Node.js Tutorial

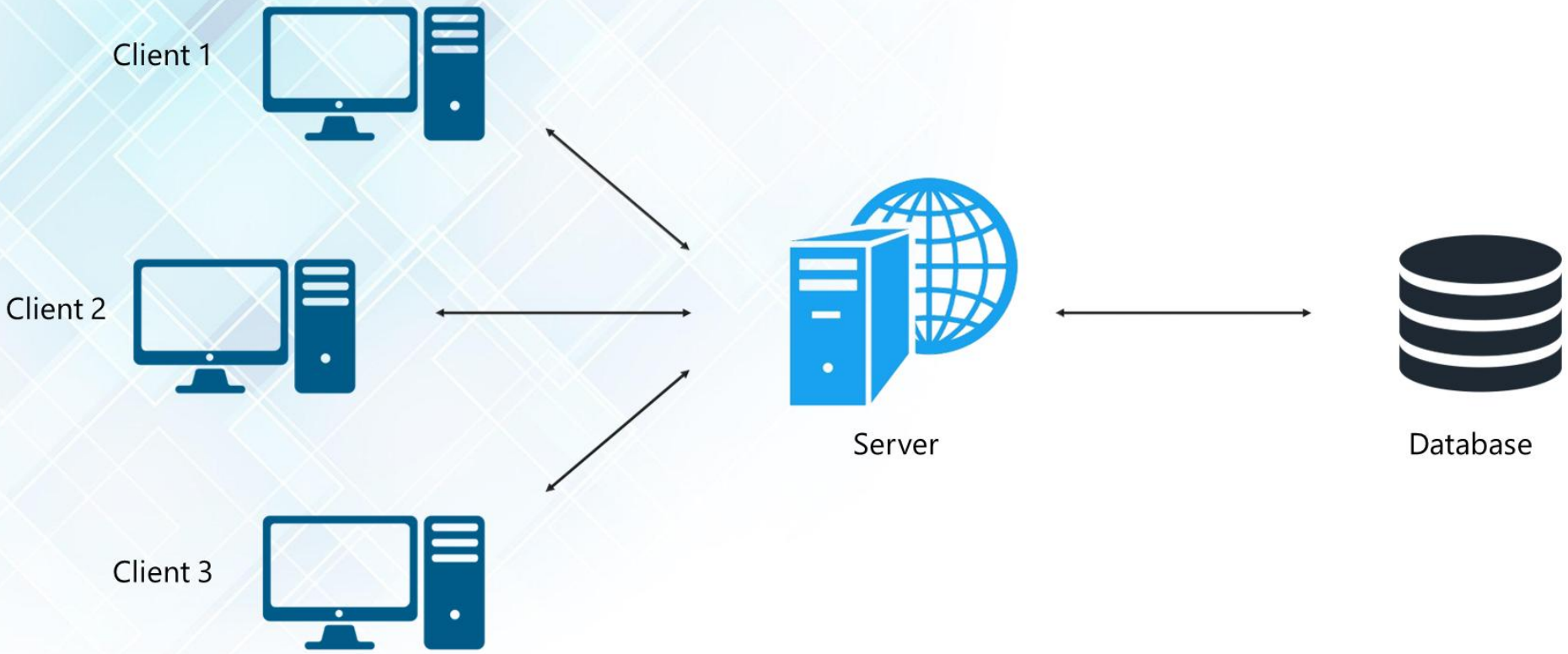
Vijesh M. Nair
Assistant Professor
Dept. of CSE (AI-ML)

Agenda

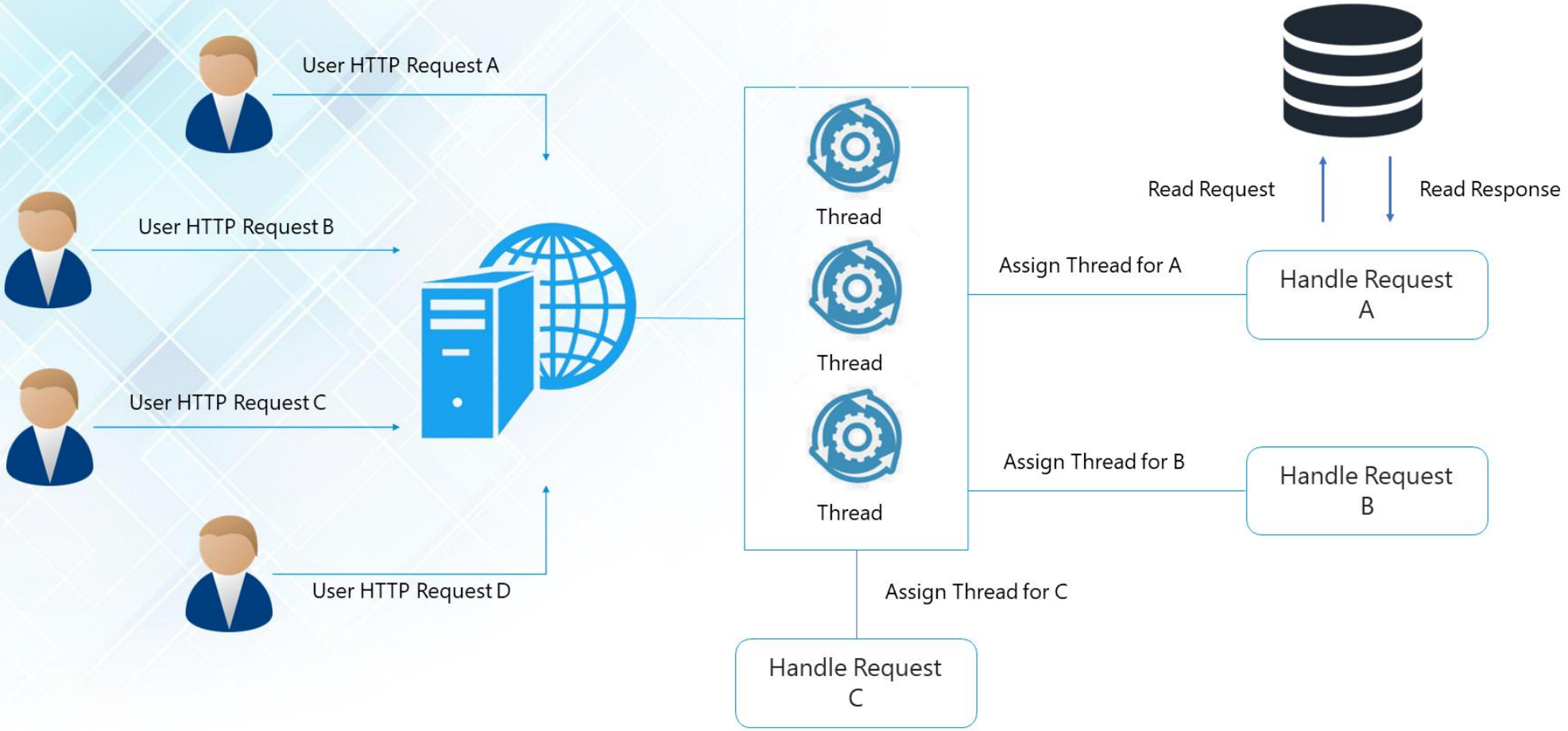
- Client Server Architecture
- Limitations of Multi–Threaded Model
- What is Node.js?
- Features of Node.js
- Node.js Installation
- Blocking Vs. Non – Blocking I/O
- Creating Node.js First Program
- Node.js Modules

Client – Server Architecture

Client Server Architecture

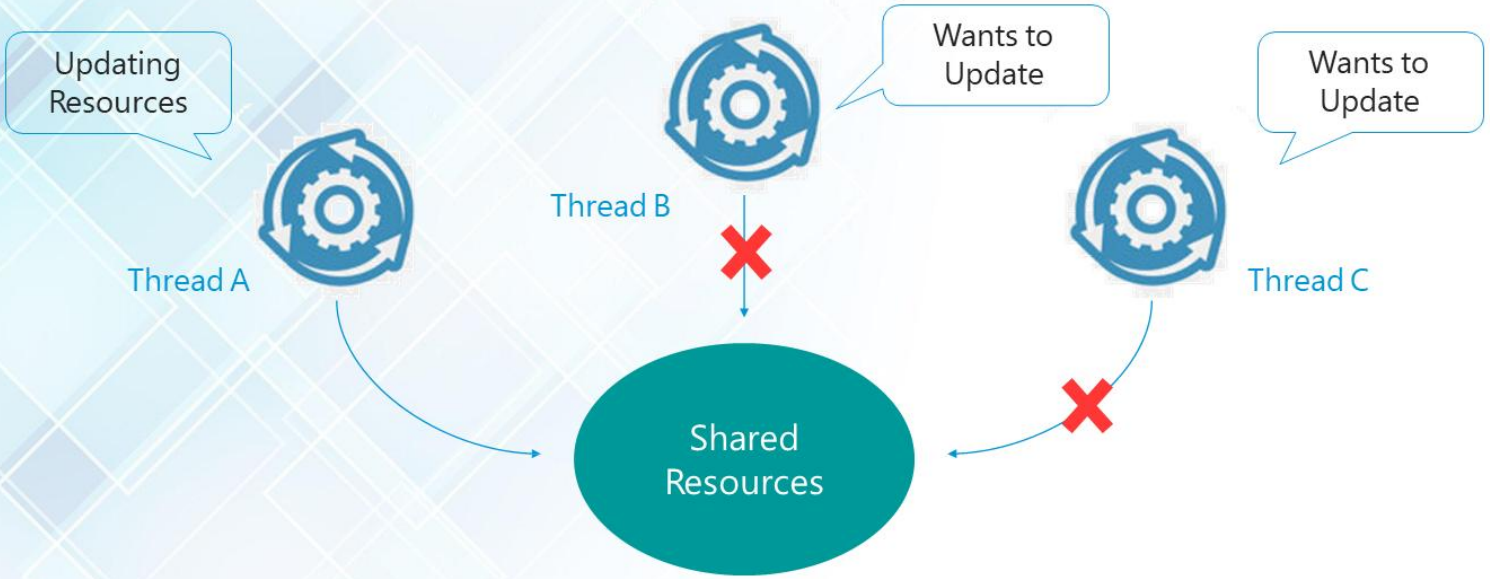


Traditional Multi-Threaded Model



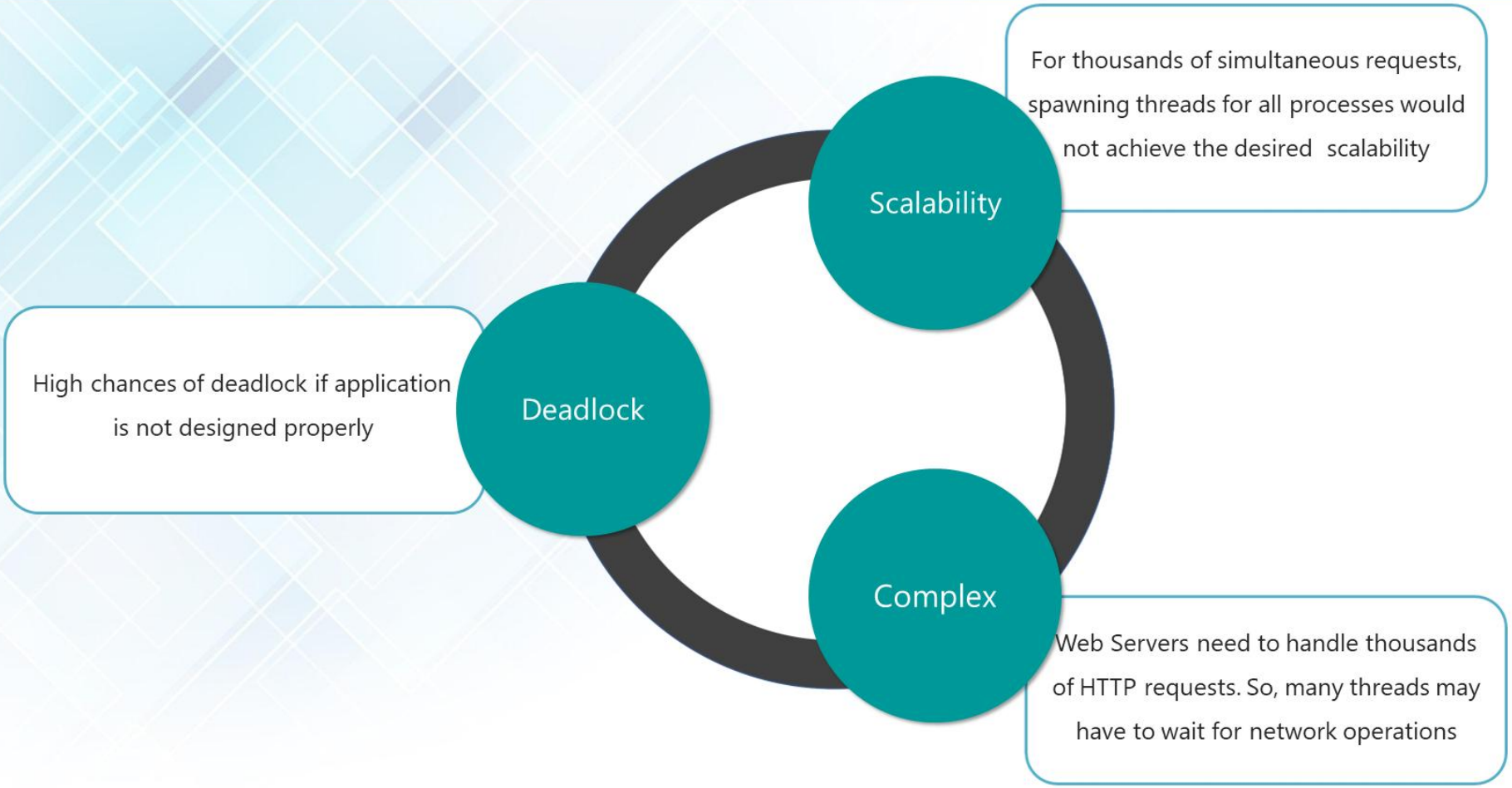
Limitations of Multi – Threaded Approach

Limitations of Multi-Threaded Models



- In a multi-threaded HTTP server, for each and every request that the server receives, it creates a separate thread which handles that request
- If a request acquires a lock in the shared resource and it is 'exclusive', it will affect result of other requests

Limitations of Multi-Threaded Models



What is Node.js ?

What is Node.js?

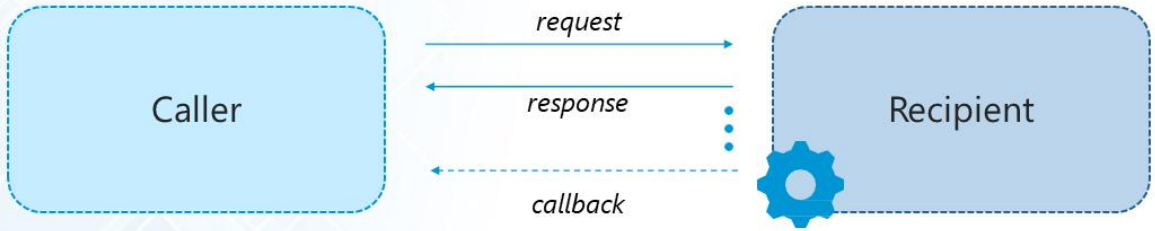
- Node.js is an open source runtime environment for server-side and networking applications and is single threaded.
- Uses Google JavaScript V8 Engine to execute code.
- It is cross platform environment and can run on OS X, Microsoft Windows, Linux, FreeBSD, NonStop and IBM.
- Provides an event driven architecture and non blocking I/O that is optimized and scalable.

Features of Node.js

Features of Node.js

- Asynchronous
- Event Driven
- Very Fast
- Scalable

1. Asynchronous



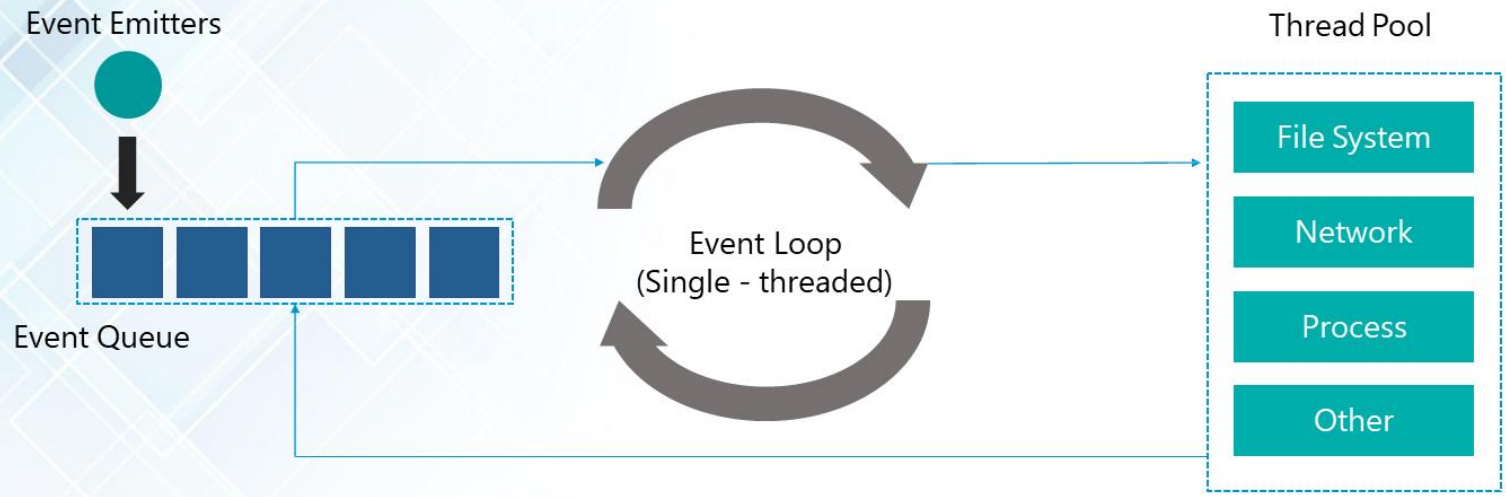
Asynchronous Model

- When request is made to server, instead of waiting for the request to complete, server continues to process other requests
- When request processing completes, the response is sent to caller using callback mechanism

Features of Node.js

- Asynchronous
- Event Driven
- Very Fast
- Scalable

2. Single Threaded and Event Driven:



- As soon as these requests reach the application, they go to the Event Queue, which is a **queue where all the events that occur in the application goes first**, and where they await to be send to be processed in the main thread called Event Loop.
- When a request (Blocking Operation) enters in the Event Loop, which is a **single thread platform that runs the V8 Engine in its core to compile JavaScript**, it's delegated to the Thread Pool platform to be processed in background.
- In the Thread Pool, which is a **multi thread platform that runs a library called libuv and has C++ in its core**, the request (Blocking Operation) is processed asynchronously in the background until it's completed and ready to be returned.

Features of Node.js

- Asynchronous
- Event Driven
- Very Fast
- Scalable

3. Very Fast

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

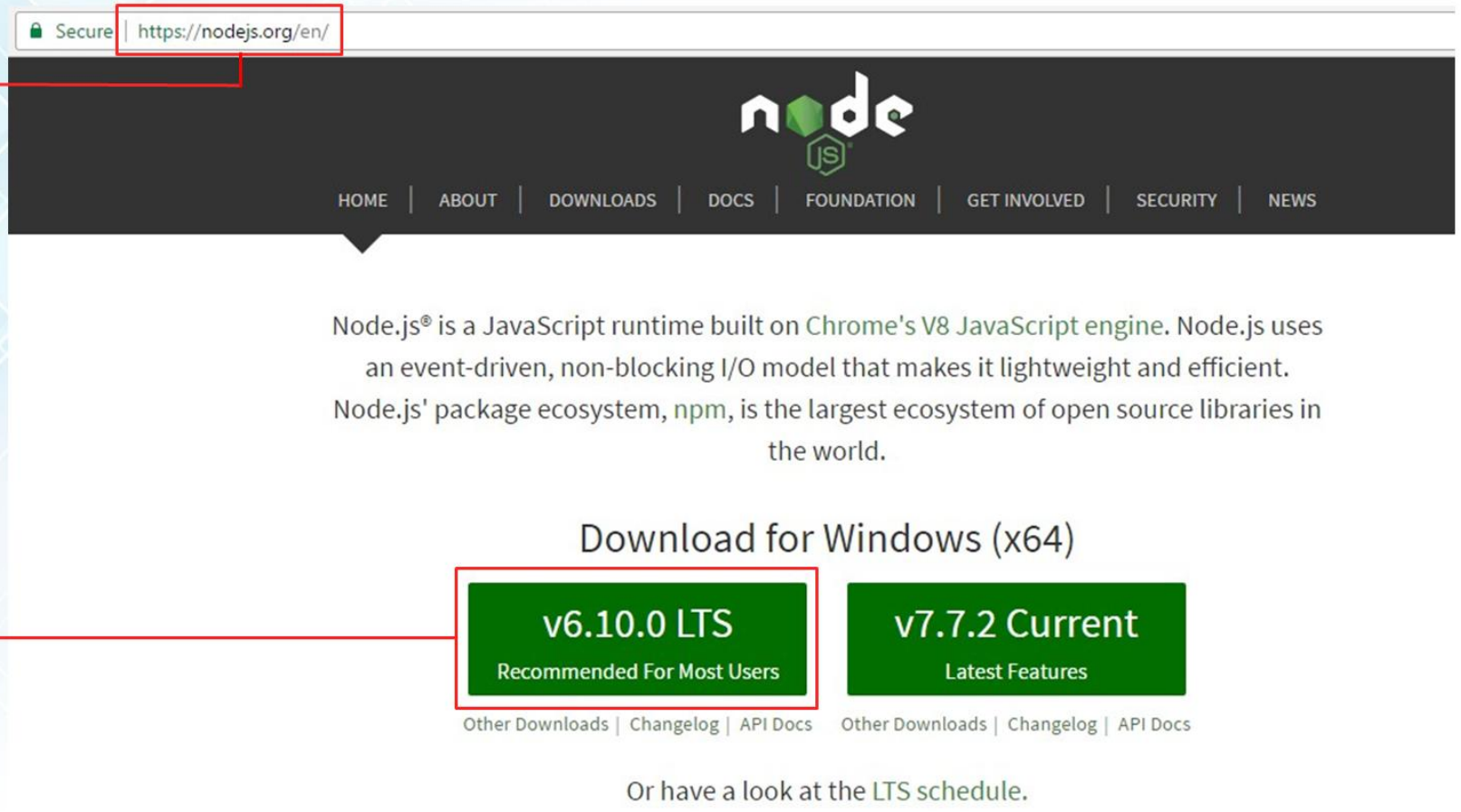
4. Single Threaded but Highly Scalable

Node.js is highly scalable because event mechanism helps the server to respond in a non-blocking way.

Node.js Installation

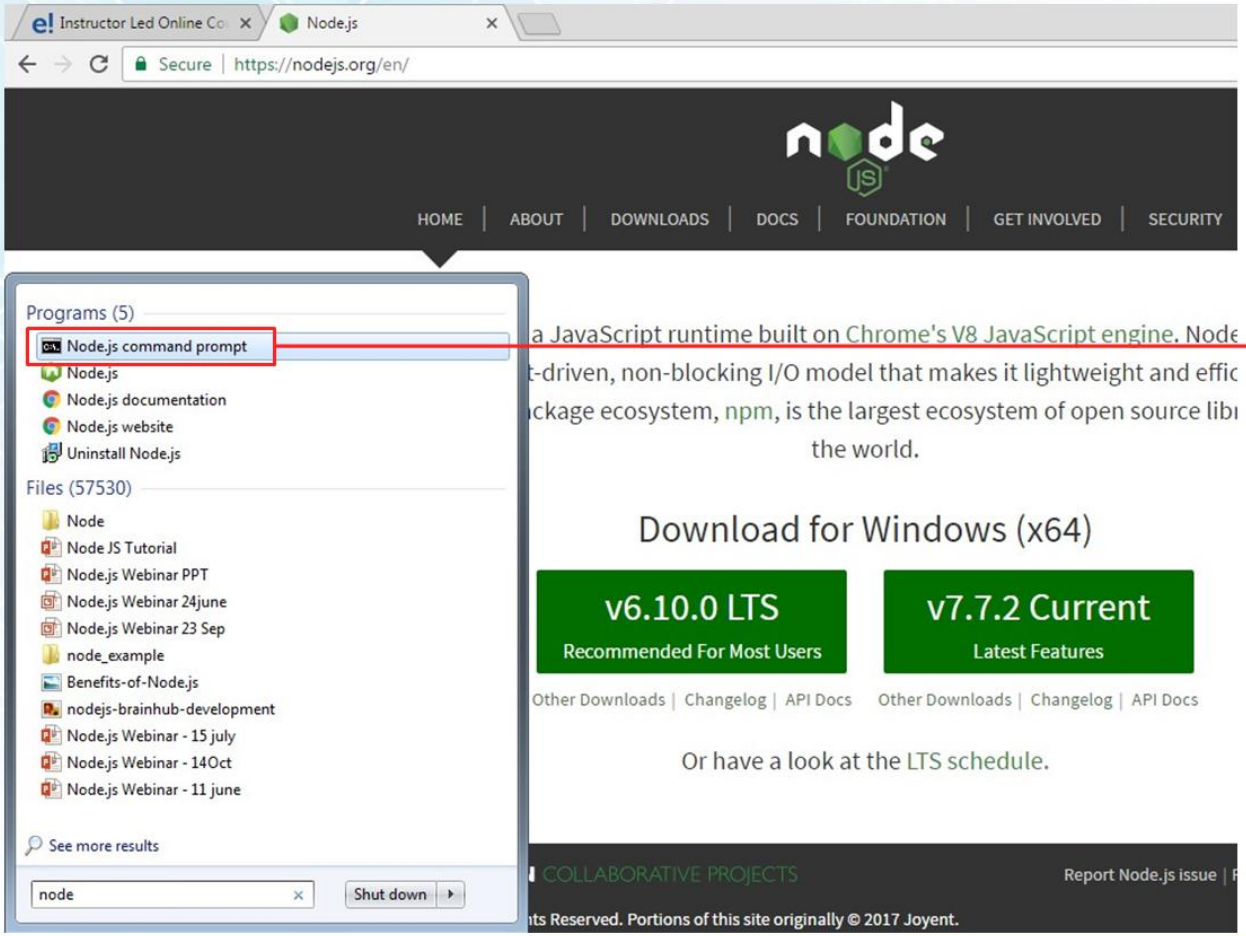
Node.js Installation

1. Go to *https://nodejs.org/en/*



2. Download and Install

Node.js Installation



3 Open node.js command prompt

- node –version

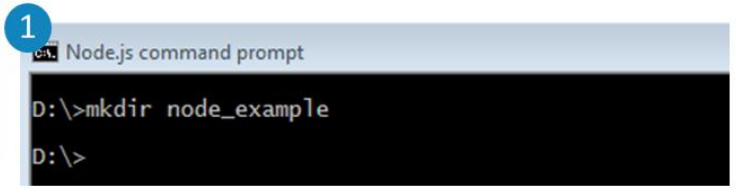
Check the version of Node.js installed

Creating Your First Node.js Program

Node.js First Program

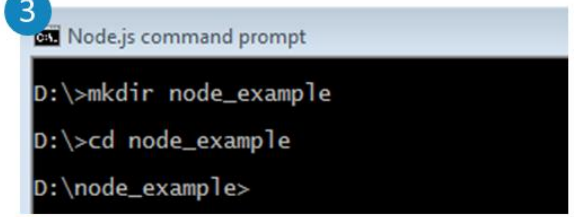
1. Create a Directory: mkdir node_example
2. Create a File: first_example.js
3. Go to the node_example directory
4. Execute the java script file: node first_example.js

1

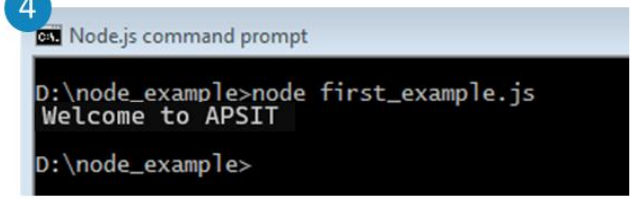


2

3



4



Simple Web Application Example

Simple Web Application Example

1

Importing Module:

'require' is used to load a Node.js modules.

Example:

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

2

Creating Server

- Used to create web server object
- Function (request, response) is called once for every HTTP request to the server, so it's called the request handler.

Example:

```
var server = http.createServer(function(request, response){
    console.log('creating Server Object');
});
```

3

listen(<port_no>)

Bind the server instance for listening to a particular port.

Example:

```
server.listen(port, function(){
        console.log('Server running at http://localhost:3000/');
});
```

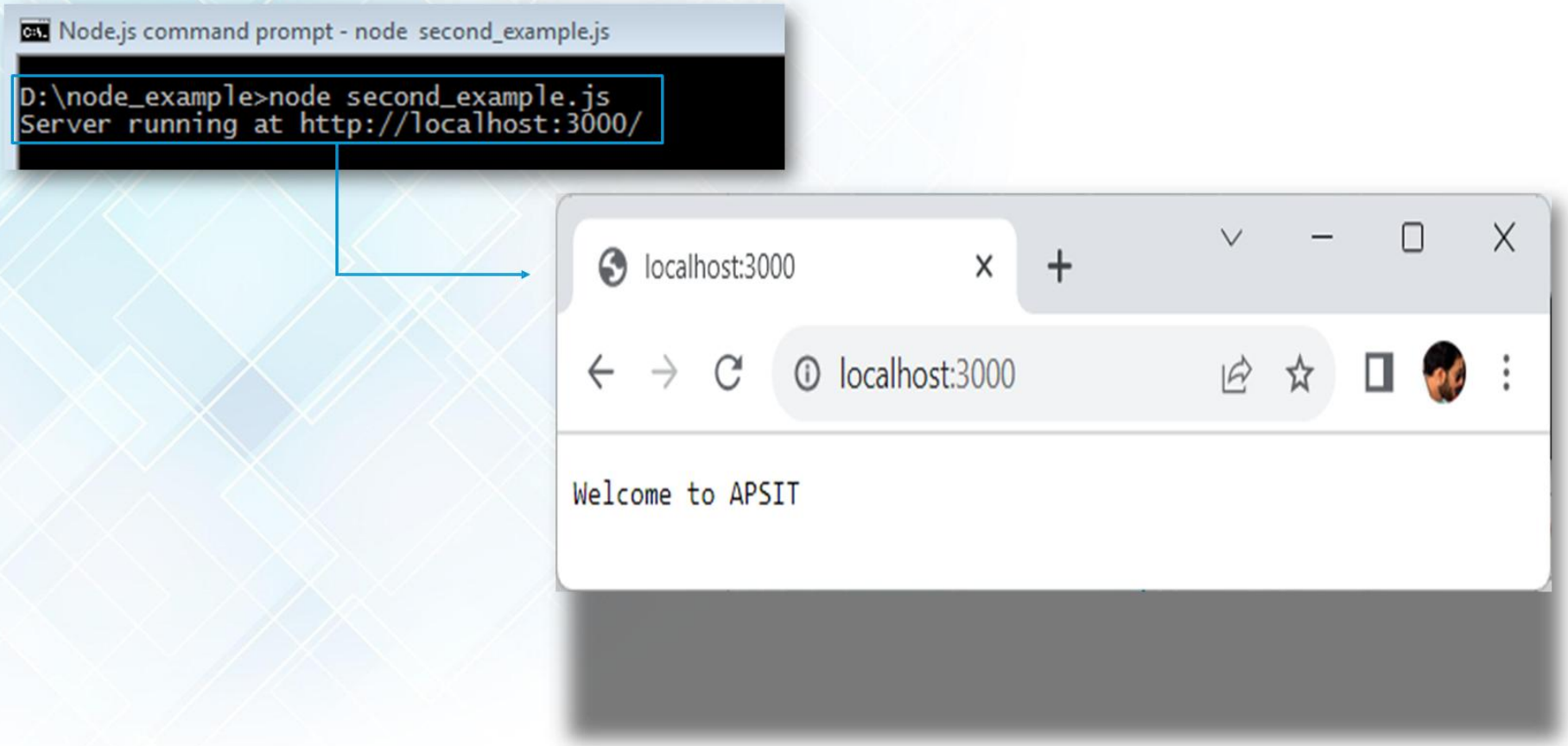
Simple Web Application Example

second_example.js

```
1  var http =  require('http');
2
3  var port = 3000;
4
5  var server = http.createServer(function (request, response) {
6
7     response.writeHead(200, {'Content-Type': 'text/plain'});
8
9    response.end('Welcome to APSIT\n');
10 })
11
12 server.listen(port, function(){
13         console.log('Server running at http://localhost:3000/');
14 });
15
```

1. Import HTTP Module
2. Define Port No.
3. Creating Server Instance
4. Sending HTTP Header
5. Sending Data
6. Binding Server Instance with the Port No. 3000

Simple Web Application Example



Asynchronous programming Model: Blocking vs Non - Blocking

Blocking vs Non–Blocking I/O

- **Blocking** is when the execution of additional JavaScript in the Node.js process must wait until a non-JavaScript operation completes.
- "I/O" refers primarily to interaction with the system's disk and network supported by libuv.
- **Blocking** methods execute **synchronously** and **non-blocking** methods execute **asynchronously**.

```
var fs = require('fs');
var data = fs.readFileSync('test1.txt');
console.log(data.toString());
//more methods ...
```

More methods will be blocked till the read method is not executed

```
var fs = require('fs');
fs.readFile('test1.txt', function(err, data){
        if(!err)
            console.log(data.toString());
});
//more methods ...
```

More method will execute asynchronously (non-blocking way)

Blocking vs Non–Blocking I/O

Non – Blocking Example: ***(Asynchronous)***

```
1   var fs = require('fs');
2
3   fs.readFile('test_file.txt','utf8', function(err, data) {
4       if (err)
5       {
6         console.log(err);
7       }
8       setTimeout(() => {
9           console.log('Displaying content after 5 seconds');
10          console.log(data);
11      }, 5000);
12  });
13
14  console.log('Will execute First');
```

```
D:\node_example>node test.js
Will execute First
Displaying content after 5 seconds
Welcome to Web Computing Class..!
This is Node.js Tutorial taught by VJsh Nair.
*******
```

Blocking vs Non–Blocking I/O

Blocking Example: ***(Synchronous)***

```
1  var fs = require('fs');
2
3  var data = fs.readFileSync('test_file.txt','utf8');
4  console.log(data)
5  console.log('End here');
```

```
D:\node_example>node test1.js
Welcome to Web Computing Class..!
This is Node.js Tutorial taught by VJsh Nair.
*******
End here
```

CONCLUSION:

- In case of Asynchronous (non-blocking), once file I/O is complete, it will call the callback function
- This allows Node.js to be scalable and process large number of request without worrying about blocking I/O

Node.js Modules

01	02	03	04	05	06
CALLBACKS	NPM	GLOBALS	FILE SYSTEM	EVENT	HTTP

Callback Concept

- A callback function is a function that is passed to another function as a parameter, and the callback function is called (executed) inside the second function.
- Callback function is when a task is completed, thus helping in preventing any kind of blocking and a callback function allows other code to run in the meantime
- Callbacks functions execute asynchronously, instead of reading top to bottom procedurally.
- Using the Callback concept, Node.js can process a large number of requests without waiting for any function to return the result which makes Node.js highly scalable.
- ***For example:*** In Node.js, when a function starts reading the file, it returns the control to the execution environment immediately so that the next instruction can be executed. Once file I/O gets completed, the callback function will get called to avoid blocking or waiting for File I/O.

Need of Callback

```
1  var content;
2  function readingfile() {
3  var fs = require('fs');
4  content = fs.readFileSync("readme.txt", "utf8");
5  return content;
6  }
7  readingfile();
8  console.log(content);
```

- In the above code block we have defined a function called readingfile() who's task is to read a file and store it's content to a variable.
- When the function is called, if reading data takes a long time to load the data, then this cause the whole program to 'block' - otherwise known as sitting still and waiting.
- This is the expectation of synchronous code - it sequentially runs top to bottom.
- Node.js uses callbacks, being an asynchronous platform, it does not wait around like database query, file I/O to complete. The callback function is called at the completion of a given task; this prevents any blocking, and allows other code to be run in the meantime. See the following example :

Need of Callback

- In the above example the *mycontent()* function can get passed in an argument that will become the callback variable inside the *readingfile()* function.
- After file reading is completed through *readFile()* the callback variable (*callback()*) will be invoked. Only function can be invoked, so passing anything other than function will cause an error.
- When a function gets invoked, the code inside that function get executed (here *console.log(fcontent)* statement will execute).

```
1  var fs = require('fs');
2  var fcontent ;
3  function readingfile(callback){
4  fs.readFile("readme.txt", "utf8", function(err, content) {
5  fcontent=content;
6   if (err)
7   {
8   return console.error(err.stack);
9    }
10 callback(content);
11 })
12 };
13 function mycontent() {
14   console.log(fcontent);
15 }
16 readingfile(mycontent);
17 console.log('Reading files....');
```

- At first glace, the above code looks unnecessary complicated, but callback is the foundation of Node.js. This allows you to have as many I/O operations as your OS can handle happening at the same time.

Need of Callback

- For example, suppose in a web server there are hundreds or thousands of pending requests with multiple blocking queries, but performing the blocking queries asynchronously gives you the ability to work continuously and not just sit still and wait until the blocking operations come back.
- In the above environment, generally, the callback is the last parameter and error value is the first parameter of the callback.
- The callback gets called after the function is done with all of its operations. If any error occurs the function calls the callback with the first parameter being an error object otherwise (in the case of clear execution) the function calls the callback with the first parameter being null and the rest being the return value(s).

Callback Concept

Callback is an asynchronous equivalent for a function and is called at the completion of each task

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

fs.readFile('test_file.txt', function(err, data){
    if(err)
    {
        console.log(err);
    }
    setTimeout(() => {
        console.log('Displaying content after 5 seconds');
        console.log(data);
    }, 5000);
});

console.log('Will execute First');
```

Callback: will execute after file read is complete

Output:

```
D:\node_example>node test.js
Will execute First
Displaying content after 5 seconds
Welcome to Web Computing Class..!
This is Node.js Tutorial taught by VJsh Nair.
*******
```

Basic Concepts:

01	02	03	04	05	06
CALLBACKS	NPM	GLOBALS	FILE SYSTEM	EVENT	HTTP

NPM

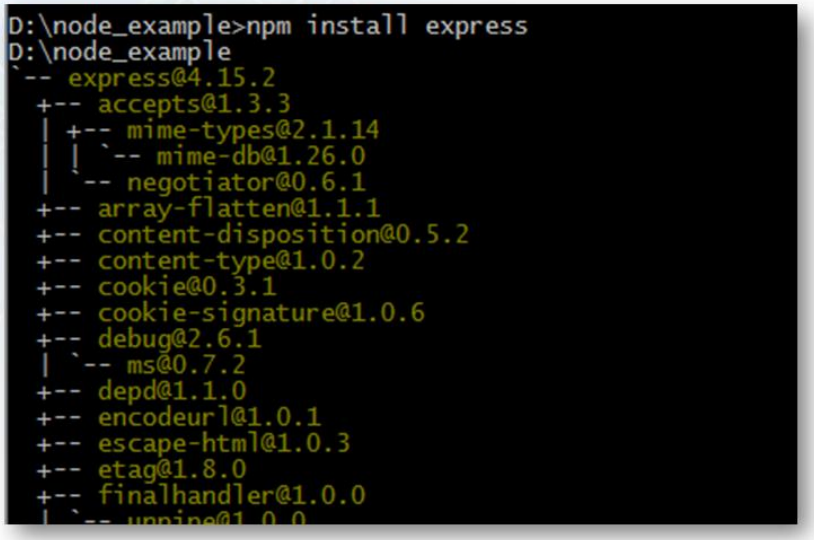
- NPM stands for Node Package Manager
- Provides online repositories for node.js packages/modules
- Provides command line utility to install Node.js packages along with version management and dependency management.

```
D:\node_example>npm version
{ npm: '3.10.10',
  ares: '1.10.1-DEV',
  http_parser: '2.7.0',
  icu: '57.1',
  modules: '48',
  node: '6.9.5',
  openssl: '1.0.2k',
  uv: '1.9.1',
  v8: '5.1.281.89',
  zlib: '1.2.8' }

D:\node_example>
```

NPM

npm install	Install all the modules as specified in package.json
npm install <**Module** Name>	Install Module using npm
npm install <**Module** Name> **-g**	Install dependency globally



Node.js Modules

01	02	03	04	05	06
CALLBACKS	NPM	GLOBALS	FILE SYSTEM	EVENT	HTTP

Global Objects

These objects are available in all modules and therefore, are referred as Global Objects

__dirname

specifies the name of the directory that currently contains the code.

__filename

specifies the name of the file that currently contains the code.

test.js

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

Node.js command prompt

```
D:\node_example>node test.js
D:\node_example
D:\node_example\test.js
```

Global Objects

A timer in Node.js is an internal construct that calls a given function after a certain period of time.

setTimeout(callback, delay[, ...args])

- Schedules execution of a one-time callback after delay milliseconds
- Returns a Timeout for use with clearTimeout( )

setInterval(callback, delay[, ...args])

- Schedules repeated execution of callback every delay milliseconds.
- Returns a Timeout for use with clearTimeout( )

setImmediate(callback, [,..args])

- Schedules an immediate execution of the callback after I/O events' callbacks but before setTimeout() and setInterval() timers are triggered.
- Returns an Immediate for use with clearImmediate().

Global Objects

```
setTimeout(()=>{
    console.log('Will display one time after 5 seconds');
},5000);

setInterval(()=>{
    console.log('Display in interval: 2 seconds');
},2000);

setImmediate(()=>{
    console.log('Will display immediately');
});
```

output

```
D:\node_example>node test.js
Will display immediately
Display in interval: 2 seconds
Display in interval: 2 seconds
Will display one time after 5 seconds
Display in interval: 2 seconds
Display in interval: 2 seconds
Display in interval: 2 seconds
```

Node.js Modules

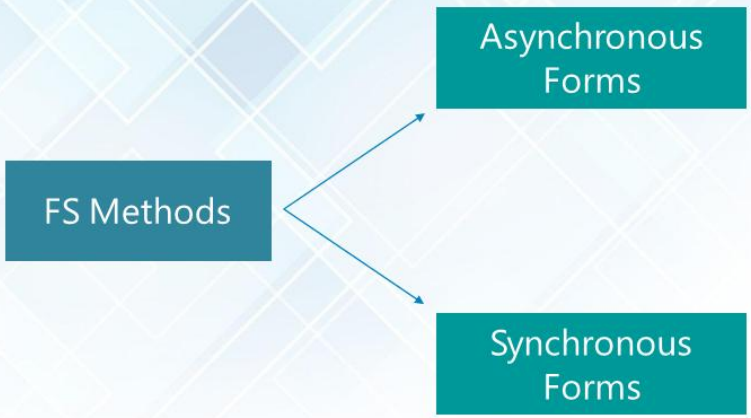
01	02	03	04	05	06
CALLBACKS	NPM	GLOBALS	FILE SYSTEM	EVENT	HTTP

File System

File I/O is provided by simple wrappers around standard POSIX functions. For importing File System Module (fs), we use: *var fs = require("fs");*

Important Note:

1. All methods have asynchronous and synchronous forms



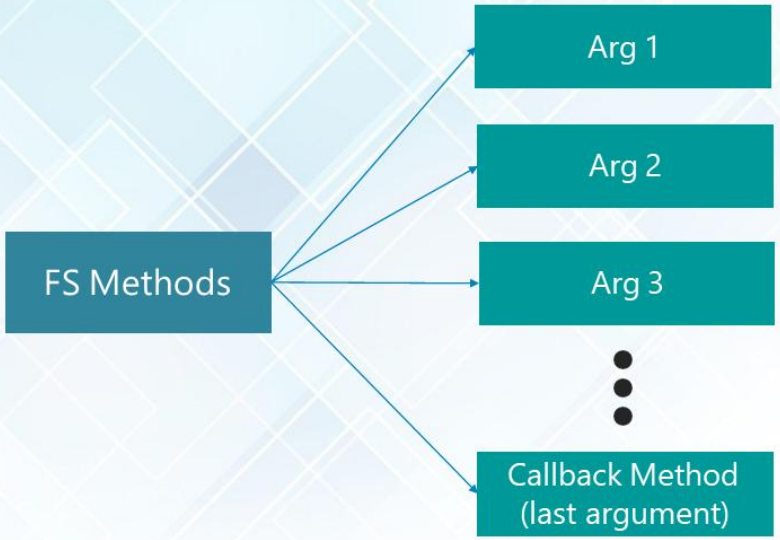

```
var fs = require("fs");

// Asynchronous read
fs.readFile('test.txt', 'utf8', function (err, data) {  });

// Synchronous read
var data = fs.readFileSync('input.txt', 'utf8');
```

File System

Important Note: 2 The asynchronous form always takes a completion callback as its last argument.



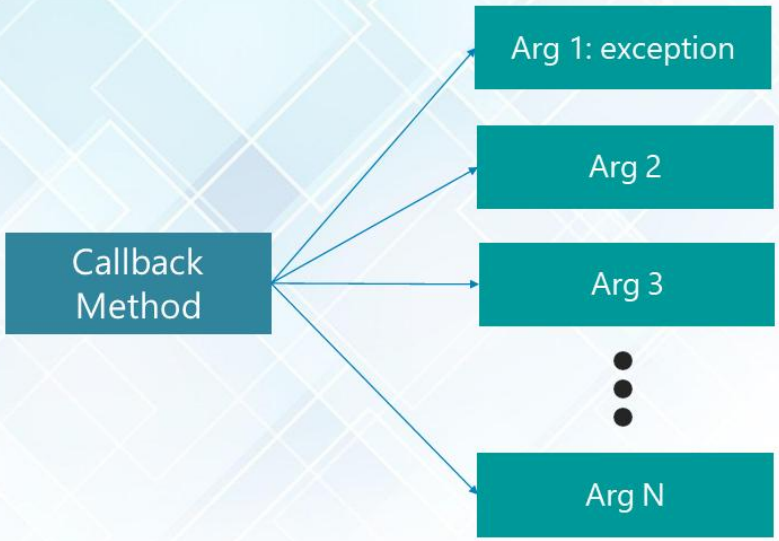

```
var fs = require("fs");

// Asynchronous read
fs.readFile('test.txt', function (err, data) {
        if (err) {
    return console.error(err);
  }
  console.log(data.toString());
});
```

Callback As Last Arg.

File System

Important Note: 3 First argument in completion callback is always reserved for an exception




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

// Asynchronous read
fs.readFile('test.txt', function (err, data) {
    if (err) {
        return console.error(err);
    }
    console.log(data.toString());
});
```

Reserved for exception

returns null or undefined (successful completion)

Some Methods in File System Module – Open()

- var fs = require('fs');

fs.open(path, flags[, mode], callback) → Open File Asynchronously

fs.openSync(path, flags[, mode]) → Open File Synchronously

fs.close(fd, callback) → Closing File

Arguments	Description
Path < string >	Path of the file
Flags < string >	Access Modifiers
Mode < integer >	sets the permission and sticky bits, but only if the file was created
Callback < function >	Callback signature - function(err, fd)

Flag Modes in Open Method

Modes	Description
r	Open file for reading. an exception occurs if the file does not exist
r+	Open file for reading and writing. Exception: if the file does not exist
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. File is created (if it does not exist) or truncated (if it exists)
wx+	Same as '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

Some Methods in File System Module – Read()

read(fd, buffer, offset, length, position, callback) → Read Content of a File into Buffer

readFile(file[, options], callback) → Reads File Asynchronously

readFileSync(file[, options]) → Reads File Synchronously

Arguments	Description
fd < integer >	File Descriptor
buffer <string | Buffer | Unit8Array >	The buffer that the data will be written to.
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 >	Read Callback signature - function(err, bytesRead, buffer)

Some Methods in File System Module – Write()

writeFile(file, data[, options], callback) → Writes into a File Asynchronously

writeFileSync(file, data[, options]) → Writes into a File Synchronously

Arguments	Description
File	Filename or File Descriptor
Data	The buffer that the data will be written to.
Options:	Encoding, Mode or flag
Callback < function >	Callback signature - function(err, bytesRead, buffer)

Node.js Modules

01	02	03	04	05	06
NPM	GLOBALS	FILE SYSTEM	CALLBACKS	EVENT	HTTP

Events

- Node.js follows event-driven architecture where certain objects (called "emitters") periodically emit named events which further invokes the listeners (function).
- Node.js provide concurrency by using the concept of events and callbacks
- All objects that emit events are instances of the EventEmitter class.

```
var event = require('events');
```
→ Importing 'events' module

```
var eventEmitter = event.EventEmitter();
```
→ Object of EventEmitter Class

```
eventEmitter.on('event', eventHandler)
```
→ Registering listeners

```
eventEmitter.emit('event')
```
→ Trigger event

Events

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

const myEmitter = new event.EventEmitter();

fs.readFile('test1.txt',(err, data) => {
    console.log(data.toString());
    myEmitter.emit('readFile');
});

myEmitter.on('readFile', () => {
  console.log('\nRead Event Occurred!');
});
```

Import Events Module

Creating object of EventEmitter

Emitting event

Registering Listener and defining event handler

OUTPUT

```
D:\node_example>node test2.js
Welcome to Web Computing Class..!
This is Node.js Tutorial taught by VJsh Nair.
*******

Read Event Occured
```

Event Loop

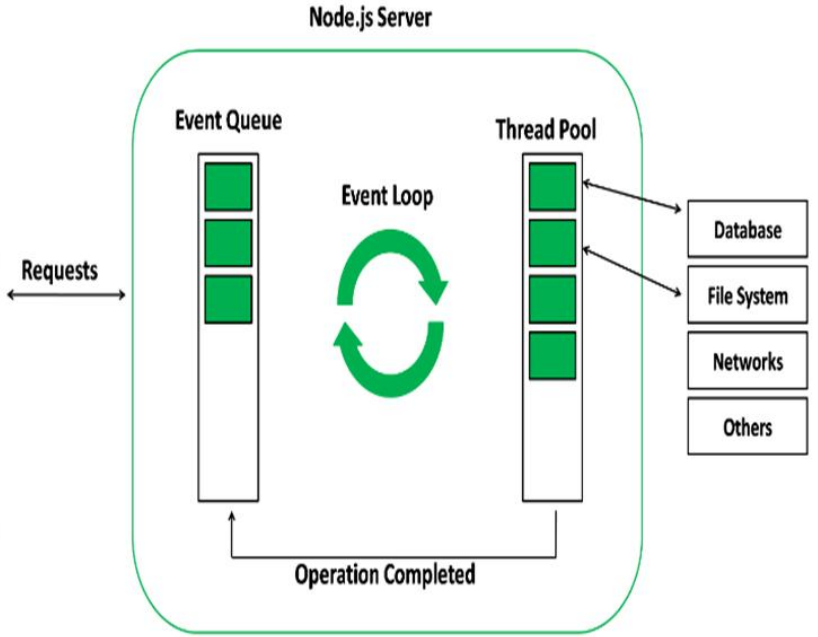
- The ***event loop*** allows Node.js to perform non-blocking I/O operations despite the fact that JavaScript is single-threaded.
- It is done by assigning operations to the operating system whenever and wherever possible.
- Most operating systems are multi-threaded and hence can handle multiple operations executing in the background.
- When one of these operations is completed, the kernel tells Node.js, and the respective callback assigned to that operation is added to the event queue which will eventually be executed.

Features of Event Loop:

- An event loop is an endless loop, which waits for tasks, executes them, and then sleeps until it receives more tasks.
- The event loop executes tasks from the event queue only when the call stack is empty i.e. there is no ongoing task.
- The event loop executes the tasks starting from the oldest first.

Working of Event Loop

- When Node.js starts, it initializes the event loop, processes the provided input script which may make async API calls, schedules timers, then begins processing the event loop.
- When using Node.js, a special library module called *libuv* is used to perform async operations.
- This library is also used, together with the back logic of Node, to manage a special thread pool called the *libuv* thread pool. This thread pool is composed of four threads used to delegate operations that are too heavy for the event loop. I/O operations, Opening and closing connections, setTimeouts are examples of such operations.



- When the thread pool completes a task, a callback function is called which handles the error(if any) or does some other operation. This callback function is sent to the event queue. When the call stack is empty, the event goes through the event queue and sends the callback to the call stack.

Event Loop

```
console.log("This is the first statement");

setTimeout(function(){
    console.log("This is the second statement");
}, 1000);

console.log("This is the third statement");
```

- In the example, the first console log statement is pushed to the call stack, and "*This is the first statement*" is logged on the console, and the task is popped from the stack.

Output:

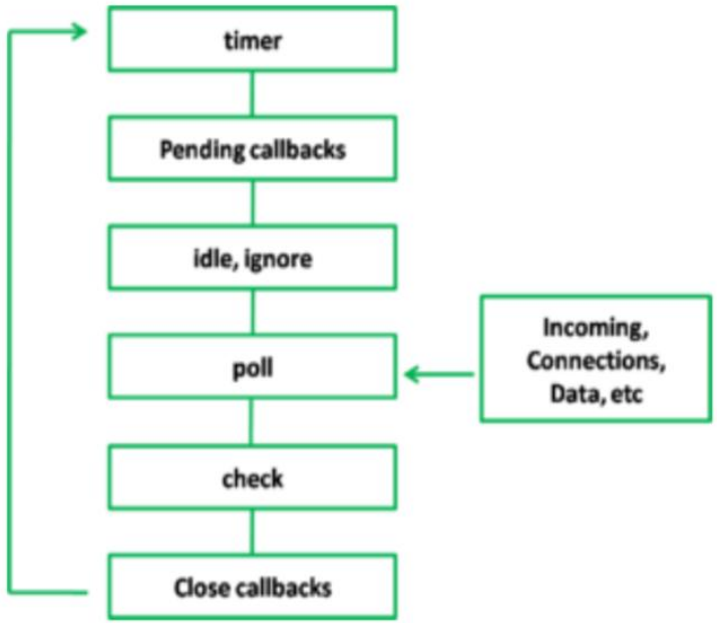
```
D:\node_example>node sam.js
This is the first statement
This is the third statement
This is the second statement
```

- Next, the *setTimeout* is pushed to the queue and the task is sent to the Operating system and the timer is set for the task.
- This task is then popped from the stack. Next, the third console log statement is pushed to the call stack, and "*This is the third statement*" is logged on the console and the task is popped from the stack.
- When the timer set by the *setTimeout* function (in this case 1000 ms) runs out, the callback is sent to the event queue.
- The event loop on finding the call stack empty takes the task at the top of the event queue and sends it to the call stack.
- The callback function for the *setTimeout* function runs the instruction and "*This is the second statement*" is logged on the console and the task is popped from the stack.

Phases of the Event Loop

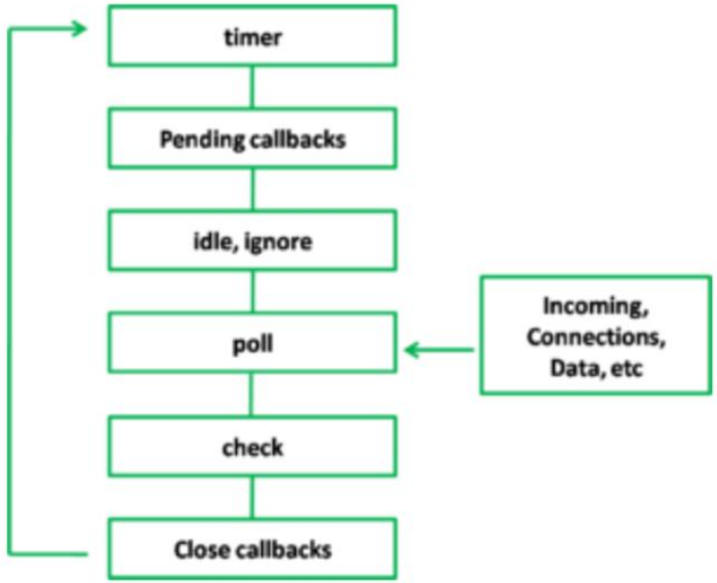
- The event loop in Node.js consists of several phases, each of which performs a specific task. These phases include:

1. **Timers:** This phase processes timers that have been set using *setTimeout()* and *setInterval()*.
2. **Pending Callbacks:** This phase processes any callbacks that have been added to the message queue by asynchronous functions.
3. **Idle, Prepare:** The "idle.prepare" phase is not a standard phase of the event loop in Node.js. It means it's Used internally only. The "idle" phase is a period of time during which the event loop has nothing to do and can be used to perform background tasks, such as running garbage collection or checking for low-priority events.
   - "idle.ignore" is not an official phase of the event loop, it is a way to ignore the idle phase, meaning that it will not use the time of the idle phase to perform background tasks.



Phases of the Event Loop

4. **Poll:** This phase is used to check for new I/O events and process any that have been detected.
5. **Check:** This phase processes any setImmediate() callbacks that have been added to the message queue.
6. **Close Callbacks:** This phase processes any callbacks that have been added to the message queue by the close event of a socket. This means that any code that needs to be executed when a socket is closed is placed in the message queue and processed during this phase.



- It's important to note that the order of execution of these phases can vary depending on the specific implementation of the event loop, but generally, the event loop will process them in the order mentioned above.
- Each phase is executed in order, and the event loop will continue to cycle through these phases until the message queue is empty.

Read-Eval-Print-Loop (REPL)

- The Node.js *Read-Eval-Print-Loop (REPL)* is an interactive shell that processes Node.js expressions.
- The shell *reads* JavaScript code the user enters, *evaluates* the result of interpreting the line of code, *prints* the result to the user, and *loops* until the user signals to quit.
- The REPL is bundled with every Node.js installation and allows you to quickly test and explore JavaScript code within the Node environment without having to store it in a file.

```
D:\node_example>node
Welcome to Node.js v18.17.1.
Type ".help" for more information.
> 10 + 20
30
> 'Hello' + "World"
'HelloWorld'
> console.log("Hi! APSIT.")
Hi! APSIT.
undefined
```

- **Read** – Reads user's input, parses the input into JavaScript data-structure, and stores in memory.
- **Eval** – Takes and evaluates the data structure.
- **Print** – Prints the result.
- **Loop** – Loops the above command until the user presses **ctrl-c** twice.

Read-Eval-Print-Loop (REPL)

Use Variables

You can make use variables to store values and print later like any conventional script. 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()**.

```
D:\node_example>node
Welcome to Node.js v18.17.1.
Type ".help" for more information.
> x = 10
10
> var y = 20
undefined
> x + y
30
```

Node.js Multiline expressions

Node REPL supports multiline expressions like JavaScript. See the following do-while loop example:

```
D:\node_example>node
Welcome to Node.js v18.17.1.
Type ".help" for more information.
> var x = 0
undefined
> do {
... x++;
... console.log("x: " + x);
... }
... while(x<5);
x: 1
x: 2
x: 3
x: 4
x: 5
undefined
```

Node.js Underscore Variable

You can also use underscore _ to get the last result.

```
D:\node_example>node
Welcome to Node.js v18.17.1.
Type ".help" for more information.
> var a = 10
undefined
> var b = 23
undefined
> a+b
33
> var sum = _
undefined
> console.log(sum)
33
undefined
```

Read-Eval-Print-Loop (REPL)

Node.js REPL Commands

Commands	Description
ctrl + c	It is used to terminate the current command.
ctrl + c twice	It terminates the node repl.
ctrl + d	It terminates the node repl.
up/down keys	It is used to see command history and modify previous commands.
tab keys	It specifies the list of current command.
.help	It specifies the list of all commands.
.break	It is used to exit from multi-line expressions.
.clear	It is used to exit from multi-line expressions.
.save filename	It saves current node repl session to a file.
.load filename	It is used to load file content in current node repl session.

.help

To list all the available commands, use the `.help` command:

```
D:\node_example>node
Welcome to Node.js v18.17.1.
Type ".help" for more information.
> .help
.break    Sometimes you get stuck, this gets you out
.clear    Alias for .break
.editor   Enter editor mode
.exit     Exit the REPL
.help     Print this help message
.load     Load JS from a file into the REPL session
.save     Save all evaluated commands in this REPL session to a file

Press Ctrl+C to abort current expression, Ctrl+D to exit the REPL
```

Read-Eval-Print-Loop (REPL)

.save and .load

The `.save` command stores all the code you ran since starting the REPL, into a file. The `.load` command runs all the JavaScript code from a file inside the REPL.

.break/.clear

Using `.break` or `.clear`, it's easy to exit a multi-line expression. For example, begin a `for loop` as follows:

```
D:\>node
Welcome to Node.js v13.14.0.
Type ".help" for more information.
> for (let i=0; i<7;i++){
... i
... .clear
> for (let i=0; i<7;i++){
..... console.log(i);
..... .break
>
```

Quit the session using the `.exit` command or with the `CTRL+D` shortcut. Now start a new REPL with `node`. Now only the code you are about to write will be saved.

```
D:\>node
Welcome to Node.js v13.14.0.
Type ".help" for more information.
> let name = "Johny";
undefined
> let age = 12;
undefined
> `${name} is ${age} years old.`
'Johny is 12 years old.'
> .exit

D:\>
```

```
D:\>node
Welcome to Node.js v13.14.0.
Type ".help" for more information.
> function greet(name){
... console.log("Hello, " + name);
... }
undefined
> greet("Kevin");
Hello, Kevin
undefined
> .save main.js
Session saved to: main.js
>
(To exit, press ^C again or ^D or type .exit)
>

D:\>type main.js
function greet(name){
console.log("Hello, " + name);
}
greet("Kevin");
D:\>
```

```
D:\>node
Welcome to Node.js v13.14.0.
Type ".help" for more information.
> .load main.js
function greet(name){
console.log("Hello, " + name);
}
greet("Kevin");
Hello, Kevin
undefined
> greet("Javascript");
Hello, Javascript
undefined
>
```

Node.js Modules

01	02	03	04	05	06
NPM	GLOBALS	FILE SYSTEM	CALLBACKS	EVENT	NET

Net Module

Node.js **net** module is used to create both servers and clients. This module provides an asynchronous network wrapper and it can be imported using the following syntax.

```
var net = require("net")
```

Sr.No.	Method & Description
1	**net.createServer([options][, connectionListener])** Creates a new TCP server. The connectionListener argument is automatically set as a listener for the 'connection' event.
2	**net.connect(options[, connectionListener])** A factory method, which returns a new 'net.Socket' and connects to the supplied address and port.
3	**net.createConnection(options[, connectionListener])** A factory method, which returns a new 'net.Socket' and connects to the supplied address and port.
4	**net.connect(port[, host][, connectListener])** Creates a TCP connection to port on host. If host is omitted, 'localhost' will be assumed. The connectListener parameter will be added as a listener for the 'connect' event. It is a factory method which returns a new 'net.Socket'.
5	**net.createConnection(port[, host][, connectListener])** Creates a TCP connection to port on host. If host is omitted, 'localhost' will be assumed. The connectListener parameter will be added as a listener for the 'connect' event. It is a factory method which returns a new 'net.Socket'.
6	**net.connect(path[, connectListener])** Creates Unix socket connection to path. The connectListener parameter will be added as a listener for the 'connect' event. It is a factory method which returns a new 'net.Socket'.
7	**net.createConnection(path[, connectListener])** Creates Unix socket connection to path. The connectListener parameter will be added as a listener for the 'connect' event. It is a factory method which returns a new 'net.Socket'.
8	**net.isIP(input)** Tests if the input is an IP address. Returns 0 for invalid strings, 4 for IP version 4 addresses, and 6 for IP version 6 addresses.
9	**net.isIPv4(input)** Returns true if the input is a version 4 IP address, otherwise returns false.
10	**net.isIPv6(input)** Returns true if the input is a version 6 IP address, otherwise returns false.

Class - net.Server

This class is used to create a TCP or local server.

Sr.No.	Events & Description
1	**listening** Emitted when the server has been bound after calling server.listen.
2	**connection** Emitted when a new connection is made. Socket object, the connection object is available to event handler. Socket is an instance of net.Socket.
3	**close** Emitted when the server closes. Note that if connections exist, this event is not emitted until all the connections are ended.
4	**error** Emitted when an error occurs. The 'close' event will be called directly following this event.

Sr.No.	Method & Description
1	**server.listen(port[, host][, backlog][, callback])** Begin accepting connections on the specified port and host. If the host is omitted, the server will accept connections directed to any IPv4 address (INADDR_ANY). A port value of zero will assign a random port.
2	**server.listen(path[, callback])** Start a local socket server listening for connections on the given path.
3	**server.listen(handle[, callback])** The handle object can be set to either a server or socket (anything with an underlying _handle member), or a {fd: <n>} object. It will cause the server to accept connections on the specified handle, but it is presumed that the file descriptor or handle has already been bound to a port or domain socket. Listening on a file descriptor is not supported on Windows.
4	**server.listen(options[, callback])** The port, host, and backlog properties of options, as well as the optional callback function, behave as they do on a call to server.listen(port, [host], [backlog], [callback]) . Alternatively, the path option can be used to specify a UNIX socket.
5	**server.close([callback])** Finally closed when all connections are ended and the server emits a 'close' event.
6	**server.address()** Returns the bound address, the address family name and port of the server as reported by the operating system.
7	**server.unref()** Calling unref on a server will allow the program to exit if this is the only active server in the event system. If the server is already unrefd, then calling unref again will have no effect.
8	**server.ref()** Opposite of unref, calling ref on a previously unrefd server will not let the program exit if it's the only server left (the default behavior). If the server is refd, then calling the ref again will have no effect.
9	**server.getConnections(callback)** Asynchronously get the number of concurrent connections on the server. Works when sockets were sent to forks. Callback should take two arguments err and count.

Class - net.Socket

- This object is an abstraction of a TCP or local socket.
- *net.Socket* instances implement a duplex Stream interface.
- They can be created by the user and used as a client (with *connect()*) or they can be created by Node and passed to the user through the '*connection*' event of a server.

Sr.No.	Events & Description
1	**lookup** Emitted after resolving the hostname but before connecting. Not applicable to UNIX sockets.
2	**connect** Emitted when a socket connection is successfully established.
3	**data** Emitted when data is received. The argument data will be a Buffer or String. Encoding of data is set by socket.setEncoding().
4	**end** Emitted when the other end of the socket sends a FIN packet.
5	**timeout** Emitted if the socket times out from inactivity. This is only to notify that the socket has been idle. The user must manually close the connection.
6	**drain** Emitted when the write buffer becomes empty. Can be used to throttle uploads.
7	**error** Emitted when an error occurs. The 'close' event will be called directly following this event.
8	**close** Emitted once the socket is fully closed. The argument had_error is a boolean which indicates if the socket was closed due to a transmission error.

Class - net.Socket

Sr.No.	Property & Description
1	**socket.bufferSize** This property shows the number of characters currently buffered to be written.
2	**socket.remoteAddress** The string representation of the remote IP address. For example, '74.125.127.100' or '2001:4860:a005::68'.
3	**socket.remoteFamily** The string representation of the remote IP family. 'IPv4' or 'IPv6'.
4	**socket.remotePort** The numeric representation of the remote port. For example, 80 or 21.
5	**socket.localAddress** The string representation of the local IP address the remote client is connecting on. For example, if you are listening on '0.0.0.0' and the client connects on '192.168.1.1', the value would be '192.168.1.1'.
6	**socket.localPort** The numeric representation of the local port. For example, 80 or 21.
7	**socket.bytesRead** The amount of received bytes.
8	**socket.bytesWritten** The amount of bytes sent.

Sr.No.	Method & Description
1	**new net.Socket([options])** Construct a new socket object.
2	**socket.connect(port[, host][, connectListener])** Opens the connection for a given socket. If port and host are given, then the socket will be opened as a TCP socket, if host is omitted, localhost will be assumed. If a path is given, the socket will be opened as a Unix socket to that path.
3	**socket.connect(path[, connectListener])** Opens the connection for a given socket. If port and host are given, then the socket will be opened as a TCP socket, if host is omitted, localhost will be assumed. If a path is given, the socket will be opened as a Unix socket to that path.
4	**socket.setEncoding([encoding])** Set the encoding for the socket as a Readable Stream.
5	**socket.write(data[, encoding][, callback])** Sends data on the socket. The second parameter specifies the encoding in the case of a string--it defaults to UTF8 encoding.
6	**socket.end([data][, encoding])** Half-closes the socket, i.e., it sends a FIN packet. It is possible the server will still send some data.
7	**socket.destroy()** Ensures that no more I/O activity happens on this socket. Necessary only in case of errors (parse error or so).
8	**socket.pause()** Pauses the reading of data. That is, 'data' events will not be emitted. Useful to throttle back an upload.
9	**socket.resume()** Resumes reading after a call to pause().
10	**socket.setTimeout(timeout[, callback])** Sets the socket to timeout after timeout milliseconds of inactivity on the socket. By default, net.Socket does not have a timeout.

Net Module Example

D: > node_example > JS server.js > ...

```
1   var net = require('net');
2   var server = net.createServer(function(connection) {
3       console.log('client connected');
4
5       connection.on('end', function() {
6           console.log('client disconnected');
7       });
8
9       connection.write('Hello World!\r\n');
10      connection.pipe(connection);
11  });
12
13  server.listen(8080, function() {
14      console.log('server is listening');
15  });
```

D: > node_example > JS client.js > ...

```
1   var net = require('net');
2   var client = net.connect({port: 8080}, function() {
3       console.log('connected to server!');
4   });
5
6   client.on('data', function(data) {
7       console.log(data.toString());
8       client.end();
9   });
10
11  client.on('end', function() {
12      console.log('disconnected from server');
13  });
```

Output

```
D:\node_example>node server.js
server is listening
client connected
client disconnected
```

```
D:\node_example>node client.js
connected to server!
Hello World!

disconnected from server
```

Node.js Modules

01	02	03	04	05	06
NPM	GLOBALS	FILE SYSTEM	CALLBACKS	EVENT	NET

07
BUFFERS

Buffer Module

- Buffer refers to space in memory which is used to store data temporarily.
- A buffer has traditionally been used between devices with speed mis-match so that they can keep on operating at their respective speeds without loss of data.
- In Node.js Buffers are used when dealing with file streams or tcp streams which are mainly octets of binary data.
- Node provides Buffer class which provides instances to store raw data similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap.
- Buffer class is a global class that can be accessed in an application without importing the buffer module.
- The buffers module provides a way of handling streams of binary data.

The Buffer object is a global object in Node.js, and it is not necessary to import it using the `require` keyword.

Buffer Methods

There are the *encodings* accepted while creating a buffer. We have listed a few of them :

- ASCII
- UTF-8
- Base64
- Latin1
- Binary
- Hex

- **Buffer.alloc() :** This method is used to create a Buffer object of given length with initializing all the value to `fill` or `0` .
  1. **Syntax :** The syntax of `Buffer.alloc()` method is given below :

```
Buffer.alloc(size[, fill[, encoding]])
```

Where ,

- `size` : Desired length of new Buffer. It accepts `integer` type of data.
- `fill` : The value to prefill the buffer. Default value is `0` .It accepts any of the follwing : `integer` , `string` , `buffer` type of data.
- `encoding`

  2. **Example :** Coding example of `Buffer.alloc()` method is given below :

```
//Name of the file : buffer.alloc.js
var buff = Buffer.alloc(20);
console.log(buff);
```

  3. **Run :** We can run it in the following way :

```
>node buffer.alloc.js
<Buffer 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00>
```

Buffer Methods

- **Buffer.allocUnsafe() :** This method is used to create a Buffer object of given length but it will not initialize the values.Due to which contents of the newly created buffer are not known which causes a security threat because it might contain some sensitive or confidential data.
  1. **Syntax :** The syntax of `Buffer.allocUnsafe()` method is given below :

```
Buffer.allocUnsafe(size)
```

  Where ,
  - `size` : Desired length of new Buffer. It accepts `integer` type of data.
  2. **Example :** Coding example of `Buffer.allocUnsafe()` method is given below :

```
//Name of the file : buffer.allocUnsafe.js
var buff = Buffer.allocUnsafe(10);
console.log(buff);
```

  3. **Run :** We can run it in the following way :

```
>node buffer.allocUnsafe.js
<Buffer 00 00 00 00 08 00 00 00 07 00>
```

Buffer Methods

- **Buffer.from() :** This method is used to create a Buffer from an string, object , array or buffer.
  1. **Syntax :** The syntax of `Buffer.from()` method is given below :

```
Buffer.from(string[, encoding])
```

  Where ,

  - `string` : data is passed here.
  - `encoding` : The encoding of string. Default value is `utf8`. This is an optional parameter.

  2. **Example :** Coding example of `Buffer.from()` method is given below :

```
//Name of the file : buffer.from.js
var buff1 = Buffer.from('Nodejsera');
console.log("buff1 : " + buff1);
```

  3. **Run :** We can run it in the following way :

```
>node buffer.from.js
buff1 : Nodejsera
```

- **buf.compare() :** This method is used to compare buffers. It returns
  - `0` : If both buffers are same
  - `1` : If target buffer comes before the source buffer.
  - `-1` : If source buffer comes before the target buffer.

  1. **Syntax :** The syntax of `buf.from()` method is given below :

```
buf.compare(target[, targetStart[, targetEnd[, sourceStart[, sourceEnd]]]])
```

  Where ,

  - `target` : target buffer.
  - `targetStart` : Position from where comparion begins on target buffer.
  - `targetEnd` : Position on which comparison ends on target buffer.
  - `sourceStart` : Position from where comparion begins on source buffer.
  - `sourceEnd` : Position on which comparison ends on source buffer.

  2. **Example :** Coding example of `buf.compare()` method is given below :

```
//Name of the file : buffer.compare.js
var buffer1 = Buffer.from('Nodejsera');
var buffer2 = Buffer.from('Nodejsera');
var output = buffer1.compare(buffer2);
console.log(output)
if(output < 0) {
   console.log(buffer1 +" comes before " + buffer2);
}else if(output == 0){
   console.log(buffer1 +" is same as " + buffer2);
}else {
   console.log(output +" comes after " + buffer2);
}
```

  3. **Run :** We can run it in the following way :

```
>node buffer.compare.js
0
Nodejsera is same as Nodejsera
```

Buffer Methods

- **Buffer.concat() :** This method is used to concatenate two or more buffers together.
  1. **Syntax :** The syntax of `Buffer.concat()` method is given below :

```
Buffer.concat(list)
```

  Where ,

  - `list` : List of buffers to be concatenated.

  2. **Example :** Coding example of `Buffer.concat()` method is given below :

```
//Name of the file : buffer.concat.js
var buff1 = Buffer.from('Nodejsera for nodejs');
var buff2 = Buffer.from('- 30 days of node');
var buff3 = Buffer.concat([buff1,buff2]);
console.log("buff3 content: " + buff3.toString());
```

  3. **Run :** We can run it in the following way :

```
>node buffer.concat.js
buff3 content: Nodejsera for nodejs- 30 days of node
```

- **buf.copy() :** This method is used to copy specified amount of bytes from source buffer to target buffer.
  1. **Syntax :** The syntax of `buf.copy()` method is given below :

```
buf.copy(target[, targetStart[, sourceStart[, sourceEnd]]])
```

  Where ,

  - `target` : buffer to which we need to copy.
  - `targetStart` : Position from where copy starts from source.
  - `sourceStart` : position from which copy starts.
  - `sourceEnd` : Position till which copy is done.

  2. **Example :** Coding example of `buf.copy()` method is given below :

```
//Name of the file : buffer.copy.js
var buff = Buffer.from('Nodejsera');
var newbuff = Buffer.alloc(20);
buff.copy(newbuff);
console.log("Content of newbuff :  " + newbuff.toString());
```

  3. **Run :** We can run it in the following way :

```
>node buffer.copy.js
Content of newbuff :  Nodejsera
```

Buffer Methods

- **buf.equals() :** This method is used to compare 2 buffers. It returns `true` if buffers match, otherwise it will return `false` .
  1. **Syntax :** The syntax of `buf.equals()` method is given below :

```
buf.equals(otherBuffer)
```

Where ,

- `otherBuffer` : Buffer to compare with.

2. **Example :** Coding example of `buf.equals()` method is given below :

```
//Name of the file : buffer.equals.js
var buff1 = Buffer.from('nodejsera');
var buff2 = Buffer.from('nodejsera');

console.log(buff1.equals(buff2));
```

3. **Run :** We can run it in the following way :

```
>node buffer.equals.js
true
```

- **buf.fill() :** This method is used to fill the buffer with a specified value.
  1. **Syntax :** The syntax of `buf.fill()` method is given below :

```
buf.fill(value)
```

Where ,

- `value` : The value with which we will fill the buffer.

2. **Example :** Coding example of `buf.fill()` method is given below :

```
//Name of the file : buffer.fill.js
var buff = Buffer.allocUnsafe(10).fill('nj');
console.log(buff.toString());
```

3. **Run :** We can run it in the following way :

```
>node buffer.fill.js
njnjnjnjnj
```

Buffer Methods

- **buf.indexOf() :** This method is used to check whether the buffer contains a specified value. If the value is present it will return the index of first occurrence of the value, otherwise it will return `-1` .
  1. **Syntax :** The syntax of `buf.indexOf()` method is given below :

```
buf.indexOf(value)
```

  Where ,

  - `value` : the value we are looking for.

  2. **Example :** Coding example of `buf.indexOf()` method is given below :

```
//Name of the file : buffer.indexOf.js
var buff1 = Buffer.from('Nodejsera');
console.log(buff1.indexOf('j'));
```

  3. **Run :** We can run it in the following way :

```
>node buffer.indexOf.js
4
```

- **buf.length :** This method is used to return the length of buffer object.
  1. **Syntax :** The syntax of `buf.length` method is given below :

```
buf.length
```

  2. **Example :** Coding example of `buf.length` method is given below :

```
//Name of the file : buffer.length.js
var buf = Buffer.from('Nodejsera for nodejs');
console.log( buf.length);
```

  3. **Run :** We can run it in the following way :

```
>node buffer.length.js
20
```

Buffer Methods

- **Buffer.slice() :** This method is used to slice the buffer into a new buffer with specified start and end point.

1. **Syntax :** The syntax of `buf.slice()` method is given below :

```
buf.slice([start[, end]])
```

Where ,

- `start` : start offset of new buffer.
- `end` : end offset of new buffer.

2. **Example :** Coding example of `buf.slice()` method is given below :

```
//Name of the file : buffer.slice.js
var buff1 = Buffer.from('Nodejsera');
var buff2 = buff1.slice(0,5);
console.log("content of buff2 : " + buff2.toString());
```

3. **Run :** We can run it in the following way :

```
>node buffer.slice.js
content of buff2 : Nodej
```

- **buf.toJSON() :** This method is used to convert the buffer into JSON.

1. **Syntax :** The syntax of `buf.toJSON()` method is given below :

```
buf.toJSON()
```

2. **Example :** Coding example of `buf.toJSON()` method is given below :

```
//Name of the file : bufferToJSON.js
var buf = Buffer.from('Nodejsera');
var json = buf.toJSON(buf);
console.log(json);
```

3. **Run :** We can run it in the following way :

```
>node bufferToJSON.js
{ type: 'Buffer',
  data: [ 78, 111, 100, 101, 106, 115, 101, 114, 97 ] }
```

- **buf.toString() :** This method is used to convert the buffer into string.

1. **Syntax :** The syntax of `buf.toString()` method is given below :

```
buf.toString([encoding[, start[, end]]])
```

Where ,

- `encoding` : Character encoding to decode to.
- `start` : starting offset.
- `end` : ending offset.

2. **Example :** Coding example of `buf.toString()` method is given below :

```
//Name of the file : bufferToString.js
var buf = Buffer.from('Nodejsera for nodejs');
console.log( buf.toString('ascii'));
```

3. **Run :** We can run it in the following way :

```
>node bufferToString.js
Nodejsera for nodejs
```

Node.js Modules

01	02	03	04	05	06
NPM	GLOBALS	FILE SYSTEM	CALLBACKS	EVENT	NET

07	08
BUFFERS	STREAMS

Stream Module

- Streams are used to handle streaming data in node.js
- Streams can be readable, writable or both.
- All streams are instances of `eventEmitter` class.
- We can use the `stream` module via requiring it in the following way :

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

Types of streams

There are four types of streams which are as follows :

1. **Readable stream :** The streams which is used to perform read operations are readable streams.
2. **Writable stream :** The streams which is used to perform write operations are writable streams.
3. **Duplex stream :** Duplex streams are the streams which implements both readable and writable stream.
4. **Transform stream :** Transform streams are duplex streams that can transform or modify data as it is read and written. Also, In transform stream output is in some way related to the input.

Readable stream

The streams which is used to perform read operations are readable streams.All aspects of readable streams are explained below :

- **Modes:** These are the two modes of readables
  1. **paused :**
     - If the readable is in paused mode, then we need to call `stream.read()` explicitly to read the chunks of data.
     - By default, all readable streams are in paused mode.
     - We can switch readable to pause mode by calling stream.pause() method when there are no pipe destinations
     - We can also call stream.unpipe() method when pipe destinations are available , in order to switch readable to pause mode.
  2. **flowing :**
     - If the readable is in flowing mode, then the data is successfully emitted.
     - We can switch the readable stream to flowing mode by calling stream.resume() method.
     - We can switch the readable stream to flowing mode by calling stream.pipe() method.
     - If the readable is in flowing mode and there is no consumer to handle the data then it can lead to data loss.

- **Examples:** Examples of methods or modules which uses readable streams directly or in the form of duplex/transform stream are as follows :
  - HTTP requests (Server)
  - HTTP responses (Client)
  - fs module read streams
  - zlib module
  - crypto module
  - TCP sockets
  - process.stdin
- **Events :**
  - **readable :** This event is fired when there is data available to be read from the stream.
  - **data :** This event is fired when the stream is vacating the ownership of the chunk of data to the consumer.
  - **error :** This event is fired when the stream is unable to generate data due to some internal error or when stream tries to push invalid chunk of data.
  - **close :** This event is fired when the stream is closed. It indicates that no more events will be emitted and no further computation will occur.
  - **end :** This event is fired when all the data is read. It indicates that there is no more data to be consumed.

Readable stream

- Methods :
  1. `readable.pause()` : This method is used to change the mode of the stream from `flowing` to `paused` and also all the data availble keeps residing in the internal buffer.
  2. `readable.resume()` : This method is used to change the mode of the stream from `paused` to `flowing` and also stream will resume emitting events.
  3. `readable.isPaused()` : This method is used to check the current operating state of the readable stream. If it returns `true` then that signifies that readable stream is in paused mode.
  4. `readable.pipe()` : This method is used to attach a writable stream to the readable which will make the stream switch to flowing mode and start pushing data to the attached writable.
  5. `readable.unpipe()` : This method is used to detach the writable stream previously attached to the readable stream.
  6. `readable.read()` : This method is used to pull the data out of the internal buffer where data is returned in the form of buffers unless any other format is specified using `readable.setEncoding()` . If there is no data to pull , then null is returned.
  7. `readable.setEncoding()` : This method is used to set the encoding for readable stream. By default the data is pulled in the form of buffers.
  8. `readable.unshift()` : This method is used to push the data back to the internal buffer.
  9. `readable.wrap()` : This method is used to read the data from the readables where the data sources uses the old streams.
  10. `readable.destroy()` : This method is used to signifies the end of readable stream and stream releases any resources , if held.

Writable stream

The streams which is used to perform write operations are writable streams. All aspects of writable streams are explained below :

- **Examples:** Examples of methods or modules which uses writable streams directly or in the form of duplex/transform stream are as follows :
  - HTTP requests (Client)
  - HTTP responses (Server)
  - fs module write streams
  - zlib module
  - crypto module
  - TCP sockets
  - process.stdout
  - process.stderr

- **Events :**
  - **drain :** This event is fired when a call to `system.write(chunk)` method returns false and it indicates when it will be appropriate to resume writing data.
  - **pipe :** This event is fired when `stream.pipe()` method is called on a readable stream indicating the addition of the writable in the set of destinations of the readable.
  - **unpipe :** This event is fired when `stream.unpipe()` method is called on a readable stream indicating the removal of the writable from the set of destinations of the readable.
  - **error :** This event is fired when an error occured while writing or piping the data.
  - **close :** This event is fired when the stream is closed. It indicates that no more events will be emitted and no further computation will occur.
  - **finish :** This event is fired when all the data is successfully flushed.

- **Methods :**
  1. `writable.cork()` : This method is used to force all the written data to be buffered in memory. This buffered data is flushed in either of the following scenarios :
     - `stream.uncork()` method is called.
     - `stream.end()` method is called.
  2. `writable.uncork()` : This method is used to flush all the data buffered by `stream.cork()` method.
  3. `writable.write()` : This method is used to write some data to the stream and call the given callback when the data is handled successfully.
  4. `writable.setDefaultEncoding()` : This method is used to set the default encoding for the writable stream.
  5. `writable.end()` : This method is used to signifies that no more data will be written to the writable stream.
  6. `writable.destroy()` : This method is used to signifies the end of writable stream.

Duplex stream

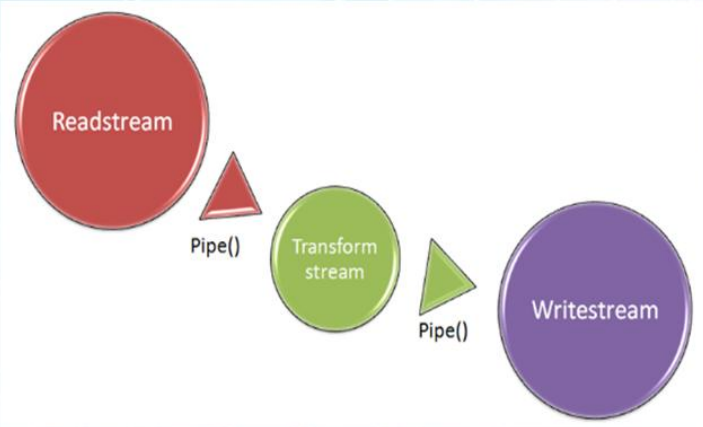
`Duplex streams` are the streams which implements both `readable` and `writable` streams simultaneously.Most common example of `duplex` stream include `net.socket` class of `net` module. A better explanation of how duplex streams works is as follows :

Suppose we build a socket in node.js to implement the functionality of transmit and receive data simulataneously, then that can be achieved using `duplex` stream. We will be having two independent channels in the network where one channel is used for transmitting data and other for receiving data.

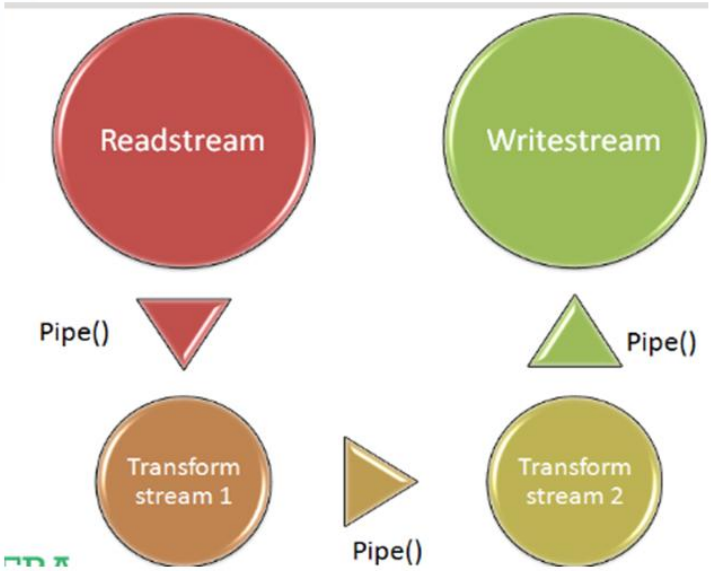
- **Examples:** Examples of methods or modules which uses duplex streams are as follows :
  - Sockets (TCP) : It uses `duplex` streams for implementing sockets.
  - zlib : It uses `duplex` streams for gzip compression and decompression.
  - crypto : It used `duplex` stream for performing encryption, decryption and creating message digests.

Transform stream

`Transform` streams are `duplex` streams that can transform or modify data as it is read and written. Also where output is in some way related to the input. These streams read the input data , transform it using the manipulating function and output the new data as shown below :



We can also chain streams together to create complex processes by piping one to next as shown below :



Transform stream

- **Examples:** Examples of methods or modules which uses transform streams are as follows :
  - zlib : It uses `transform` streams for gzip compression and decompression like in `zlib.createDeflate()` method.
  - crypto : It used `transform` stream for performing encryption, decryption and creating message digests.

- **Methods :**
  1. `transform.destroy()` : This method is used to destroy the stream and emit `error` . Moreover , The `tranform` stream would release all internal resouces being used after this method call.

Example

D: > node_example > JS filename_streams.js > ...

```
1  // require fs module for file system
2  var fs = require('fs');
3  // write data to a file using writeable stream
4  var wdata = "I am working with streams for the first time";
5
6  var myWriteStream = fs.createWriteStream('aboutMe.txt');
7
8  // write data
9
10 myWriteStream.write(wdata);
11
12 // done writing
13 myWriteStream.end();
14
15 // write handler for error event
16 myWriteStream.on('error', function(err){
17    console.log(err);
18 });
19
20 myWriteStream.on('finish', function() {
21     console.log("data written successfully using streams.");
22     console.log("Now trying to read the same file using read streams ");
23     var myReadStream = fs.createReadStream('aboutMe.txt');
24     // add handlers for our read stream
25     var rContents = '' // to hold the read contents;
26     myReadStream.on('data', function(chunk) {
27         rContents += chunk;
28     });
29     myReadStream.on('error', function(err){
30         console.log(err);
31     });
32     myReadStream.on('end',function(){
33         console.log('read: ' + rContents);
34     });
35     console.log('performed write and read using streams');
36
37 });
```

Output:

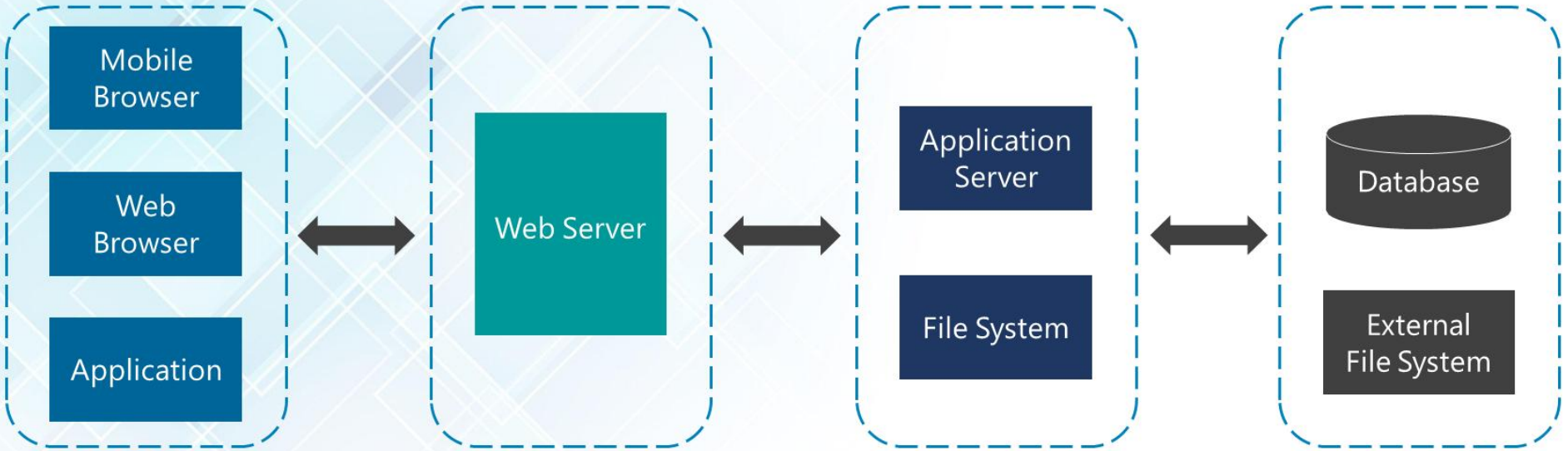
```
D:\node_example>node filename_streams.js
data written successfully using streams.
Now trying to read the same file using read streams
performed write and read using streams
read: I am working with streams for the first time
```

Node.js Modules

01	02	03	04	05	06
NPM	GLOBALS	FILE SYSTEM	CALLBACKS	EVENT	NET

07	08	09
BUFFERS	STREAMS	WEB/HTTP

Web Application Architecture



Client	Server	Business Layer	Data Layer
Makes HTTP Request to the server	Serve the request Made by client and pass the response	contains application utilized by web server to do required processing.	Consists of Database and other data storage

HTTP

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

http.createServer( function (request, response) {

    var pathname = url.parse(request.url).pathname;

    console.log("Request for " + pathname + " received.");

    fs.readFile(pathname.substr(1), function (err, data) {
       if (err) {

          console.log(err);
          response.writeHead(404, {'Content-Type': 'text/html'});

       }else{
          response.writeHead(200, {'Content-Type': 'text/html'});
          response.write(data.toString());
       }
       response.end();
    });
}).listen(3000);

console.log('Server running at localhost:3000');
```

Import Required Modules

Creating Server

Parse the fetched URL to get pathname

Request file to be read from file system (index.html)

Creating Header with content type as text or HTML

Generating Response

Listening to port: 3000

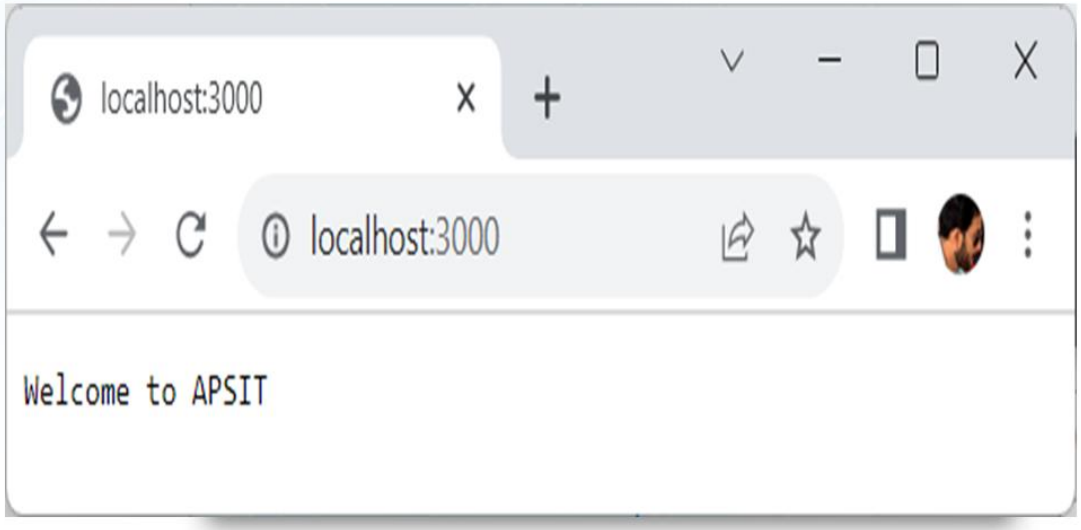
HTTP

D: > node_example > JS second_example.js > ...

```
1   var http = require('http');
2
3   var port = 3000;
4
5   var server = http.createServer(function (request, response) {
6
7       response.writeHead(200, {'Content-Type':'text/plain'});
8
9       response.end('Welcome to APSIT\n');
10
11  })
12
13  server.listen(port,function() {
14      console.log('Server running at http://localhost:3000');
15  });
```

```
D:\node_example>node second_example.js
Server running at http://localhost:3000
```

Node.js cmd



Output


Thank You!