The Solution Optimist

Code refactors and Technical solutions for Web and Mobile

Enhancing AngularJS Logging using Decorators

October 7, 2013 by Burleson Thomas

ANGULARJS JAVASCRIPT REQUIREJS

Let's explore how to use Decorators to enhance AngularJS logging and supercharge the $log service. AngularJS has a great hidden feature `$provider.decorator()` that allows developers to **intercept** services and **substitute**, **monitor**, or **modify** features of those *intercepted* services. The *decorator* feature is not deliberately hidden... rather it is masked by so many other great AngularJS features. In this article, I will introduce the Decorator and show how to incrementally add functionality to the **$log** service... with practically no changes to your custom services and controllers.

Introducing `$provide.decorator()`

This interception process occurs when the service is constructed and can be used for:

- AngularJS built-in services: $log, $animate, $http, etc.
- Custom services: $twitter, $facebook, authenticator, etc.

In fact `angular-mocks.js` uses the `decorator()` to add

- `flush()` behaviors to $timeout,
- `respond()` behaviors to $httpBackend, and
- `flushNext()` behaviors to $animate

Since we are adding or changing behaviors at service construction, I like to say that `decorator()` allows us to **inject** custom behaviors. So, let's explore how you can use `decorator()` to **enhance** the AngularJS `$log` service with extra functionality.

> **Before** you continue reading this article, I highly recommend that you first read the Dependency Injection using RequireJS and AngularJS tutorial; since many of the examples use RequireJS `define()` and dependency injection.

Presenting the AngularJS `$log`

AngularJS has a built-in service **$log** that is very useful for logging debug and error messages to a console. Using this injected service, developers can easily monitor application workflows, confirm call sequences, etc. And since it is such a common useful mechanism, developers often complain about wanting more features.

> I posit that AngularJS developers should **NEVER** directly use `console.log()` to log debugger messages. Use `$log` instead...

Before we talk about adding more features, let's first look at standard $log usages. For our current purposes, I will use a Demo application which displays a login dialog to the user and provides mock authentication services. Below is a snapshot of the Demo application.

Here is an example use of the normal (un-enhanced) **$log** service within a mock Authenticator service class:

```
// ***********************************************
// bootstrap.js
// ***********************************************

(function()
{
  "use strict";

    /**
     * Mock Authenticator with promise-returning API
     */
    var Authenticator = function( session, $q, $log)
        {
                /**
                 * Mock login() service for authenticatator.
                 * @returns {Deferred.promise|*}
                 */
```

```
        var login = function(username, password)
            {
                var dfd      = $q.defer(),
                    errorMsg = "Bad credentials. Please use

                $log.debug( supplant("login( `{0}` )", [user

                if( (username != "admin") && (password != "s
                {
                    $log.debug( supplant( "login_onFault( `{
                    dfd.reject( errorMsg );
                }
                else
                {
                    $log.debug( supplant("login_onResult(use

                    session.sessionID = "SESSION_83732";
                    session.username = username;

                    dfd.resolve(session);
                }

                return dfd.promise;
            },
            /**
             * Mock service for logout.
             * @returns {Deferred.promise|*}
             */
            logout = function()
            {
                var dfd = $q.defer();

                $log.debug("logout()");

                session.sessionID = null;
                dfd.resolve();

                return dfd.promise;
            };

        return {
            login : login,
            logout: logout
        };
    },
    /**
     * LoginController used with the login template
     */
    LoginController = function( authenticator, $scope, $log
    {
        var onLogin = function()
        {
            $log.debug( supplant( "login( `{userName}` )", $
```

```
                authenticator
                    .login( $scope.userName, $scope.password )
                    .then(
                    function (result)
                    {
                        $log.debug( supplant( "login_onResult( `

                        $scope.hasError = false;
                        $scope.errorMessage = '';
                    },
                    function (fault)
                    {
                        $log.debug( supplant("login_onFault( `{0

                        $scope.hasError = true;
                        $scope.errorMessage = fault;
                    }
                );
            };

            $scope.login    = onLogin;
            $scope.userName = '';
            $scope.password = '';

            $scope.hasError = false;
            $scope.errorMessage = '';
        },
        appName = "myApp.Application";

    /**
     * Start the main application
     * We manually start this bootstrap process; since ng:app is
     * ( necessary to allow Loader splash pre-AngularJS activity
     */

    angular.module(     appName,            [ ]             )
           .value(      "session",          { sessionID : null
           .factory(    "authenticator",    ["session", "$q", "
           .controller( "LoginController",  [ "authenticator",

    angular.bootstrap( document.getElementsByTagName("body"), [

}());
```

When the Login form submits and `LoginController::login('Thomas Burleson', 'unknown')` is invoked, the $log output to the browser console will show:

```
login( `Thomas Burleson` )
login( `Thomas Burleson` )
login_onFault( `Bad credentials. Please use a username of 'admin' for mock logins !`
)
```

login_onFault(`Bad credentials. Please use a username of 'admin' for mock logins !`)

This console output shows that both the `LoginController` and the `Authenticator` are logging correctly to the console.

- But the console output is confusing!
- Which instance made which `$log.debug()` call?

To resolve this we *could* modify each $log.debug() to manually prepend the classname and even add a time stamp to each; since timestamps would also allow us to casually check our code flow.

But do NOT do this... that is a beginner's solution and is fuUgly.

Here is a sample output that we would *like* to see:

10:22:15:143 – LoginController::login(`Thomas Burleson`)
10:22:15:167 – Authenticator::login(`Thomas Burleson`)
10:22:15:250 – Authenticator::login_onFault(`Bad credentials. Please use a username of 'admin' for mock logins !`)
10:22:15:274 – LoginController::login_onFault(`Bad credentials. Please use a username of 'admin' for mock logins !`)

Before we start modifying *ALL* of our classes (like a beginning developer), let's pause and realize the the `$provide.decorator()` will allow us to centralize and hide all of the additional behavior and functionaltiy we want.

Using `$provider.decorator()`

Let's use `$provider.decorator()` to intercept `$log.debug()` calls and dynamically prepend timestamp information.

```
(function() {
  "use strict";

    angular
      .module( appName, [ ] )
      .config([ "$provide", function( $provide )
      {
            // Use the `decorator` solution to substitute or att
            // original service instance; @see angular-mocks for

            $provide.decorator( '$log', [ "$delegate", function(
```

```
        {
            // Save the original $log.debug()
            var debugFn = $delegate.debug;

            $delegate.debug = function( )
            {
              var args    = [].slice.call(arguments),
                  now     = DateTime.formattedNow();

              // Prepend timestamp
              args[0] = supplant("{0} - {1}", [ now, args[0]

              // Call the original with the output prepended
              debugFn.apply(null, args)
            };

            return $delegate;
        }]);
    }]);

})();
```

> In this case, we used a **head-hook** interceptor to build a prepend string and **then** called the original function. Other decorators could use **tail-hook** interceptors or replace interceptors. The versatility of choices is really powerful.

With the above enhancements, the console output would show something like this (red text indicates changes):

> 10:22:15:143 – login(`Thomas Burleson`)
> 10:22:15:167 – login(`Thomas Burleson`)
> 10:22:15:250 – login_onFault(`Bad credentials. Please use a username of 'admin' for mock logins !`)
> 10:22:15:274 – login_onFault(`Bad credentials. Please use a username of 'admin' for mock logins !`)

But we also wanted to easily include the classname for each method invoked!

- And did you notice that only `$log.debug()` was decorated?
- What about decorating the other functions such as `.error()`, `.warning()`, etc?

To achieve those additional feature is a little more complicated... but not as difficult as you might think!

Refactoring for Code Reuse

Before we extend the `LogEnhancer` with more functionality, let's first refactor our current code. We will refactor for easy reuse across multiple applications.

```
// ***********************************
// Module: bootstrap.js
// ***********************************

(function() {
  "use strict";

  var dependencies = [
    'angular',
    'myApp/logger/LogDecorator',
    'myApp/services/Authenticator',
    'myApp/controllers/LoginController'
  ];

  define( dependencies, function( angular, LogDecorator, Authent
  {
    // Configure the AngularJS HTML5 application `myApp`

    angular
      .module(     "myApp",           [ ]              )
      .config(     LogDecorator                        )
      .service(    "authenticator",   Authenticator    )
      .controller( "LoginController", LoginController  );

  });

})();
```

```
// ***************************************
// Module: myApp/logger/LogDecorator.js
// ***************************************

(function()
{
  "use strict";

  var dependencies = [
    'myApp/logger/LogEnhancer'
  ];

  define( dependencies, function( enchanceLoggerFn )
  {
      var LogDecorator = function( $provide )
          {
              // Register our $log decorator with AngularJS $pro
```

```
            $provide.decorator( '$log', [ "$delegate", functio
            {
                // NOTE: the LogEnchancer module returns a FUN
                //       All the details of how the `enchancem

                enchanceLoggerFn( $delegate );

                return $delegate;
            }]);
        };

    return [ "$provide", LogDecorator ];
  });

})();
```

```
// ****************************************
// Module: myApp/logger/LogEnhancer.js
// ****************************************

(function()
{
  "use strict";

  var dependencies = [
    'myApp/utils/DateTime',
    'myApp/utils/supplant'
  ];

  define( dependencies, function( DateTime, supplant )
  {
    var enchanceLogger = function( $log )
        {
          // Save the original $log.debug()
          var debugFn = $log.debug;

          $log.debug = function( )
          {
            var args    = [].slice.call(arguments),
                now     = DateTime.formattedNow();

                // prepend a timestamp to the original output me
                args[0] = supplant("{0} - {1}", [ now, args[0] ]

            // Call the original with the output prepended with
            debugFn.apply(null, args)
          };

          return $log;
        };
```

```
    return enchanceLogger;
  });

})();
```

> This package structuring (above) conforms to the Principle of Least Knowledge. This approach encapsulates all the actual details of how `$log` is enhanced in the the LogEnhancer module.

Now we are ready to continue adding functionality to the `LogEnhancer` ... functionality that will *inject* classNames and prepended them into the output messages.

Extending `LogEnhancer`

To easily support the `$log` functionality which prepends classNames to output messages, we need to allow `$log` to generate unique instances of itself; where each instance is registered with a specific classname.

Perhaps a code snippet will explain:

```
// ****************************************
// Module: myApp/controllers/LoginController.js
// ****************************************

define( dependencies, function( supplant )
{
  var LoginController = function( authenticator, $scope, $log )
    {
        $log = $log.getInstance( "LoginController" );

        // ...
    };

  return [ "authenticator", "$scope", "$log", LoginController ];

});
```

We used the `$log.getInstance()` method to return an object that looks like a $log but is NOT an actual AngularJS $log. How did we do this... our LogEnhancer decorated the AngularJS $log service and ADDED a `getInstance()` method to that service.

> The elegance to the above solution is that `$log` still passes the Duck test but internally knows how to prepend `LoginController` for any `$log` calls performed

in the LoginController class.

One line of code to support this feature within any class... that is pretty cool!

Let's itemize the set of features that we still need in order to fully-enable our LogEnhancer module:

- Intercept all the `$log` methods: log, info, warn, debug, error
- Ability to build output with tokenized messages and complex parameters (this will not be discussed in this article)
- Add `getInstance()` function with ability to generate custom instances with specific classNames

Using Partial Application within our `LogEnhancer`

We can use the partial application (sometimes known as Function Currying) technique to capture the specific log function that we want to intercept.
This technique allows us to use a generic handler that is partially applied to each `$log` function.

> Personally I love elegant tricks like these! Function currying is a technique that becomes a magic wand for developers.

```
// **********************************
// Module: myApp/logger/LogEnhancer.js
// **********************************

(function()
{
  "use strict";

  var dependencies = [
    'myApp/utils/DateTime'
    'myApp/utils/supplant'
  ];

  define( dependencies, function( DateTime, supplant )
  {
    var enchanceLogger = function( $log )
        {
                /**
                 * Partial application to pre-capture a logger f
                 */
            var prepareLogFn = function( logFn )
                {
                    /**
```

```
                * Invoke the specified `logFn<` with the su
                */
               var enhancedLogFn = function ( )
               {
                   var args = Array.prototype.slice.call(ar
                       now  = DateTime.formattedNow(),

                       // prepend a timestamp to the origin
                       args[0] = supplant("{0} - {1}", [ no

                   logFn.call( null,  supplant.apply( null,
               };

               // Special... only needed to support angular
               enhancedLogFn.logs = [ ];

               return enhancedLogFn;
           };

       $log.log   = prepareLogFn( $log.log );
       $log.info  = prepareLogFn( $log.info );
       $log.warn  = prepareLogFn( $log.warn );
       $log.debug = prepareLogFn( $log.debug );
       $log.error = prepareLogFn( $log.error );

       return $log;
   };

  return enchanceLogger;
});

})();
```

Notice that the `debugFn.call( … )` method also uses a `supplant()` method to transform any tokenized content into a final output string:

```
var user = { who:"Thomas Burleson"", email="ThomasBurleson@gmail

// This should output:
// A warning message for `Thomas Burleson` will be sent to `Thom

$log.warn( "A warning message for `{who}` will be sent to `{emai
```

So not only have we intercepted the $log functions to prepend a timestamp, we also **supercharged** those functions to support tokenized strings.

Adding `$log.getInstance()`

Finally we need to implement the `getInstance()` method and publish it as part of the AngularJS `$log` service.

```
// *********************************
// Module: myApp/logger/LogEnhancer.js
// *********************************
(function()
{
  "use strict";

  var dependencies = [
            'myApp/utils/DateTime',
            'myApp/utils/supplant'
      ];

  define( dependencies, function( DateTime, supplant )
  {
    var enhanceLogger = function( $log )
        {
                /**
                 * Capture the original $log functions; for use
                 */
              var _$log = (function( $log )
                  {
                      return {
                          log   : $log.log,
                          info  : $log.info,
                          warn  : $log.warn,
                          debug : $log.debug,
                          error : $log.error
                      };
                  })( $log ),
                  /**
                   * Partial application to pre-capture a logger f
                   */
                  prepareLogFn = function( logFn, className )
                  {
                      /**
                       * Invoke the specified `logFn` with the sup
                       */
                      var enhancedLogFn = function ( )
                      {
                          var args = Array.prototype.slice.call(ar
                              now  = DateTime.formattedNow();

                          // prepend a timestamp and optional
                          args[0] = supplant("{0} - {1}{2}", [

                          logFn.call( null,  supplant.apply( null,
                      };

                      // Special... only needed to support angular
```

```
                enhancedLogFn.logs = [ ];

                return enhancedLogFn;
            },
            /**
             * Support to generate class-specific logger ins
             *
             * @param name
             * @returns Object wrapper facade to $log
             */
            getInstance = function( className )
            {
                className = ( className !== undefined ) ? cl

                return {
                    log   : prepareLogFn( _$log.log,    clas
                    info  : prepareLogFn( _$log.info,   clas
                    warn  : prepareLogFn( _$log.warn,   clas
                    debug : prepareLogFn( _$log.debug,  clas
                    error : prepareLogFn( _$log.error,  clas
                };
            };

        $log.log   = prepareLogFn( $log.log );
        $log.info  = prepareLogFn( $log.info );
        $log.warn  = prepareLogFn( $log.warn );
        $log.debug = prepareLogFn( $log.debug );
        $log.error = prepareLogFn( $log.error );

        // Add special method to AngularJS $log
        $log.getInstance = getInstance;

        return $log;
    };

  return enhanceLogger;
});

})();
```

We modified our partial application function `prepareLogFn()` to accept an optional `className` argument. And we implemented a `getInstance()` method that builds a special facade object with the same API as the original `$log` service.

Finally, we need to modify our original example code to use `$log.getInstance()`:

```
(function()
{
  "use strict";

  var dependencies = [
```

```
    'myApp/utils/supplant'
  ];

  define( dependencies, function( supplant )
  {
    var Authenticator = function( session, $q, $log)
    {
        $log = $log.getInstance( "Authenticator" );

        // … other code here
    };

        return ["session", "$q", "$log", Authenticator];

  });

  define( dependencies, function( supplant )
  {
    var LoginController = function( authenticator, $scope, $log
        {
      $log = $log.getInstance( "LoginController" );

      var onLogin = function()
          {
              $log.debug( "login( `{userName}` )", $scope );

              // … other code here
          };

          // … other code here
        };

    return [ "authenticator", "$scope", "$log", LoginController

  });

}());
```

Did you notice the additional change shown on line 30 of the source above? The `$log.debug()` has internalized all `supplant()` features into the LogEnhancer… so now an additional $log feature supports logging calls with a tokenized string and an hashmap (or array).

Then our browser console output would be:

10:22:15:143 – LoginController::login(`Thomas Burleson`)
10:22:15:167 – Authenticator::login(`Thomas Burleson`)
10:22:15:250 – Authenticator::login_onFault(`Bad credentials. Please use a username of 'admin' for mock logins !`)
10:22:15:274 – LoginController::login_onFault(`Bad credentials. Please use a

username of ‘admin’ for mock logins !`)

Summary

This is just one example of how Decorators can be used to add or modify behavior in AngularJS applications. And the LogEnhancer could be further extended with features to:

- Output to a custom application console… for remote, customer reporting
- Color coding and grouping of log messages by category; @see Chrome Dev Tools
- Logging client-side errors to the remote server

I am sure there are many more elegant applications of this technique. Don’t forget to share with the AngularJS community if you have a cool decorator!

Resources

I have created a public GitHub repository with the source and examples used within this tutorial.

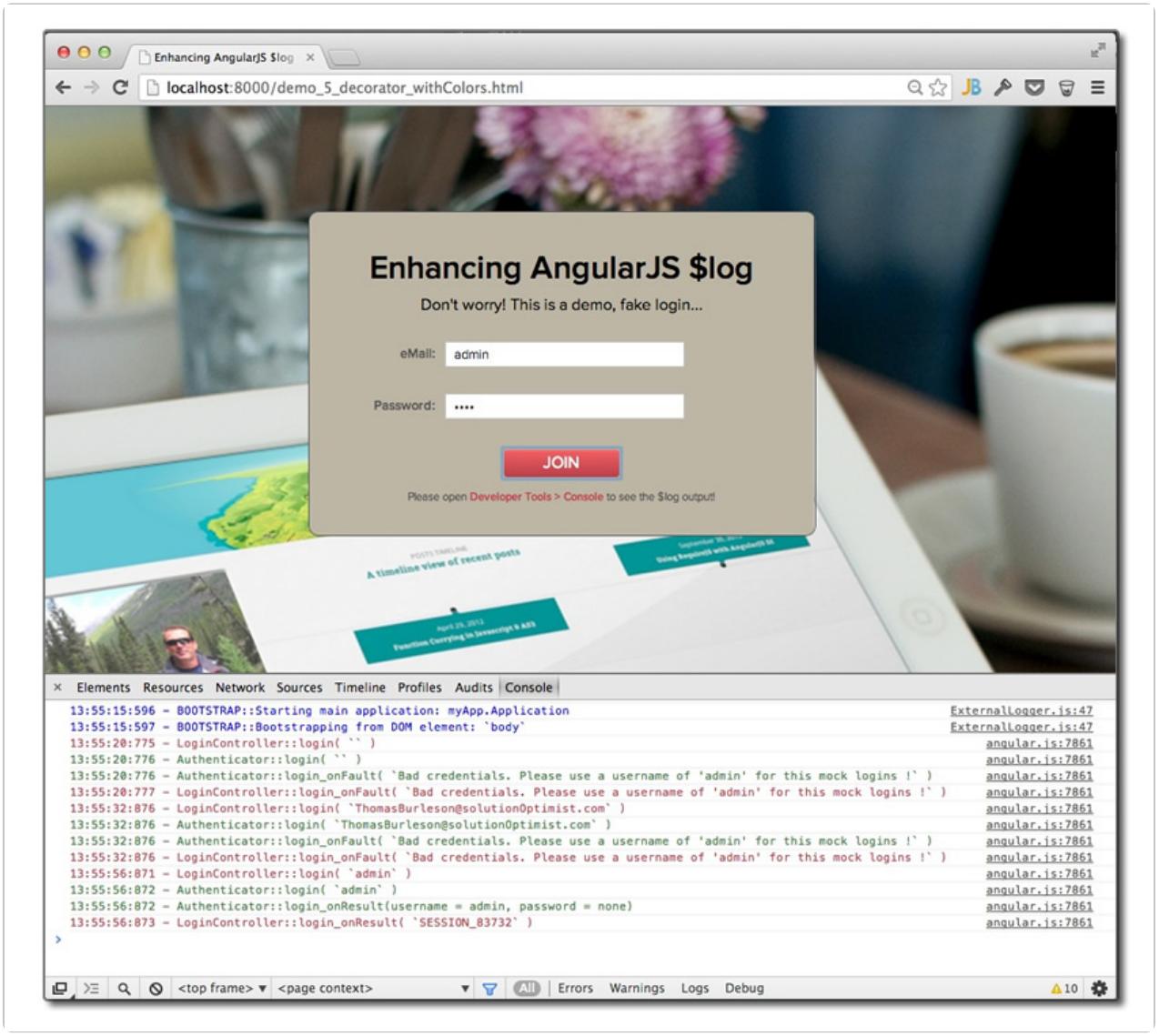
Source Code Demos

Extra Credit

As an end-of-show teaser, do you know how I achieved the results shown below?

* Do you see the console coloring?
* Do you see the logging output that shows activity before AngularJS starts ?

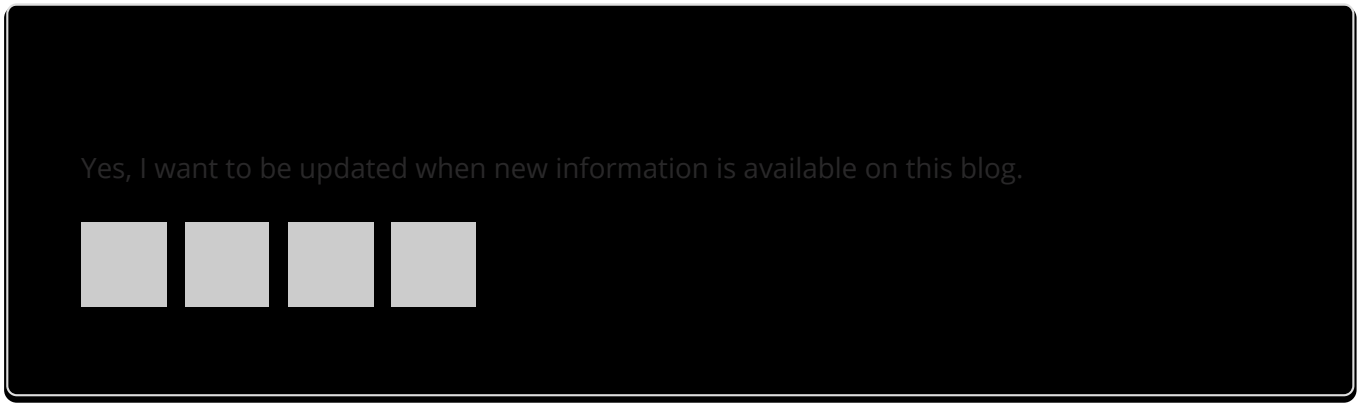
I integrated features to support the Chrome Dev tools functionality for console logging with color. But how did I integrated that into the LogDecorator ?



To see how the `LogDecorator` and other features have been extended to support the above features, see the code in Demo #5.

Tags: angularjs, behaviors, console, decorators, log, logging

View all posts by Burleson Thomas →



Related Posts:

- Scaffolding AngularJS SPA(s)
- AngularJS ng-Conf 2014
- GitHub Tricks: Upload Images & Live Demos
- Flattening Promise Chains

← **Dependency Injection using RequireJS & AngularJS**

Partial Applications with Javascript & AS3 →

ENHANCING ANGULARJS LOGGING USING DECORATORS

14 Responses

"I posit that AngularJS developers should NEVER directly use console.log() to log debugger messages. Use $log instead..."
Why?

sj
August 5, 2014 at 4:13 am #

Reply

Hey I used your tips and created a module. check it out
https://github.com/lwhiteley/AngularLogExtender

Layton
February 2, 2014 at 11:02 am #

Reply

A great decorator on $log. Liked the way getInstantance has been created and used (that reminds me of the singleton pattern)

Sagar
January 15, 2014 at 6:19 am #

Reply

@Sagar,
Thx. Just to follow-up that `$log.getInstance()` is not a singleton pattern. `getInstance()` is a factory method used to get a special, distinct *locked* instance of $log... with *context* already assigned (where the context is the name of the class that instance is being used).

Thomas Burleson
January 21, 2014 at 10:38 am #

Reply

Dear lord, so much 'generic' code just to pre-pend a classname and a date to a log statement. There's nothing elegant about bloated hyper nested code like this; code is elegant exactly when it *doesn't* require all kinds of magic wand trickery, is easy to understand and maintainable. From an academic point of view I understand the appeal to see 'under the head' and verify a solution like this works, but in a real world application however, I would use the following code instead:

```
// configure getInstance function
yourAppModule.run(['$log', function($log) {
$log.getInstance = function(context) {
return {
log : enhanceLogging($log.log, context),
info : enhanceLogging($log.info, context),
warn : enhanceLogging($log.warn, context),
debug : enhanceLogging($log.debug, context),
error : enhanceLogging($log.error, context)
};
};

function enhanceLogging(loggingFunc, context) {
return function() {
var modifiedArguments = [].slice.call(arguments);
modifiedArguments[0] = [moment().format("dddd h:mm:ss a") + '::[' + context + ']>
'] + modifiedArguments[0];
loggingFunc.apply(null, modifiedArguments);
};
}
}]);

// usage:
var logger = $log.getInstance('My Context');
logger.info('you can't handle the truth');
// Monday 9:37:18 pm::[My Context]> you can't handle the truth
```

I used momentjs for datetime formatting in this example. It's also easy to move this code away into a requirejs module, which would simply have one function: .enhanceLogger($log) and then perform exactly the above code.

Benny Bottema
December 23, 2013 at 2:39 pm #

Reply

> There's nothing elegant about bloated hyper nested code like this

That was gratuitous. Remove all the comments and think away requirejs for a second, it's not as entirely as bloated as it looks on first sight. Still 'quite' a bit of wiring going on…

Benny Bottema
December 23, 2013 at 3:09 pm #

Reply

@Benny,
You are right that the code base could be MUCH leaner; using RequireJS appears to add `bloat`. In real-world projects (with RequireJS) the above demonstrates the packaging features.

1) Using angular.run() will not work because I need to enhance the $log methods BEFORE any injection of $log occurs… hence the decorator approach.
2) And yes, the `enhance` log could be leaner also… with use we discovered more and more features that we wanted on our project(s). The above code reflects that evolution.

So many people read but do not comment. Thanks for your feedback and debate ;-).

Burleson Thomas
December 28, 2013 at 7:57 pm #

Reply

Brilliant, thanks !

Bretto
October 18, 2013 at 6:54 pm #

Reply

TRACKBACKS/PINGBACKS

1. **AngularJS-Learning | Nisar Khan** - May 1, 2014
   [...] Enhancing AngularJS Logging using Decorators [...]

2. **Using the AngularJS $logProvider | RealEyes Media** - April 16, 2014
   [...] version 1.2.16 was used as a basis for this article. Thanks to Burleson Thomas and this post for the extension [...]

3. **Using Angular's $log provider** - April 15, 2014

[...] version 1.2.16 was used as a basis for this article. Thanks to Burleson Thomas and this post for the extension [...]

4. **Enhancing $log in AngularJs the simple way | Benny's Blog** - December 23, 2013
   [...] I found was a rather... extensive, a solution utilizing Angular's built in support for decorators. Ofcourse, the fact it actually works is great and it's a nice insight into some advanced [...]

5. **AngularJS Highlights: Week Ending 20 October 2013 | SyntaxSpectrum** - October 20, 2013
   [...] Enhancing AngularJS Logging using Decorators [...]

6. **AngularJS Highlights: Week Ending 13 October 2013 | SyntaxSpectrum** - October 13, 2013
   [...] Enhancing AngularJS $log using Decorators [...]

LEAVE A REPLY

Name (Required) Email (Required) Website

Comment

Submit Comment

RECENT POSTS

Scaffolding AngularJS SPA(s)

AngularJS ng-Conf 2014

GitHub Tricks: Upload Images & Live Demos

Flattening Promise Chains

Partial Applications with Javascript & AS3

SUBSCRIBE

Yes, I want to be updated when new information is available on this blog.