

# JavaScript on the Server: Node.js

CPEN 400A – Lecture 8

(Based on “JavaScript: The Definitive Guide” by David Flanagan, and

# Outline

- **Server-side JavaScript**
- Node.js modules
- Events
- Files
- Network and Http

# 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 C++

# Server-Side JS: Advantages

- 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

- 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

- 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

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

```
// New stuff – can't do this in client-side JavaScript  
var fs = require("fs"); // Load file system object  
var contents = fs.readFileSync( fileName );  
console.log(contents);
```

# Outline

- Server-side JavaScript
- **Node.js modules**
- Events
- 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
- Use the *require* keyword to import a module and its associated functions or objects

# Exporting functions

Can be used to create one's own modules

## **Calculator.js:**

```
function sum(a, b) {  
    return a + b;  
}
```

```
// This exports the sum function  
module.exports.sum = sum;
```

# Exporting Objects (Constructors)

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

## **Shapes.js**

```
var Point = function(x, y) {  
    this.x = x; this.y = y;  
};
```

```
module.exports = Point;
```

# Using modules: require

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

```
// Imports the Calculator module  
var calculator = require("Calculator.js");  
calculator.sum(10, 20);
```

```
// Imports the shapes module  
var Point = require("Shapes.js");  
var p = new Point(1, 2);
```

# Points to Note

- 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

# Outline

- Server-side JavaScript
- Node.js modules
- **Events**
- Files
- Network and Http Server

# Events

- 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



# Example of 'on' and emit

```
var EventEmitter = require('events').EventEmitter;
if (! EventEmitter) process.exit(1);
var myEmitter = new EventEmitter();
var connection = function(id) { ... };
var message = function(msg) { ... };
// Add event handlers
myEmitter.on("connection", connection);
myEmitter.on("message", message);
// Emit the events
myEmitter.emit("connection", 100);
myEmitter.emit("message", "hello");
```

# addListener and removeListener

- It's possible to add event listeners using 'addListener' too – exactly the same as 'on'
  - `emitter.addListener(name, f);`
- Event handlers can also be removed (much like in the case of DOM handlers)
  - `emitter.removeListener(name, f);`
- Can remove all handlers associated with an event using the `removeAllListeners` functions
  - `emitter.removeAllListeners(name);`

# Outline

- Server-side JavaScript
- Node.js modules
- Events
- **Files**
- Network and Http Server

# File handling in Node

- 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

# Asynchronously reading a file

```
var fs = require("fs");    // Filesystem module in node.js
var length = 0;
var fileName = "sample.txt";

fs.readFile(fileName, function(err, buf) {
    if (err) throw err;
    length = buf.length;;
    console.log("Number of characters read = "
                + length);
});
```

# Asynchronous Reads using Streams

- 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

# Example of Using Streams

```
var fs = require('fs');
var length = 0;
var fileName = "sample.txt";
var readStream = fs.createReadStream(fileName);

readStream.on("data", function(blob) {
    console.log("Read " + blob.length);
    length += blob.length;
});

readStream.on("end", function() {
    console.log("Total number of chars read = " + length);
});

readStream.on("error", function() {
    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

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



# 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

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

var readStream = fs.createReadStream("sample.txt");
var writeStream = fs.createWriteStream("sample-copy.txt");

readStream.on("data", function(blob) {
    console.log("Read " + blob.length);
    writeStream.write(blob);
} );

readStream.on("end", function() {
    console.log("End of stream");
    writeStream.end();
} );
```

# Alternate method: Using Pipe

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

```
readStream = fs.createReadStream("sample.txt");
```

```
var writeStream = fs.createWriteStream("sample-  
copy.txt");
```

```
// Copies contents of read stream to write stream
```

```
readStream.pipe( writeStream );
```

# Synchronized Reads and Writes

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

```
var f= fs.readFileSync(fileName);
```

```
var f = fs.writeFileSync(fineName, data);
```

# 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.

# Outline

- Server-side JavaScript
- Node.js modules
- Events
- 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

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

// Create a simple function to serve a request
var serveRequest = function(request, response) {
  console.log( request.headers );
  response.write("Welcome to node.js");
  response.end();
};

// Start the server on the port and setup response
var port = 8080;
var server = http.createServer(serveRequest);
server.listen(port);
```



# Method 2: Using Streams

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

// Create a simple function to serve a request
var serveRequest = function(request, response) {
  console.log("Received request " + request);
  response.write("Received: " + request.url);
  response.end();
};

// Start the server on the port and setup response
var port = 8080;
var server = http.createServer();
server.on("request", serveRequest);
server.listen(port);
```

# Inside `serveRequest`

- 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

- 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 1: AJAX Messages

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

# AJAX-Server 2: File Reading

```
else if ( request.url.endsWith(".html") || request.url.endsWith(".js")) {  
    // If it's a HTML or JS file, retrieve the file in the request  
    response.statusCode = 200;  
    var fileName = path + request.url;  
    var rs = fs.createReadStream(fileName);  
    rs.on("error", function(error) {  
        console.log(error);  
        response.write("Unable to read file : " + fileName);  
        response.statusCode = 404;  
    });  
    rs.on("data", function(data) {  
        response.write(data);  
    });  
    rs.on("end", function() {  
        response.end();  
    });  
}
```

# AJAX Server-3: Rest of the code

```
} else {  
    response.write("Unknown request " + request.url);  
    response.statusCode = 404;  
    response.end();  
}  
};
```

```
// Start the server on the port and setup response  
var port = 8080;  
var server = http.createServer(serveRequest);  
server.listen(port);  
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).

# Outline

- Server-side JavaScript
- Node.js modules
- Events
- Files
- Network and Http Server