Week 3: Key Concepts:
1. Method Chaining & THIS
2. Pure Functions vs Side Effects
3. Array Methods: New Array or Not?
4. Middleware

Resources (1 highly recommended this weekend; 3-7: ongoing reading)
1. The Express Routing Guide
2. *Node Beginner Book
3. *Effective JavaScript
4. *Principles of Object Oriented JavaScript
5. *JavaScript The Good Parts
6. *JavaScript The Definitive Guide

1. METHOD CHAINING & THIS

This week you've been digging your hands into the nitty gritty of data retrieval with SQL, FQL and PostgreSQL. Along the way, you've encountered method chaining, which is a common JS pattern in which a sequence of methods is called on an object in a single, unbroken line of code.

How is this possible?

Any method to which you can chain another method returns THIS. It's that simple. When a method returns THIS, the entire object on which the method is called is returned (recall how THIS is defined from last week's concept review).

Check out this gist

2. SIDE EFFECTS VS PURE FUNCTIONS

Pure functions are functions that always 1) return a value and 2) return the same value when given the same arguments. They take any number of arguments, return a value based on these arguments and don't produce any side effects (i.e. changing the color of a button on the DOM). A function produces a side effect if it modifies **state** or if it interacts with the world outside of the function (for example, modifying HTML elements or changing a global variable).

For example:

```
function add(a, b) {
      return a + b;
}
```

add() is a pure function because it will return 5 every time you give it 2 and 3 as parameters.

```
function alert(){
      alert('Hello')
}
```

Because alert() doesn't return a value, the value (pun intended) of the alert function lies in producing the side effect of alerting 'Hello'.

```
var global = 2;

function changeGlobal(){
      global++;
      return true;
}
```

changeGlobal() both returns a value and produces the side effect of incrementing the global variable, global. The key thing to remember when considering whether a function produces a side effect or not is that side effects are produced when a function's behavior depends on the history of the program, the order in which functions are evaluated, and the state of entities external to the function (in our example, the global variable).

3. ARRAY METHODS: NEW ARRAY OR NOT?

Understanding the difference between pure functions and functions that produce side effects can help us understand array methods more thoroughly.

In the 'functional' portion of Foundations, you had to code the forEach array method. You probably wrote something like this:

```
function forEach(array, callback){
      for (var i = 0, len = array.length; i < len; i++){
            callback(array[i])
      }
}
```

Notice that forEach does NOT return a value (and so it returns undefined). We invoke forEach only when we want to apply a callback to each element of an array, not when we want an entirely new array.

Contrast the code for forEach() with map():

```
function map(array, callback){
      var mapped = [];
      for (var i = 0, len = array.length; i < len; i++){
            mapped.push(callback(array[i]))
      }
      return mapped;
}
```

The map method applies a function to each element of an array and returns the new values of the function application in an entirely new array.

Check out the gist

So if we want to turn an array of lowercase names into an array of uppercase names, we can do this:

```
var names = [‘nimit’, ‘david’, ‘zeke’, ‘omri’]
var upperCaseNames = names.map(function(name) {
      return name.toUppercase();
});
console.log(upperCaseNames)  //[‘NIMIT’, ‘DAVID’, ‘ZEKE’, ‘OMRI’]
```

For a complete list of array methods and what they return, go here

5. MIDDLEWARE

This week you built WikiStack, your first fullstack application: you had a front-end comprised of HTML pages, some of which contained elements that were attached to click events that, fueled by callbacks, sent client requests (REQ) to your server to be routed through middleware that manipulated the REQ object (which carries any data required to fulfill the client’s request) in order to retrieve the necessary data from your PostgreSQL database, attach any necessary data to the RES object and then send that response (RES) object to the front-end.

Check out a diagram of the request/response flow.

So What Is Middleware?
Middleware is a suite of functions that 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. In other words, 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.

Check out this gist

Router.Params

For this section, follow along with this gist.

Oftentimes, many layers of our middleware stack require the same data. Instead of retrieving a document from our database in each layer that needs that object (we would do this by invoking Page.findOne in each route, for example), we can look this document up in one layer and attach the document object to the request (REQ) object so that each subsequent layer has the data it needs already (remember that the order of your middleware is crucial because your request object passes through each layer in order)

With Express.js, we can easily reduce the number of times we have to look up a document by leveraging **router.params** and **next().**

For example, with WikiStack, we had a number of routes that needed to find a page by id. Instead of asynchronously retrieving the page in each route that needs it (and copying and pasting the same page loading logic), we can define how the :id parameter will be handled for all routes that contain this parameter by using the **.param() method**.

The .param() method takes the name of the parameter (id in this case) and a function whose parameters include 1) REQ, 2) RES, 3) the callback NEXT() and 4) the ID which contains the value of the id parameter.

Things To Remember

1. Postgres vs psql: Opening the Postgres app starts the background server process that handles requests sent to your database. **psql** is the command line interface for accessing PostgreSQL databases. To ease your querying, consider downloading Postico.
2. The Req Object:
    a. Holds the request query, params, body, headers and cookies
    b. Is just a JavaScript object whose properties you can change
3. The Res Object:
    a. Holds the response sent to the client browser
    b. Once you res.send() or res.redirect() or res.render(), you cannot do this again. Each requests gets one response