Udemy Notes: Asychronous JavaScript

Sean al-Baroudi

January 10, 2022

1

1.1 Introduction:

- **Setting up for the Course:**
- Wordnik [1] requires an API key. Pre-register for this.
- SWAPI: [2] Star Wars API: Allows us to practise fetch requests and gather data. Has a large nested database on planets, characters, etc we can test on. Cannot write-to.
- JSONPlaceHolder: [5] Another database website that allows us to query and update data using the fetch() command.
-

2 Understanding Asychronous Coding:

2.1 Main:

- Synchronous code runs in order, and each line blocks until it finishes, before we proceed to the next line.
- Synchronous Code Pros/Cons:
- + Easy to write and reason about
- – may block.
- – less performance.
- Asynchronous Code Pros/Cons are the reverse of the above
- Asychronicity is achieved through the JS eventloop:
- Under the hood, JS has a Call Stack and and a Message Queue.

  The former stores statements of code that need to be run.

  The latter stores event messages that signal other bits of code are ready to be run (as some event has occured).

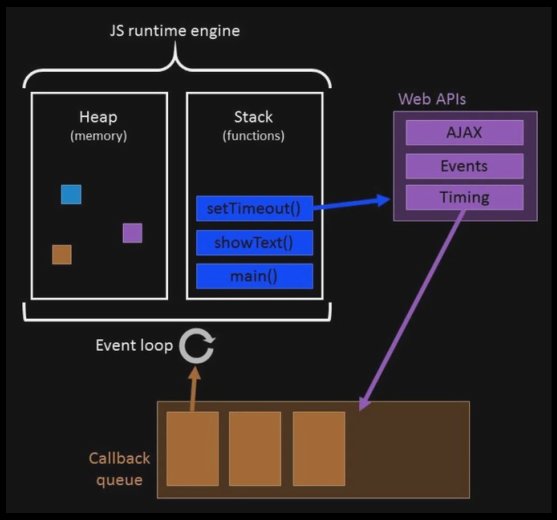
  The language loops by focusing on the call stack first, and then focusing on the message queue when the required statements have been executed.
- More infomration about how JS runs, with its event loop is illustrated in the two links below.

2.2 Illustrated Event Loop: [3]

- Objects of our Story:
  1. Call Stack: Each function that is called is placed on top of the stack. (First In, Last Out)
  2. WebAPI: Our asych calls (such as setTimeOut() or alert()) are not apart of core JS - they are provided in a separate default library.
  3. Message Queue: First In, First Out list of messages. These are associated with functions (via events). When an event listener is set, we add a new message to the queue.
  4. Event Loop: A loop process that checks if the call stack is empty. If so, the oldest message in the Queue is popped, and the associated function is placed on the call stack.
  5. JS Engine:
- This example gives a cute visualization of how different objects in JavaScript work together. We see that call stacks take priority of message queues.
- Also, for alerts, the WebAPI hands the message to the Message Queue - this is not done by the browser directly.
-

2.3 Derek Kwok: In depth JS Engine and Event Loop: [4]

- JS is considered synchronous because it only has one call stack. We need additional functionality to make it behave in an asychronous way.
- Visualization of the JavaScript Engine:



- More details on the inter-related parts:
  1. **Heap:** Stores variables, objects and acts as memory. Has an auto-garbage collector.
  2. **Stack:** Function and API calls placed here (FILO).
  3. **API's:** Our event functions are not apart of JS's core, the Web or C++ API contains calls such as *setTimeout()* and *fetch()* to help us with this behaviour.
  4. **Callback Queue:** Callbacks for asych functions are placed here. These entries are also known as *Event Messages*. This is a FIFO structure.
  5. **Event Loop:** An algorithm that regularly checks to see if any calls are on the stack. Once empty, the first entry on the callback queue is placed on the call stack.
- A thread for our purposes, is an entity that runs lines of code. So single threaded means there is just one entity running code.
- **Problem** with sychronous coding: Bottlenecks from functions that run a long time - later functions must wait.
- What actually happens when we run the following code?

```
console.log("Hello");
const takesALongTime = () => console.log("I am a function.");
setTimeout(takesALongTime, 2000);
console.log("Bye");
```

  1. Hello statement is pushed onto call stack. It is the only thing, so it is executed and then popped.
  2. Arrow function is defined, and pushed onto the Heap.
  3. setTimeOut pushed onto call stack, and then executed.
  4. setTimeOut calls the WebAPI to call 2 second timer.
  5. setTimeOut has done its job, so it is popped off the stack.
  6. the Bye statement is pushed onto the stack, executed and popped.
  7. WebAPI then finishes, and takesALongTime function is put in the callback queue.
  8. The Event loop sees an empty stack, and loaded callback queue. takesALongTime is pushed on the stack and executed.

3 Understanding Ascyhronous Coding:

- Callbacks are the **original feature** in JS that allowed for asychronous coding. Even Promises and other advanced features rely on Callbacks - which serve as the lower level for the code.
- **Basic setTimeout() Usage:**

```
setTimeout(function() {
    console.log("The function was called back 2");
}, 3000);
```

First parameter is a function (or reference to), second is the time period (in milliseconds).

- **Basic EventListener Usage:**

```
\\For a button on an HTML page
let btn = document.querySelector("#item1");

btn.addEventListener("click", function(e) {
    console.log("The button was clicked.");
});
```

Note: The event "e" argument in the passed function. This is an event object that can be modified.

- **Def: Callback** This is a kind of coding pattern where a function is passed into another function, and then run inside the function. Example:

```
let myFunction = function(data, callback) {

//Do Something
callback(data, other args...)
return something.
}
```

- Note that callbacks **on their own do not** make asychronous code - you need to invoke **the callback queue and the event loop** in order to do this.
- **Caveats with setTimeout:**
- Don't execute callback functions with () in the argument list. This will cause the function to run and return before the callback is set.

```
Wrong:

//Will run the funciton first
setTimeout(funname(myargs),300);


Correct:

//Third argument is for the parameters we will pass.
setTimeout(funname,300,myarg1,myarg2.,..);
```

- **Problems with Callback:**
- *Callback Hell:* Nested callbacks, or chains of callbacks all over the place. This is painful to debug and read through.
- *Indetermenancy of Run Order:* With many asych calls, you can't easily determine the order in which things will execute.
- *Inversion of Control:* Your program's control is turned over to something else (like the return of a server query). It may hang.
- In the next section, **Promises** will alleviate some of these issues!

4 Promises:

4.1

- **What is a Promise?**
  - Promises exist to alleviate the **inversion of control**, that can happen with async calls.
  - In terms of implementation, it is a JavaScript Object that must return a value.
  - It represents the eventual completion or failure of an asychronous operation.
-
- We wrap asynchronous operations with promises, so we can acertain if they will finish or not This helps us with our program flow and later computation.
- **Basic Code for a Promise:**

```
let asyncFunction = function() {
    return new Promise(function(resolve, reject) {
        setTimeout(function() {
            resolve("asyncFunction has resolved.");
        }, 4000);
    });
};
```

Note that two callbacks are provided to our anonymous function: *resolve and reject.* This allows us to deal with success and failure results.

- **The .then() Method:** When our promise is created, it may be a while before we get a result. we can call then with a callback function, to process the promise's result.
- An Example of .then() usage is below:

```
let p2 = p1.then(function(val) {
  console.log("Promise: " + val);
  return asyncFunction2();
});
```

Note that we can return another promise from our .then() method. Even if we do not, the returned result will be wrapped in a promise (!!). There is a good reason for this: Functional Chaining

- **Promises return results with other promise objects, in order to facilitate functional chaining.**
- **Code Style: Chained .then() methods:**

```
fetch(wordnikWords + "randomWord" + apiKey)
.then(function(response) {
  wordObj = response;
  //console.log(wordObj);
  return response.json();
})
.then(function(data) {
  console.log(data.word);
  return fetch(wordnikWord + data.word +
  "/definitions" + apiKey);
})
.then(function(response) {
  return response.json();
})
.then(function(def) {
  console.log(def[1].text)
})
.catch(function(err) {
  console.log(err);
});
```

Note that each .then() cleanly delimits what each section of code does.

We have just one .catch() method at the end, to handle errors.

Because a promise is returned after each call, we can just tack another .then() method at the end to continue the work.

- **The fetch() Method:** This new method replaces the original ajax.HTTPResponse() method back in the days of yore. This method will send an asychronous request to a server, and forms a promise on the expectation of a result from that server.
- **Note:** Fetch() will return a response **wrapped in a Promise, by default.**
- **Sending a fetch() GET Request:**

```
fetch(baseUrl + <path_to_resource> + <coded parameters>)
```

All parameters are stored in the URL.

- **Sending a fetch POST Request:**

```
//Our Data to send
let todo = {
  completed: false,
  userId: 1,
  title: "Promises, Promises..."
};

fetch('https://jsonplaceholder.typicode.com/todos/', {
  method: "POST",
  headers: {
    "Conent-type": "application/json"
  },
  body: JSON.stringify(todo)
})
```

- **The Promise Coding Pattern:** A promise is basically setup, for an arbitrary function below.

```
Let the body of our function be placed in the RES() function.

let funPromise = function(args) {
  return new Promise(function (res, rej) {
    if (TESTS/TIMEOUT LOGIC) {
      rej("Some error message")
    }
    setTimeout(res, 0)
    })
}

funPromise.then(<response callback>, <reject callback>)
```

  You also don't have to put the body in the res function - it can be slammed in the promise, with the res function being more lightweight.
- **Caveats with SetTimeout:** Note that without the setTimeOut usage, we cannot have async behaviour. Other Promise functions (such as fetch) - will invoke the message queue without needing this behaviour.

  If you are querying remotely, you do not have to setTime
- **Catching Errors for Promises:** You can either provide a reject() function as a second argument to the then() function. Or you can use an outer catch statement:

```
fumPromise.then(...)
.catch( myerr => console.log(err));
```

- Finally Method for Promises: This is used when we want to take down some resource or code, no-matter how the resource settles. For example, we set-up a connection/pipe and want to close it down after a series of queries. Code is tacked to the end of the then() chain:

```
fumPromise.then(...).catch(...)
.finally(<deconstruct code>);
```

-

4.2 More on Functions and IIFE's:

- **Def:** Function Declaration: A funciton definition that is not assigned to a variable (not first class). Such as below:

```
function funname(args....) {
//Body
return val;
}
```

  A declared function will be *hoisted* (put in a global scope) - and can be run from any part of the script.
- **Def:** Function Expression: A function definition that is first class - the function is defined as a variable. Note: you can still name the function, and also assign it to a varaible:

```
let myvar = function funname(args....) {
//Body
return val;
};
```

  These are not hoisted, so they must be declared before another part of the script uses them. You can use the variable assignment to pass the functions as callbacks - instead of relying on global scope (bad side-effects/scoping).
- IIFEs: Immediately Invoked Function Expressions: Like the name suggests, these are FEs that are run on the spot, **one time only.**.

  These are useful if you wish to run your code immediately, keep the scope local, and not waste memory on a function that will not be used again. Notation is below:

```
(function() {
  console.log(5*6);
})();
```

  Or *(function expression) ();*
- For regular function expressions, we can also use the *();* invoke parentheses to run it immediately after being defined. However as we have assigned it to a variable, it will stick around.
-
- **The Module Pattern:** This code pattern gives us the following benefits:
  - ☐ Local scoping of code.
  - ☐ IIFE capability.
  - ☐ Can return an object, which reveals public fields and methods. Local scope will stay around due to closure.
  - ☐ The rest of methods (not passed in return object), are kept private.
  - ☐ Our template code (MAINAPP) accounts for the fact that another developer may have code with a MAINAPP namespace, and does not destroy it by initializing with an empty object {}.
- The pattern itself is templated below:

```
var MAINAPP = MAINAPP || {}

(function(var) {
    "use strict";

    \\Definitions and functions

return {
field1:value1,
...,
fun1:function1
};

)(MAINAPP);
```

- In the end, a JS module is just a function. Here, we exploited IIFEs to get the propeties we wanted.
-

4.3 Promise Static Methods:

- **Def: Static Method:** One that is called via the Object classes constructor, instead of being applied to an instantiated object. So: CLASSNAME.method(). The function can run without any object attached to it.
- **Def "Settled":** Means resolved or rejected, some comnclusion has been made.
- **Def "Resolved":** Promise has completed successfully.
- **Promise.all():** Static method that takes in an array of promises. Only when all input promises return successfully (resolved), will this promise resolve.

  Analogous behaviour to an AND gate.

  Note: If just one promise fails, it will reject as well.

  Use this kind of multiplexing process when you have a number of necessary or related processes, before moving to the next step of async code.
  Usage:

```
Promise.all(promfun1(), .... , promfunN())
.then( //Process result);
```

- **Promise.race():** Method that returns when just one of the provided promises settles. Takes in an array of promises as input. Even if the first settlement is a rejection, will still return.
- **Promise.allSettled():** similar to .all(), it will return all results, even if some of them are rejected or time out.

  Corollary: No matter what the result of each input promise, this will always settle as "resolved".

  Use this multiplexing promise when you have a number of sub-promises that are independent of each other.

- **Promise.any():** Will respond when **any** of the input promises have status of fulfilled (read: resolved or complete). So promises that reject before one fulfills will be ignored!
-

5 Async and Await Functionality:

5.1

- **Def: Async Keyword:** A functional modifier that forces the said function to wrap its return value in a promise. The promise returned is always "resolved" in status, and a promise is still returned even if there is no return value (data field empty).
- **Note:** The async keyword **does not** make code asynchronous - it simply signals to JS that some asynchronous code will occur in the function.
- **Def: Await:** A keyword that blocks a line of code. Must be nested in a function that is predicated with async. For an async function, the rest of the code will after the *await* line will not execute, and wait for a Promise to return.
- *Other Caveats of Await:*
- ☐ Can only be used in an async function.
- ☐ Will wait for a promise to complete. code afterwards will block.
- ☐ For other code outside the async block, it will run as normal and **not** block.
- ☐ Await will **extract the response**, and not allow the response to be returned in a promise. *So using await will break .then() chains, as they act on promises.*
- **Try and Catch Functionality:**
- For our async/await functions, we can implement try/catch functionality to test code, and handle errors accordingly. This is done by making **try** and **catch** blocks of code. Usage is below:

```
const moviePlanets = async function(movieNum) {

  try {
//Body of Code to try.
      };

  } catch(e) {
    console.error(e);
  }
```

Note, you can do this instead of a reject() function, or outer .catch() statement on a then() chain. Any method is acceptable.

- **For - Await - OF Loop:** A new construct, that allows us to attach the await keyword for a loop. This is done when we are looping over an array of promises. As they will all arrive at different times (async) - we must await them!

Code Example:

```
for await (let pl of promises) {
  console.log(pl.name)
};
```

- Using Async for Promise.all(). This will cause the line to block until all promises have returned. Simply put await in front:

```
(async function() {
  let names = await Promise.all([firstName(), lastName()]);
  console.log(names[0] + " " + names[1]);
})();
```

-
-
-
-
-
-
-

6 Generators and Iterators:

- **Def:** Generator: A kind of modified function that can be iterated on, and can be re-entered arbitrarily.
- Generators require two constructs to be recognized by JS:
  1. Tagging the function as a generator (with the * operator)
  2. Usage of the **yield** keyword.
- When a generator is defined, it is not run by calling its name. We run it by calling the .next() method. It will run until it encounters its first yield statement. When we call .next() again, it will continue to run from the last yield statement, to the next one.
- Keyword: Yield: This signals JS to pause the function, and possibly return a value. When invoked again by .next(), it JS will re-enter whereever the last yield was placed.
- Caveats of Yield:
- ☐ An empty yield will yield a status object.
- ☐ A yield with a return value will populate the status object with the returned value, in addition to the generators current state.
- **Generators terminate** when they reach the end of the functional body, and no yield statement is encountered. A "Done" status will be returned.
- **Code Examples:** Define with function declarations, as follows:

```
function *genTest() {
    let x = 0;
    console.log('start');
    yield ++x;
    console.log(x);
    return x;
};
```

Note the use of the * operator. For functional expresions, use the following syntax:

```
const test = function *() {
\\Body
yield <value>;
\etc...
}

let it = test();
```

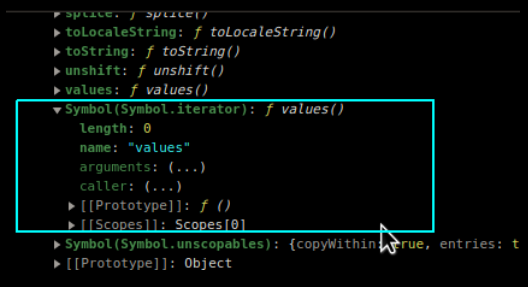
- Practical Example: Random Number Generator:

```
const randGen = function *(end = 1) {
  while (true) {
    yield Math.floor(Math.random() * end) + 1;
  }
};

let myGen1000 = randGen(1000);
```

This code will generate numbers forever, but only when we call it. It does not tie up the CPU.

- **What benefits do Generators give us?**
- ☐ Infinite data streams.
- ☐ Selective, iterative execution, to avoid code/memory blowups.
- ☐ Data and State concurrency (a normal function just ends w/o closure).
- **Iterators:** An iterator is a generator defined on a finine array/other data structure. They are used to traverse structures sequentially - using the next() method.
- In javascript, many datatypes are considered "iterables", incluidng arrays and strings.
- Iterable datatypes have default iterator functions built into them. These can be seen when they are examined in console:



- **Calling Iterator in an Iterable Structure:** Use array notation and invoke directly:

```
let it = arr[Symbol.iterator]();
```

- **Construct Iterator from Generator:** Like So:

```
const arrIt = function *(arr) {
  for (let i = 0; i < arr.length; i++) {
    yield arr[i];
  };
};

let it2 = arrIt(arr);
```

- **2 Way Generators: Passing In Values:** The **yield** keyword can accept values, in addition to passing them out of a generator function. An example with explanation is below:

```
1: function *yieldConsole() {
2:   let val = yield "Enter a Value:";
3:   console.log(val);
4: };
5:
6: let it = yieldConsole();
7: let prompt = it.next().value;
8: console.log(prompt);
```

- **The code works/executes as follows:**
  1. We first define *it* as a functional expression.
  2. We call the generator with .next(), and pull the value that yield has provided ("Enter a Value:"). We pause here and exit the function.
  3. Console.log executes, and prints "Enter a Value";
  4. Upon calling it.next(5000), we pass IN a value at the continuation point of the generator. The inner console.log statement prints our passed in value.
-

7 Residual Questions:

7.1

- How do I structure a promise for a function? There seem to be a number of different ways to do it.
- Is setTimeOut always required for promises? As the async message Queue is invoked with the sTO funciton. No, although this will make local asych operations schedule to the message queue. External async operations (at server) are naturally asynchronous.
- Can you await a .then() chain? **Answer:** NO, await breaks the promise chain.
-

References

[1] https://www.wordnik.com/

[2] https://swapi.dev/

[3] https://dev.to/kapantzak/js-illustrated-the-event-loop-4mco

[4] https://levelup.gitconnected.com/javascript-and-asynchronous-magic-bee537edc2da

[5] https://jsonplaceholder.typicode.com

[6]