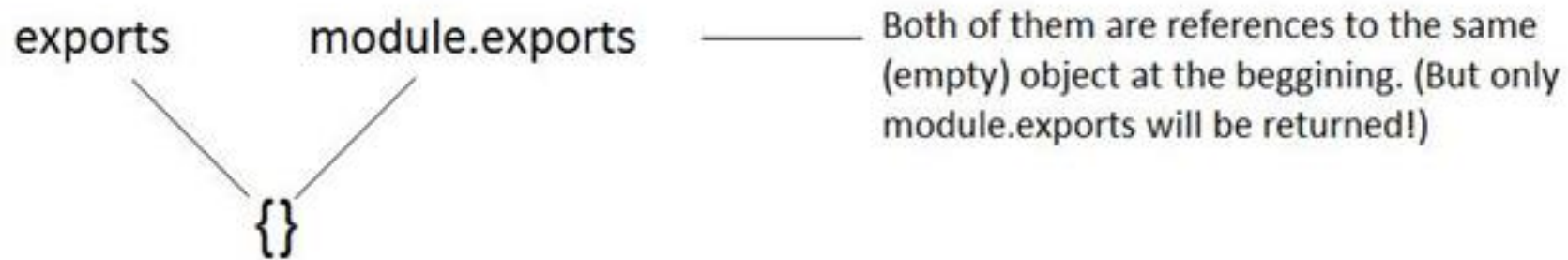


```
// app.js
let hello = require('./helloModule');

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.

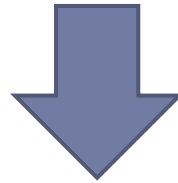


```
exports = function doSomething() {  
  console.log('blah blah');  
}
```

`doSomething()` isn't
exported.

Using Modules

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



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

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.
```

Resources

- ▶ [What is a Full Stack developer?](#)
 - ▶ [Wait, Wait... What is a Full-stack Web Developer After All?](#)
 - ▶ [The Myth of the Full-stack Developer](#)
 - ▶ [Multi-tier Architecture](#)
 - ▶ [What is the 3-Tier Architecture?](#)
 - ▶ <https://www.journaldev.com/7462/node-js-architecture-single-threaded-event-loop>
-
- ▶ [Nodejs.org](#)
 - ▶ [Npmjs.com](#)
 - ▶ [Node API Documentation](#)
 - ▶ [NPM Documentation](#)
 - ▶ [The Node.js Event Loop, Timers, and process.nextTick\(\)](#)
 - ▶ <https://www.builder.io/blog/visualizing-nodejs-close-queue>