Week 2 Key Concepts:

1. The JavaScript Engine/Interpreter
2. Node & Event-Driven Programming
3. Asynchrony: managing the *now* and *later*
4. Middleware & Node Globals
5. Databases: Storing data, reading data and manipulating data

Resources

(In order of recommendation)

1. How The Event Loop Works
2. Callbacks Explained
3. How The V8 Engine Works
4. Node Beginner Book
5. Effective JavaScript
6. Principles of Object Oriented JavaScript
7. JavaScript The Good Parts
8. JavaScript The Definitive Guide

Key Concept References

1 - Introducing The JavaScript Engine/Interpreter:

Last week you learned about the DOM and about how the Browser parses and translates your HTML into a tree-like representation that it traverses recursively in order to respond to client queries. You then put what you learned with selector.js to work in order to build Game Of Life. This week, you started with node-shell and since **we haven't officially covered the JavaScript Engine yet, we'd like you to have a general idea about how it works.** Recall from 'How Browsers Work' that the Browser has 7 components:

1. **The User Interface:** includes every part of the browser display except the window that presents the requested page
2. **The Browser Engine:** guides interactions between the UI and the rendering engine
3. **The Rendering Engine:** displays requested content by parsing HTML and CSS and rendering the parsed content on the screen
4. **Networking:** configures calls like HTTP requests
5. **UI Backend:** draws widgets like combo boxes and windows
6. **JavaScript Engine/Interpreter:** parses and executes JavaScript code
7. **Data Storage:** persists data via local storage and cookies

The JavaScript Engine performs three main functions:

1. **Tokenizes and Lexes:** involves breaking up a string of characters into chunks, called tokens, that are meaningful to JavaScript. For example, the line **var x = 10;** will be broken up into the tokens: **var, x, =, 10,** and **;**
   Note: There is a subtle distinction between tokenizing and lexing but we won't get into that here as that would require you to understand the nuances of **statelessness** and **statefulness**.

2. **Parsing:** takes that stream of tokens, and turns it into a tree of nested elements that, together, represent the grammatical structure of a program. This tree is called an 'AST' (**A**bstract **S**yntax **T**ree)

3. **Code Generation:** the process of taking an AST and turning it into executable code. It turns the AST for **var x = 10;** into a set of machine instructions to actually *create* a variable called **x** (including reserving memory, etc), and then store a value into **x**.

   The JavaScript Interpreter performs these functions mere microseconds (or less) before executing your scripts. Modern browsers use a technology known as Just-In-Time (JIT) compilation, which compiles JavaScript to executable bytecode just as it is about to run.

   Note: this explanation simplifies what the engine does in order to provide a basic overview.

Let's take an example

When V8 enters the execution context of bar in the following code, a few steps will occur.

```
var foo = 'bar';
function bar() {
    var foo = 'baz';
}
function baz(foo) {
    foo = 'bam';
    bam = 'yay';
}
```

1. Lexing/Tokenizing:
   a. The compiler starts by lexing (or tokenizing) the script, which means it splits your code into atomic tokens like `foo` `=` or `bar`.
2. Parsing:
   a. The compiler then uses these tokens to create an Abstract Syntax Tree (AST) of the entire script.
   b. The tree for var foo = 'bar' might begin with a top-level node called VariableDeclaration, which a child node called Identifier (the value of which is bar), and another child called AssignmentExpression, which has a child called StringLiteral (the value of which is 'foo'.

c. It then creates a map of all of the scopes in the script. Each time the compiler encounters a variable or function declaration, it attaches that declaration to its scope. With regards to this script, the compiler notes five things: 1. The existence of a **var foo** declaration that is in the **global scope.** 2. The **bar function declaration**, which is also in the **global scope.** 3. The **var foo** declaration in the **scope bar**. 4. The **baz function declaration** in the **global scope.** 5. It notes the **foo parameter declaration** and that it is in the **scope baz.** The named parameter is treated as a local variable.

3. Code Generation:
    a. The compiler takes the AST and transforms it into executable code. An explanation of this process would be a dissertation so let's just take a moment to appreciate the wonder of JavaScript engines in general and V8 in particular.

2 - Node & Event-Driven Programming

Node.js is part runtime environment and part library for building network applications using server-side JavaScript. It uses Chrome's JavaScript runtime engine to execute JavaScript without the browser sandbox.

EVENT DRIVEN:

At JavaScript's core is an asynchronous event model that enables its single-threaded design. Defined simply: event-driven programming is application flow control that is determined by events or changes in state. The general implementation is to have a central mechanism that listens for events and invokes a callback function once an event has been detected (i.e. state has changed). Sound familiar? It should. That's the basic principle behind Node's event loop. Once you grasp callbacks, this event model will be freeing, as opposed to confusing, because it means that your code is **uninterruptible** and that the events you schedule will be fired in order.

CALLBACKS

When we want to make a piece of code run in the future in JavaScript (on an event, when a button is clicked, after the user mouses over a form, etc), we place it in a callback. A callback is just like any other function. Again: any function can be a callback. What makes it a callback is not *how* it's defined but where it is invoked/how it is fired. So a callback is a normal function that is passed to a function like **setTimeout** as a parameter or bound as a property like **document.onready**. When a callback runs, the event (e.g. the timeout elapsing or the document becoming ready) has fired.

Check out this gist
Try it out in repl: http://repl.it/8bO/1

You probably expected the **first** loop to console.log 1, 2, 3.
To understand why the output is 4, 4, 4 instead, there are three things you need to keep in mind:

1. There's only one variable named **i**, scoped by the declaration **var i** (which scopes it not within the loop but within the closest function containing the loop - remember, in JavaScript, scope is delimited by functions; the only mechanism other than a function that provides scope is the

try-catch block, the only block scoping tool in ECMAScript 5, though ECMAScript 6 equips us with the powerful block scoping tool, **let**)

2. After the loop, i === 4 because it was incremented until it failed the condition i <= 3;
3. JavaScript event handlers do not run until the thread is free.

Check out this gist on Immediately Invoked Function Expressions (IIFEs), closures and for-loops to learn more.

BLOCKING THE THREAD

To fully appreciate JavaScript's single-threaded nature, study this code:

```
var start = new Date;
setTimeout(function(){
        var end = new Date;
        console.log('Time elapsed:', end - start, 'ms');
}, 500);
while (new Date - start < 1000) {
   console.log('hello!')
}
```

check it out in the repl

In a multithreaded context, only 500ms would pass before the JavaScript engine executed setTimeout's callback function. This would mean that the loop would be momentarily interrupted/paused. However, if you run the code, you'll get something like this:

Time elapsed: 1002 ms

You may see a different number. That's because setTimeout and setInterval are far less precise than anyone would like. But it will definitely be at least 1000 ms, because the setTimeout callback CANNOT fire until the WHILE loop has finished running. Why can't it? It can't because JavaScript is single-threaded.

So if setTimeout isn't using another thread to execute its callback function, how does it know to execute the callback once the thread is free?

MEET THE QUEUE

When the JavaScript Engine executes setTimeout, a timeout event is *queued,* that is, put on the queue. Once it's placed on the queue, the execution continues: the line after setTimeout call runs, and then the line after that, and so on, until there are no lines left. Only once the JavaScript Engine has executed every other ***synchronous*** line does it return to the queue.

If there's at least one event on the queue that's eligible to 'fire' (like a 500ms timeout that was set 1000ms ago), the Engine will pick one and call its handler/callback (e.g. the function passed to setTimeout). When the callback returns, the Engine returns to the queue.

3 - Asynchrony: managing the *now* and *later*

As software engineers, we are tasked with figuring out how to express program behavior that occurs over a period of time. Most non-trivial programs have at least two parts: one part that runs *now* and another part that runs *later* - there's a gap between the *now* and *later* during which the program isn't actively executing. The ability to deftly manage this gap between *now* and *later* separates the novice JavaScript developers from the adepts.

Every time we make an AJAX request, part of our program executes *now*, and part of it executes *later*.

If our program were not spread out over time, we could simply write:

```
// ajax(..) is some arbitrary third-party AJAX function
var data = ajax('http://some.url.1')
```

Standard AJAX requests do not complete synchronously, which is why we don't write AJAX calls this way. Instead, we do:

```
ajax('http://some.url.1', function callback(data) {
   console.log(data)
}
```

The callback is the simplest (but not the only nor the best) way of getting from *now* (the AJAX request) to *later* (receiving the requested data).

Note: it is indeed possible to make synchronous AJAX requests but we should never ever do this. Synchronous requests lock the browser UI (buttons, menus, scrolling, etc) and prevent all user interaction.

Just to hammer in the *now* and *later* aspect, consider this code:

```
function now() {
   return 21;
}

function later() {
   answer = answer * 2;
   console.log(('Why are we here: ', answer);
}
```

```
var answer = now();
setTimeout(later, 1000)  // why are we here: 42
```

This program has two chunks: the stuff that will run *now* and the stuff that will run *later*. Check out this gist to get the complete breakdown.
Any time we wrap a portion of code into a function and then specify that it should be executed in response to some event (a mouse click, a timer, AJAX response, etc), we create a *later* chunk of our code, thereby introducing asynchrony into our program.

THE EVENT LOOP, REVISITED

Despite the fact that you've written several lines of asynchronous JavaScript code (the setTimeout we just looked at, for example), asynchrony is not codified into the JavaScript language itself (that is, not until ES6). That can't be true, can it? It is. The JavaScript engine does no more than execute a single chunk of your program at any given moment, when instructed to.

Notice those last three words: 'when instructed to'. By what? That's the critical part: The JavaScript engine does not run in isolation; it runs inside of a *hosting environment*, which is the browser for most JavaScript developers, but of course we know JavaScript can run on the server via Node.js and in robots (e.g. tessel.io). All environments that employ JavaScript have a mechanism that instructs the JavaScript engine to execute multiple chunks of code *over time*. This orchestrating mechanism is known as the 'event loop'.

The event loop schedules events and thereby tells the JavaScript engine when to execute which lines of code. Again: Pre-ES6 JavaScript has no innate sense of time. So, in order to make an AJAX request to fetch data from the server, we set up a callback function to handle the response. When the JavaScript engine gets to the AJAX request in our script, it sends off the request (the *now* portion of our code), and then it tells the hosting environment, 'I'm going to suspend execution of this function for now, but whenever that network request finishes and you have data, please *call* this function *back*.' The browser is configured to listen for the response from the network and when it receives the response data, it schedules the callback function to be executed by inserting it into the *event loop*.
So, what is this event loop? Here's some pseudocode that captures its main functionality:

```
// `eventQueue` is an array that acts as a queue (first-in, first-out)
var eventQueue = [ ];
var event;
// the event loop keeps going "forever"
while (true) {
   // perform a "tick"
   if (eventQueue.length > 0) {
      // get the next event in the queue
      event = eventQueue.shift();
      // now, execute the next event
      try {
         event();
```

```
        }
        catch (err) {
          reportError(err);
        }
    }
}
```

The while loop represents the continuously running nature of the event loop, the main thread of the browser. The event loop is an infinite loop that keeps the browser process alive. It waits for events and enqueues and dequeues them. Each iteration of this loop is referred to as a 'tick'. For each 'tick', if an event is waiting on the queue, it's taken off and executed. These events are your callback functions.

Note: setTimeout(..) does not put your callback on the event loop queue. It sets up a timer; when that timer expires, the *environment* places your callback into the event loop, such that some future tick will pick it up and execute it. What if there are already 20 items in the event loop when your setTimeout timer hits t=0? Your callback gets in line behind the others; callbacks can't skip the line. This is why setTimeout(...) timers do not always fire with perfect temporal accuracy. You're only guaranteed that your callback won't fire before the specified time interval, but it can fire at any time after that, depending on the state of the event queue.

ASYNC vs. PARALLEL

Software developers commonly conflate the terms "async" and "parallel" when discussing JavaScript but they are actually quite different. **Async is about the gap between now and later. Parallel is about things being able to occur simultaneously**.

The most common tools for parallel computing are processes and threads. Most browsers use several parallel threads to execute network operations. Processes and threads execute independently and may execute simultaneously: on separate processors, or even on separate computers, but multiple threads can share the memory of a single process.

By contrast, the event loop breaks its work into tasks and executes them serially, disallowing parallel access and changes to shared memory. As a single-threaded environment, the event loop is not a parallel process. Often when someone uses the word 'parallel' to describe asynchronous JavaScript, he or she is talking about **concurrency.**

CONCURRENCY (the following UI example is provided by Kyle Simpson)

Concurrency is when two or more "processes" are executing simultaneously over the same period regardless of whether their component operations occur in parallel (at the same instant on separate processors or cores) or not. You can think of concurrency then as "process"-level (or task-level) parallelism, as opposed to operation-level parallelism (separate-processor threads).

Imagine a site that displays a list of status updates (like a social network news feed) that progressively loads as the user scrolls down the list. To make such a feature work correctly, (at least) two separate task-level "processes" will need to be executing simultaneously (by simultaneous we mean *during the same window of time*, NOT at the same instant, because, again, the event loop is single-threaded: it cannot rub its belly and pat its head at the same time, as you surely can).

The first "process" will respond to onscroll events (making AJAX requests for new content) as they fire while the user is scrolling down the page. The second "process" will receive AJAX responses (to render content onto the page)

Process 1 (onscroll events):
onscroll, request 1
onscroll, request 2
onscroll, request 3
onscroll, request 4
onscroll, request 5
onscroll, request 6
onscroll, request 7

Process 2 (AJAX response events):
response 1
response 2
response 3
response 4
response 5
response 6
response 7

It's quite possible that an onscroll event and an AJAX response event could be ready to be processed at exactly the same moment. For example let's visualize these events in a timeline:

Process 1	Process 2
onscroll, request 1	
onscroll, request 2	response 1
onscroll, request 3	response 2
response 3	
onscroll, request 4	
onscroll, request 5	
onscroll, request 6	response 4
onscroll, request 7	
response 6	
response 5	
response 7	

The JavaScript engine can handle only one event at a time, so either onscroll, request 2 is going to happen first or response 1 is going to happen first, but they cannot happen at literally the same moment.

Let's visualize the interleaving of all these events onto the event loop queue.

Event Loop Queue:
onscroll, request 1 <--- Process 1 starts
onscroll, request 2
response 1 <--- Process 2 starts
onscroll, request 3
response 2
response 3
onscroll, request 4
onscroll, request 5
onscroll, request 6
response 4
onscroll, request 7 <--- Process 1 finishes
response 6
response 5
response 7 <--- Process 2 finishes

“Process” 1 and “Process” 2 run concurrently (task-level parallel), but their individual events run sequentially on the event loop queue.

Note: response 6 and response 5 came back out of the expected order. Callbacks do not enable us to control the order of events invoked by the event queue. ES6 promises and generators will afford us this control.

Main Takeaways:

1. The single-threaded event loop is one expression of concurrency
2. A JavaScript program is (practically) always broken up into at least two chunks, one that runs *now* (e.g. a button click) and one that runs *later* (the rendering of the form that is the response to that button click).
3. Even though JavaScript programs are executed chunk-by-chunk, all of them share the same access to the program scope and state, so each modification to state is made on top of the previous state.
4. Whenever there are events to run, the event loop runs until the queue is empty. Each iteration of the event loop is a “tick.” User interaction, IO, and timers enqueue events on the event queue.

5. At any given moment, only one event can be processed from the queue at a time. While an event is executing, it can directly or indirectly cause one or more subsequent events.
6. Concurrency is when two or more chains of events interleave over time, such that from a high-level perspective, they appear to be running simultaneously (even though at any given moment only one event is being processed).
7. It's often necessary to do some form of interaction coordination between these concurrent "processes" to ensure their proper ordering (i.e. displaying response 5 before we display response 6).
8. Callbacks may not be the best way to handle interaction coordination (more on this in next week's concept review).

Next week's concept review will delve into the nature of 'callback hell'. We'll explore why callbacks are insufficient to the task of managing concurrency and introduce the concepts that underlie other tools that manage asynchrony more gracefully (promises & generators). To get a head start, check out these gists: callback hell & the pain of coordinating concurrency

4 - Middleware & Node Globals

What Is Middleware?

Each line/statement above the routes in app.js is middleware:

```
// the functionality imported here is used in the function calls below
var favicon = require('static-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
// invocations of the middleware stack
app.use(favicon());
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));
```

Middleware includes pass-through functions that either do something useful or add something helpful to the request as it travels to its endpoint. These functions are invoked by the Express routing layer before the final request handler is invoked in order to augment or change the request object so that the proper response object can be returned. For example, bodyParser() and cookieParser() add HTTP request payload (req.body) and parsed cookie data (req.cookie), respectively.

A single layer in the middleware stack is simply a function that takes three parameters: 1) the REQ object, 2) the RES object, and 3) the NEXT callback. We refer to middleware as a stack since each layer is invoked in the order that it is added.

FIVE BUILT-IN NODE GLOBALS TO KNOW ABOUT:

1. The Process Object:
    a. provides us with actions that can be taken on the current node process as well as information about the node process. The process object is available in any of your node programs and provides interesting properties about the process itself and some methods for you to control the process.
    b. includes information about the machine it runs on (i.e. environment variables, OS architecture, the OS platform)
    c. allows for passing of arguments to the process (we begin our node programs/processes from the terminal so we can use the process objects to receive arguments when our programs are started
    d. process.nextTick: a function defined on the process object that takes a callback. It is used to put the callback into the next cycle of the Node.js event loop. It is designed to be highly efficient, and it is used by a number of Node.js core libraries.
2. __dirname / __filename:
    a. these globals are strings that represent the name or directory name of the current file
    b. wherever the code that uses these variables is written - that's the file we're talking about
3. The module object and the require object (explained below)
4. console: similar to your browser's console object function. In Node, the output is sent to stdout and stderr. The definition of console.log in Node:
5. console.log = function(d){
6. process.stdout.write(d + '\n');
7. };
8. global: somewhat analogous to the window object (all of the other Node globals are members to the global variable). In your browser the top-level property is the globe scope which is why it's easy to create (intentionally and unintentionally) global variables. In Node, when you declare a global variable in a module (file), that variable is local to that module. In order to declare a variable that is global throughout the running process, you must set a property on the global object. Of course, this is discouraged.

    Learn about Node's other globals

MODULE.EXPORTS & REQUIRE

The node ecosystem is full of functionality commonly referred to as modules. In order to use this functionality we need import it into our scripts. We must import any and all functionality that we

want that is not defined in the file in which we want to use it. In the node world, this process of importing functionality is achieved through the **require** keyword.

So, for example, we have a file called file1.js:

File1.js:

```
var hello = function(){
  console.log('hello world')
}
```

To use this hello functionality in file2, we have to do two things:

1. In file1.js, we have to **export** this functionality by attaching the hello function to the **module.exports object**.
2. In file2.js, we have to **import** this functionality by **requiring** file1.js. By requiring file1.js in file2.js, we import any functionality that file1.js attached to the module.exports object.

File1.js:

```
var hello = function(){
  console.log('hello world')
}

//COMMON EXPORTING METHOD 1:
module.exports = {hello: hello}

//COMMON EXPORTING METHOD 2:
module.exports.hello = function(){
  console.log('hello world')
}
```

File2.js:

```
var import = require('./file1.js')
import.hello()                  // 'hello world'
```

To learn more about **require** and **module.exports**, check out this gist

5 - Databases

ACID

A. Atomicity - all or nothing
   - A set of database operations that must occur together. Atomicity requires that each transaction be "all or nothing": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes. To the outside world, a committed transaction appears (by its effects on the database) to be indivisible ("atomic"), and an aborted transaction does not happen.

C. Consistency - must bring us to a valid state
   - If a process has a writer, no other process can read from it, and no other process can write to it, which gives us consistent information at all times. The consistency property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including **constraints, cascades, triggers**, and any combination thereof. This does not guarantee correctness of the transaction in all ways the application programmer might have wanted (that is the responsibility of application-level code) but merely that any programming errors cannot result in the violation of any defined rules.

I. Isolation - concurrent execution that mirror serial execution results
   - Multiple clients can make queries to read and update without the risk of deadlock or starvation. The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed **serially**, i.e., one after the other. Providing isolation is the main goal of concurrency control. Depending on concurrency control method, the effects of an incomplete transaction might not even be visible to another transaction.

D. Durability - transactions persist
   - Only finishes transaction if it is finally done (rule to rollback if shut off). Durability means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter). To defend against power loss, transactions (or their effects) must be recorded in a **non-volatile memory**.

RDBMS - relational database management system

The relational model (based on Set Theory) standardizes storage formatting. Meaning that we have a databases which are a collection of tables (relations) which have columns (attributes) and rows (instances or tuples), with each instance being unique.

To communicate with the database we have a simple, structured query language - SQL. With SQL programmers can specify what answers a query should return, but not how the query is executed. This declarative language is standardized for all of your applications! And our RDMS picks an efficient

execution strategy based on indexes, data, workload, etc. We don't need to implement this, just use the syntax given to us - INSERT, SELECT, UPDATE, DELETE, CREATE, INNER JOIN, etc.

PostgreSQL

The open-source, powerful, popular relational database management system we use. It has deep SQL standard compliance with its SQL dialect and focuses on data integrity with ACID reliability. Unlike sqlite3 which is embedded, PostgreSQL is remote; it can be on a different server (machine), if on our machine it is a background process listening on port 5432 by default - a daemon.

We access PostgreSQL through a CLI (command line interface), a GUI (such as Postico) or from an application with the use of a database driver such as `pg` - the node-postgres Library. A database driver is basically a library that knows how to connect with and format communication between an application and the database; `pg`, specifically, communicates through the postgres protocol and the content of messages consist of SQL.

ORM - Object Relational Mapping

An ORM tool is a tool that converts data between incompatible type systems. This creates a virtual object database that can be used from within our programming environment - JS. The theme of the Third Manifesto is how to avoid the object-relational impedance mismatch, in which we have a relation composed of tables and instances in contrast to JavaScript objects. In lieu of having a database and query language that handle this, we utilize an ORM tool, specifically Sequelize

Sequelize

A high level interface for various SQL dialects - postgres included - that we can use from our JavaScript, Node program. It represents "tables" as "models" and "rows" as "instances".

MODELS: Sequelize models, once synced, correspond to a particular table in our PostgreSQL database. Sequelize models *enhance* tables by representing a class with not only class methods and hooks, but also validations, instance methods, and getter methods.
INSTANCES: An instance is a specific row in a particular model. They represent a row by holding the row's data as key/value pairs. Instances come with ready-made methods such as .save, .update, and .destroy. They also will have any methods defined (by us) as instance methods on their containing model.

Order of Events

- **Sequelize** - ORM
  - Reads the JS code `Users.findAll( )` =>
  - Constructs the string SQL (postgres dialect) query `SELECT * FROM users` =>
  - Passes that SQL query to the JS library `pg` =>
- **pg** - protocol
  - Connects via TCP/IP to PostgreSQL =>
  - Uses the postgres protocol to tell PostgreSQL it has an incoming SQL query =>
  - Sends the SQL query to PostgreSQL =>

- **PostgreSQL** - dialect, database
    - Parses the query =>
    - Reads the data on disk =>
    - Sends a response back to `pg` via the postgres protocol on the TCP connection =>
- **pg** - protocol
    - Receives raw string data =>
    - Turns raw string into an array of row objects `[{id: 1, name: “Nimit”}, {id: 2, name: “David”}]` and hands it to Sequelize =>
- **Sequelize** - ORM
    - Mutates the returned data and constructs new, powerful objects with prototypal methods, e.g. `save` =>
    - Resolves the promise it returned from `findAll` with this array of Sequelize instance objects.