

# NODE.JS

---

Server Side Javascript

# Module contents

- Introduction to Node.js
- Node modules
- Selectors
- Syntax
- Developing node.js web application
- Event-driven I/O server-side JavaScript

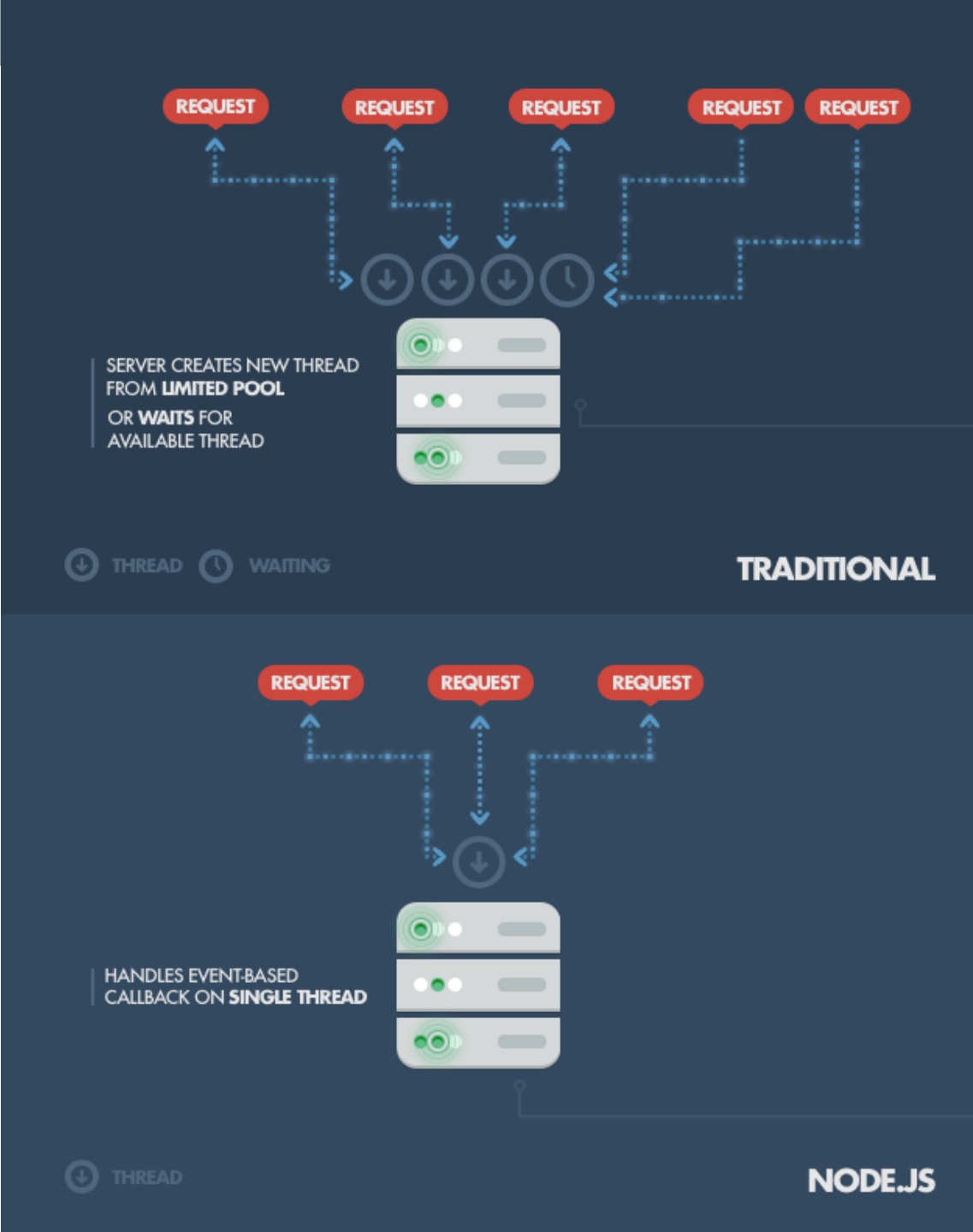
# Node.js – an intro

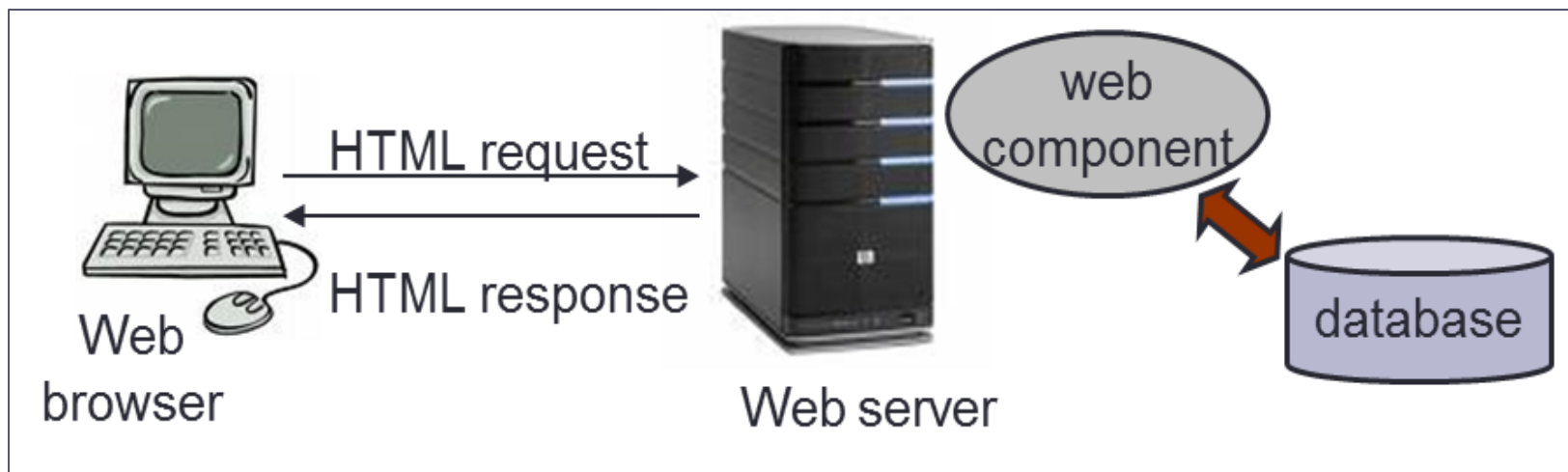
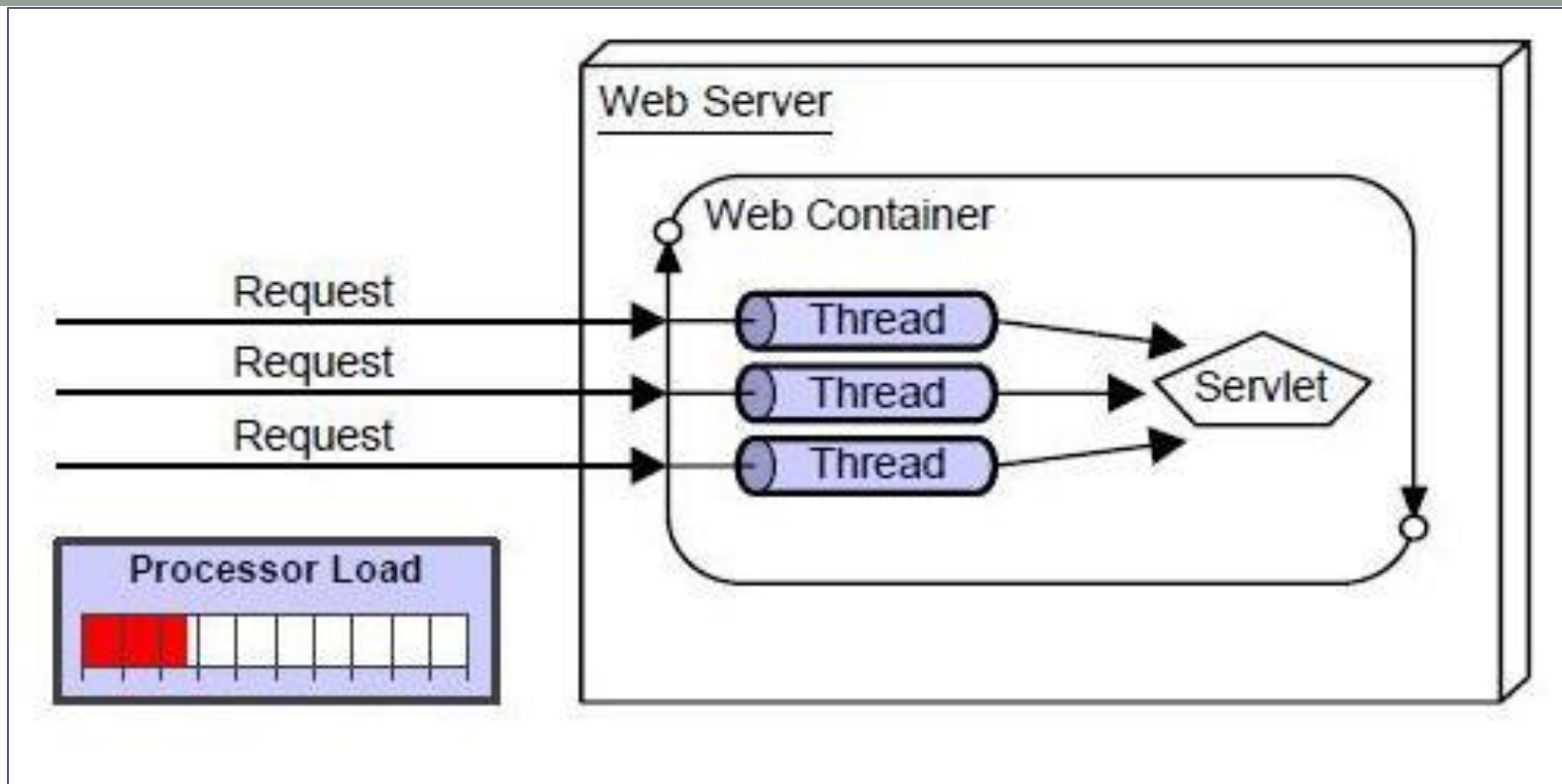
- In 2009 Ryan Dahl created Node.js or Node, a framework primarily used to create highly scalable servers for web applications. It is written in C++ and JavaScript.
- Node.js is a development framework that is based on Google's V8 JavaScript engine.
- You write Node.js code in JavaScript, and then V8 compiles it into machine code to be executed.
- You can write most—or maybe even all—of your server-side code in Node.js, including the webserver and the server-side scripts and any supporting web application functionality.
  - The fact that the webserver and the supporting web application scripts are running together in the same server-side application allows for much tighter integration between the webserver and the scripts.
  - *Using JavaScript on the server also reduces the context switching that needs to happen in your brain as you change programming language and associated code convent*

# Node.js – an intro

- It's a highly scalable system that uses asynchronous, non-blocking I/O model (input/output), rather than threads or separate processes
- It is not a framework like jQuery nor a programming language like C# or JAVA
- It's a new kind of web server that has a lot in common with other popular web servers, like Microsoft's Internet Information Services (IIS) or Apache
- IIS / Apache processes only HTTP requests, leaving application logic to be implemented in a language such as PHP or Java or ASP.NET.
  - Node removes a layer of complexity by combining server and application logic in one place.

- Compared to traditional web-serving techniques where each connection (request) spawns a new thread, taking up system RAM and eventually maxing-out at the amount of RAM available, Node.js operates on a single-thread, using non-blocking I/O calls, allowing it to support tens of thousands of concurrent connections ([held in the event loop](#)).





# Traditional Programming Limitations

- In traditional programming I/O is performed in the same way as it does local function calls. i.e. Processing cannot continue until the operation is completed.
- When the operation like executing a query against database is being executed, the whole process/thread idles, waiting for the response. This is termed as “Blocking”
- Due to this blocking behavior we cannot perform another I/O operation, the call stack becomes frozen waiting for the response.
- Multi-threading is one alternative to this programming model.
  - Makes use of separate CPU Cores as “Threads”
  - Uses a single process within the Operating System
- If the application relies heavily on a shared state between threads accessing and modifying shared state increase the complexity of the code and It can be very difficult to configure, understand and debug.

# Event Driven programming style

- Event-driven programming or Asynchronous programming is a programming style where the flow of execution is determined by events.
- Events are handled by event handlers or event callbacks
  - An event callback is a function that is invoked when something significant happens like when the user clicks on a button or when the result of a database query is available.
- This style of programming — whereby instead of using a return value you define functions that are called by the system when interesting events occur — is called event-driven or asynchronous programming.

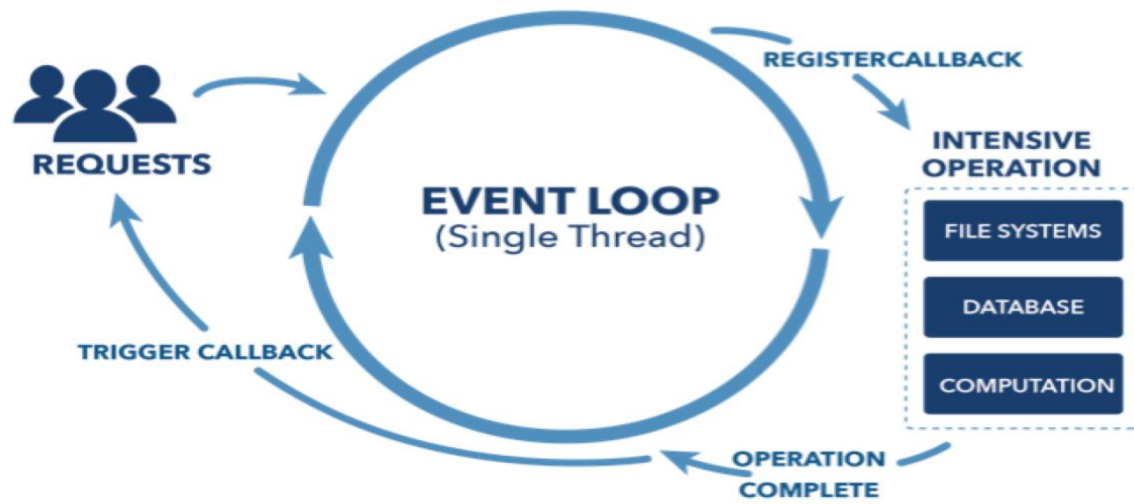
```
result = query('SELECT * FROM posts WHERE id = 1');  
do_something_with(result);
```

```
query_finished = function(result) {  
    do_something_with(result);  
}  
query('SELECT * FROM posts WHERE id = 1', query_finished);
```

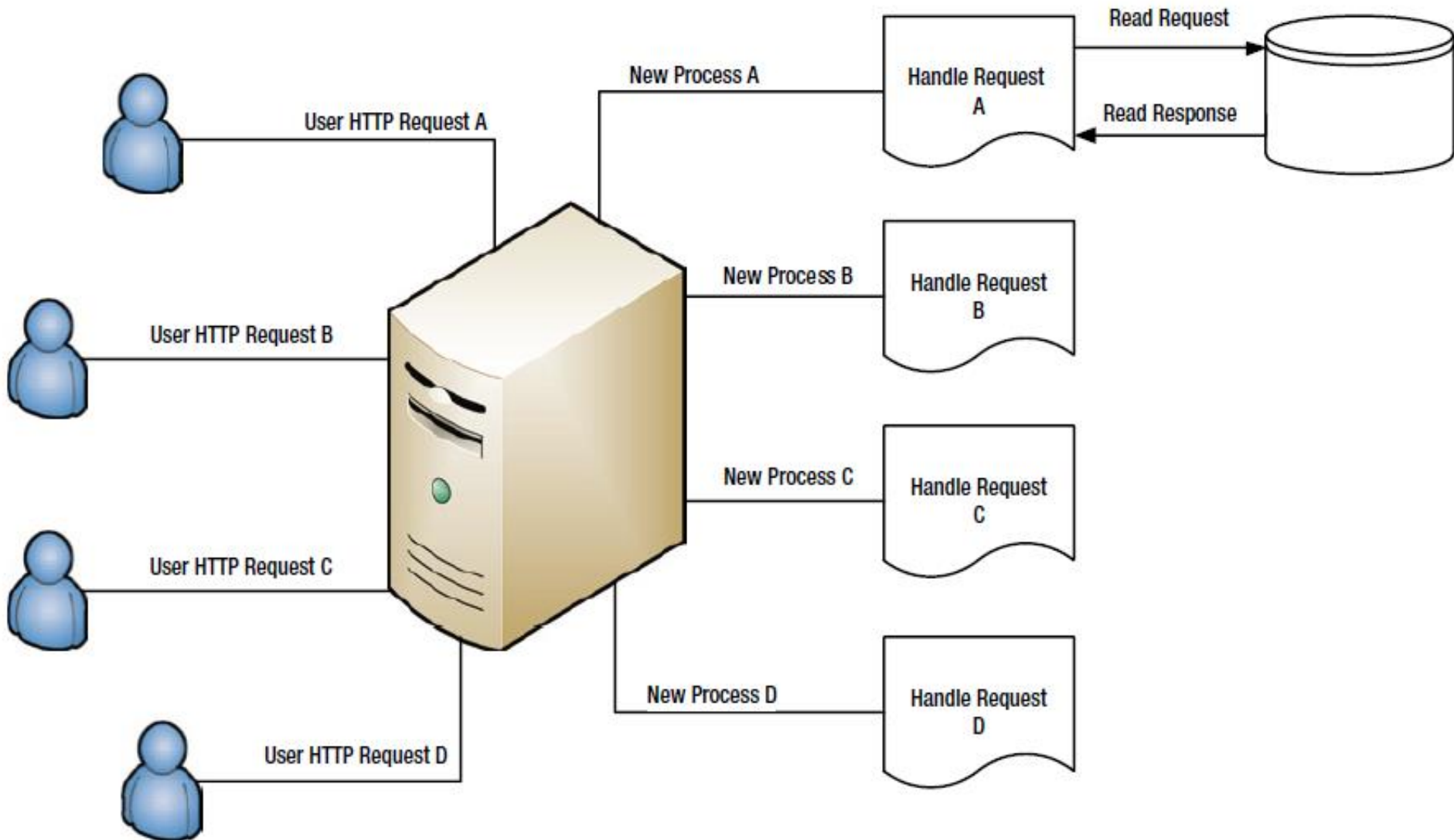


# Event loop

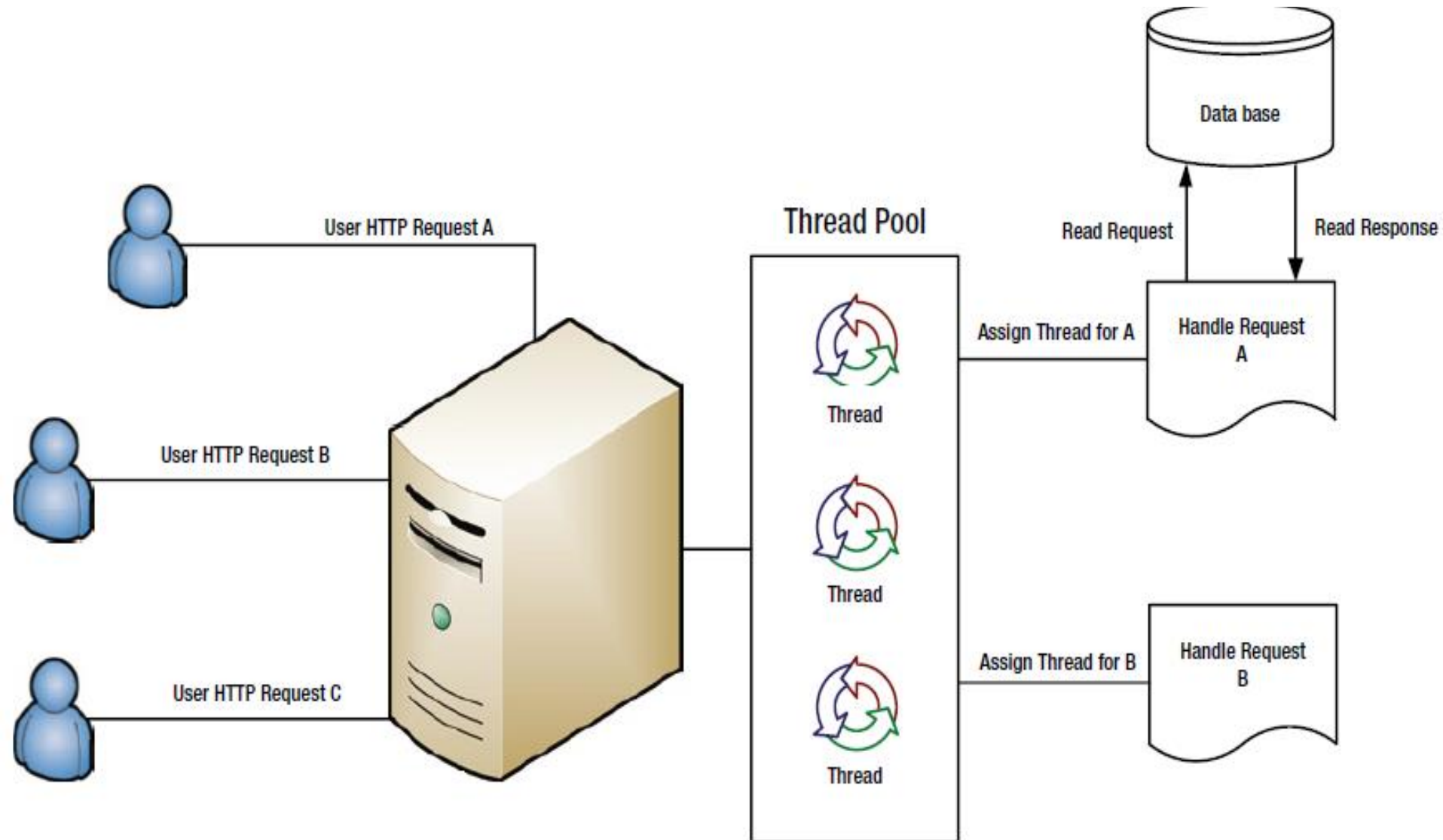
- An event loop is a construct that mainly performs two functions in a continuous loop — event detection and event handler triggering.
  - In any run of the loop, it has to detect which events just happened.
  - Then, when an event happens, the event loop must determine the event callback and invoke it.
- This event loop is just one thread running inside one process, which means that, when an event happens, the event handler can run without interruption. This means the following:
  - There is at most one event handler running at any given time.
  - Any event handler will run to completion without being interrupted.



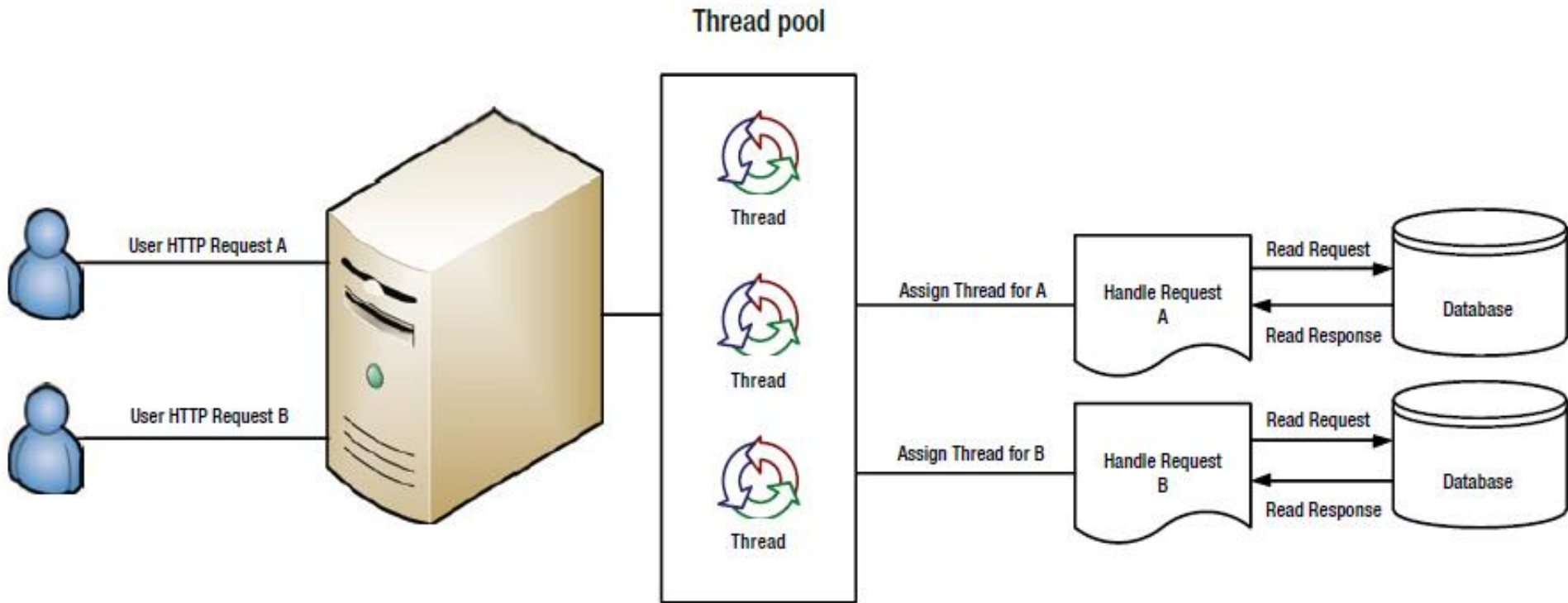
# Traditional web server using Processes



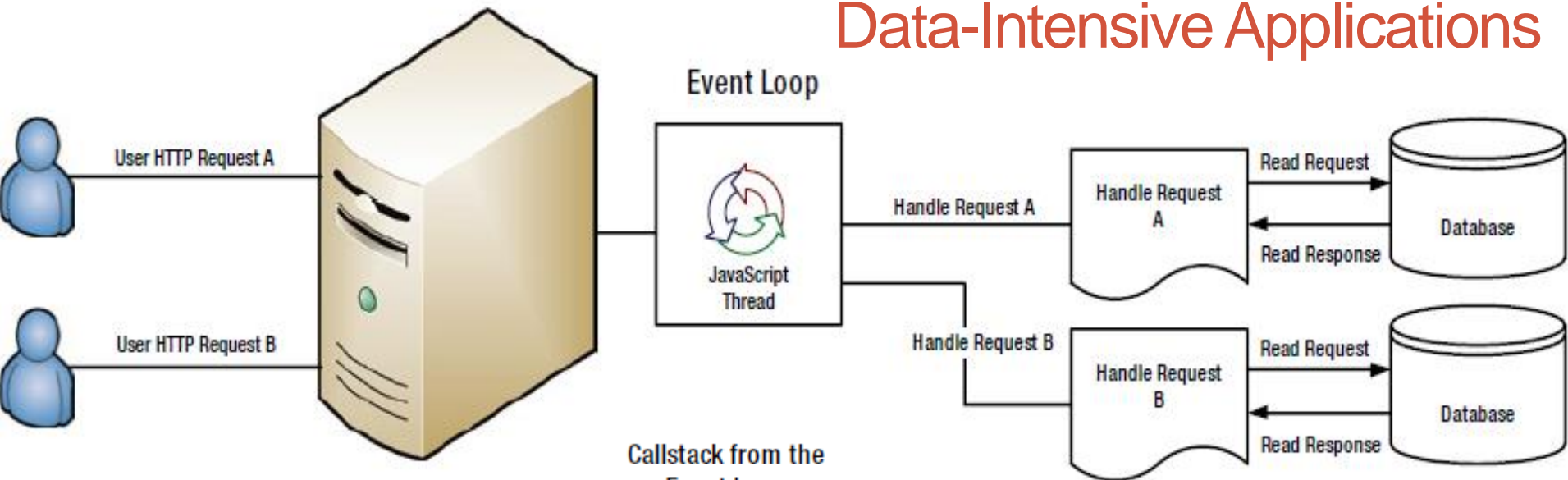
# Traditional Web Servers Using a Thread Pool



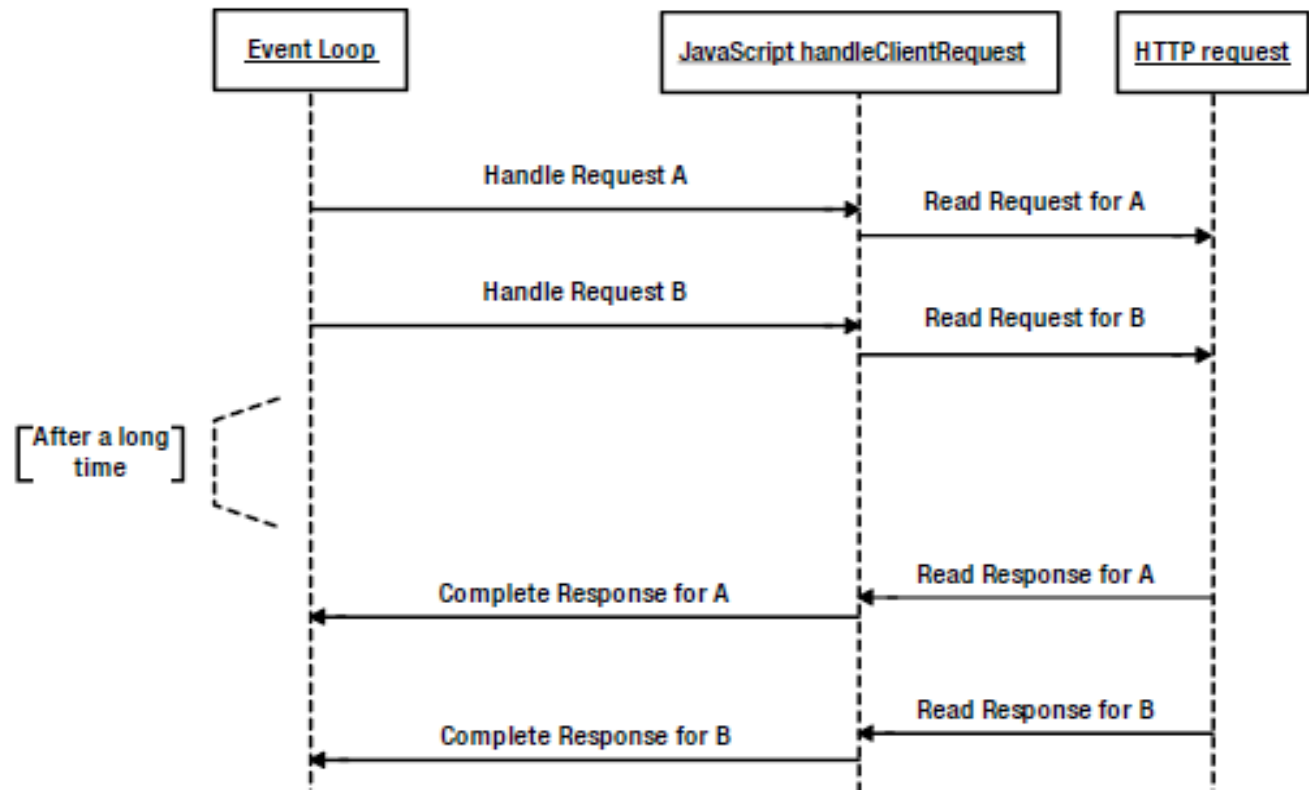
# Data-Intensive Applications



# Data-Intensive Applications



Callstack from the Event Loop

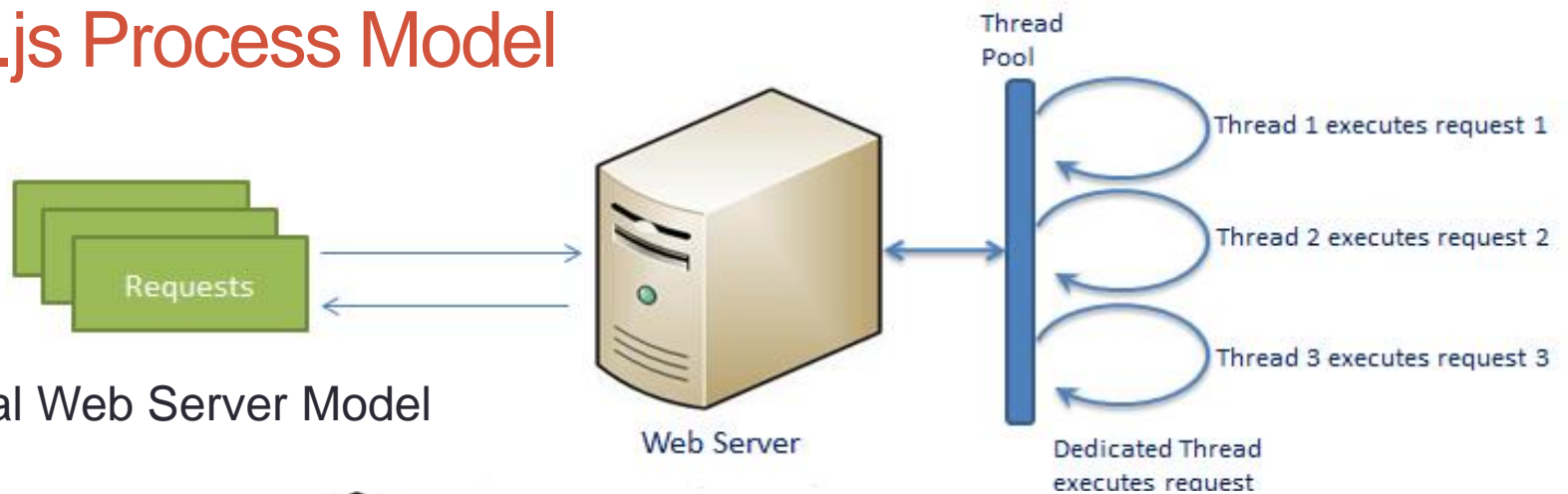


# Asynchronous and Event Driven

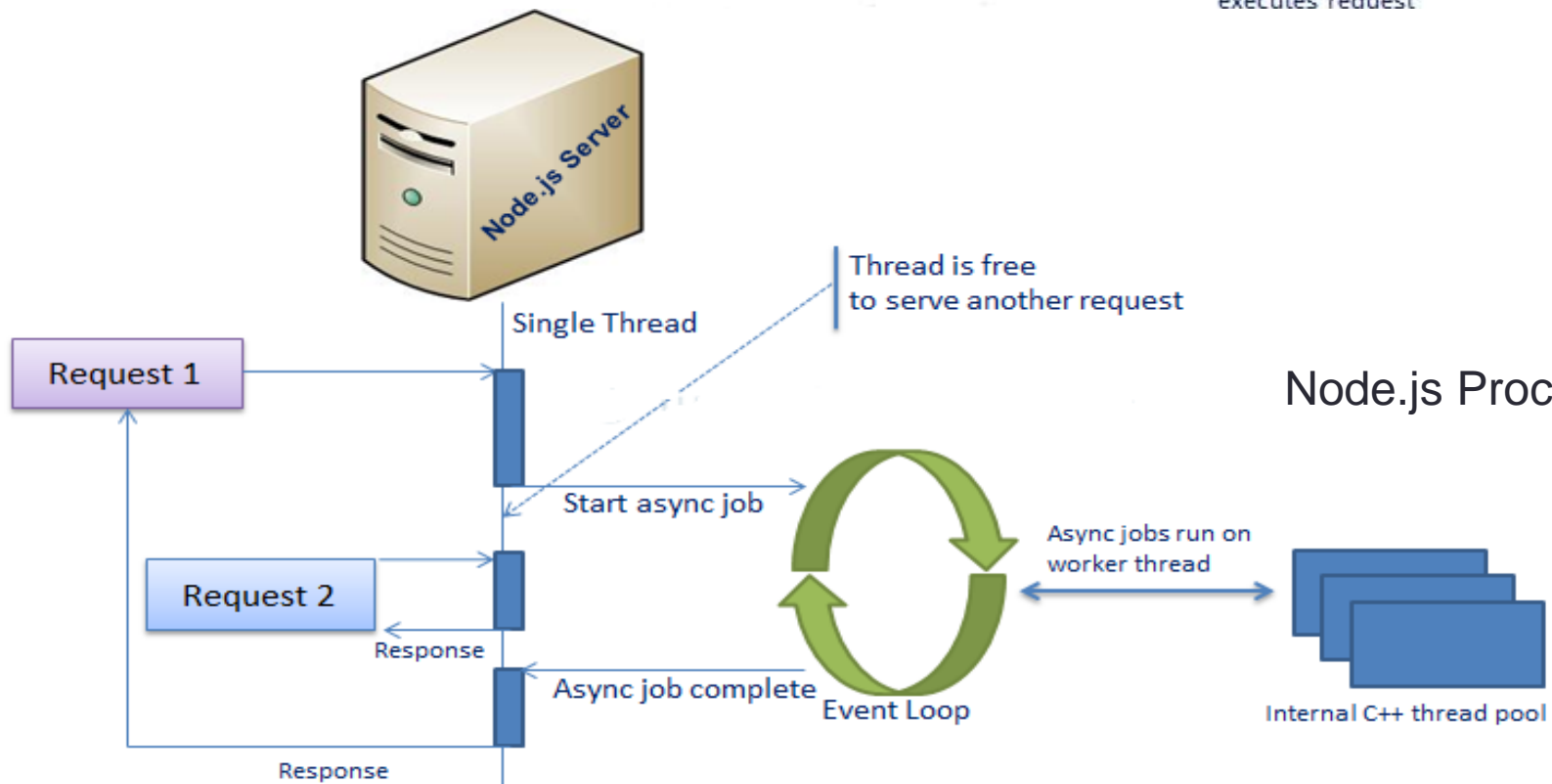
- All APIs of Node.js library are asynchronous that is, non-blocking.
  - It essentially means a Node.js based server never waits for an API to return data.
  - The server moves to the next API after calling it and a notification mechanism of Node.js helps the server to get a response from the previous API call.
  - It is non-blocking, so it doesn't make the program wait, but instead it registers a callback and lets the program continue.
  - Event mechanism helps the server to respond in a non-blocking way and makes the server highly scalable as opposed to traditional servers which create limited threads to handle requests.
  - Node.js uses a single threaded program and the same program can provide service to a much larger number of requests than traditional servers like Apache HTTP Server.

# Node.js Process Model

Traditional Web Server Model



Node.js Process Model



# Where can I use Node.js?

- Node.js **is not fit** for an application which performs CPU-intensive operations like image processing or other heavy computation work because it takes time to process a request and thereby blocks the single thread.
- Node.js is great for data-intensive applications.
  - Using a single thread means that Node.js has an extremely low-memory footprint when used as a web server and can potentially serve a lot more requests.
  - Eg, a data intensive application that serves a dataset from a database to clients via HTTP
- **What Node is NOT!**

Node is **not** a webserver. By itself it doesn't do anything. It doesn't work like Apache. There is no config file where you point it to your HTML files. If you want it to be a HTTP server, you have to write an HTTP server (with the help of its built-in libraries). Node.js is just another way to execute code on your computer. It is simply a JavaScript runtime.



# Who Uses Node.js?

- An exhaustive list of projects, application and companies are using Node.js.
- This list includes eBay, General Electric, GoDaddy, Microsoft, PayPal, Uber, Wikipins, Yahoo!, and Yammer to name a few.
  - Paypal – A lot of sites within Paypal have also started the transition onto Node.js.
  - LinkedIn - LinkedIn is using Node.js to power their [Mobile](#) Servers, which powers the iPhone, Android, and Mobile Web products.
  - Mozilla has implemented Node.js to support browser APIs which has half a billion installs.
  - Ebay hosts their HTTP API service in Node.js
  - Walmart's black Friday sale is powered via Node
  - Yahoo!
  - *The New York Times*
  - Dow Jones
  - Microsoft

<https://nodejs.org/en/>

<https://nodejs.org/en/>



[HOME](#) | [ABOUT](#) | [DOWNLOADS](#) | [DOCS](#) | [FOUNDATION](#) | [GET INVOLVED](#) | [SECURITY](#) | [NEWS](#)

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, [npm](#), is the largest ecosystem of open source libraries in the world.

Important [security releases](#), please [update now!](#)

## Download for Windows (x64)

**v4.4.0 LTS**

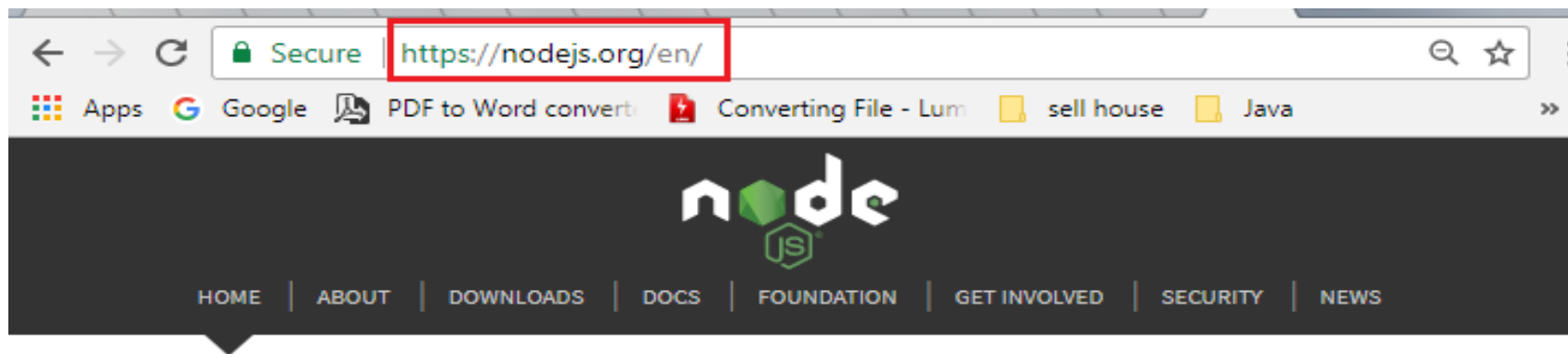
Recommended For Most Users

**v5.8.0 Stable**

Latest Features

[Other Downloads](#) | [Changelog](#) | [API Docs](#)

[Other Downloads](#) | [Changelog](#) | [API Docs](#)



Node.js® is a JavaScript runtime built on [Chrome's V8 JavaScript engine](#). Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, [npm](#), is the largest ecosystem of open source libraries in the world.

Important [DOS security vulnerability](#), Release coming Tuesday  
October 24th

## Download for Windows (x64)

**6.11.4 LTS**

Recommended For Most Users

**8.7.0 Current**

Latest Features

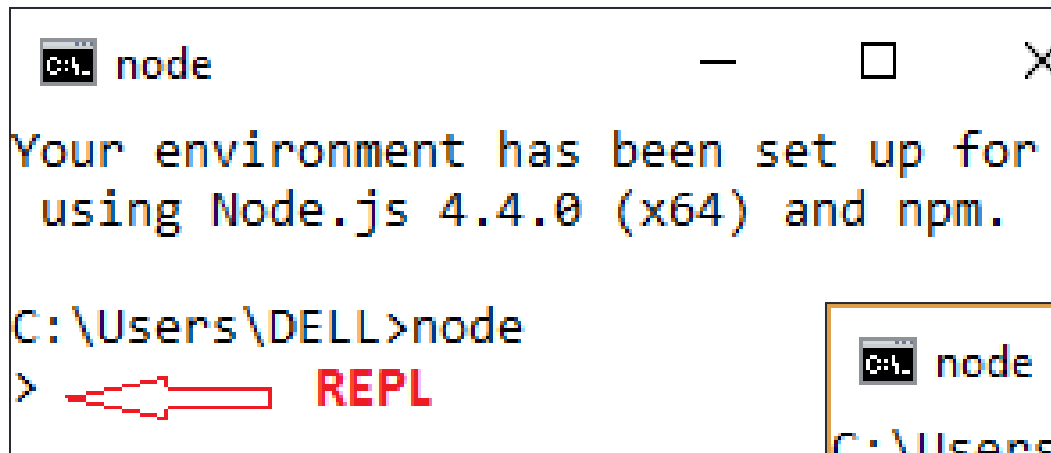
[Other Downloads](#) | [Changelog](#) | [API Docs](#)

[Other Downloads](#) | [Changelog](#) | [API Docs](#)

```
C:\Users\Administrator>node --version  
v6.11.0
```


# Using the Node CLI : REPL (Read-Eval-Print-Loop)

- There are two primary ways to use Node.js on your machines: by using the Node Shell or by saving JavaScript to files and running those.
  - Node shell is also called the Node REPL; a great way to quickly test things in Node.
  - When you run “node” without any command line arguments, it puts you in REPL

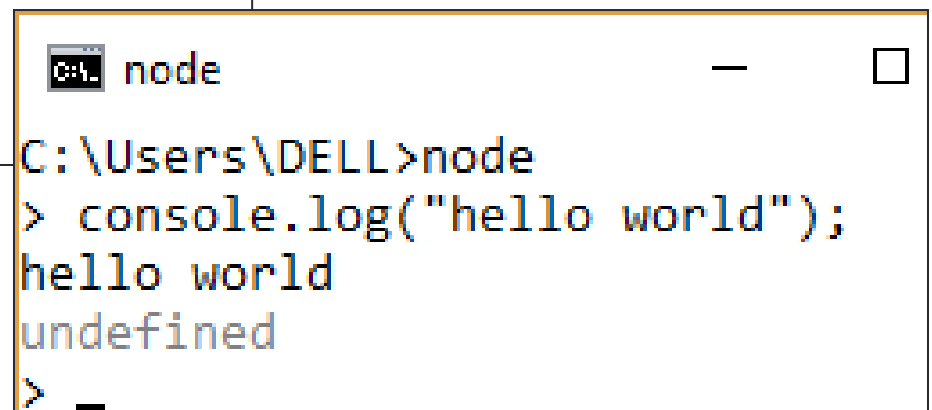


```
C:\> node


Your environment has been set up for
using Node.js 4.4.0 (x64) and npm.

C:\Users\DELL> node
>  REPL
```

A terminal window titled 'node' showing the initial setup for the Node.js REPL. It displays a message about the environment being set up for Node.js 4.4.0 (x64) and npm. The prompt 'C:\Users\DELL>' is followed by 'node', and the next line shows a red arrow icon pointing to the word 'REPL'.



```
C:\> node

C:\Users\DELL> node
> console.log("hello world");
hello world
undefined
> 
```

A terminal window titled 'node' showing the Node.js REPL in action. The prompt 'C:\Users\DELL>' is followed by 'node'. The next line shows the command 'console.log("hello world");' being entered, followed by the output 'hello world' and 'undefined'. The prompt 'C:\Users\DELL>' is followed by a black cursor icon.

# Using the REPL

```
c:\ node
> 10+20
30
> x=50
50
> x
50
> _
```

```
> var foo = [];
undefined
> foo.push(123);
1
> foo
[ 123 ]
>
```

```
> function add(a,b){
...   return (a+b);
... }
undefined
> add(10,20)
30
> _
```

```
> var x = 10, y = 20;
undefined
> x+y
30
```

- You can also create a js file and type in some javascript.

```
C:\Users\DELL>node helloworld.js
Hello World!
```

```
//helloworld.js
console.log("Hello World!");
```

*Because you are not in the Node shell, you don't get any information on the return values of the code executed.*

# Using the REPL

- To view the options available to you in REPL type `.help` and press Enter.

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

- Eg.

```
> .load helloworld.js
> console.log('Hello World!!');
Hello World!!
undefined
>
```

# Demo

```
//loopAndArrayDemo.js
```

```
for(var i=1;i<11;i++)  
    console.log(i );
```

```
var arr1 = [10,20,30];  
arr1.push(40);
```

```
console.log('arr length: ' + arr1.length);  
console.log('arr contents: ' + arr1);
```

```
> .load loopandarraydemo.js  
> for(var i=1;i<11;i++)  
...     console.log(i);  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
undefined  
> var arr1 = [10,20,30];  
undefined  
> arr1.push(40);  
4  
> console.log('arr length: ' + arr1.length);  
arr length: 4  
undefined  
> console.log('arr contents: ' + arr1);  
arr contents: 10,20,30,40  
undefined  
> _
```

functionsEx.js  
JSObjEx.js



# UNDERSTANDING NODE.JS

---



# Variables and functions (Recap)

- Variables can be declared as usual.
  - But, if **var** keyword is not used, then the value is stored in the variable and printed.
  - Whereas if **var** keyword is used, then the value is stored but not printed.
  - You can print variables using **console.log()**.

```
> a=100;
100
> var b=200;
undefined
> a+b
300
> console.log("Welcome");
Welcome
undefined
> _
```

- **Functions:**
  - All functions return a value in JavaScript. If no explicit return statement, func returns undefined.

- **Anonymous Function**

- A function without a name

```
var foo2 = function () {
    console.log('foo2');
}
foo2(); // foo2
```

```
> .load functionsEx.js
> function foo() { return 123; }
undefined
> console.log(foo()); // 123
123
undefined
> function bar() { }
undefined
> console.log(bar()); // undefined
undefined
undefined
>
```

# Higher-Order Functions (Recap)

- Since JavaScript allows us to assign functions to variables, we can pass functions to other functions.
  - Functions that take functions as arguments are called *higher-order functions*
  - Eg, `geolocation.getCurrentPosition(func1, func2)`
  - Eg `$(document).ready(function(){});`
  - Eg

```
setTimeout(function () {  
    console.log('2 secs have passed since demo started'); }, 2000);
```

# Closures

- A closure is an inner function that has access to the outer (enclosing) function's variables—scope chain.
  - The closure has three scope chains: it has access to its own scope (variables defined between its curly brackets), it has access to the outer function's variables, and it has access to the global variables.

```
function showName (firstName, lastName) {  
  var nameIntro = "Your name is ";  
  // inner function has access to the outer function's variables, including the parameter  
  function makeFullName ()  
    return nameIntro + firstName + " " + lastName;  
  }  
  return makeFullName ();  
}  
showName ("Michael", "Jackson"); // Your name is Michael Jackson
```



closureEg.html

- With closures, the inner function still has access to the outer function's variables even after the outer function has returned.
- Therefore, you can call the inner function later in your program!!!

# Closures

```
var landscape = function() {  
  var result = "";  
  var flat = function(size) {  
    for (var count = 0; count < size; count++)  
      result += "_";  
  };  
  var mountain = function(size) {  
    result += "/";  
    for (var count = 0; count < size; count++)  
      result += "";  
    result += "\\";  
  };  
  
  flat(3);  
  mountain(4);  
  flat(6);  
  mountain(1);  
  flat(1);  
  return result;  
};  
console.log(landscape()); // → ____/""\____/\_
```

- The flat and mountain functions can “see” the variable called result, since they are inside the function that defines it.
- But they cannot see each other’s count variables since they are outside each other’s scope.
- The environment outside of the landscape function doesn’t see any of the variables defined inside landscape.

# Closures (Recap)

- If a function defined inside another function, the inner function has access to the variables declared in the outer function.

```
function outerFunction(arg) {  
    var outerVar = arg;  
    function bar() {  
        console.log(outerVar); // Access a variable from the outer scope  
    }  
    bar(); // Call the local function to demonstrate that it has access to arg  
}  
outerFunction('hello closure!'); // logs hello closure!
```

```
function outerFunc(arg) {  
    var outerVar = arg;  
    return function () {  
        console.log(variableInOuterFunction);  
    }  
}  
  
var innerFunc = outerFunc('hello closure!'); // Note the outerFunc has returned  
innerFunc(); // logs hello closure!
```

# Closures

- Closures are functions that inherit variables from their enclosing environment.
  - Eg, to listen for an event, say a button click, you can do something like:

```
var clickCount = 0;
document.getElementById('myButton').onclick = function() {
    clickCount += 1;
    alert("clicked " + clickCount + " times.");
};
```

JavaScript

```
var clickCount = 0;
$('#button#mybutton').click(function() {
    clickedCount++; alert('Clicked ' + clickCount + ' times.');
```

jQuery:

```
(function() {
    var clickCount = 0;
    $('#button#mybutton').click(function() {
        clickCount++; alert('Clicked ' + clickCount + ' times.');
```

```
IIFE
(function () {
    // logic here
})();
```

# Some Important Node.js Globals

- Node.js has a few key global variables that are always available to you.
  - One of them is the **console object**, which we have been using up to this point.
  - In a browser, global scope is the window object.
  - In the browser's JavaScript, variables declared without var keyword become global. In Node.js, everything becomes local by default.
  - In Node.js, **global** object represents the global scope.
  - Anything attached to it is available anywhere in your node application
  - `global.cname = 'TriTech'`
  - `global['cname']` // We can directly access the members attached to global.

```
global.fish = "swordfish";
global.pet = "cat";
function printit(var_name) {
    console.log(global[var_name]);
}
printit("fish");    // prints swordfish
printit("pet");     // prints cat
printit("fruit");   // prints undefined
```



UserDefGlobalObj.js

# Some Important Node.js Globals

- The other key global in Node is the **process** global variable.
  - Each Node.js script runs in a process. It includes **process** object to get all the information about the current process of Node.js application like the PID, environment variables, platform, memory usage etc.
  - `process.platform` //eg win32
  - `process.exit( )`
- Process : contains a lot of info and methods:
  - `exit()` terminate your Node.js programs
  - `env` function returns an object with current user environment variables,
  - `cwd()` returns current working directory of the app.

```
G:\FreeLanceTrg\Node.js\Demo\Intro>node
> process.execPath
'C:\\Program Files\\nodejs\\node.exe'
> process.pid
12032
> process.cwd()
'G:\\FreeLanceTrg\\Node.js\\Demo\\Intro'
>
```



# Node.js Global Objects

- Node.js global objects are global in nature and available in all modules.
  - You don't need to include these objects in your application; rather they can be used directly.
  - These objects are modules, functions, strings and object etc.

- A list of Node.js global objects are:

- `__dirname`
- `__filename`
- `Console`
- `Process`
- `Buffer`
- `setImmediate(callback[, arg][, ...])`
- `setInterval(callback, delay[, arg][, ...])`
- `setTimeout(callback, delay[, arg][, ...])`
- `clearImmediate(immediateObject)`
- `clearInterval(intervalObject)`
- `clearTimeout(timeoutObject)`

```
console.log( __filename );  
console.log( __dirname );
```

```
G:\FreeLanceTrg\Node.js\Demo\filesystem\module1.js  
G:\FreeLanceTrg\Node.js\Demo\filesystem
```

**Global functions**

```
console.log("This file is " + __filename);  
console.log("It's located in " + __dirname);
```

# MODULE SYSTEM

---

# Node js Modules

- Consider modules to be the same as JavaScript libraries.
  - A set of functions you want to include in your application.
  - Module in Node.js is a simple or complex functionality organized in single or multiple JavaScript files which can be reused throughout the Node.js application.
  - If you have a library of functions or classes for working with a particular database server, for example, it would make a lot of sense to put that code into a module and package it for consumption.
- Node.js uses a module architecture to simplify creation of complex apps
  - Modules are akin to libraries in C and Java; each module contains a set of functions related to the "subject" of the module.
  - For ex, the http module contains functions specific to HTTP. Eg `http.createServer()`



# Node js Modules

- Node.js has a set of built-in modules which you can use without any further installation.
- Built-in modules provide a core set of features we can build upon.
  - Each module in Node.js has its own context, so it cannot interfere with other modules or pollute global scope.
  - Also, each module can be placed in a separate .js file under a separate folder.
  - To include a module, use the **require()** function with the name of the module.
- In Node, modules are referenced either by file path or by name
  - For example, we can require some native modules:

```
var http = require('http');  
var dns = require('dns');
```

- We can also require relative files

```
var myFile = require('./myFile'); // loads myFile.js
```

# Node.js Web App

//RunServer.js

```
var http = require("http");  
function process_request(req, res) {  
  var body = 'Hello World\n';  
  var content_length = body.length ;  
  res.writeHead(200, {  
    'Content-Length': content_length,  
    'Content-Type': 'text/plain'  });  
  res.end(body);  
}  
var srv = http.createServer(process_request);  
srv.listen(1337, '127.0.0.1');  
console.log('Server running at http://127.0.0.1:1337/');
```

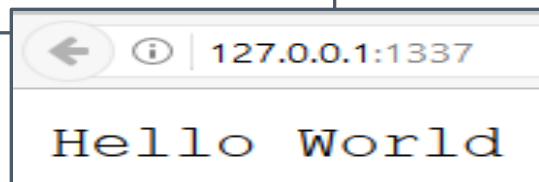
Import required module using **require**; load http module and store returned HTTP instance into http variable

creates an HTTP server which listens for request over 1337 port on local machine



Intro/RunServer.js

```
G:\FreeLanceTrg\Node.js\Demo\Intro>node runserver.js  
Server running at http://127.0.0.1:1337/
```



# Node.js Web App

```
var http = require('http');  
http.createServer(function (req, res) {  
  res.writeHead(200, {'Content-Type': 'text/plain'});  
  res.end('Hello World\n');  
}).listen(1337, '127.0.0.1');
```

```
var http=require("http");
var fs=require('fs');
var url=require('url');
const querystring=require("querystring");
var m1=require("./formmodule");
function proces_request(req,resp){
    var u=req.url;
        const p=url.parse(req.url);
        console.log(p);
        const query=querystring.parse(p.query);
        console.log("query");
        console.log(query);
        resp.writeHead(200,{'Content-Type':"text/html"});
        switch(u){
            case "/": fs.readFile("form.html",function(err,data){
                    if(err){
                        resp.write("some error");
                        console.log(err);
                    }else{
                        resp.write(data);
                        resp.end();    }
                });
            break;
```



```
default:                var bdata="";
                        req.on('data',function(d){
                                bdata+=d.toString();
                        });
                        req.on("end",function(){
                                var b=querystring.parse(bdata);
                                console.log(b);
                                //resp.end("done");
                                console.log(bdata);
                                var arr=bdata.split("&");
                                console.log(arr);
                                var a=arr[0].split("=");
                                console.log(a[1]);
                                var b=arr[1].split("=");
                                console.log(b[1]);
                                //var b=bdata.parse();
                                var s=m1.add(a[1],b[1]);
                                resp.write("num1:"+a[1]);
                                resp.write("num2:"+b[1]);
                                resp.write("Addition:"+s);
                                resp.end();    });
```

```
break;
```

# Node.js Web App

- **require** is a global function that just loads a file named `http.js`.
  - If this file isn't found in the current directory, it'll pull it from the standard libraries that come with Node.js.
  - When you require a script, it returns a value specified within the file you pulled in; In this case, you get an object that represents an HTTP server.
- **createServer()** takes a callback function that is executed every time the server receives a request and it sends objects representing the request and response into that callback function.
  - `createServer` also returns a server object.
  - `res.writeHead()` sets headers for the response.
  - `res.end()` means we've reached the end of response
  - Can also be written as
- `createServer().listen()` takes a port number and an IP address to listen to for requests.

```
res.write("Hello World\n");  
res.end();
```

# Loading a module

- Loading a core module

- Node has several modules compiled into its binary distribution. These are called the core modules. It is referred solely by the module name, not by the path and are preferentially loaded even if a third-party module exists with the same name.
- `var http = require('http');`

- Loading a file module (User defined module)

- We can load non-core modules by providing the absolute path / relative path. Node will automatically adding the .js extension to the module referred.
- `var myModule = require('d:/karthik/nodejs/mymodule');` // Absolute path for module.js
- `var myModule = require('../mymodule');` // Relative path for module.js (one folder up level)
- `var myModule = require('./mymodule');` // Relative path for module.js (Exists in current directory)

# Loading a module

- Loading a folder module (User defined module)
  - It is convenient to organize programs and libraries into self-contained directories, and then provide a single entry point to that library.
- How to pass a folder to require() as an argument.
  - Create a package.json file in the root of the folder, which specifies a main module. An example package.json file might look like this:

```
{ "name" : "some-library",  
  "main" : "./lib/some-library.js" }
```
  - If this was in a folder at ./some-library, then require('./some-library') would attempt to load ./some-library/lib/some-library.js.
  - Node will presume the given folder as a package and look for a package definition file inside the folder. Package definition file name should be named as package.json
  - Node will try to parse package.json and look for and use the main attribute as a relative path for the entry point.

# Creating Package.json

- We need to use npm init command to create package.json.
  - D:\NodeJs\modules> npm init

```
{
  "name": "nodejs",
  "version": "1.0.0",
  "description": "",
  "main": "formadata.js",
  "dependencies": {
    "body-parser": "^1.18.2",
    "express": "^4.16.3"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Kishori",
  "license": "ISC"
}
```

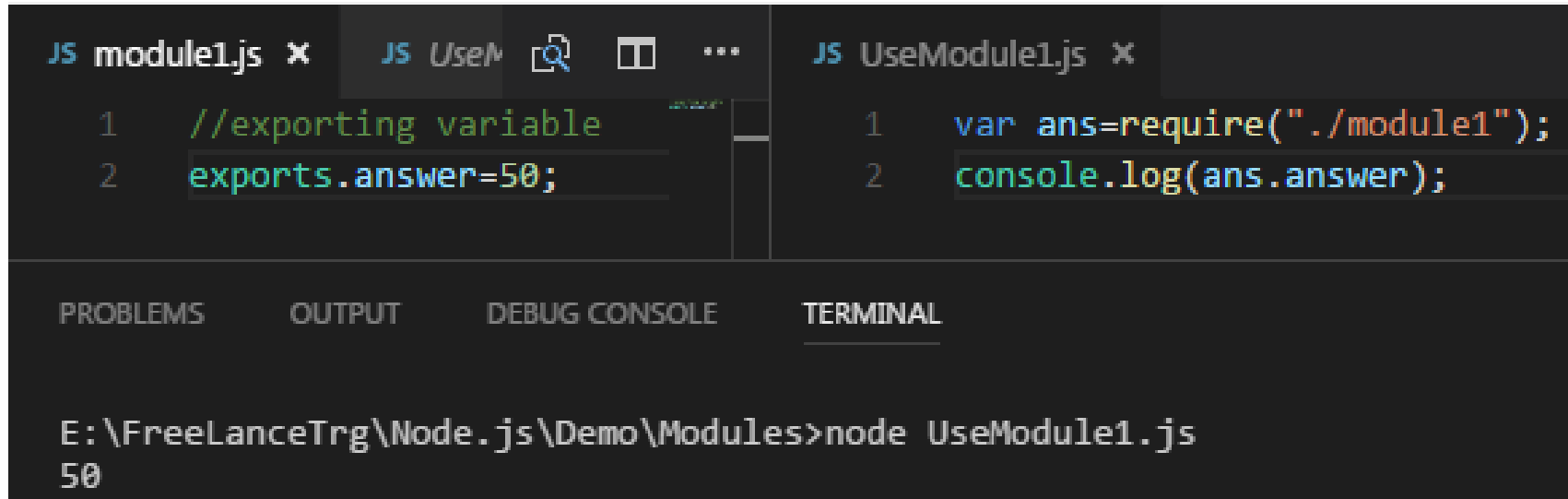
- `var myModule = require('E:/Node.js/Demo/foldermodule ');` // refer index.js placed in *foldermodules* folder

# package.json

- package.json is a configuration file from where the npm can recognize dependencies between packages and installs modules accordingly.
  - It must be located in project's root directory.
- The following fields are used to build the schema for package.json file
  - name and version : package.json must be specified at least with a name and version for package. Without these fields, npm cannot process the package.
  - description and keywords : description field is used to provide a textual description of package. Keywords field is used to provide an array of keywords to further describe the package.
  - author : The primary author of a project is specified in the author field.
  - main : Instruct Node to identify its main entry point.
  - dependencies : Package dependencies are specified here
  - scripts : The scripts field, contains a mapping of npm commands to script commands. The script commands, which can be any executable commands, are run in an external shell process. Two of the most common commands are start and test. The start command launches your application, and test runs one or more of your application's test scripts.

# Create Your Own Modules

- You can create your own modules, and easily include them in your applications.



```
JS module1.js x JS UseModule1.js x
1 //exporting variable
2 exports.answer=50;

1 var ans=require("./module1");
2 console.log(ans.answer);

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
E:\FreeLanceTrg\Node.js\Demo\Modules>node UseModule1.js
50
```

```
//module2.js
exports.sayHelloInEnglish = function(){
    return "Hello";
};
exports.sayHelloInSpanish = function(){
    return "Hola";
};
```

```
//UseModule2.js
var greet=require("./module2");
console.log(greet.sayHelloInSpanish());
//Hola
```

Module1,UseModule1.js  
Module2,UseModule2.js



# Create Your Own Modules

```
//module4.js : exporting result of a function that takes args
exports.add = function() {
    var sum = 0, i = 0, args = arguments, l = args.length;
    while (i < l) {
        sum += args[i++];
    }
    return sum;
};
```

```
//UseModule4.js
var result=require("./module4").add(10,20,30,40);
console.log(result);
```

This is equivalent to:

```
var addNumbers=require("./module4");
console.log(addNumbers.add(10,20,30,40));
```

Module4,UseModule4.js





# Node.js Module

- Node.js includes three types of modules:
  - Core Modules
  - Local Modules
  - Third Party Modules

# Node.js Core Module

- Unlike other programming technologies, Node.js doesn't come with a heavy standard library. The core modules of node.js are a bare minimum, and the rest can be cherry-picked via the NPM registry.
  - In order to use Node.js core or NPM modules, you first need to import it using `require()` function: `var module = require('module_name');`
- Some of the important core modules in Node.js

Core Module	Description
<a href="#"><u>http</u></a>	http module includes classes, methods and events to create Node.js http server.
<a href="#"><u>url</u></a>	url module includes methods for URL resolution and parsing.
<a href="#"><u>querystring</u></a>	querystring module includes methods to deal with query string.
<a href="#"><u>path</u></a>	path module includes methods to deal with file paths.
<a href="#"><u>fs</u></a>	fs module includes classes, methods & events to work with file I/O.
<a href="#"><u>util</u></a>	util module includes utility functions useful for programmers.

# List of the built-in modules of Node.js version 6.10.3:

Module	Description
<a href="#">assert</a>	Provides a set of assertion tests
<a href="#">buffer</a>	To handle binary data
<a href="#">child_process</a>	To run a child process
<a href="#">cluster</a>	To split a single Node process into multiple processes
<a href="#">crypto</a>	To handle OpenSSL cryptographic functions
<a href="#">dgram</a>	Provides implementation of UDP datagram sockets
<a href="#">dns</a>	To do DNS lookups and name resolution functions
<a href="#">domain</a>	Deprecated. To handle unhandled errors
<a href="#">events</a>	To handle events
<a href="#">fs</a>	To handle the file system
<a href="#">http</a>	To make Node.js act as an HTTP server
<a href="#">https</a>	To make Node.js act as an HTTPS server.

Module	Description
<a href="#">net</a>	To create servers and clients
<a href="#">os</a>	Provides information about the operation system
<a href="#">path</a>	To handle file paths
<a href="#">punycode</a>	Deprecated. A character encoding scheme
<a href="#">querystring</a>	To handle URL query strings
<a href="#">readline</a>	To handle readable streams one line at the time
<a href="#">stream</a>	To handle streaming data
<a href="#">string_decoder</a>	To decode buffer objects into strings
<a href="#">timers</a>	To execute a function after a given number of milliseconds
<a href="#">tls</a>	To implement TLS and SSL protocols
<a href="#">tty</a>	Provides classes used by a text terminal

# Node.js Module

- There are number of utility modules available in Node.js module library.
  - These are very common and frequently used while developing Node based apps

OS Module	Provides basic operating-system related utility functions
Path Module	Provides utilities for handling and transforming file paths
Net Module	Provides both servers and clients as streams. Acts as a network wrapper
DNS Module	Provides functions to do actual DNS lookup as well as to use underlying operating system name resolution functionalities
Domain Module	Provides way to handle multiple different I/O operations as a single group

# Node.js Local Module

- Node.js Local Module : are modules created locally in your Node.js app
  - Can also be packaged & distributed via NPM, so that Node.js community can use it
  - Eg, if you need to connect to MongoDB and fetch data then you can create a module for it, which can be reused in your application.

```
//Log.js
var log = {
  info: function (info) {
    console.log('Info: ' + info);
  },
  warning: function (warning) {
    console.log('Warning: ' + warning);
  },
  error: function (error) {
    console.log('Error: ' + error);
  }
};
module.exports = log
```

```
//app.js
var myLogModule = require('./Log.js');
myLogModule.info('Node.js started');
```

```
C:\> node app.js
Info: Node.js started
```

Module3,UseModule3.js  
Module3a,UseModule3a.js



# Third Party Modules

- Node.js also has the ability to embedded external functionality or extended functionality by making use of custom modules.
- These modules have to be installed separately (using NPM)
- An example of a module is the MongoDB module which allows you to work with MongoDB databases from your Node.js application.

# NPM (Node Package Manager)

- Loading a module(Third party) installed via NPM
  - Apart from writing our own modules and core modules, we will frequently use the modules written by other people in the Node community and published on the Internet (npmjs.com).
  - We can install those third party modules using the **Node Package Manager** which is installed by default with the node installation.
- NPM is a command line tool that installs, updates or uninstalls Node.js packages in your application.
  - NPM is included with Node.js installation.
  - After you install Node.js, verify NPM installation
  - To access NPM help, write **npm help** in the command prompt
  - npm manages Node modules and their dependencies

```
C:\Users\DELL>npm -v  
2.14.20
```

# NPM

- **Installing a package using NPM**
  - `$ npm install <Package Unique Name>`
- **Installing a module globally**
  - `$ npm install -g <Package Unique Name>`
- **To remove an installed package**
  - `npm uninstall < Package Unique Name>`
- **To remove a global package**
  - `npm uninstall -g < Package Unique Name>`
- **To update a package to its latest version**
  - `npm update < Package Unique Name>`
- **To update a global package**
  - `npm update -g < Package Unique Name>`



# Buffers

- JavaScript doesn't have a byte type. It just has strings.
- Node is based on JavaScript, but with just string type it is very difficult to perform the operations like communicate with HTTP protocol, working with databases, manipulate images and handle file uploads.
- Node includes a binary buffer implementation, which is exposed as a JavaScript API under the Buffer pseudo-class.
- Using buffers we can manipulate, encode, and decode binary data in Node.
- Buffer class provides instances to store raw data similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap.

# Buffers

- Buffer class is a global class and can be accessed in application without importing buffer module.
- Buffer can be constructed in a variety of ways:
  - `var buf = new Buffer(10);` //create an uninitiated Buffer of 10 bytes
  - `var buf = new Buffer([10, 20, 30, 40, 50]);` //create a Buffer from a given array
  - `var buf = new Buffer("Node JS", "utf-8");` //create a Buffer from a given string and optionally encoding type. utf-8 is the default encoding in Node.

```
> var buff = new Buffer(5)
undefined
> console.log(buff)
<Buffer d0 07 3a 00 00>
undefined
> var buff = new Buffer([1,2,3,4,5])
undefined
> console.log(buff)
<Buffer 01 02 03 04 05>
undefined
> var buff = new Buffer('training')
undefined
> console.log(buff)
<Buffer 74 72 61 69 6e 69 6e 67>
undefined
>
```

# Buffers

- To write into a Node Buffer:  
`buf.write(string[, offset][, length][, encoding])`
  - string - string data to be written to buffer.
  - offset - index of the buffer to start writing at. Default is 0.
  - length - number of bytes to write.  
Defaults to buffer.length
  - encoding - encoding to use.  
'utf8' is the default encoding

```
> var buff = new Buffer(10)
undefined
> buff.write('Node JS')
7
> buff.write('Node', 3, 'utf-8')
4
```

- `buf.length` : Returns the size of the buffer in bytes

# Various String Encoding Types Supported by Node

Encoding Type	Description
<code>utf8</code>	Multibyte encoded Unicode characters. UTF-8 encoding is used by many web pages and to represent string data in Node.
<code>ascii</code>	7-bit American Standard Code for Information Interchange (ASCII) encoding.
<code>utf16le</code>	Little-endian-encoded Unicode characters. Each character is 2 or 4 bytes.
<code>ucs2</code>	This is simply an alias for <code>utf16le</code> encoding.
<code>base64</code>	Base64 string encoding. Base64 is commonly used in URL encoding, e-mail, and similar applications.
<code>binary</code>	Allows binary data to be encoded as string using only the first 8 bits of each character. As this is deprecated in favor of the <code>Buffer</code> object, it will be removed in future versions of Node.
<code>hex</code>	Encodes each byte as two hexadecimal characters.

# Buffers

- To read data from a Node Buffer:
  - `buf.toString([encoding], [start], [end])` method decodes and returns a string from buffer data.
  - `buf.toString()` returns method reads the entire buffer and returns as a string.
  - `buf.toJSON()` method is used to get the JSON-representation of the Buffer instance, which is identical to the output for JSON Arrays.

```
buf = new Buffer(26);
for (var i = 0 ; i < 26 ; i++) {
  buf[i] = i + 97;
}
console.log( buf.toString('ascii'));
console.log( buf.toString('utf8',0,5));
console.log( buf.toString('utf8',5,8));
var buf1= new Buffer('Node');
console.log(buf1.toJSON());
```

```
abcdefghijklmnopqrstuvwxyz
```

```
abcde
```

```
fgh
```

```
{ type: 'Buffer', data: [ 78, 111, 100, 101 ] }
```

# Slicing and copying a buffer

- Slicing a buffer

- `buffer.slice([start],[end])` : We can slice a buffer and extract a portion from it, to create another smaller buffer by specifying the starting and ending positions.

```
var buf2= new Buffer('Hello All');  
var sub = buf2.slice(6,9);  
console.log(buf2.toString());  
console.log(sub.toString());
```

- Copying a buffer

- `buffer.copy(targetBuffer, [targetStart], [sourceStart], [sourceEnd])` : It is used to copy the contents of one buffer onto another

```
var buf2= new Buffer('Hello All');  
var tgtbuf = new Buffer(5);  
buf2.copy(tgtbuf,0,0,5);  
console.log(buf2.toString());  
console.log(tgtbuf.toString());
```

# EVENT HANDLING IN NODE

---

# Asynchronous Programming

```
$file = fopen('info.txt', 'r');  
// wait until file is open  
$contents = fread($file, 100000);  
  
// wait until contents are read  
// do something with those contents
```

```
setTimeout(function () {  
    console.log("I've done my work!");  
}, 2000);  
console.log("Waiting for my work to finish.");
```

Waiting for my work to finish.  
I've done my work!

- Callback is an asynchronous equivalent for a function.
  - A callback function is called at the completion of a given task
  - All APIs of Node are written in such a way that they support callbacks.
  - Two rules for defining an error-first callback:
    - The first argument of the callback is reserved for an error object, if an error occurred
    - The second argument is reserved for any successful response data. If no error, err will be set to null and any successful data will be returned in the second argument.

```
fs.readFile('/foo.txt', function(err, data) {  
    if (err) { // If an error occurred, handle it (throw, propagate, etc)  
        console.log("ERROR: " + err.code + " (" + err.message + ")");  
    } else { // success! continue working here  
    }  
});
```



# The Old Way of Doing Things

```
$file = fopen('info.txt', 'r');  
// wait until file is open  
  
$contents = fread($file, 100000);  
// wait until contents are read  
  
// do something with those contents
```

- If you were to analyze the execution of this script, you would find that it spends a vast majority of its time *doing nothing at all*.
  - *Indeed, most of the clock time taken by this script is spent* waiting for the computer's file system to do its job and return the file contents you requested.
  - To generalize things a step further - for most IO-based applications—those that frequently connect to databases, communicate with external servers, or read and write files—your scripts will spend a majority of their time sitting around waiting

# The Node.js Way of Doing Things

- The `setTimeout()` function in JavaScript- takes a function to call and a timeout after which it should be called:

```
setTimeout(function () {  
    console.log("I've done my work!");  
}, 2000);
```

```
console.log("I'm waiting for all my work to finish.");
```

If you run the preceding code, you see the following output:

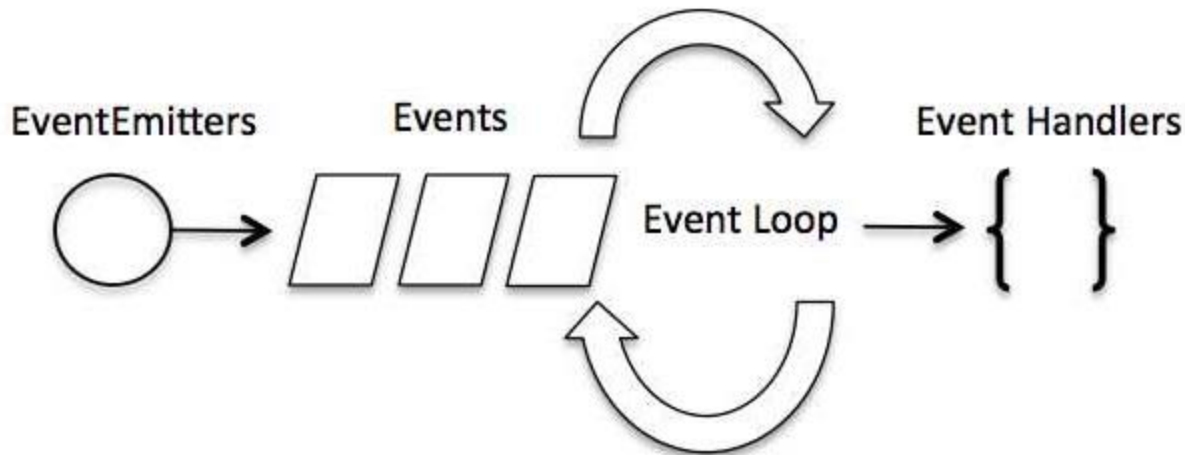
```
I'm waiting for all my work to finish.
```

```
I've done my work!
```

- Pending events for which you are awaiting a response, are put on a **event queue**.
- Eg, The `setTimeout()` function just adds an event to the Node event queue (instructing it to call the provided function after the specified time interval—2000ms in this example), and then returns

# Node.js Events

- Node.js uses events heavily and it is also one of the reasons why Node.js is pretty fast compared to other similar technologies.
- 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.



# Handling repeating events with event emitters

- Event emitters fire events and include the ability to handle those events when triggered.
- Some important Node API components, such as HTTP servers, TCP servers, and streams, are implemented as event emitters.
- **You can also create your own.**
- Events are handled through the use of listeners.
  - A *listener* is the association of an event with a callback function that gets triggered each time the event occurs.
  - For example, a TCP socket in Node has an event called data that's triggered whenever new data is available on the socket:
    - **`socket.on('data', handleData);`**
  - The *on* or *addListener* method allows us to subscribe the callback to the event.

# EventEmitter

- Node.js has multiple in-built events available through events module and EventEmitter class which are used to bind events and event-listeners
- All objects which emit events in node are instances of **events.EventEmitter** which is available inside Event module.
- We can access the Event module using `require("events")`

```
// Import events module  
var events = require('events');
```

```
// Create an EventEmitter object  
var EventEmitter = new events.EventEmitter();
```

Following is the syntax to bind an event handler with an event –

```
// Bind event and event handler as follows  
eventEmitter.on('eventName', eventHandler);
```

We can fire an event programmatically as follows –

```
// Fire an event  
eventEmitter.emit('eventName');
```

```
var events = require('events');    // Import events module
var EventEmitter = new events.EventEmitter(); // Create an EventEmitter object

// Create an event handler as follows
var connectHandler = function connected() {
    console.log('connection succesful.');
```

    // Fire the data\_received event  
    eventEmitter.emit('data\_received');

```
}

// Bind the connection event with the handler
eventEmitter.on('connection', connectHandler);

// Bind the data_received event with the anonymous function
eventEmitter.on('data_received', function(){
    console.log('data received succesfully.');
```

});

```
// Fire the connection event
eventEmitter.emit('connection');
```

console.log("Program Ended.");

**EVENT NAMES** Events are simply keys and can have any string value: data, join, or some crazy long event name except for a one special event, called error

SimpleEventEmitterEg1.js  
SimpleEventEmitterEg2.js



# File System Module

- By default Node.js installations come with the file system module.
- This module provides a wrapper for the standard file I/O operations.
- We can access the file system module using `require("fs")`
- All the methods in this module has asynchronous and synchronous forms.
- synchronous methods in this module ends with 'Sync'. For instance `renameSync()` is the synchronous method for rename synchronous method.
- The asynchronous form always take a completion callback as its last argument.
  - The arguments passed to the completion callback depend on the method, but the first argument is always reserved for an exception. If the operation was completed successfully, then the first argument will be null or undefined.
- When using the synchronous form any exceptions are immediately thrown. You can use try/catch to handle exceptions or allow them to bubble up.

# Node.js File System

- Node JS fs module provides an API to interact with FileSystem and to perform some IO Operations like create a file, read a File, delete a File etc..
  - fs module is responsible for all the **async or synchronous** file I/O operations.

```
var fs = require('fs');  
// write  
fs.writeFileSync('test.txt', 'Hello fs!');  
// read  
console.log(fs.readFileSync('test.txt').toString()); //prints Hello fs!
```

```
var fs = require("fs");  
fs.readFile('test.txt', function (err, data) { // Asynchronous read  
  if (err)  
    return console.error(err);  
  console.log("Asynchronous read: " + data.toString());  
});  
var data = fs.readFileSync('test.txt'); // Synchronous read  
console.log("Synchronous read: " + data.toString());  
  
console.log("Program Ended");
```

```
Synchronous read: Hello fs!  
Program Ended  
Asynchronous read: Hello fs!
```



Filesystem/module1.js  
Filesystem/module2.js



# Node.js File System

- **fs.open**(path, flags[, mode], callback) : opens a file in asynchronous mode:
  - path - string having file name including path.
  - flags - tells the behavior of the file to be opened
  - mode - sets the file mode; defaults to 0666, readable and writeable.
  - callback - function which gets two arguments (err, fd).
- **fs.readFile**(fileName [,options], callback) : read the physical file asynchronously.
- **fs.writeFile**(filename, data[, options], callback) : writes data to a file
  - If file already exists then it overwrites the existing content otherwise it creates a new file and writes data into it.
  - Data: String or Buffer to be written into the file.
- **fs.appendFile()**: appends the content to an existing file
- **fs.unlink**(path, callback); delete an existing file
- **fs.exists**(path, callback) : determines if specified file exists or not

## *Various Flags Available to open()*

Flags	Description
r	Open for reading. An exception occurs if the file does not exist.
r+	Open for reading and writing. An exception occurs if the file does not exist.
rs	Open for reading in synchronous mode. This instructs the operating system to bypass the system cache. This is mostly used for opening files on NFS mounts. This does <i>not</i> make <code>open()</code> a synchronous method.
rs+	Open for reading and writing in synchronous mode.
w	Open for writing. If the file does not exist, it is created. If the file already exists, it is truncated.
wx	Similar to the <code>w</code> flag, but the file is opened in exclusive mode. Exclusive mode ensures that the file is newly created.
w+	Open for reading and writing. If the file does not exist, it is created. If the file already exists, it is truncated.
wx+	Similar to the <code>w+</code> flag, but the file is opened in exclusive mode.
a	Open for appending. If the file does not exist, it is created.
ax	Similar to the <code>a</code> flag, but the file is opened in exclusive mode.
a+	Open for reading and appending. If the file does not exist, it is created.
ax+	Similar to the <code>a+</code> flag, but the file is opened in exclusive mode.

# Opening Files


```
var fs = require("fs");
var path = "/path-to-file";
fs.open(path, "w+", function(error, fd) {
  if (error) {
    console.error("open error: " + error.message);
  } else {
    console.log("Successfully opened " + path);
  }
});
```

- It is advisable to use a file path that does not currently exist, as the contents of an existing file, as in this example, are overwritten

# File System

- To read an opened file : `fs.read(fd, buffer, offset, length, position, callback)`
  - `fd <Integer>` - is a file descriptor, a handle, to the file
  - `buffer <String> | <Buffer>` : buffer into which the data will be read
  - `offset <Integer>` : offset in the buffer to start writing at.
  - `length <Integer>` : specifies the number of bytes to read
  - `position <Integer>` : specifies where to begin reading from in the file
  - `callback <Function>`
- `fs.stat(path, callback)` : gets the information about a file. Methods of **fs.Stats** class:

callback function  
gets two arguments  
(err, stats) where  
stats is an object of  
fs.Stats type



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

# File System : example

```
var fs = require("fs");
console.log("Going to get file info!");
fs.stat('input.txt', function (err, stats) {
  if (err) {
    return console.error(err);
  }
  console.log(stats);
  console.log("Got file info successfully!");

  // Check file type
  console.log("isFile ? " + stats.isFile());
  console.log("isDirectory ? " + stats.isDirectory());
});
```

```
Going to get file info!
Stats {
  dev: 1281842521,
  mode: 33206,
  nlink: 1,
  uid: 0,
  gid: 0,
  rdev: 0,
  blksize: undefined,
  ino: 844424930151342,
  size: 9,
  blocks: undefined,
  atime: 2017-06-27T14:04:13.416Z,
  mtime: 2017-10-30T16:01:28.181Z,
  ctime: 2017-10-30T16:01:28.181Z,
  birthtime: 2017-06-27T14:04:13.416Z }
Got file info successfully!
isFile ? true
isDirectory ? false
```



DEMO

Filesystem/module5.js

Going to get file info!

```
Stats {
  dev: 1281842521,
  mode: 33206,
  nlink: 1,
  uid: 0,
  gid: 0,
  rdev: 0,
  blksize: undefined,
  ino: 844424930151342,
  size: 9,
  blocks: undefined,
  atime: 2017-06-27T14:04:13.41Z,
  mtime: 2017-10-30T16:01:28.18Z,
  ctime: 2017-10-30T16:01:28.18Z,
  birthtime: 2017-06-27T14:04:13.41Z
}
```

Got file info successfully!

isFile ? true

isDirectory ? false

Property	Description
dev	ID of the device containing the file.
mode	The file's protection.
nlink	The number of hard links to the file.
uid	User ID of the file's owner.
gid	Group ID of the file's owner.
rdev	The device ID, if the file is a special file.
blksize	The block size for file system I/O.
ino	The file's inode number. An inode is a file system data structure.
size	The file's total size in bytes.
blocks	The number of blocks allocated for the file.
atime	Date object representing the file's last access time.
mtime	Date object representing the file's last modification time.
ctime	Date object representing the last time the file's inode was changed.

# Determining if a File Exists

- The `exists()` and `existsSync()` methods are used to determine if a given path exists.
- Both methods take a path string as an argument.
- If the synchronous version is used, a Boolean value representing the path's existence is returned.
- If the asynchronous version is used, the same Boolean value is passed as an argument to the callback function.

```
var fs = require("fs");
var path = "/";
var existsSync = fs.existsSync(path);

fs.exists(path, function(exists) {
  if (exists !== existsSync) {
    console.error("Something is wrong!");
  } else {
    console.log(path + " exists: " + exists);
  }
});
```

# File System : example

```
var fs = require("fs");
var fileName = "foo.txt";

fs.exists(fileName, function(exists) {
  if (exists) {
    fs.stat(fileName, function(error, stats) {
      fs.open(fileName, "r", function(error, fd) {
        var buffer = new Buffer(stats.size);

        fs.read(fd, buffer, 0, buffer.length, null, function(error, bytesRead, buffer) {
          var data = buffer.toString("utf8", 0, buffer.length);

          console.log(data);
          fs.close(fd);
        });
      });
    });
  }
});
```



DEMO

Filesystem/module4.js



# File I/O methods

- `fs.stat(path, callback)`
  - Used to retrieve meta-info on a file or directory.
- `fs.readFile(filename, [options], callback)`
  - Asynchronously reads the entire contents of a file.
- `fs.writeFile(filename, data, [options], callback)`
  - Asynchronously writes data to a file, replacing the file if it already exists. Data can be a string or a buffer.
- `fs.unlink(path, callback)`
  - Asynchronously deletes a file.
- `fs.watchFile(filename, [options], listener)`
  - Watch for changes on filename. The callback listener will be called each time the file is accessed. Second argument is optional by default it is { persistent: true, interval: 5007 }. The listener gets two arguments the current stat object and the previous stat object.

# File I/O methods

- `fs.exists(path, callback)`
  - Test whether or not the given path exists by checking with the file system. The callback argument assigned with either true or false based on the existence.
- `fs.rmdir(path, callback)`
  - Asynchronously removes the directory.
- `fs.mkdir(path, [mode], callback)`
  - Asynchronously created the directory.
- `fs.open(path, flags, [mode], callback)`
  - Asynchronously open the file.
- `fs.close(fd, callback)`
  - Asynchronously closes the file.
- `fs.read(fd, buffer, offset, length, position, callback)`
  - Read data from the file specified by fd.

# Web development with Node

- Webserver like IIS / Apache serves static files(like html files) so that a browser can view them over the network.
- We need to place the files in a proper directory(like wwwroot in IIS), so that we can navigate to it using http protocol. The web server simply knows where the file is on the computer and serves it to the browser.
- Node offers a different paradigm than that of a traditional web server i.e. it simply provides the framework to build a web server.
- Interestingly building a webserver in node is not a cumbersome process, it can be written in just a few lines, moreover we'll have full control over the application.

# HTTP module in Node.js

- We can easily create an HTTP server in Node.
- To use the HTTP server and client one must `require('http')`.
- The HTTP interfaces in Node are designed to support many features of the protocol which have been traditionally difficult to use. In particular, large, possibly chunk-encoded, messages.
- Node's HTTP API is very low-level. It deals with stream handling and message parsing only. It parses a message into headers and body but it does not parse the actual headers or the body.
- HTTP response implements the Writable Stream interface and request implements Readable Stream interface.

# How Node presents incoming HTTP requests

- To create an HTTP server, call the `http.createServer()` function.
  - It accepts a single argument, a callback function, that will be called on each HTTP request received by the server.
  - This request callback receives, as arguments, the request and response objects, which are commonly shortened to `req` and `res`

```
var http = require('http');  
var server =  
    http.createServer(function(req, res){  
        // handle request  
    });
```

```
var http = require('http');  
var server = http.createServer();  
server.on('request', function(req, res) {  
    // handle request  
});
```

- For every HTTP request received by the server, the request callback function will be invoked with new `req` and `res` objects.
- Node will not automatically write any response back to the client. After the request callback is triggered, it's your responsibility to end the response using the `res.end()` method; if you fail to end the response, the request will hang until the client times out or it will just remain open.

## Example

```
var http = require('http');
var server = http.createServer(function(req, res){
    res.write('Hello World');
    res.end();
});
```

- As shorthand, `res.write()` and `res.end()` can be combined into one statement, which can be nice for small responses:
  - `res.end('Hello World');`
- You can use existing buffer to write response:
  - `var buffer = new Buffer('Hello World');`
  - `res.write(buffer);`
- The last thing you need to do is bind to a port so you can listen for incoming requests.
  - You do this by using the `server.listen()` method, which accepts a combination of arguments
  - `server.listen(3000);`
    - With Node now listening for connections on port 3000, you can visit `http://localhost:3000` in your browser. When you do, you should receive a plain-text age consisting of the words “Hello World.”

# Setting response headers and status code

- You can explicitly queue any HTTP header in the response using `response.setHeader(name, value)`
  - You can add and remove headers in any order, but only up to the first `res.write()` or `res.end()` call. After the first part of the response body is written, Node will flush the HTTP headers that have been set.
  - When you want to explicitly send the headers (not just queue them) call `resp.writeHead()`. Eg `resp.writeHead(200, { 'Content-Type': 'text/html' });`

```
var body = 'Hello World';
res.setHeader('Content-Length', body.length);
res.setHeader('Content-Type', 'text/plain');
res.end(body);
```

```
var url = 'http://google.com';
var body = '<p>Redirecting to <a href="' + url + '">' + url + '</a></p>';
res.setHeader('Location', url);
res.setHeader('Content-Length', body.length);
res.setHeader('Content-Type', 'text/html');
res.statusCode = 302;
res.end(body);
```

# Creating HTTP Server

```
/* Loading http module*/  
var http = require('http');  
  
/* Returns a new web server object*/  
var server = http.createServer(function(req,res){  
  
    /* Sends a response header to the request.*/  
    res.writeHead(200,{ 'content-type': 'text/html' });  
  
    /*sends a chunk of the response body*/  
    res.write('<h1>Hello Node!</h1>')  
  
    /* signals server that all the responses has been sent */  
    res.end('<b>Response Ended</b>')  
});  
  
/* Accepting connections on the specified port and hostname. */  
server.listen(3000);  
  
console.log('server listening on localhost:3000');
```





# http.ServerRequest

- When listening for request events, the callback gets an `http.ServerRequest` object as the first argument (`function(req,res)`)
- This object contains some properties:
  - `req.url`: This property contains the requested URL as a string
    - It does not contain the schema, hostname, or port, but it contains everything after that. Eg : if URL is `:http://localhost:3000/about?a=20` then `req.url` will return `/about?a=20`
  - `req.method`: This contains the HTTP method used on the request. It can be, for example, GET, POST, DELETE, or HEAD.
  - `req.headers`: This contains an object with a property for every HTTP header on the request.



```
var server = http.createServer(function (req, res) {  
  console.log('request headers...');  
  console.log(req.headers);
```

```
  res.end('hello client!');  
}).listen(3000);
```

`httpmodule/reqheaders.js`

```
request headers...  
{ host: 'localhost:3000',  
  connection: 'keep-alive',  
  'user-agent': 'Mozilla/5.0 (Windows NT  
TML, like Gecko) Chrome/61.0.3163.100 Saf  
'upgrade-insecure-requests': '1',  
  accept: 'text/html,application/xhtml+xml  
e/apng,*/*;q=0.8',  
  'accept-encoding': 'gzip, deflate, br',  
  'accept-language': 'en-US,en;q=0.8' }
```

# Serving file on request

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

function send404(response) {
  response.writeHead(404, { 'Content-Type': 'text/plain' });
  response.write('Error 404: Resource not found. ');
  response.end();
}

var server = http.createServer(function (req, res) {
  if (req.method === 'GET' && req.url === '/') {
    res.writeHead(200, { 'content-type': 'text/html' });
    fs.createReadStream('./public/index.html').pipe(res);
  }
  else {
    send404(res);
  }
}).listen(3000);
console.log('server running on port 3000');
```

```
<html>
<head>
<title>Hello there</title>
</head>
<body>
<h3>Welcome to Node!
Serving your file now</h3>
</body>
</html>
```



# Routing

- Routing refers to the mechanism for serving the client the content it has asked

```
var http = require('http');
var server = http.createServer(function(req,res){
var path = req.url.replace(/^?(?:\?.*)?$/, '').toLowerCase();
switch(path) {
    case "":
        res.writeHead(200, {'Content-Type': 'text/html'});
        res.end('<h1>Home Page</h1>');
        break;
    case '/about':
        res.writeHead(200, {'Content-Type': 'text/html'});
        res.end('<h1>About us</h1>');
        break;
    default:
        res.writeHead(404, { 'Content-Type': 'text/plain' });
        res.end('Not Found');
        break;
}
});
server.listen(3000);
```



# HTTP Properties and Methods

- `http.STATUS_CODES`
  - Returns object which is a collection of all the standard HTTP response status codes and its short descriptions
- `http.createServer([requestListener])`
  - Returns a new web server object.
- `http.request(options, [callback])`
  - Returns a new web server object.
- `http.get(options, [callback])`
  - Returns a new web server object.
- `server.listen(port, [hostname], [backlog], [callback])`
  - Begin accepting connections on the specified port and hostname.

# HTTP Properties and Methods

- `server.close([callback])`
  - Stops the server from accepting new connections
- `response.writeHead(statusCode, [reasonPhrase], [headers])`
  - Sends a response header to the request.
- `response.setHeader(name, value)`
  - Sets a single header value for implicit headers.
- `response.write(chunk, [encoding])`
  - This sends a chunk of the response body. This method may be called multiple times to provide successive parts of the body.

# HTTP Events

- request
  - Emitted each time there is a request.
- close
  - Emitted when the server closes.
- connect
  - Emitted each time a client requests a http CONNECT method
- finish
  - Emitted when the response has been sent.

- http is the main module responsible for the Node.js HTTP server. The main methods are as follows:
  - `http.createServer()`: returns a new web server object
  - `http.listen()`: begins accepting connections on the specified port and hostname
  - `http.createClient()`: is a client and makes requests to other servers
  - `http.ServerRequest()`: passes incoming requests to request handlers
    - **data: emitted when a part of the message body is received**
    - **end: emitted exactly once for each request**
    - `request.method()`: the request method as a string
    - `request.url()`: request URL string
  - `http.ServerResponse()`: creates this object internally by an HTTP server — not by the user— and is used as an output of request handlers
    - `response.writeHead()`: sends a response header to the request
    - `response.write()`: sends a response body
    - `response.end()`: sends and ends a response body