

Lecture 7: JavaScript on the Server: Node.js

CPEN400A - Building Modern Web Applications - Winter 2018-1

Karthik Pattabiraman and Julien Gascon-Samson,

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



Electrical and
Computer
Engineering



Tuesday October 30, 2018

Server-side Javascript



2

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

History of Server-side JS



3

- 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 C++

Server-Side JS: Advantages



4

- 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
 - Native support for JSON objects in both
- Much more scalable than traditional solutions
 - Due to use of asynchronous methods everywhere

Comparison with Traditional Solutions



5

- 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
 - Composition with libraries is straightforward

Node.js Features



6

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

Node.js Example



7

```
1 console.log("Hello"); // Same as before
2 setTimeout( function() { // Same as before
3     console.log("World") }, 1000);
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);
```

Node.js Modules



8

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

Node.js



9

- 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
- Use the `require` keyword to import a module and its associated functions or objects

Exporting Functions



10

- 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;
```

Exporting Objects (Constructors)



11

- 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;
```

Using modules: *require*



12

- Used to express dependency on a certain module's functionality

Shapes.js

```
1 // Imports the Calculator module
2 var calculator = require("Calculator.js");
3 calculator.sum(10, 20);
4
5 // Imports the shapes module
6 var Point = require("Shapes.js");
7 var p = new Point(1, 2);
```

Points to Note



13

- Need to provide the full path of the module to the requires function
- 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



14

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

Event Streams



15

- 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



16

- 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);
3 var myEmitter = new EventEmitter();
4 var connection = function(id) { /* ... */ };
5 var message = function(msg) { /* ... */ };
6 // Add event handlers
7 myEmitter.on("connection", connection);
8 myEmitter.on("message", message);
9 // Emit the events
10 myEmitter.emit("connection", 100);
11 myEmitter.emit("message", "hello");
```

Class Activity



17

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 [EventEmitter.on](#) for monitoring the stream, i.e., you should not directly scan the stream for the strings. The function should return a function that retrieves the count of each string.

Files



18

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

File handling in Node



19

- Node.js supports two ways to read/write files
 - Asynchronous reads and writes
 - Synchronous reads and writes
- The asynchronous methods require callback functions to be specified and are more scalable
- Synchronous is similar to regular reads and writes in other languages

Synchronized Reads and Writes



20

- `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);
```

Asynchronously reading a file



21

```
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) {
7          console.log("Error in reading file " + err;
8      } else {
9          length = buf.length;
10         console.log("Number of characters read = " + length);
11     }
12 } );
```

Asynchronous Reads using Streams



22

- Asynchronous reads allow us to overlap computation with reads as long as we don't depend on the data being read
 - But what if we want to start processing a file as it's being read (in chunks or blocks at a time).
 - 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

Example of Using Streams



23

```
1  var fs = require('fs');
2  var length = 0;
3  var fileName = "sample.txt";
4  var readStream = fs.createReadStream(fileName);
5
6  readStream.on("data", function(blob) {
7      console.log("Read " + blob.length);
8      length += blob.length;
9  });
10
11 readStream.on("end", function() {
12     console.log("Total number of chars read = " + length);
13 });
14
15 readStream.on("error", function() {
16     console.log("Error occurred when reading from file " +
17         fileName);
17 });
```

Asynchronous Writes



24

- 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



25

- 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



26

```
1  var fs = require("fs");
2
3  var readStream = fs.createReadStream("sample.txt");
4  var writeStream = fs.createWriteStream("sample-copy.txt");
5
6  readStream.on("data", function(blob) {
7      console.log("Read " + blob.length);
8      writeStream.write(blob);
9  } );
10
11 readStream.on("end", function() {
12     console.log("End of stream");
13     writeStream.end();
14 } );
```

Alternate method: Using Pipe



27

```
1 var fs = require("fs");
2
3 // Open the read and write streams
4 var readStream = fs.createReadStream("sample.txt");
5 var writeStream = fs.createWriteStream("sample-copy.txt");
6
7 // Copies contents of read stream to write stream
8 readStream.pipe( writeStream );
```

Class Activity



28

- 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



29

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

Network Server



30

- 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



31

```
1  var http = require('http');
2
3  // Create a simple function to serve a request
4  var serveRequest = function(request, response) {
5      console.log( request.headers );
6      response.write("Welcome to node.js");
7      response.end();
8  };
9
10 // Start the server on the port and setup response
11 var port = 8080;
12 var server = http.createServer(serveRequest);
13 server.listen(port);
```

Method 2: Using Streams



32

```
1  var http = require('http');
2
3  // Create a simple function to serve a request
4  var serveRequest = function(request, response) {
5      console.log("Received request " + request);
6      response.writeHead(200, { "Content-type": "text/htm" });
7      response.write("Received: " + request.url);
8      response.end();
9  };
10
11 // Start the server on the port and setup response
12 var port = 8080;
13 var server = http.createServer();
14 server.on("request", serveRequest);
15 server.listen(port);
```

Inside `serveRequest`



33

- Both request and response are streams
- 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



34

- Let's write a simple AJAX server for the AJAX client we wrote earlier
- 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



35

Part 1

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

AJAX Server - Solution



35

Part 2

```
1     else if ( request.url.endsWith(".html") ||  
2         request.url.endsWith(".js")) {  
3         // If it's a HTML or JS file, retrieve the  
4         // file in the request  
5         response.statusCode = 200;  
6         var fileName = path + request.url;  
7         var rs = fs.createReadStream(fileName);  
8         rs.on("error", function(error) {  
9             console.log(error);  
10            response.write("Unable to read file : " +  
11                fileName);  
12            response.statusCode = 404;  
13        });  
14        rs.on("data", function(data) {  
15            response.write(data);  
16        });  
17        rs.on("end", function() {  
18            response.end();  
19        });  
20    }  
21 }
```

AJAX Server - Solution



35

Part 3

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

Class Activity



36

- 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



37

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