Node: Advanced Topics

Chapters 1-5 in Professional Node.js Textbook

Direction

In the coming weeks we will:

- Make our web-server more powerful using Express to provide robust MVC support
- Learn to connect to a database and work with data more effectively
- Use hashing and HTTP authentication to create secure applications

We are going to need to learn a bit more about the way Node.js works before all this though - otherwise it will be very confusing!

Tasks, Processes, and Threads

- Each instruction a CPU executes can essentially be categorized into two classes: CPU-bound, I/O-bound
- When you make a series of calculations, the CPU performs them - and it does so amazingly quickly.
- When you read a byte from disk, the CPU issues a command to the device (disk) controller.
    - This operation is amazingly slow in comparison.

Tasks, Process, and Threads

- I/O-bound calls are not limited to accessing files on disk - they include:
  - Reading/Writing data from/to a socket (sound familiar?)
  - Waiting for a keyboard or mouse event

- These are more commonly referred to as "**blocking calls**" - meaning the CPU is blocked, waiting for a device to complete.

Tasks, Processes, and Threads

- Moreover, we can generalize the concept of “blocking calls” to any “**long-running task**” that would prevent further execution.
- Classifying a task as “long running” is subjective - but some heavy computation tasks like encryption could be considered as such…
- Often the completion of a blocking task is called an “**event**”.

Tasks, Processes, and Threads

So imagine you have the following set of tasks to perform:

1: CPU	x = y + z;
2: Blocking	read 8 bytes from disk, store in w
3: Blocking	read 8 bytes from socket, store in s
4: CPU	y *= 1000;

If this were a single program, instruction 3 could not be started until instruction 2 was complete.

But note - instruction 2 and 3 are independent (and so is 4!). **Why wait?**

Multi-threaded

If you know about threads, you are probably thinking.... threads.

Threads allow parts of your program to execute in parallel (or nearly in parallel)

Thread 1:

start threads 2 and 3

x = y + z; (CPU)

y *= 1000;

Thread 2:

Read 8 bytes from disk and store in w

Thread 3:

Read 8 bytes from socket and sto
in s

Fantastic - now nothing is held up and the total time equals the maximum run time of thread 1, 2, or 3 - not the sum!

Multi-threaded web server

A web server's workflow is like this:

1. Read HTTP Request from Socket (Blocking)
2. Compute a "model" (probably CPU, with some blocking sprinkled in, such as DB access)
3. Read template (ejs) from disk (blocking)
4. Transform Model + Template into HTML (CPU)
5. Write HTTP Response to socket (Blocking)

Notice - these steps need to happen IN ORDER - for a given request….
…. So there is no opportunity to multi-thread… yet...

At scale...

But when you think about many HTTP requests arriving to the same server - you can treat the task of "serving request" as a "long running task"

Push each request out to its own thread, and you achieve parallelism (between requests)

For about 30 years, this was considered "best-practice". Now... that is changing.

Problems with multi-threaded

Multi-threaded in general is great - however

1. Can be difficult to program
2. Threads cost memory and time - they are expensive to keep creating and tearing down.
3. When a single request contains blocking and CPU sub-tasks (which it will), the 1 thread per request model doesn’t achieve maximum parallelism
   - The developer could of course use multi-threading *within* the request, but see #1.

Node's solution

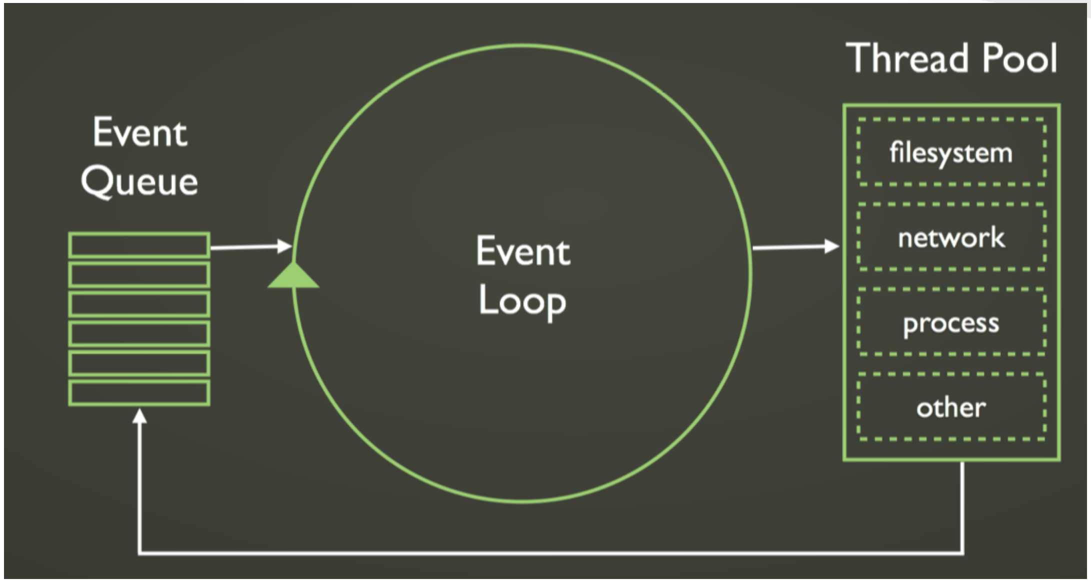
The unique aspect of Node is not JavaScript, or V8 - its how it solves the blocking problem… by not allowing you to block!

- Node programs are **never** multithreaded, from the user's perspective.
- Instead, each blocking call is automatically dispatched into *essentially* a new thread. Thus your blocking calls don't actually block!

Pooling Requests

- Node maintains a thread pool, where one thread is dedicated to each resource (disk, socket, etc.).
    - These are automatically started for you
    - Your code executes in the main thread
- Each time you make a blocking call, you provide a **callback** function
- When the blocking call is completed, node will call your callback function, typically with some parameters.

Node event loop



Your code runs until the last statement is executed, and there are no more outstanding events in the queue

http://www.slideshare.net/JeffKunkle/nodejs-explained

- The power of this model is that **all** threads involved are started immediately, there is no overhead to making blocking calls.
- It frees the developers from needing to deal with concurrency, since they cannot write multi-threaded code themselves.
- **Note - this isn't for all types of problems, but its fantastic for a web server.**

Comparing C++ and Node

C++

```
ifstream in ("file.txt");
int x;
in >> x;  blocking… takes milliseconds
cout << x;
cout << "Bye" << endl;
```

In C++, "Bye" **always** appears after the contents of x…

Node.js

```
fs = require('fs')
fs.readFile('file.txt', 'utf8',
  function (err,data) {
    if (err) {
      return console.log(err);
    }
    console.log(data);
});
console.log("Bye");
```

In Node.js, "Bye" will very likely appear first!

readFile returns IMMEDIATELY
The read from disk happens asynchronously, in a separate thread
The callback executes when read completes

Node.js callbacks

Most node callback functions are called with two parameters: **err** and **result**

- The **err** object contains data if there was a problem.
- It should be of type "Error", and will be if you are using standard libraries

```
var error = new Error("The error message");
console.log(error);
console.log(error.stack);
```

The **result** object could be anything... depending on what you are doing.

Node.js callback

```
fs.readFile('file.txt', 'utf8',
  function (err,data) {
    if (err) {
      return console.log(err);
    }
    console.log(data);
});
```

While common, you don't need to inline all your callbacks!

```
function handleData (err,data) {
    if (err) {
      return console.log(err);
    }
    console.log(data);
}
fs.readFile('file.txt', 'utf8', handleData);
```

The asynchronous pattern

For beginners, this style of programming can be very confusing - and difficult to master.

However - it forces your program to be “implemented” in the most parallel way possible, which is exceptional in server environments

And... threading is complicated, even when you **have** mastered it.... so there is a pay off in complexity too.

Simulating with setTimeout

You can play around with this callback concept just using the timeout facility - your callback will be called after a specified number of milliseconds

```
setTimeout(
    function () {
        console.log ("After 1000ms")
    }, 1000);


console.log("Here");



setTimeout(
    function () {
        setTimeout ( function () {
            console.log("After 2000ms")
        }, 1000);
    }, 1000);
```

```
setTimeout(
    function () {
        console.log ("After 1000ms A")
    }, 1000);

setTimeout(
    function () {
        console.log ("After 1000ms B")
    }, 1000);
```

Problems with this...

Fine - but what if you have:

Task A - blocking

Task B - CPU, but **depends** on task A

Do task B in the callback of task A

What about:

Task A - blocking

Task B - blocking

Task C - CPU, depends on A and B?

Your code can get very messy…
Do task B in callback of A
Do task C in callback of B

Synchronous problems in an asynchronous world...

Perhaps you can actually change your program so tasks A and B don't depend on each other.

If you can, great. Your program may run faster!

Lots of time you can't though

A Pull filename from database of records

B Read contents of file

C Print the file to the screen.

Scaling…

The nesting of many callbacks in series is called the “boomerang effect” and should be avoided

```
db.query(query, function(error, users) {
    if (!error) {
        db.query(query, function(error, posts) {
            if (!error) {
                db.query(query, function(error, comments) {
                    if (!error) {
                        console.log(comments);
                    } else {
                        // Handle error
                    }
                } else {
                    // Handle error
                }
            }
        } else {
            // Handle error
        }
    } else {
        // Handle error
    }
});
```

3 database queries in series (presumably they each depend on the previous…

Imagine how bad this could get!

async modules

There are actually dozens of great libraries to help with execution flow - all available through **npm**

Chapter 19 introduces the async module

```
async = require('async');
function taskA(next) {
    // do task A
    next();
}
```

```
function taskB(next) {
    // do task B
    next();
}
```

```
function done() {
    // do something else
}
```

```
async.series([taskA, taskB], done);
```

There are async functions for executing in series, in parallel, and many hybrid workflows as well

Another excellent tools is the [promise] library

event emitters

Often you are confronted with ***extremely* long running tasks**.

- Reading a 10GB file
- Reading 100,000 rows from a database

- Having a callback called once, with ALL the data would be crazy
  - Wasteful of memory, and would be very unresponsive!

Event Emitters

- Many objects instead implement an emitter pattern, where the object fires events over and over again.
- This patterns works for one-time events as well

```
var stream = fs.createReadStream('sample.txt');

function handleData(data) {

        console.log(data);

}

stream.on('data', handleData);

stream.on('end', console.log("End"));
```

Read streams read chunks of data (likely complete disk blocks).

Then **'data'** event is fired in success.

The **'end'** event is fired once all chunks are read.

Chapter 5 in the "Professional Node.js" text book

Event Emitters

We've actually already seen this pattern when we read post data directly off the HTTP request object (before we learned about bodyParser middleware)

```
function process_post(req, res) {
    var body = "";
    req.on('data', function (chunk) {
        body += chunk;
    });
    req.on('end', function() {
        qs = require('querystring');
        var response = "<html><body><h1>Posted data</h1>";
        var post =  qs.parse(body);
        for ( q in post ) {
            console.log(q + " -> " + post[q]);
            response += ("<p> " + q + "->" + post[q] + "</p>");
        }
        response += "</body></html>";
        res.end(response);
    });
}
```

Taken from 05_form_processing

And of course... the exception

There is one notable exception to the non-blocking idea - **require**

require does a few things:

1. reads a file (javascript file) from disk
2. executes the code in the file as if it were a javascript program
3. returns an object filled with "exports"

Because of they way it is used, require **is** blocking, when it returns, you can be sure all the steps have been completed.

Learning more

There is much more to learn - we will learn as we go, starting with looking at Express, then Authentication, and finally Databases.

- Its a very good idea to read at least through chapter 6 in the Professional Node.js text book **now**
- I also recommend Node.js In Action (by Cantelon, Harter, Holowaychuk, and Rajlich) if you are looking for another book.