NODE.JS EXECUTION MODEL

Stephen Schaub

Topics

- Node.js Execution Model
- Node.js Stream API

Concurrency Models

Two concurrency programming models:

- Threads + Synchronous methods
- Asynchronous methods

Node.js Execution Model

- A Node.js program operates on a single thread
- Most I/O functions are asynchronous (nonblocking)
  - They start the I/O operation and return immediately
  - When the I/O operation completes, the results are passed to a callback function
- Example:

```
var fs = require("fs");
fs.readFile('/etc/passwd', function (err, data) {
    console.log(data);
});
```

Callback Functions


```
fs.readFile('/etc/passwd', function (err, data) {
    console.log(data);
});
```

Node.js convention:

- Callback functions are the final argument in asynchronous method calls

Control Flow in an Async World

Synchronous Flow

```
data1 = fs.readFileSync(filename1, 'utf8');
console.log(data1);

data2 = fs.readFileSync(filename2, 'utf8');
console.log(data2);

data3 = fs.readFileSync(filename3, 'utf8');
console.log(data3);

// do more stuff ...
```

Asynchronous Flow

```
fs.readFile(filename1, 'utf8', function(err, data) {
  console.log(data);
  fs.readFile(filename2, 'utf8', function(err, data) {
    console.log(data);
    fs.readFile(filename3, 'utf8', function(err, data) {
      console.log(data);
      // do more stuff...
    });
  });
});
```

Error Handling in an Async World


```
fs.readFile('/etc/passwd', function (err, data) {
  if (err) { console.log("oopsies:", err); }
  else { console.log(data); }
});
```

- Many callback functions receive an Error object as the first parameter
- Must explicitly test error parameter and deal with error
- **Note:** Throwing exceptions is not an appropriate way to handle callback errors in Node.js
  - Terminates the application

Incorrect Error Handling


```
try {
   fs.readFile('/etc/passwd', function (err, data) {
     if (err) { throw(err); }
     console.log(data);
   });
} catch (err) {
  // attempt to deal with error
}
```

- This does not work (why)?

Streams and Events

Reading Files, Revisited

- We've seen how to read files with fs.readFile

  ```
  fs.readFile('/etc/passwd', function (err, data) { … })
  ```

- What if the file is big?
  - Don't want to read it all into memory at once
- We need Streams

Streams

- A Stream is a source or sink for data
- Many Node.js modules use the Stream API to provide/consume data
  - fs
  - http

Streams and Events

- Stream API is built on Node's Event API
- ReadStream events:
  - **data** event triggered each time data arrives from the stream
  - **end** event triggered when end of stream has been reached
  - **error** event raised on problem reading data
- The **data** event
  - Occurs multiple times
  - Passes the received data as a Buffer object (call .toString() to get a string)
- Either the **end** or **error** event occurs exactly once
  - By default, **end** and **error** close the stream

Read a File Using Stream API

- Step 1: Create the ReadStream

```
var fs = require('fs');
var rdstrm = fs.createReadStream("mydata.txt");
```

- Step 2: Use **on()** to register callbacks for the data, end, and error events

```
rdstrm.on("data", function(buffer) {
    console.log(buffer.toString());
});
rdstrm.on("end", function() {
    console.log("\n\nThat's all, folks!");
});
rdstrm.on("error", function(err) {      // Note: If you do not register a handler for error, errors raise exceptions
    console.log("Oopsies:", err);       // that will terminate your program
});
```

Stream API is a "Fluent" API

- Alternate coding style:

```
var fs = require('fs');
fs.createReadStream("mydata.txt").on("data", function(data) {
    process.stdout.write(data);
}).on("end", function() {
    console.log("\n\nThat's all, folks!");
}).on("error", function(err) {
    console.log("Oopsies:", err);
});
```

Writing HTTP Clients

- The http module presents a stream-oriented API

```
var http = require("http");
http.get("http://localhost:8000/", function (response) {
    response.setEncoding("utf8");  // makes data event emit strings instead of buffers
    response.on("data", function (data) {
        console.log(data);
    });
    response.on("end", function () {
        console.log();
    });
});
```

Writing HTTP Clients

- We can get the entire response in one variable like this:

```
var http = require("http");
http.get("http://localhost:8000/", function (response) {
    response.setEncoding("utf8");
    let body = "";
    response.on("data", function (data) {
        body += data;
    });
    response.on("end", function () {
        console.log(body);
    });
});
```

- See complete example with error handling: http_request.js

HTTP Client Case Study

- https://github.com/palmerabollo/node-isbn
  - https://www.npmjs.com/package/node-isbn
  - Version: https://github.com/palmerabollo/node-isbn/blob/56ea185ca1418d4b4710829dbcf28edd0636f473/index.js

HTTP Clients

- It would be nice if http offered a function we could call like this:
  - http.download("http://localhost:8080/", function(err, data) {
    // data
    });
- It doesn't, so let's design our own

Designing Asynchronous Functions

Designing Asynchronous Functions

- Consider the following blocking function:

```
function loadBooks(filename) {
  var books = JSON.parse(fs.readFileSync(filename, 'utf8'));
  return books;
}
```

- Uses a blocking I/O API (readFileSync)
- What if we want loadBooks() to use readFile() instead of readFileSync()?

Reworking loadBooks()

- Consider substituting readFile() in place of readFileSync():

```
function loadBooks(filename) {
  var books = JSON.parse(fs.readFile(filename, 'utf8', function(err, data) {  }));
  return books;
}
```

- We can't continue down this path
  - Why not?
- Key idea: Functions that call asynchronous functions must themselves be asynchronous
  - We must do a fundamental redesign

Async Function Design

- First, redesign loadBooks() to receive a callback function:

```
function loadBooks(filename, callback) {
  …
}
```

- It will be called this way:

```
loadBooks("books.json", function(err, books) {
   // do stuff with books
});
```

In Class Exercise

- Implement loadBooks using async function fs.readFile()

Synchronous vs. Asynchronous Functions I

- Synchronous functions return results to the caller

```
function loadBooks(filename) {
    var books = JSON.parse(fs.readFileSync(filename, 'utf8'));
    return books;
}
```

- Asynchronous functions pass results to a callback

```
function loadBooks(filename, callback) {
    fs.readFile(filename, 'utf8', function(err, data) {
        var books = JSON.parse(data);
        callback(null, books);
    });
}
```

Error Handling in Async Functions

- Async functions must pass errors to their callback and then return

```
function loadBooks(filename, callback) {
   fs.readFile(filename, 'utf8', function(err, data) {
      if (err) {
         callback(err);
         return;
      }
      var books = JSON.parse(data);
      callback(null, books);
   });
}
```

Error Handling in Async Functions

- Async functions must catch any errors from synchronous functions and pass them to the callback

```
function loadBooks(filename, callback) {
   fs.readFile(filename, 'utf8', function(err, data) {
      var books;
      if (err) {
         callback(err);
         return;
      }
      try {
         books = JSON.parse(data);
      } catch (err) {
         callback(err);
      }
      callback(null, books);
   });
}
```

Synchronous vs. Asynchronous Functions II

- Synchronous functions
  - Return results to the caller
  - Throw exceptions to report problems
- Asynchronous functions
  - Pass results to a callback
  - Report errors to callback
  - Never throw exceptions

Summary

Key observations:

- An asynchronous function receives a callback function and does not return a value
- Functions that call asynchronous functions must themselves be asynchronous

Now, Your Turn

- Write a downloadHttp function with this API:
  - httpDownload("http://localhost:8080/", function(err, data) {
    // data
    });