Paul Galvin Follow
Solution Principal @ Slalom New York Father, Author, Not-Great-But-Always-Trying Zen Buddhist
Sep 27, 2016 · 5 min read

Use Reduce() and Promises to Execute Multiple Async Calls Sequentially

A little while back, I had boxed myself into a corner. I had refactored a bunch of code down to many granular functions. I was happy with the result, but came to realize that I had started invoking some things asynchronously that wrote over each other's responses as they resolved. This caused a bit of mayhem. I made a half hearted attempt to refactor that part of the code stateless, but in the end, decided to accept the technical debt and implement an easier short term solution. To wit, I would run these various async calls in sequence so that the stomping-on-each-other thing wouldn't happen.

```
const resolveIfAllOperationsComplete = () => {
    outstandingOperationCount--;
    if (outstandingOperationCount === 0) {
        deferred.resolve(true);
    }
};
```

One little operation, two little operations, three little operations...

I suspect that many people do the same thing I did when confronted with this problem. I invented a little promise tracking system. I knew in advance how many promises I had to run, so I would invoke one and wait for it to resolve. If I still had more to run, I'd bump a counter and invoke the next one and so forth until all operations completed. It was easy enough to implement but felt *off*. I did a bit of research and found something very interesting on the stack overflow machine. My eyes pretty much glazed over when I first saw it, but it's been eating at me, so I went and implemented a sequential async call thing in the tradition of the reduce() school.

Solution Setup

To simulate the real world scenario, I populate an array with Functions that return promises:

```
private _populatePromisesArray() {

    this._allPromises = [];

    const promiseFn = (url, trackingID) => {
        return this._sequentialPromisesService.GetSomething(url, trackingID);
    }

    for (let i = 0; i < 10; i++) {
        this._allPromises.push(promiseFn);
    }
}
```

Promises, promises

promiseFn is a JS function that takes in two parameters—the URL of a web site and a tracking ID. When invoked, any given promiseFn runs **GetSomething()** on _sequentialPromisesService, passing the URL and tracking ID. The URL is obviously to a web site and the trackingID is meant to demonstrate the sequentionality of the process by logging its progress out to the console.

[Total aside—"sequentionality"—I love the English language :)]

_sequentialPromiseService happens to be an Angular service but that doesn't really matter for the bigger point. That code looks like this:

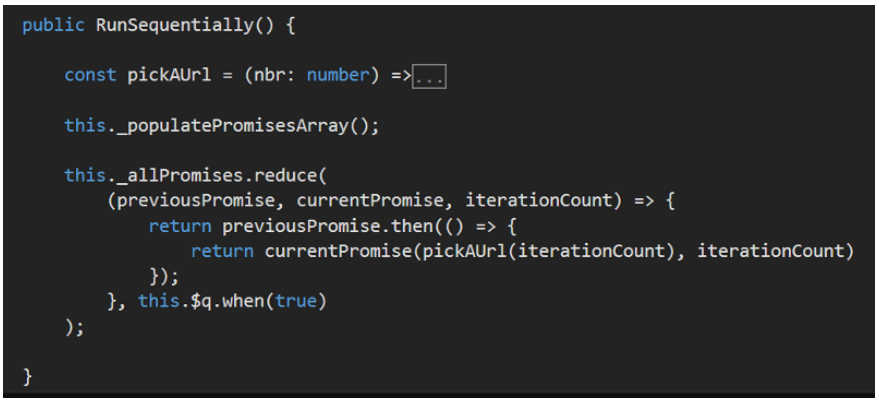
```
public GetSomething(url: string, myID: number): ng.IPromise<any>{

    const deferred = this.$q.defer<any>();

    this.$http.get(url).then(
        () => {
            this.$log.debug(`SequentialPromisesController: promise id ${myID} resolved.`);
            deferred.resolve("worked");
        }
    );

    return deferred.promise;

}
```

Get Me Something, Anything

There's a bit of angular jibber jabber here. The key point is that the call to **$http.get()** retrieves the indicated URL—eventually. It's asynchronous and once it gets the web page, it logs out that fact and resolves the promise with a throwaway value "worked." In the real world, it would return actual data.

Running Things Sequentially

Let's see how to run a bunch of these calls in sequence. Observe the following code:



```
public RunSequentially() {

    const pickAUrl = (nbr: number) =>...

    this._populatePromisesArray();

    this._allPromises.reduce(
        (previousPromise, currentPromise, iterationCount) => {
            return previousPromise.then(() => {
                return currentPromise(pickAUrl(iterationCount), iterationCount)
            });
        }, this.$q.when(true)
    );

}
```

Put one foot in front of the other
And soon you'll be walking 'cross the floor
Put one foot in front of the other
And soon you'll be walking out the door

RunSequentially defines a little lambda function, pickAUrl. This function takes a number as input, mods it against two and picks one of two URLs. Nothing special.

The function **_populatePromisesArray** creates 10 entries in the _allPromises array, alternating between URLs.

And finally at long last, we've arrived at the reduce function.

As usual, this iterates over a collection—"this._allPromises" in this case. It passes in three values:

1. The previous promise.
2. The current promise.
3. An index. This is only used for logging out to the console in this example.

Lastly, it initializes the first result to **this.$q.when(true)**. This is just an Angular technique that immediately resolves a value (true, in this case). Other libraries do it differently. You just need to have one promise like thing kick off the process with a successful resolve.

The real magic happens here:

```
return previousPromise.then(() => {
    return currentPromise(pickAUrl(iterationCount), iterationCount)
});
```

First, keep in mind that the very first "**previousPromise**" automatically resolves. It's that angular thing, $q.when(true)—resolving immediately and passing along the value "true" to the code catching the resolve. On the first go-around, it's previousPromise. Since its waiting on a resolve and it gets it from that when() call, it goes straight into its then().

The "then" logic, shown as the lambda function "() => {…}," itself returns the result from invoking **currentPromise**, passing in the URL and iteration count (again, for logging). As a reminder, that's promiseFn down below:

```
private _populatePromisesArray() {

    this._allPromises = [];

    const promiseFn = (url, trackingID) => {
        return this._sequentialPromisesService.GetSomething(url, trackingID);
    }

    for (let i = 0; i < 10; i++) {
        this._allPromises.push(promiseFn);
    }
}
```

Promises, promises

currentPromise itself immediately returns the result of _sequentialPromisesService.GetSomething().

GetSomething() makes an http call and sits around waiting for the response. but not before it responds back with a deferred promise.

Meanwhile, back at reduce, it immediately moves on to the next entry in the array because it just made an asynchronous call to GetSomething(). Happily for us, it slams into the then() of the previous promise. So, it waits. Once the previous promise resolves, it moves on in its shark-like way to the next entry in the array. That

engenders a new returned promise, reduce iterates and once again slams up against the previous promise's then().

Eventually that promise and all the rest of them resolve. Before you know it, you have completed every async call in sequence. In other words, Bob's your uncle.

In Summary

This is yet another example where that word "reduce" feels like a poor description of functionality. I think it's hard to make any case, let alone a reasonable case, that we have "reduced" anything here.

I do think it's very cool. Reduce, used this way, provides a concise and neat way to control the flow of your application. With some imagination and a bit of work, but probably not that much, you could chain together quite an assortment of async calls, some of which run in parallel and some that must be sequential. In a sense, you could transform entire pieces of functionality
into a sequence of chained promises that fire in response to user input. It's an interesting idea and something I hope to pursue.

In the meantime, happy coding!

</end>