NodeJS: Server-side JS

Building Modern Web Applications - VSP2019

Karthik Pattabiraman
Kumseok Jung

Server-side JavaScript

1. **Server-side JavaScript**
2. Node.js Modules
3. Events
4. Files

Server-side JS: History

- 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 ECMAScript constructors, properties, functions and globals

Node.js Example

```
1  console.log("Hello");   // console object is available in Node.js
2  setTimeout(() => {      // setTimeout is available
3     console.log("World");
4  }, 1000);
5
6  // Available in Node.js (can't do this in client-side JS)
7  var fs = require("fs"); // require is a Node.js built-in function
8  var content = fs.readFileSync("hello.txt");
9  console.log(content);
10
```

Node.js Modules

1. Server-side JavaScript
2. **Node.js Modules**
3. Events
4. Files

Node.js

- In Node.js, you use **modules to package functionality** together
- Use the `module.exports` built-in object to export a function or object as part of a module
- Use the `require` built-in function 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;
};

// module is a special built-in object in Node.js
// module.exports object can be used to expose an API
module.exports.sum = sum;
```

Exporting Objects

- Can also export entire objects through the `module.exports`


```
1  // Point.js
2  var Point = function(x, y){
3     this.x = x;
4     this.y = y;
5  };
6
7  // module.exports is initially equivalent to {}.
8  // We replace the entire object with the Point function
9  module.exports = Point;
10
```

Using modules: `require`

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

```
1  // Imports the Calculator module
2  var calculator = require("Calculator.js");
3  calculator.sum(10, 20);
4
5  // Imports the Point module
6  var Point = require("Point.js");
7  var p = new Point(1, 2);
8
9
10
```

Points to Note

- Need to provide the **full path of the module** to the `require` function
- Need to check the return value of `require`. If it's undefined, then the module was not found.
- Only functions/objects that are exported using `export` are visible in the line that calls `require`

Events

1. Server-side JavaScript
2. Node.js Modules
3. **Events**
4. Files

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

Event

```
1  // import the EventEmitter constructor from built-in events library
2  var EventEmitter = require("events").EventEmitter;
3
4  // create an EventEmitter object
5  var myEmitter = new EventEmitter();
6  var onConnection = function(id){ /* ... some code */ };
7  var onMessage = function(msg){ /* ... some code */ };
8
9  // attach event listeners
10 myEmitter.on("connection", onConnection);
11 myEmitter.on("message", onMessage);
12
13 // emit events (somewhere else in the code)
14 myEmitter.emit("connection", 100);
15 myEmitter.emit("message", "hello");
```

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 sent through the stream. Tip: you should use `EventEmitter.on` for monitoring the stream. The function should return a function that prints the count of each string.
- For testing your code:
  - You can use the text in file sample.txt. However, we haven't covered streams yet – this'll be done in the next section.
  - To read the contents of file sample.txt:
    ```
    var text = fs.readFileSync("sample.txt").toString();
    ```
  - To get an array of words: `var words = text.split(" ");`

Files

1. Server-side JavaScript
2. Node.js Modules
3. Events
4. **Files**

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

Synchronous 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

```
1  // synchronous file read
2  var f = fs.readFileSync("sample.txt");
3
4  // synchronous file write
5  var f = fs.writeFileSync("sample.txt", "Hello World!");
6
7
8
9
10
```

Asynchronous Reads and Writes

```
1  var fs = require("fs");       // import built-in fs library
2  var length = 0;               // for keeping track of # characters
3  var fileName = "sample.txt";
4
5  // asynchronous file read takes in a callback
6  fs.readFile(fileName, function(err, buf){
7     if (err) throw err;
8     length = buf.length;
9     console.log("# of Characters = " + length);
10 })
```

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

```
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 # of Characters = " + length);
13 });
14
15 readStream.on("error", function(err){
16    console.log("Error occurred trying to read " + fileName);
17 });
```

Asynchronous Writes

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

```
fs.writeFile("example.txt", "Hello World", function(err){
   if (err)
      console.log("Error writing to example.txt");
   else
      console.log("Finished writing data");
});
```

Writeable Stream

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

Example: Copying one file to another

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

Alternate method: Using Pipe

```
1  var fs = require("fs");
2  var readStream = fs.createReadStream("example.txt");
3  var writeStream = fs.createWriteStream("example-copy.txt");
4
5  readStream.pipe(writeStream);
6
7
8
9
10
```

Class Activity

- Write a Node.JS script 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.
- Also, you may not make any assumptions regarding the size of the chunk (except that it is non-zero). Your script should work for all chunk sizes.

Class Activity

- In the previous slide, the solution (solution.js) wasn't 100% perfect as it assumed the size of the word to search to be lower than the Node.js Read buffer size. How can you improve it to avoid this problem?
- **Assumption**: the search string shouldn't have a prefix that is a substring of itself (e.g., hehello)