Getting Started with Node

Tools

Tools you need:

- NodeJS – basically a Google V8 Javascript engine
  - available from nodejs.org, easy install
- Npm – the Node Package Manager
  - Available from https://www.npmjs.com/
  - Helps you install and manage 3rd party modules

Optional tools you don't need but might like:

- Nvm – the Node Version Manager
  - Available via npm install –g nvm, or by download from nodejs.org
- Bower - "front-end package manager" – you shouldn't need
- Gulp – build system. Kinda like ant, but you write Javascript tasks in a "gulpfile" and then run.
- Grunt – ditto, just goes about its business differently

What about an IDE?

- Yes, you can use Eclipse, get the nodeclipse plugin
- Or there are several good JS editors and run node on the CL

First example:

Hello World:

```
console.log("Hello World!");
$ node hello.js
```

Um yeah…maybe a little more interesting:

```
/* From theprojectspot.com */
var http = require('http');
```

Load module, like "import" (kinda)

Create the http server, and on each request execute the callback

```
http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
}).listen(8080);
```

What port (and optionally hostname) we listen on

```
console.log('Server started');
```

Well that was less code & setup than a servlet → Stay tuned…

Functional Programming

A review of SER221:

- In functional programming, you encapsulate *behaviors* not *state*
- Unlike procedural programming, there are *no side-effects*
- It has seen a bit of resurgence lately
    - Languages include Haskell, LISP, Clojure, F#
    - Javascript and Scala walk the line between functional and OO
    - Java8 even added lambda expressions to move this way

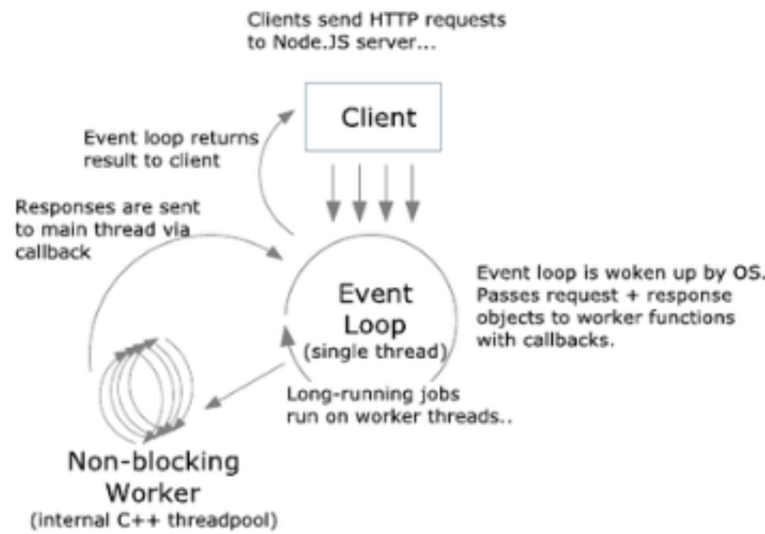
And this matters for web development because?

- The functional approach means we layer *functional* application semantics over our request/response synchronous HTTP
    - In other words, we model a single request/response as a pipelined process of discrete stateless computational tasks
- Compared to OO (like Java), we avoid the overhead of object construction/deconstruction and impedance mismatch

Asynchronous Processing

In the conceptual overview, we talked about the asynchronous single-threaded execution model

- Single-threaded
- Non-blocking
- Use of callbacks



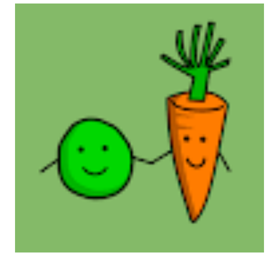
Pros and Cons of the Asynchronous Model

Pros:	*Cons:*
1. No "synchronized" blocks	1. Awkward coding model at first
2. Event loop is efficient	2. Threads natural extension of process
3. No context switch	3. Not natural to OO

It is more natural to think in terms of a functional, task-oriented paradigm, where each distinct computation is executed off the "queue"

Functional & Asynchronous Programming

…these go together like…



The event queue is important

- It is that constant line of work (think treadmill) that the single thread keeps chewing through

How do you manipulate it?

- The natural flow is to provide a callback
  - From a functional design perspective, you have to make sure work is broken into discrete chunks
    - Your function, on completion, puts more work on the queue
  - From an asynchronous perspective, you have to realize you are not sequential, but you are not concurrent either

There are other ways of adding work to the queue:

1. Make a blocking I/O call that ends up on a worker thread
2. Use an event listener for a built-in event
3. Emit your own event and write custom listeners
4. Use process.nextTick as a scheduling mechanism
5. Use explicit timers to schedule work

Explicit Timers

3 types of timers:

1. 1-shot timeout: setTimeout
2. Periodic: setInterval
3. Immediate: setImmediate – after I/O callbacks but before “regular” work

```
// simpleblock.js
// Example from Ch. 2 of Mixu's Node Book, http://book.mixu.net/node/ch2.html
var myTimer = setTimeout(function() {
    console.log('Timeout at ' + new Date().toTimeString());
}, 500);
myTimer.unref();  // if timers are all that is on the event queue, exit
// Change the function between setTimeout, setInterval, and setImmediate
// start time
var sTime = new Date();
console.log('Started app processing loop at ' + sTime.toString());

// delay block
var i = 0;
while (new Date().getTime() < sTime.getTime() + 2500) { i++ }
console.log('Exiting processing loop at ' + new Date().toTimeString() + ' after
   ' +i+ ' iterations');
```

Process.nextTick()

This gets your work on the event queue before blocking I/O

- But you could starve I/O if you are not careful
- It is slightly different from setImmediate in that it will invoke its callback before even going back to the event queue.
- Technically what it does is insert itself as the 1st thing done after the current event loop completes by placing its work on the *next tick queue*
- Most common usage is to ensure truly async behavior

- Example: simplenexttick.js

```
console.log('start');
process.nextTick(() => {
  console.log('nextTick callback');
});
console.log('scheduled');
function hello() {
     console.log('hello world');
}
hello();
process.nextTick(hello);
```

Output:

```
start
scheduled
hello world
nextTick callback
hello world
```

NextTick example

Here is a nexttick brain teaser – what is the output?

```
// nexttick.js - what will be the order of the output?
setTimeout(function timeout() {
  console.log('TIMEOUT FIRED');
}, 0)
process.nextTick(function () {
     process.nextTick(function () {
       console.log(1);
       process.nextTick(function () { console.log(2); });
       process.nextTick(function () { console.log(3); });
     });
     process.nextTick(function () {
       console.log(4);
       process.nextTick(function () { console.log(5); });
       process.nextTick(function () { console.log(6); });
     });
   });
```

Event listeners

```
// modified from http://www.hacksparrow.com/node-js-eventemitter-tutorial.html
// see simpleEventListener.js
var EventEmitter = require('events').EventEmitter;
var radium = new EventEmitter();

function cb(ray) {
    console.log(ray);
}
// on and addListener are the same thing
radium.on('radiation', cb);
radium.on('foo', function(ray) {
   console.log("Boo hoo I will never get called");
});
radium.once('radiation', function(ray) {
   console.log("JUST ONCE " + ray);
});
setTimeout(function() {
   radium.removeListener('radiation', cb);
}, 5000);
setInterval(function() {
    radium.emit('radiation', 'GAMMA');
}, 500);
```

Set callbacks
- One for radiation
- One that is never called

- Another for radiation that will only get called once

Make a 1-shot timeout to remove our listener

Make an interval time out that generates events

More on events

Often we will listen on events that come from built-in node modules, like in module http.

In the previous example we created our own event generator, and *emitted* events

- At the top we new'd up an EventEmitter

To do this in general, you modify the prototype of your object so it can emit events directly:

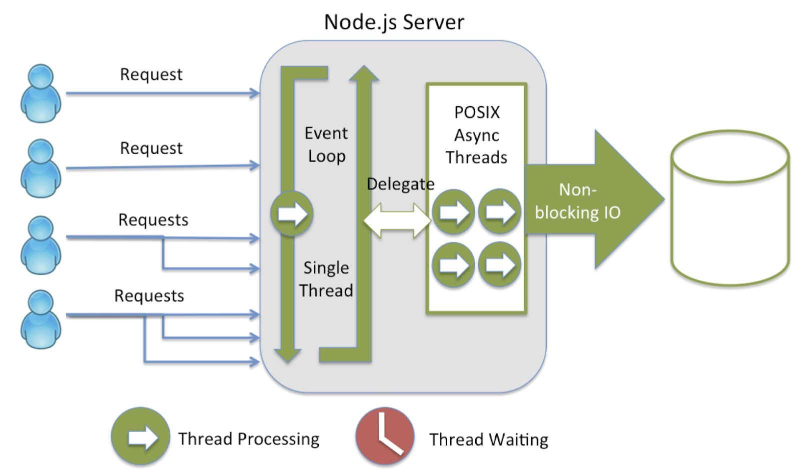
```
function myObj() { /* do your constructor work here */ }
myObj.prototype.__proto__ =
    require('events').EventEmitter.prototype;
// Now you can just do this:
myObj.emit('myEvent');
```

See emitter_listener.js

Blocking I/O and the event queue

Node realizes that the most common thing a program will do that blocks is I/O

- Traditionally, we read in data from a file or write data out to a file and simply wait for the operation to complete
- If the file I/O is a long-running operation, we either tell the user to wait, or we get cute and put it in a thread
- Recognizing the danger of blocking I/O, and that it is about the only thing it knows would block ahead of time, node knows it needs to farm it off to a worker thread and provide a callback
- The 5th way you can manipulate the event queue
- Also note that because it is special, this is why we have things like process.nextTick that pre-empt I/O calls on the queue



[A]Synchronous File I/O

You can do File I/O synchronously, which basically looks like every other programming language's simple file I/O libraries.

- Some distinctions in asynch file operations
  - Exceptions are caught by node and returned as a 1st param err object
  - Asynch functions require extra param for a callback
  - Synchronous file I/O starts right away before any subsequent events are processed by node; whereas asynch file I/O puts the next behavior on the event queue for deferred execution.

Functions: (`var fs = require('fs')`)

- Opening and Closing files:
  - `fs.open[Sync](path, flags [,mode] {,callback})` – standard file open call. Path is either relative or absolute, flags determines, read, write, append, etc. Mode is the file access mode (default rw) and callback is for async only.
  - `fs.close[Sync](fd {,callback})` – closes the file described by file descriptor fd. Closing flushes any buffers

[A]Synchronous File I/O

More file I/O Operations:

- `writeFile[Synch](path, data, [options], callback)`
  - data can be String or a Buffer object
  - writes the entire file in one swoop (*See file_write.js*)
- `readFile[Synch]`
  - reads entire file *(See file_read.js)*

Note when we read/write an entire file we may have a long-running operation. To keep with the principle of short discrete computational tasks, we can read/write bytes at a time:

- `write[read][Synch](fd, data, offset, length, position, callback)` – reads bytes either into data (Buffer) or writes bytes to it (String or Buffer) *(See file_read[write]_[a]sync)*
- You can also get file metadata. *(See file_stats.js, file_readdir.js, file_folders.js)*

[A]Synchronous File I/O

Streaming I/O

- A stream is an I/O abstraction that
  - Is defined by its I/O properties not by its physical implementation
  - Works well in a *pipeline* way of thinking about your I/O
- Node provides the Stream module to working with streaming I/O
  - Emits events when data is written or read
  - We saw this already with the low-level http module – requests and responses are instances of streams (example: see http_server_post.js)

A 4th way to read/write from files is to treat them as Streams

- `fs.createReadStream(path [,options])`
- `fs.createWriteStream(path [,options])`
  - Options is just an object that has configuration information as before, like encoding and access mode, etc.)
  - Your code should listen for the ‘data’, ‘end’ and ‘err’ events in its event listeners (“on” and “end” methods)
  - See file_write_stream.js and file_read_stream.js

Intro to NodeJS summary

We are just getting started!

- Embrace the functional programming paradigm
- Embrace the asynchronous model – it takes a while for it to feel natural, so you have to practice!

Node is a modular, functional, asynchronous platform

- We will be using modules like http and events often
  - We may even write a few of our own – very easy!
- Functional – thing of your tasks, not your objects
- Asynchronous – lots of callbacks, you get used to “chaining” your callbacks to get that semi-sequential feel
  - But realize other functions may try to manipulate the queue
- 5 ways to manipulate the asynchronous event queue
- Keep your tasks small, discrete, and well-defined!
  - Node wants to run the treadmill at the highest speed!