



Content

- Node.js Introduction
- Get Started
- Modules
- HTTP Module
- File System
- URL Module

Content

- NPM
- Events
- Upload files
- Email

What is Node.js

- Node.js is an **open source** server environment built on Google's Chrome JS engine (V8)
- Node.js was developed by **Ryan Dahl** in **2009**
- 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 official definition

- Node.js is a platform built on **Chrome's JavaScript runtime** for easily building **fast** and **scalable** network applications.
- Node.js uses an **event-driven, non-blocking I/O** model that makes it **lightweight** and **efficient**, perfect for **data-intensive real-time** applications that run across distributed devices

Features of Node.js

- Asynchronous programming and Event-driven
- Very fast (built-on Google Chrome's V8 JS engine)
- Single threaded but Highly scalable
- Makes non-blocking calls
- Memory efficient
- No buffering
- MIT License

Functions of Node.js

- can generate **dynamic page content**
- can create, open, read, write, delete, and close files on the server (**file operations**)
- can **collect form data**
- can add, delete, modify data in your database (**CRUD operations**)

Where to use Node.js

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

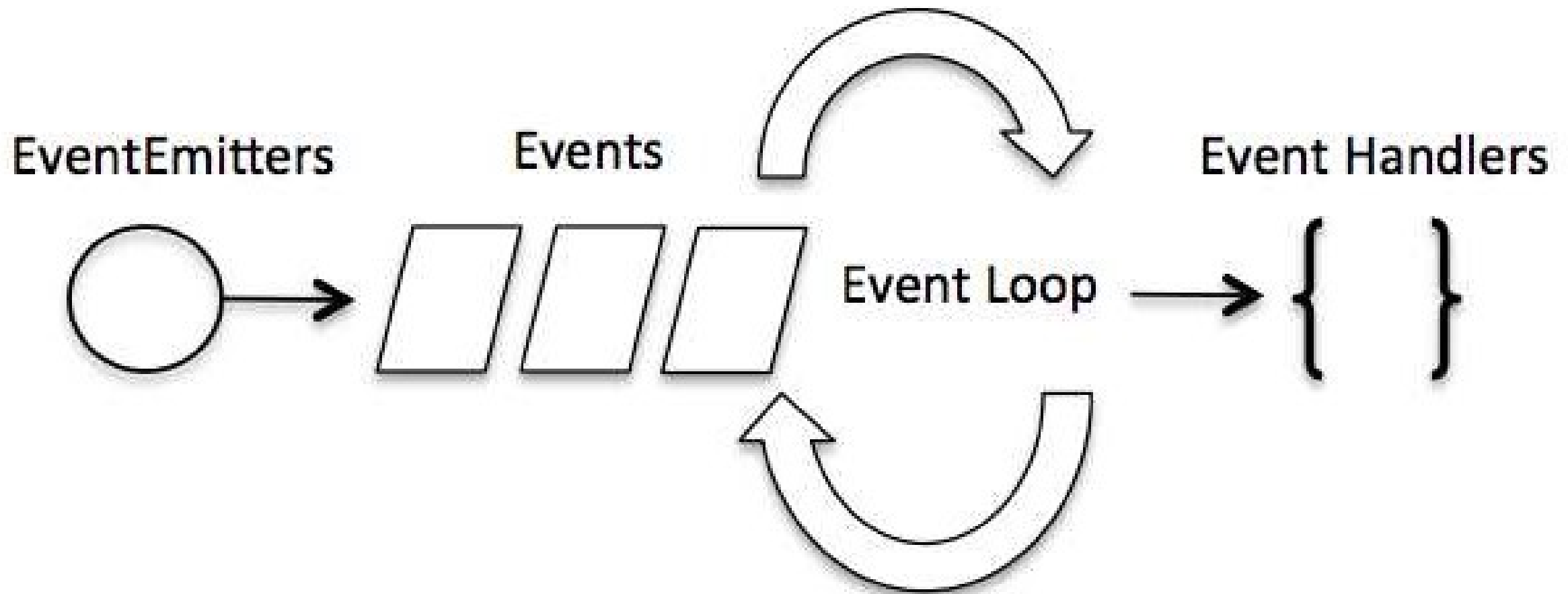
Where not to use Node.js

- CPU intensive applications

Big Companies using Node.js

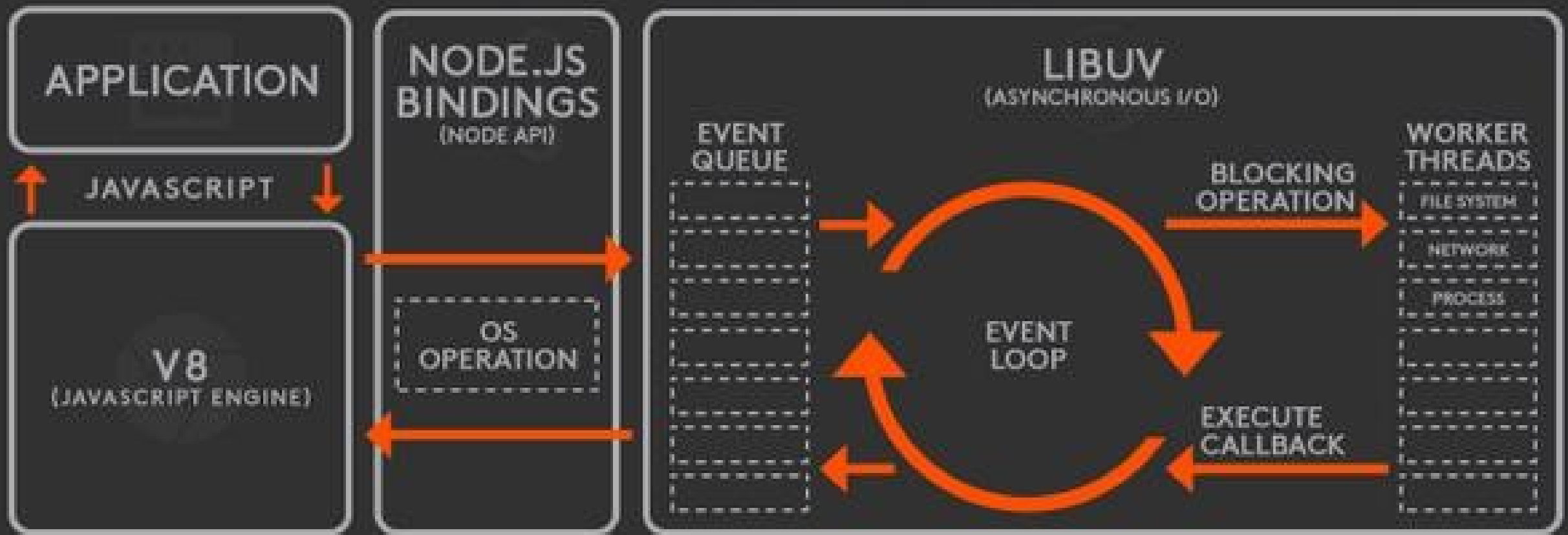
NetFlix	Medium
Walmart.com	duckduckgo.com
LinkedIn	divar.ir
Uber – App	coursera.org
Paypal	app.plularsight.com
Ebay	flickr.com

Event Driven Programming



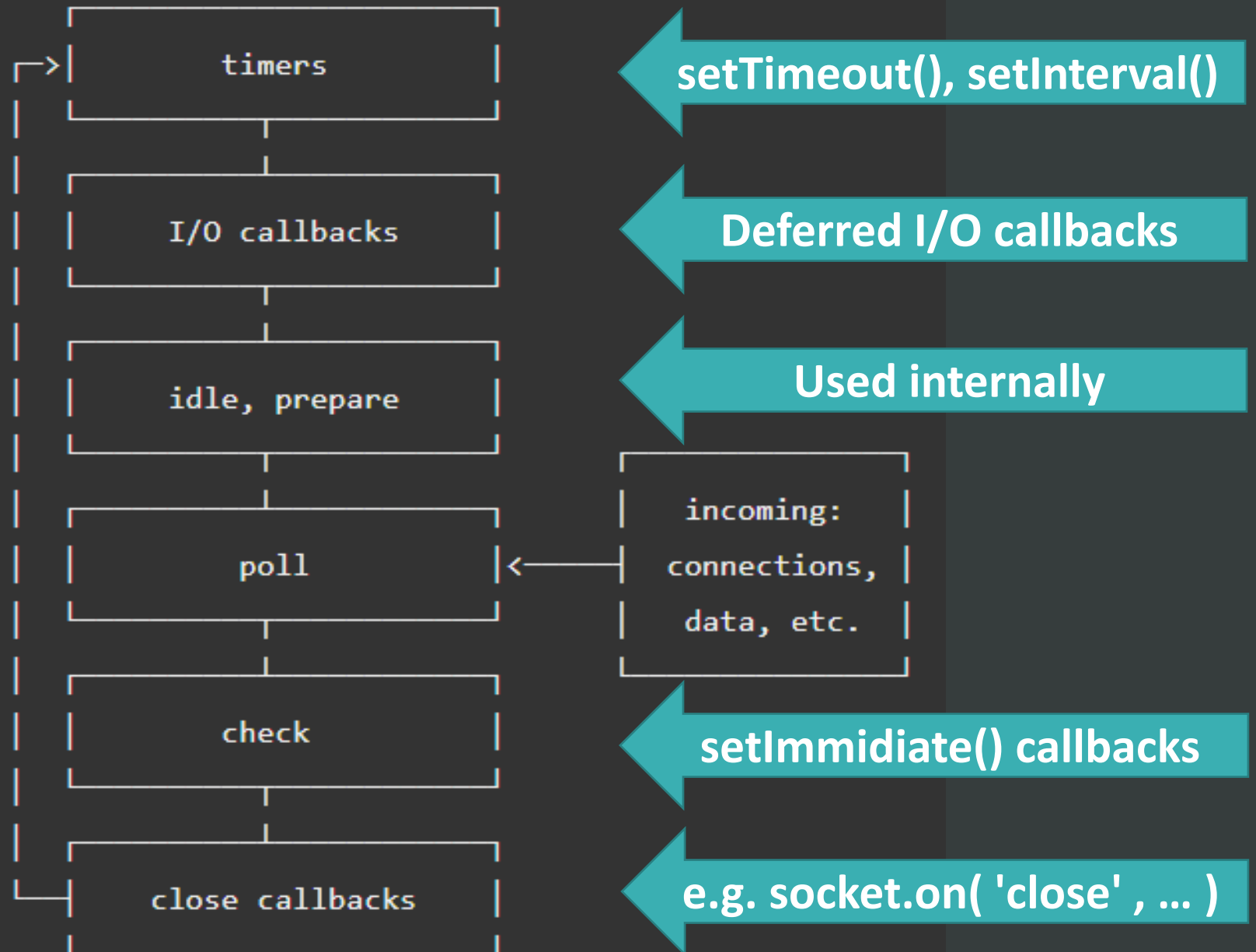
THE NODE.JS SYSTEM

A DIVISION FROM
MODULES



<https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/>

Node.js Event Loop



Phases Overview

- **timers**: executes callbacks scheduled by `setTimeout()` and `setInterval()`.
- **pending callbacks**: executes I/O callbacks deferred to the next loop iteration.
- **idle, prepare**: only used internally.
- **poll**: retrieve new I/O events; execute I/O related callbacks (almost all with the exception of close callbacks, the ones scheduled by timers, and `setImmediate()`); node will block here when appropriate.
- **check**: `setImmediate()` callbacks are invoked here.
- **close callbacks**: some close callbacks, e.g. `socket.on('close', ...)`.

Timers Phase

- A timer specifies the **threshold** *after which* a provided **callback** *may be executed*.

Pending Callbacks Phase

- This phase executes **callbacks** for some **system operations** such as types of TCP errors.
- For example if a TCP socket receives ECONNREFUSED when attempting to connect, some *nix systems want to wait to report the error.
- This will be queued to execute in the pending callbacks phase.

Poll Phase

- The poll phase has two main functions:
 1. Calculating how long it should block and poll for I/O, then
 2. Processing events in the poll queue.

Poll Phase

- When the event loop enters the poll phase and there are no timers scheduled, one of two things will happen:
 1. If the **poll queue is not empty**, the event loop will iterate through its queue of callbacks executing them synchronously until either the queue has been exhausted, or the system-dependent hard limit is reached.
 2. If the **poll queue is empty**, one of two more things will happen:
 1. If scripts have been scheduled by **setImmediate()**, the event loop will end the poll phase and continue to the check phase to execute those scheduled scripts.
 2. If scripts have not been scheduled by **setImmediate()**, the event loop will wait for callbacks to be added to the queue, then execute them immediately.

Poll Phase

- Once the **poll** queue is empty the event loop will check for timers *whose time thresholds have been reached*.
- If one or more timers are ready, the event loop will wrap back to the **timers** phase to execute those timers' callbacks.

Check Phase

- If the poll phase becomes idle and scripts have been queued with `setImmediate()`, the event loop may continue to the check phase rather than waiting.
- `setImmediate()` is actually a special timer that runs in a separate phase of the event loop. It uses a libuv API that schedules callbacks to execute after the poll phase has completed.
- Generally, as the code is executed, the event loop will eventually hit the poll phase where it will wait for an incoming connection, request, etc. However, if a callback has been scheduled with `setImmediate()` and the poll phase becomes idle, it will end and continue to the check phase rather than waiting for poll events.

Close Callabacks Phase

- If a socket or handle is closed abruptly (e.g. `socket.destroy()`), the 'close' event will be emitted in this phase. Otherwise it will be emitted via `process.nextTick()`.

process.nextTick()

- Every time the event loop takes a full trip (processes all the callback of the current queue), we call it a tick.
- When we pass a function to `process.nextTick()`, we instruct the engine to invoke this function at the end of the current operation, before the next event loop tick starts:

```
process.nextTick( () => {  
    ...  
})
```

- The event loop is busy processing the current function code.
- When this operation ends, the JS engine runs all the functions passed to nextTick calls during that operation.
- It's the way we can tell the JS engine to process a function asynchronously (after the current function), but as soon as possible, not queue it.

event_loop1.js

```
setImmediate(() => {  
    console.log("immediate")  
});  
  
setTimeout(() => {  
    console.log("timeout")  
},0);
```

Output (In-deterministic)

timeout
immediate

OR

immediate
timeout

event_loop2.js

```
const fs = require('fs');  
fs.readFile(__filename, () => {  
  setTimeout(() => {  
    console.log('timeout');  
  }, 0);  
  setImmediate(() => {  
    console.log('immediate');  
  });  
});
```

Output (deterministic)

immediate
timeout

event_loop3.js

```
setImmediate(() => {  
    console.log("immediate")  
});  
  
setTimeout(() => {  
    console.log("timeout")  
},0);  
  
process.nextTick(()=>console.log("next"));  
  
console.log("hello1");  
  
console.log("hello2");
```

Output

```
hello1  
hello2  
next  
timeout  
immediate
```

Greeting from Node.js

hello_world.js

```
console.log('Hello from Node.js');
```

Running hello_world.js

Run following command on command line to execute the node.js file hello_world.js:

```
$ node hello_world.js
```

http_server.js

```
const http = require('http'); // http module
const server = http.createServer(function (req, res) {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.write('Hello World!');
  res.end();
});
server.listen(8080);
```

http_server.js (arrow function)

```
const http = require('http');  
const server = http.createServer( (req, res)=> {  
  res.statusCode = 200;  
  res.setHeader('Content-Type', 'text/plain');  
  res.end('Hello World!');  
});  
server.listen(8080);
```

http_server.js (Multiple Web Pages)

```
const http = require('http');
const server = http.createServer( (req, res)=> {
  if (req.url === '/') {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.write('<html><body><p>home Page.</p></body></html>');
    res.end();
  }
  else if (req.url === '/student') { ... }
  else if (req.url === '/admin') { ... }
  else { ... }
});
server.listen(8080);
```

Node.js Module

- Used to **organize** your code in one or more JavaScript files
- Each module has its own **context**
- It **doesn't pollute** global scope
- Uses **CommonJS** modules standard

Node.js Module Types

- 1) Core Modules
- 2) Local/Custom Modules
- 3) Third Party Modules

Node.js Core (Built-in) Modules

Module	Description
assert	Provides a set of assertion tests
buffer	To handle binary data
child_process	To run a child process
cluster	To split a single Node process into multiple processes
crypto	To handle OpenSSL cryptographic functions
dgram	Provides implementation of UDP datagram sockets
dns	To do DNS lookups and name resolution functions
events	To handle events
fs	To handle the file system

Node.js Built-in Modules

Module	Description
http	To make Node.js act as an HTTP server
https	To make Node.js act as an HTTPS server.
net	To create servers and clients
os	Provides information about the operation system
path	To handle file paths
querystring	To handle URL query strings
readline	To handle readable streams one line at the time
stream	To handle streaming data
string_decoder	To decode buffer objects into strings

Node.js Built-in Modules

Module	Description
timers	To execute a function after a given number of milliseconds
tls	To implement TLS and SSL protocols
tty	Provides classes used by a text terminal
url	To parse URL strings
util	To access utility functions
v8	To access information about V8 (the JavaScript engine)
vm	To compile JavaScript code in a virtual machine
zlib	To compress or decompress files

Node.js Local/Custom Module

- Created locally in Node.js application
- Can be packaged and distributed via NPM
- `module` variable represents `current module`
- `exports` is an `object` to be exposed as a module
- Use `module.exports` or `exports` to expose module

Export Literals

msg.js

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

//or

```
exports = 'hello world';
```

app.js

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

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

Export Object (Ex. 1)

msg.js

```
exports.msg = 'hello world';
```



```
{ msg : 'hello world' }  
//object
```

app.js

```
var m = require('./msg.js');  
console.log(m.msg);
```

Export Object (Ex. 2)

log.js

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



```
{ log : function(msg) { ... } }
```

app.js

```
var m = require('./log.js');  
m.log('hello world');
```

Export Object (Ex. 3)

emp.js

```
exports = {  
  fName: 'Jack',  
  lName: 'Martin'  
};
```

app.js

```
var m = require('./emp.js');  
console.log(m.fName);
```


Export Function

log.js

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



function(msg) { ... }

app.js

```
var log = require('./log.js');  
log('hello world');
```

Creating Custom Module

mydatemodule.js

```
exports.myDateTime = function () {  
    return Date();    // new Date().toString()  
};
```

Using Custom Module

custom_module.js

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

Load Module from Folder

custom_module.js

```
var log = require('./utility');
```



Utility folder

./utility/package.json

```
{
```

```
  "name" : "log",
```

```
  "main" : "./log.js"
```

```
}
```



Load module log.js

Built-in URL Module

```
var url = require('url');  
var adr = 'http://localhost:8080/default.htm?year=2020&month=Feb';  
var q = url.parse(adr, true);  
console.log("Host:      " + q.host);    // 'localhost:8080'  
console.log("Pathname:" + q.pathname); // '/default.htm'  
console.log("Search:"   + q.search);   // '?year=2020&month=Feb'  
console.log("Query:"     + q.query);  
                        //returns an object: { year: 2020, month: Feb' }
```

HTTP and URL Modules

```
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;  
    res.end(q.year + " " + q.month);  
}).listen(8080);
```

<http://localhost:8080/?year=2019&month=February>

File System Module

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

Read Files

```
var http = require('http');  
var fs = require('fs');  
http.createServer(function (req, res) {  
    fs.readFile('demofile1.html', function(err, data) {  
        res.writeHead(200, {'Content-Type': 'text/html'});  
        res.write(data);  
        res.end();  
    });  
}).listen(8080);
```


Create Files

- File is opened for writing. If the file does not exist, the file will be created

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

```
fs.open('mynewfile2.txt', 'w', function (err, fd) {  
    if (err) throw err;  
    console.log('File opened for writing!');  
});
```

Flag Description

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

Flag Description

w+	Open for reading and writing. If the file does not exist, it is created. If the file already exists, it is truncated.
wx+	Similar to the w+ flag, but the file is opened in exclusive mode.
a	Open for appending . If the file does not exist, it is created.
ax	Similar to the a flag, but the file is opened in exclusive mode.
a+	Open for reading and appending. If the file does not exist, it is created.
ax+	Similar to the a+ flag, but the file is opened in exclusive mode .

File Read

`fs.read(fd, buffer, offset, length, position, callback)`

- Read data from the file specified by `fd`
- `buffer` is the buffer that the data will be written to
- `offset` is the offset in the buffer to start writing at
- `length` is an integer specifying the number of bytes to read
- `position` is an argument specifying where to begin reading from the file. If position is null or -1, data will be read from the current file position, and the file position will be updated. If position is an integer, the file position will remain unchanged
- The `callback` is given the three arguments, (err, bytesRead, buffer).

File Open and then Read

```
fs.open('input.txt', 'r', function (err, fd) {  
  if (err) {    return console.error(err);  }  
  var buffr = new Buffer(1024);  
  
  fs.read(fd, buffr, 0, buffr.length, 0, function (err, bytes) {  
    if (err) throw err;  
    if (bytes > 0) {  
      console.log(buffr.slice(0, bytes).toString());  
    }  
    fs.close(fd, function (err) {  
      if (err) throw err;  
    });  
  });  
});
```

fs.stats class (to get file information)

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

fs.stats class (to get file information)

```
var fs = require("fs");  
  
fs.stat('input.txt', function (err, stats) {  
    if (err) {    return console.error(err);  }  
  
    console.log(stats);  
  
    console.log("isFile ? " + stats.isFile());  
  
    console.log("isDirectory ? " + stats.isDirectory());  
  
});
```

Synchronous Read

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

```
var data = fs.readFileSync('demofile1.html');
```

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


File Operations in fs Module

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

File Operations in fs Module

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

Delete File

```
var fs = require('fs');  
  
fs.unlink('mynewfile2.txt',  
    function (err) {  
        if (err)  
            throw err;  
    }  
);
```

Rename File

```
var fs = require('fs');  
  
fs.rename('mynewfile1.txt',  
          'myrenamedfile.txt',  
          function (err) {  
            if (err)  
              throw err;  
          })  
);
```

Node.js File Server

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

http.createServer(function (req, res) {

    var q = url.parse(req.url, true);

    var filename = "." + q.pathname;

    fs.readFile(filename, function(err, data) {

        if (err) {

            res.writeHead(404, {'Content-Type': 'text/html'});

            return res.end("404 Not Found");

        }

    })

})
```

Node.js File Server

```
res.writeHead(200, {'Content-Type': 'text/html'});  
res.write(data);  
return res.end();  
});  
}).listen(8080);
```

Built-in Module "Events"

```
var events = require('events');  
  
var EventEmitter = new events.EventEmitter();  
  
var myEventHandler = function () {  
    console.log('Event Handler invoked!');  
}  
  
eventEmitter.on('custom_event', myEventHandler);  
eventEmitter.emit('custom_event');
```

EventEmitter Methods

Method	Description
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.
on (event, listener)	Save as above
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.

EventEmitter Methods

Method	Description
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. If any single listener has been added multiple times to the listener array for the specified event, then removeListener must be called multiple times to remove each instance. Returns emitter, so calls can be chained.
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.
listeners(event)	Returns an array of listeners for the specified event.

EventEmitter Methods

Method	Description
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.
emit(event, [arg1], [arg2], [...])	Execute each of the listeners in order with the supplied arguments. Returns true if the event had listeners, false otherwise.

EventEmitter Class Method

Method	Description
listenerCount(emitter, event)	Returns the number of listeners for a given event.

Upload files (Formidable Module)

Install "formidable" module using npm:

```
npm install formidable
```

Upload files (file_upload.js)

```
const express      = require('express');
const formidable    = require('formidable');
const fs           = require('fs')
const path         = require('path')
const app          = express();

app.get('/', (req, res) => {
  res.sendFile(__dirname + '/upload_file.html')
});
```

Upload files (file_upload.js)

```
app.post('/api/upload', (req, res, next) => {  
  const form = new formidable.IncomingForm();  
  
  form.parse(req, function (err, fields, files) {  
  
    var oldPath = files.student_data.path;  
    var newPath = path.join(__dirname, 'uploads')  
                  + '/' + files.student_data.name  
    var rawData = fs.readFileSync(oldPath)  
  
    fs.writeFile(newPath, rawData, function (err) {  
      if (err) console.log(err)  
      return res.send("Successfully uploaded")  
    })  
  })  
});
```

Upload files (upload_file.html)

```
<form    action="/api/upload"
        enctype="multipart/form-data"
        method="post">
  <div>Text field title:
    <input type="text" name="title" />
  </div>
  <div>File:
    <input  type="file"
            name="student_data"
            multiple="multiple"
          />
  </div>
  <input type="submit" value="Upload" />
</form>
```

Send emails (Nodemailer Module)

Install "nodemailer" module using npm:

```
npm install -g nodemailer
```


Send emails (Nodemailer Module)

If you are using **gmail** then enable "less secure app" to send email:


<https://myaccount.google.com/lesssecureapps>

Send emails (Nodemailer Module)

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

Send emails (Nodemailer Module)

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



Email sent: 250 2.0.0 OK 5368739 b64-vm426195p.7 - gsmtptp

Buffers

- Pure JavaScript is **Unicode** friendly, but it is not so for **binary data**
- While dealing with **TCP streams** or the **file system**, it's necessary to handle **octet streams**
- Buffer class provides instances to store **raw data** similar to an array of integers
- Data is allocated as a raw memory **outside the V8 heap**

Creating Buffers

- `var buf = new Buffer(10);`
- `var buf = new Buffer([10, 20, 30, 40, 50]);`
- `var buf = new Buffer("Learn Node.js", "utf-8");`

Possible values of encoding are : "ascii", "utf8", "utf16le", "ucs2", "base64" or "hex"

Writing to Buffers

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

- string – string data to be written to buffer
- offset – index of the buffer to start writing at (Default – 0)
- length – number of bytes to write (Default - buffer.length)
- encoding – Encoding to use. 'utf8' is the default encoding.

Reading from Buffers

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

```
buf = new Buffer(26);
```

```
for (var i = 0 ; i < 26 ; i++) {  buf[i] = i + 97;  }
```

```
console.log( buf.toString('ascii'));           // a..z
```

```
console.log( buf.toString('ascii',0,5));       // abcde
```

```
console.log( buf.toString('utf8',0,5));        // abcde
```

```
console.log( buf.toString(undefined,0,5));     // abcde
```

Unicode Encoding

- UTF-8 and UTF-16 are **variable length encodings**
- UTF-8: minimum 8 bits
- UTF-16: minimum 16 bits
- UTF-32: Fixed 32 bits

Convert buffer to JSON

`buf.toJSON()`

```
var buf = new Buffer('abc');
```

```
var json = buf.toJSON(buf);
```

```
console.log(json); // [97, 98, 99]
```

Buffer Class Methods

Sr. No.	Method & Description
1	Buffer. isEncoding (encoding) Returns true if the encoding is a valid encoding argument, false otherwise.
2	Buffer. isBuffer (obj) Tests if obj is a Buffer.
3	Buffer. byteLength (string[, encoding]) Gives the actual byte length of a string. encoding defaults to 'utf8'. It is not the same as String.prototype.length, since String.prototype.length returns the number of characters in a string.
4	Buffer. concat (list[, totalLength]) Returns a buffer which is the result of concatenating all the buffers in the list.
5	Buffer. compare (buf1, buf2) The same as buf1.compare(buf2). Useful for sorting an array of buffers.

Streams

- Streams are objects that let you **read** data from a source or **write** data to a destination in **continuous fashion**
-
1. **Readable** – Stream which is used for **read** operation.
 2. **Writable** – Stream which is used for **write** operation.
 3. **Duplex** – Stream which can be used for **both read and write** operation.
 4. **Transform** – A type of duplex stream where the output is computed based on input.

Stream as an Instance of EventEmitter

- Each type of **Stream** is an **EventEmitter** instance and throws several **events** at different instance of times
- **data** – when there is data is available to read
- **end** – when there is no more data to read
- **error** – when there is any error receiving or writing data
- **finish** – when all the data has been flushed to underlying system

Reading from a Stream

```
var fs = require("fs");  
  
var data = "";  
  
var rs = fs.createReadStream('input.txt');  
  
rs.on('data', function(chunk) {  
    data += chunk;  
  
});  
  
rs.on('end', function() { console.log(data); });  
  
rs.on('error', function(err){ console.log(err.stack);});
```

Writing to a Stream

```
var fs = require("fs");  
  
var data = 'Learning Node.js';  
  
var ws = fs.createWriteStream('output.txt');  
  
ws.write(data, 'UTF8');  
  
ws.end();  
  
ws.on('finish', function() {console.log("Write completed.");    });  
ws.on('error', function(err){ console.log(err.stack);    });
```

Piping the Streams

- Piping is a mechanism where we provide the output of one stream as the input to another stream
- It is normally used to get data from one stream and to pass the output of that stream to another stream
- There is no limit on piping operations

Piping the Streams (Ex.)

```
var fs = require("fs");  
var readerStream = fs.createReadStream('input.txt');  
var writerStream = fs.createWriteStream('output.txt');  
readerStream.pipe(writerStream);
```


Chaining the Streams (Ex.)

```
var fs      = require("fs");  
var zlib    = require('zlib');  
fs.createReadStream('input.txt')  
    .pipe(zlib.createGzip())  
    .pipe(fs.createWriteStream('input.txt.gz'));  
fs.createReadStream('input.txt.gz')  
    .pipe(zlib.createGunzip())  
    .pipe(fs.createWriteStream('input_decompressed.txt'));
```

Global Objects

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

Global Objects

- `__filename` - filename of the code
- `__dirname` - name of the current directory of running script
- `setTimeout (cb, ms)` - run callback function `cb` after at least `ms` milliseconds
- `clearTimeout(t)` - stop a timer that was previously created with `setTimeout()`
- `setInterval (cb, ms)` - run callback function `cb` repeatedly after at least `ms` milliseconds

Global Objects (Console)

```
console.info("Program Started");  
var i, j;  
console.time("Getting data");  
for(i=0; i<100; i++)  
    for(j=0; j < 100; j++)  
        console.log('running');  
console.timeEnd('Getting data');  
console.info("Program Ended")
```

Global Objects (Process : Events)

Sr. No.	Event & Description
1	exit Emitted when the process is about to exit. There is no way to prevent the exiting of the event loop at this point, and once all exit listeners have finished running, the process will exit.
2	beforeExit This event is emitted when node empties its event loop and has nothing else to schedule. Normally, the node exits when there is no work scheduled, but a listener for 'beforeExit' can make async calls, and cause the node to continue.
3	uncaughtException Emitted when an exception bubbles all the way back to the event loop. If a listener is added for this exception, the default action will not occur.
4	Signal Events Emitted when the processes receives a signal such as SIGINT, SIGHUP, etc.

Global Objects (Process : Properties)

stdout	stdin	stderr	argv
execPath	execArgv	env	exitCode
version	versions	config	Pid
title	arch	platform	mainModule

Global Objects (Process : Methods)

<code>abort()</code>	<code>chdir(dir)</code>	<code>cwd()</code>	<code>exit([code])</code>
<code>getgid()</code>	<code>setgid(id)</code>	<code>getuid()</code>	<code>setuid(id)</code>
<code>getgroups()</code>	<code>setgroups(groups)</code>	<code>kill(pid,sig)</code>	<code>memoryUsage()</code>
<code>nextTick(cb)</code>	<code>initgroups(user, extra_group)</code>	<code>umask(mask)</code>	<code>uptime()</code>
<code>hrtime()</code>			

References

1. <https://nodejs.org/en/docs/guides/>
2. <https://www.w3schools.com/nodejs/>
3. <https://www.tutorialspoint.com/nodejs/>
4. <https://www.tutorialsteacher.com/nodejs>