NodeJS in Details

Unit#5

Marwadi University

Department of Information Technology

Advanced Web Programming (3161611)

Tejas Chauhan

Highlights

- Events and Event Loop, Timers, Error Handling, Buffers, Streams, Work with File System, Networking with Node (TCP, UDP and HTTP clients and servers), Web Module, Debugging, Node JS REST API, Sessions and Cookies, Design patterns, Caching, Scalability.

Modules

- Before we deep dive in NodeJS, we should understand modules of NodeJS first.
- Node.js includes three types of modules:
  - Core Modules
  - Local Modules
  - Third Party Modules

Core Modules

- Core modules are compiled into its binary distribution and load automatically when Node.js process starts.
- However, you need to import the core module first in order to use it in your application.
- The following table lists some of the important core modules in Node.js.

Core Module	Description
os	os module provides operating system-related utility methods and properties.
fs	fs module includes classes, methods, and events to work with file I/O.
path	path module provides utilities for working with file and directory paths.
http	http module includes classes, methods and events to create Node.js http server.
url	url module provides utilities for URL resolution and parsing.

Loading Modules

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

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

- Example:

```
const os = require('os');

console.log(os.cpus());
```


Local Module

- Local modules are modules created locally in your Node.js application.
- These modules include different functionalities of your application in separate files and folders.
- You can also package it and distribute it via NPM, so that Node.js community can use it.
- For example, if you need to connect to MongoDB and fetch data then you can create a module for it, which can be reused in your application.

Local Module

- Let's write a simple logging module which logs the information, warning or error to the console.
- In Node.js, module should be placed in a separate JavaScript file.
- So, let’s create a log.js file for log module and app.js file for our application.

Local Module

- log.js:

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

Local Module

- app.js:

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

myLogModule.info('This is info
message');
myLogModule.warning('This is warning
message');
myLogModule.error('This is error
message');
```


Third Party Modules

- Third Party Modules are such modules which are developed by other developers and it is outside of your application. (e.g.: express, pdfkit, qrcode, lightgallery…)
- You make these modules available to your application using npm and install it.
- Installing to app only or globally.
- Define dependency in package.json file and then import it using require() function.
- once you have imported a module, you can use its functionality like other modules.

Events

- Much of the Node.js core API is built around an idiomatic asynchronous event-driven architecture in which certain kinds of objects (called "emitters") emit named events that cause Function objects ("listeners") to be called.
- For instance:
  - a *net.Server* object emits an event each time a peer connects to it;
  - a *fs.ReadStream* emits an event when the file is opened;
  - a *stream* emits an event whenever data is available to be read.

Events

- All objects that emit events are instances of the EventEmitter class.
- These objects expose an eventEmitter.on() function that allows one or more functions to be attached to named events emitted by the object.
- Typically, event names are camel-cased strings but any valid JavaScript property key can be used.
- When the EventEmitter object emits an event, all of the functions attached to that specific event are called synchronously.
- Any values returned by the called listeners are ignored and discarded.

Events

- The following example shows a simple EventEmitter instance with a single listener.
- The eventEmitter.on() method is used to register listeners, while the eventEmitter.emit() method is used to trigger the event.

```
const EventEmitter = require('events');
class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter();

myEmitter.on('event', () => {
  console.log('an event occurred!');
});
myEmitter.emit('event');
```


Events

- The eventEmitter.emit() method allows an arbitrary set of arguments to be passed to the listener functions.
- When an ordinary listener function is called, the standard *this* keyword is intentionally set to reference the EventEmitter instance to which the listener is attached.

```
const myEmitter = new MyEmitter();
myEmitter.on('event', (a, b) => {
  console.log(a, b, this);
  // Prints: a b {}
});
myEmitter.emit('event', 'a', 'b');
```


Events

- When a listener is registered using the eventEmitter.on() method, that listener is invoked every time the named event is emitted.
- Using the eventEmitter.once() method, it is possible to register a listener that is called at most once for a particular event.
- Once the event is emitted, the listener is unregistered and then called.

Events and Error Handling

- When an error occurs within an EventEmitter instance, the typical action is for an 'error' event to be emitted.
- If an EventEmitter does not have at least one listener registered for the 'error' event, and an 'error' event is emitted, the error is thrown, a stack trace is printed, and the Node.js process exits.
- As a best practice, listeners should always be added for the 'error' events.

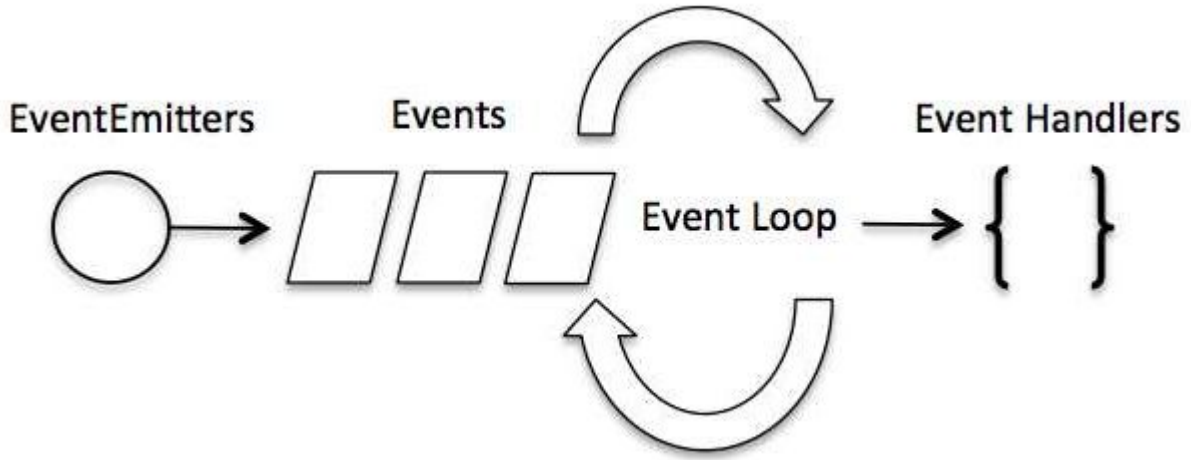
```
const myEmitter = new MyEmitter();
myEmitter.on('error', (err) => {
  console.error('whoops! there was an error');
});
myEmitter.emit('error', new Error('whoops!'));
```


Events

- Class: EventEmitter
  - Event: 'newListener'
  - Event: 'removeListener'
  - emitter.addListener(eventName, listener)
  - emitter.emit(eventName[, ...args])
  - emitter.eventNames()
  - emitter.listenerCount(eventName)
  - emitter.listeners(eventName)
  - emitter.off(eventName, listener)
  - emitter.on(eventName, listener)
  - emitter.once(eventName, listener)
  - emitter.removeAllListeners([eventName])
  - emitter.removeListener(eventName, listener)
  - .....

Event Loop

- In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function when one of those events is detected.



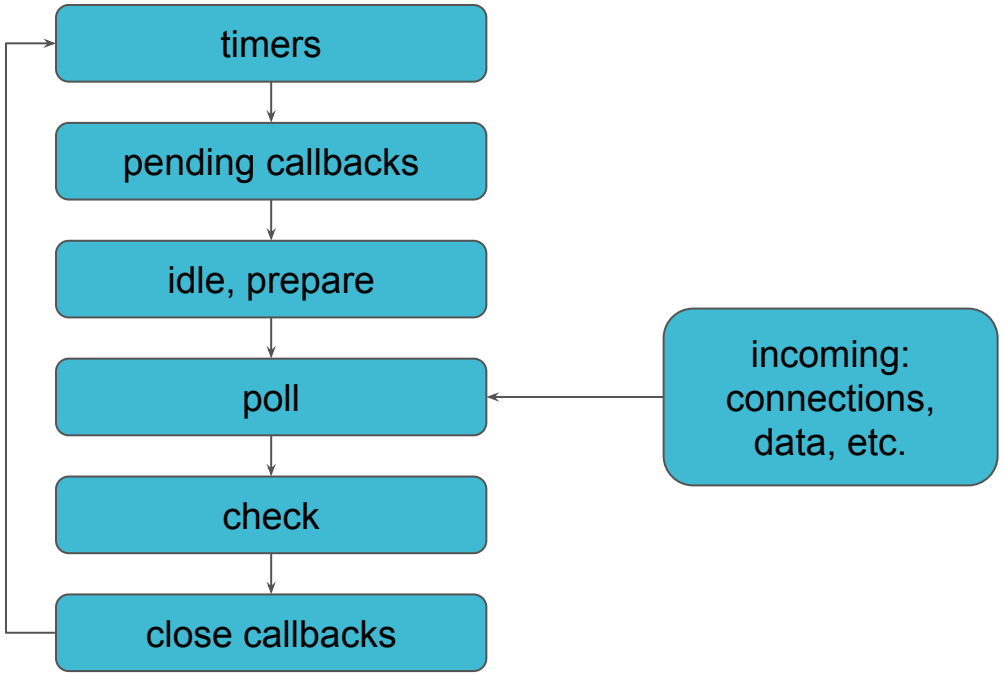
Source: https://www.tutorialspoint.com/nodejs/nodejs_event_loop.htm

Event Loop

- The event loop is what allows Node.js to perform non-blocking I/O operations — despite the fact that JavaScript is single-threaded — by offloading operations to the system kernel whenever possible.
- Since most modern kernels are multi-threaded, they can handle multiple operations executing in the background.
- When one of these operations completes, the kernel tells Node.js so that the appropriate callback may be added to the poll queue to eventually be executed.
- When Node.js starts, it initializes the event loop, processes the provided input script (or drops into the REPL) which may make async API calls, schedule timers, or call process.nextTick(), then begins processing the event loop.

Event Loop

- The following diagram shows a simplified overview of the event loop's order of operations.
- Each box will be referred to as a "phase" of the event loop.



Event Loop

- Each phase has a FIFO queue of callbacks to execute.
- When the event loop enters a given phase, it will perform any operations specific to that phase, then execute callbacks in that phase's queue until the queue has been exhausted or the maximum number of callbacks has executed.
- When the queue has been exhausted or the callback limit is reached, the event loop will move to the next phase, and so on.

Event Loop

- Phases Overview:
  - *timers*: this phase 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; node will block here when appropriate.
  - *check*: setImmediate() callbacks are invoked here.
  - *close callbacks*: some close callbacks, e.g. socket.on('close', ...).
- Between each run of the event loop, Node.js checks if it is waiting for any asynchronous I/O or timers and shuts down cleanly if there are not any.

Timers

- A timer specifies the threshold after which a provided callback may be executed rather than the exact time a person wants it to be executed.
- Timers callbacks will run as early as they can be scheduled after the specified amount of time has passed; however, Operating System scheduling or the running of other callbacks may delay them.
- setTimeout(), setInterval(), setImmediate().

setImmediate() vs setTimeout()

- setImmediate() and setTimeout() are similar, but behave in different ways depending on when they are called.
  - setImmediate() is designed to execute a script once the current poll phase completes.
  - setTimeout() schedules a script to be run after a minimum threshold in ms has elapsed.
- The order in which the timers are executed will vary depending on the context in which they are called.
- If both are called from within the main module, then timing will be bound by the performance of the process.

setImmediate() vs setTimeout()

- If we run the following script which is not within an I/O cycle (i.e. the main module), the order in which the two timers are executed is non-deterministic, as it is bound by the performance of the process:

```
setTimeout(() => {
  console.log('timeout');
}, 0);

setImmediate(() => {
  console.log('immediate');
});
```


setImmediate() vs setTimeout()

- However, if you move the two calls within an I/O cycle, the immediate callback is always executed first:

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


setImmediate() vs setTimeout()

- The main advantage to using setImmediate() over setTimeout() is setImmediate() will always be executed before any timers if scheduled within an I/O cycle, independently of how many timers are present.

Error Handling

- Applications running in Node.js will generally experience four categories of errors:
  - *Standard JavaScript errors* such as <EvalError>, <SyntaxError>, <RangeError>, <ReferenceError>, <TypeError>, and <URIError>.
  - *System errors* triggered by underlying operating system constraints such as attempting to open a file that does not exist or attempting to send data over a closed socket.
  - *User-specified errors* triggered by application code.
  - *AssertionErrors* are a special class of error that can be triggered when Node.js detects an exceptional logic violation that should never occur. These are raised typically by the assert module.

Error Handling

- All JavaScript and system errors raised by Node.js inherit from, or are instances of, the standard JavaScript <Error> class and are guaranteed to provide at least the properties available on that class.
- Node.js supports several mechanisms for propagating and handling errors that occur while an application is running.
- How these errors are reported and handled depends entirely on the type of Error and the style of the API that is called.

Error Handling

- All JavaScript errors are handled as exceptions that immediately generate and throw an error using the standard JavaScript throw mechanism.
- These are handled using the *try…catch* construct provided by the JavaScript language.

```
// Throws with a ReferenceError because z is not
defined.
try {
  const m = 1;
  const n = m + z;
} catch (err) {
  // Handle the error here.
}
```


Error Handling

- Errors that occur within Asynchronous APIs may be reported in multiple ways:
  - Most asynchronous methods that accept a callback function will accept an Error object passed as the first argument to that function.
  - If that first argument is not null and is an instance of Error, then an error occurred that should be handled.

```
const fs = require('fs');
fs.readFile('a file that does not exist', (err, data) => {
  if (err) {
    console.error('There was an error reading the file!', err);
    return;
  }
  // Otherwise handle the data
});
```


Error Handling

- Errors that occur within Asynchronous APIs may be reported in multiple ways:
  - When an asynchronous method is called on an object that is an EventEmitter, errors can be routed to that object's 'error' event.

```
const net = require('net');
const connection = net.connect('localhost');
// Adding an 'error' event handler to a stream:
connection.on('error', (err) => {
  // If the connection is reset by the server, or if it can't
  // connect at all, or on any sort of error encountered by
  // the connection, the error will be sent here.
  console.error(err);
});
connection.pipe(process.stdout);
```


Buffers

- JavaScript has historically had subpar binary support.
- Typically, parsing binary data would involve various tricks with strings to extract the data you want.
- Not having a good mechanism to work with raw memory in JavaScript was one of the problems Node core developers had to tackle when the project started getting traction.
- This was mostly for performance reasons.
- All of the raw memory accumulated in the Buffer data type.
- Buffers are raw allocations of the heap, exposed to JavaScript in an array-like manner.

Buffers

- Buffer objects are used to represent a fixed-length sequence of bytes.
- Many Node.js APIs support Buffers.
- The Buffer class is a subclass of JavaScript's Uint8Array class and extends it with methods that cover additional use cases.
- Node.js APIs accept plain Uint8Arrays wherever Buffers are supported as well.
- The Buffer class is within the global scope, making it unlikely that one would need to ever use require('buffer').Buffer.

Buffers

```
// Creates a zero-filled Buffer of length 10.
const buf1 = Buffer.alloc(10);

// Creates a Buffer of length 10,
// filled with bytes which all have the value `1`.
const buf2 = Buffer.alloc(10, 1);

// Creates a Buffer containing the bytes [1, 2, 3].
const buf3 = Buffer.from([1, 2, 3]);

// Creates a Buffer containing the bytes [1, 1, 1, 1]
// the entries are all truncated using `(value & 255)`
// to fit into the range 0-255.
const buf4 = Buffer.from([257, 257.5, -255, '1']);
```


Buffers

```
// Creates a Buffer containing the UTF-8-encoded bytes
// for the string 'tést':
// [0x74, 0xc3, 0xa9, 0x73, 0x74](in hexadecimal notation)
// [116, 195, 169, 115, 116] (in decimal notation)
const buf5 = Buffer.from('tést');

// Creates a Buffer containing the Latin-1 bytes
// [0x74, 0xe9, 0x73, 0x74].
const buf6 = Buffer.from('tést', 'latin1');
```


Buffers and character encodings

- When converting between Buffers and strings, a character encoding may be specified.
- If no character encoding is specified, UTF-8 will be used as the default.

```
const buf = Buffer.from('hello world', 'utf8');

console.log(buf.toString('hex'));
// Prints: 68656c6c6f20776f726c64
console.log(buf.toString('base64'));
// Prints: aGVsbG8gd29ybGQ=

console.log(Buffer.from('fhqwhgads', 'utf8'));
// Prints: <Buffer 66 68 71 77 68 67 61 64 73>
console.log(Buffer.from('fhqwhgads', 'utf16le'));
// Prints: <Buffer 66 00 68 00 71 00 77 00 68 00 67 00
61 00 64 00 73 00>
```


Buffers and character encodings

- Example: Working with data URIs, Using the Buffer API can be helpful.
- Data URIs allow a resource to be embedded inline on a web page using the following scheme:

```
data:[MIME-type][;charset=<encoding>][;base64],<data>
```

- For example, a PNG image can be represented as a data URI:

```
...
```

```
var fs = require('fs');
var mime = 'image/png';
var encoding = 'base64';
var data =
fs.readFileSync('./monkey.png').toString(encoding);
var uri = 'data:' + mime + ';' + encoding + ',' + data;
console.log(uri);
```


Streams

- A stream is an abstract interface for working with streaming data in Node.js.
- The stream module provides an API for implementing the stream interface.
- There are many stream objects provided by Node.js.
- For instance, a request to an HTTP server and process.stdout are both stream instances.
- Streams can be readable, writable, or both.
- All streams are instances of EventEmitter.

Streams

- To access the stream module:

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

- The stream module is useful for creating new types of stream instances.
- It is usually not necessary to use the stream module to consume streams.

Types of streams

- There are four fundamental stream types within Node.js:
  - Writable: streams to which data can be written (for example, fs.createWriteStream()).
  - Readable: streams from which data can be read (for example, fs.createReadStream()).
  - Duplex: streams that are both Readable and Writable (for example, net.Socket).
  - Transform: Duplex streams that can modify or transform the data as it is written and read (for example, zlib.createDeflate()).

Types of streams

- Streams always involve I/O of some kind, and they can be classified into groups based on the type of I/O they deal with:
  - Built-in: Many of Node's core modules implement streaming interfaces; for example, fs.createReadStream.
  - HTTP: Although technically network streams, there are streaming modules designed to work with various web technologies.
  - Parsers: Historically parsers have been implemented using streams. Popular third-party modules for Node include XML and JSON parsers.
  - Browser: Node's event-based streams have been extended to work in browsers, offering some unique opportunities for interfacing with client-side code.

Types of streams

- Streams always involve I/O of some kind, and they can be classified into groups based on the type of I/O they deal with:
  - Audio: James Halliday has written some novel audio modules that have streamable interfaces.
  - RPC (Remote Procedure Call): Sending streams over the network is a useful way to implement interprocess communication.
  - Test: There are stream-friendly test libraries, and tools for testing streams themselves.
  - Control, meta, and state: There are also more abstract uses of streams, and modules designed purely for manipulating and managing other streams.

When to use streams

- When reading a file synchronously with fs.readFileSync, the program will block, and all of the data will be read to memory.
- Using fs.readFile will prevent the program from blocking because it's an asynchronous method, but it'll still read the entire file into memory.
- What if there were a way to tell fs.readFile to read a chunk of data into memory, process it, and then ask for more data? That's where streams come in.

When to use streams

- Memory becomes an issue when working with large files - compressed backup archives, media files, large log files, and so on.
- Instead of reading the entire file into memory, you could use fs.read with a suitable buffer, reading in a specific length at a time.
- Or, preferably, you could use the streams API provided by fs.createReadStream.
- Streams are asynchronous by design.
- Rather than reading that entire file into memory, a buffer's worth will be read, the desired operations will be performed, and then the result will be written to the output stream.

Streams

- A simple static web server that uses streams:

```
var http = require('http');
var fs = require('fs');
http.createServer(function(req, res) {
   fs.createReadStream(__dirname +
'/index.html').pipe(res);
}).listen(8000);
```


Streams

- Catching errors during streaming

```
var fs = require('fs');
var stream =
fs.createReadStream('not-found');
stream.on('error', (err) => {
   console.trace();
   console.error('Stack:', err.stack);
   console.error('The error raised
was:', err);
});
```


File System

- The fs module enables interacting with the file system in a way modeled on standard POSIX functions.
- To make use of file system functionality you need to require a file system module.

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


File System

- *fs.readFile(path[, options], callback);*
- Asynchronously reads the entire contents of a file.
- The callback is passed two arguments (err, data), where data is the contents of the file.
- If no encoding is specified, then the raw buffer is returned.

```
fs.readFile('/etc/passwd', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```


File System

- *fs.writeFile(file, data[, options], callback);*
- When file is a filename, asynchronously writes data to the file, replacing the file if it already exists. data can be a string or a buffer.
- When file is a file descriptor, the behavior is similar to calling fs.write() directly.
- The encoding option is ignored if data is a buffer.
- If data is a normal object, it must have an own toString function property.

```
fs.writeFile('message.txt', data, (err) => {
  if (err) throw err;
  console.log('The file has been saved!');
});
```


File System

- *fs.appendFile(path, data[, options], callback);*
- Asynchronously append data to a file, creating the file if it does not yet exist. data can be a string or a <Buffer>.

```
fs.appendFile('message.txt', 'data to
append', 'utf8', (err) => {
  if (err) throw err;
  console.log('The "data to append" was
appended to file!');
});
```


File System

- *fs.read(fd, buffer, offset, length, position, callback);*
- *fs.read(fd, [options,] callback);*
- Read data from the file specified by fd.
- The callback is given the three arguments, (err, bytesRead, buffer).

File System

- *fs.write(fd, buffer[, offset[, length[, position]]], callback);*
- Write buffer to the file specified by fd.
- offset determines the part of the buffer to be written, and length is an integer specifying the number of bytes to write.
- position refers to the offset from the beginning of the file where this data should be written.
- If typeof position !== 'number', the data will be written at the current position.
- The callback will be given three arguments (err, bytesWritten, buffer) where bytesWritten specifies how many bytes were written from buffer.

File System

- *fs.write(fd, string[, position[, encoding]], callback);*
- Write string to the file specified by fd.
- position refers to the offset from the beginning of the file where this data should be written.
- If typeof position !== 'number' the data will be written at the current position.
- encoding is the expected string encoding.
- The callback will receive the arguments (err, written, string) where written specifies how many bytes the passed string required to be written.

File System

- *fs.createReadStream(path[, options]);*
- options can include start and end values to read a range of bytes from the file instead of the entire file.
- Both start and end are inclusive and start counting at 0, allowed values are in the range.
- If fd is specified and start is omitted or undefined, fs.createReadStream() reads sequentially from the current file position.
- If fd is specified, ReadStream will ignore the path argument and will use the specified file descriptor.
- By default, the stream will emit a 'close' event after it has been destroyed, like most Readable streams. Set the emitClose option to false to change this behavior.

File System

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

var readerStream=fs.createReadStream('input.txt');
readerStream.setEncoding('UTF8');

readerStream.on('data',(chunk)=> {
   data += chunk;
});
readerStream.on('end',()=> {
   console.log(data);
});
readerStream.on('error',(err)=> {
   console.log(err.stack);
});
```


File System

- *fs.createWriteStream(path[, options]);*
- options may also include a start option to allow writing data at some position past the beginning of the file, allowed values are in the range.
- Modifying a file rather than replacing it may require the flags option to be set to r+ rather than the default w.
- If autoClose is set to true (default behavior) on 'error' or 'finish' the file descriptor will be closed automatically.
- If autoClose is false, then the file descriptor won't be closed, even if there's an error.
- Like <fs.ReadStream>, if fd is specified, <fs.WriteStream> will ignore the path argument and will use the specified file descriptor.

File System

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

var writerStream=fs.createWriteStream('output.txt');
writerStream.write(data,'UTF8');

writerStream.end();

writerStream.on('finish',() => {
    console.log("Write completed.");
});
writerStream.on('error',(err) => {
    console.log(err.stack);
});
```


File System

- *fs.mkdir(path[, options], callback);*
- Asynchronously creates a directory.
- The callback is given a possible exception and, if recursive is true, the first directory path created, (err, [path]).
- path can still be undefined when recursive is true, if no directory was created.
- The optional options argument can be an integer specifying mode (permission and sticky bits), or an object with a mode property and a recursive property indicating whether parent directories should be created.
- Calling fs.mkdir() when path is a directory that exists results in an error only when recursive is false.

File System

- *fs.readdir(path[, options], callback);*
- Reads the contents of a directory.
- The callback gets two arguments (err, files) where files is an array of the names of the files in the directory excluding '.' and '..'.
- The optional options argument can be a string specifying an encoding, or an object with an encoding property specifying the character encoding to use for the filenames passed to the callback.
- If the encoding is set to 'buffer', the filenames returned will be passed as <Buffer> objects.

File System

- *fs.chmod(path, mode, callback);*
- Asynchronously changes the permissions of a file. No arguments other than a possible exception are given to the completion callback.
- File modes: fs.constants.S_IRUSR(0o400:read by owner), fs.constants.S_IWUSR(0o200:write by owner), fs.constants.S_IXUSR(0o100:execute/search by owner)...

```
chmod('my_file.txt', 0o775, (err) => {
  if (err) throw err;
  console.log('The permissions for file
"my_file.txt" have been changed!');
});
```


File System

- fs.open(path[, flags[, mode]], callback)
- fs.close(fd[, callback])
- fs.opendir(path[, options], callback)
- fs.copyFile(src, dest[, mode], callback)
- fs.rename(oldPath, newPath, callback)
- fs.rmdir(path[, options], callback)
- fs.rm(path[, options], callback)
- fs.unlink(path, callback)
- …

Networking with Node

- The Node.js platform itself is billed as a solution for writing fast and scalable network applications.
- To write network-oriented software, you need to understand how networking technologies and protocols interrelate.
- Node has a suite of networking modules that allows you to build web and other server applications.
- Node's networking modules includes the dgram, dns, http, and net modules.

Networking concepts

Term	Description
Layer	A slice of related networking protocols that represents a logical group. The application layer, where we work, is the highest level; physical is the lowest.
HTTP	Hypertext Transfer Protocol - An application-layer client-server protocol built on TCP.
TCP	Transmission Control Protocol - Allows communication in both directions from the client to the server, and is built on to create application-layer protocols like HTTP.
UDP	User Datagram Protocol—A lightweight protocol, typically chosen where speed is desired over reliability.
Socket	The combination of an IP address and a port number is generally referred to as a socket.
Packet	TCP packets are also known as segments—the combination of a chunk of data along with a header.
Datagram	The UDP equivalent of a packet.
MTU	Maximum Transmission Unit—The largest size of a protocol data unit. Each layer can have an MTU: IPv4 is at least 68 bytes, and Ethernet v2 is 1,500 bytes.

TCP clients and servers

- Node has a simple API for creating TCP connections and servers.
- Most of the lowest level classes and methods can be found in the net module.

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


Creating a TCP server and tracking clients

```
var net = require('net');
var clients = 0;
var server = net.createServer((client)=> {
   clients++;
   var clientId = clients;
   console.log('Client connected:', clientId);
   client.on('end',() => {
      console.log('Client disconnected:',clientId);
   });
   client.write('Welcome client: '+clientId);
   client.pipe(client);
});
server.listen(8000,() => {
   console.log('Server started on port 8000');
});
```


Testing TCP servers with clients

```
server.listen(8000, () => {
  console.log('Server started on port 8000');
  runTest(1, () => {
    runTest(2, () => {
      console.log('Tests finished');
      server.close();
    });
  });
});
function runTest(expectedId, done) {
  var client = net.connect(8000);
  client.on('data', (data) => {
    console.log(expectedId+":"+data.toString('utf8'));
    client.end();
  });
  client.on('end', done);
}
```


UDP clients and servers

- Compared to TCP, UDP is a much simpler protocol.
- That can mean more work for you: rather than being able to rely on data being sent and received, you have to cater to UDP's more volatile nature.
- UDP is suitable for query-response protocols, which is why it's used for the Domain Name System (DNS).
- It's also stateless—if you want to transfer data and you value lower latency over data integrity, then UDP is a good choice.
- Media streaming protocols and online games generally use UDP.

UDP clients and servers

```
var dgram = require("dgram");
var fs = require("fs");
var port = 41230;
var defaultSize = 16;
function Client(remoteIP) {
  var inStream = fs.createReadStream('input.txt');
  var socket = dgram.createSocket("udp4");
  inStream.on("readable", () => sendData());
  function sendData() {
    var message = inStream.read(defaultSize);
    if (!message) {
      return socket.unref();
    }
    socket.send(message, 0, message.length, port,
remoteIP, (err, bytes) => sendData());
  }
}
```


UDP clients and servers

```
function Server() {
  var socket = dgram.createSocket("udp4");
  socket.on("message", (msg, rinfo) => {
    process.stdout.write(msg.toString());
  });
  socket.on("listening", () => {
    console.log("Server ready:", socket.address());
  });
  socket.bind(port);
}

if (process.argv[2] === "client") {
  new Client(process.argv[3]);
} else {
  new Server();
}
```


HTTP clients and servers

- Today most of us work with HTTP - whether we're producing or consuming web services, or building web applications.
- The HTTP protocol is stateless and built on TCP, and Node's HTTP module is similarly built on top of its TCP module.

HTTP server and client

```
var http = require('http');
var server = http.createServer((req, res) => {
  res.writeHead(200,{'Content-Type':'text/plain' });
  res.write('Hello, world.\r\n');
  res.end();
});
server.listen(8000, () => {
  console.log('Listening on port 8000');
});

var req = http.request({port: 8000},(res) => {
  console.log('HTTP headers:', res.headers);
  res.on('data', (data) => {
    console.log('Body:', data.toString());
    server.unref();
  });
});
req.end();
```


Express Module

- Web Applications:
- Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.
- APIs:
- With a myriad of HTTP utility methods and middleware at your disposal, creating a robust API is quick and easy.
- To install:

```
$ npm install express --save
```


Express Example

```
const path = require('path');
const express = require('express');
const app = express();
app.use(express.static('./assets'));

app.get('/index', (req, res) => {
    res.sendFile(path.join(__dirname, '/index.html'));
})
app.get('/about', (req, res) => {
    res.sendFile(path.join(__dirname, '/about.html'));
})
app.get('/contact', (req, res) => {
    res.sendFile(path.join(__dirname, '/contact.html'));
})

app.listen(8000, () => {
    console.log("Listening to port 8000.");
});
```


Node JS REST API

- An API is always needed to create mobile applications, single page applications, use AJAX calls and provide data to clients.
- An popular architectural style of how to structure and name these APIs and the endpoints is called REST(Representational Transfer State).

Node JS REST API

Method	URI	Details	Function
GET	/api/contacts	Safe, cachable	Gets the list of all contacts and their details
GET	/api/contacts/10	Safe, cachable	Gets the details of contact id 10
POST	/api/contacts	N/A	Creates a new contact with the details provided. Response contains the URI for this newly created resource or its id.
PUT	/api/contacts/10	Idempotent	Modifies contact id 10(creates one if it doesn't already exist). Response contains the URI for this newly created resource or its id.
DELETE	/api/contacts/10	Idempotent	Contact id 10 should be deleted, if it exists. Response should contain the status of the request.
DELETE or PUT	/api/contacts	Invalid	Should be invalid. DELETE and PUT should specify which resource they are working on.

Node JS REST API Example

```
const express = require("express");
const app = express();
const port = 8000;
app.use(express.json());
// contacts array of json objects for contact details
let contacts = [{...}];
// Retrieve all contacts
app.get("/api/contacts", (req, res) => {
  res.send(contacts);
});
// Retrieve single person details
app.get("/api/contacts/:id", (req, res) => {
  let person = contacts.find((per) => per.id ===
parseInt(req.params.id));
  if (!person) {
    res.status(404).send("Person not found...");
    return;
  }
  res.send(person);
});
```


Node JS REST API Example

```
// Add new person
app.post("/api/contacts", (req, res) => {
  const person = {id: contacts.reduce((pre,curr) => {
      return (pre.id < curr.id) ? curr : pre; }).id +1,
    photo: req.body.photo,
    title: req.body.title,
    …
    address: {
      line1: req.body.address.line1,
      …
      zip: req.body.address.zip,
      country: req.body.address.country
    },
    note: req.body.note
  };

  contacts.push(person);
  res.send("Person added...");
});
```


Node JS REST API Example

```
// Edit/Update person
app.put("/api/contacts/:id", (req,res) => {
  let person = contacts.find((per) => per.id ===
parseInt(req.params.id));
  if (!person) {
    res.status(404).send("Person not found...");
    return;
  }
  person.photo= req.body.photo;
  person.title= req.body.title;
  …
  person.address.line1= req.body.address.line1;
  person.address.line2= req.body.address.line2;
  …
  person.address.country = req.body.address.country;
  person.note = req.body.photo;

  res.send("Person updated...");
});
```


Node JS REST API Example

```
// Delete a person
app.delete("/api/contacts/:id", (req,res) => {
  let person = contacts.find((per) => per.id ===
parseInt(req.params.id));
  if (!person) {
    res.status(404).send("Person not found...");
    return;
  }

  let idx = contacts.indexOf(person);
  contacts.splice(idx,1);
  res.send("Person deleted...");
});

app.listen(port, () => {
  console.log(`ContactsAPI application is listening to
http://localhost:${port}`);
});
```


Debugging

- Express uses the Debug module to internally log information about route matching, middleware functions, application mode, etc.
- To see all internal logs used in Express, set the DEBUG environment variable to Express:* when starting the app:

```
DEBUG=express:* node index.js
```


Cookies

- To use cookies with Express, we need the cookie-parser middleware.
- To install it, use the following code:

```
npm install cookie-parser --save
```

- Now to use cookies with Express, we will require the cookie-parser.
- cookie-parser is a middleware which parses cookies attached to the client request object.

Cookies Example

```
const express = require("express");
const cookieParser = require("cookie-parser");
const app = express();
app.use(cookieParser());
app.get("/setcookie", (req, res) => {
  res.cookie("userData", { name: "MEFGI", age: "13yrs"
});
  res.send("Cookie saved with user data.");
});
app.get("/getcookie", (req, res) => {
  if (req.cookies.userData) {
    res.send(req.cookies.userData);
  } else {
    res.send("User Data cookie not found.");
  }
});
app.get("/delcookie", (req, res) => {
  res.clearCookie("userData");
  res.send("User Data cookie deleted.");
});
```


Sessions

- We will need the Express-session, so install it use
  `npm install --save express-session`
- In this example, we will use the default store for storing sessions, i.e., MemoryStore.
- Never use this in production environments.
- The session middleware handles all things for us, i.e., creating the session, setting the session cookie and creating the session object in req object.
- Whenever we make a request from the same client again, we will have their session information stored with us (given that the server was not restarted).
- We can add more properties to the session object.

Sessions Example

```
const express = require("express");
const session = require("express-session");
const app = express();
app.use(session({ secret: "MEFGI", resave: false,
saveUninitialized: false }));
app.get("/", (req, res) => {
  if (req.session.views) {
    req.session.views++;
    res.send("You visited this website " + req.session.views
+ " times...");
  } else {
    req.session.views = 1;
    res.send("You are visiting this website first time...");
  }
});
app.get("/delsession", (req, res) => {
  req.session.destroy((err) => {
    res.send("Session destroyed...");
  });
});
```


Design patterns

- Design patterns, simply put, are a way for you to structure your solution's code in a way that allows you to gain some kind of benefit.
- Such as faster development speed, code reusability, and so on.
- All patterns lend themselves quite easily to the OOP paradigm.
- A design pattern provides a general reusable solution for the common problems occurs in software design.

Types of Design Patterns

- There are many design patterns but all those can be classified into 3 types:
  1. Creational: These patterns are designed for class instantiation. They can be either class-creation patterns or object-creational patterns.
  2. Structural: Designed with regard to a class's structure and composition. The main goal of most of these patterns is to increase the functionality of the class(es) involved, without changing much of its composition.
  3. Behavioural: These patterns are designed depending on how one class communicates with others.
- We will Learn some most important Design Patterns.

Design patterns

- ***Singletons***
- The singleton patterns restrict the number of instantiations of a "class" to one.
- Creating singletons in Node.js is pretty straightforward, as require is there to help you.

```
//area.js
var PI = Math.PI;
function circle (radius) {
  return radius * radius * PI;
}
module.exports.circle = circle;
```


Design patterns

- ***Singletons***
- It does not matter how many times you will require this module in your application; it will only exist as a single instance.

```
var areaCalc = require('./area');
console.log(areaCalc.circle(5));
```

- Because of this behaviour of require, singletons are probably the most common Node.js design patterns among the modules in NPM.

Design patterns

- ***Observers***
- An object maintains a list of dependents/observers and notifies them automatically on state changes.
- To implement the observer pattern, EventEmitter comes to the rescue.

```
// MyObservable.js
var util = require('util');
var EventEmitter =
require('events').EventEmitter;

function MyObservable() {
  EventEmitter.call(this);
}
util.inherits(MyObservable, EventEmitter);
```


Design patterns

- ***Observers***
- we just made an observable object! To make it useful, let's add some functionality to it.

```
MyObservable.prototype.hello = function (name) {
  this.emit('hello', name);
};
```

- Now our observable can emit event:

```
var MyObservable = require('MyObservable');
var observable = new MyObservable();
observable.on('hello', function (name) {
  console.log(name);
});
observable.hello('john');
```


Design patterns

- ***Factories***
- The factory pattern is a creational pattern that doesn't require us to use a constructor but provides a generic interface for creating objects.
- This pattern can be really useful when the creation process is complex.

```
function MyClass (options) {
  this.options = options;
}
function create(options) {
  // modify the options here if you want
  return new MyClass(options);
}
module.exports.create = create;
```


Caching

- Caching is a commonly used technique to improve the performance of any application, be it desktop, mobile or web.
- When dealing with web applications we can make good use of client-side caching using response headers that all browsers currently support.
- But, what if we have a complex and heavy page that takes 2 second to generate the HTML output?
- Even if we enable client-side cache for this page, the web server will still need to render the page for each different user accessing our web application.

Caching

- This is where server-side cache comes in handy.
- The goal of server side cache is responding to the same content for the same request independently of the client's request.
- In our example above, the first request that reaches our server would still take 2 seconds to generate the HTML, but the following requests would hit the cache instead and the server would be able to send the response in a few milliseconds.

Caching

- There are many ways of doing it, it could be done with NGINX or a CDN like CloudFlare,
- But here we'll see how to do it with Node.js and Express with minimal work and in a flexible way.
- Express is the most extensible web framework.
- The framework's middleware architecture makes it easy to plug-in extra features with minimal effort and in a standardized way.

Caching

- There are many ways of doing it, it could be done with NGINX or a CDN like CloudFlare,
- But here we'll see how to do it with Node.js and Express with minimal work and in a flexible way.
- Express is the most extensible web framework.
- The framework's middleware architecture makes it easy to plug-in extra features with minimal effort and in a standardized way.
- We'll make use of memory-cache npm module in order to be able to add content to cache.

Caching

```
const cache = require('memory-cache');
let cacheMiddleware = (duration) => {
  return (req, res, next) => {
    let key =  '__express__' + req.originalUrl || req.url;
    let cacheContent = cache.get(key);
    if(cacheContent){
      res.send( cacheContent );
      return;
    }else{
      res.sendResponse = res.send;
      res.send = (body) => {
          cache.put(key,body,duration*1000);
          res.sendResponse(body);
      }
      next();
    }
  }
}
```


Caching

- To use this cache middleware add it to your express route for whichever page you want cache to be used:

```
app.get("/", cacheMiddleware(30),
    (req, res) => {
        ...
});
```


Scalability

- Scalability in Node.js is not an afterthought.
- It's something that's baked into the core of the runtime.
- Node is named Node to emphasize the idea that a Node application should comprise multiple small distributed nodes that communicate with each other.
- Are you running multiple nodes for your Node applications?
- Are you running a Node process on every CPU core of your production machines and load balancing all the requests among them?
- Did you know that Node has a built-in module to help with that?

Scalability

- Node's cluster module not only provides an out-of-the-box solution to utilizing the full CPU power of a machine, but it also helps with increasing the availability of your Node processes and provides an option to restart the whole application with a zero downtime.
- The cluster module allows easy creation of child processes that all share server ports.

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;
```


Scalability

```
if (cluster.isMaster) {
  console.log(`Master ${process.pid} is running`);
  // Fork workers.
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }
  cluster.on('exit', (worker, code, signal) => {
    console.log(`worker ${worker.process.pid} died`);
  });
} else {
  // Workers can share any TCP connection
  // In this case it is an HTTP server
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('hello world\n');
  }).listen(8000);
  console.log(`Worker ${process.pid} started`);
}
```


Scalability

- We create a master process and that master process forks a number of worker processes and manages them.
- Each worker process represents an instance of the application that we want to scale.
- All incoming requests are handled by the master process, which is the one that decides which worker process should handle an incoming request.

Scalability

- The master process's job is easy because it actually just uses a round-robin algorithm to pick a worker process.
- This is enabled by default on all platforms except Windows and it can be globally modified to let the load-balancing be handled by the operation system itself.
- The round-robin algorithm distributes the load evenly across all available processes on a rotational basis.
- The first request is forwarded to the first worker process, the second to the next worker process in the list, and so on.
- When the end of the list is reached, the algorithm starts again from the beginning.

Scalability

- This is one of the simplest and most used load balancing algorithms.
- But it's not the only one.
- More featured algorithms allow assigning priorities and selecting the least loaded server or the one with the fastest response time.

Review

- Events and Event Loop, Timers, Error Handling, Buffers, Streams, Work with File System, Networking with Node (TCP, UDP and HTTP clients and servers), Web Module, Debugging, Node JS REST API, Sessions and Cookies, Design patterns, Caching, Scalability.

Thank You.