AttainU

Attain Your Dreams

Let’s roll by 09:05 !!!

Node.js 1

Gopi Krishnan R

Topics to be Discussed

- What is server?
- Client Server Model
- Creating a Node.js Application
- Modules

Server

A server is a computer connected to a network of other workstations called 'clients'. Client computers request information from the server over the network.

Servers tend to have more storage, memory and processing power than a normal workstation.

Types of servers

- Peer to Peer

  In a peer-to-peer network, each workstation plays the role of client and server. The server is the computer that is providing information or services to the other computer. The networks rely on each other to provide and share information and services. These are typically only used in small offices or homes.

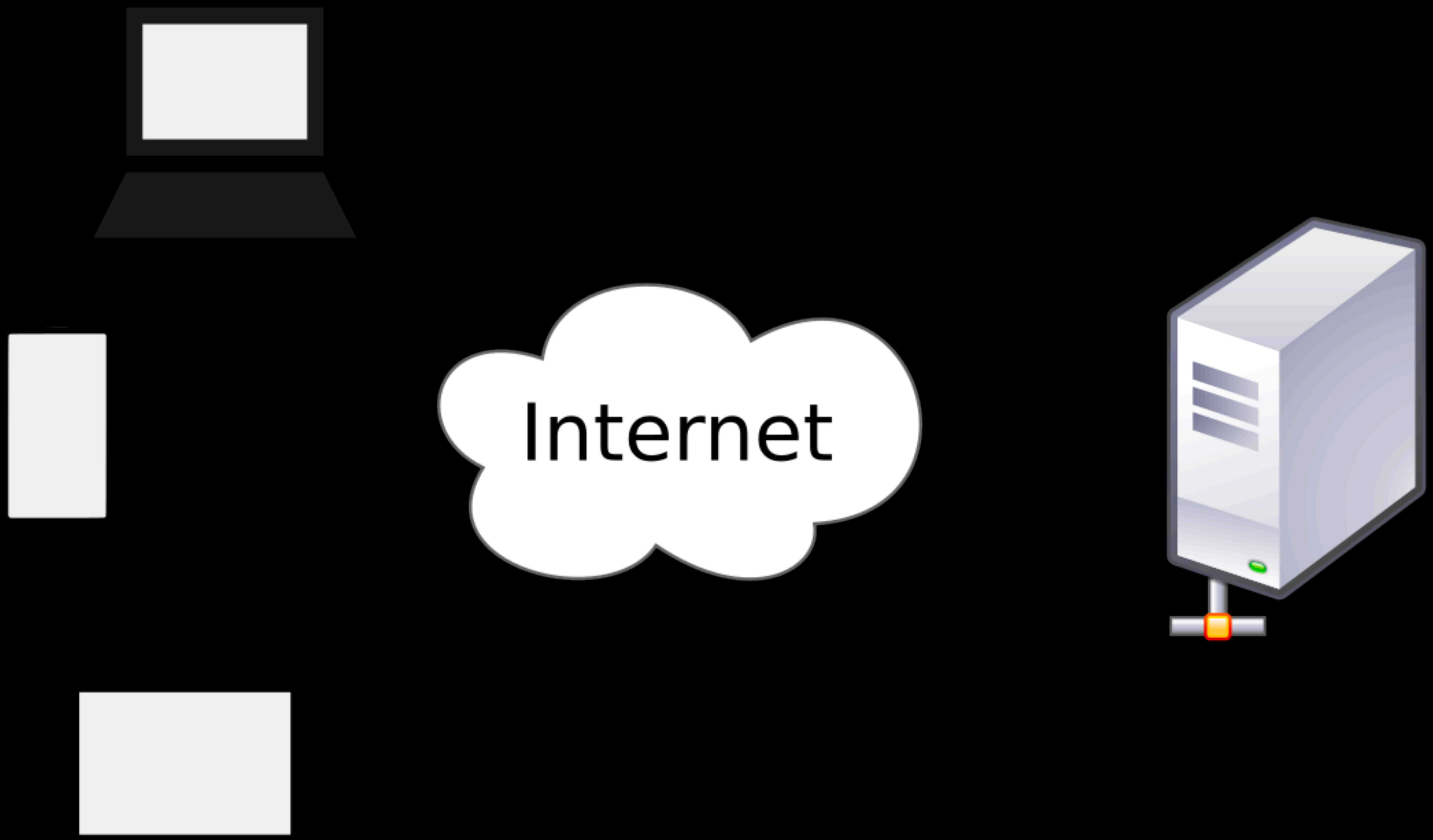
- Client / Server

  In a client/server network there is one server that is only dedicated to provide services to the other client workstations. You can have multiple servers in these networks but they can not function as workstations. Client/server networks are the most commonly used.

- Specialized Servers

  There are a few common types of specialized servers. These include DNS, Web or Mail servers. These can hold databases of internet domains, supply access to websites or to email accounts.

Client Server Model



Node.js

Node.js is an open-source, cross-platform, JavaScript runtime environment. It executes JavaScript code outside of a browser. Node.js runs the V8 JavaScript engine outside of the browser.

A Node.js app is run in a single process, without creating a new thread for every request. Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm. Node.js = Runtime Environment + JavaScript Library

When Node.js needs to perform an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

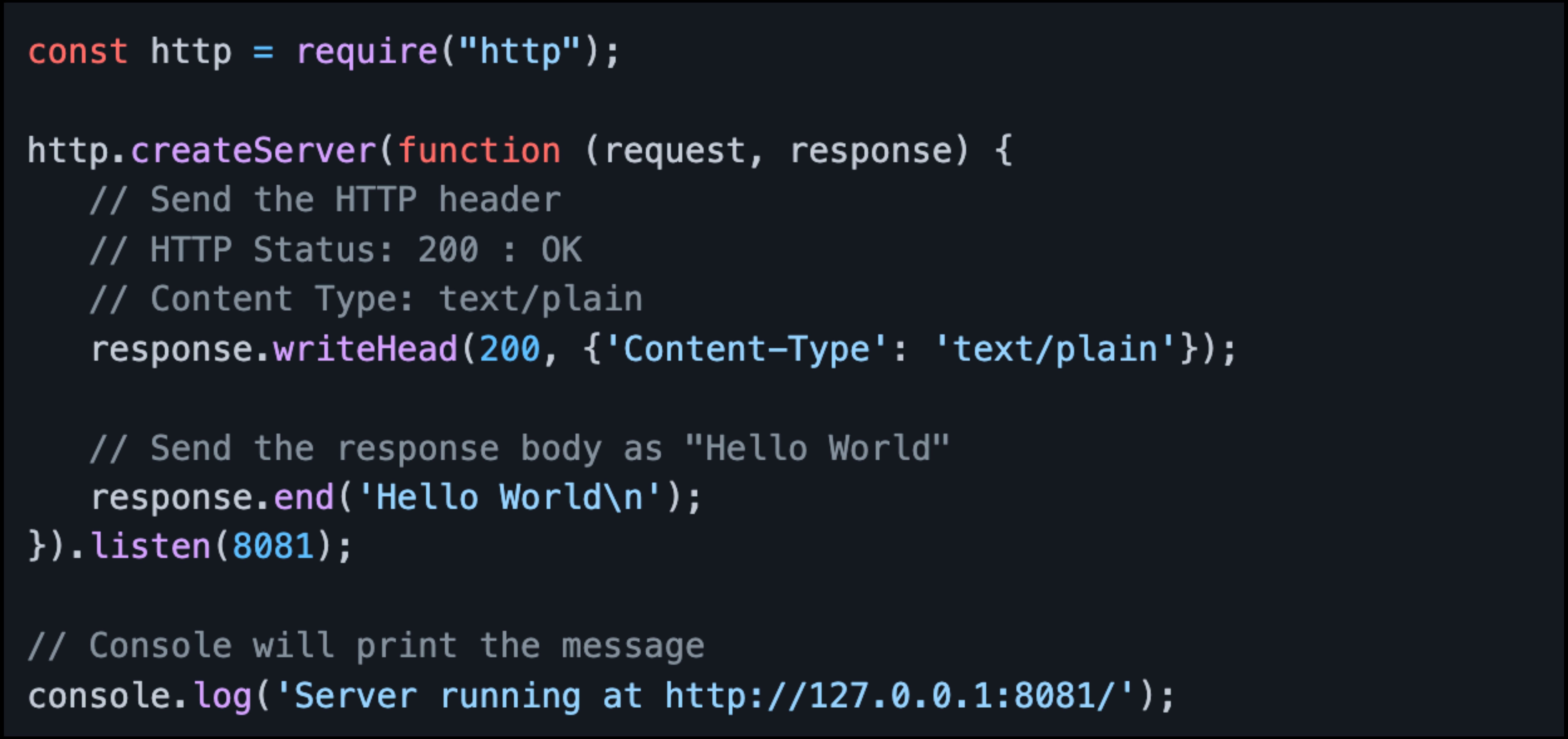
Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.

In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers - you are in charge of deciding which ECMAScript version to use by changing the Node.js version

Basic Server

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

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



```javascript
const http = require("http");

http.createServer(function (request, response) {
   // Send the HTTP header
   // HTTP Status: 200 : OK
   // Content Type: text/plain
   response.writeHead(200, {'Content-Type': 'text/plain'});

   // Send the response body as "Hello World"
   response.end('Hello World\n');
}).listen(8081);

// Console will print the message
console.log('Server running at http://127.0.0.1:8081/');
```

Modules

Module in Node.js is a simple or complex functionality organized in single or multiple JavaScript files which can be reused throughout the Node.js application.

Each module in Node.js has its own context, so it cannot interfere with other modules or pollute global scope. Also, each module can be placed in a separate .js file under a separate folder.

How to expose different types as a module using module.exports?

- The module.exports or exports is a special object which is included in every JS file in the Node.js application by default. module is a variable that represents current module and exports is an object that will be exposed as a module. So, whatever you assign to module.exports or exports, will be exposed as a module.

Node.js includes three types of modules:

- Core Modules: (Built-In Modules)
  - Node Js Core Modules comes with it's Intallation by default. You can use them as per application requirments
  - To use Modules, we need to use a function in Node called require. The require() function will return an object, function, property or any other JavaScript type, depending on what the specified module returns. The demo server we previously built used the built-in require for http module
- Local Modules
  - It's user defined node.js module.
  - Local modules are mainly used for specific projects and locally available in separate files or folders within project folders.

Third Party Modules

- The 3rd party modules can be downloaded using NPM (Node Package Manager).
- 3rd party modules can be install inside the project folder or globally.
- 3rd party modules can be downloaded using NPM and then can be used in application by using require() function.

AttainU

Attain Your Dreams

Let’s roll by 09:05 !!!

Node.js 2

Gopi Krishnan R

Topics to be Discussed

- Modules
- HTTP Protocol
- Introduction to express.js
- Package.json file
- Request and Response object

Modules

Module in Node.js is a simple or complex functionality organized in single or multiple JavaScript files which can be reused throughout the Node.js application.

Each module in Node.js has its own context, so it cannot interfere with other modules or pollute global scope. Also, each module can be placed in a separate .js file under a separate folder.

How to expose different types as a module using module.exports?

- The module.exports or exports is a special object which is included in every JS file in the Node.js application by default. module is a variable that represents current module and exports is an object that will be exposed as a module. So, whatever you assign to module.exports or exports, will be exposed as a module.

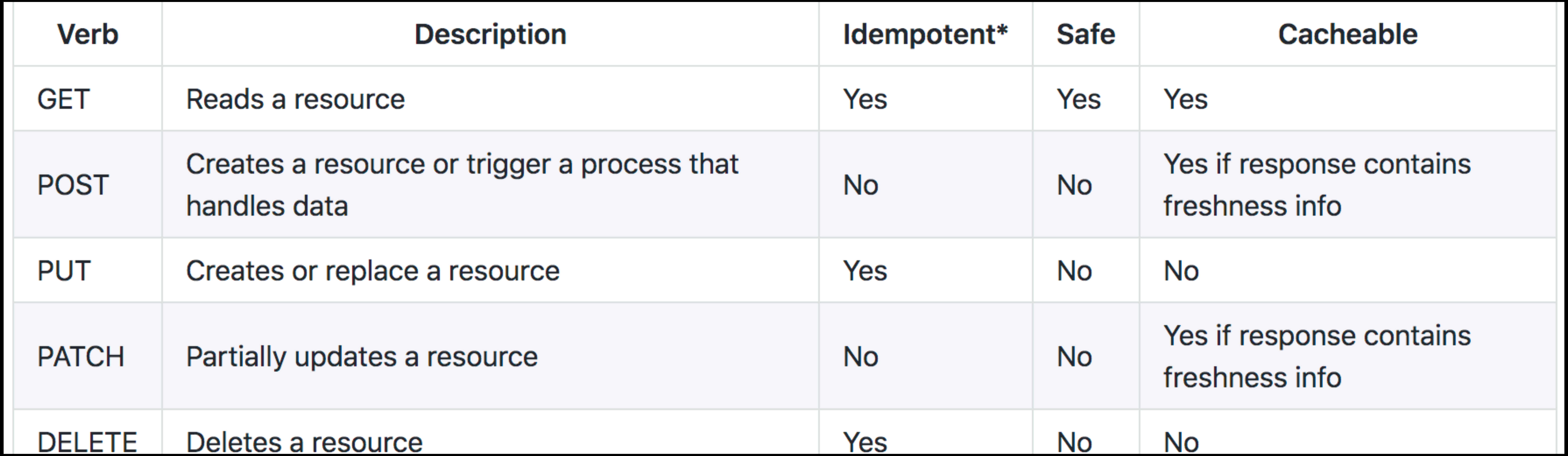
Node.js includes three types of modules:

- Core Modules: (Built-In Modules)
  - Node Js Core Modules comes with it's Intallation by default. You can use them as per application requirments
  - To use Modules, we need to use a function in Node called require. The require() function will return an object, function, property or any other JavaScript type, depending on what the specified module returns. The demo server we previously built used the built-in require for http module
- Local Modules
  - It's user defined node.js module.
  - Local modules are mainly used for specific projects and locally available in separate files or folders within project folders.

Third Party Modules

- The 3rd party modules can be downloaded using NPM (Node Package Manager).
- 3rd party modules can be install inside the project folder or globally.
- 3rd party modules can be downloaded using NPM and then can be used in application by using require() function.

HTTP Protocol



Verb	Description	Idempotent*	Safe	Cacheable
GET	Reads a resource	Yes	Yes	Yes
POST	Creates a resource or trigger a process that handles data	No	No	Yes if response contains freshness info
PUT	Creates or replace a resource	Yes	No	No
PATCH	Partially updates a resource	No	No	Yes if response contains freshness info
DELETE	Deletes a resource	Yes	No	No

TCP

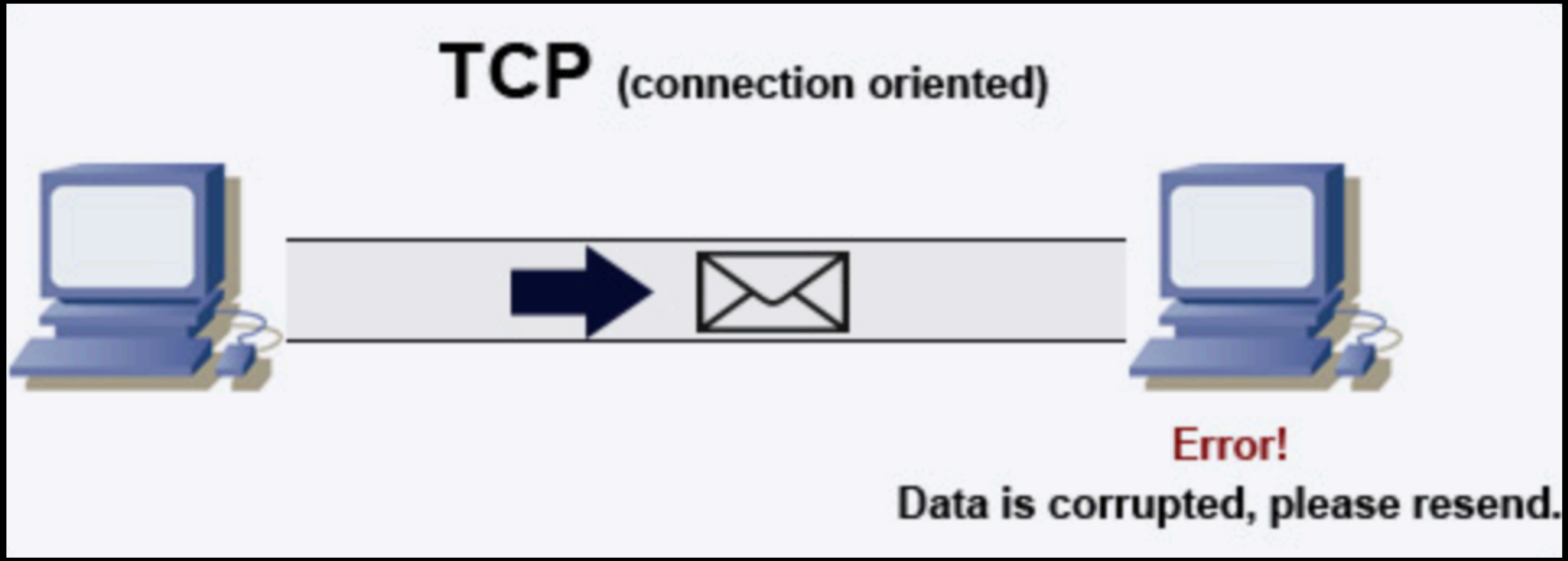
TCP is a connection-oriented protocol over an IP network. Connection is established and terminated using a handshake. All packets sent are guaranteed to reach the destination in the original order and without corruption.

If the sender does not receive a correct response, it will resend the packets. If there are multiple timeouts, the connection is dropped.

TCP is useful for applications that require high reliability but are less time critical. Some examples include web servers, database info, SMTP, FTP, and SSH.

Use TCP over UDP when:

- You need all of the data to arrive intact
- You want to automatically make a best estimate use of the network throughput



UDP

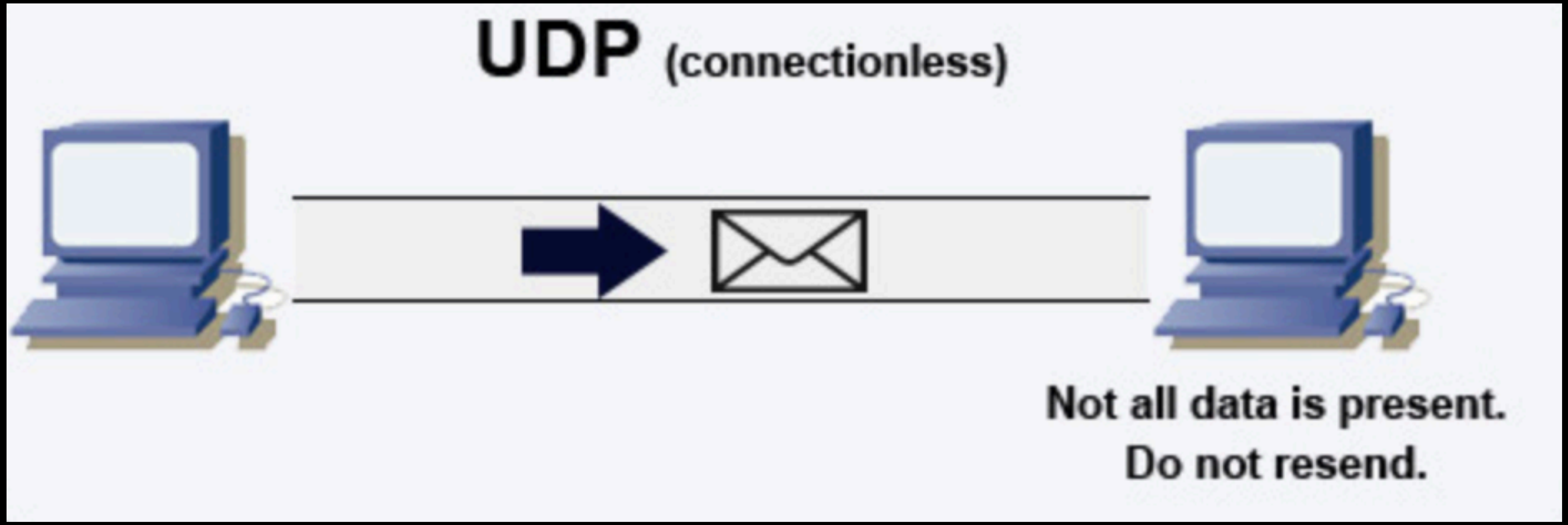
User DataGram protocol is connection-less. DataGrams(analogous to packets) might reach their destination out of order or not at all.

UDP can broadcast.

UDP is less reliable but works well in real time use cases such as VoIP, video chat, streaming, and realtime multiplayer games.

Use UDP over TCP when:

- You need the lowest latency
- Late data is worse than loss of data



Express

- Most developers adore Node.js for its raw speed. Express provides a thin layer on top of Node.js with web application features such as basic routing, middleware, template engine and static files serving, so the drastic I/O performance of Node.js doesn’t get compromised.

- Express is a minimal, un-opinionated framework. it doesn’t apply any of the prevalent design patterns such as MVC, MVP, MVVM or whatever is trending out of the box. For fans of simplicity, this is a big plus among all other frameworks because you can build your application with your own preference and no unnecessary learning curve. This is especially advantageous when creating a new personal project with no historical burden, but as the project or developing team grows, lack of standardization may lead to extra work for project/code management, and worst case scenario it may lead to the inability to maintain.

```javascript
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => res.send('Hello World!'))

app.listen(port, () => console.log(`Example app listening on port ${port}!`))
```

Request

Request object

- req.body
  - It contains key-value pairs of data submitted in the request body. By default, it is undefined, and is populated when you use body-parsing middleware such as body-parser.
- req.params
  - An object containing properties mapped to the named route ?parameters?. For example, if you have the route /user/:name, then the "name" property is available as req.params.name. This object defaults to {}.
- req.query
  - An object containing a property for each query string parameter in the route.

Request Object Methods

Following is a list of some generally used request object methods:

- req.param(name [, defaultValue])
  - This method is used to fetch the value of param name when present.

Response

The Response object (res) specifies the HTTP response which is sent by an Express app when it gets an HTTP request.
What it does

- It sends response back to the client browser.
- Once you res.send() or res.redirect() or res.render(), you cannot do it again, otherwise, there will be uncaught error.

Response Object Methods
Following are some methods:

- Response End method
    - Syntax:
        - res.end([data] [, encoding])
    - This method is used to end the response process.
    - Example:

```
res.end();
```

- Response JSON method:
    - Syntax:
        - res.json([body])
    - This method returns the response in JSON format.
    - Example:

```
res.json(null)
res.json({ name: 'ajeet' })
```

- Response REDIRECT method:
  - Syntax:
    - res.redirect([status,] path)
  - This method is used to redirect to the URL dervied from the specified path, with specified HTTP status code status.
  - Following are a few examples –

```
res.redirect('/foo/bar');
res.redirect('http://example.com');
res.redirect(301, 'http://example.com');
```

- Response RENDER method:
  - Syntax:
    - res.render(view [, locals] [, callback])
    - This method is used to render a view and sends the rendered HTML string to the client.
    - Following are a few examples –

```
// send the rendered view to the client
res.render('index');

// pass a local variable to the view
res.render('user', { name: 'Tobi' }, function(err, html) {
// ...
});
```

- Response SEND method:
  - Syntax:
    - res.send([body])
    - This method is used to send the HTTP response.
    - Following are a few examples –

```
res.send(new Buffer('whoop'));
res.send({ some: 'json' });
res.send('<p>some html</p>');
```

- Response STATUS method:
  - Syntax:
    - res.status(code)
    - This method is used to set the HTTP status for the response.
    - Following are a few examples –

```
res.status(403).end();
res.status(400).send('Bad Request');
```

AttainU

Attain Your Dreams

Let’s roll by 09:05 !!!

Node.js 3

Gopi Krishnan R

Topics to be Discussed

- Nodemon
- Buffers
- Streams
- File-System

Nodemon

nodemon is a tool that helps develop node.js based applications by automatically restarting the node application when file changes in the directory are detected.

nodemon does not require any additional changes to your code or method of development. nodemon is a replacement wrapper for node, to use nodemon replace the word node on the command line when executing your script.

```
npm install -g nodemon
```

Buffer

Why Buffers?

- Pure JavaScript, while great with unicode-encoded strings, does not handle straight binary data very well. This is fine on the browser, where most data is in the form of strings. However, Node.js servers have to also deal with TCP streams and reading and writing to the filesystem, both of which make it necessary to deal with purely binary streams of data.
- One way to handle this problem is to just use strings anyway, which is exactly what Node.js did at first. However, this approach is extremely problematic to work with; It's slow, makes you work with an API designed for strings and not binary data, and has a tendency to break in strange and mysterious ways.
- Don't use binary strings. Use buffers instead!

What Are Buffers?

- The Buffer class in Node.js is designed to handle raw binary data. The Buffer class was introduced as part of the Node.js API to make it possible to manipulate or interact with streams of binary data. Each buffer corresponds to some raw memory allocated outside V8. Buffers act somewhat like arrays of integers, but aren't resizable and have a whole bunch of methods specifically for binary data. The integers in a buffer each represent a byte and so are limited to values from 0 to 255 inclusive. When using console.log() to print the Buffer instance, you'll get a chain of values in hexadecimal values.

Where You See Buffers:

- In the wild, buffers are usually seen in the context of binary data coming from streams, such as fs.createReadStream.

Creating Buffers

`Method 1`

Following is the syntax to create an uninitiated Buffer of 10 octets –

```
let buf = new Buffer(10);
console.log(buf);
```

output:

```
<Buffer 00 00 00 00 00 00 00 00 00 00>
```

`Method 2`

Following is the syntax to create a Buffer from a given array –

```
let buf = new Buffer([10, 20, 30, 40, 50]);
console.log(buf);
```

Creating Buffer

`Method 3`

Following is the syntax to create a Buffer from a given string and optionally encoding type –

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

Though "utf8" is the default encoding, you can use any of the following encodings "ascii", "utf8", "utf16le", "ucs2", "base64" or "hex".

output

```
<Buffer 53 69 6d 70 6c 79 20 45 61 73 79 20 4c 65 61 72 6e 69 6e 67>
```

Streams

Streams are one of the fundamental concepts that power Node.js applications. They are data-handling method and are used to read or write input into output sequentially.

Streams are a way to handle reading/writing files, network communications, or any kind of end-to-end information exchange in an efficient way.

What makes streams unique, is that instead of a program reading a file into memory all at once like in the traditional way, streams read chunks of data piece by piece, processing its content without keeping it all in memory.

This makes streams really powerful when working with large amounts of data, for example, a file size can be larger than your free memory space, making it impossible to read the whole file into the memory in order to process it. That’s where streams come to the rescue!

Using streams to process smaller chunks of data, makes it possible to read larger files.

Let’s take a “streaming” services such as YouTube or Netflix for example: these services don’t make you download the video and audio feed all at once. Instead, your browser receives the video as a continuous flow of chunks, allowing the recipients to start watching and/or listening almost immediately.

However, streams are not only about working with media or big data. They also give us the power of ‘*composability*’ in our code. Designing with *composability* in mind means several components can be combined in a certain way to produce the same type of result. In Node.js it’s possible to compose powerful pieces of code by piping data to and from other smaller pieces of code, using streams.

Streams basically provide two major advantages compared to other data handling methods:

- Memory efficiency: you don’t need to load large amounts of data in memory before you are able to process it
- Time efficiency: it takes significantly less time to start processing data as soon as you have it, rather than having to wait with processing until the entire payload has been transmitted

In Node.js, there are four types of streams –

- Readable – Stream which is used for read operation.
- Writable – Stream which is used for write operation.
- Duplex – Stream which can be used for both read and write operation.
- Transform – A type of duplex stream where the output is computed based on input.

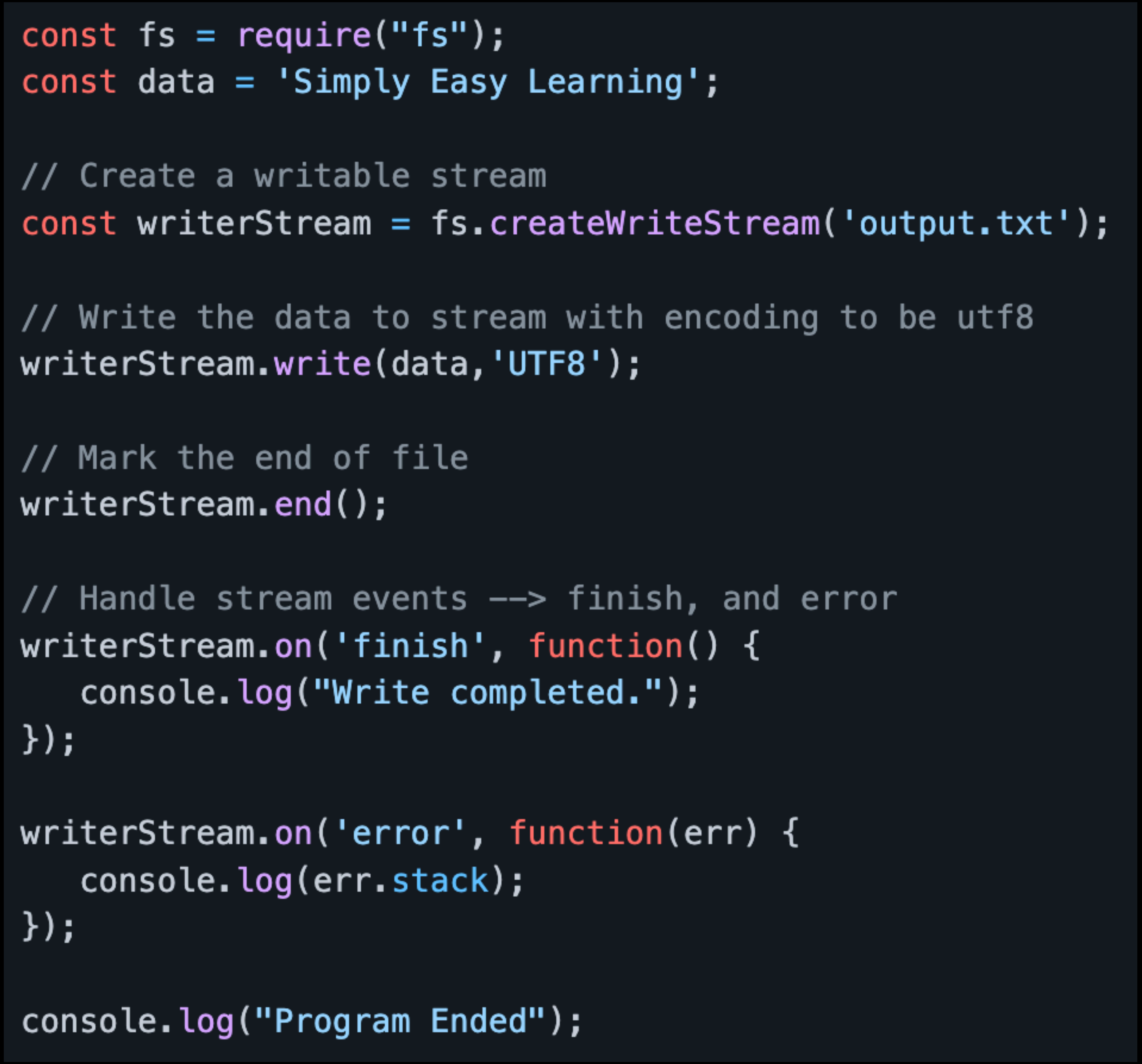
Each type of Stream is an EventEmitter instance and throws several events at different instance of times. For example, some of the commonly used events are :

- data – This event is fired when there is data is available to read.
- end – This event is fired when there is no more data to read.
- error – This event is fired when there is any error receiving or writing data.
- finish – This event is fired when all the data has been flushed to underlying system.

Reading from stream

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

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

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

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

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

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

console.log("Program Ended");
```

Writing to a stream



```
const fs = require("fs");
const data = 'Simply Easy Learning';

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

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

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

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

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

console.log("Program Ended");
```

FileSystem

The Node File System (fs) module can be imported using the following syntax –

```
const fs = require("fs")
```

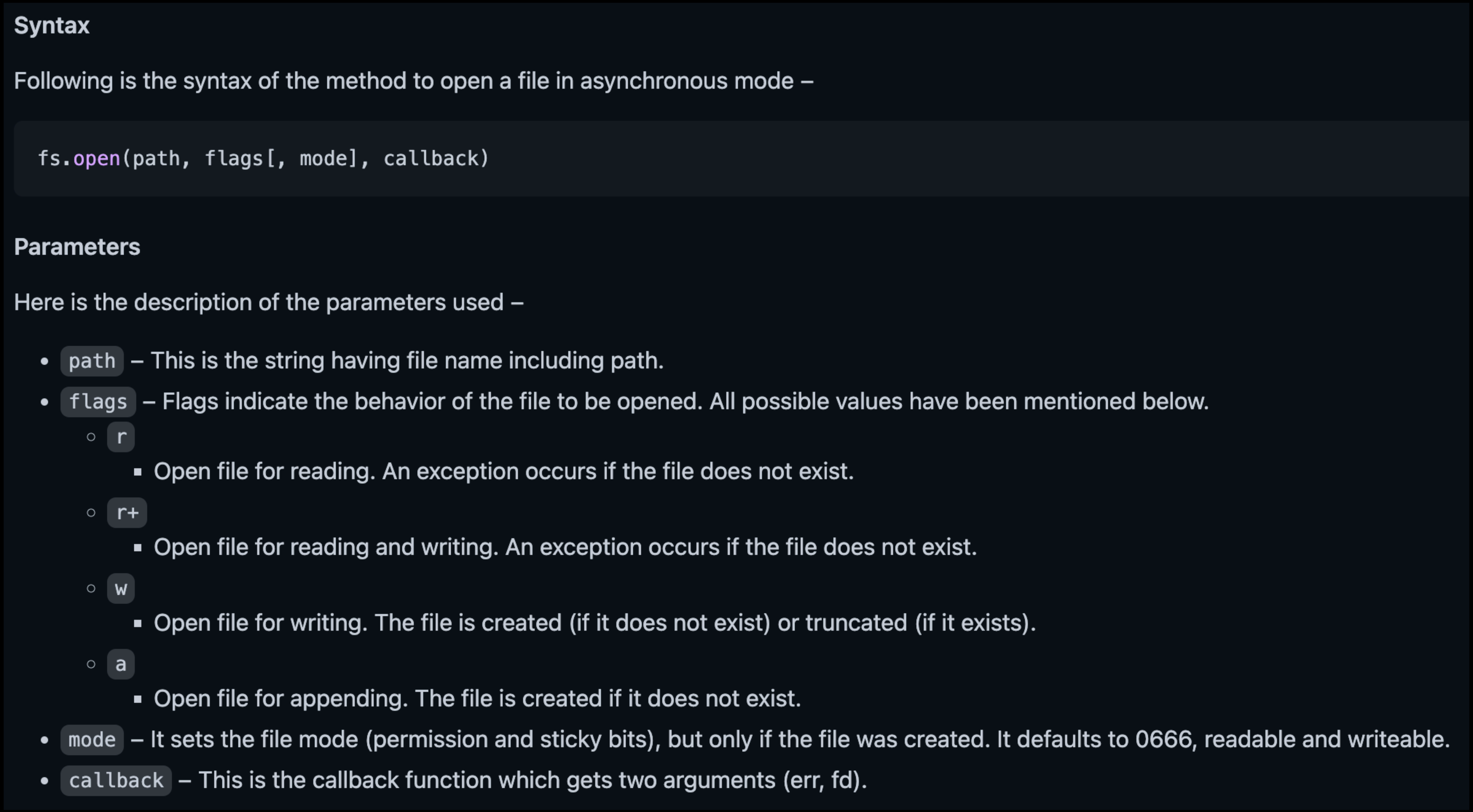
Synchronous vs Asynchronous

Every method in the fs module has synchronous as well as asynchronous forms. Asynchronous methods take the last parameter as the completion function callback and the first parameter of the callback function as error. It is better to use an asynchronous method instead of a synchronous method, as the former never blocks a program during its execution, whereas the second one does.

Async/ Sync Example

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

// Asynchronous read
fs.readFile('input.txt', function (err, data) {
   if (err) {
      return console.error(err);
   }
   console.log("Asynchronous read: " + data.toString());
});

// Synchronous read
const data = fs.readFileSync('input.txt');
console.log("Synchronous read: " + data.toString());
```

Open a file



Syntax

Following is the syntax of the method to open a file in asynchronous mode –

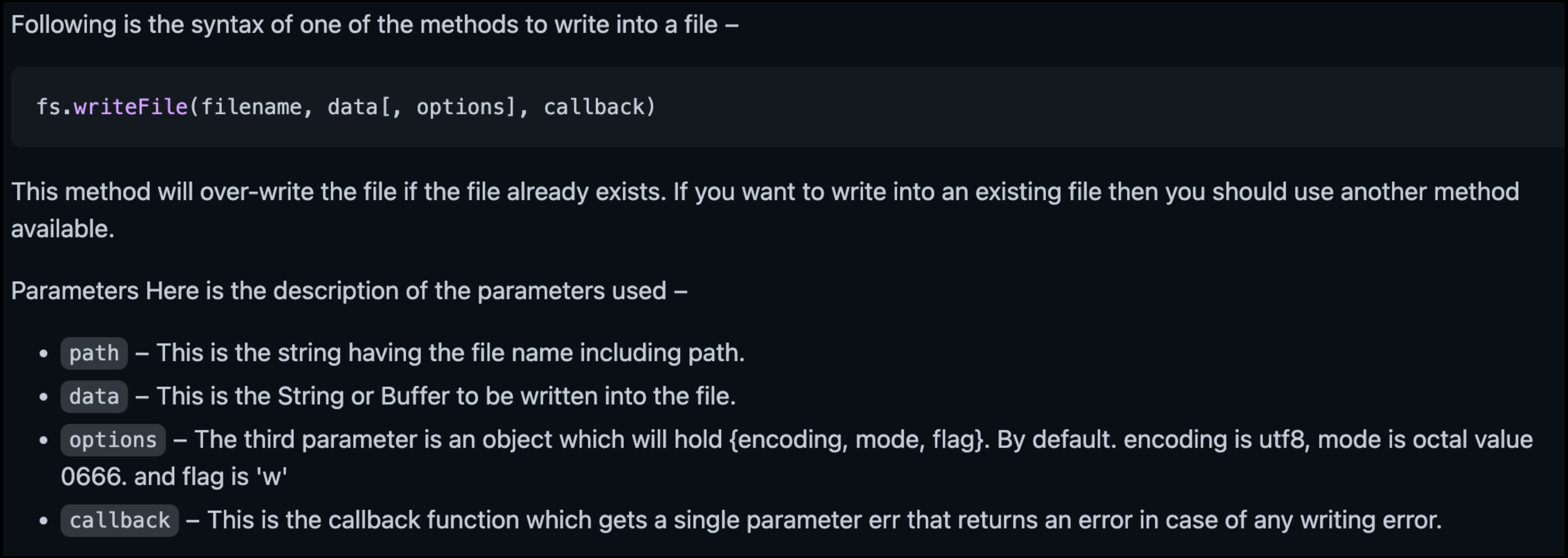
```
fs.open(path, flags[, mode], callback)
```

Parameters

Here is the description of the parameters used –

- `path` – This is the string having file name including path.
- `flags` – Flags indicate the behavior of the file to be opened. All possible values have been mentioned below.
  - `r`
    - Open file for reading. An exception occurs if the file does not exist.
  - `r+`
    - Open file for reading and writing. An exception occurs if the file does not exist.
  - `w`
    - Open file for writing. The file is created (if it does not exist) or truncated (if it exists).
  - `a`
    - Open file for appending. The file is created if it does not exist.
- `mode` – It sets the file mode (permission and sticky bits), but only if the file was created. It defaults to 0666, readable and writeable.
- `callback` – This is the callback function which gets two arguments (err, fd).

Writing a file



Following is the syntax of one of the methods to write into a file –

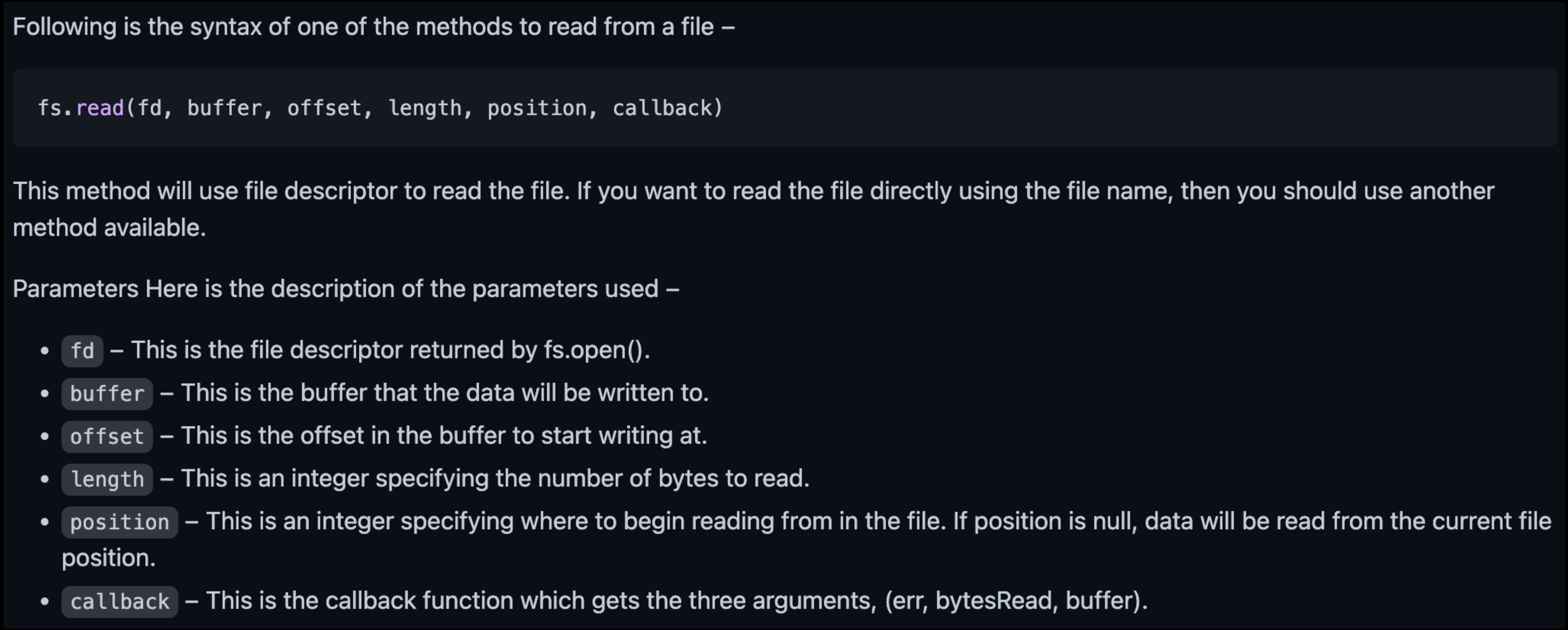
```
fs.writeFile(filename, data[, options], callback)
```

This method will over-write the file if the file already exists. If you want to write into an existing file then you should use another method available.

Parameters Here is the description of the parameters used –

- `path` – This is the string having the file name including path.
- `data` – This is the String or Buffer to be written into the file.
- `options` – The third parameter is an object which will hold {encoding, mode, flag}. By default. encoding is utf8, mode is octal value 0666. and flag is 'w'
- `callback` – This is the callback function which gets a single parameter err that returns an error in case of any writing error.

Read a file



Following is the syntax of one of the methods to read from a file –

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

This method will use file descriptor to read the file. If you want to read the file directly using the file name, then you should use another method available.

Parameters Here is the description of the parameters used –

- `fd` – This is the file descriptor returned by fs.open().
- `buffer` – This is the buffer that the data will be written to.
- `offset` – This is the offset in the buffer to start writing at.
- `length` – This is an integer specifying the number of bytes to read.
- `position` – This is an integer specifying where to begin reading from in the file. If position is null, data will be read from the current file position.
- `callback` – This is the callback function which gets the three arguments, (err, bytesRead, buffer).

AttainU
Attain Your Dreams

Let’s roll by 09:05 !!!

Node.js 4

Gopi Krishnan R

Topics to be Discussed

- File-System
- Routing using express

FileSystem

The Node File System (fs) module can be imported using the following syntax –

```
const fs = require("fs")
```

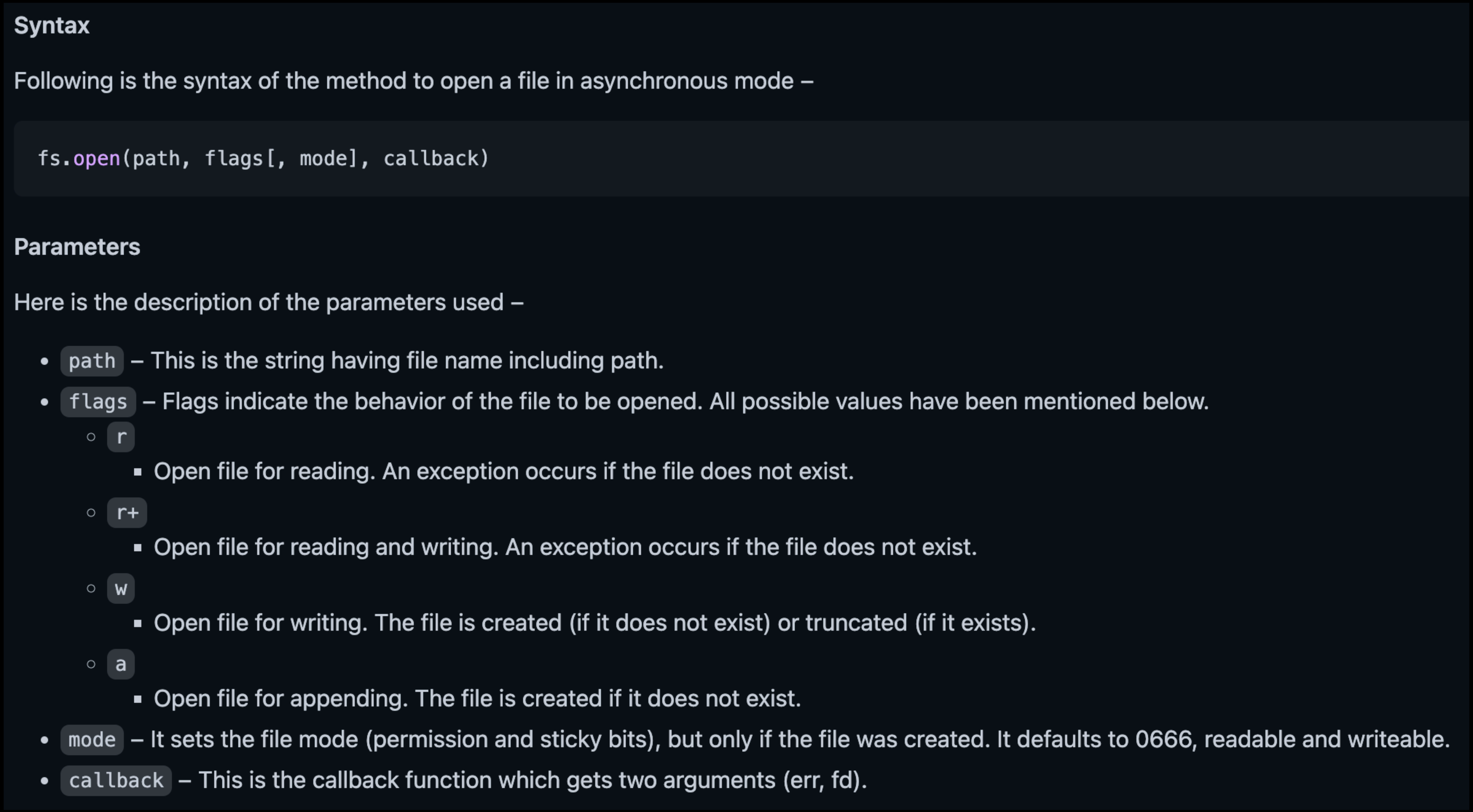
Synchronous vs Asynchronous

Every method in the fs module has synchronous as well as asynchronous forms. Asynchronous methods take the last parameter as the completion function callback and the first parameter of the callback function as error. It is better to use an asynchronous method instead of a synchronous method, as the former never blocks a program during its execution, whereas the second one does.

Async/ Sync Example

```javascript
const fs = require("fs");

// Asynchronous read
fs.readFile('input.txt', function (err, data) {
   if (err) {
      return console.error(err);
   }
   console.log("Asynchronous read: " + data.toString());
});

// Synchronous read
const data = fs.readFileSync('input.txt');
console.log("Synchronous read: " + data.toString());
```

Open a file



Syntax

Following is the syntax of the method to open a file in asynchronous mode –

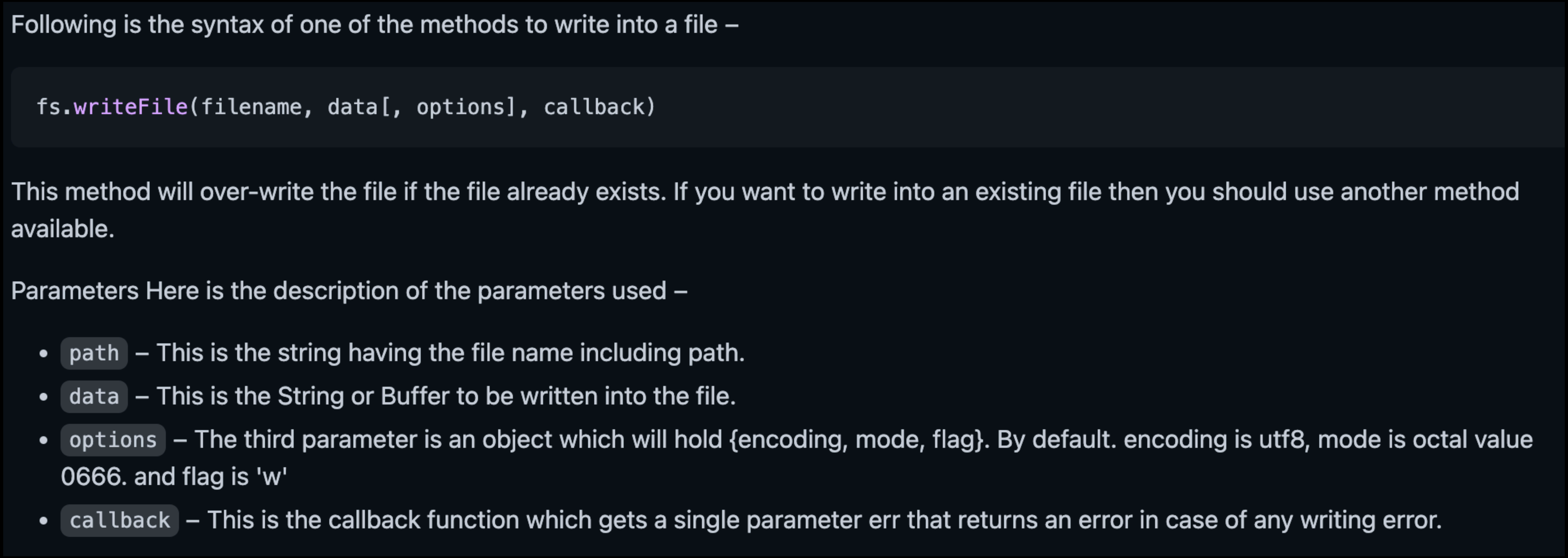
```
fs.open(path, flags[, mode], callback)
```

Parameters

Here is the description of the parameters used –

- `path` – This is the string having file name including path.
- `flags` – Flags indicate the behavior of the file to be opened. All possible values have been mentioned below.
    - `r`
        - Open file for reading. An exception occurs if the file does not exist.
    - `r+`
        - Open file for reading and writing. An exception occurs if the file does not exist.
    - `w`
        - Open file for writing. The file is created (if it does not exist) or truncated (if it exists).
    - `a`
        - Open file for appending. The file is created if it does not exist.
- `mode` – It sets the file mode (permission and sticky bits), but only if the file was created. It defaults to 0666, readable and writeable.
- `callback` – This is the callback function which gets two arguments (err, fd).

Writing a file



Following is the syntax of one of the methods to write into a file –

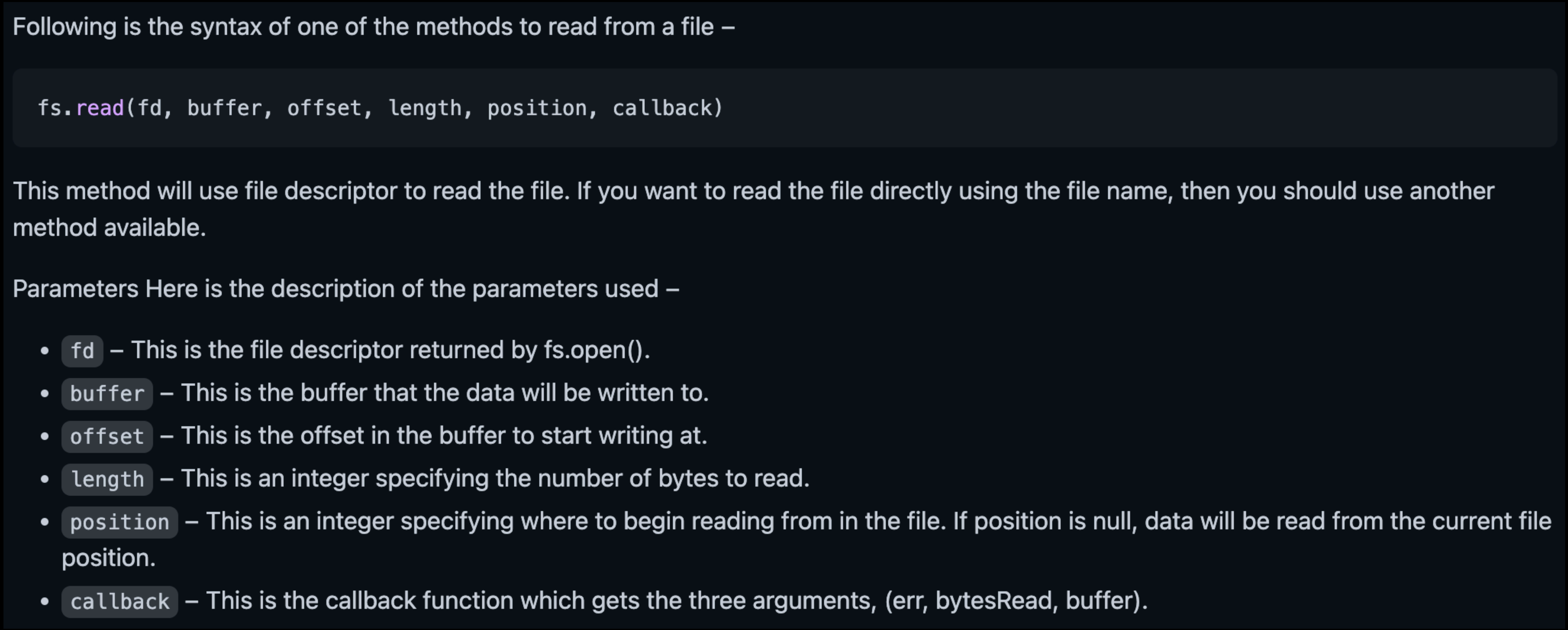
```
fs.writeFile(filename, data[, options], callback)
```

This method will over-write the file if the file already exists. If you want to write into an existing file then you should use another method available.

Parameters Here is the description of the parameters used –

- `path` – This is the string having the file name including path.
- `data` – This is the String or Buffer to be written into the file.
- `options` – The third parameter is an object which will hold {encoding, mode, flag}. By default. encoding is utf8, mode is octal value 0666. and flag is 'w'
- `callback` – This is the callback function which gets a single parameter err that returns an error in case of any writing error.

Read a file



Following is the syntax of one of the methods to read from a file –

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

This method will use file descriptor to read the file. If you want to read the file directly using the file name, then you should use another method available.

Parameters Here is the description of the parameters used –

- `fd` – This is the file descriptor returned by fs.open().
- `buffer` – This is the buffer that the data will be written to.
- `offset` – This is the offset in the buffer to start writing at.
- `length` – This is an integer specifying the number of bytes to read.
- `position` – This is an integer specifying where to begin reading from in the file. If position is null, data will be read from the current file position.
- `callback` – This is the callback function which gets the three arguments, (err, bytesRead, buffer).

Routing

Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).

Each route can have one or more handler functions, which are executed when the route is matched.

Route definition takes the following structure:

app.METHOD(PATH, HANDLER)

Where:

- app is an instance of express.
- METHOD is an HTTP request method, in lowercase.
- PATH is a path on the server.
- HANDLER is the function executed when the route is matched.

The following examples illustrate defining simple routes.

Parameters in Express

Express provides several different "parameters" objects:

- req.params for path parameters (aka route parameters) signified with a : in the route matcher
- req.query for query parameters which appear after the ? in the URL
- req.body for post parameters which appear inside the request body

Query Parameters in Express

For query parameters like ?season=winter&weather=cold

Express will grab the name and value from the query string, and put it into the request.query object for you to use later

Example

```
app.get('/hello', (request, response)=> {
    response.send('Hello, ' + request.query.friend + '!')
});
```

Then visiting http://localhost:5000/hello?friend=Gandalf should send the Hello, Gandalf!

Path Parameters in Express

The special character : means "this is a path parameter"

Example

Path: /hello/Gandalf

Route: /hello/:name

Params: {name: 'Gandalf'}

Express will grab the value from the path itself, and put it into
the request.params object for you to use later.

In the above example, the value of name, Gandalf will be accessible in the route
as request.params.name

Body Parameters in Express

Body parameters are the content which comes packed in the HTTP's request body. It is the payload of the request. There can be two formats primarily, in which the payload might come in the body of the request, one being URL encoded (data in URL), other being in JSON format.

The most basic way to send data to the server is through the HTML forms.

Since request bodies can appear in several different formats, you need to use the correct middleware to extract them.

- express.urlencoded parses incoming requests with URL-encoded payloads
- express.json parses incoming requests with JSON payloads