Event-Based Programming: What Async Has Over Sync

Jonathan Creamer on Feb 13th 2013 with 14 Comments

Tutorial Details

-
  - Topic: JavaScript
  - Difficulty: Medium
  - Estimated Completion Time: 30 Minutes

View post on Tuts+ Beta**Tuts+ Beta** is an optimized, mobile-friendly and easy-to-read version of the Tuts+ network.

One of JavaScript's strengths is how it handles asynchronous (async for short) code. Rather than blocking the thread, async code gets pushed to an event queue that fires after all other code executes. It can, however, be difficult for beginners to follow async code. I'll help clear up any confusion you might have in this article.

Understanding Async Code

JavaScript's most basic async functions are `setTimeout` and `setInterval`. The `setTimeout` function executes a given function after a certain amount of time passes. It accepts a callback function as the first argument and a time (in milliseconds) as the second argument. Here's an example of its usage:

```
console.log( "a" );
setTimeout(function() {
    console.log( "c" )
}, 500 );
setTimeout(function() {
    console.log( "d" )
}, 500 );
setTimeout(function() {
    console.log( "e" )
}, 500 );
console.log( "b" );
```

As expected, the console outputs "a", "b", and then 500 ms(ish) later, we see "c", "d", and "e". I use "ish" because `setTimeout` is actually unpredictable. In fact, even the HTML5 spec talks about this issue:

> "This API does not guarantee that timers will run exactly on schedule. Delays due to CPU load, other tasks, etc, are to be expected."

Interestingly, a timeout will not execute until all of the remaining code in a block has executed. So if a timeout is set, and then some long running function executes, the timeout will not even start until that long running function has finished. In actuality, async functions like `setTimeout` and `setInterval` are pushed onto an queue known as the **Event Loop**.

The **Event Loop** is a queue of callback functions. When an async function executes, the callback function is pushed into the queue. The JavaScript engine doesn't start processing the event loop until the code after an async function has executed. This means that JavaScript code is not multi-threaded even though it appears to be so. The event loop is a first-in-first-out (FIFO) queue, meaning that callbacks execute in the order they were added onto the queue. JavaScript was chosen for the language of node because of how easy it is to write this kind of code.

Ajax

Asynchronous JavaScript and XML (Ajax) forever changed the landscape of JavaScript. All of the sudden, a browser could update a web page without having to reload it. The code for implementing Ajax in different browsers can be long and tedious to write; however, thanks to jQuery (and other libraries), Ajax became an extremely easy and elegant solution to facilitate client-server communication.

Asynchronously retrieving data with jQuery's `$.ajax` is an easy cross-browser process, but it's not immediately evident as to what exactly happens behind the scenes. For example:

```
var data;
$.ajax({
    url: "some/url/1",
    success: function( data ) {
        // But, this will!
        console.log( data );
    }
})
// Oops, this won't work...
console.log( data );
```

It's common, but incorrect, to assume the data is available immediately after calling `$.ajax`, but what actually happens is this:

```
xmlhttp.open( "GET", "some/ur/1", true );
xmlhttp.onreadystatechange = function( data ) {
    if ( xmlhttp.readyState === 4 ) {
        console.log( data );
    }
};
xmlhttp.send( null );
```

The underlying `XmlHttpRequest` (XHR) object sends the request, and the callback function is set to handle the XHR's `readystatechange` event. Then the XHR's `send` method executes. As the XHR performs its work, an internal `readystatechange` event fires every time the `readyState` property changes, and it's only when the XHR finishes receiving a response from the remote host that the callback function executes.

Working With Async Code

Async programming lends itself to what’s commonly referred to as “callback hell”. Because virtually all async functions in JavaScript use callbacks, performing multiple sequential async functions result in many nested callbacks—resulting in hard to read code.

Many of the functions within node.js are async. So, code like the following is quite common.

```
var fs = require( "fs" );

fs.exists( "index.js", function() {
    fs.readFile( "index.js", "utf8", function( err, contents ) {
        contents = someFunction( contents ); // do something with
        fs.writeFile( "index.js", "utf8", function() {
            console.log( "whew! Done finally..." );
        });
    });
});
console.log( "executing..." );
```

It’s also common to see client-side code, like the following:

```
GMaps.geocode({
    address: fromAddress,
    callback: function( results, status ) {
        if ( status == "OK" ) {
            fromLatLng = results[0].geometry.location;
            GMaps.geocode({
                address: toAddress,
                callback: function( results, status ) {
                    if ( status == "OK" ) {
                        toLatLng = results[0].geometry.location;
                        map.getRoutes({
                            origin: [ fromLatLng.lat(), fromLatLng
                            destination: [ toLatLng.lat(), toLatLn
                            travelMode: "driving",
                            unitSystem: "imperial",
                            callback: function( e ){
                                console.log( "ANNNND FINALLY here'
                                // do something with e
                            }
                        });
                    }
                }
            });
        }
    }
});
```

Nested callbacks can get really nasty, but there are several solutions to this style of coding.

> The problem isn’t with the language itself; it’s with the way programmers use the language — Async Javascript.

Named Functions

An easy solution that cleans nested callbacks is simply avoiding nesting more than two levels. Instead of passing anonymous functions to the callback arguments, pass a named function:

```
var fromLatLng, toLatLng;

var routeDone = function( e ){
    console.log( "ANNNND FINALLY here's the directions..." );
    // do something with e
};

var toAddressDone = function( results, status ) {
    if ( status == "OK" ) {
        toLatLng = results[0].geometry.location;
        map.getRoutes({
            origin: [ fromLatLng.lat(), fromLatLng.lng() ],
            destination: [ toLatLng.lat(), toLatLng.lng() ],
            travelMode: "driving",
            unitSystem: "imperial",
            callback: routeDone
        });
    }
};

var fromAddressDone = function( results, status ) {
    if ( status == "OK" ) {
        fromLatLng = results[0].geometry.location;
        GMaps.geocode({
            address: toAddress,
            callback: toAddressDone
        });
    }
};

GMaps.geocode({
    address: fromAddress,
    callback: fromAddressDone
});
```

Additionally, the async.js library can help handle multiple Ajax requests/responses. For example:

```
async.parallel([
    function( done ) {
        GMaps.geocode({
            address: toAddress,
            callback: function( result ) {
                done( null, result );
            }
        });
    },
    function( done ) {
        GMaps.geocode({
            address: fromAddress,
            callback: function( result ) {
                done( null, result );
            }
        });
    }
], function( errors, results ) {
    getRoute( results[0], results[1] );
});
```

This code executes the two asynchronous functions, and each function accepts a “done” callback that executes after the async function finishing running. When both “done” callbacks finish, the `parallel` function’s callback executes and handles any errors or results from the two async functions.

Promises

From the CommonJS/A:

> A promise represents the eventual value returned from the single completion of an operation.

There are many libraries that incorporate the promise pattern, and jQuery users already have a nice promise API available to them. jQuery introduced the `Deferred` object in version 1.5, and using the `jQuery.Deferred` constructor results in a function that returns a promise. A promise-returning function performs some sort of async operation and resolves the deferred upon completion.

```
var geocode = function( address ) {
    var dfd = new $.Deferred();
    GMaps.geocode({
        address: address,
        callback: function( response, status ) {
            return dfd.resolve( response );
        }
    });
    return dfd.promise();
};

var getRoute = function( fromLatLng, toLatLng ) {
    var dfd = new $.Deferred();
    map.getRoutes({
        origin: [ fromLatLng.lat(), fromLatLng.lng() ],
        destination: [ toLatLng.lat(), toLatLng.lng() ],
        travelMode: "driving",
        unitSystem: "imperial",
        callback: function( e ) {
            return dfd.resolve( e );
        }
    });
    return dfd.promise();
};

var doSomethingCoolWithDirections = function( route ) {
    // do something with route
};

$.when( geocode( fromAddress ), geocode( toAddress ) ).
    then(function( fromLatLng, toLatLng ) {
        getRoute( fromLatLng, toLatLng ).then( doSomethingCoolWith
    });
```

This allows you to execute two asyncrhonous functions, wait for their results, and then execute another function with the results of the first two calls.

> A promise represents the eventual value returned from the single completion of an operation.

In this code, the `geocode` method executes twice and returns a promise. The async functions then execute and call `resolve` in their callbacks. Then, once both have called `resolve`, the `then` executes, returning the results of the first two calls to `geocode`. The results are then passed to `getRoute`, which also returns a promise. Finally, when the promise from `getRoute` is resolved, the `doSomethingCoolWithDirections` callback executes.

Events

Events are another solution to communicate when async callbacks finish executing. An object can become an emitter and publish events that other objects can listen for. This type of eventing is called the **observer pattern**. The backbone.js library has this type of functionallity built in with `Backbone.Events`.

```
var SomeModel = Backbone.Model.extend({
   url: "/someurl"
});

var SomeView = Backbone.View.extend({
    initialize: function() {
        this.model.on( "reset", this.render, this );

        this.model.fetch();
    },
    render: function( data ) {
        // do something with data
    }
});

var view = new SomeView({
    model: new SomeModel()
});
```

There are other mixin examples and libraries for emitting events, such as jQuery Event Emitter, EventEmitter, monologue.js, and node.js has a built-in EventEmitter module.

> The Event Loop is a queue of callback functions.

A similar method of publishing messages uses the **mediator pattern**, used in the postal.js library. In the mediator pattern, a middleman for all objects listens to and publishes events. In this approach, one object does not have a direct reference to another, thereby de-coupling the objects from each other.

Never return a promise accross a public API. This couples the API consumers to using promises and makes refactoring difficult. However, a combination of promises for internal purposes and eventing for external APIs can lead to a nicely de-coupled and testable app.

In the previous example, the `doSomethingCoolWithDirections` callback function executes when the two previous `geocode` functions have completed. The `doSomethingCoolWithDirections` can then take the response it recieved from `getRoute` and publish the reponse as a message.

```
var doSomethingCoolWithDirections = function( route ) {
    postal.channel( "ui" ).publish( "directions.done",  {
        route: route
    });
};
```

This allows other areas of the application to respond to the asynchronous callback without needing a direct reference to the request-making object. It's possible that multiple areas of a page need to update when directions have been retrieved. In a typical jQuery Ajax setup, the success callback would need to be adjusted when recieving a change of directions. This can become difficult to maintain, but by using messaging, updating multiple parts of the UI is much easier to work with.

```
var UI = function() {
    this.channel = postal.channel( "ui" );
    this.channel.subscribe( "directions.done", this.updateDirectio
};

UI.prototype.updateDirections = function( data ) {
    // The route is available on data.route, now just update the U
};

```

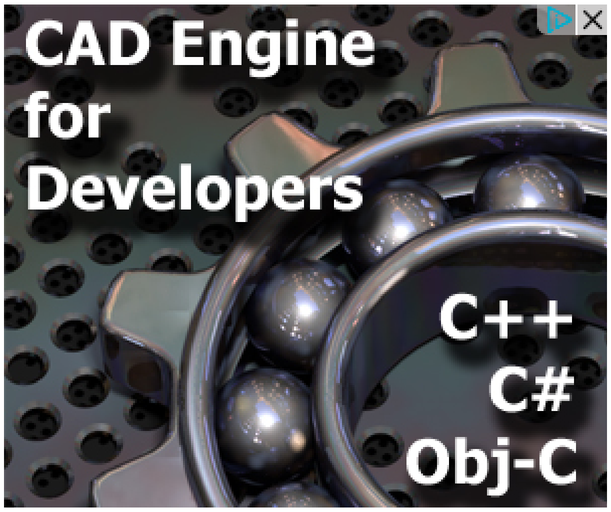
```
app.ui = new UI();
```

Some other mediator pattern–based messaging libraries are amplify, PubSubJS, and radio.js.

Conclusion

JavaScript makes writing async code very easy. Using promises, eventing, or named functions eliminates the nasty “callback hell”. For more information on async JavaScript, checkout Async JavaScript: Build More Responsive Apps with Less Code. Many of the examples from the post reside in a Github repository called NetTutsAsyncJS. Clone away!

Like 181 people like this. Be the first of your friends.



Tags: asyncasynchronous

By Jonathan Creamer

Currently a JavaScript Engineer at appendTo. Lover of all things Web, but mostly ASP.NET MVC, JavaScript, jQuery, and C#. I believe that you cannot ever stop learning which is why I stay active in the .NET world attending User Groups, blogging for FreshBrewedCode, http://jcreamerlive.com and scouring Twitter via @jcreamer898 and the interwebs for as much knowledge I can squeeze into my brain. I work at

AppendTo and am having a great time developing front end applications in jQuery and JavaScript. Please feel free to contact me, I love meeting other devs who are passionate about what they do...