Asynchronous Javascript and the Event Loop

goals: Get familiar with concurrency, parallelism, threads, and the difference between these.

* Event loop is something that is not often dived into
* A thread is a linear sequence of machine instructions that cannot be divided into parallel sequences.
* A runtime maintains a single call-stack, it usually means it’s single-threaded.
* JS when run synchronously does “sequential computing.”
* Concurrent computing is when several computations are executed during overlapping time periods.
  + This is when there are multiple things that are happening at the same time.
* Concurrency ex:
  + For example, a function is invoked, which invokes another function.
  + What we’re really doing in JS is context-switching. And so there isn’t really any overlap, although it seems so.
* Concurrency
* Parallelism relates to the simultaneous execution of separate parts of a program.
  + JS cannot do this on its own.
* Concurrency is not necessarily parallel;
  + JS can execute code concurrently via context-switching.
* Multithreaded languages allow programmers to separate concurrent operations across separate threads of execution.
* Whether or not multiple threads can actually be computed in parallel depends on the architecture of the system – for example, the number of cores available to run computations.
* A single-cored multi-threaded system may only still be able to compute one thing at a time.
  + Most systems will divide computation power across the different threads.
* Asynchronicity and the event Loop
  + setTimeout and other asynchronous functions are handled by the Web Browser API.
  + Remember – the console itself is an API itself as well!
  + This allows us to pass off tasks to the API rather than blocking the thread and waiting for tasks to complete.
  + Browsers use JS in their engines
  + And expose a webAPI which handles asyncrhonicity.
  + The setTimeout would push momentarily onto the callstack before passing off its functionality to the webBrowser API
    - And then after the setTimeout time, it would then move the function to the event/callback queue.
    - It checks if all of our **synchronous code has run**, and if **the call stack is empty.**
  + Only then would the event loop push the function onto the call stack.
* There are **two queues**
* Async/Await, under the hood, is achieving the same thing as utilizing a promise chain.
* Not all callbacks are created equal.
  + **Server** related deferred functionality is more important than **setTimeout** deferred functionality.
  + The **microtask queue** is a “vip” line for Promises.
  + It DOES NOT MATTER which one of the setTimeout or the fetch request was completed first.
  + Promises (server-related deferred functionality) are given priority over setTimeout.