**Week 6 Notes**

**WEB APIs**

* API: lets you do stuff, kind of like importing a library to a python script to use its functions.

Good example: Open Weather Map; they have a service that does something useful (allows you to obtain the weather, and they keep their own access to their own database). They want others to use their service so people can embed weather info in their applications.

Making Calls

Generally, we will be sending data either in a GET or POST. The server gets the data, parses it, then send back some sort of result (indicating success or failure, and potentially sending what the requester asked for).

Largely these are similar to function calls.

API Keys

Also called appid or clientid sometimes – these let the provider of the service know who is making the requests. This lets them limit how many requests a particular user is sending so no one may abuse the system by flooding it with many requests.

**HTTP AND JAVASCRIPT**

How does one use JS to make HTTP calls?

AJAX

**A**synchronous **J**avaScript **a**nd **X**ML

(XML is not really used anymore, JSON has taken its place almost completely)

GET REQUEST

example:

var req = new XMLHttpRequest();

req.open("GET", "example/fruit.json", false);

req.send(null);

console.log(JSON.parse(req.responseText));

// -> {banana: "yellow", lemon: "yellow", cherry: "red"}

This code sample is making a request to obtain data.

*XMLHttpRequest* is an object that can make an HTTP request and return the data it gets from the request (similar to filestream in C++).

The sample opens a GET request to the resource *example/fruit.json* which is a .json file on the same server the JS is on. The *false* arg says that it is a synchronous request. *null* means there is no additional data sent with the request, then *responseText* is logged from the request.

**ASYNCHRONOUS HTTP REQUESTS**

* Make a call and some time later it gets done. There are some requests that could take time to finish (e.g. a network request), and if you leave it synchronous, the page would freeze for the time to finish. Asynchronous calls don’t cause this freeze

Asynchronous Calls

* Works via callback functions
  + You pass the thing doing the work, then it “calls you back” when it is done e.g. pass a callback function that appens content to a span. An arbitrary timer is set, time passes, and the window calls the function we passed it and span is populated.

Asynchronous requests:

1. Make a request object
2. Tinker with URLs and maybe add a header or data
3. *Register a listener on the request’s* load *event* (often called a *callback)*
4. Send the request
5. Wait for the request to finish (you can do other things during this time)
6. *In the callback* do stuff with the response, usually by accessing *request.responseText*

**JAVASCRIPT PROMISES**

How JS Works

JavaScript isn’t actually an asynchronous language by nature, instead it is a *single threaded synchronous language*

* **Single Threaded**: Only one line of code can be executed at any given time
* **Synchronous**: Code is executed in order from top to bottom

In vanilla JS, each line of code can potentially *block* an entire program (not necessarily a negative trait, and is the case for many other languages)

* This trait can be negative when building a webpage (one might see extremely slowly loading pages)

Solution: JS devs developed a system to work around the *single threaded synchronous* limitations.

**JS’s 3 Parts**

1. Call Stack

* Every time a function is encountered during execution, it is pushed onto the call stack and only removed when the function returns or finishes.
* Can contain nested function calls (e.g. function A calls function B, which calls function C, the stack would look like: A, B, C.)

1. Heap

* Where the computer allocates memory for the variables and objects that a program declares. Not unique to JS

1. Queue

* Main divergence from other languages, this allows JS to behave in an *asynchronous* way
* This data structure stores messages
  + Can take multiple forms (events [like mouse clicks], callbacks)
* Works in tandem with the *Event Loop*
  + Event loop has the responsibility of monitoring the *call stack* and the *queue*
  + When the calls stack is empty, the Event Loop will remove the oldest message from the queue and add it to the call stack for execution. This continues until the call stack and queue are both empty.