Nodejs

- Introduction
- Modules
- HTTP Module
- File System
- URL Module
- Uploading Files
- Event Loop
- Event Emitter
- Callback’s Concept, Buffers, Streams, Nodejs Email, Packaging

What is Node.js?

- Node.js is an open source server environment
- Node.js is free
- Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)
- Node.js uses JavaScript on the server
- Node.js also provides a rich library of various JavaScript modules which simplifies the development of web applications using Node.js to a great extent.

Node.js = Runtime Environment + JavaScript Library

Why Node.js?

- **Node.js uses asynchronous programming!**
- A common task for a web server can be to open a file on the server and return the content to the client.

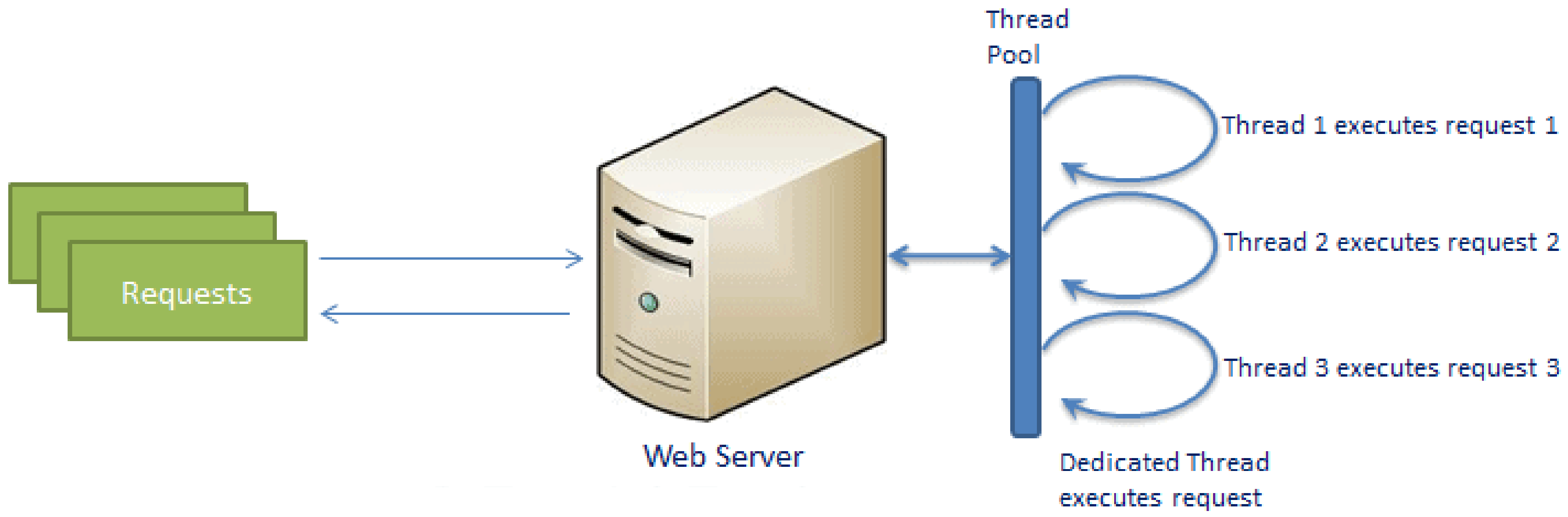
Here is how PHP or ASP handles a file request:

- Sends the task to the computer's file system.
- Waits while the file system opens and reads the file.
- Returns the content to the client.
- Ready to handle the next request.

Here is how Node.js handles a file request:

- Sends the task to the computer's file system.
- Ready to handle the next request.
- When the file system has opened and read the file, the server returns the content to the client.
- Node.js eliminates the waiting, and simply continues with the next request.
- Node.js runs single-threaded, non-blocking, asynchronously programming, which is very memory efficient.

Why Node.js?


Requests
Web Server
Thread Pool
Thread 1 executes request 1
Thread 2 executes request 2
Thread 3 executes request 3
Dedicated Thread executes request

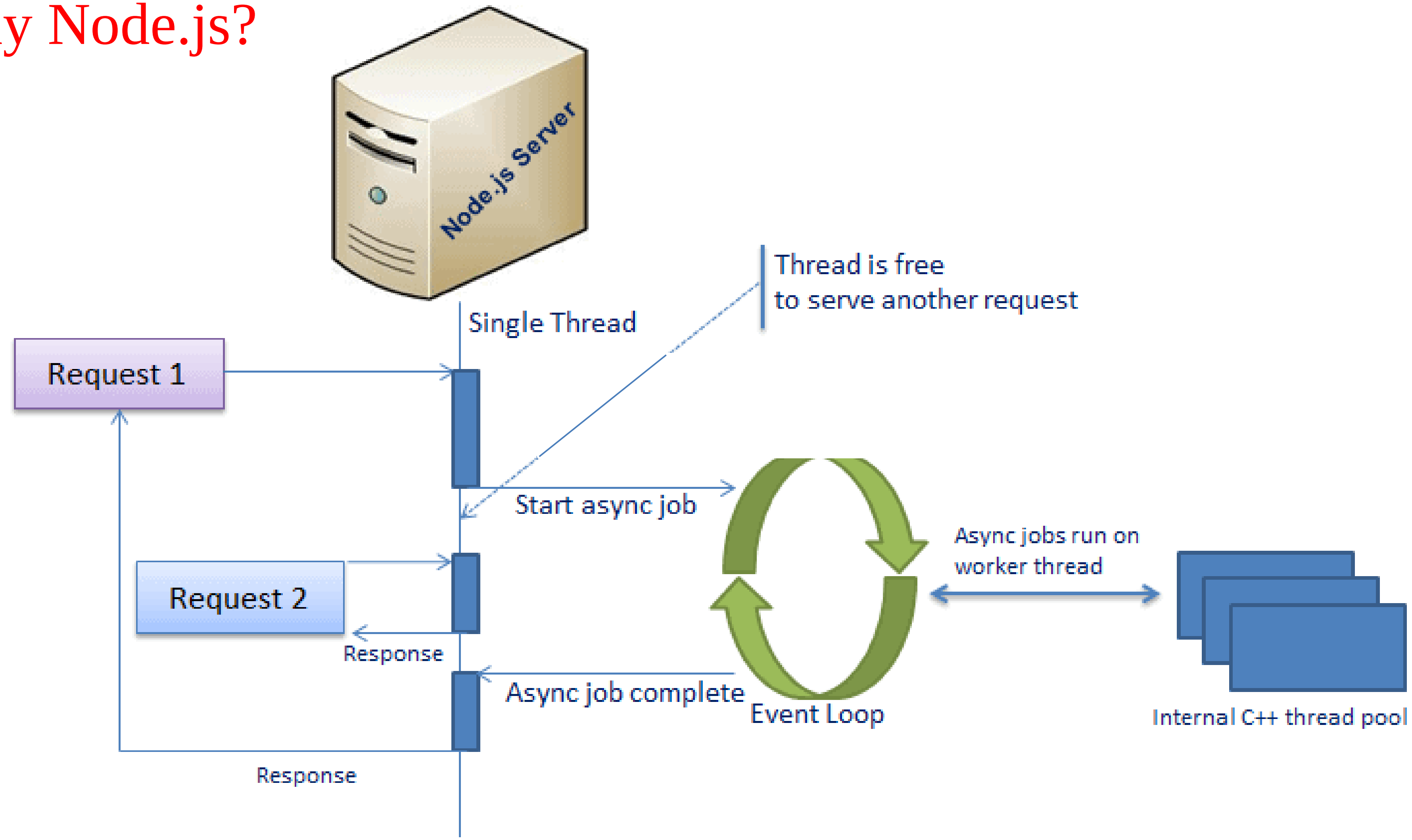
Traditional Web Server Model

- In the traditional web server model, each request is handled by a dedicated thread from the thread pool.

- If no thread is available in the thread pool at any point of time then the request waits till the next available thread.

- Dedicated thread executes a particular request and does not return to thread pool until it completes the execution and returns a response.

Why Node.js?



- Node.js process model increases the performance and scalability with a few caveats.
- 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.

Why Node.js?

Node.js Process Model

- Node.js processes user requests differently when compared to a traditional web server model.

- Node.js runs in a single process and the application code runs in a single thread and thereby needs less resources than other platforms.

- All the user requests to your web application will be handled by a single thread and all the I/O work or long running job is performed asynchronously for a particular request.

- So, this single thread doesn't have to wait for the request to complete and is free to handle the next request. When asynchronous I/O work completes then it processes the request further and sends the response.

- An event loop is constantly watching for the events to be raised for an asynchronous job and executing callback function when the job completes.

- Internally, Node.js uses libev for the event loop which in turn uses internal C++ thread pool to provide asynchronous I/O.

Node.js

What Can Node.js Do?

- Node.js can generate dynamic page content
- Node.js can create, open, read, write, delete, and close files on the server
- Node.js can collect form data
- Node.js can add, delete, modify data in your database

What is a Node.js File?

- Node.js files contain tasks that will be executed on certain events
- A typical event is someone trying to access a port on the server
- Node.js files must be initiated on the server before having any effect
- Node.js files have extension ".js"

Features of Node.js

- Asynchronous and Event Driven
- Very Fast
- Single Threaded but Highly Scalable
- No Buffering

Features of Node.js

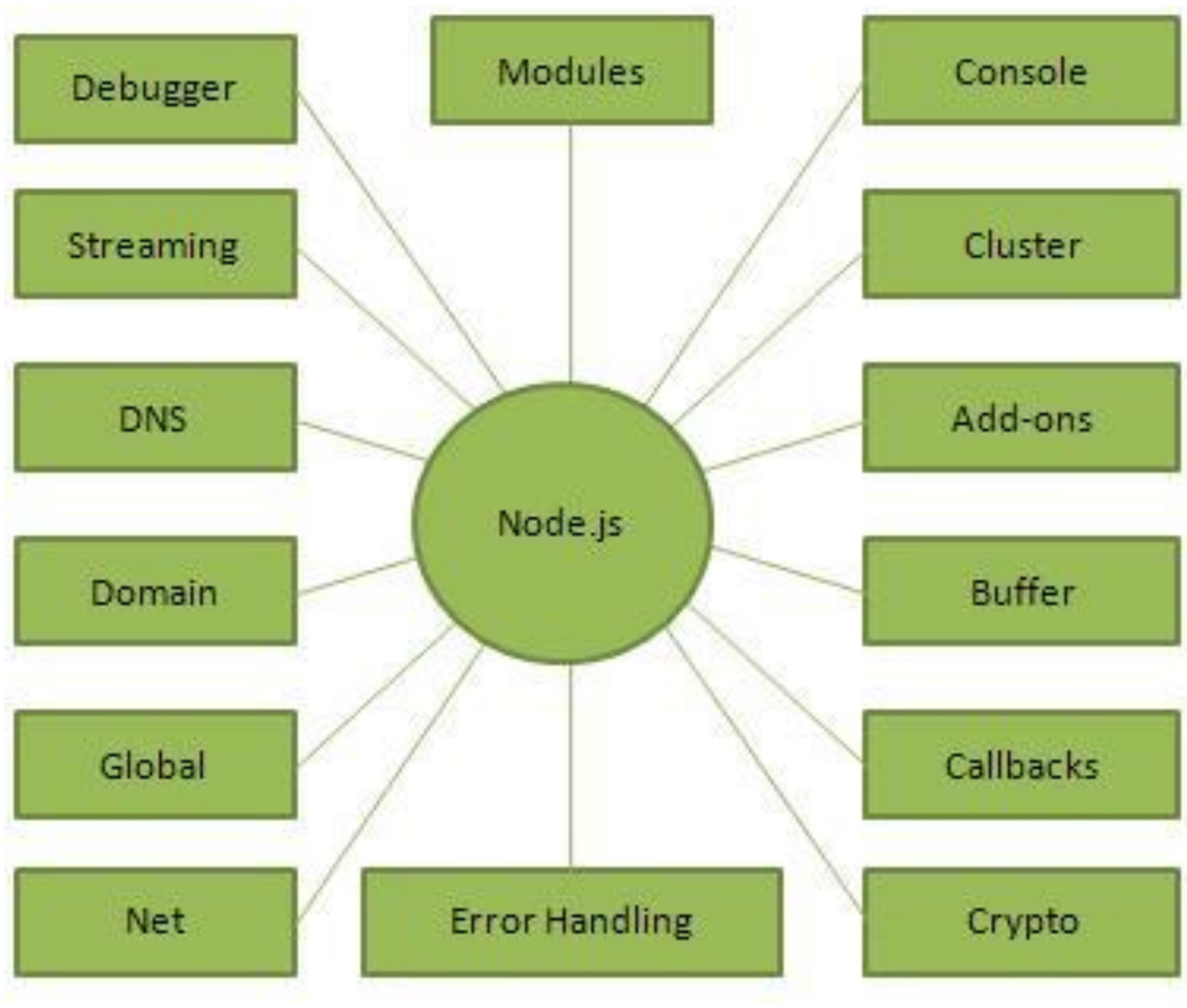
- **Asynchronous and Event Driven** − All APIs of Node.js library are asynchronous, that is, non-blocking. It essentially means a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call.

- **Very Fast** − Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.

- **Single Threaded but Highly Scalable** − Node.js uses a single threaded model with event looping. 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.

- **No Buffering** − Node.js applications never buffer any data. These applications simply output the data in chunks.

Concepts of Node.js



Where to Use Node.js?

Following are the areas where Node.js is proving itself as a perfect technology partner.

- I/O bound Applications
- Data Streaming Applications
- Data Intensive Real-time Applications (DIRT)
- JSON APIs based Applications
- Single Page Applications

Node.js Basics

Primitive Types

Node.js includes following primitive types:

- String
- Number
- Boolean
- Undefined
- Null
- RegExp

Everything else is an object in Node.js.

Loose Typing

- JavaScript in Node.js supports loose typing like the browser's JavaScript. Use var keyword to declare a variable of any type.

Node.js Basics

- Object Literal: Object literal syntax is same as browser's JavaScript.

Example:

```
var obj = {
    authorName: 'Ryan Dahl',
    language: 'Node.js'
}
```

- Functions

Functions are first class citizens in Node's JavaScript, similar to the browser's JavaScript. A function can have attributes and properties also. It can be treated like a class in JavaScript.

Example:

```
function Display(x)
{
    console.log(x);
}
Display(100);
```

Node.js Basics

- Buffer: Node.js includes an additional data type called Buffer (not available in browser's JavaScript).
- Buffer is mainly used to store binary data, while reading from a file or receiving packets over the network.
- Process object: 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.

Node.js Basics

REPL Command	Description
.help	Display help on all the commands
tab Keys	Display the list of all commands.
Up/Down Keys	See previous commands applied in REPL.
.save filename	Save current Node REPL session to a file.
.load filename	Load the specified file in the current Node REPL session.
ctrl + c	Terminate the current command.
ctrl + c (twice)	Exit from the REPL.
ctrl + d	Exit from the REPL.
.break	Exit from multiline expression.
.clear	Exit from multiline expression.

Node.js Modules

A Node.js application consists of the following three important components −

- **Import required modules** − We use the **require** directive to load Node.js modules.
- **Create server** − A server which will listen to client's requests similar to Apache HTTP Server.
- **Read request and return response** − The server created will read the HTTP request made by the client which can be a browser or a console and return the response.

Node.js Modules

- 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.

- 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.

- Node.js implements CommonJS modules standard. CommonJS is a group of volunteers who define JavaScript standards for web server, desktop, and console application.

Node.js Modules

Node.js includes three types of modules:

- Core Modules
- Local Modules
- Third Party Modules

Node.js Core Modules

- Node.js is a light weight framework. The core modules include bare minimum functionalities of Node.js.

- These core modules are compiled into its binary distribution and load automatically when Node.js process starts.

- However, you need to import the core module first in order to use it in your application.

Node.js Core Modules

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

Node.js Core Modules

Loading Core Modules

- In order to use Node.js core or NPM modules, you first need to import it using require() function as shown below.

```
var module = require('module_name');
```

- The require() function will return an object, function, property or any other JavaScript type, depending on what the specified module returns.

Node.js Core Modules

Loading Core Modules

- Example demonstrates how to use Node.js http module to create a web server.

```
var http = require('http');
var server = http.createServer(function(req, res)
{
  //write code here
});
server.listen(5000);
```

- require() function returns an object because http module returns its functionality as an object, you can then use its properties and methods using dot notation e.g. http.createServer().

Node.js Local Modules

Code Explanation:

- The basic functionality of the "require" function is that it reads a JavaScript file, executes the file, and then proceeds to return an object. Using this object, one can then use the various functionalities available in the module called by the require function. So in our case, since we want to use the functionality of HTTP and we are using the require(http) command.

- In this 2nd line of code, we are creating a server application which is based on a simple function. This function is called, whenever a request is made to our server application.

- When a request is received, we are asking our function to return a "Hello World" response to the client. The writeHead function is used to send header data to the client, and while the end function will close the connection to the client.

- We are then using the server.listen function to make our server application listen to client requests on port no 8080. You can specify any available port over here.

Node.js Local Modules

- Local modules are modules created locally in your Node.js application.
- These modules include different functionalities of your application in separate files and folders.
- You can also package it and distribute it via NPM, so that Node.js community can use it.
- For example, 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.

Node.js Local Modules

Writing Simple Module

- Let's write simple logging module which logs the information, warning or error to the console.

- In Node.js, module should be placed in a separate JavaScript file. So, create a Log.js file and write the following code in it.

- We have created an object with three functions - info(), warning() and error(). At the end, we have assigned this object to module.exports. The module.exports in the above example exposes a log object as a module.

- The module.exports is a special object which is included in every JS file in the Node.js application by default. Use module.exports or exports to expose a function, object or variable as a module in Node.js.

Node.js Local Modules

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
```

Node.js Local Modules

Loading Local Module

- To use local modules in your application, you need to load it using require() function in the same way as core module. However, you need to specify the path of JavaScript file of the module.

- The following example demonstrates how to use the above logging module contained in Log.js.

app.js

```
var myLogModule = require('./Log.js');

myLogModule.info('Node.js started');
```

Node.js Local Modules

Loading Local Module

- In the above example, app.js is using log module. First, it loads the logging module using require() function and specified path where logging module is stored.

- Logging module is contained in Log.js file in the root folder. So, we have specified the path './Log.js' in the require() function. The '.' denotes a root folder.

- The require() function returns a log object because logging module exposes an object in Log.js using module.exports.

- So now you can use logging module as an object and call any of its function using dot notation e.g myLogModule.info() or myLogModule.warning() or myLogModule.error()

Node.js Export Modules

Here, you will learn how to expose different types as a module using module.exports.

- The module.exports is a special object which is included in every JavaScript file in the Node.js application by default.

- The module is a variable that represents the current module, and exports is an object that will be exposed as a module.

- So, whatever you assign to module.exports will be exposed as a module.

Export Literals

- As mentioned above, exports is an object. So it exposes whatever you assigned to it as a module.

- For example, if you assign a string literal then it will expose that string literal as a module.

```
module.exports = 'Hello world';
```

Node.js Export Modules

Export Object

- The exports is an object. So, you can attach properties or methods to it.

Messages.js

```
exports.SimpleMessage = 'Hello world';

//or

module.exports.SimpleMessage = 'Hello world';
```

- We have attached a property SimpleMessage to the exports object.
- Now, import and use this module, as shown below.

app.js

```
var msg = require('./Messages.js');

console.log(msg.SimpleMessage);
```

Node.js Export Modules

You can also attach an object to module.exports, as shown below.

data.js

```
module.exports = {
    firstName: 'James',
    lastName: 'Bond'
}
```

app.js

```
var person = require('./data.js');
console.log(person.firstName + ' ' + person.lastName);
```

Node.js Export Modules

Export Function

You can attach an anonymous function to exports object as shown below.

Log.js

```
module.exports = function (msg) {
    console.log(msg);
};
```

Now, you can use the above module, as shown below.

app.js

```
var msg = require('./Log.js');

msg('Hello World');
```

The msg variable becomes a function expression in the above example. So, you can invoke the function using parenthesis ().

Node.js Export Modules

Export Function as a Class

In JavaScript, a function can be treated like a class. The following example exposes a function that can be used like a class.

Person.js

```
   module.exports = function (firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.fullName = function () {
        return this.firstName + ' ' + this.lastName;
    }
}
```

app.js

```
var person = require('./Person.js');
var person1 = new person('James', 'Bond');
console.log(person1.fullName());
```

As you can see, we have created a person object using the new keyword.

Node.js HTTP Modules

The Built-in HTTP Module

- Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP).

- To include the HTTP module, use the require() method:

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

Node.js as a Web Server

- The HTTP module can create an HTTP server that listens to server ports and gives a response back to the client.

- Use the createServer() method to create an HTTP server.

Node.js HTTP Modules

Node.js as a Web Server

Example:

```
var http = require('http');                    //create a server object:
http.createServer(function (req, res)
{
   res.write('Hello World!');                   //write a response to the client
   res.end();                                   //end the response
}).listen(8080);                                //the server object listens on port 8080
```

- The function passed into the http.createServer() method, will be executed when someone tries to access the computer on port 8080.

Node.js HTTP Modules

Add an HTTP Header

If the response from the HTTP server is supposed to be displayed as HTML, you should include an HTTP header with the correct content type:

Example:

```
var http = require('http');
http.createServer(function (req, res)
{
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('Hello World!');
  res.end();
}).listen(8080);
```

- The first argument of the res.writeHead() method is the status code, 200 means that all is OK, the second argument is an object containing the response headers.

Node.js HTTP Modules

Read the Query String

The function passed into the http.createServer() has a req argument that represents the request from the client, as an object (http.IncomingMessage object).

This object has a property called "url" which holds the part of the url that comes after the domain name:

Example:

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write(req.url);
  res.end();
}).listen(8080);
```

Node.js HTTP Modules

Split the Query String

There are built-in modules to easily split the query string into readable parts, such as the URL module.

Example:

Split the query string into readable parts:

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

http.createServer(function (req, res)
 {
  res.writeHead(200, {'Content-Type': 'text/html'});
  var q = url.parse(req.url, true).query;
  var txt = q.year + " " + q.month;
  res.end(txt);
}).listen(8080);
```

Node.js File System Module

The Node.js file system module allows you to work with the file system on your computer.

To include the File System module, use the require() method:

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

Common use for the File System module:

- Read files
- Create files
- Update files
- Delete files
- Rename files

Node.js File System Module

Reading File

Use fs.readFile() method to read the physical file asynchronously.

Syntax:

fs.readFile(fileName [,options], callback)

Parameter Description:

- filename: Full path and name of the file as a string.
- options: The options parameter can be an object or string which can include encoding and flag. The default encoding is utf8 and default flag is "r".
- callback: A function with two parameters err and fd. This will get called when readFile operation completes.

Node.js File System Module

Writing File

Use fs.writeFile() method to write 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.

Syntax:
fs.writeFile(filename, data[, options], callback)

Parameter Description:

- filename: Full path and name of the file as a string.

- Data: The content to be written in a file.

- options: The options parameter can be an object or string which can include encoding, mode and flag. The default encoding is utf8 and default flag is "r".

- callback: A function with two parameters err and fd. This will get called when write operation completes.

Node.js File System Module

Open File

Alternatively, you can open a file for reading or writing using fs.open() method.

Syntax:
fs.open(path, flags[, mode], callback)

Parameter Description:

- path: Full path with name of the file as a string.
- Flag: The flag to perform operation
- Mode: The mode for read, write or readwrite. Defaults to 0666 readwrite.
- callback: A function with two parameters err and fd. This will get called when file open operation completes.

Node.js File System Module

Flag	Description
r	Open file for reading. An exception occurs if the file does not exist.
r+	Open file for reading and writing. An exception occurs if the file does not exist.
rs	Open file for reading in synchronous mode.
rs+	Open file for reading and writing, telling the OS to open it synchronously. See notes for 'rs' about using this with caution.
w	Open file for writing. The file is created (if it does not exist) or truncated (if it exists).
wx	Like 'w' but fails if path exists.
w+	Open file for reading and writing. The file is created (if it does not exist) or truncated (if it exists).
wx+	Like 'w+' but fails if path exists.
a	Open file for appending. The file is created if it does not exist.
ax	Like 'a' but fails if path exists.
a+	Open file for reading and appending. The file is created if it does not exist.
ax+	Like 'a+' but fails if path exists

Node.js File System Module

Delete File

Use fs.unlink() method to delete an existing file.

Syntax:

```
fs.unlink(path, callback);
```

Example:

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

fs.unlink('test.txt', function ()
{
    console.log('write operation complete.');
});
```

Node.js File System Module

Important method of fs module

Method	Description
fs.readFile(fileName [,options], callback)	Reads existing file.
fs.writeFile(filename, data[, options], callback)	Writes to the file. If file exists then overwrite the content otherwise creates new file.
fs.open(path, flags[, mode], callback)	Opens file for reading or writing.
fs.rename(oldPath, newPath, callback)	Renames an existing file.
fs.chown(path, uid, gid, callback)	Asynchronous chown.
fs.stat(path, callback)	Returns fs.stat object which includes important file statistics.
fs.link(srcpath, dstpath, callback)	Links file asynchronously.

Node.js File System Module

Important method of fs module

Method	Description
fs.rmdir(path, callback)	Renames an existing directory.
fs.mkdir(path[, mode], callback)	Creates a new directory.
fs.readdir(path, callback)	Reads the content of the specified directory.
fs.utimes(path, atime, mtime, callback)	Changes the timestamp of the file.
fs.exists(path, callback)	Determines whether the specified file exists or not.
fs.access(path[, mode], callback)	Tests a user's permissions for the specified file.
fs.appendFile(file, data[, options], callback)	Appends new content to the existing file.

Debug Node.js Application

You can debug Node.js application using various tools including following:

- Core Node.js debugger
- Node Inspector
- Built-in debugger in IDEs

Core Node.js Debugger

Node.js provides built-in non-graphic debugging tool that can be used on all platforms. It provides different commands for debugging Node.js application.

Example:

```
var fs = require('fs');
fs.readFile('test.txt', 'utf8', function (err, data) {
        debugger;
    if (err) throw err;
    console.log(data);
});
```

Debug Node.js Application

Core Node.js Debugger

node debug app.js

The above command starts the debugger and stops at the first line as shown below.

Use **next** to move on the next statement.

The > is now pointing to next statement.
Use **cont** to stop the execution at next "debugger", if any.

Use **watch('expression')** command to add the variable or expression whose value you want to check.

Debug Node.js Application

Core Node.js Debugger

Command	Description
next	Stop at the next statement.
cont	Continue execute and stop at the debugger statement if any.
step	Step in function.
out	Step out of function.
watch	Add the expression or variable into watch.
watcher	See the value of all expressions and variables added into watch.
Pause	Pause running code.

Debug Node.js Application

Node Inspector

In this section, we will use node inspector tool to debug a simple Node.js application contained in app.js file.

<u>app.js</u>

```
var fs = require('fs');
fs.readFile('test.txt', 'utf8', function (err, data) {

    debugger;

    if (err) throw err;

    console.log(data);
});
```

Node inspector is GUI based debugger. Install Node Inspector using NPM in the global mode by writing the following command in the terminal window (in Mac or Linux) or command prompt (in Windows).

Debug Node.js Application

Node Inspector

```
npm install -g node-inspector
```

After installing run it using node-inspector command.

It will display an URL for debugging purpose.

So, point your browser to http://127.0.0.1:8080/?ws=127.0.0.1:8080&port=5858 and start debugging. Sometimes, port 8080 might not be available on your computer.

So you will get the following error.

```
Cannot start the server at 0.0.0.0:8080. Error: listen EACCES.
```

In this case, start the node inspector on a different port using the following command.

```
>node-inspector --web-port=5500
```

Node.js EventEmitter

- Node.js allows us to create and handle custom events easily by using events module.

- Event module includes EventEmitter class which can be used to raise and handle custom events.

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

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

Node.js EventEmitter

Node.js allows us to create and handle custom events easily by using events module. Event module includes EventEmitter class which can be used to raise and handle custom events.

EventEmitter Class

As we have seen in the previous section, EventEmitter class lies in the events module. It is accessible via the following code −

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

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

When an EventEmitter instance faces any error, it emits an 'error' event. When a new listener is added, 'newListener' event is fired and when a listener is removed, 'removeListener' event is fired.

EventEmitter provides multiple properties like on and emit. on property is used to bind a function with the event and emit is used to fire an event.

Node.js EventEmitter

Sr. No.	Method & Description
1	**addListener(event, listener)** - Adds a listener at the end of the listeners array for the specified event. - No checks are made to see if the listener has already been added. Multiple calls passing the same combination of event and listener will result in the listener being added multiple times. - Returns emitter, so calls can be chained.
2	**on(event, listener)** - Adds a listener at the end of the listeners array for the specified event. - No checks are made to see if the listener has already been added. Multiple calls passing the same combination of event and listener will result in the listener being added multiple times. - Returns emitter, so calls can be chained.
3	**once(event, listener)** Adds a one time listener to the event. This listener is invoked only the next time the event is fired, after which it is removed. Returns emitter, so calls can be chained.

Node.js EventEmitter

Sr. No.	Method & Description
4	**removeListener(event, listener)** Removes a listener from the listener array for the specified event. **Caution −** It changes the array indices in the listener array behind the listener. removeListener will remove, at most, one instance of a listener from the listener array.
5	**removeAllListeners([event])** Removes all listeners, or those of the specified event. It's not a good idea to remove listeners that were added elsewhere in the code, especially when it's on an emitter that you didn't create (e.g. sockets or file streams). Returns emitter, so calls can be chained.
6	**setMaxListeners(n)** By default, EventEmitters will print a warning if more than 10 listeners are added for a particular event. This is a useful default which helps finding memory leaks. Obviously not all Emitters should be limited to 10. This function allows that to be increased. Set to zero for unlimited.

Node.js EventEmitter

Sr. No.	Method & Description
7	**listeners(event)** Returns an array of listeners for the specified event.
8	**emit(event, [arg1], [arg2], [...])** Execute each of the listeners in order with the supplied arguments. Returns true if the event had listeners, false otherwise.

Class Methods

Sr.No.	Method & Description
1	**listenerCount(emitter, event)** Returns the number of listeners for a given event.

Node.js EventEmitter

Events

Sr. No.	Events & Description
1	**newListener** • **event** − String: the event name • **listener** − Function: the event handler function This event is emitted any time a listener is added. When this event is triggered, the listener may not yet have been added to the array of listeners for the event.
2	**removeListener** • **event** − String The event name • **listener** − Function The event handler function This event is emitted any time someone removes a listener. When this event is triggered, the listener may not yet have been removed from the array of listeners for the event.

Node.js EventEmitter

Common Patterns for EventEmitters

There are two common patterns that can be used to raise and bind an event using EventEmitter class in Node.js.

1.Return EventEmitter from a function

2.Extend the EventEmitter class

Return EventEmitter from a function

In this pattern, a constructor function returns an EventEmitter object, which was used to emit events inside a function. This EventEmitter object can be used to subscribe for the events.

Extend EventEmitter Class

In this pattern, we can extend the constructor function from EventEmitter class to emit the events.

Node.js - Callbacks Concept

What is Callback?

- Callback 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 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.

Node.js Buffers

- Node.js provides Buffer class to store raw data similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap.
- Buffer class is used because pure JavaScript is not nice to binary data. So, when dealing with TCP streams or the file system, it's necessary to handle octet streams.
- Buffer class is a global class. It can be accessed in application without importing buffer module.

Create an uninitiated buffer: creating an uninitiated buffer of 10 octets:

```
var buf = new Buffer(10);
```

Create a buffer from array: create a Buffer from a given array:

```
var buf = new Buffer([10, 20, 30, 40, 50]);
```

Create a buffer from string: create a Buffer from a given string and optionally encoding type:

```
var buf = new Buffer("Simply Easy Learning", "utf-8");
```

Node.js Buffers

Create an uninitiated buffer: creating an uninitiated buffer of 10 octets:

```
var buf = new Buffer(10);
```

Create a buffer from array: create a Buffer from a given array:

```
var buf = new Buffer([10, 20, 30, 40, 50]);
```

Create a buffer from string: create a Buffer from a given string and optionally encoding type:

```
var buf = new Buffer("Simply Easy Learning", "utf-8");
```

Node.js Writing to buffers

Syntax:
buf.write(string[, offset][, length][, encoding])

Parameter explanation:

string: It specifies the string data to be written to buffer.

offset: It specifies the index of the buffer to start writing at. Its default value is 0.

length: It specifies the number of bytes to write. Defaults to buffer.length

encoding: Encoding to use. 'utf8' is the default encoding.

Return values from writing buffers:
This method is used to return number of octets written. In the case of space shortage for buffer to fit the entire string, it will write a part of the string.

Node.js Reading from buffers

Syntax:
buf.toString([encoding][, start][, end])

Parameter explanation:

encoding: It specifies encoding to use. 'utf8' is the default encoding

start: It specifies beginning index to start reading, defaults to 0.

end: It specifies end index to end reading, defaults is complete buffer.

Return values reading from buffers:
This method decodes and returns a string from buffer data encoded using the specified character set encoding.

Node.js Streams

Streams are the objects that facilitate you to read data from a source and write data to a destination.

There are four types of streams in Node.js:

- **Readable:** This stream is used for read operations.
- **Writable:** This stream is used for write operations.
- **Duplex:** This stream can be used for both read and write operations.
- **Transform:** It is type of duplex stream where the output is computed according to input.

Each type of stream is an Event emitter instance and throws several events at different times.

Following are some commonly used events:

- **Data:**This event is fired when there is data available to read.
- **End:**This event is fired when there is no more data available to read.
- **Error:** This event is fired when there is any error receiving or writing data.
- **Finish:**This event is fired when all data has been flushed to underlying system.

Node.js Streams

Example:

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

// Create a readable stream
            var readerStream = fs.createReadStream('input.txt');

// Set the encoding to be utf8.
            readerStream.setEncoding('UTF8');

// Handle stream events --> data, end, and error
            readerStream.on('data', function(chunk) {
  data += chunk;
});

readerStream.on('end',function(){
  console.log(data);
});

readerStream.on('error', function(err){
  console.log(err.stack);
});
```

Node.js Streams

Example:

```
var fs = require("fs");
var data = 'A Solution of all Technology';

// Create a writable stream
                var writerStream = fs.createWriteStream('output.txt');

// Write the data to stream with encoding to be utf8
                writerStream.write(data,'UTF8');

// Mark the end of file
                writerStream.end();

// Handle stream events --> finish, and error
writerStream.on('finish', function() {
    console.log("Write completed.");
});

writerStream.on('error', function(err){
   console.log(err.stack);
});
console.log("Program Ended");
```

Node.js Send an Email

The Nodemailer Module

- The Nodemailer module makes it easy to send emails from your computer.
- The Nodemailer module can be downloaded and installed using npm:

C:\Users\Your Name>npm install nodemailer

- After you have downloaded the Nodemailer module, you can include the module in any application:

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

Node.js Send an Email

Example:

```
var nodemailer = require('nodemailer');
var transporter = nodemailer.createTransport({
  service: 'gmail',
  auth: {
    user: 'youremail@gmail.com',
    pass: 'yourpassword'
  }
});

var mailOptions = {
  from: 'youremail@gmail.com',
  to: 'myfriend@yahoo.com',
  subject: 'Sending Email using Node.js',
  text: 'That was easy!'
};
transporter.sendMail(mailOptions, function(error, info){
  if (error) {
    console.log(error);
  } else {
    console.log('Email sent: ' + info.response);
  }
});
```

Node.js Send an Email

Multiple Receivers

To send an email to more than one receiver, add them to the "to" property of the mailOptions object, separated by commas:

Example

Send email to more than one address:

```
var mailOptions = {
  from: 'youremail@gmail.com',
  to: 'myfriend@yahoo.com, myotherfriend@yahoo.com',
  subject: 'Sending Email using Node.js',
  text: 'That was easy!'
}
```

Node.js Send an Email

Send HTML

To send HTML formatted text in your email, use the "html" property instead of the "text" property:

Example

Send email containing HTML:

```
var mailOptions = {
  from: 'youremail@gmail.com',
  to: 'myfriend@yahoo.com',
  subject: 'Sending Email using Node.js',
  html: '<h1>Welcome</h1><p>That was easy!</p>'
}
```

References:

- https://www.tutorialsteacher.com/nodejs/nodejs-basics
- https://www.tutorialspoint.com/nodejs/nodejs_first_application.htm
- https://www.javatpoint.com/nodejs-streams