Introduction To Nodejs

Prepared by
Mohamed Abdelmeged
Senior Software Engineer - Fixed Solutions
Freelancer Nodejs Instructor - ITI & AMIT
www.linkedin.com/in/mabdelmeged/

AGENDA

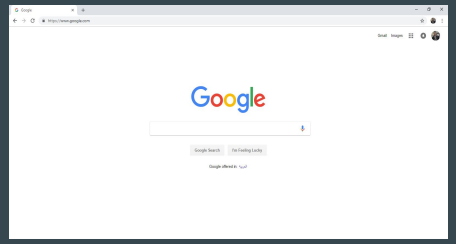
1. Why server side JavaScript?
2. What is NodeJS?
3. Node Specifics.
4. NodeJs Architecture.
5. Is Node Single/Multi Threaded?
6. Node Core Modules.
7. NPM (Node Package Manager).
8. Building Our Custom Modules

Always remember we want to build a web server

1. Why server side JavaScript ?

1. JavaScript is lightweight and fast especially due to using V8 engine. (JIT compiler)
2. JavaScript offers a uniform language on both frontend and backend.
3. We can build reusable code and use it on both backend and frontend.
4. Node is easy to configure.
5. It has the largest Ecosystem NPM.
6. It has a Solid standard. (ECMAScript).

Client Agent



DNS server

(1) DNS look up **www.google.com**

(2) DNS resolves name to **IP address** (8.8.8.8)

2.2 HTTP Request Response Cycle

(3) HTTP Request

(8) HTTP Response

Web Server (nginx / Apache)

(7) HTTP Response

(4) Redirect request to application server

Application Server (express)

(6) Fetched data

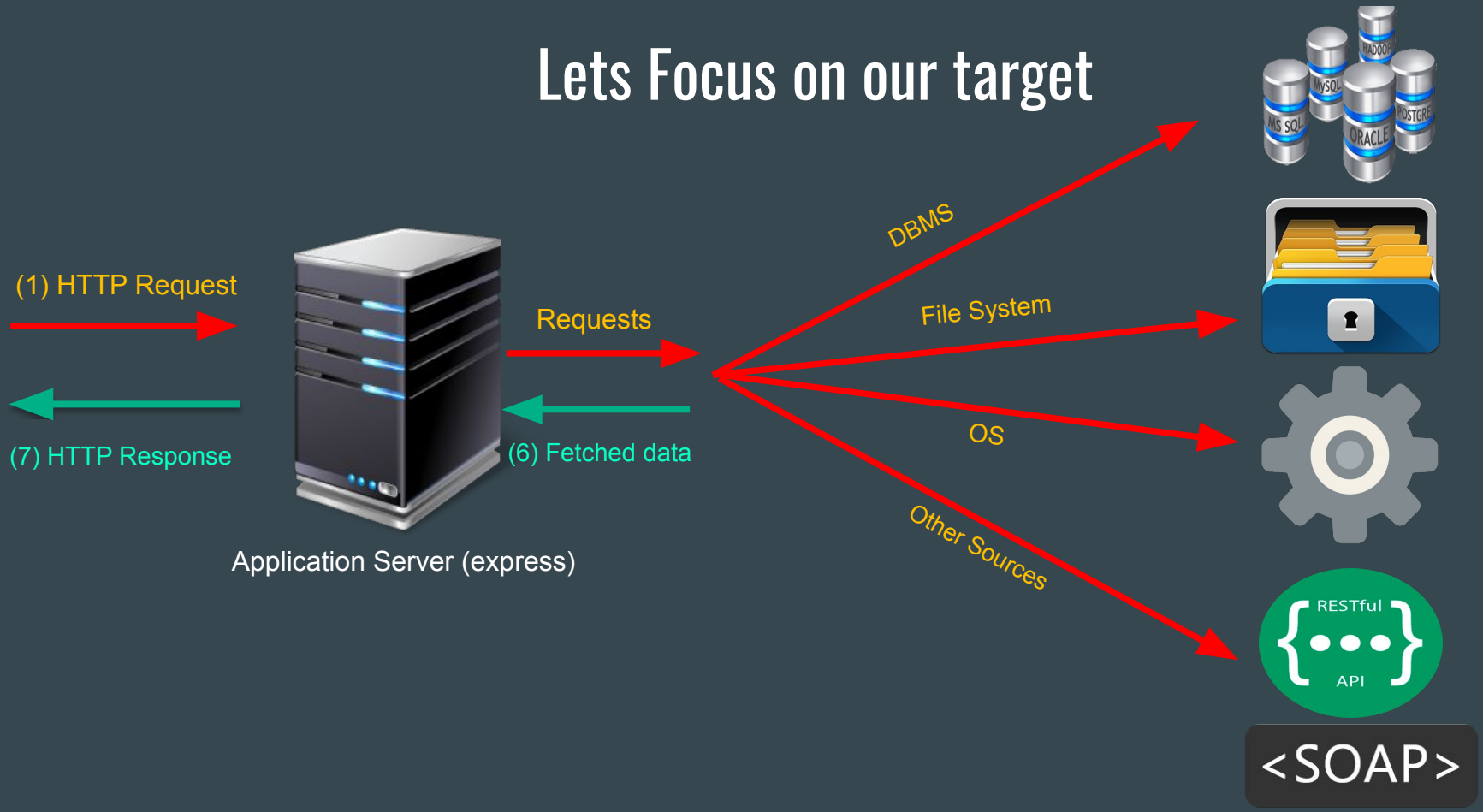
(5) Fetch data

Database server

2. What is Nodejs ..?

1. NodeJs is a JavaScript runtime environment built on Chrome's V8 engine.
2. Node.js was developed by **Ryan Dahl** in 2009
3. NodeJS is an **open source** project.
4. It is **cross platform**. (runs on Linux, Mac and Windows)
5. It uses **event driven** , **non blocking I/O** approach.
6. It uses Google **V8 Engine** (written in C++) to Execute Code.
    1. Generally we need a VM to run Node process. (Chakra, SpiderMonkey, ...etc.)
    2. Open source JIT compiler. It compiles JavaScript into Machine code.
7. It is single threaded but highly scalable.
8. It can handle many concurrent connections at a time

Lets Focus on our target



2.1 What is meant by Runtime Environment Library?

1. It means that Node prepares the environment necessary to run a script.
2. It sets global variables. (process, global, console, Buffer, Promise, ...etc.)
3. It loads core modules that enable us to deal with functionalities that do not exist in JavaScript by default.
4. Runs the main script that is given passed to NodeJS CLI as an argument.

NODE IS DEPLOYED BY BIG BRANDS

Big brands are using Node to power their business

3. Node Specifics

3.1 Node REPL

1. REPL stands for Read, Evaluate, Print, Loop.
2. It is an interactive language shell
3. It takes **user inputs (expressions)**, evaluates them, and returns the result to the user.
4. It is very characteristic of scripting languages.
5. It is much like the console developer tool in the browser.
6. Node REPL can be started from the Terminal by typing the command **node** without any arguments.

3.2 Node Global Objects

These objects are available in all modules.

1. **global**:
    1. An object that includes all other global objects. (as **window** obj. in browser)
    2. It includes, setTimeout, setInterval, console, debugger, Promise, Buffer, process, ...etc.
2. **process**:
    1. it is the interface to the Node process running.
    2. it is an instance of **Process module.** “a built-in module in node”
    3. It includes some data and functions about the current process, command line args, exit, cwd, ...etc.
    4. It includes the environment variables via (**process.env**)
    5. It is an **EventEmitter** Object.

3.3 Environment Variables

1. An environment variable is a named variable that contains data used by one or more applications.
2. They provide a simple way to share configuration settings between multiple applications and processes in Linux.
3. They are available via **process.env** object.
4. It can be used for setting sensitive data (credentials) instead of hardcoding them. Then we can reference them.
5. Ex: process.env.MYSQL_PASSWORD / process.env.NODE_ENV

3.4 Command Line Arguments

1. Command line argument is an argument provided to the program when executed.
2. They are available via **process.argv**
3. Ex: $node app.js --inspect=127.0.0.1:4000
    1. process.argv = [‘path/to/node’, ‘app.js’, ‘--inspect=127.0.0.1:4000’]
4. **yargs** is one of the most famous libraries used to interpret command line arguments.

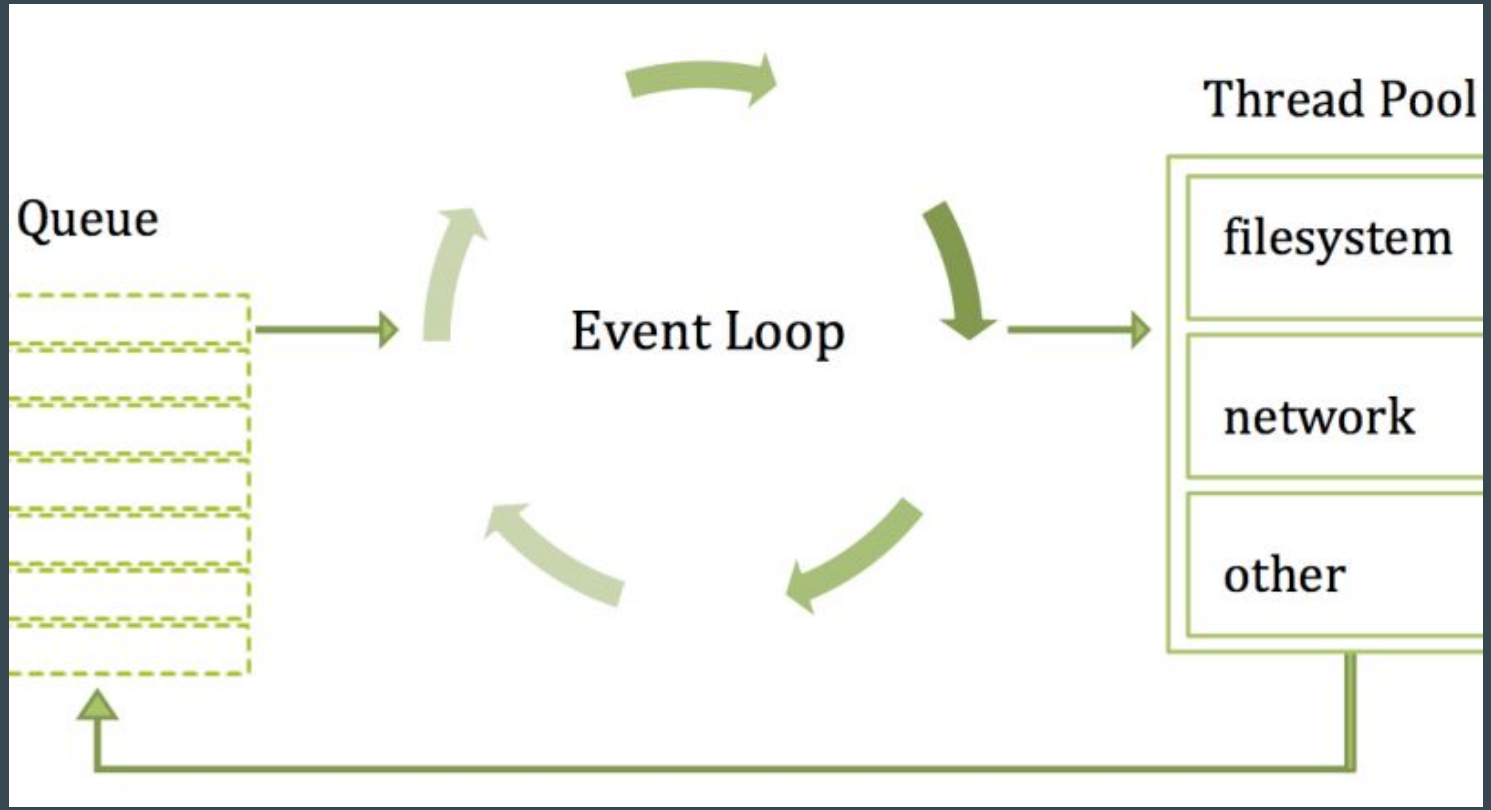
3.5 Debugging Node Application

1. Debugging means locating and correcting code errors in a computer program.
2. Node.js includes an out-of-process debugging utility.
    1. node **inspect** myscript.js
3. **V8 Inspector Integration for Node.js**:
    1. V8 Inspector integration allows attaching Chrome DevTools to Node.js instances for debugging.
    2. V8 Inspector can be enabled by passing the **--inspect** flag when starting a Node.js app.
        i. node --inspect myscript
        ii. To break on the first line of the code, pass the **--inspect-brk** flag instead
        iii. By default, it will listen at host and port **127.0.0.1:9229.**
        iv. Open chrome://inspect, click on configure to confirm the target host and port are listed.
        v. you can check the debugger settings on **http://localhost:9229/json/list**

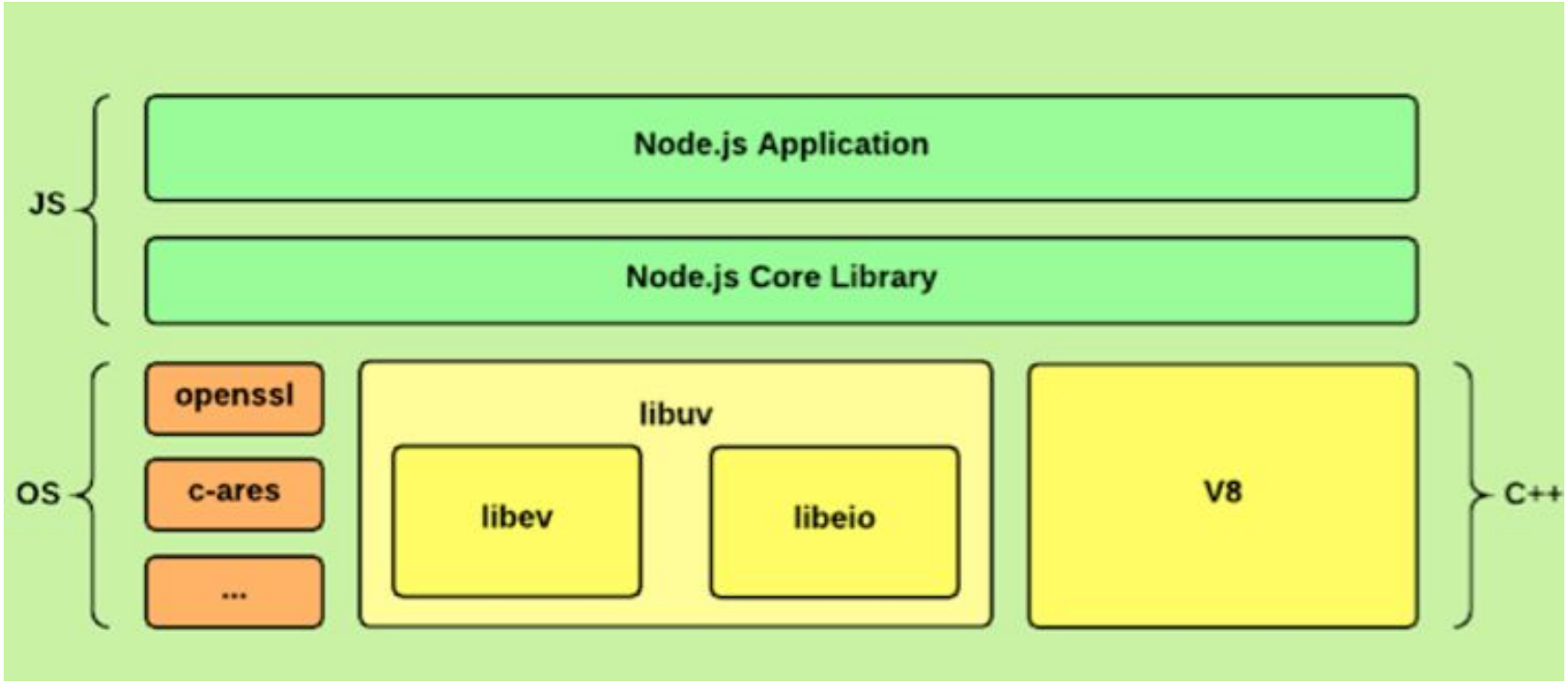
3.6 Let's heat up

1. JSON
2. Map
3. Filter
4. Reduce
5. Functions are first class citizens: they can be treated as normal objects.
6. Arrow functions
7. Callbacks
8. event loop
    - http://latentflip.com/loupe/

3.7 Nodejs Event Loop

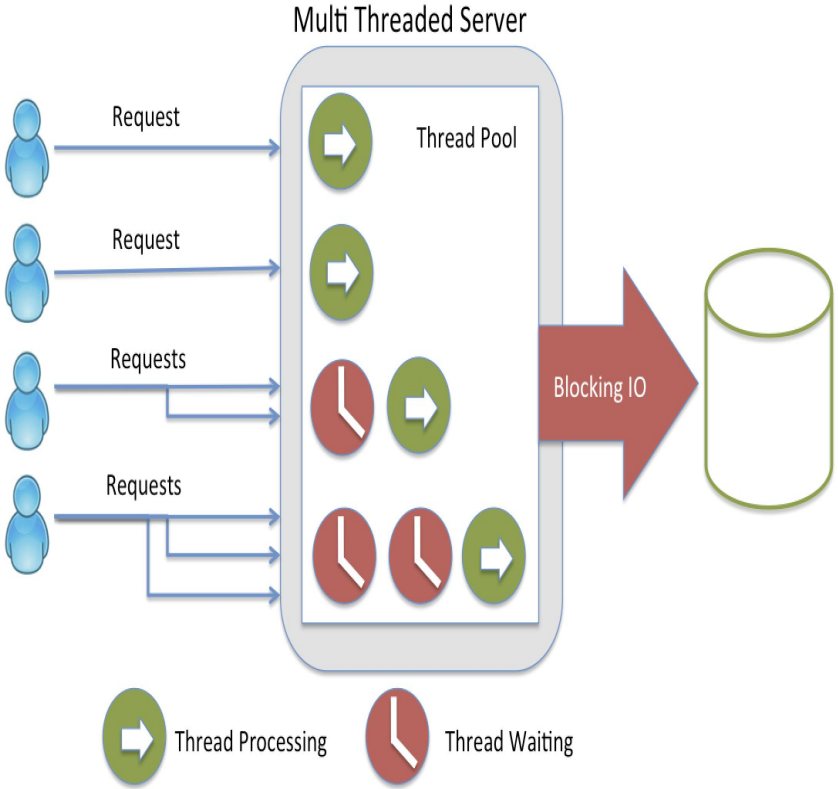


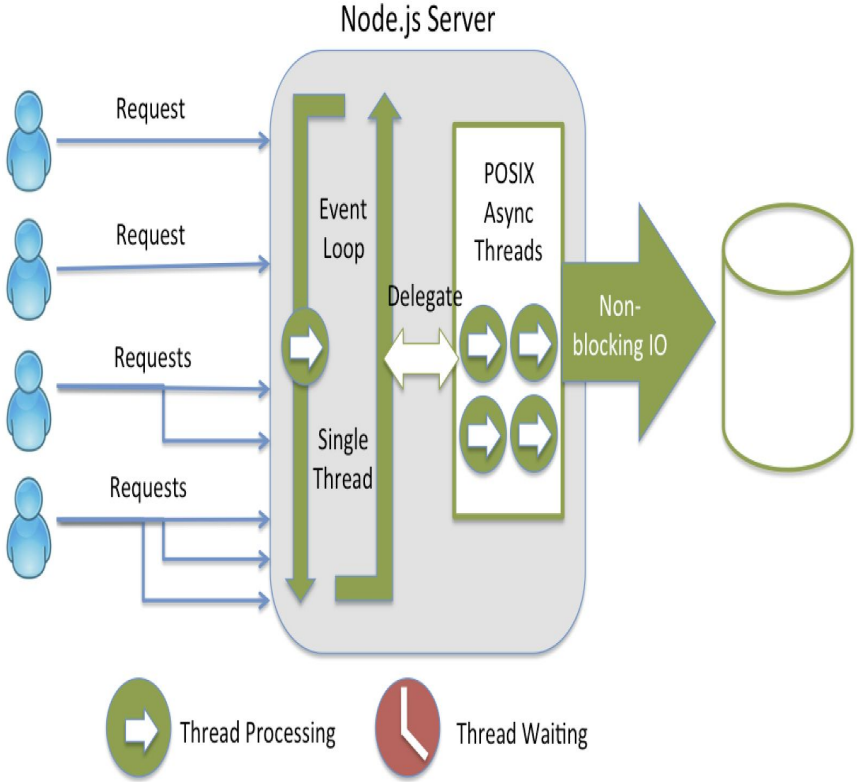
4. NodeJs Architecture



5. Is Node Single/Multi Threaded ...?

5.1 Is Node Single/Multi Threaded ...?





5.2. Blocking Vs Non- Blocking

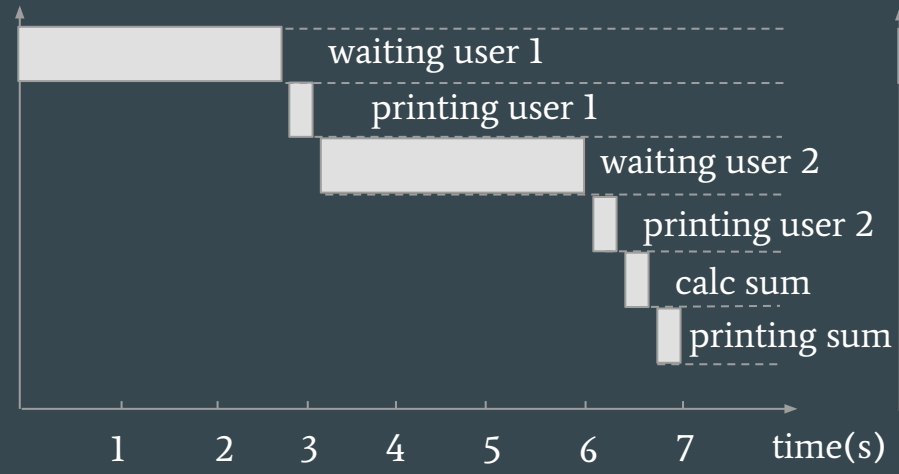
Blocking

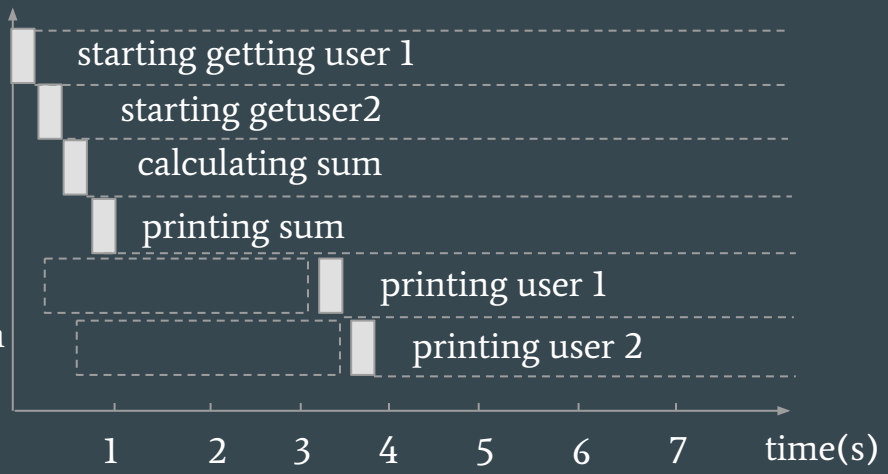
Non-Blocking

5.2. Blocking Vs Non- Blocking

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

var user1 = getUserSync('123');
console.log('user1', user1);

var user2 = getUserSync('321');
console.log('user2', user2);

var sum = 1 + 2;
console.log('The sum is ' + sum);
```

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

getUser('123', function (user1) {
  console.log('user1', user1);
});

getUser('321', function (user2) {
  console.log('user2', user2);
});

var sum = 1 + 2;
console.log('The sum is ' + sum);
```





DO NOT USE NODE FOR CPU INTENSIVE TASKS

6.1 Building Our Custom Modules

1. A Module is a reusable block of code whose existence does not accidently impact other code.
2. Node wraps(encloses) any module in an **IIFE** (immediately Invoking Function Expression)
3. This IIFE is invoked by Node runtime when the module is required.
4. Node passes to this function 5 parameters:
    1. exports: it is the object returned when we require a module. it is an alias to module.exports.
    2. require: it is a function used to require external modules to the current module.
    3. module: it is an object that includes all the information related to the current module.
    4. __filename: the absolute path of the current file.
    5. __dirname: the absolute path of the current working directory.

6.1 Building Our Custom Modules – cont.

5. To expose any variable/function to other modules, attach it to **module.exports** object
6. Modules are **cached** when they are first required. Therefore if you require the module twice or more, it will refer to the same instance of **module.exports** object.
7. **exports** is an alias to **module.exports**, so be careful of this:
   - ➢ exports = function() { }; // this will break the reference to **module.exports** and it will be a local variable

6.2 Exporting & requiring our modules

1. we can **export** our modules as follows:
   a. module.exports = function() { }
   b. module.exports = {

```
function1: () => {},
function2: () => {},
property1: value,
}
```

   c. module.exports.myFunction1 = () => { } ; module.exports.myFunction2 = () => { }
   d. exports.myFunction1 = () => { } ; exports.myFunction2 = () => { }

1. we can **require** our custom modules as follows:
   b. const myCustomModule = require(‘ ./path/to/mymodule’) // we can omit .js

6.3 How Modules are wrapped in Node

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

            var greet = function() {
              console.log('Hello!');
            };

            module.exports = greet;


});


return module.exports;
```

7. Node Core Modules

1. Node has a lot of built-in modules with an extremely handy APIs.
2. To use any of these modules, just require them by name.
3. The Most Important Modules are:
    1. **fs (File System)**: used to deal with file system. (read/write)
    2. **os:** used to get some information from the operating system.
    3. **path:** provides utilities for working with file and directory paths.
    4. **http**: provides utilities to deal with http protocol featues. (server, clientRequest ...etc.)
    5. **events:** provide utilities to deal and create events.
    6. **queryString:** provides utilities to deal with url query string part.
    7. **url:** provides utilities to deal with urls.
    8. **Timers**: all functions that run according to predefined time. (setTimeout/ setInterval)

7.1 File System Module (fs)

1. All the methods have asynchronous and synchronous forms.

```
var fs = require('fs');
fs.writeFile('message.txt', 'Hello Node', function
(err) {
  if (err) throw err;
  console.log('It\'s saved!');
});
```

7.2 OS Module

1. Provides a few basic operating-system related utility functions.

```
var os = require('os');
console.log(os.hostname());
console.log(os.type());
console.log(os.platform());
console.log(os.arch());
console.log(os.release());
console.log(os.uptime());
console.log(os.loadavg());
```

7.3 Path Module

1. This module contains utilities for handling and transforming file paths.
2. Almost all these methods perform only string transformations.

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

const filepath = path.join(__dirname, '3-path.js')
console.log(filepath); // ⟹ /home/mhassan/node-course-iti-39/day1/node-playground/3-path.js


const unNormalizedPath = './..////////node-playground/data/'
const normalizedPath = path.normalize(unNormalizedPath)
console.log(normalizedPath)   //  ⟹ ../node-playground/data/
```

7.4 HTTP Module

1. provides utilities to deal with http protocol features. (server, clientRequest ...etc.)

```
var http = require('http');
http.createServer(function (request, response) {
  response.writeHead(200,
    {'Content-Type': 'text/plain'});
     response.end('Hello World\n');
}).listen(8124);
console.log('Server running at http://127.0.0.1:8124/');
```

8. Node Package Manager (NPM)

8.1 What is NPM ?

1. NPM is the world's largest software registry.
2. Open source developers use npm to share and borrow packages.
3. npm consists of three distinct components:
   a) **Website** "https://www.npmjs.com"
      ➢ used to discover packages, set up profiles, ..etc.
   b) **Command Line Interface (CLI) utility** "npm [command] [pkg]
      ➢ developers use cli to interact with npm. Install, update, audit, uninstall ...etc.
   c) **Registry**
      ➢ large public **database** of **JavaScript software** and the **meta-information** surrounding it.
4. We can use any of the installed modules by just requiring it by its name.
   a) Ex: const express = require('express')

8.2 NPM Commands

1. **npm init**: creates package.json file.
2. **npm install** <pkg (s)>: install a package(s).
3. **npm uninstall** <pkg(s)>: uninstall a package(s).
4. **npm update** <pkg(s)>: updates a package(s).
5. **npm run** <script>: runs a command from a "scripts" object named <script>
6. **npm ls**: List installed packages.

Options passed to npm commands

1. **-g**: global module to be saved on a specific directory on a location according to npm configurations. Otherwise it will be saved to the current project node_modules.
2. **--save**: to save changes made to package.json file

8.3 Package.json

1. It is a JSON file that includes some meta data about the project.
2. These properties include:
   a) **name**: project name.
   b) **version**: project version.
   c) **main:** entry point of the project (defaults to server.js).
   d) **License**: project license and permissions
   e) **author**: project author
   f) **scripts**: it is a dictionary containing script commands.
   g) **dependencies**: object that maps a package name to a version range.
   h) **devDependencies**: pkgs that are used for development.

8.4 NPM Folders

1. Global Install:
   a) puts modules usually in **/usr/local** or where node is installed.
   b) Global installs go to {prefix}/lib/node_modules.
   c) Mostly used for tools. (scaffolding, test, build, deploy, ...etc.)

2. Local install:
   a) puts modules in "./node_modules" of current package.

Stay Tuned for Next part.....

Introduction to Expressjs