



Edison



Percepción y **Sistemas** Inteligentes



Raspberry Pi
(Pi Zero incl.)



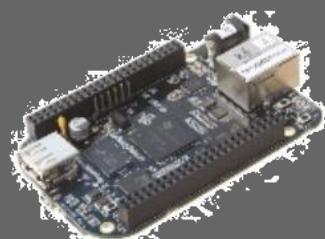
Introduction to NodeJS

Professor:

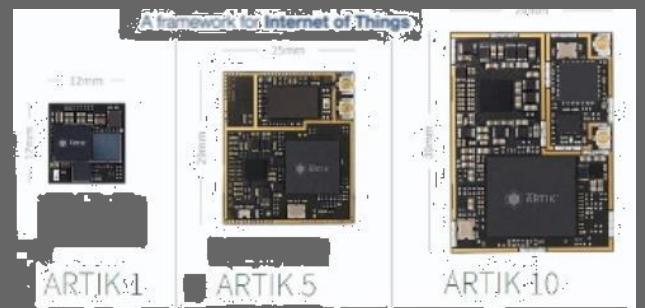
Baldimir Bacca Cortes Ph.D.

Baldimir.bacca@correounivalle.edu.co

Grupo de Investigación en Percepción y Sistemas Inteligentes.

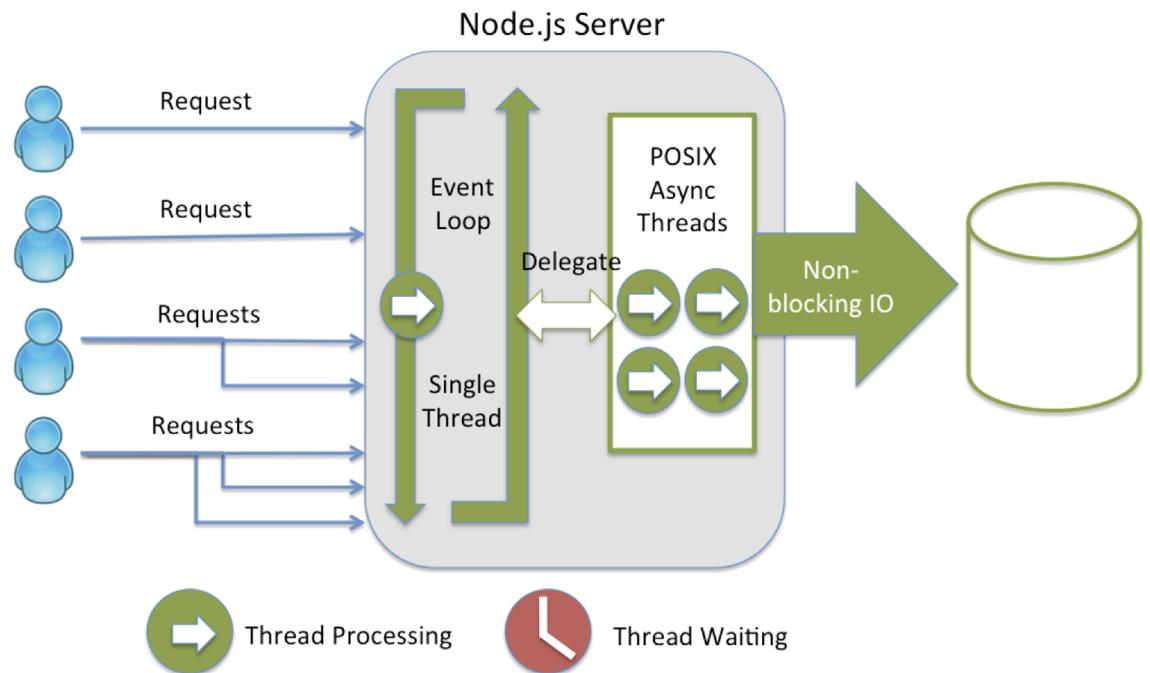


Beaglebone



Contents

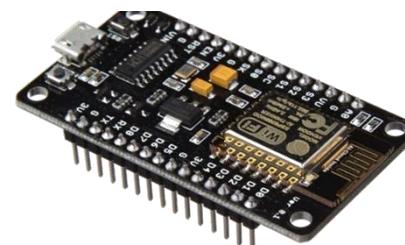
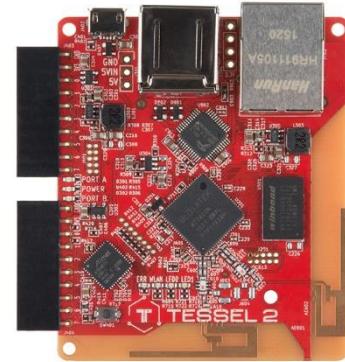
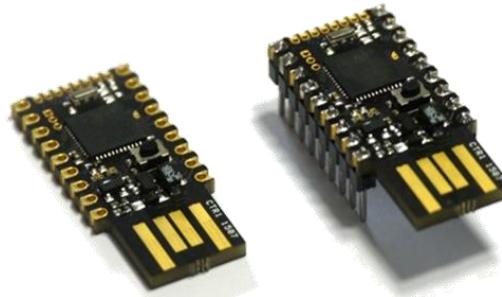
- Introduction.
- Environment setup.
- Architecture and First applications.
- REPL terminal
- NPM
- Callbacks and Events
- Buffers and Streams
- File System.
- Global objects
- Utility modules



Introduction

- **What is NodeJS?**

- NodeJS is an [open source](#) , cross platform runtime environment for [server side](#) and [networking](#) application.
- It is written in [JavaScript](#) and can run on Linux , Mac , Windows , FreeBSD.
- It provided an [event driven architecture](#) and a non blocking I/O that optimize and scalability. These technology uses for real time application.
- It used [Google JavaScript V8](#) Engine to Execute Code.
- V8 compiles JavaScript to native machine code (IA-32, x86-64, ARM, or MIPS ISAs) before executing it.
- It is used by Groupon, SAP , LinkedIn , Microsoft,Yahoo ,Walmart ,Paypal



Edison

Beaglebone

Introduction

- **Why to use NodeJS?**

- It is very **lightweight** and fast
- Node.js was **easy to configure**
- There are **lots of modules available** for free. For example, I found a Node.js module for PayPal.
- NodeJS work with **NoSQL** as well

- **When not use NodeJS**

- Your server request is **dependent** on **heavy CPU** consuming algorithm/Job.
- Node.JS itself does not utilize all core of underlying system and it is single threaded by default, you **have to write logic** by your own to **utilize multi core** processor and make it multi threaded.

Node.JS = RuntimeEnvironment + JavaScriptLibrary

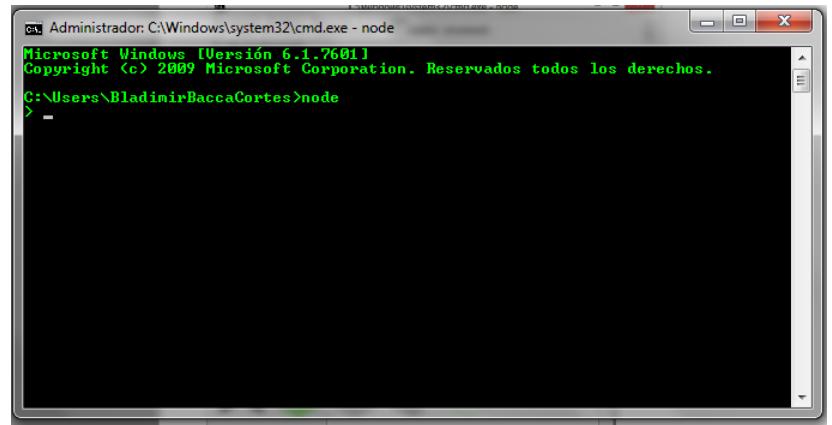
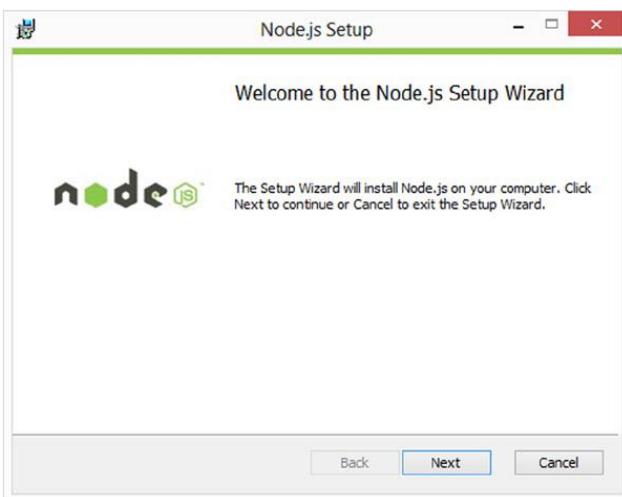
Introduction

- **Features of NodeJS**

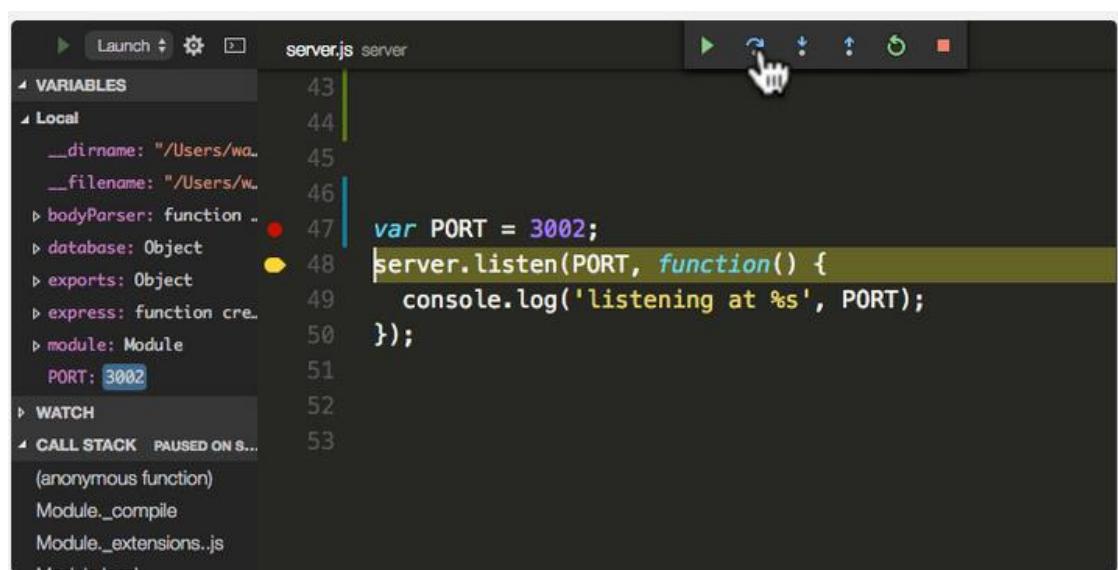
- *Asynchronous and Event Driven* - All APIs of Node.js library are asynchronous, that is, non-blocking. It essentially means a Node.js based server never waits for an API to return data.
- Node.js uses an *event-driven, non-blocking I/O model*, which makes it lightweight.
- It makes use of *event-loops via JavaScript's callback* functionality to implement the non-blocking I/O
- *Single Threaded but Highly Scalable* - Node.js uses a single threaded model with event looping
- *No Buffering* - Node.js applications never buffer any data. These applications simply output the data in chunks
- *License* - Node.js is released under the MIT license
- In not-so-simple words Node.js is a *high performance network applications framework*, well optimized for high concurrent environments.

Environment Setup

- **Download:**
 - From <https://nodejs.org/en/download/>
- **Check the installation:**
 - Open a command terminal.
 - Type *node*.
- **IDE – Visual Studio Code**
 - Download:
<https://code.visualstudio.com/download>



A screenshot of a Windows Command Prompt window titled 'cmd Administrador: C:\Windows\system32\cmd.exe - node'. The window shows the command 'node' being typed at the prompt. The text in the window is in Spanish, indicating it's a Microsoft Windows version 6.1.7601, Copyright © 2009 Microsoft Corporation. Reservados todos los derechos. The path shown is C:\Users\BladimirBaccaCortes>node> -



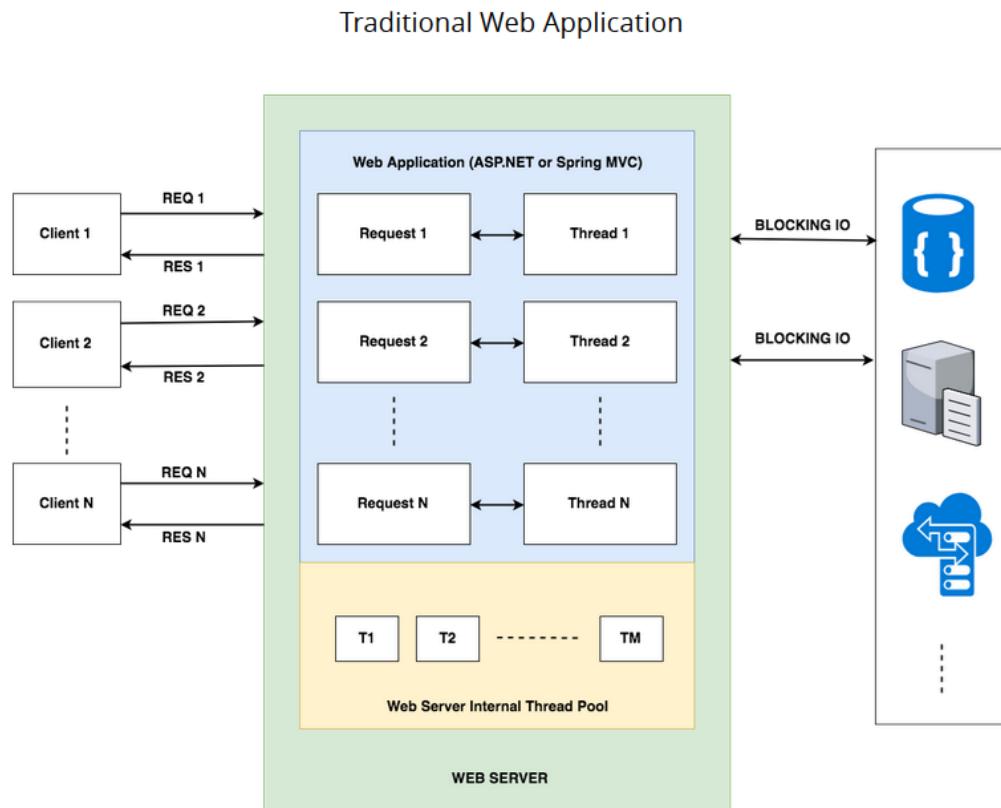
A screenshot of the Visual Studio Code interface. The top bar shows the file 'server.js' and the title 'server'. The main area displays a portion of a Node.js script with syntax highlighting. The code includes variables like 'PORT' set to 3002, and a function that logs 'listening at %s' to the console. On the left, the 'Variables' sidebar shows local variables such as '_dirname', '_filename', 'bodyParser', 'database', 'exports', 'express', 'module', and 'PORT'. The 'WATCH' and 'CALL STACK' sections are also visible.

```
var PORT = 3002;
server.listen(PORT, function() {
  console.log('listening at %s', PORT);
});
```

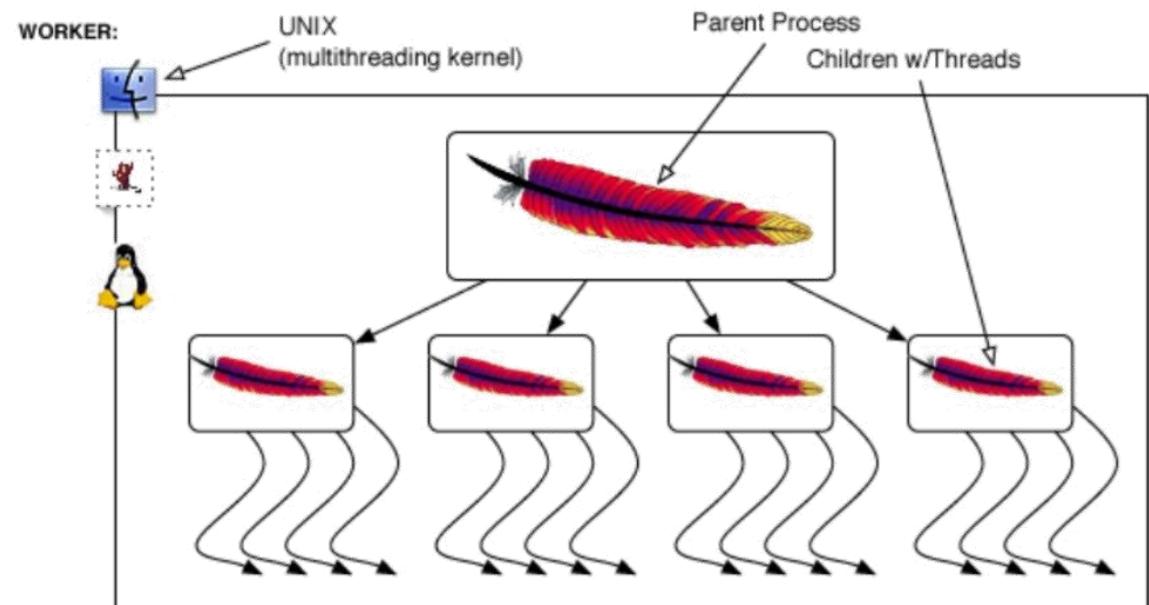
Architecture and First Applications

- Architecture – *Traditional processing model*

Processing Model

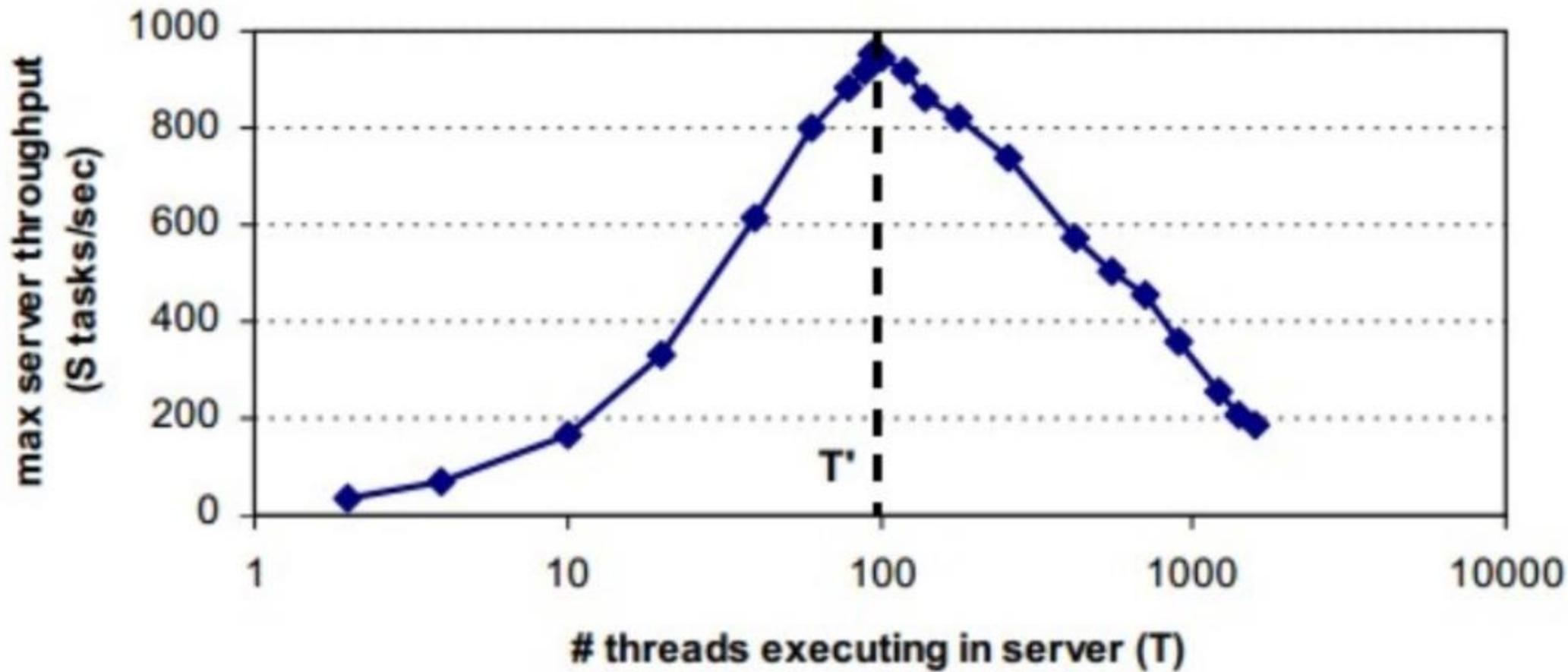


Threads used in Apache(MPM)



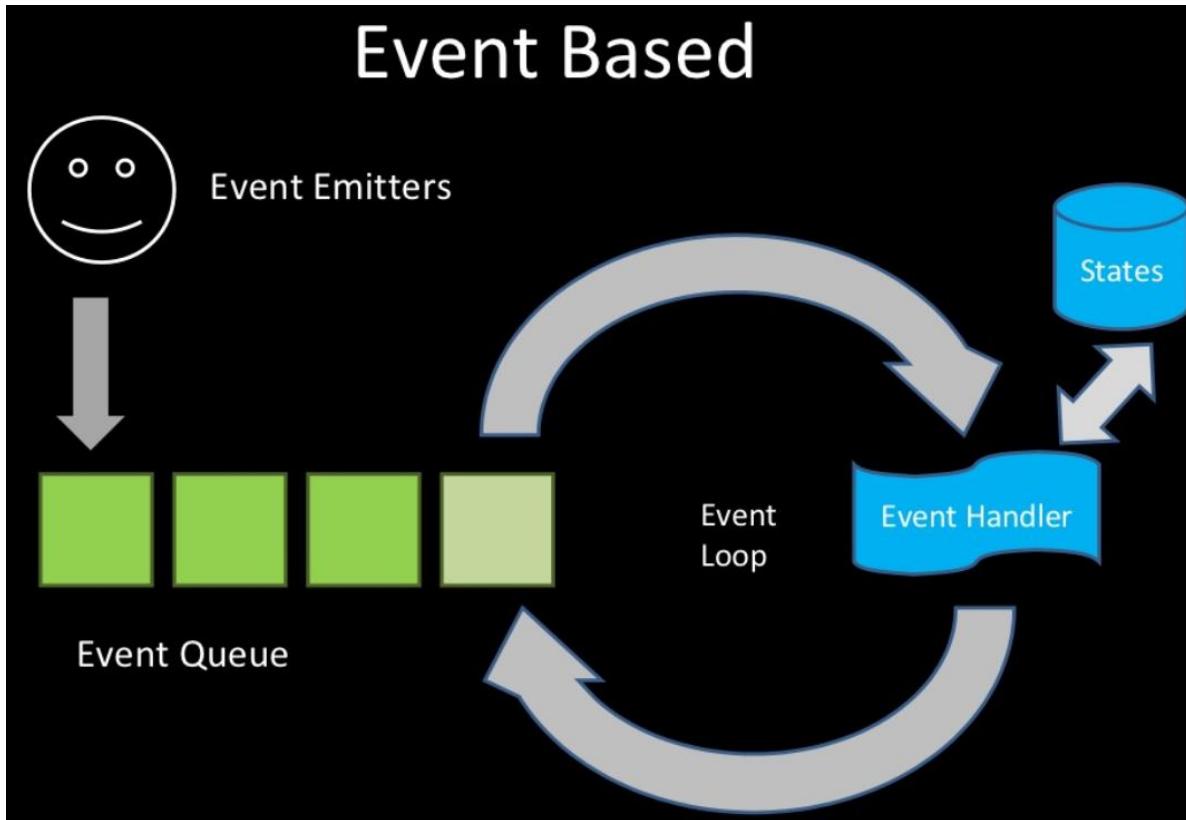
Architecture and First Applications

- Architecture – *Traditional processing model – Scalability Issues*

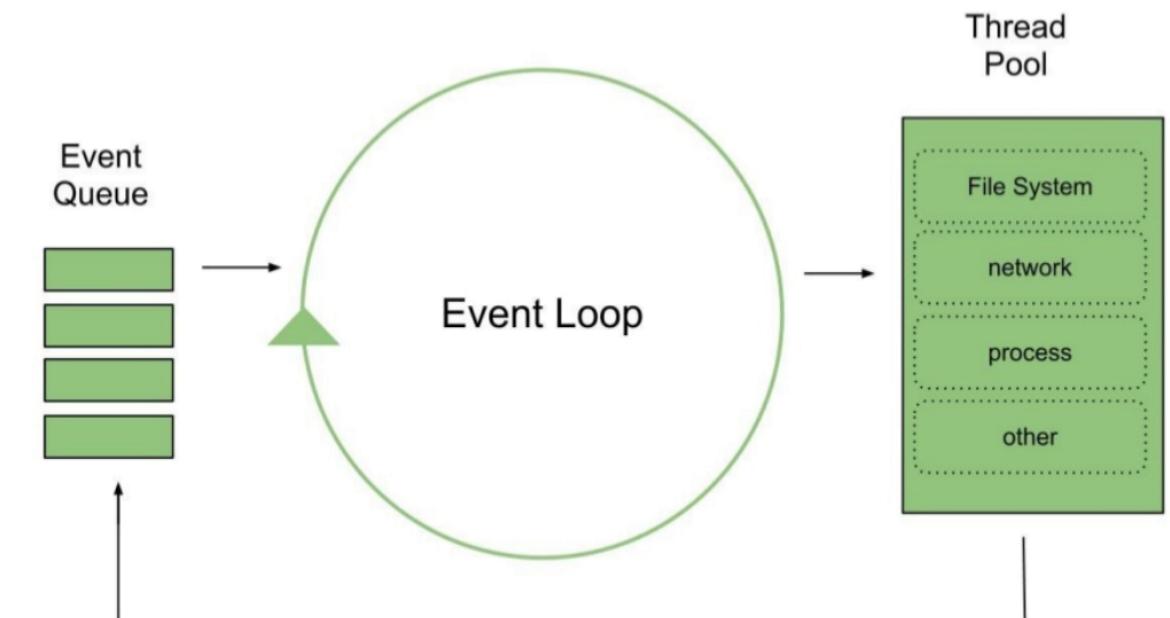


Architecture and First Applications

- Architecture – NodeJS



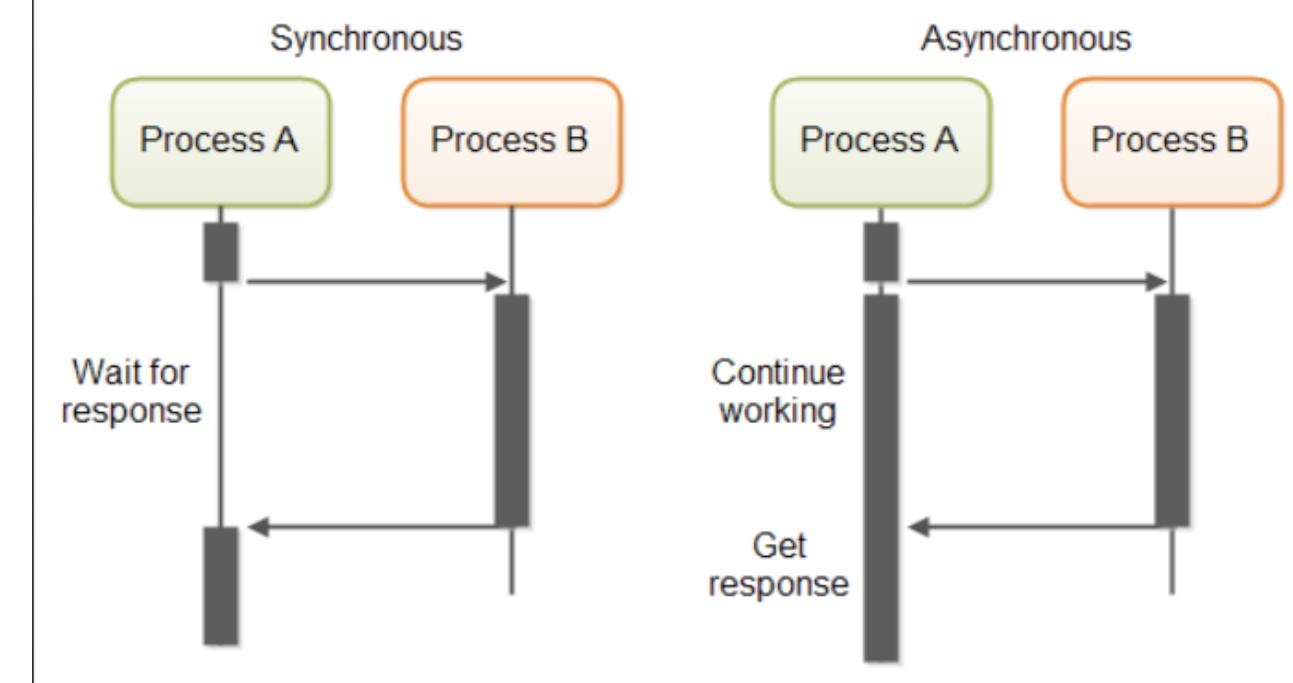
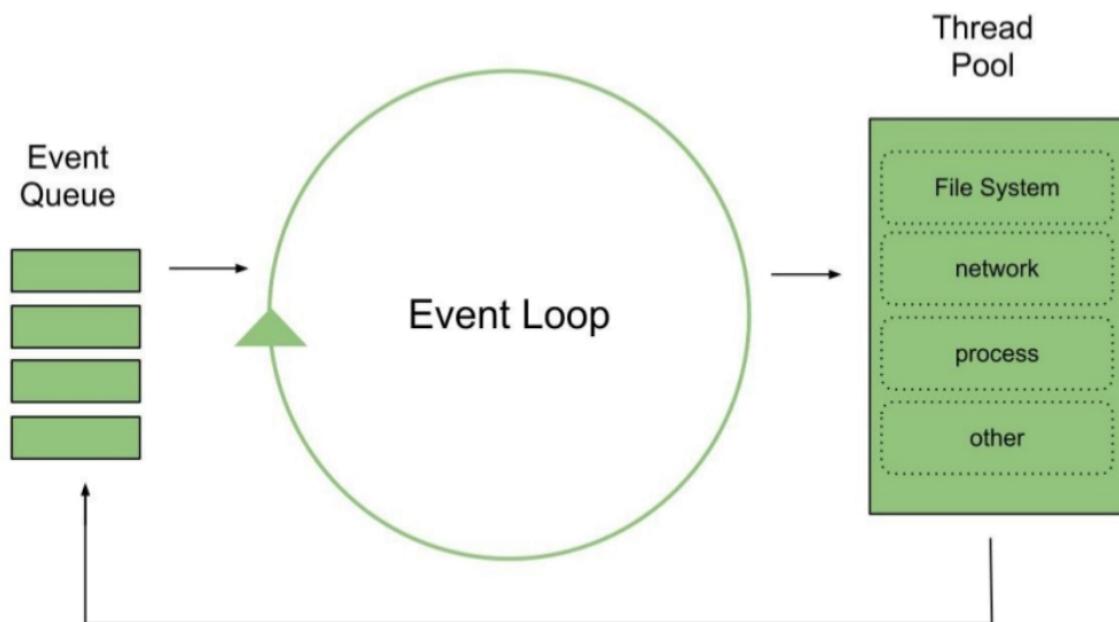
The EventDriven model



Architecture and First Applications

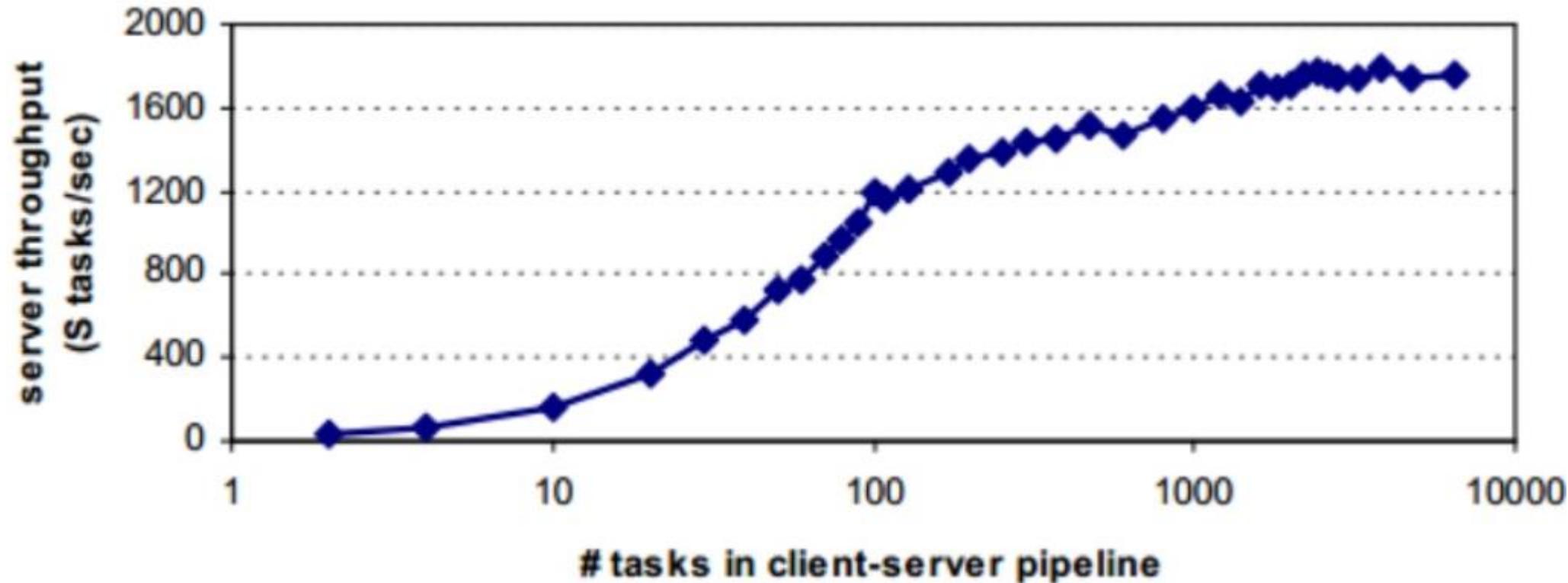
- **Architecture – NodeJS – Synchronous /Asynchronous**

The EventDriven model



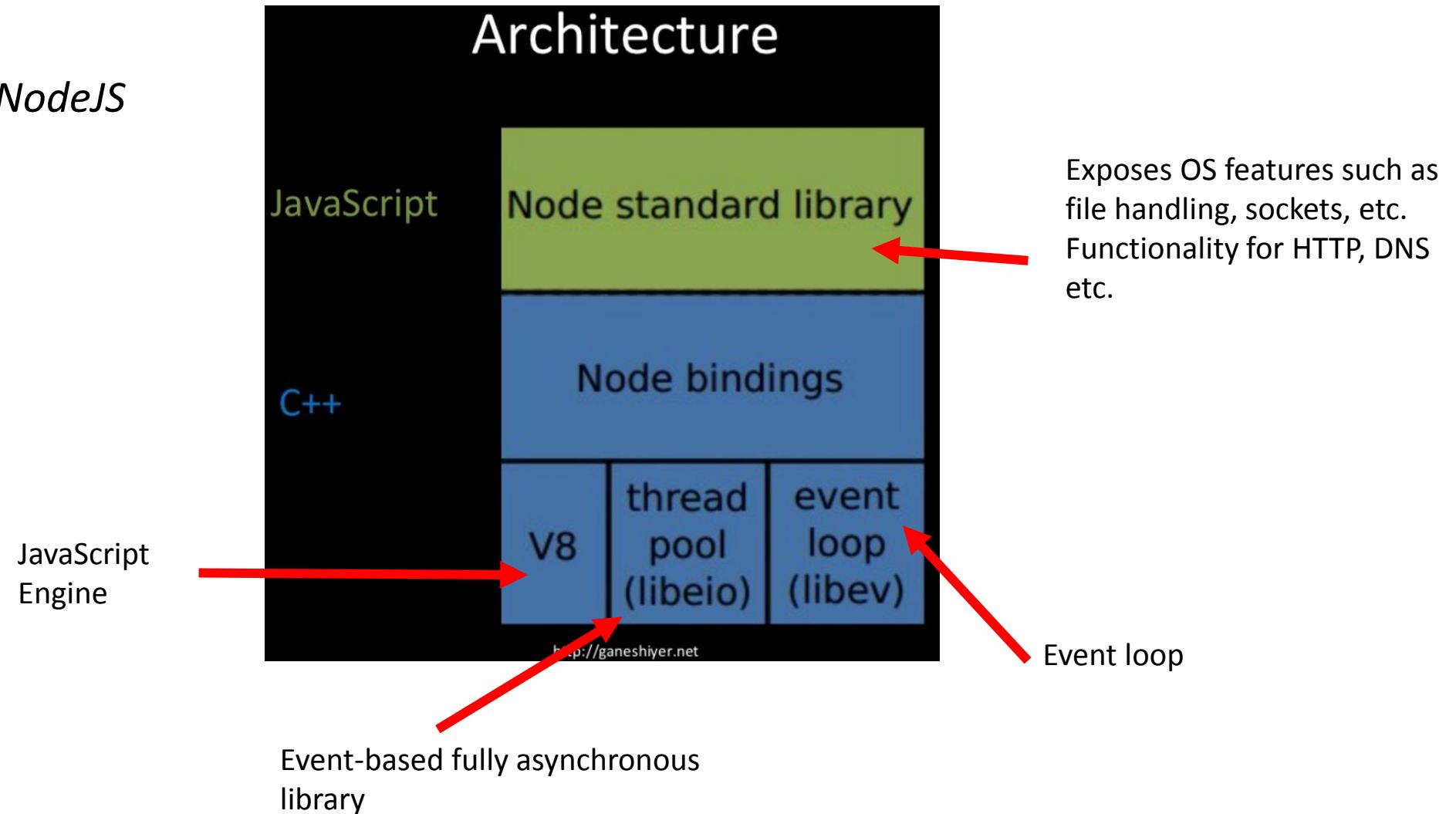
Architecture and First Applications

- Architecture – NodeJS – Scalability Issues



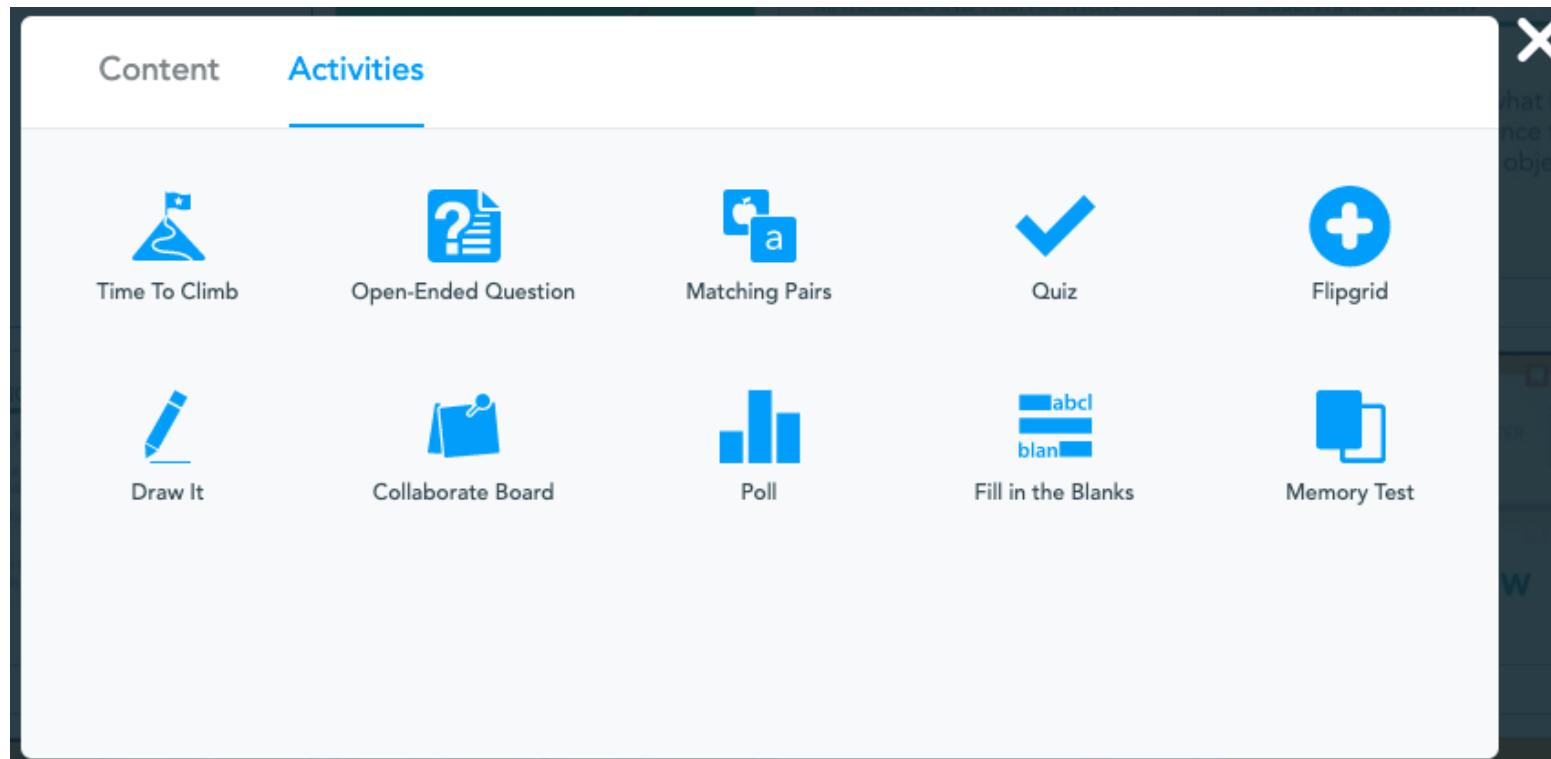
Architecture and First Applications

- **Architecture – NodeJS**



Nearpod Activity

- Please go to the Nearpod link shared in the chat.
- Fulfil the Nearpod activity.
- Analyze the results with your teacher.

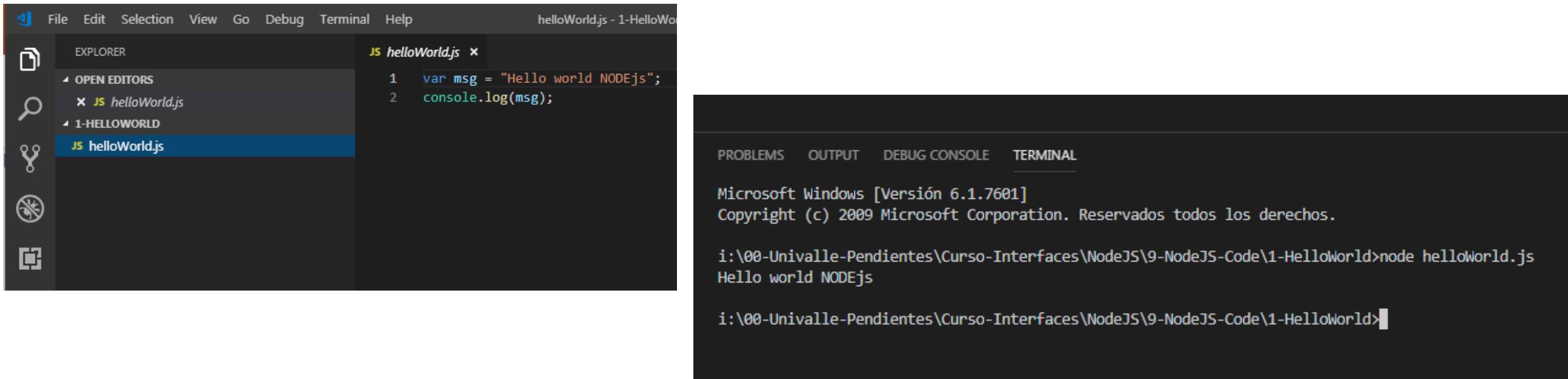


Architecture and First Applications

- **First Application – *helloWorld.js***

- Open Visual Studio Code.
- Press the *Explorer* icon.
- Press the *Open Folder* button, and selects the *1-Helloworld* folder.

- To execute it, press right button of mouse on the *helloWorld.js* file.
- Then, select *Open in Terminal* option and command line session is opened in the Workspace path.
- Then, type *node helloWorld.js*



The screenshot shows the Visual Studio Code interface. In the Explorer sidebar, a file named *helloWorld.js* is selected under the *1-HELLOWORLD* folder. The main editor window displays the following code:

```
JS helloWorld.js ×
1 var msg = "Hello world NODEjs";
2 console.log(msg);
```

Below the editor is a terminal window showing the output of running the script:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

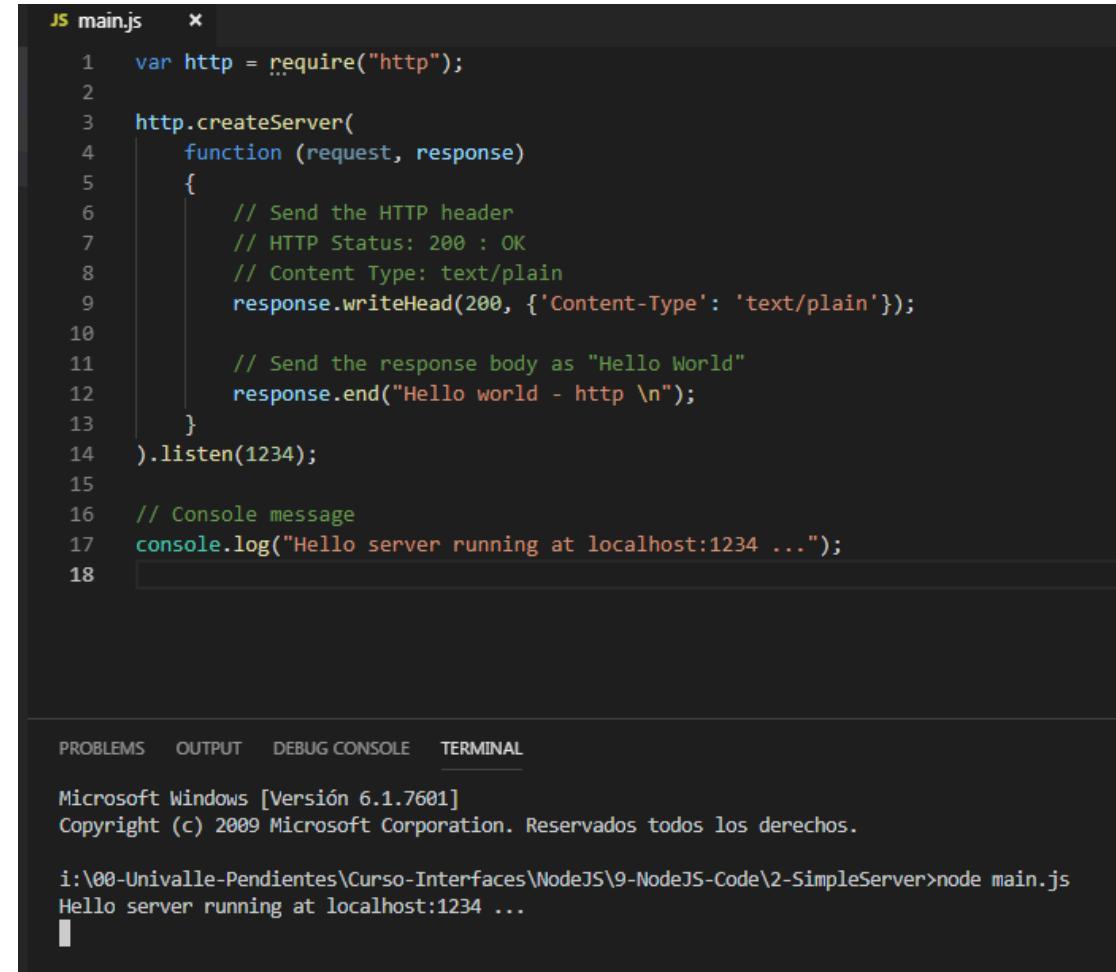
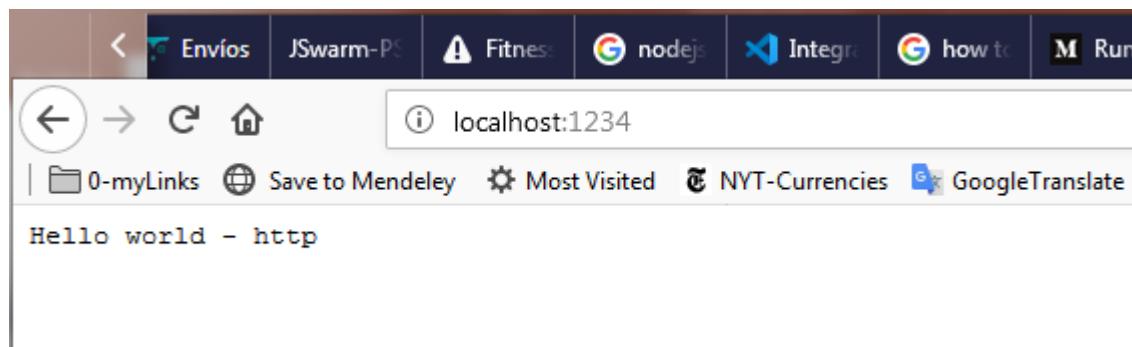
i:\00-Univalle-Pendientes\Curso-Interfaces\NodeJS\9-NodeJS-Code\1-Helloworld>node helloWorld.js
Hello world NODEjs

i:\00-Univalle-Pendientes\Curso-Interfaces\NodeJS\9-NodeJS-Code\1-Helloworld>
```

Architecture and First Applications

- **First Application – *main.js***

- Open Visual Studio Code.
- Press the *Explorer* icon.
- Press the *Open Folder* button, and selects the 2-*SimpleServer* folder.



The image shows the Visual Studio Code interface. On the left is the code editor with the file "main.js" open, containing the following Node.js code:

```
JS main.js  x
1 var http = require("http");
2
3 http.createServer(
4   function (request, response)
5   {
6     // Send the HTTP header
7     // HTTP Status: 200 : OK
8     // Content Type: text/plain
9     response.writeHead(200, {'Content-Type': 'text/plain'});
10
11    // Send the response body as "Hello World"
12    response.end("Hello world - http \n");
13  }
14 ).listen(1234);
15
16 // Console message
17 console.log("Hello server running at localhost:1234 ...");
18
```

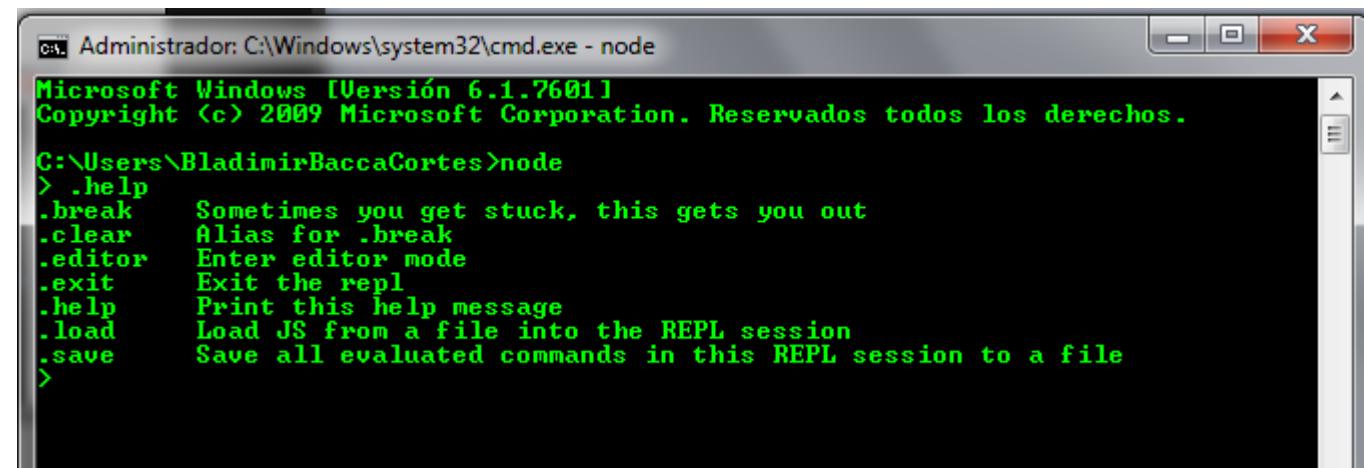
On the right is the terminal window showing the output of the command "node main.js":

```
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

i:\00-Univalle-Pendientes\Curso-Interfaces\NodeJS\9-NodeJS-Code\2-SimpleServer>node main.js
Hello server running at localhost:1234 ...
```

REPL Terminal

- **REPL** stands for Read Eval Print Loop and it represents a computer environment like a Windows console or Unix/Linux shell where a command is entered and the system responds with an output in an interactive mode.
- **Tasks:**
 - **Read** - Reads user's input, parses the input into JavaScript data-structure, and stores in memory.
 - **Eval** - Takes and evaluates the data structure.
 - **Print** - Prints the result.
 - **Loop** - Loops the above command until the user presses ctrl-c twice.
- **How to invoke it?**
 - Open a command terminal.
 - Type *node*.



A screenshot of a Windows command prompt window titled "Administrador C:\Windows\system32\cmd.exe - node". The window shows the Microsoft Windows [Versión 6.1.7601] Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos. prompt. The user has typed "node" and then ".help", which outputs the following help message:

```
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

C:\Users\BladimirBaccaCortes>node
> .help
.break    Sometimes you get stuck, this gets you out
.clear    Alias for .break
.editor   Enter editor mode
.exit    Exit the repl
.help    Print this help message
.load    Load JS from a file into the REPL session
.save    Save all evaluated commands in this REPL session to a file
>
```

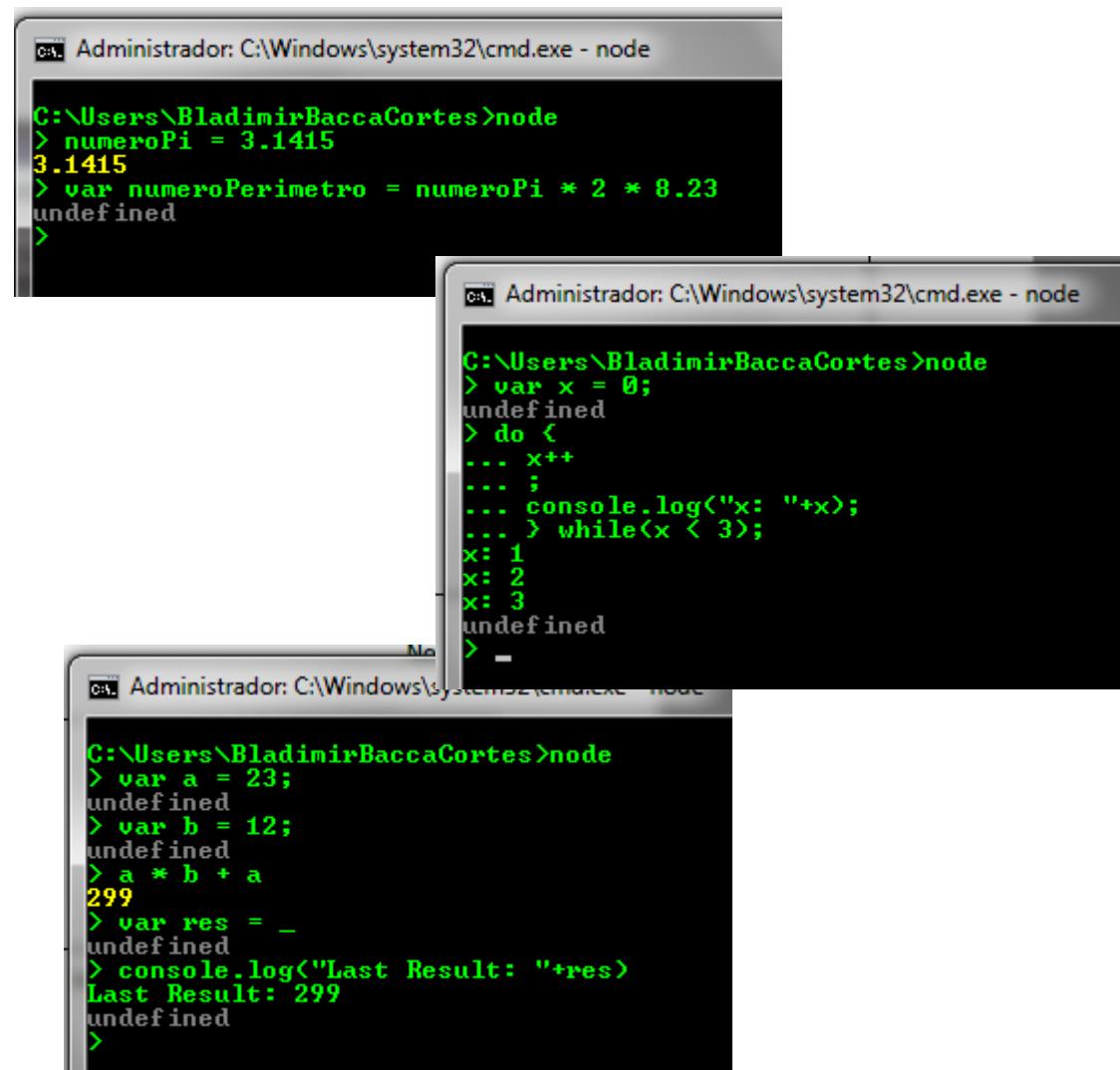
REPL Terminal

- Some examples

- **Variables:** if var keyword is used, variables are not printed.
- **Multi-line:** Node checks continuity of expressions, and then it adds '...'.
- **Last result:** it is stored in a variable named '_'

- Other features:

- ctrl + c - terminate the current command.
- ctrl + c twice - terminate the Node REPL.
- ctrl + d - terminate the Node REPL.
- Up/Down Keys - see command history and modify previous commands.
- tab Keys - list of current commands.
- .help - list of all commands.
- .break - exit from multiline expression.
- .clear - exit from multiline expression.
- .save filename - save the current Node REPL session to a file.
- .load filename - load file content in current Node REPL session



The image displays three separate Node.js REPL sessions in cmd.exe windows:

- Session 1 (Top):** Shows variable assignment and printing. It creates a variable `numeroPi` with value 3.1415, then creates `numeroPerimetro` as the product of `numeroPi` and a constant. The last result is stored in `_`.
- Session 2 (Middle):** Demonstrates a multi-line expression. It defines a loop that increments `x` from 0 to 2, logging each value to the console.
- Session 3 (Bottom):** Shows command history and saving results. It performs calculations involving `a` and `b`, stores the result in `res`, and then logs the last result back to the console.

NPM – Package Manager

• Functionalities:

- Online repositories for node.js packages/modules which are searchable on <http://search.nodejs.org>
- Command line utility to install Node.js packages, do version management and dependency management of Node.js packages.

• How to use it?

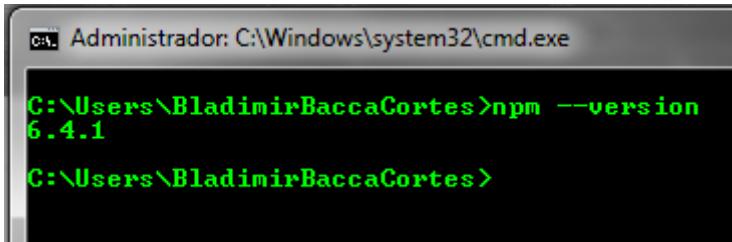
- Open a command terminal
- Type `npm --version`

• How to install modules?

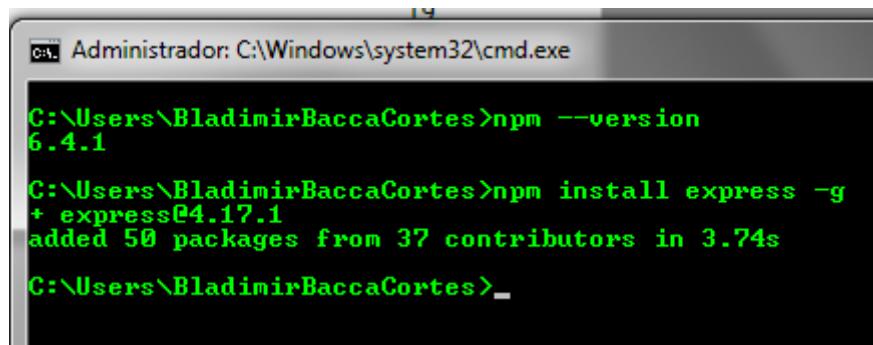
- Type: `npm install <MODULE_NAME>`
- Example: `npm install express -g`
- *Local installation* (without `-g` option): modules will be installed in the current folder solution.
 - JS files: “`var express = require('express');`”
- *Global installation* (with `-g` option): modules will be installed in system folder.

• Checking global installed modules:

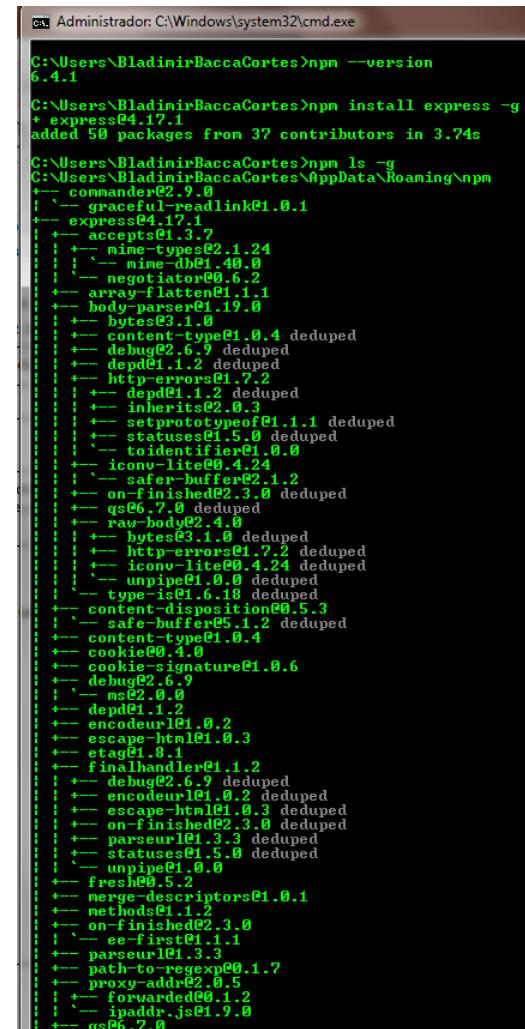
- `npm ls -g`



```
C:\ Administrador: C:\Windows\system32\cmd.exe
C:\Users\BladimirBaccaCortes>npm --version
6.4.1
C:\Users\BladimirBaccaCortes>
```



```
C:\ Administrador: C:\Windows\system32\cmd.exe
C:\Users\BladimirBaccaCortes>npm --version
6.4.1
C:\Users\BladimirBaccaCortes>npm install express -g
+ express@4.17.1
added 50 packages from 37 contributors in 3.74s
C:\Users\BladimirBaccaCortes>
```



```
C:\ Administrador: C:\Windows\system32\cmd.exe
C:\Users\BladimirBaccaCortes>npm --version
6.4.1
C:\Users\BladimirBaccaCortes>npm install express -g
+ express@4.17.1
added 50 packages from 37 contributors in 3.74s
C:\Users\BladimirBaccaCortes>npm ls -g
C:\Users\BladimirBaccaCortes>npm ls -g
C:\Users\BladimirBaccaCortes>npm ls -g
+-- commander@2.9.0
  +-- graceful-readlink@1.0.1
  +-- express@4.17.1
    +-- accept@1.3.7
      +-- mime-types@2.1.24
        +-- mime-db@1.40.0
        +-- negotiator@0.6.2
    +-- array-flatten@1.1.1
    +-- body-parser@1.19.0
      +-- bytes@3.1.0
      +-- content-type@1.0.4 deduped
      +-- debug@2.6.9 deduped
      +-- depd@1.1.2 deduped
      +-- http-errors@1.7.2
      +-- depd@1.1.2 deduped
      +-- inherits@2.0.3
      +-- setprototypeof@1.1.1 deduped
      +-- statuses@1.5.0 deduped
      +-- toidentifier@1.0.0
    +-- iconv-lite@0.4.24
    +-- safer-buffer@2.1.2
    +-- on-finished@2.3.0 deduped
    +-- qs@6.7.0 deduped
    +-- raw-body@2.4.0
      +-- bytes@3.1.0 deduped
      +-- http-errors@1.7.2 deduped
      +-- iconv-lite@0.4.24 deduped
      +-- unpipe@1.0.0 deduped
      +-- type-is@1.6.18 deduped
    +-- content-disposition@0.5.3
    +-- safe-buffer@5.1.2 deduped
    +-- content-type@1.0.4
    +-- cookie@0.4.0
    +-- cookie-signature@1.0.6
    +-- debug@2.6.9
    +-- ms@2.0.0
    +-- depd@1.1.2
    +-- encodeurl@1.0.2
    +-- escape-html@1.0.3
    +-- etag@2.9.1
    +-- finalhandler@1.1.2
      +-- debug@2.6.9 deduped
      +-- encodeurl@1.0.2 deduped
      +-- escape-html@1.0.3 deduped
      +-- on-finished@2.3.0 deduped
      +-- parseurl@1.3.3 deduped
      +-- statuses@1.5.0 deduped
      +-- unpipe@1.0.0
    +-- fresh@0.5.2
    +-- merge-descriptors@1.0.1
    +-- methods@1.1.2
    +-- on-finished@2.3.0
      +-- ee-first@1.1.1
    +-- parseurl@1.3.3
    +-- path-to-regexp@0.1.7
    +-- proxy-addr@2.0.5
      +-- forwarded@2.0.1.2
        +-- ipaddr.js@1.9.0
    +-- qs@6.7.0
```

NPM – Package Manager

- Other commands:

- Updating modules:
 - Type: *npm update <MODULE_NAME>*
- Uninstalling modules:
 - Type: *npm uninstall <MODULE_NAME>*
- Search for a module:
 - Type: *npm search <MODULE_NAME>*

```
Administrator: C:\Windows\system32\cmd.exe
C:\Users\BladimirBaccaCortes>npm update express
C:\Users\BladimirBaccaCortes>
```

```
Administrator: C:\Windows\system32\cmd.exe
C:\Users\BladimirBaccaCortes>npm search serial
          NAME      | DESCRIPTION           | AUTHOR        | DATE
serial       | Serial and parallel. | =nherment.    | 2016-02-21
js-yaml      | YAML 1.2 parser and. | =vitaly       | 2019-04-05
parse5        | HTML parser and.    | =inikulin     | 2018-08-13
serialize-javascript | Serialize.          | =ericf =okuryu | 2019-04-16
serialize-error   | Serialize an error. | =sindresorhus | 2019-04-02
p-each-series  | Iterate over.       | =sindresorhus | 2019-04-04
emotion/serialize | serialization utils. | =mitchellhamil | 2019-06-09
serialport     | Node.js package to. | =jjrosent.    | 2019-04-27
dom-serialize   | Serializes any DOM. | =tootallnate  | 2015-11-05
slate-plain-serializer | A plain text.       | =ianstormtaylor | 2019-06-07
p-map-series    | Map over promises. | =sindresorhus | 2019-04-03
smart-buffer     | smart-buffer is a. | =joshglazebrook | 2019-01-24
slate-html-serializer | An HTML serializer. | =ianstormtaylor | 2019-06-07
ssri           | Standard.          | =iarna =zkat   | 2018-08-27
p-pipe          | Compose.            | =sindresorhus | 2019-04-06
slate-base64-serializer | A Base64 serializer. | =ianstormtaylor | 2019-06-07
fast-safe-stringify | Safely and quickly. | =bridgear.    | 2018-08-15
cookie          | HTTP server cookie. | =dougwilson   | 2019-05-16
protobufjs       | Protocol Buffers.  | =dcode         | 2018-07-19
modbus-serial    | A pure JavaScript. | =kzamir       | 2019-05-31
C:\Users\BladimirBaccaCortes>
```

Variables in NodeJS

- Variables are defined in JavaScript using the `var` keyword
- They support all types of variables without specifying it on the declaration.
- **They support**
 - Numbers
 - Arrays
 - Booleans
 - Object literals
 - Functions.
- Open folder: **3-BasicNodeJS**
- Observe and run: **checkVariables.js**

```
JS checkVariables.js ×
JS checkVariables.js ▶ ...
1 // Numbers
2 console.log("Operations with numbers: ");
3 var a = 23.34;
4 var b = 0.234;
5 console.log("Sum: "+(a + b));
6 console.log("Mult.: "+(a * b));
7 console.log("Division: "+(a / (b*10)));
8 console.log("*****");
9
10 // Booleans
11 console.log("Operations with booleans: ");
12 var x = true;
13 var y = false;
14 console.log("x * y: "+(x && y));
15 console.log("x + y: "+(x || y));
16 console.log("Not x: "+(!x));
17 console.log("*****");
18
19 // Arrays
20 console.log("Arrays: ");
21 var r = [1, 2.3, 4.3];
22 console.log("r: "+r+"\n");
23
24 r.push(5.3);
25 console.log("Pushing ... r: "+r+"\n");
26
27 var lastR = r.pop();
28 console.log("Pop ... r: "+r);
29 console.log("Last R: "+lastR+"\n");
30
31 r.unshift(0.2);
32 console.log("Unshifting ... r: "+r+"\n");
33
34 var firstR = r.shift();
35 console.log("Shifting ... r: "+r);
36 console.log("First R: "+firstR+"\n");
37
38 var delItem = r.splice(2, 1);
39 console.log("Splice ... r: "+r+"\n");
```

Functions in NodeJs

- **Syntax:**

```
function functionName (parameter_list)
```

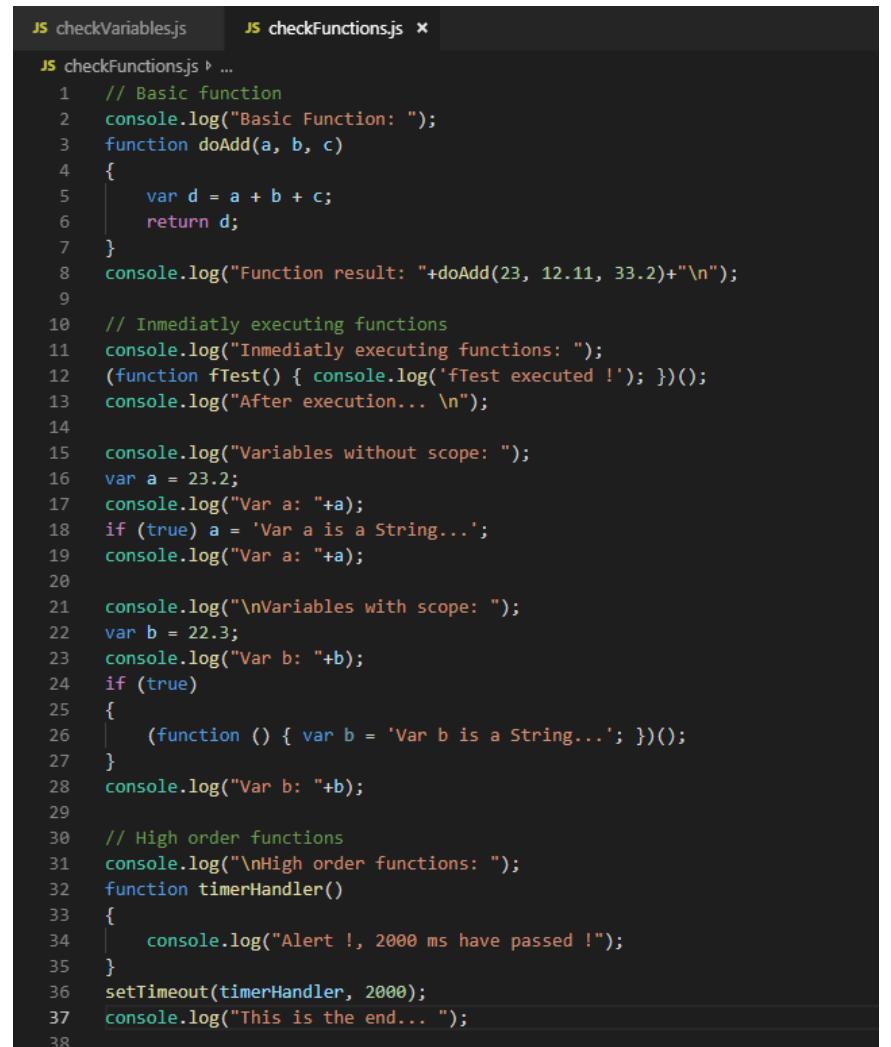
```
{
```

```
    // Function body
```

```
    return expression; // optional
```

```
}
```

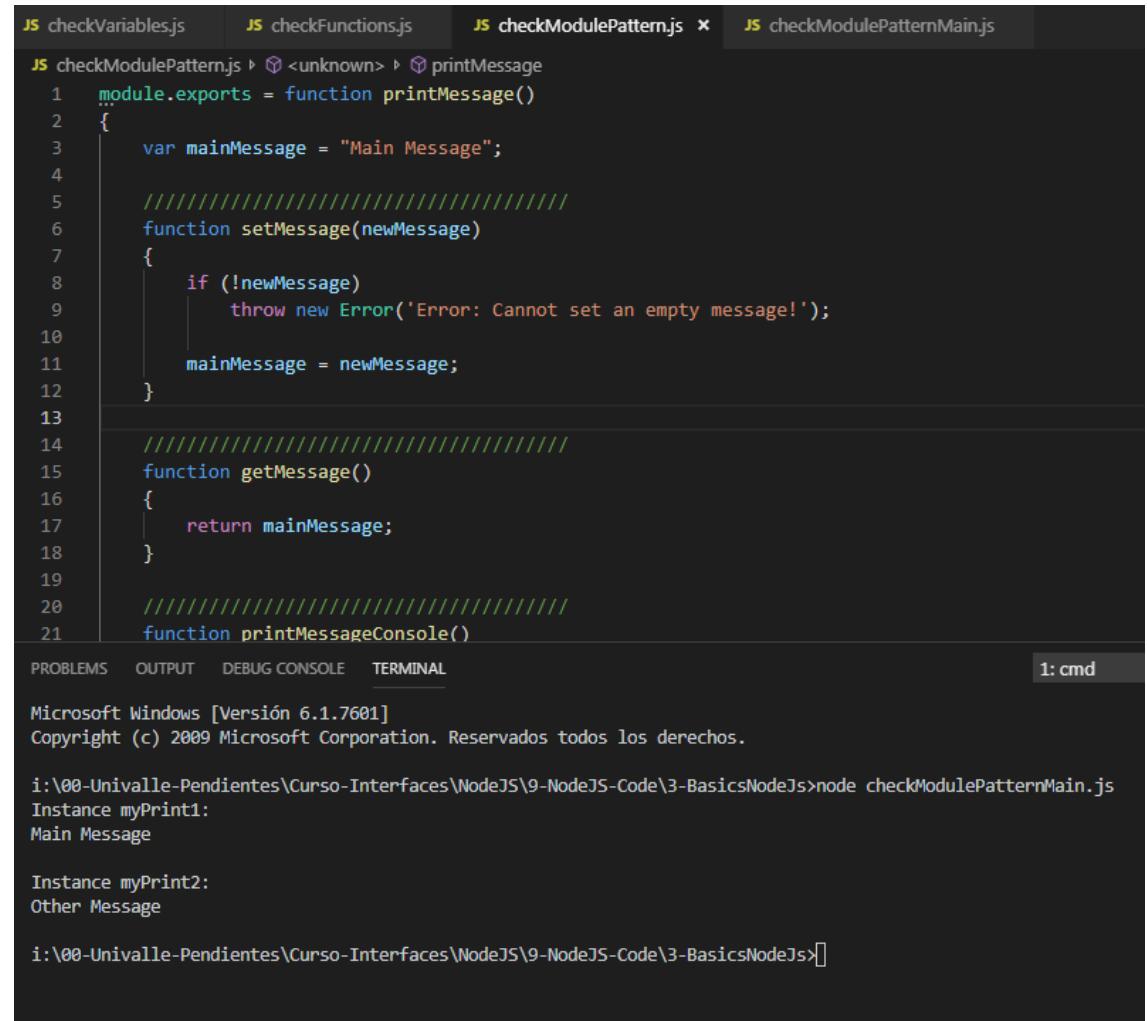
- Most of the power of JavaScript comes from the way it handles the function type.
- All functions return a value in JavaScript. In the absence of an explicit return statement, a function returns undefined.
- **Immediately executing functions:** The reason for having an immediately executing function is to create a new variable scope. An if, else, or while does not create a new variable scope in JavaScript.
- **Anonymous functions:** A function without a name is called an anonymous function. In JavaScript, you can assign a function to a variable. If you are going to use a function as a variable, you don't need to name the function
- **High-order functions:** Functions that take functions as arguments are called higher-order functions.
- Run **checkFunctions.js**, in the **3-BasicsNodeJS** folder.



```
JS checkVariables.js JS checkFunctions.js ×  
JS checkFunctions.js > ...  
1 // Basic function  
2 console.log("Basic Function: ");  
3 function doAdd(a, b, c)  
4 {  
5     var d = a + b + c;  
6     return d;  
7 }  
8 console.log("Function result: "+doAdd(23, 12.11, 33.2)+"\n");  
9  
10 // Immediately executing functions  
11 console.log("Immediately executing functions: ");  
12 (function fTest() { console.log('fTest executed !'); })();  
13 console.log("After execution... \n");  
14  
15 console.log("Variables without scope: ");  
16 var a = 23.2;  
17 console.log("Var a: "+a);  
18 if (true) a = 'Var a is a String...';  
19 console.log("Var a: "+a);  
20  
21 console.log("\nVariables with scope: ");  
22 var b = 22.3;  
23 console.log("Var b: "+b);  
24 if (true)  
25 {  
26     (function () { var b = 'Var b is a String...'; })();  
27 }  
28 console.log("Var b: "+b);  
29  
30 // High order functions  
31 console.log("\nHigh order functions: ");  
32 function timerHandler()  
33 {  
34     console.log("Alert !, 2000 ms have passed !");  
35 }  
36 setTimeout(timerHandler, 2000);  
37 console.log("This is the end... ");  
38
```

Module Pattern and Exports

- Functions that return objects are a great way to create similar objects.
- An object here means data and functionality bundled into a nice package, which is the most basic form of Object Oriented Programming (OOP) that one can do.
- At the heart of the revealing module pattern is JavaScript's support for closures and ability to return arbitrary (function + data) object literals.
- **Module system properties:**
 - Each file is its own module.
 - Each file has access to the current module definition using the module variable.
 - The export of the current module is determined by the module.exports variable.
 - To import a module, use the globally available require function
- Open and run **checkModulePattern.js**



The screenshot shows a code editor with several tabs open, all labeled 'JS'. The active tab contains the following code:

```
JS checkVariables.js      JS checkFunctions.js      JS checkModulePattern.js ×      JS checkModulePatternMain.js
JS checkModulePattern.js ↗ <unknown> ↗ printMessage
1  module.exports = function printMessage()
2  {
3      var mainMessage = "Main Message";
4
5      ///////////////////////////////////////////////////
6      function setMessage(newMessage)
7      {
8          if (!newMessage)
9              throw new Error('Error: Cannot set an empty message!');
10
11         mainMessage = newMessage;
12     }
13
14     ///////////////////////////////////////////////////
15     function getMessage()
16     {
17         return mainMessage;
18     }
19
20     ///////////////////////////////////////////////////
21     function printMessageConsole()
```

Below the code editor is a terminal window showing the execution of the script:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1: cmd
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

i:\00-Univalle-Pendientes\Curso-Interfaces\NodeJS\9-NodeJS-Code\3-BasicsNodeJs>node checkModulePatternMain.js
Instance myPrint1:
Main Message

Instance myPrint2:
Other Message

i:\00-Univalle-Pendientes\Curso-Interfaces\NodeJS\9-NodeJS-Code\3-BasicsNodeJs>[]
```

Callbacks and Events

- **Definition of Callback:** It is an [asynchronous](#) equivalent for a [function](#). A callback function is [called](#) at the [completion](#) of a given [task](#). Node makes heavy use of callbacks. All the APIs of Node are written in such a way that they support callbacks.
- **For example:**
 - A [function](#) to [read a file](#) may start reading a file and [return the control](#) to the [execution environment immediately](#) so that the next instruction can be executed.
 - Once file [I/O](#) is [complete](#), it will [call the callback function](#) while passing the [callback function](#), the content of the file as a parameter.
 - So there is [no blocking](#) or wait for [File I/O](#).
 - This makes [Node.js highly scalable](#), as it can process a high number of requests without waiting for any function to return results.
- Open folder **4-CaLLBaCKs**, and run **fsBlocking.js** and **fsNoBlocking.js**.

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

i:\00-Univalle-Pendientes\Curso-Interfaces\NodeJS\9-NodeJS-Code\4-CallBacks>node fsBlocking.js
Start reading...
Interfaces Course.

NodeJS class Introduction.

Nice things are easy!!

End...

i:\00-Univalle-Pendientes\Curso-Interfaces\NodeJS\9-NodeJS-Code\4-CallBacks>
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
i:\00-Univalle-Pendientes\Curso-Interfaces\NodeJS\9-NodeJS-Code\4-CallBacks>node fsNoBlocking.js
Start reading...
End ...
Interfaces Course.

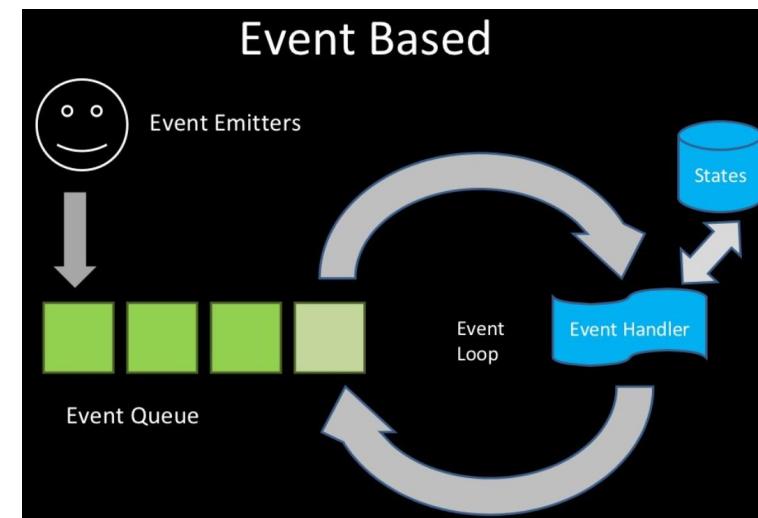
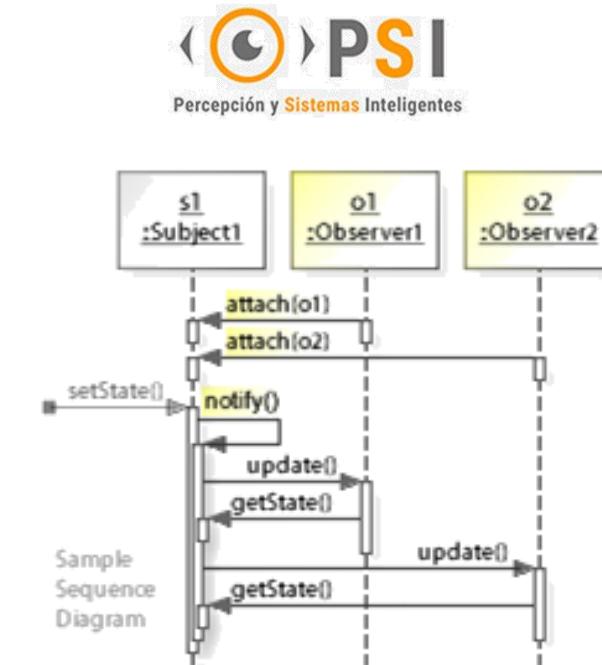
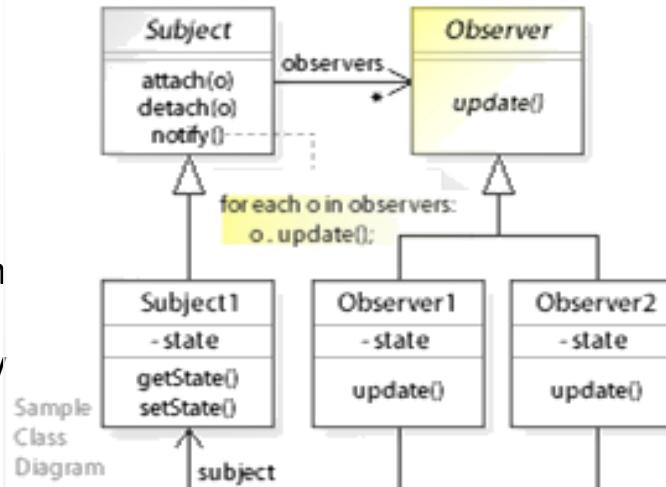
NodeJS class Introduction.

Nice things are easy!!

i:\00-Univalle-Pendientes\Curso-Interfaces\NodeJS\9-NodeJS-Code\4-CallBacks>
```

Callbacks and Events

- Node.js is a single-threaded application, but it can support concurrency via the concept of **event** and **callbacks**.
- Every API of Node.js is asynchronous and being single-threaded, they use **async function calls** to maintain concurrency.
- **Node uses observer pattern:** It is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.
- As soon as **Node starts its server**, it simply **initiates its variables**, declares **functions**, and then simply **waits for the event** to occur.
- In an **event-driven** application, there is generally a **main loop** that **listens for events**, and then **triggers a callback** function when one of those events is detected.
- Although events look quite similar to callbacks, the **difference** lies in the fact that **callback** functions are **called** when an **asynchronous** function returns its result, **whereas event handling works** on the **observer pattern**. The functions that listen to events act as Observers



Callbacks and Events

- **Steps to implement events:**

1. Load events module: `events`.
 2. Instance emitter event class: `EventEmitter()`
 3. Attaching event handler with events.
 4. Firing events.

- Open folder **5-Events**, then run **testEvents.js**.
 - All objects which emit events are the instances of *events.EventEmitter*.
 - When an **EventEmitter** instance faces any **error**, it emits an '**error**' event.
 - When a new **listener** is **added**, '**newListener**' event is fired
 - When a **listener** is **removed**, '**removeListener**' event is fired.

JS testEvents.js ×

```
JS testEvents.js ↵ ...
1 // Import module of events
2 var events = require('events');
3
4 // Create event emitter
5 var myEventEmitter = new events.EventEmitter();
6
7 // Create event handler
8 var myConnectHandler = function connectHandler() {
9     console.log('Ok - Connection succesful !');
10
11     // Fire data received by event.
12     myEventEmitter.emit('Data_Received');
13 }
14
15 // Attaching the connection event with the handler
16 myEventEmitter.on('connection', myConnectHandler);
17
18 // Attaching Data_received event with an anonymous function
19 myEventEmitter.on('Data_Received', function(){
20     console.log('Data received successfully !');
21 });

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
```

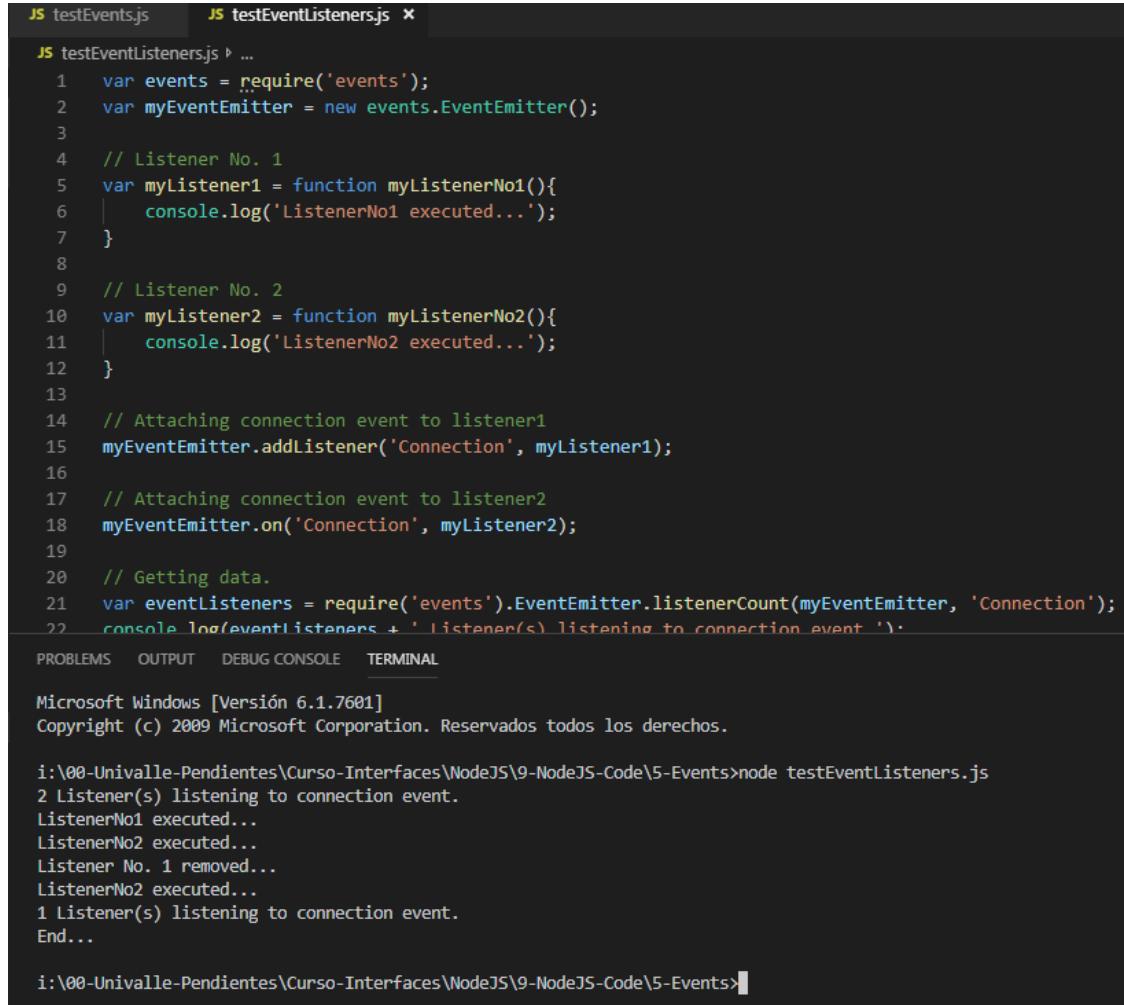
i:\00-Univalle-Pendientes\Curso-Interfaces\NodeJS\9-NodeJS-Code\5-Events>node testEvents.js
Ok - Connection succesful !
Data received successfully !
End...

i:\00-Univalle-Pendientes\Curso-Interfaces\NodeJS\9-NodeJS-Code\5-Events>]

Callbacks and Events

- **Event emitter methods:**

- *addListener(event, listener)*: Adds a listener at the end of the listeners array for the specified event
- *on(event, listener)*: Adds a listener at the end of the listeners array for the specified event. But it cannot be removed.
- *once(event, listener)*: Adds a one-time listener to the event.
- *removeListener(event, listener)*: Removes a listener from the listener array for the specified event.
- *removeAllListeners([event])*: Removes all listeners, or those of the specified event.
- *setMaxListeners(n)* By default, EventEmitters will print a warning if more than 10 listeners are added for a particular event.
- *listeners(event)* Returns an array of listeners for the specified event.



The screenshot shows a code editor with two tabs: `testEvents.js` and `testEventListeners.js`. The `testEventListeners.js` tab is active and contains the following code:

```
JS testEvents.js JS testEventListeners.js ×
JS testEventListeners.js ▾ ...
1 var events = require('events');
2 var myEventEmitter = new events.EventEmitter();
3
4 // Listener No. 1
5 var myListener1 = function myListenerNo1(){
6   console.log('ListenerNo1 executed...');
7 }
8
9 // Listener No. 2
10 var myListener2 = function myListenerNo2(){
11   console.log('ListenerNo2 executed...');
12 }
13
14 // Attaching connection event to listener1
15 myEventEmitter.addListener('Connection', myListener1);
16
17 // Attaching connection event to listener2
18 myEventEmitter.on('Connection', myListener2);
19
20 // Getting data.
21 var eventListeners = require('events').EventEmitter.listenerCount(myEventEmitter, 'Connection');
22 console.log(eventListeners + ' Listener(s) listening to connection event');

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

i:\00-Univalle-Pendientes\Curso-Interfaces\NodeJS\9-NodeJS-Code\5-Events>node testEventListeners.js
2 Listener(s) listening to connection event.
ListenerNo1 executed...
ListenerNo2 executed...
Listener No. 1 removed...
ListenerNo2 executed...
1 Listener(s) listening to connection event.
End...

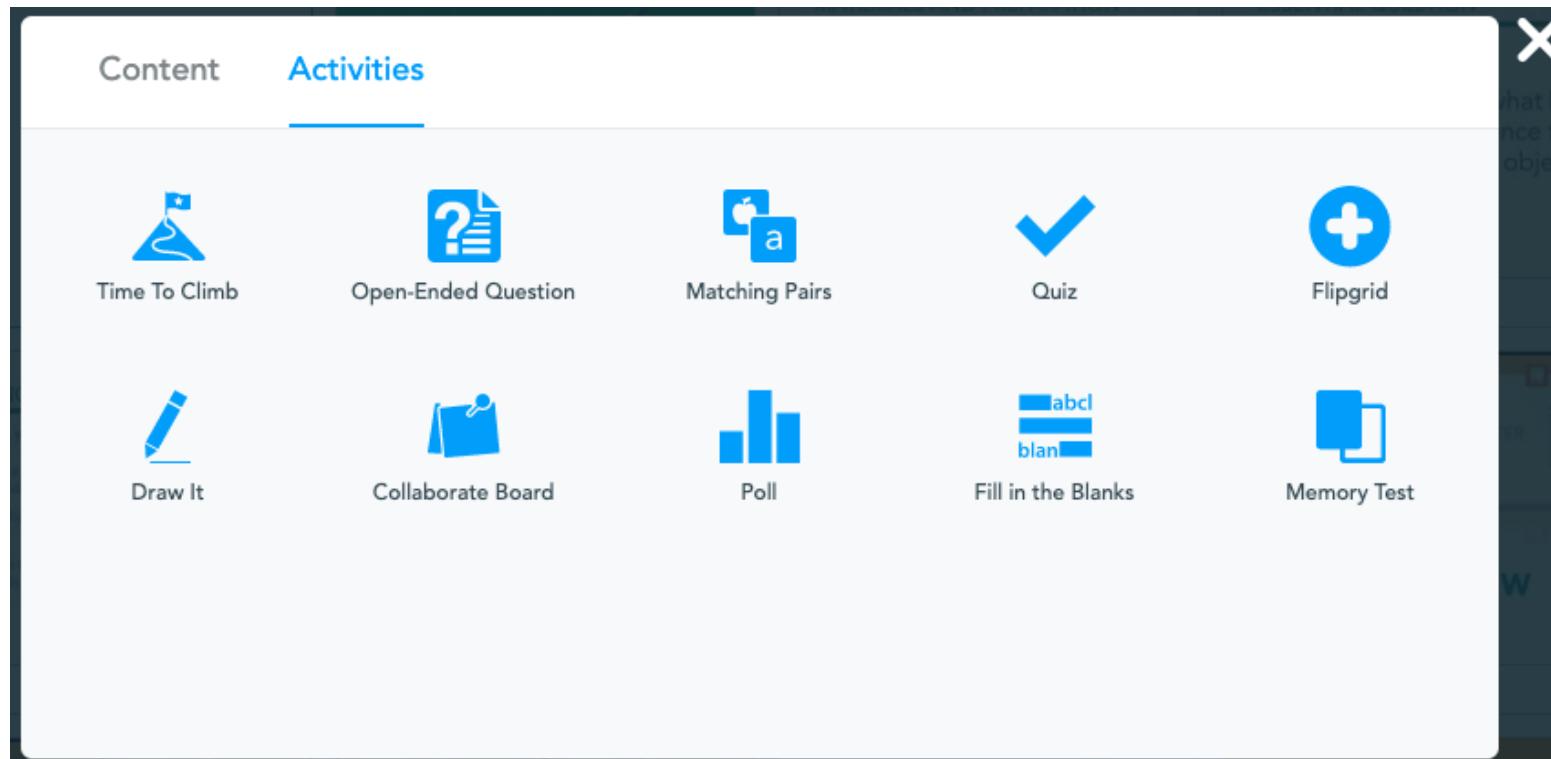
i:\00-Univalle-Pendientes\Curso-Interfaces\NodeJS\9-NodeJS-Code\5-Events>
```

NodeJS Modules

- Core modules
- Assertion Testing
- File System
- Buffer
- HTTP/HTTPS
- C/C++ Addons
- Net
- Child Processes
- OS
- Cluster
- Path
- Crypto
- Process
- Debugger
- Punycode
- DNS
- Query Strings
- Domain
- REPL
- Events
- Stream
- String Decoder
- Timers
- TLS/SSL
- TTY
- UDP/Datagram
- URL
- Utilities
- VM
- ZLIB

Nearpod Activity

- Please go to the Nearpod link shared in the chat.
- Fulfil the Nearpod activity.
- Analyze the results with your teacher.



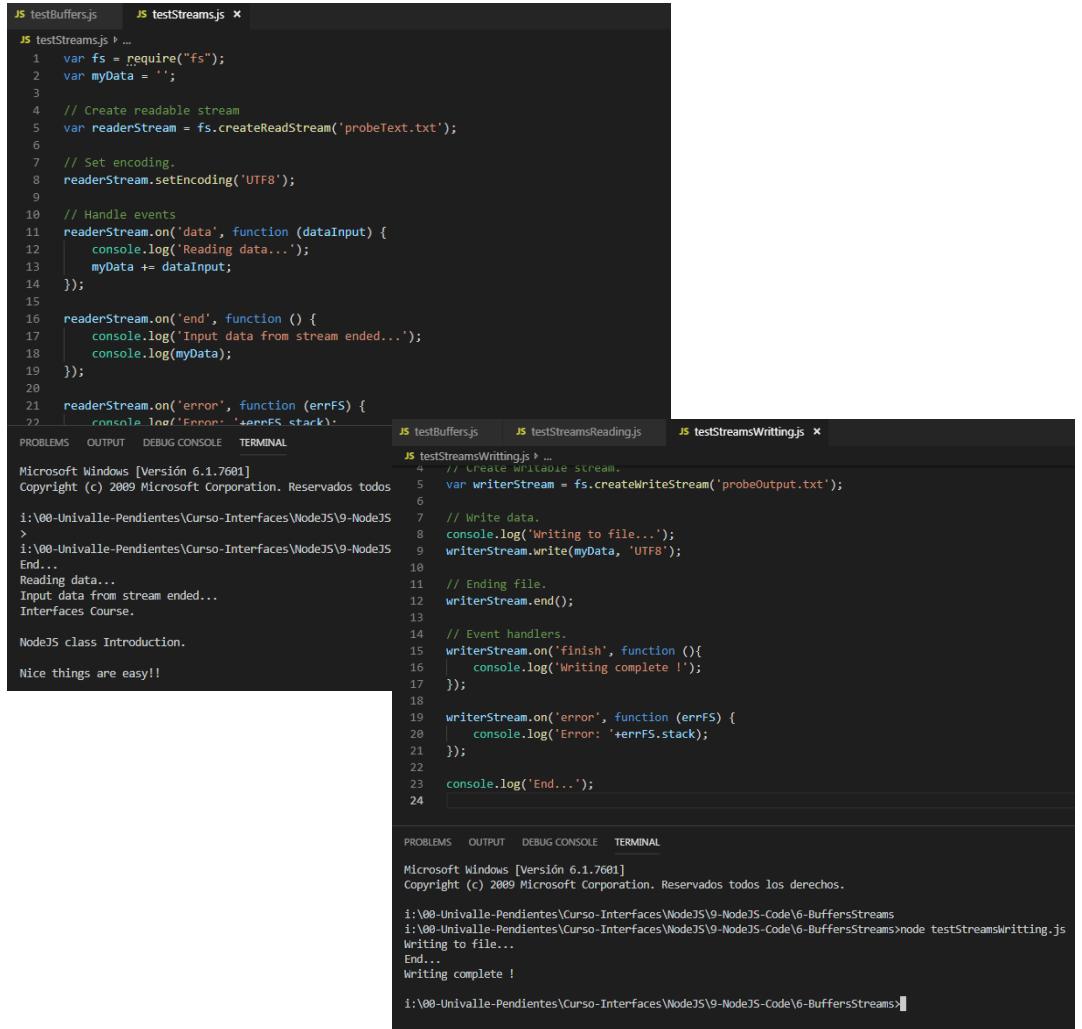
Buffers and Streams

- **Buffers:**
- While dealing with [TCP streams](#) or the file system, it's necessary to handle [octet streams](#).
- Node provides [Buffer](#) class which provides instances to [store raw data](#) similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap
- [Buffer class](#) is a [global class](#) that can be accessed in an application without importing the buffer module.
- Open **6-BuffersStreams** folder, and run **testBuffers.js**.

```
JS testBuffers.js ×
JS testBuffers.js ▶ ...
1 // Creating buffers
2 console.log('Creating buffers...');
3 var buf1 = new Buffer(100);
4 var buf2 = new Buffer([10, 34.2, 23, 45]);
5 var buf3 = new Buffer("Simple things are nice !!!", "utf-8");
6
7 // Writing...
8 console.log('\nWritting...');
9 var lenBuffer = buf1.write('NodeJS Class ...');
10 console.log('Buffer size: '+lenBuffer);
11 console.log('Buffer: '+buf1.toString('utf-8'));
12
13 // Converting to json
14 console.log('\nTo Json: ');
15 var bufJson = buf2.toJSON(buf2);
16 console.log('Buf2 to JSON: '+bufJson);
17
18 // Comparing...
19 console.log('\nComparing...');
20 var res = buf3.compare(buf1);
21 if (res == 0) {
22 |   console.log('Buffers are the same');
23 }
```

Buffers and Streams

- **Streams:**
- Streams are objects that let you read data from a source or write data to a destination in continuous fashion. In Node.js, there are four types of streams:
 - **Readable** – It is used for read operation.
 - **Writable** – It is used for write operation.
 - **Duplex** – It can be used for both read/write operation.
 - **Transform** – A type of duplex stream where the output is computed based on input.
- Each type of Stream is an **EventEmitter** instance and throws several events at different instance of times:
 - **data** – It is fired when there is data is available to read.
 - **end** – It is fired when there is no more data to read.
 - **error** – It is fired when there is any error receiving or writing data.
 - **finish** – It is fired when all the data has been flushed to underlying system.
- Open **6-BuffersStreams** folder, and run **testStreamsReading.js** and **testStreamsWriting.js**.



```
JS testBuffers.js JS testStreams.js x
JS testStreams.js > ...
1 var fs = require("fs");
2 var myData = '';
3
4 // Create readable stream
5 var readerStream = fs.createReadStream('probeText.txt');
6
7 // Set encoding.
8 readerStream.setEncoding('UTF8');
9
10 // Handle events
11 readerStream.on('data', function (dataInput) {
12   console.log("Reading data...");
13   myData += dataInput;
14 });
15
16 readerStream.on('end', function () {
17   console.log("Input data from stream ended...");
18   console.log(myData);
19 });
20
21 readerStream.on('error', function (errFS) {
22   console.log('Error: '+errFS.stack);
23 });

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.
i:\00-Univalle-Pendientes\Curso-Interfaces\NodeJS\9-NodeJS
> i:\00-Univalle-Pendientes\Curso-Interfaces\NodeJS\9-NodeJS
End...
Reading data...
Input data from stream ended...
Interfaces Course.

NodeJS class Introduction.

Nice things are easy!!
```



```
JS testBuffers.js JS testStreamsReading.js JS testStreamsWriting.js x
JS testStreamsWriting.js > ...
4 // Create writable stream.
5 var writerStream = fs.createWriteStream('probeOutput.txt');
6
7 // Write data.
8 console.log("Writing to file...");
9 writerStream.write(myData, 'UTF8');
10
11 // Ending file.
12 writerStream.end();
13
14 // Event handlers.
15 writerStream.on('finish', function (){
16   console.log('Writing complete !');
17 });
18
19 writerStream.on('error', function (errFS) {
20   console.log('Error: '+errFS.stack);
21 });
22
23 console.log('End...');

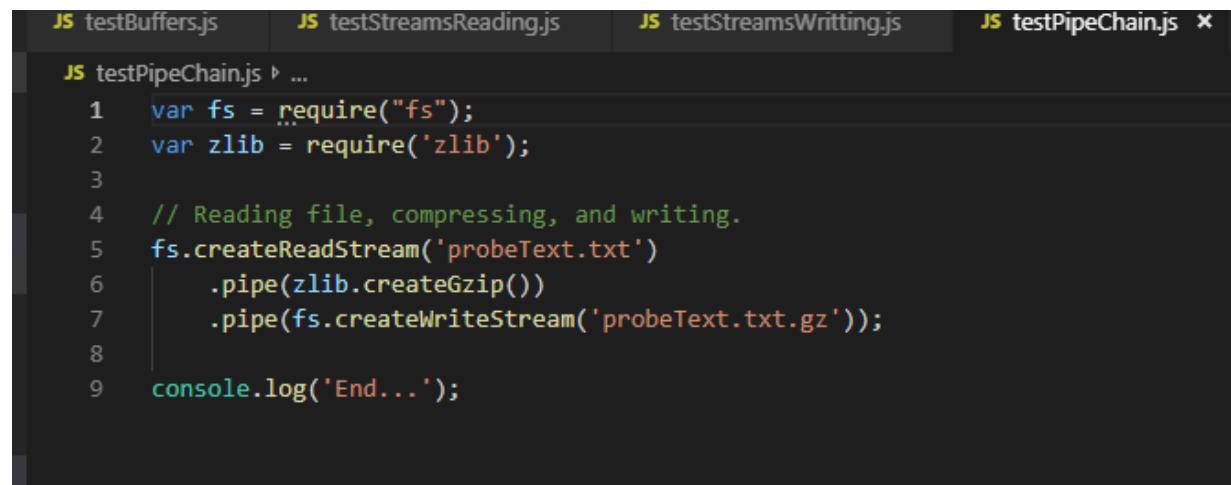
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.
i:\00-Univalle-Pendientes\Curso-Interfaces\NodeJS\9-NodeJS-Code\6-BuffersStreams
i:\00-Univalle-Pendientes\Curso-Interfaces\NodeJS\9-NodeJS-Code\6-BuffersStreams>node testStreamsWriting.js
Writing to file...
End...
Writing complete !

i:\00-Univalle-Pendientes\Curso-Interfaces\NodeJS\9-NodeJS-Code\6-BuffersStreams>
```

Buffers and Streams

- **Piping and Channing:**

- Piping is a mechanism where we provide the output of one stream as the input to another stream.
- It is normally used to get data from one stream and to pass the output of that stream to another stream.
- There is no limit on piping operations
- Chaining is a mechanism to connect the output of one stream to another stream and create a chain of multiple stream operations.
- It is normally used with piping operations
- Open **6-BufferStream** folder, and run **testPipeChain.js**.



```
JS testBuffers.js JS testStreamsReading.js JS testStreamsWriting.js JS testPipeChain.js ×  
JS testPipeChain.js ↗ ...  
1 var fs = require("fs");  
2 var zlib = require('zlib');  
3  
4 // Reading file, compressing, and writing.  
5 fs.createReadStream('probeText.txt')  
6   .pipe(zlib.createGzip())  
7   .pipe(fs.createWriteStream('probeText.txt.gz'));  
8  
9 console.log('End...');
```

File System

- Every method in the fs module has **synchronous** as well as **asynchronous** forms.
- **Asynchronous** methods take the last parameter as the completion **function callback** and the first parameter of the callback function as error

- **Methods:**

- *Open files*: fs.open(path, flags[, mode], callback)
- *File info*: fs.stat(path, callback)
- *Writing*: fs.writeFile(filename, data[, options], callback)
- *Reading*: fs.read(fd, buffer, offset, length, position, callback)
- *Closing*: fs.close(fd, callback)
- *Create directory*: fs.mkdir(path[, mode], callback)
- *Read directory*: fs.readdir(path, callback)
- Open folder **7-FileSystem**, and run **testFS.js**.



The screenshot shows a code editor window with a dark theme. The file `testFS.js` contains the following code:

```

JS testFS.js  x
JS testFS.js ↵  ↵ fs.readdir(".") callback
43   {
44     |
45     }
46   }
47   }
48   });
49 });
50
51 // Reading directory
52 console.log('\nReading directory ...');
53 fs.readdir(".", function (err, dirFiles) {
54   if (err) [
55     return console.log('Error: '+err);
56   ]
57
58   dirFiles.forEach(function (myFile){
59     console.log(myFile);
60   });
61 });

```

Below the code editor is a terminal window with the following output:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1-HelloWorld
2-SimpleServer
3-BasicsNodeJs
4-CaLLbacks
5-Events
6-BuffersStreams
7-FileSystem
Number of bytes read: 78
Data read: Interfaces Course.

NodeJS class Introduction.

Nice things are easy!!

File closed...

i:\00-Univalle-Pendientes\Curso-Interfaces\NodeJS\9-NodeJS-Code\7-FileSystem>

```

File System

- **Read and write flags (open):**

- r - Open file for reading.
- r+ - Open file for reading and writing
- rs - Open file for reading in synchronous mode
- rs+ - Open file for reading and writing, asking the OS to open it synchronously.
- w - Open file for writing.
- wx - Like 'w' but fails if the path exists
- w+ - Open file for reading and writing.
- wx+ - Like 'w+' but fails if path exists
- a - Open file for appending
- ax - Like 'a' but fails if the path exists.
- a+ - Open file for reading and appending
- ax+ - Like 'a+' but fails if the path exists

- **File information:**

- *stats.isFile()* - Returns true if file type of a simple file.
- *stats.isDirectory()* - Returns true if file type of a directory.
- *stats.isBlockDevice()* - Returns true if file type of a block device.
- *stats.isCharacterDevice()* - Returns true if file type of a character device.
- *stats.isSymbolicLink()* - Returns true if file type of a symbolic link.
- *stats.isFIFO()* - Returns true if file type of a FIFO.
- *stats.isSocket()* - Returns true if file type of a socket.

Global objects

- Node.js global objects are global in nature and they are available in all modules.
- We do not need to include these objects in our application, rather we can use them directly.

• Objects:

- *__filename* – It represents the filename of the code being executed.
- *__dirname* – It represents the name of the directory that the currently executing script resides in.
- *setTimeout(cb, ms)* – It is a global function used to run callback cb after at least ms milliseconds.
- *clearTimeout(t)* – It is a global function used to stop a timer that was previously created with *setTimeout()*.
- *setInterval(cb, ms)* – It is global function used to run callback cb repeatedly after at least ms milliseconds.
- *process.argv* – Command line parameters.
- Open **8-GlobalObjects** folder, and run **testGlobalObj.js**.



```
JS testGlobalObj.js ×
JS testGlobalObj.js ↗ ⚡ setTimeout() callback
1 // __filename.
2 console.log('Current file: '+__filename);
3
4 // __dirname.
5 console.log('Current folder: '+__dirname);
6
7 // Set timer.
8 console.log('\nSetting up a timer...');
9 function timerHandler (){
10   console.log('Introduction to NodeJS - Interfaces course.');
11 }
12
13 setTimeout(timerHandler, 1000);
14 console.log('In 1000ms a message will appear...');
15
16 // Clearing a timer.
17 console.log('\nClearing a timer...');
18 var myTimer = setTimeout(timerHandler, 2000);
19 console.log('In 2000ms a message will appear...');
20 clearTimeout(myTimer);
21 console.log('Timer cleared... no message.');
22

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Setting up a timer...
In 1000ms a message will appear...

Clearing a timer...
In 2000ms a message will appear...
Timer cleared... no message.

Interval...
End...
Introduction to NodeJS - Interfaces course.
Clearing interval...
Introduction to NodeJS - Interfaces course.

i:\00-Univalle-Pendientes\Curso-Interfaces\NodeJS\9-NodeJS-Code\8-globalObjects>
```

Utility Modules

- *OS module*: Provides a few basic operating-system related utility functions.
- *Crypto module*: The crypto module offers a way of encapsulating secure credentials to be used as part of a secure HTTPS net or http connection. It also offers a set of wrappers for OpenSSL's hash, hmac, cipher, decipher, sign and verify methods.
- *Child process*: It is possible to stream data through a child's stdin, stdout, and stderr in a fully non-blocking way.
- Open folder **8-GlobalObjects**, and run **testUtilities.js**.



The screenshot shows a VS Code interface with two tabs: `testGlobalObjs.js` and `testUtilities.js`. The `testUtilities.js` tab contains the following code:

```
JS testGlobalObjs.js • JS testUtilities.js ×
JS testUtilities.js ► s.on('end') callback
  9  console.log(os.uptime());
10  console.log(os.loadavg());
11
12 // Crypto module.
13 console.log('\nCrypto module: ')
14 var crypto = require('crypto');
15 var fs = require('fs');
16 var shasum = crypto.createHash('sha1');
17 var s = fsReadStream('probeText.txt');
18
19 s.on('data', function(d) {
20   shasum.update(d);
21 });
22
23 s.on('end', function() {
24   var d = shasum.digest('hex');
25   console.log(d + ' probeText.txt\n');
26 });
27
28 // Child process.
29 console.log('\nChild process: ')
30 var exec = require('child_process').exec
```

The terminal below shows the execution results:

```
Child process:
8223651159c3500745eeae4c73dde6a2c91e377e probeText.txt

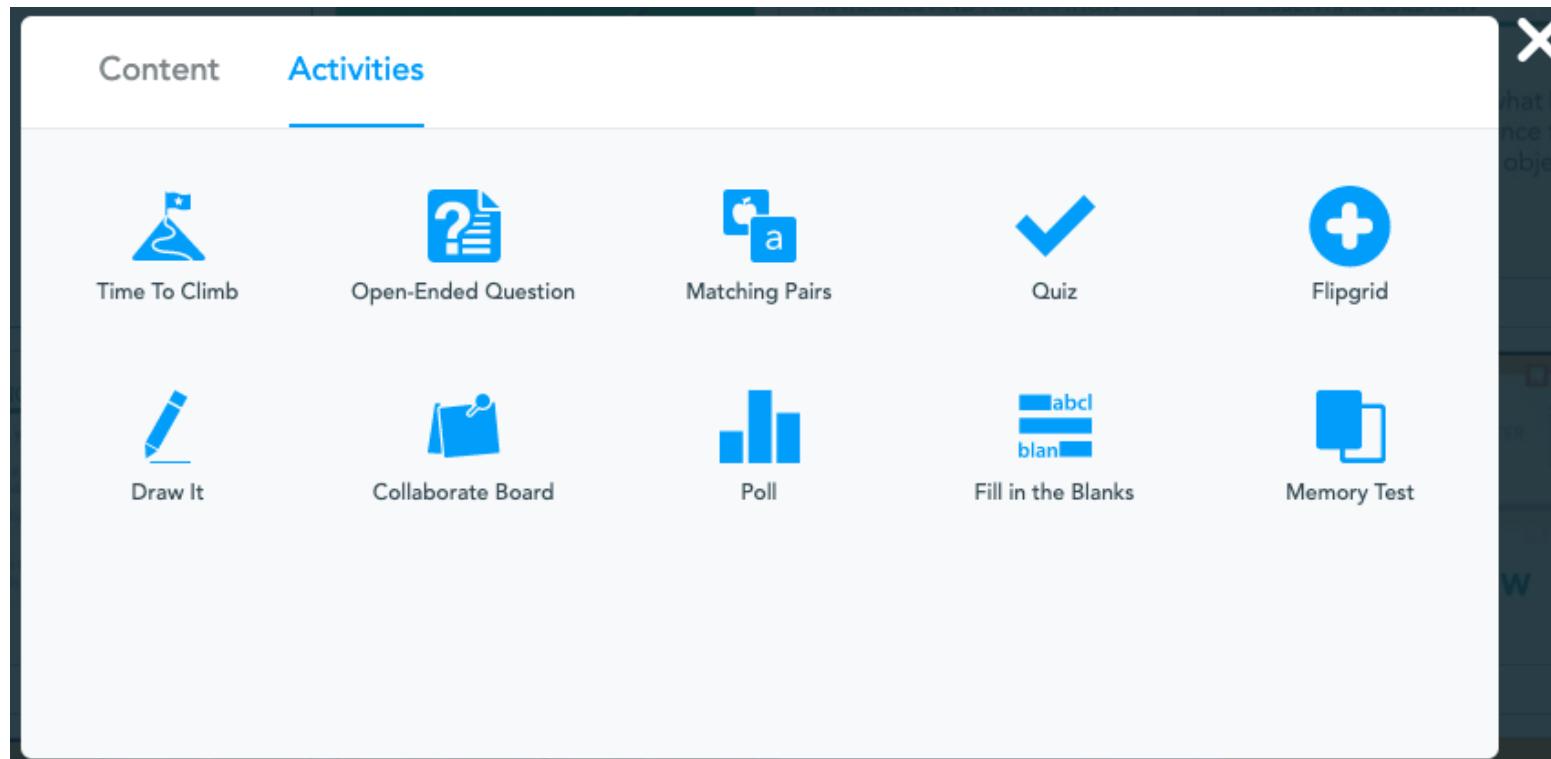
El volumen de la unidad I es EVLADY32G
El número de serie del volumen es: B4D0-1C29

Directorio de i:\00-Univalle-Pendientes\Curso-Interfaces\NodeJS\9-NodeJS-Code

2019-06-17 04:57 p.m. <DIR> .
2019-06-17 04:57 p.m. <DIR> ..
2019-06-20 03:05 p.m. <DIR> 1-HelloWorld
2019-06-20 03:37 p.m. <DIR> 2-SimpleServer
2019-06-20 04:31 p.m. <DIR> 3-BasicsNodeJs
2019-06-25 11:12 a.m. <DIR> 4-CallBacks
2019-06-27 08:52 a.m. <DIR> 5-Events
2019-06-27 12:02 p.m. <DIR> 6-BuffersStreams
2019-06-28 10:44 a.m. <DIR> 7-FileSystem
```

Nearpod Activity

- Please go to the Nearpod link shared in the chat.
- Fulfil the Nearpod activity.
- Analyze the results with your teacher.



¿Questions?

