Ben Nadel

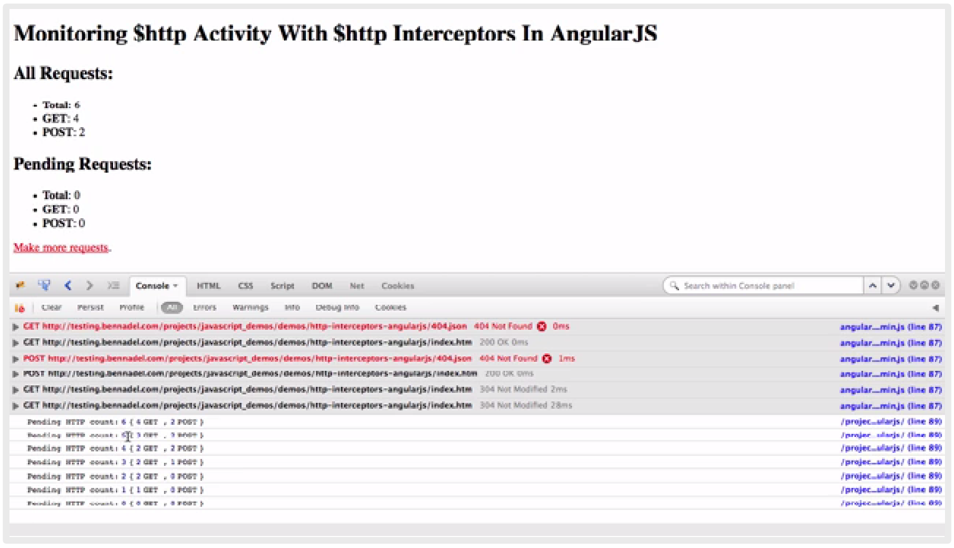
On User Experience (UX) Design, JavaScript, ColdFusion, Node.js, Life, and Love.

« Previous Photo Next Photo »

Monitoring $http Activity With $http Interceptors In AngularJS

By Ben Nadel on February 4, 2015
Tags: Javascript / DHTML

After experimenting with the $provide.decorator() method in AngularJS, I wanted to try looking at another form of decorator (of sorts) - an interceptor. For a long time, AngularJS has allowed us to intercept $http responses. Then, in AngularJS 1.1.4, they extended interceptor functionality to include both outgoing requests and incoming responses. Now, with our ability to straddle both sides of the HTTP fence, I wanted to see if I could use interceptors to track and monitor HTTP requests within an AngularJS application.



Monitoring $http Activity With $http Interceptors In AngularJS

All Requests:

- Total: 6
- GET: 4
- POST: 2

Pending Requests:

- Total: 0
- GET: 0
- POST: 0

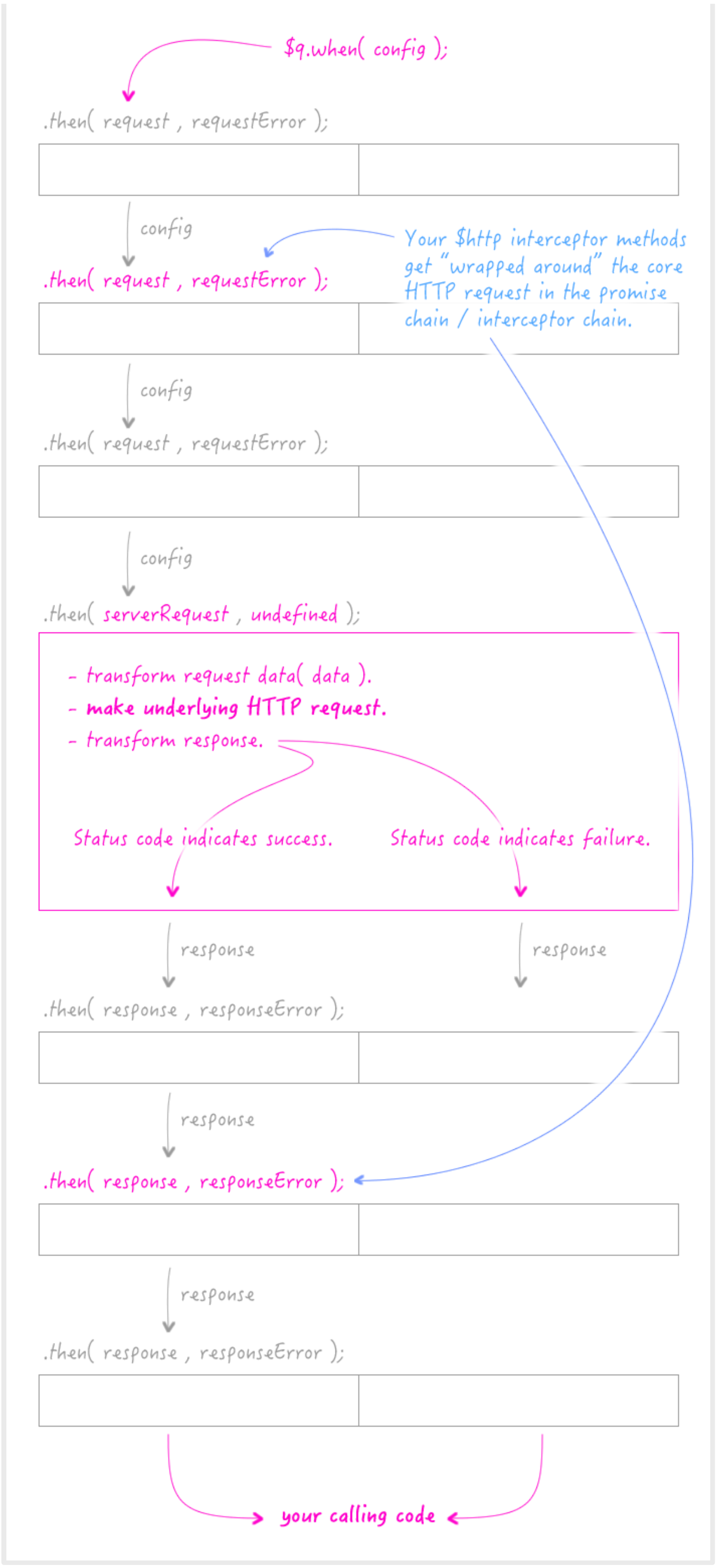
Make more requests.

```
GET http://testing.bennadel.com/projects/javascript_demos/demos/http-interceptors-angularjs/404.json 404 Not Found 0ms      angular...min.js (line 87)
GET http://testing.bennadel.com/projects/javascript_demos/demos/http-interceptors-angularjs/index.htm 200 OK 0ms           angular...min.js (line 87)
POST http://testing.bennadel.com/projects/javascript_demos/demos/http-interceptors-angularjs/404.json 404 Not Found 1ms     angular...min.js (line 87)
POST http://testing.bennadel.com/projects/javascript_demos/demos/http-interceptors-angularjs/index.htm 200 OK 0ms          angular...min.js (line 87)
GET http://testing.bennadel.com/projects/javascript_demos/demos/http-interceptors-angularjs/index.htm 304 Not Modified 2ms  angular...min.js (line 87)
GET http://testing.bennadel.com/projects/javascript_demos/demos/http-interceptors-angularjs/index.htm 304 Not Modified 28ms angular...min.js (line 87)
   Pending HTTP count: 6 ( 4 GET , 2 POST )                                                                              /projec...ularjs/ (line 89)
   Pending HTTP count: 5 ( 3 GET , 2 POST )                                                                              /projec...ularjs/ (line 89)
   Pending HTTP count: 4 ( 2 GET , 2 POST )                                                                              /projec...ularjs/ (line 89)
   Pending HTTP count: 3 ( 2 GET , 1 POST )                                                                              /projec...ularjs/ (line 89)
   Pending HTTP count: 2 ( 2 GET , 0 POST )                                                                              /projec...ularjs/ (line 89)
   Pending HTTP count: 1 ( 1 GET , 0 POST )                                                                              /projec...ularjs/ (line 89)
   Pending HTTP count: 0 ( 0 GET , 0 POST )                                                                              /projec...ularjs/ (line 89)
```

Run this demo in my **JavaScript Demos project** on GitHub.

Like the $provide.decorator() in my previous blog post, the $http request interceptors have to be setup during the configuration phase of the bootstrapping. The config phase is the only time we have access to the providers before the underlying services have been instantiated.

The entire $http request / response interceptor chain is implement as a Promise chain. The core $http request sits, as a promise, in the middle of the chain. Your request interceptors are then added before the core request while your response interceptors are added after the core request:


$q.when(config);
.then(request , requestError);
config
.then(request , requestError);
Your $http interceptor methods get "wrapped around" the core HTTP request in the promise chain / interceptor chain.
config
.then(request , requestError);
config
.then(serverRequest , undefined);
- transform request data(data).
- make underlying HTTP request.
- transform response.
Status code indicates success.
Status code indicates failure.
response
response
.then(response , responseError);
response
.then(response , responseError);
response
.then(response , responseError);
your calling code

It's quite beautiful, isn't it? Using interceptors, we should be able to tally up all the outgoing requests and then decrement pending requests on the way back. In practice, this actually becomes quite easy.

In theory, however, this is problematic. See, in theory, you could have a competing request interceptor or a response interceptor that completely modifies the config object, or throws an error making the config object unavailable. In such - albeit outlier edge-cases - it will become very difficult (in an elegant way) to reconcile an outgoing request with an incoming response.

REALITY CHECK: Now, again, in practice, I want to stress that this will likely never happen. In practice, you probably only have one interceptor (if that) and know exactly what it's doing. The complexity would really only occur in cases where you're using a third-party module that has implemented an $http interceptor that is doing something silly and unfortunate.

That said, I wanted to try to and explore $http interceptors in a way that makes the code a bit more defensive. In the following demo, I have two main blocks - the trafficCop service which manages the tally of HTTP request activity and, the config() block which sets up the interceptor that pipes $http activity into the trafficCop service:

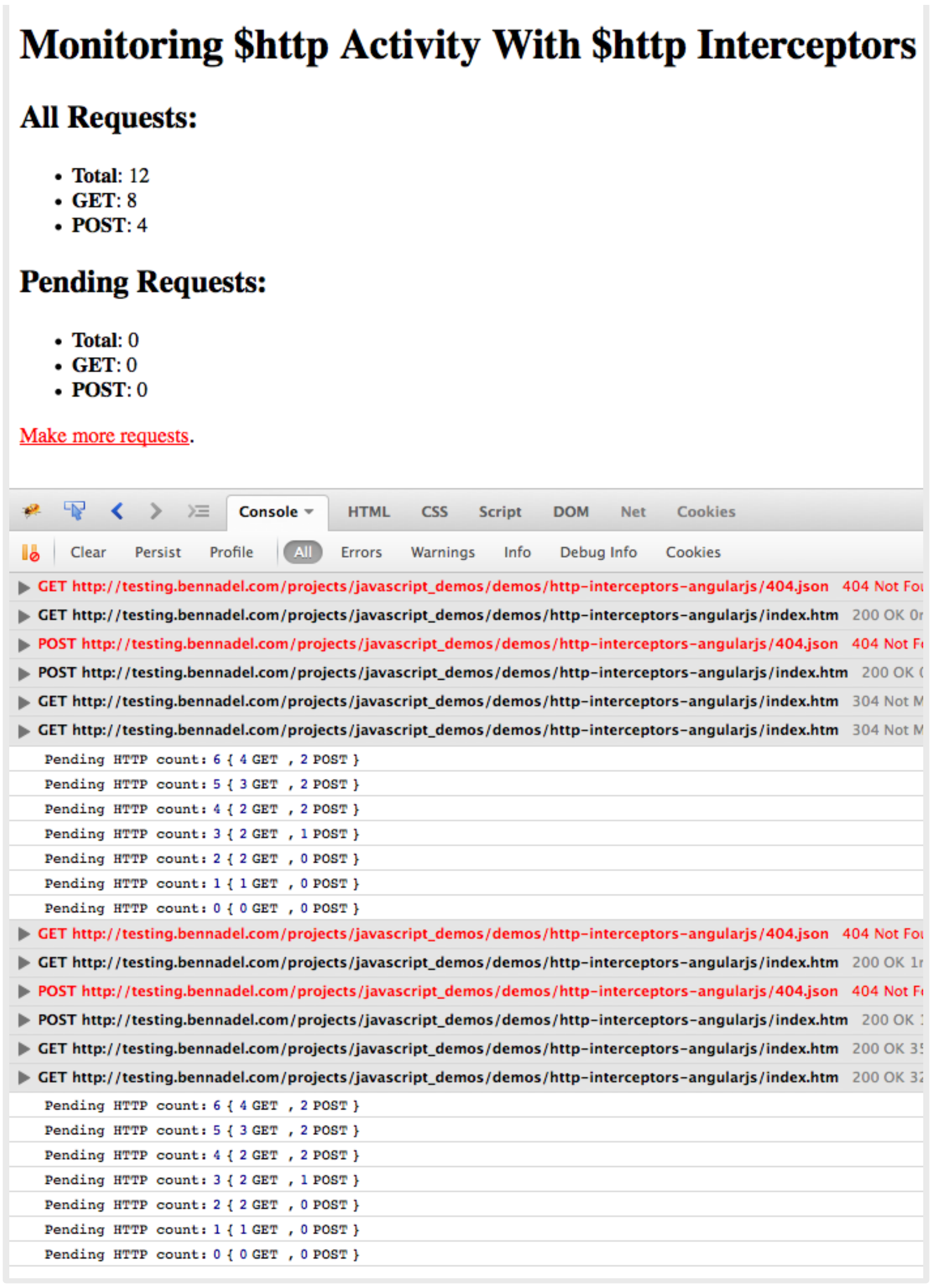
```
<!doctype html>
<html ng-app="Demo">
<head>
    <meta charset="utf-8" />

    <title>
        Monitoring $http Activity With $http Interceptors In AngularJS
    </title>

    <link rel="stylesheet" type="text/css" href="./demo.css"></link>
</head>
<body ng-controller="AppController">

    <h1>
        Monitoring $http Activity With $http Interceptors In AngularJS
    </h1>


    <h2>
        All Requests:
    </h2>

    <ul>
        <li>
            <strong>Total</strong>: {{ trafficCop.total.all }}
        </li>
        <li>
            <strong>GET</strong>: {{ trafficCop.total.get }}
        </li>
        <li>
            <strong>POST</strong>: {{ trafficCop.total.post }}
        </li>
    </ul>


    <h2>
        Pending Requests:
    </h2>

    <ul>
        <li>
            <strong>Total</strong>: {{ trafficCop.pending.all }}
        </li>
        <li>
            <strong>GET</strong>: {{ trafficCop.pending.get }}
        </li>
        <li>
            <strong>POST</strong>: {{ trafficCop.pending.post }}
        </li>
    </ul>

    <p>
        <a ng-click="makeRequests()">Make more requests</a>.
    </p>


    <!-- Load scripts. -->
    <script type="text/javascript" src="../../vendor/angularjs/angular-1.3.8.min.js"></script>
    <script type="text/javascript">

        // Create an application module for our demo.
        var app = angular.module( "Demo", [] );


        // -------------------------------------------------- //
        // -------------------------------------------------- //
        // I control the root of the application.
        app.controller(
            "AppController",
            function setupController( $scope, $http, trafficCop ) {

                // Attach the traffic cop directly to the scope so we can more easily
                // output the totals (rather than having to set up a watcher to pipe the
                // totals out of the service and into the scope for little-to-no benefit).
                $scope.trafficCop = trafficCop;

                // We can now watch the trafficCop service to see when there are pending
                // HTTP requests that we're waiting for.
                $scope.$watch(
                    function calculateModelValue() {

                        return( trafficCop.pending.all );

                    },
                    function handleModelChange( count ) {

                        console.log(
                            "Pending HTTP count:", count,
                            "{",
                                trafficCop.pending.get, "GET ,",
                                trafficCop.pending.post, "POST",
                            "}"
                        );

                    }
                );

                // Initiate some HTTP traffic to observe.
                initiateRequests();


                // ---
                // PUBLIC METHODS.
                // ---


                // I initiate some more HTTP traffic.
                $scope.makeRequests = function() {

                    initiateRequests();

                };


                // ---
                // PRIVATE METHODS.
                // ---


                // I spawn HTTP requests.
                function initiateRequests() {

                    $http({
                        method: "get",
                        url: "./404.json"
                    });

                    $http({
                        method: "get",
                        url: "./index.htm"
                    });

                    $http({
                        method: "post",
                        url: "./404.json"
                    });

                    $http({
                        method: "post",
                        url: "./index.htm"
                    });

                    $http({
                        method: "get",
                        url: "./index.htm"
                    });

                    $http({
                        method: "get",
                        url: "./index.htm"
                    });
                }

            }
        );


        // -------------------------------------------------- //
        // -------------------------------------------------- //


        // I run during the application bootstrap and hook the $http activity into the
        // trafficCop service so we can monitor the activity.
        app.config(
            function setupConfig( $httpProvider ) {

                // Wire up the traffic cop interceptors. This method will be invoked with
                // full dependency-injection functionality.
                // --
                // NOTE: This approach has been available since AngularJS 1.1.4.
                $httpProvider.interceptors.push( interceptHttp );


                // We're going to TRY to track the outgoing and incoming HTTP requests.
                // I stress "TRY" because in a perfect world, this would be very easy
                // with the promise-based interceptor chain; but, the world of
                // interceptors and data transformations is a cruel she-beast. Any
                // interceptor may completely change the outgoing config or the incoming
                // response. As such, there's a limit to the accuracy we can provide.
                // That said, it is very unlikely that this will break; but, even so, I
                // have some work-arounds for unfortunate edge-cases.
                function interceptHttp( $q, trafficCop ) {

                    // Return the interceptor methods. They are all optional and get
                    // added to the underlying promise chain.
                    return({
                        request: request,
                        requestError: requestError,
                        response: response,
                        responseError: responseError
                    });


                    // ---
                    // PUBLIC METHODS.
                    // ---


                    // Intercept the request configuration.
                    function request( config ) {

                        // NOTE: We know that this config object will contain a method as
                        // this is the definition of the interceptor - it must accept a
                        // config object and return a config object.
                        trafficCop.startRequest( config.method );

                        // Pass-through original config object.
                        return( config );

                    }


                    // Intercept the failed request.
                    function requestError( rejection ) {

                        // At this point, we don't why the outgoing request was rejected.
                        // And, we may not have access to the config - the rejection may
                        // be an error object. As such, we'll just track this request as
                        // a "GET".
                        // --
                        // NOTE: We can't ignore this one since our responseError() would
                        // pick it up and we need to be able to even-out our counts.
                        trafficCop.startRequest( "get" );

                        // Pass-through the rejection.
                        return( $q.reject( rejection ) );

                    }


                    // Intercept the successful response.
                    function response( response ) {

                        trafficCop.endRequest( extractMethod( response ) );

                        // Pass-through the resolution.
                        return( response );
                    }


                    // Intercept the failed response.
                    function responseError( response ) {

                        trafficCop.endRequest( extractMethod( response ) );

                        // Pass-through the rejection.
                        return( $q.reject( response ) );

                    }


                    // ---
                    // PRIVATE METHODS.
                    // ---


                    // I attempt to extract the HTTP method from the given response. If
                    // another interceptor has altered the response (albeit a very
                    // unlikely occurrence), then we may not be able to access the config
                    // object or the the underlying method. If this fails, we return GET.
                    function extractMethod( response ) {

                        try {

                            return( response.config.method );

                        } catch ( error ) {

                            return( "get" );

                        }

                    }

                }

            }
        );


        // I keep track of the total number of HTTP requests that have been initiated
        // and completed in the application. I work in conjunction with an HTTP
        // interceptor that pipes data from the $http service into get/end methods.
        app.service(
            "trafficCop",
            function setupService() {

                // I keep track of the total number of HTTP requests that have been
                // initiated with the application.
                var total = {
                    all: 0,
                    get: 0,
                    post: 0,
                    delete: 0,
                    put: 0,
                    head: 0
                };

                // I keep track of the total number of HTTP requests that have been
                // initiated, but have not yet completed (ie, are still running).
                var pending = {
                    all: 0,
                    get: 0,
                    post: 0,
                    delete: 0,
                    put: 0,
                    head: 0
                };

                // Return the public API.
                return({
                    pending: pending,
                    total: total,
                    endRequest: endRequest,
                    startRequest: startRequest,
                });


                // ---
                // PUBLIC METHODS.
                // ---
```

```
// I stop tracking the given HTTP request.
function endRequest( httpMethod ) {

    httpMethod = normalizedHttpMethod( httpMethod );

    pending.all--;
    pending[ httpMethod ]--;

    // EDGE CASE: In the unlikely event that the interceptors were not
    // able to obtain the config object; or, the method was changed after
    // our interceptor reached it, there's a chance that our numbers will
    // be off. In such a case, we want to try to redistribute negative
    // counts onto other properties.
    if ( pending[ httpMethod ] < 0 ) {

        redistributePendingCounts( httpMethod );

    }

}


// I start tracking the given HTTP request.
function startRequest( httpMethod ) {

    httpMethod = normalizedHttpMethod( httpMethod );

    total.all++;
    total[ httpMethod ]++;

    pending.all++;
    pending[ httpMethod ]++;

}


// ---
// PRIVATE METHODS.
// ---


// I make sure the given HTTP method is recognizable. If it's not, it is
// converted to "get" for consumption.
function normalizedHttpMethod( httpMethod ) {

    httpMethod = ( httpMethod || "" ).toLowerCase();

    switch ( httpMethod ) {
        case "get":
        case "post":
        case "delete":
        case "put":
        case "head":
            return( httpMethod );
        break;
    }

    return( "get" );

}


// I attempt to redistribute an [unexpected] negative count to other
// non-negative counts. The HTTP methods are iterated in likelihood of
// execution. And, while this isn't an exact science, it will normalize
// after all HTTP requests have finished processing.
function redistributePendingCounts( negativeMethod ) {

    var overflow = Math.abs( pending[ negativeMethod ] );

    pending[ negativeMethod ] = 0;

    // List in likely order of precedence in the application.
    var methods = [ "get", "post", "delete", "put", "head" ];

    // Trickle the overflow across the list of methods.
    for ( var i = 0 ; i < methods.length ; i++ ) {

        var method = methods[ i ];

        if ( overflow && pending[ method ] ) {

            pending[ method ] -= overflow;

            if ( pending[ method ] < 0 ) {

                overflow = Math.abs( pending[ method ] );
```

```
                                pending[ method ] = 0;

                            } else {

                                overflow = 0;

                            }

                        }

                    }

                }

            }
        );

    </script>

</body>
</html>
```

As you can see, the bulk of the code deals with the "what if" situations. What if the request is rejected? What if the response doesn't have a config object? What if the response config method doesn't match a pending method? In reality, none of these "what ifs" would ever occur. But, the mechanism provided by the promise chain means that they "could." And, I thought it would make the exploration more interesting.

And, when we run the above code, we get the following console output:



Monitoring $http Activity With $http Interceptors

All Requests:

- **Total**: 12
- **GET**: 8
- **POST**: 4

Pending Requests:

- **Total**: 0
- **GET**: 0
- **POST**: 0

Make more requests.

```
▶ GET http://testing.bennadel.com/projects/javascript_demos/demos/http-interceptors-angularjs/404.json  404 Not Fou
▶ GET http://testing.bennadel.com/projects/javascript_demos/demos/http-interceptors-angularjs/index.htm  200 OK 0
▶ POST http://testing.bennadel.com/projects/javascript_demos/demos/http-interceptors-angularjs/404.json  404 Not F
▶ POST http://testing.bennadel.com/projects/javascript_demos/demos/http-interceptors-angularjs/index.htm  200 OK
▶ GET http://testing.bennadel.com/projects/javascript_demos/demos/http-interceptors-angularjs/index.htm  304 Not M
▶ GET http://testing.bennadel.com/projects/javascript_demos/demos/http-interceptors-angularjs/index.htm  304 Not M
  Pending HTTP count: 6 { 4 GET , 2 POST }
  Pending HTTP count: 5 { 3 GET , 2 POST }
  Pending HTTP count: 4 { 2 GET , 2 POST }
  Pending HTTP count: 3 { 2 GET , 1 POST }
  Pending HTTP count: 2 { 2 GET , 0 POST }
  Pending HTTP count: 1 { 1 GET , 0 POST }
  Pending HTTP count: 0 { 0 GET , 0 POST }
▶ GET http://testing.bennadel.com/projects/javascript_demos/demos/http-interceptors-angularjs/404.json  404 Not Fou
▶ GET http://testing.bennadel.com/projects/javascript_demos/demos/http-interceptors-angularjs/index.htm  200 OK 1
▶ POST http://testing.bennadel.com/projects/javascript_demos/demos/http-interceptors-angularjs/404.json  404 Not F
▶ POST http://testing.bennadel.com/projects/javascript_demos/demos/http-interceptors-angularjs/index.htm  200 OK
▶ GET http://testing.bennadel.com/projects/javascript_demos/demos/http-interceptors-angularjs/index.htm  200 OK 3
▶ GET http://testing.bennadel.com/projects/javascript_demos/demos/http-interceptors-angularjs/index.htm  200 OK 3
  Pending HTTP count: 6 { 4 GET , 2 POST }
  Pending HTTP count: 5 { 3 GET , 2 POST }
  Pending HTTP count: 4 { 2 GET , 2 POST }
  Pending HTTP count: 3 { 2 GET , 1 POST }
  Pending HTTP count: 2 { 2 GET , 0 POST }
  Pending HTTP count: 1 { 1 GET , 0 POST }
  Pending HTTP count: 0 { 0 GET , 0 POST }
```

The more I dig into AngularJS, the more I learn to love Promises. And, with the way that AngularJS wraps $http requests in a promise chain, it gives us an opportunity to track outgoing and incoming HTTP activity.

Tweet This   Deep thoughts by @BenNadel - Monitoring $http Activity With $http Interceptors In AngularJS

B N

Enjoyed This? You Might Also Enjoy Reading:

- Using Module.decorator() In AngularJS 1.4
- Fun With Emoticons And Service Providers In AngularJS
- AngularJS Will Parse JSON Payloads In Non-2xx HTTP Responses
- ng-Template Requests Are Affected By $http Interceptors In AngularJS

B N

Looking For A New Job ?

- JavaScript/Ruby Mid-Senior Engineer at ListenLoop
- ColdFusion Application Developer / Portland Oregon at DealerPeak
- Web Applications Developer at University of California, Davis
- Advanced ColdFusion Developer at HD Web Studio
- ColdFusion Developer / Programmer at Blue Tangerine Solutions

100% of job board revenue is donated to Kiva. *Loans that change lives* — Find out more »

B N

Anne-Laure
Feb 5, 2015 at 3:02 AM
1 Comments

If you add an object to config in the request interceptor, you will find it again in the response config... That's very useful to track which request is which.
I use it to store a promise that I resolve in the response interceptor, so I can automatically queue requests (my server is not really configured to handle async requests but fast enough so that the user usually doesn't notice, saves me some headaches)

BTW I love your articles on Angular, I always learn a lot.

Reply to this Comment

Ben Nadel
Feb 6, 2015 at 3:50 PM
12,304 Comments

@Anne-Laure,

Super interesting - queuing up requests. I would have never thought of that. I am wondering what kind of server you have that you need to do that. Just interested.

In practice, altering the config should be fine. But, in *theory*, another interceptor that executes after yours could... *theoretically* replace the entire config object before the request even goes out. Obviously, that will never happen; but, there's nothing about promises NOR the documentation that said that it "shouldn't". In fact, the documentation for $http says that you can replace the config object entirely.

Obviously, we're just talking philosophy at this point, but I wish the docs had a strong opinion on what a valid use-case was. Then at least we could point-fingers :P

Thanks for the kind words - glad you like my exploration :D

Reply to this Comment

Brian Swartzfager
Feb 24, 2015 at 1:45 PM
27 Comments

My first use case for using interceptors was when I wanted to add a timestamp URL parameter for each GET request made for retrieving data (GET requests for HTML content would be unaffected) to circumvent some of the caching I was seeing in IE 8 at the time.

What I realized later was that I needed a way to "turn off" that behavior during unit testing, otherwise my $httpBackend GET request mocks wouldn't work without a timestamped URL (doable with regex, but a bit of a pain). So I always make sure I inject some object into the interceptors whose state conditionally affects what does and doesn't happen within the interceptor function.

Reply to this Comment

Vignesh
Apr 8, 2015 at 8:52 AM
1 Comments

@Ben,
Thanks for the idea of queuing request, I never thought of that kind of things can be done. But..

How the modified config data is sent back in the response, should we need to capture it and attach it to the response from server side or is there any magic I don't understand.??

Reply to this Comment

You — Get Out Of My Dreams, Get Into My Comments

Live in the Now

Name:	
Email:	(I keep this private)
Website:	
Comment:	

☑ Subscribe to comments.

Comment Etiquette: Please do not post spam. Please keep the comments on-topic. Please **do not post unrelated questions** or **large chunks of code**. And, above all, please be nice to each other - we're trying to have a good conversation here.

SUBMIT COMMENT

About Ben Nadel

I am the co-founder and lead engineer at InVision App, Inc — the world's leading prototyping, collaboration & workflow platform. I also rock out in JavaScript and ColdFusion 24x7 and I dream about promise resolving asynchronously.

GitHub | Twitter | LinkedIn | Google+ | Facebook