The asynchronous, non-blocking Javascript runtime can be a real challenge for those of us who are used to writing in a synchronous style in languages such as Python or Java. Especially tough is when we need to do several inherently asynchronous things in a particular order…maybe a filter chain…in which the result of a preceding step is used in the next. The typical JS approach is to nest callbacks, but this leads to code that can be hard to maintain.

The following programs illustrate the problem and work toward a solution. In each, the leading comments describe the approach and any issues that it creates. The final solution can be used as a pattern to solve general synchronization problems in Javascript. The formatting options in the BU WordPress editor are a little limited, so you might want to cut and paste each example into your code editor for easier reading.

**1. The problem**

/\*

If you are used to writing procedural code in a language like Python, Java, or C++,

you would expect this code to print step1, step2, step3, and so on. Because

JavaScript is non-blocking, this isn't what happens at all. The HTTP requests take

time to execute, and so the JS runtime moves the call to a queue and just keeps

going. Once all of the calls on the main portion of the call stack are complete, an

event loop visits each of the completed request(s) in the order they completed and

executes their callbacks.

 Starting demo

Finished demo

step3: UHub

step2: CNN

step1: KidPub

So what if we need to execute the requests in order, maybe to build up a result from

each of them?

\*/

var request = require('request');

var step1 = function (req1) {

    request(req1, function (err, resp) {

        console.log('step1: KidPub');

    });

};

var step2 = function (req2) {

    request(req2, function (err, resp) {

        console.log('step2: CNN');

    });

};

var step3 = function(req3) {

    request(req3, function (err, resp) {

        console.log('step3: UHub');

    });

};

console.log('Starting demo');

step1('[http://www.kidpub.com](http://www.kidpub.com/)');

step2('[http://www.cnn.com](http://www.cnn.com/)');

step3('[http://universalhub.com](http://universalhub.com/)');

console.log('Finished demo');

/\*

**2. Callbacks work just fine, but…**

/\*

This is the classic way to synchronize things in Javascript using callbacks. When

each request completes, its callback is executed. The callback is still placed in the

event queue, which is why this code prints

Starting demo

Finished demo

step1: BU

step2: CNN

step3: UHub

There's nothing inherently wrong with this approach, however it can lead to what is

called 'callback hell' or the 'pyramid of doom' when the number of synchronized items

grows too large.

\*/

var request = require('request');

var req1 = '[http://www.bu.edu](http://www.bu.edu/)';

var req2 = '[http://www.cnn.com](http://www.cnn.com/)';

var req3 = '[http://universalhub.com](http://universalhub.com/)';

var step1 = function () {

    request(req1, function (err, resp) {

        console.log('step1: BU');

        request(req2, function (err, resp) {

            console.log('step2: CNN');

            request(req3, function(err,resp) {

                console.log('step3: UHub');

            })

        })

    });

};

console.log('Starting demo');

step1();

console.log('Finished demo');

**3. Promises help solve the problem, but there’s a gotcha to watch out for**

One way to avoid callback hell is to use thenables (pronounced THEN-ables), which essentially implement the callback in a separate function, as described in the Promise/A+ specification.

A Javascript Promise is a value that can be returned that will be filled in or completed at some future time. Most libraries can be wrapped with a Promise interface, and many implement Promises natively. In the code here we're using the request-promise library, which wraps the standard HTTP request library with a Promise interface. The result is code that is much easier to read...an event happens, THEN another event happens, THEN another and so on.

This might seem like a perfectly reasonable approach, chaining together calls to external APIs in order to build up a final result.

The problem here is that a Promise is returned by the rp() call in each step...we are effectively nesting Promises. The code below appears to work, since it prints the steps on the console in the correct order. However, what's really happening is that each rp() does NOT complete before moving on to its console.log(). If we move the console.log() statements inside the callback for the rp() you'll see them complete out of order, as I've done in step 2. Uncomment the console.log() and you'll see how it starts to unravel.

\*/

var rp = require('request-promise');

var Promise = require('bluebird');

var req1 = '[http://www.bu.edu](http://www.bu.edu/)';

var req2 = '[http://www.cnn.com](http://www.cnn.com/)';

var req3 = '[http://universalhub.com](http://universalhub.com/)';

function doCalls(message) {

    return new Promise(function(resolve, reject) {

        resolve(1);

    }).then(function (result) {

            rp(req1, function (err, resp) {

            });

            console.log('step:', result, ' BU ');

            return ++result;

        }).then(function (result) {

                rp(req2, function (err, resp) {

//                    console.log('step:', result, ' CNN');

                });

            console.log('step:', result, ' CNN');

            return ++result;

            }

        )

        .then(function (result) {

            rp(req3, function (err, resp) {

            });

            console.log('step:', result, ' UHub');

            return ++result;

        })

        .then(function (result) {

            console.log('Ending demo at step ', result);

            return;

        })

}

doCalls('Starting calls')

    .then(function (resolve, reject) {

        console.log('Complete');

    })

**4.  Using Promise.resolve to order nested asynchronous calls**

/\*

Here's the final approach. In this code, each step returns a Promise to the next step, and the

steps run in the expected order, since the promise isn't resolved until

the rp() and its callback are complete. Here we're just manipulating a function variable,

'step', in each step but one can imagine a use case in which each of the calls

 would be building up a final result and perhaps storing intermediate data in a db. The

 result variable passed into each proceeding then is the result of the rp(), which

would be the HTML page returned by the request.

The advantages of this over the traditional callback method are that it results

in code that's easier to read, and it also simplifies a stepwise process...each step

is very cleanly about whatever that step is intended to do, and then you move

on to the next step.

\*/

var rp = require('request-promise');

var Promise = require('bluebird');

var req1 = '[http://www.bu.edu](http://www.bu.edu/)'; var req2 = '[http://www.cnn.com](http://www.cnn.com/)'; var req3 = '[http://universalhub.com](http://universalhub.com/)';

function doCalls(message) {

    var step = 0;

    console.log(message);

    return new Promise(function(resolve, reject) {

        resolve(1);

    })

        .then(function (result) {

            return Promise.resolve(

                rp(req1, function (err, resp) {

                    console.log('step: ', ++step, ' BU ')

                })

            )

        })

        .then(function (result) {

                return Promise.resolve(

                    rp(req2, function (err, resp) {

                        console.log('step: ', ++step, ' CNN');

                    })

                )

            }

        )

        .then(function (result) {

            return Promise.resolve(

                rp(req3, function (err, resp) {

                    console.log('step: ', ++step, ' UHub');

                })

            )

        })

        .then(function (result) {

            console.log('Ending demo ');

            return Promise.resolve(step);

        })

}

doCalls('Starting calls')

    .then(function (steps) {

            console.log('Completed in step ', steps);

    });

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*promise syntax\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

var myVar = value;

var prom = new Promise(function (resolve, reject) {

if (myVar) {

some var…

resole(that some var);

} else {

var err =new Error(“some content”);

reject(err);

}

});

var consumePromFunc = function () {

prom

.then(function (fulfilled) {log success msg })

.catch(function (error) {log something });

};

consumePromFunc( );