## Lecture 7: JavaScript on the Server: Node.js CPEN322 - Building Modern Web Applications - Winter 2021-1

#### Karthik Pattabiraman

The University of British Columbia
Department of Electrical and Computer Engineering
Vancouver, Canada





October 12, 2021

## Server-side Javascript



> More events

- Server-side Javascript
- Node.js Modules
- 3 Events
- 4 Files
- Network and Http Server → M:mor AJAX Server -



## History of Server-side JS



- JavaScript evolved primarily on the client-side in the web browser
- However, JavaScript began to be used as a server side language starting in 2008-2009
  - Rhino: JavaScript parser and interpreter written in Java
  - Node.js: V8 JavaScript engine in Chrome (standalone), written in  $\mathsf{C}{+}{+}$

L> Much faster

# Server-Side JS: Advantages



> hard to aigrate programus

across-leans F conomic



Same language for both client and server

- Eases software maintenance tasks
- Eases movement of code from server to client
- Much easier to exchange data between client and server, and between server and NoSQL DBs — MoNhoDB - Support for ISON objects in both
  - Native support for JSON objects in both
- Much more scalable than traditional solutions
  - Due to use of asynchronous methods everywhere
  - · iroditional: multiple through 1 per conn. Slow, consume
- Concurrency: by events- clean model.

# Comparison with Traditional Solutions



- Traditional solutions on the server tend to spawn a new thread for each client request
  - Leads to proliferation of threads
  - No control over thread scheduling
  - Overhead of thread creation and context switches
- Server-side JS: Single-threaded nature of JS makes it easy to write code

  - Scalability achieved by asynchronous calls on do
    Composition with libraries is straightforward on parallel

L's invariants not preservedu

# Node.js Features



- Written in C++ and very fast
- Provides access to low-level UNIX APIs
- Almost all function calls are asynchronous -> Dataut.
  - File systems
  - Network calls
- Module system to manage dependencies
  - Centralized package manager for modules > dependency
- Implements all standared ECMAScript5 constructors, properties, functions and globals

Ly consistent interface. -> each thing has specialized obj



> Write to file system etc.

### Node.js Example



```
1 console.log("Hello"); // Same as before
2 setTimeout(function() { // Same as before
3 console.log("World") }, 1000); -> Not part of window.
4
5 // New stuff - can't do this in client-side JavaScript
6 var fs = require("fs"); // Load file system object
7 var contents = fs.readFilesync( fileName );
8 console.log(contents);
```

```
- Can't access DOM - not in Web browser anymore
```

## Node.js Modules



- Server-side Javascript
- 2 Node.js Modules
- 3 Events
- 4 Files
- Network and Http Server

## Node.js



- In Node.js, you use modules to package functionality together
- Use the module exports keyword to export a function or object as part of a module  $\Rightarrow$  keep red. functionality + encapsul.
- Use the require keyword to import a module and its associated functions or objects
- Choose modules to require then actually included in final Prog. / apple

## **Exporting Functions**



Can be used to create one's own modules

```
Calculator.js

1 function sum(a, b) {
2 return a + b;
3 }
4
5 // This exports the sum function
6 module.exports.sum = sum;
```

```
- Can run on command line, node. Is.
```

# Exporting Objects (Constructors)



 Can also export entire objects through the module.exports – module is optional below

```
Shapes.js

1  var Point = function(x, y) {
2    this.x = x; this.y = y;
3  };
4  
5  module.exports = Point;
```

```
- require shapes. Is. 2 whole thiny.
```

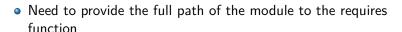
## Using modules: require



 Used to express dependency on a certain module's functionality

#### Points to Note





- Need to check the value of requires. if it's undefined, then module was not found.
- Only functions/objects that are exported using export are visible in the line that calls require

#### **Events**



- Server-side Javascript
- 2 Node.js Modules
- 3 Events
- 4 Files
- **5** Network and Http Server

#### **Event Streams**



- Node.js code can define events and monitor for the occurrence of events on a stream (e.g., network connection, file etc).
- Associate callback functions to events using the 'on()' or 'addListener()' functions
- Trigger by calling the 'emit' function

#### **Event**



- Refer to specific points in the execution
  - Example: exit, before a node process exists
  - Example: data, when data is available on connection
  - Example: end when a connection is closed
- Can be defined by the application and event registers can be added on streams
- Event can be triggered by the streams

```
1 var EventEmitter = require ('events'). EventEmitter;
2 if (! EventEmitter) process.exit(1); > Cuck exists
3 var myEmitter = new EventEmitter(); > const( > stream, can attack
4 var connection = function(id) { /* ... */ }; subscriber
5 var message = function(msg) { /* ... */ };
6 // Add event handlers > just strings made up = sill events
7 myEmitter.on("connection", connection); 7 Attack Subscribers
8 myEmitter.on("message", message);
9 // Emit the events
10 myEmitter.emit("connection", 100); anywhere, will get called
11 myEmitter.emit("message", "hello"); args
```

## Class Activity



Write a function that takes an event stream and an array of strings as arguments, and counts the number of occurrences of each string in the stream. You should use <a href="EventEmitter.on">EventEmitter.on</a> for monitoring the stream, i.e., you should not directly scan the stream for the strings. The function should return a function that prints the count of each string.

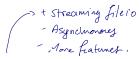
### **Files**



- Server-side Javascript
- 2 Node.js Modules
- 3 Events
- 4 Files
- **(5)** Network and Http Server

# File handling in Node





- Node.js supports two ways to read/write files
  - Asynchronous reads and writes < Preferred >
  - Synchronous reads and writes
- The asynchronous methods require callback functions to be specified and are more scalable 

  register callback when TW.

Synchronous is similar to regular reads and writes in other languages

Ly no context switching, any when done, does callback execute.

## Synchronized Reads and Writes



Least preferred way.

- readFileSync and writeFileSync to read/write files synchronously (operations block JS)
- Not suitable for reading/writing large files
  - Can lead to large performance delays

```
1  var f= fs.readFileSync(fileName);
2  var f = fs.writeFileSync(fineName, data);
```

```
-No callback refd.
```

## Asynchronously reading a file



```
1  var fs = require("fs");  // Filesystem module in node.js
2  var length = 0;
3  var fileName = "sample.txt";
4
5  fs.readFile(fileName, function(err, buf) {
6    if (err) throw err;
7    length = buf.length;
8    console.log("Number of characters read = " + length);
9  });
```

```
- exec asynchronously

- Mechanism.

- Ross err as 1st param.

- No access to org callstack.

Set err to non-zero value (str myrgys)

- Diffo calling context

Buf -> the read file.
```

## Asynchronous Reads using Streams



- if file long, waste time, otherwise calc can be done in Parallel in background.

 It's also possible to start processing a file as and when it is being read. We need to read files as event streams: fs.createReadStream

• Three types of events on files

• data: There's data available to be read

• end: The end of the file was reached

error: There was an error in reading the data

Handler code must store

partial progress



## **Example of Using Streams**



- No callback passed. - Event handler on Stream.

```
var fs = require('fs');
  var length = 0;
   var fileName = "sample.txt";
    var readStream = fs.createReadStream(fileName);;
5
                                                  - cannot ch,
6
7
    readStream.on("data", function(blob) {
        console.log("Read " + blob.length);
                                        > cannot make assumption of blob szl
8
10
11
12
13
14
15
        length += blob.length;
   } );
    readStream.on("end", function() {
       console.log("Total number of chars read = " + length);
    } );
    readStream.on("error", function() {
16
        console.log("Error occurred when reading from file " +
            fileName);
    } );
```

### Asynchronous Writes



• Like reads, writes can also be asynchronous. Just call fs.writeFile with the callback function

```
1  fs.writeFile( fileName, data, function(err) {
2    if (! err)
3        console.log("Finished writing data");
4    else
5        console.log("Error writing to " + fileName);
6  };
```

### Writeable Stream



- Like readStreams, we can define writeStreams and write data to them in blobs
  - Same events as before
  - Useful when combined with readableStreams to avoid buffering in memory
  - Need to call end() when the writing is completed

### Example: Copying one file to another

WriteStream on ( "coror", function()



```
var fs = require("fs");
   var readStream = fs.createReadStream("sample.txt");
4
5
   var writeStream = fs.createWriteStream("sample-copy.txt");
6
   readStream.on("data", function(blob) {
7
9
10
11
12
13
       console.log("Read " + blob.length);
       writeStream . write (blob);
                                              -> No error hardly
   readStream.on("end", function() {
       console.log("End of stream");
       writeStream.end();
   } );
```

## Alternate method: Using Pipe



```
Skip The middleman
```

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

// Open the read and write streams
var readStream = fs.createReadStream("sample.txt");
var writeStream = fs.createWriteStream("sample-copy.txt");

// Copies contents of read stream to write stream
readStream.pipe( writeStream );
```

## Class Activity





- Write a function that searches for a given string in a large text file in node.js. The file should be read using streams and asynchronous I/O, and should not be buffered in memory all at once (as it's too large).
- NOTE: You may get multiple calls to the callback function as file data comes in chunks. Your method must search between chunks.

## Network and Http Server



- Server-side Javascript
- 2 Node.js Modules
- 3 Events
- 4 Files
- Network and Http Server

#### **Network Server**



- Node.js has built in modules for servers
  - 'net' module for general-purpose servers
  - 'http' module for http servers
- To create a http server
  - new http.Server
    - createServer(foo): foo is called when a request arrives, with request & response parameters

### Method 1: Handling Http connections



Accept conn. from port. Main exits, wait for com.

## Method 2: Using Streams

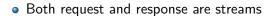


echo Server.

```
var http = require('http');
   // Create a simple function to serve a request
   var serveRequest = function(request, response) {
5
       console log("Received request" + request);
6
7
8
9
10
11
12
13
       response.writeHeader(200, { "Content-type":"text/htm"});
       response.write("Received: " + request.url);
      response.end();
   };
   // Start the server on the port and setup response
   var port = 8080:
   var server = http.createServer();
14
   server.on("request", serveRequest);
15
   server.listen(port);
```

### Inside serveRequest

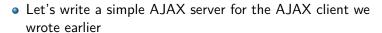




- You can add listeners on both request and response as you do on streams
  - Call end on response when you're done
- Can retrieve the headers and url of request
  - request.url
  - request.headers

### AJAX Server





- If the client requests a JS or html file, serve it from the "./client" directory
- If the client sends a message with the prefix 'hello-', send back a response 'world-' with the same suffix as that of the request
  - Add a delay of 3000 for each request

### AJAX Server - Solution



#### Part 1

```
var serveRequest = function(request, response) {
       if ( request.url.startsWith("/hello") ) {
          // If it's an AJAX request, return world
          console.log( "Received " + request.url );
5
            setTimeout( function() {
6
                var count = request.url.split("-")[1];
                response . write("world-" + count);
8
                response.statusCode = 200;
9
                response.end();
10
            }. 3000); // delay of 3 seconds
11
```



Server-side Javascript

```
1
       else if ( request.url.endsWith(".html") ||
           request.url.endsWith(".js")) {
          // If it's a HTML or JS file, retrieve the
              file in the request
3
          response.statusCode = 200;
4
          var fileName = path + request.url;
5
6
7
          var rs = fs.createReadStream(fileName);
          s.on("error", function(error) {
             console.log(error);
8
             response.write("Unable to read file: " +
                   fileName);
             response.statusCode = 404;
9
10
          });
11
12
13
          rs.on("data", function(data) {
             response.write(data);
          });
14
15
          rs.on("end", function() {
             response.end();
16
          });
17
```

### AJAX Server - Solution



#### Part 3

```
else {
          response.write("Unknown request " + request.
              url);
3
          response.statusCode = 404;
          response . end();
5
6
7
   };
   // Start the server on the port and setup response
   var port = 8080;
10
   var server = http.createServer(serveRequest);
11
   server.listen(port);
12
   console.log("Starting server on port " + port);
```





## Class Activity



 Extend the AJAX server application to log the set of all requests received from the client to a text file. The logging should be done asynchronously and right after the request is received. You should also be able to handle connections from more than 1 client (HINT: Use a separate text file for each client).

### Table of Contents



- Server-side Javascript
- 2 Node.js Modules
- 3 Events
- 4 Files
- Server
  Network and Http Server