Web Frame Work Using JavaScript

Vivek Sharma

The Course Break Down:

- Class 1: Mean Stack Fundamentals.
- Class 2: Learning Node.js
- Class 3: Implementing HTTP Services in Node.js
- Class 4: Scaling the Applications Using Node.js
- Class 5: Understanding NoSql and MongoDB
- Class 6: Understanding Express and Implementation.
- Class 7: Understanding Angular
- Class 8: Understanding the Angular Directives and Angular Web Application.
- Class 9: Creating a Shopping Cart.
- Class 10: Creating another project.

WELCOME

Some Points before we start off:

- Introduction.
- Grading/ Assignments.
- About the Class. [Basic to Professional]
- Pace.
- Interview Preparation.
- Labs.

Today's Overview

1. Core Node JS
2. Adding Workflow items to Node.js
3. Considerations for the Mobile Development.
4. Types of Mobile Apps

Learning Node

Node.js Events

1. We already have a way to execute some code based on some occurrence (event) using callbacks.
2. A more general concept for handling occurrences of significance is events. An event is like a broadcast, while a callback is like a handshake.
3. A component that raises events knows nothing about its clients, while a component that uses callbacks knows a great deal.
4. This makes events ideal for scenarios where the significance of the occurrence is determined by the client. Maybe the client wants to know, maybe it doesn't. Registering multiple clients is also simpler with this even as we will see in this section.
5. Node.js comes with built-in support for events baked into the core events module. As always, use require('events') to load the module. The events module has one simple class "EventEmitter", which we present next.

Event Emitter Class

- EventEmitter is a class designed to make it easy to emit events (no surprise there) and subscribe to raised events.[events/1basic.js]

```
var EventEmitter = require('events').EventEmitter;
var emitter = new EventEmitter();
// Subscribe
emitter.on('foo', function (arg1, arg2) {
   console.log('Foo raised, Args:', arg1, arg2);
});
// Emit
emitter.emit('foo', { a: 123 }, { b: 456 });
```

- As shown in the example, you can create a new instance with a simple `new EventEmitter` call.
- To subscribe to events, you use the `on` function passing in the event name (always a string) followed by an event handling function (also called a listener).
- Finally, we raise an event using the emit function passing in the event name followed by any number of arguments we want passed into the listeners (in snippet above we used two arguments for demonstration).

Multiple Subscribers

- As we mentioned previously, having built-in support for multiple subscribers is one of the advantages of using events. [events/2multiple.js]

```
var EventEmitter = require('events').EventEmitter;
var emitter = new EventEmitter();
emitter.on('foo', function () {
   console.log('subscriber 1');
});
emitter.on('foo', function () {
   console.log('subscriber 2');
});
// Emit
emitter.emit('foo');
```

- Another thing to note in this sample is the fact that the listeners are called in the order that they registered for the event. This is a nice consequence of the single-threaded nature of Node.js, which makes it easier for you to reason about your code. Additionally, any arguments passed in for the event are shared between the various subscribers [events/3shared.js].

Unsubscribing

- The next question to ask is how do we unsubscribe from an event.
- EventEmitter has a removeListener function that takes an event name followed by a function object to remove from the listening queue. [*events/4unsubscribe.js*] :

```
var EventEmitter = require('events').EventEmitter;
var emitter = new EventEmitter();
var fooHandler = function () {
   console.log('handler called');
   // Unsubscribe
   emitter.removeListener('foo',fooHandler);
};
emitter.on('foo', fooHandler);
// Emit twice
emitter.emit('foo');
emitter.emit('foo');
```

- In this sample, we unsubscribe from the event after it is raised once. As a result, the second event goes unnoticed.

Checking if an Event has ever been raised

- It is a common use case that you don't care about every time an event is raised—just that it is raised once.
- For this, EventEmitter provides a function `once` that calls the registered listener only once. [events/5once.js]

```
var EventEmitter = require('events').EventEmitter;
var emitter = new EventEmitter();
emitter.once('foo', function () {
   console.log('foo has been raised');
});
// Emit twice
emitter.emit('foo');
emitter.emit('foo');
```

- The event listener for foo will only be called once.

Managing the Listeners:

- There are a few additional utility functions available on the EventEmitter that you need to be aware of.
- EventEmitter has a member function, listeners, that takes an event name and returns all the listeners subscribed to that event. This can be very useful when you are debugging event listeners. [events/6listeners.js]

```
var EventEmitter = require('events').EventEmitter;
var emitter = new EventEmitter();
emitter.on('foo', function a() { });
emitter.on('foo', function b() { });
console.log(emitter.listeners('foo')); // [ [Function: a], [Function: b]]
```

- EventEmitter instances also raise a `newListener` event whenever a new listener is added and `removeListener` whenever a listener is removed, which can help you out in tricky situations such as when you want to track down the instant an event listener is registered/unregistered.
- It can also be useful for any management you want to do when listeners are added or removed[events/7listenerevents.js].

EventEmitter Memory Leaks

- A common source of memory leaks when working with events is subscribing to events in a callback but forgetting to unsubscribe at the end.
- By default, EventEmitter will tolerate 10 listeners for each event type—anymore and it will print a warning to the console. This warning is specifically for your assistance. All you code will continue to function.
- In other words, more listeners will be added without warning and all listeners will be called when an event is raised, [events/8maxEventListeners.js]

```
var EventEmitter = require('events').EventEmitter;
var emitter = new EventEmitter();
var listenersCalled = 0;
function someCallback() {
   // Add a listener
   emitter.on('foo', function () { listenersCalled++ });
   // return from callback
}
for (var i = 0; i < 20; i++) {
   someCallback();
}
emitter.emit('foo');
console.log('listeners called:', listenersCalled); // 20
```

A common cause of this memory leak is forgetting to unsubscribe for the event when in an error condition of a callback. A simple solution is to create a new event emitter in the callback. This way the event emitter is not shared, and it is disposed along with all of its subscribers when the callback terminates.

Setting the setMaxListeners.

- Finally, there are cases where having more than 10 listeners is a valid scenario. In such cases, you can increase a limit for this warning using the setMaxListeners member function [events/9setMaxListeners.js].

```
var EventEmitter = require('events').EventEmitter;
var emitter = new EventEmitter();
// increase limit to 30
emitter.setMaxListeners(30);
// subscribe 20 times
// No warning will be printed
for (var i = 0; i < 20; i++) {
    emitter.on('foo', function () { });
}
console.log('done');
```

- Note that this increases the limit for all event types on this event emitter. Also, you can pass in 0 to allow an unlimited number of event listeners to be subscribed without warning.
- Node.js tries to be safe by default; memory leaks can weigh heavily when working on a server environment, which is why this warning message exists.

Error Event

An 'error' event is treated as a special exceptional case in Node.js. If there is no listener for it, then the default action is to print a stack trace and exit the program. Listing 5-26 gives a quick sample to demonstrate this. [events/10errorEvent.js]

```
var EventEmitter = require('events').EventEmitter;
var emitter = new EventEmitter();
// Emit an error event
// Since there is no listener for this event the process terminates
emitter.emit('error', new Error('Something horrible happened'));
console.log('this line never executes');
```

If you run this code, you will get an output, as shown in the above listing . You should use an Error object if you ever need to raise an error event, as we did in this example. You can also see from the example that the last line containing the console.log never executes as the process terminated.

Hence raise the error event only for exceptional circumstances that must be handled.

Custom Event Emitters

Now that you are an expert at handling and raising events in Node.js, a lot of open source surface area opens up to you.

A number of libraries export classes that inherit from EventEmitter and, therefore, follow the same event handling mechanism.

At this stage, it is useful for you to know how you can extend EventEmitter and create a public class that has all of the functionality of EventEmitter baked in.

All you need to do to create your own EventEmitter is call the EventEmitter constructor from your class's constructor and use the util.inherits function to set up the prototype chain.

[events/11custom.js]

```
var EventEmitter = require('events').EventEmitter;
var inherits = require('util').inherits;
// Custom class
function Foo() {  EventEmitter.call(this);}
inherits(Foo, EventEmitter);
// Sample member function that raises an event
Foo.prototype.connect = function () {
   this.emit('connected');}
// Usage
var foo = new Foo();
foo.on('connected', function () {
   console.log('connected raised!');
});
foo.connect();
```

A look back into the process global:

A number of classes inside core Node.js inherit from EventEmitter. The global process object is also an instance of EventEmitter.

Don’t trust me run the command: node -e "console.log(process instanceof require('events').EventEmitter)”

Using a single thread for application processing makes Node.js processes more efficient and faster. But most servers have multiple processors, and you can scale your Node.js applications by taking advantage of them.

Node.js allows you to fork work from the main application to separate processes that can then be processed in parallel with each other and the main application.

To facilitate utilizing multiple processes, Node.js provides three specific modules.

1. The **process** module provides access to the running processes.
2. The **child_process** module enables you to create child processes and communicate with them.
3. The **cluster** module provides the ability to implement clustered servers that share the same port, thus allowing multiple requests to be handled simultaneously.

The process module gives you access to the running processes as well as information about the underlying hardware architecture.

Process as a Global Exception Handler

- Any global unhandled exceptions can be intercepted by listening on the `uncaughtException` event on process.
- You should not resume execution outside this event handler because this only happens when your application is in an unstable state.
- The best strategy is to log the error for your convenience and exit the process with an error code[process/1uncaught.js]

```
process.on('uncaughtException', function (err) {
   console.log('Caught exception: ', err);
   console.log('Stack:', err.stack);
   process.exit(1);
});
// Intentionally cause an exception, but don't try/catch it.
nonexistentFunc();
console.log('This line will not run.');
```

- The `uncaughtError` event is also raised on a process if any event emitter raises the `error` event and there are no listeners subscribed to the event emitter for this event.

Exit

- The process module gives you some control over the execution of processes.
- Specifically, it enables you to stop the current process, kill another process, or schedule work to run on the event queue. For example, to exit the current Node.js process, you would use: process.exit(0).

Method	Description
`abort()`	Causes the current Node.js application to emit an `abort` event, exit, and generate a memory core.
`exit([code])`	Causes the current Node.js application to exit and return the specified `code`.
`kill(pid, [signal])`	Causes the operating system to send a kill signal to the process with the specified `pid`. The default `signal` value is `SIGTERM`, but you can specify another.
`nextTick(callback)`	Schedules the `callback` function on the Node.js application's queue.

- The exit event is emitted when the process is about to exit. There is no way to abort exiting at this point. The event loop is already in teardown so you cannot do any async operations at this point.[process/2exit.js]

```
process.on('exit', function (code) {
  console.log('Exiting with code:', code);
});
process.exit(1);
```

- Note that the event callback is passed in the exit code that the process is exiting with.
- This event is mostly useful for debugging and logging purposes.

Signals

Node.js process object also supports the UNIX concept of signals, which is a form of inter-process communication.

It emulates the most important ones on Windows systems as well.

A common scenario that is supported on both Windows and UNIX is when the user tries to interrupt the process using Ctrl+C key combination in the terminal.

By default, Node.js will exit the process. However, if you have a listener subscribed to the `SIGINT` (signal interrupt) event, the listener is called and you can choose if you want to exit the process (process.exit) or continue execution. [process/3signals.js]

```
setTimeout(function () {
   console.log('5 seconds passed. Exiting');
}, 5000);
console.log('Started. Will exit in 5 seconds');

process.on('SIGINT', function () {
   console.log('Got SIGINT. Ignoring.');
});
```

If you execute this example and press Ctrl+C, you will get a message that we are choosing to ignore this. Finally, the process will exit after five seconds naturally once we don’t have any pending tasks

Getting Information from the process Module

The process module has a wealth of information about the running process and the system architecture.

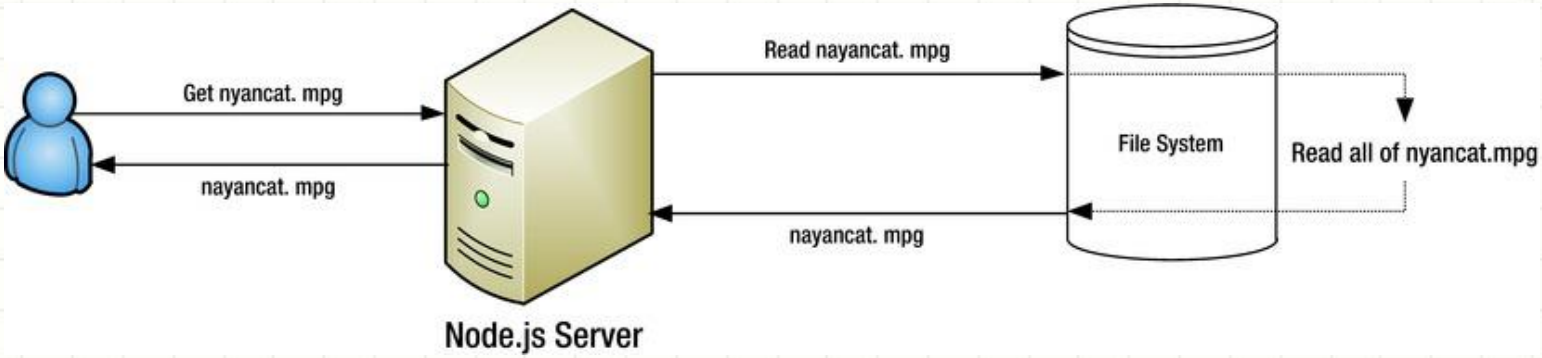
This information can be useful when you're implementing applications.

For example, the process.pid property gives you the process ID that you can then have your application use. [process/process_info.js].

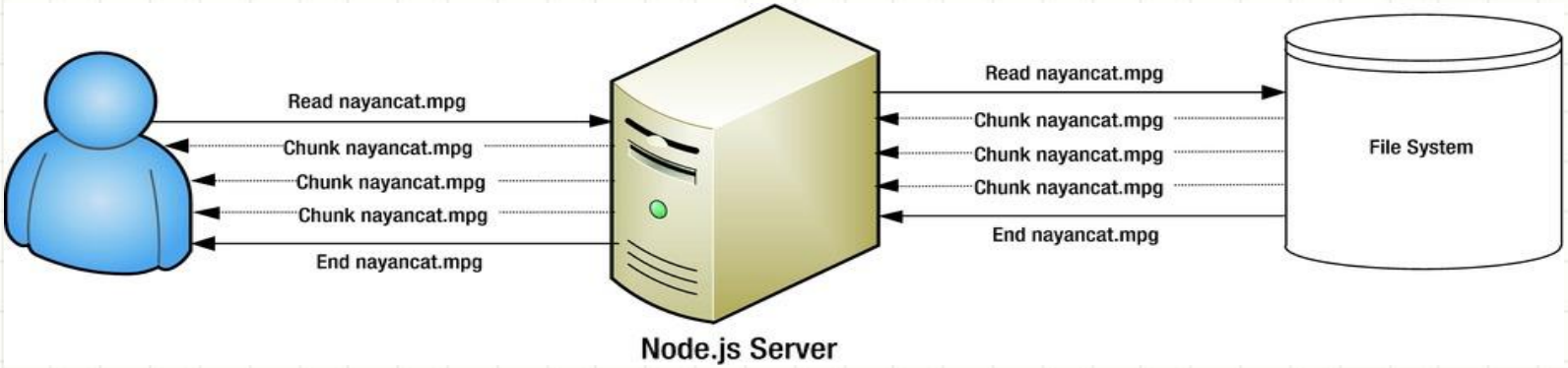
```
console.log('Current directory: ' + process.cwd());
console.log('Environment Settings: ' + JSON.stringify(process.env));
console.log('Node Args: ' + process.argv);
console.log('Execution Path: ' + process.execPath);
console.log('Execution Args: ' + JSON.stringify(process.execArgv));
console.log('Node Version: ' + process.version);
console.log('Module Versions: ' +  JSON.stringify(process.versions));
console.log('Node Config: ' +  JSON.stringify(process.config));
console.log('Process ID: ' + process.pid);
console.log('Process Title: ' + process.title);
console.log('Process Platform: ' + process.platform);
console.log('Process Architecture: ' + process.arch);
console.log('Memory Usage: ' + util.inspect(process.memoryUsage()));
```

Streams

- Streams play an important role in creating performant web applications. To understand what streams bring to the table, consider the simple case of serving a large file (1GB) from a web server. In the absence of streams, it would look like Figure 5-3. The user would have to wait a long time before they get any sign of the file they requested. This is called buffering, and we should try to limit it as much as possible. Besides the obvious bad user experience, it also wastes resources. The complete file needs to be loaded and kept in memory before we start sending it down to the user.



The same scenario looks much better when we use streaming. We start reading the file and whenever we have a new chunk of data, we send it down to the client until we reach the end, as shown in the figure in the next slide.



This improvement in user experience and better utilization of server resources is the main motivation behind steams.

The most important concepts are that of **Readable streams**, **Writable streams, Duplex streams, and Transform streams**.

1. A readable stream is one that you can read data from but not write to. A good example of this is process.stdin, which can be used to stream data from the standard input.
2. A writable stream is one that you can write to but not read from. A good example is process.stdout, which can be used to stream data to the standard output.
3. A duplex stream is one that you can both read from and write to. A good example of this is the network socket. You can write data to the network socket as well as read data from it.
4. A transform stream is a special case of a duplex stream where the output of the stream is in some way computed from the input. These are also called through streams. A good example of these is encryption and compression streams.
5. All of the basic building blocks of streams are present in the Node.js core stream module that you load using require('stream'). There are base classes for implementing streams present in this module, aptly called Readable, Writable, Duplex, and Transform.
6. Streams in Node.js are based on events, which is why it was important to have a firm understanding of events before we could dive into streams. All of these stream classes inherit from a base abstract Stream class (abstract because you should not use it directly), which in turn inherits from EventEmitter (which we saw earlier).[streams/1concepts/eventBased.js]

Pipe

- All the streams support a pipe operation that can be done using the pipe member function. This is one of the things that make streams in Node.js so awesome. Consider our simple initial scenario of loading a file from the file system and streaming it to the client. This can be as simple as a code segment fileSystemStream.pipe(userSocket).
- You can pipe from a stream you can read from (Readable/Duplex/Transform) to a stream you can write to (Writable/Duplex/Transform). This function is called pipe because it mimics the behavior of the command line pipe operator, for example, cat file.txt | grep lol.
- The fs core module provides utility functions to create readable or writable streams from a file. [streams/2pipe/1basic.js] is an example that streams a file from the file system to the user console.
- You can also chain multiple streams using pipe. For example, the code in [streams/2pipe/2chain.js] creates a read stream from a file, pipes it through a zip transform stream, and then pipes it to a writable file stream. This creates a zip file on the file system.
- Streams in Node.js are based on events. All that the pipe operation does is subscribe to the relevant events on the source and call the relevant functions on the destination. For most purposes, pipe is all that you need to know about as an API consumer, but it is worth knowing more details when you want to delve deeper into streams.

Consuming Readable Streams

- We've said it many times already that streams work based on events.
- The most important event for a readable stream is 'readable'.
- This event is raised whenever there is new data to be read from a stream.
- Once inside the event handler, you can call the read function on the stream to read data from the stream. If this is the end of the stream, the read function returns null, as demonstrated [streams/3readable/basic.js]

```
process.stdin.on('readable', function () {
    var buf = process.stdin.read();
    if (buf != null) {
        console.log('Got:');
        process.stdout.write(buf.toString());
    }
    else {
        console.log('Read complete!');
    }
});
```

Writing to Writable Streams

- To write to a stream, you simply call write to write some data. When you have finished writing (end of stream), you simply call end. You can also write some data using the end member function if you want, as [streams/4writable/basic.js]

```
var fs = require('fs');
var ws = fs.createWriteStream('message.txt');
ws.write('foo bar ');
ws.end('bas');
```

- In this sample, we simply wrote foo bar bas to a writable file stream.

Writing you own Streams:

- Creating your own stream is very similar to how you create your own EventEmitter. For streams you inherit from the relevant base, stream class and implement a few base methods. This is detailed in Table 5-1.
- The inheritance mechanism is the same as we have seen before. That is, you call the base constructor from your class constructor and call utils.inherits after declaring you class.

Use-case	Class	Method(s) to Implement
Reading only	Readable	_read
Writing only	Writable	_write
Reading and writing	Duplex	_read, _write
Operate on read data and write the result	Transform	_transform, _flush

Creating a Readable / Writable Stream

- As stated, you simply inherit from Readable class. You implement the _read member in your class, which is called by the stream API internally when someone requests data to be read. If you have data that you want to be passed on (pushed), you call the inherited member function push passing in the data. If you call push(null), this signals the end of the read stream.
- [streams/5createReadable/counter.js]is a simple example of a readable stream that returns 1-1000. If you run this, you will see all these numbers printed (as we pipe to stdout).

- Creating your own writable stream class is similar to how we created a readable stream. You inherit from the Writable class and implement the _write method. The _write method is passed in a chunk that needs processing as its first argument.
- [streams/6createWritable/logger.js]is a simple writable stream that logs to the console all the data passed in. In this example, we simply pipe from the readable file stream to this writeable stream (Logger).

Getting Started With HTTP

- Following are the main core networking modules for creating web applications in Node.js:
  1. net / require('net'): provides the foundation for creating TCP server and clients
  2. dgram / require('dgram'): provides functionality for creating UDP / Datagram sockets
  3. http / require('http'): provides a high-performing foundation for an HTTP stack
  4. https / require('https'): provides an API for creating TLS / SSL clients and servers
- We will start by using the http module to create our simple server to serve static files. Creating our web server from scratch will give us a deeper appreciation of the features provided by the community NPM modules that we will explore later.

Creating our own Webserver:

- Use http module's createServer function which is a great utility function.
- This function takes a callback to run and returns an HTTP server.
- On each client request, the callback is passed in two arguments—the incoming request stream and an outgoing server response stream. To start the returned HTTP server, simply call its listen function passing in the port number you want to listen on.
- [NodeJS/Tutorial_3/CreatingFirstServers/OurFirstServer.js] provides a simple server that listens on port 3000 and simply returns "hello client!" on every HTTP request.
- We will be using curl to test our web applications to start with. It is available by default on Mac OS X / Linux. You can get curl for windows as a part of Cygwin (www.cygwin.com/).

- A lot of Http specific stuff is hidden in the curl call. To see the http headers for example you can log the headers.

Key Members of the Response Stream

- Beyond the fact that the response implements a writable stream, there are a few other useful methods that you need to be aware of.
- The response is split into two sections:
- writing the headers and
- writing the body.
- The reason for this is that the body might potentially contain a large chunk of data that needs streaming. The headers specify how this data is going to be presented and needs to be interpreted by the client before such streaming can begin.
- As soon as you call response.write or response.end, the HTTP headers that you set up are sent, followed by the portion of body that you want written. After this, you cannot modify the headers anymore. At any point, you can check if the headers are sent using the read-only response.headersSent boolean value.
- By default, the status code is going to be 200 OK. As long as the headers are not sent, you can explicitly set the status code using the statusCode response member (for example, to send a 404 NOT FOUND you would use the following code:
- response.statusCode = 404;

Setting Headers

- You can explicitly queue any HTTP header in the response using the response.setHeader(name, value) member function. One common header that you need to set is the Content-Type of the response so that the client knows how to interpret the data the server sends in the body. For example, if you are sending down an HTML file to the client, you should set the Content-Type to text/html, which you can with the following code: response.setHeader("Content-Type", "text/html");

Name	MIME Type
HyperText Markup Language (HTML)	text/html
Cascading Style Sheets (CSS)	text/css
JavaScript	application/javascript
JavaScript Object Notation (JSON)	application/json
JPEG Image	image/jpeg
Portable Network Graphics (PNG)	image/png

- Going back to our headers discussion, you can get a header that's queued for sending using the response.getHeader function:

```
var contentType = response.getHeader('content-type');
```

- You can remove a header from the queue using the response.removeHeader function:

```
response.removeHeader('Content-Encoding');
```

What if you want to send only headers.

- When you want to explicitly send the headers (not just queue them) and move the response into body only mode, you can call the response.writeHead member function. This function takes the status code along with optional headers that will be added on to any headers you might have already queued using response.setHeader. For example, here is a snippet that sets the status code to 200 and sets the Content-Type header for serving HTML:
- response.writeHead(200, { 'Content-Type': 'text/html' });

Key Members of the Request Stream:

The request is also a readable stream. This is useful for cases when the client wants to stream data to the server , for example, file upload. The client HTTP request is also split into a head and body part. We can get useful information about the client request HTTP head.

For example, we have already seen the request.headers property, which is simply a read-only map (JavaScript Object Literal) of header names and values.

Play around with [CreatingFirstServers/OurFirstServerWithHeaders.js]

Another key piece of information that you might be interested in case you are developing a WebServices are request.url and request.method.

Lets create a File Web Server.

- What would we serve: Base HTML
- Lets Create One.
- Next lets create some utility functions.
  - One for the URLs that we donot serve.

```
function send404(response) {
  response.writeHead(404, { 'Content-Type': 'text/plain' });
  response.write('Error 404: Resource not found.');
  response.end();
}
```

If we can serve the request, we should return :

1. HTTP 200
2. A MIME type for the content.

Returning the HTML file is as simple as creating a read file stream and piping it to the response. [NodeJS/Tutorial_3/Server_WebServer/server.js].

Creating a Server to server multiple files:

- In the previous example we created s erver to serve a static html file.
- Now we would try and serve a response which is constitutes of multiple files .
- First off, create a simple JavaScript file that appends to the body after HTML loading is complete. [Server_MutipleFiles/public/main.js]
- We plan to request this JavaScript file from the server.
- Let's modify our simple HTML file by adding a script tag in the <head> to load a client-side JavaScript file
  - <script src="./main.js"></script>
- Now if we run the same old server, we will get a 404 when our browser parses index.html and tries to load main.js from the server. To support this JavaScript loading, we need to do the following:
  1. use the path module to resolve the path to the file on the file system based on the request.url property
  2. see if we have a MIME type registered for the file type requested
  3. make sure the file exists before we try to read it from the file system. [NodeJS/Tutorial_3/Server_MutipleFiles/server.js]

Although we have gained a invaluable knowledge but rarely we would be creating our own webserver since the community has done it for us in the form of connect and express.

Summary

- Define your challenges
  - Technological as well as personal
- Set realistic expectation
  - Mastery is not achieved overnight
- Keep your eye on the goal
  - Mentorship programs

Resources

- <Intranet site text here>
  <hyperlink here>
- <Additional reading material text here>
  <hyperlink here>
- This slide deck and related resources:
  <hyperlink here>

QUESTIONS?

APPENDIX