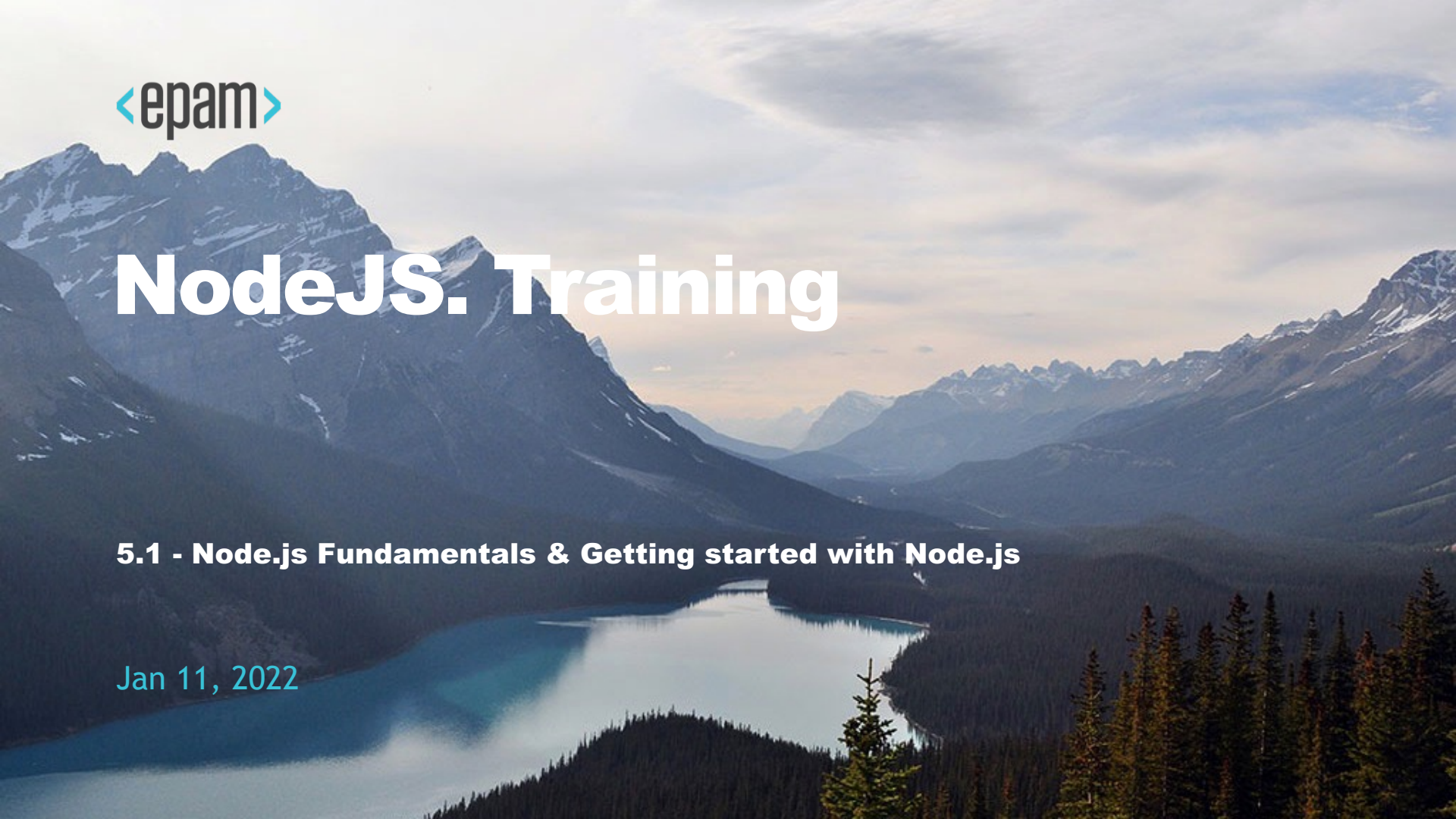


NodeJS. Training

5.1 - Node.js Fundamentals & Getting started with Node.js

Jan 11, 2022

AGENDA

- Node.js fundamentals
  - What is Node.js?
  - Why Node.js?
  - Blocking and Non-Blocking and Event Driven terms
- Events and Event Loop
- NPM and package.json
  - What is NPM?
  - NPM package structure
  - Npm tasks, dependencies, devDependancies
  - Registry, Private and public packages, scopes
- Node.js module patterns
  - CommonJS
  - ES6 modules
- Yarn
- Q&A

Node.js Fundamentals

- What is Node.js?
- Why Node.js?
- Blocking and Non-Blocking and Event Driven

Prerequisites

- JavaScript & ECMAScript
- Process and Threads
- Single Thread and Multi Thread
- Asynchronous and Synchronous Programming
- Blocking and Non-Blocking operations

Prerequisites - JavaScript

- JavaScript (JS) is a lightweight, interpreted, or just-in-time compiled programming language with first-class functions. While it is most well-known as the scripting language for Web pages, many non-browser environments also use it, such as Node.js, Apache CouchDB and Adobe Acrobat.
- JavaScript is a prototype-based, multi-paradigm, single-threaded, dynamic language, supporting object-oriented, imperative, and declarative (e.g. functional programming) styles.

-- Mozilla

- JavaScript is a general-purpose scripting language that conforms to the ECMAScript specification.

Prerequisites - ECMA Script (ES)

- ECMA Script is the standard and specification of JavaScript.
- ECMA – An organization that creates standard for technologies.
- ECMA-262 is the standard of Script Language.
- **ECMAScript provides the rules, details, and guidelines that a scripting language must observe to be considered ECMAScript compliant.**

Prerequisites – A JavaScript Engine

- A program or interpreter that understands and executes JavaScript code.
- JavaScript engines are commonly found in web browsers.

Browser Name	Engine Name
Chrome Browser	V8 Engine
Firefox Browser	SpiderMonkey
Edge Browser	Chakra
Safari Browser	Webkit

Prerequisites - A JavaScript runtime

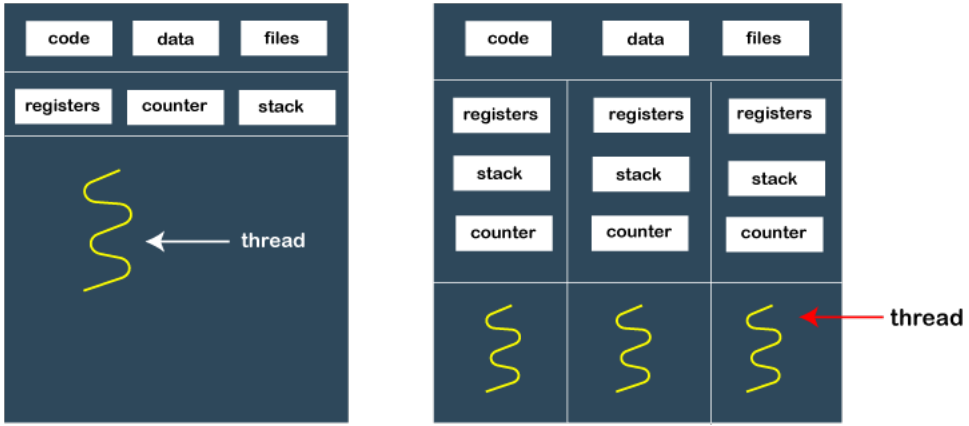
The environment in which the JavaScript code runs and is interpreted by a JavaScript engine. The runtime provides the host objects that JavaScript can operate on and work with.

1. **Client Side** – Browsers - Web APIs
2. **Server Side** - Node.js - Server-related host objects such as the file system, processes, and requests are provided in Node.js.

Prerequisites – Process & Thread

Process	Thread
Process is a Program under execution.	Thread are the subset of process and run within process.
Example: opening Chrome Browser.	**Example**: • Open new tab in browser • Downloading Assets • Rendering Websites
There are child processes for the main thread which is executed one after another.	There is no child threads.
Process can be forked.	Thread can be cloned.
Each process are isolated.	Threads shares memory.

Prerequisites – Single vs Multi Thread



Single-threaded process

Multi-threaded process

Prerequisites - Asynchronous and Synchronous Programming

Synchronous

- Executes in order.
- A task should be completed before starting the next task.

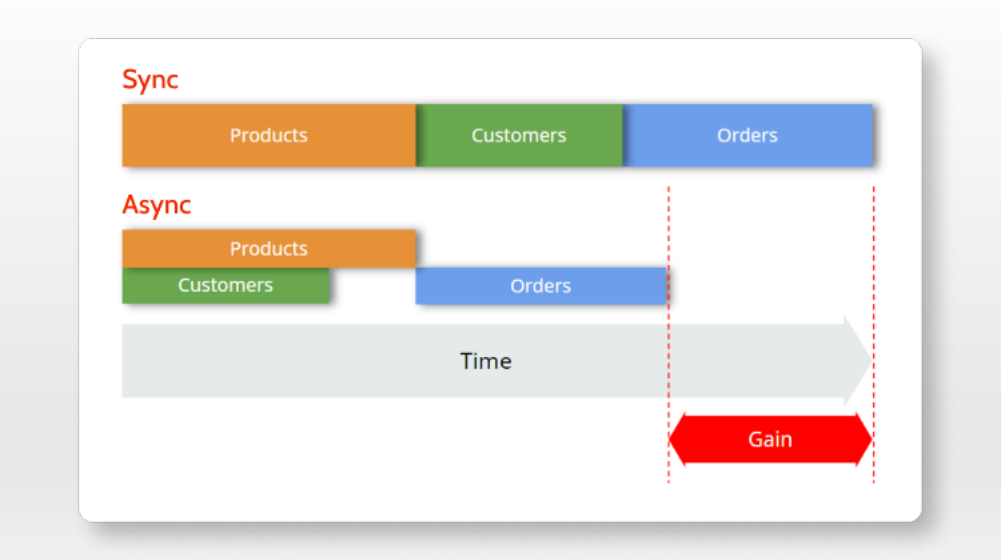
Examples: Instant Messages Chats, Phone Calls

Asynchronous

- Executes without order.
- Can move on to another task before it finishes.

Examples: Email, Recorded Voice Message

Prerequisites - Asynchronous and Synchronous Programming (cont..)



Prerequisites - Asynchronous and Synchronous Programming (cont..)

Sync	Async

let a = 1;
let b = 2;
 	
console.log('Sync');
console.log(a);
console.log(b);
	let a = 1;
	let b = 2;
	
	setTimeout(() => { console.log('Async'); }, 1000)
	console.log(a);
	console.log(b);

Prerequisites - Asynchronous and Synchronous Programming (cont..)

- Node.js promotes an asynchronous coding style from the ground up, in contrast to many of the most popular web frameworks.
- Many of the functions in Node.js core have both synchronous and asynchronous versions.

Use the asynchronous functions, avoid the synchronous ones!

Prerequisites - Asynchronous and Synchronous Programming (cont..)

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

fs.readFile('example.file', 'utf8', function (err, data) {
  if (err) {
    return console.log(err);
  }
  console.log(data);
});

*******

var fs = require('fs');

var data = fs.readFileSync('example.file','utf8');
console.log(data);
```

Prerequisites - Asynchronous and Synchronous Programming (cont..)

When only reading a file or two, or saving something quickly, the difference between synchronous and asynchronous file I/O can be quite small. On the other hand, though, when you have multiple requests coming in per second that require file or database IO, trying to do that IO synchronously would be quite thoroughly disastrous for performance.

Prerequisites - Blocking and Non-Blocking I/O

Blocking

- Blocking is when the execution of additional JavaScript in the Node.js process must wait until a non-JavaScript operation completes.
- JavaScript that exhibits poor performance due to being CPU intensive rather than waiting on a non-JavaScript operation, such as I/O, isn't typically referred to as blocking.
- Executes synchronously.

Non-Blocking

- Executes asynchronously.
- All the I/O methods in the Node.js standard library provide asynchronous versions.

Prerequisites - Blocking and Non-Blocking I/O (cont..)

Using the File System module as an example, this is a **synchronous** file read:

```
const fs = require('fs');
const data = fs.readFileSync('/file.md'); // blocks here until file is read
```

And here is an equivalent **asynchronous** example:

```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
});
```

Prerequisites - Blocking and Non-Blocking I/O (cont..)

Let's expand our example a little bit:

```
const fs = require('fs');
const data = fs.readFileSync('/file.md'); // blocks here until file is read
console.log(data);
moreWork(); // will run after console.log
```

And here is a similar, but not equivalent asynchronous example:

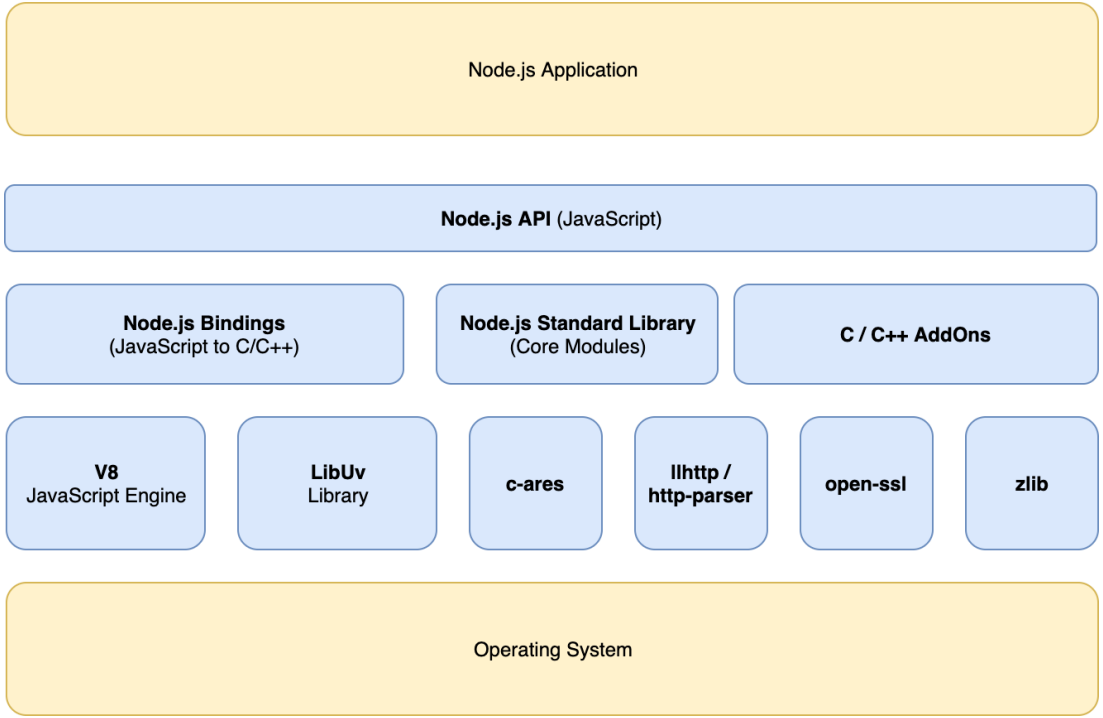
```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
  console.log(data);
});
moreWork(); // will run before console.log
```

What is Node.js

- Node.js is an open-source & cross-platform runtime environment for JavaScript.
- Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser.
- Node.js provides single threaded
- Node.js is asynchronous I/O.
- Node.js are written using non-blocking paradigms.

- Officially first release was 2009.
- Ryan Dahul is the original author of node.js
- Currently it was managed by OpenJs Foundation.

Node.js Architecture

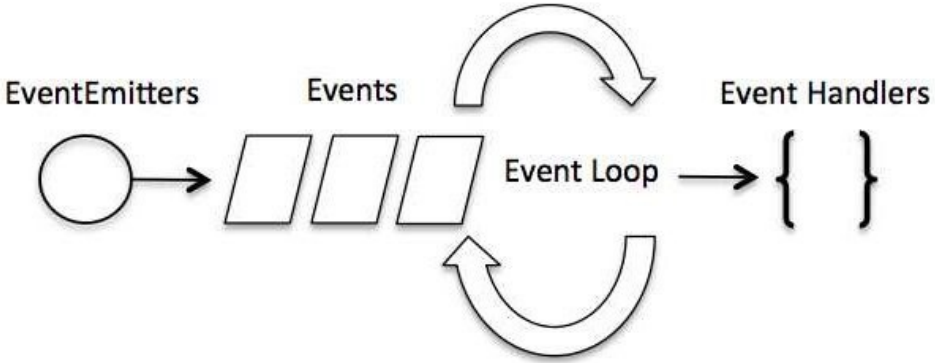


Why Node.js

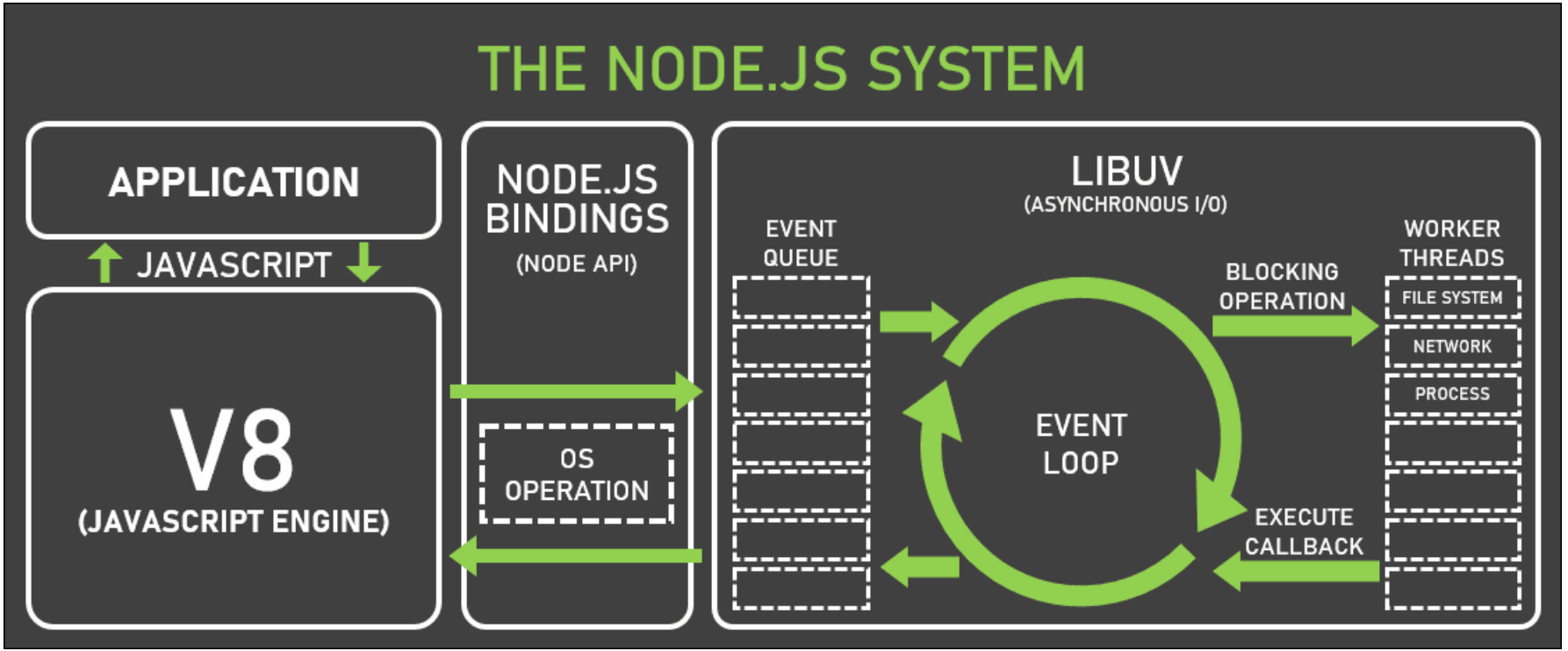
- Node.js is designed to build scalable network applications.
- Because of single-thread, no worries of dead-locking of process.
- It doesn’t perform I/O operations directly, so process never block.
- There are many pre-build packages/modules available to use.

Events

- Every action on a computer is an event. Like when a connection is made or a file is opened.
- Node.js is perfect for event-driven applications.
- As soon as Node starts its server, it simply initiates its variables, declares functions, and then simply waits for the event to occur.



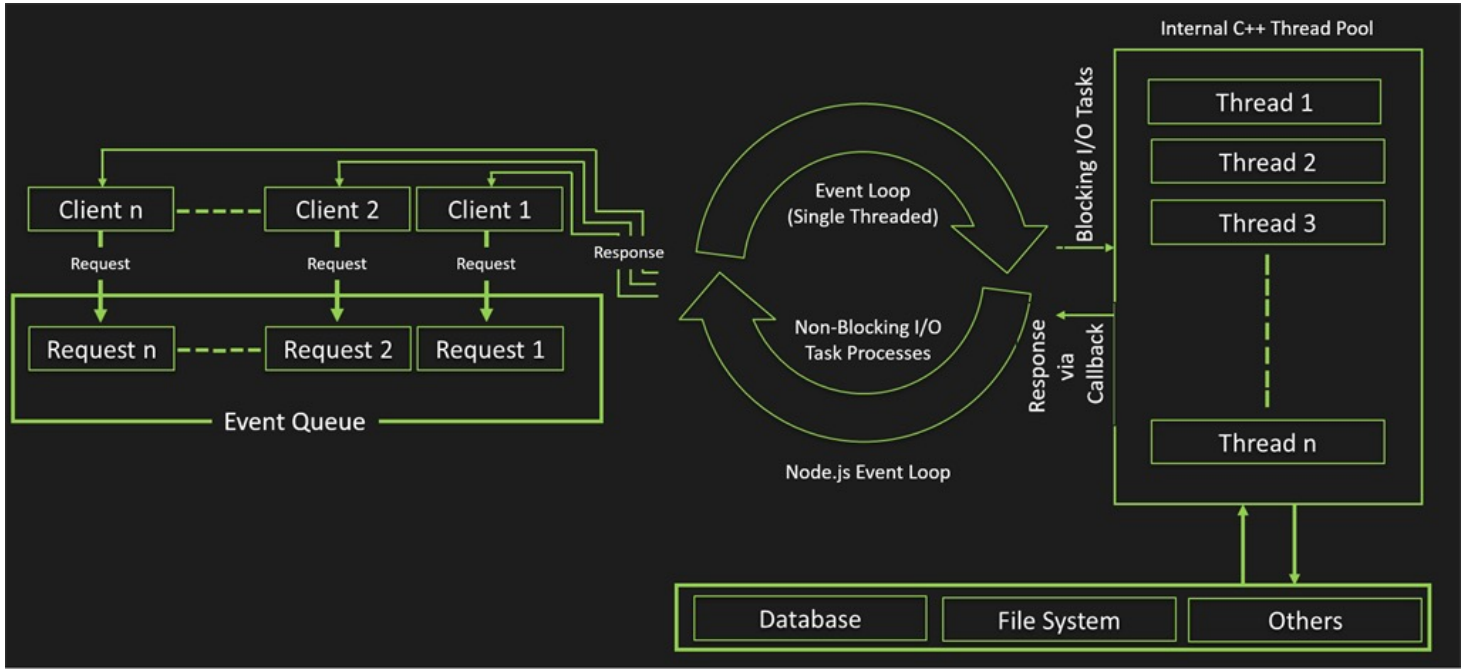
Event Loop


THE NODE.JS SYSTEM
APPLICATION
JAVASCRIPT
V8
(JAVASCRIPT ENGINE)
NODE.JS
BINDINGS
(NODE API)
OS
OPERATION
LIBUV
(ASYNCHRONOUS I/O)
EVENT
QUEUE
EVENT
LOOP
BLOCKING
OPERATION
EXECUTE
CALLBACK
WORKER
THREADS
FILE SYSTEM
NETWORK
PROCESS

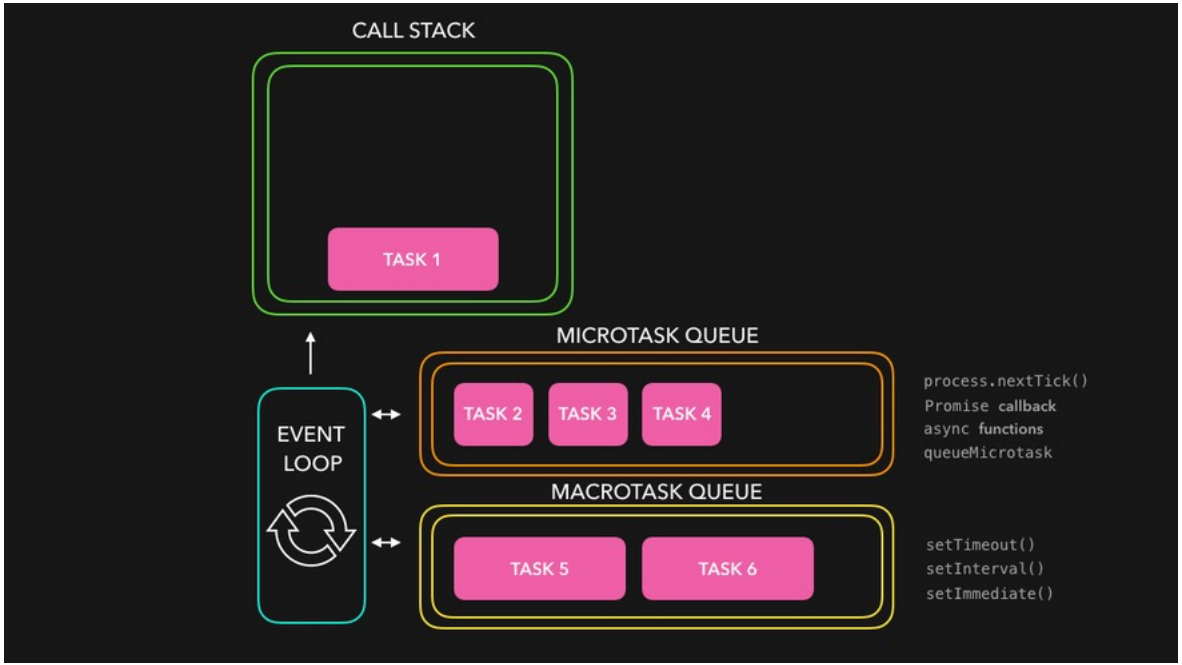
Event Loop (cont..)

1. Incoming request is added to the event queue.
2. Event Loop checks for incoming requests one by one.
3. Any async/non-blocking tasks are passed to thread pool managed by Libuv library.
4. Response from each of these non-blocking call initiates a callback.
5. Event loop executes the callbacks and return response back to client.
6. Sleep if there no request to process.

Event Loop (cont..)



Event Loop (cont..)



Event Loop Explained

timers: this phase executes callbacks scheduled by setTimeout() and setInterval().

pending callbacks: executes I/O callbacks deferred to the next loop iteration.

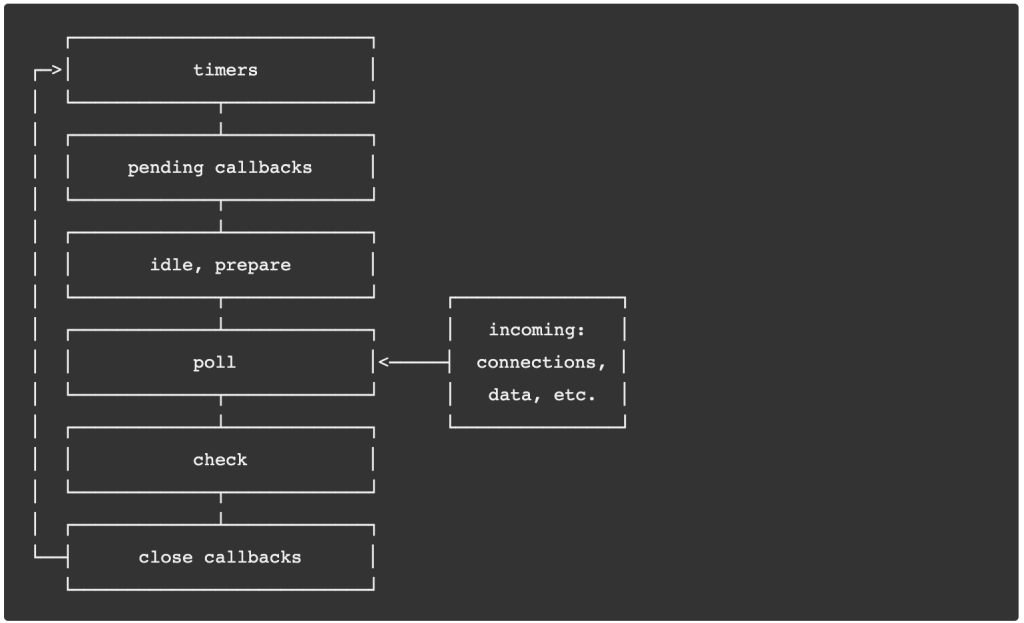
idle, prepare: only used internally.

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', ...).

The following diagram shows a simplified overview of the event loop's order of operations.



> Each box will be referred to as a "phase" of the event loop.

Event Loop Explained (cont..)

1. **timers**: this phase executes callbacks scheduled by setTimeout() and setInterval().
2. **pending callbacks**: executes I/O callbacks deferred to the next loop iteration.
3. **idle, prepare**: only used internally.
4. **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.
5. **check**: setImmediate() callbacks are invoked here.
6. **close callbacks**: some close callbacks, e.g. socket.on('close', ...).

Event Loop Explained (cont..)

- setImmediate() – immediate invoke
- setTimeout() – one time after the specific interval
- setInterval() – run continuously after the specific time
- process.nextTick() – immediate with high priority

Event Loop Explained (FAQ)

In JavaScript, what is the difference between microtasks and tasks?:

A macro task is a collection of distinct and independent tasks. Microtasks are minor tasks that update the state of an application and should be completed before the browser moves on to other activities, such as re-rendering the user interface. Promise callbacks and DOM modification changes are examples of microtasks.

What is the difference between a task queue and a call stack?:

It's up to it to check whether the callstack is empty and whether the task queue has any pending tasks to complete. If the callstack is empty, it will push the job from the queue to the callstack, where it will be processed.

Is a call stack similar to a queue?:

This type of stack is often referred to as an execution stack, control stack, run-time stack, or machine stack, and is frequently abbreviated as "the stack." In summary, a job queue is a list of tasks to be completed (typically maintained persistently), while a call stack is a collection of functions.

What happens when the maximum call stack size is reached?:

When there are too many function calls or a function lacks a base case, the JavaScript exception "too much recursion" or "Maximum call stack size exceeded" occurs.

NPM and package.json

- What is NPM?
- Package installation & usage
- NPM package structure
- Npm tasks, dependencies, devDependancies
- Registry, Private and public packages, scopes
- Global modules

Node.js module patterns

- CommonJS

```
// helper/MathHelper.js
module.exports = {
  add: function(left, right) {
    return left + right;
  },

  times: function(left, right) {
    return left * right;
  }
}
```

```
// program.js
var mathHelper = require('./helper/MathHelper');

console.log(mathHelper.add(5, 8)); // 13
console.log(mathHelper.times(3, 4)); // 12
```

- ES6 modules

```
// helper/MathHelper.js
export function add(left, right) {
  return left + right;
}

export function times(left, right) {
  return left * right;
}
```

```
// program.js
import { add, times } from './helper/MathHelper';

console.log(add(5, 8)); // 13
console.log(times(3, 4)); // 12
```

Yarn (optional)

- Package Manager for Node.js
- Alternate for npm.

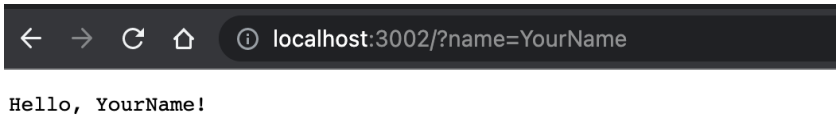
Benefits

- 2 – 3 times faster than npm install
- Yarn uses cache to make installation process faster.
- Faster application startup.

npm install yarn --global

Task

Create the node.js server application – that takes the name from http request URL and response back with ‘Hello, {your name}!’. Refer the screenshot below



Sample Starter code: https://nodejs.org/api/synopsis.html

Q&A


THANKS!
NODE.JS FUNDAMENTALS
BY
ARAVIND APPADURAI