The Solution Optimist

Code refactors and Technical solutions for Web and Mobile

Dependency Injection using RequireJS & AngularJS

September 30, 2013 by Thomas Burleson

ANGULARJS HTML5 REQUIREJS

How is RequireJS dependency injection (DI) used with AngularJS DI?

Developers use RequireJS to asynchronously load Javascript and establish package/import dependencies. And the AngularJS MVC framework is used to architect HTML5 SPAs (single-page applications) using dependency injection (DI) and databindings. Developers working on non-trivial HTMl5 applications quickly realize the importance of using RequireJS with AngularJS. Unfortunately, many of those developers easily become confused when they realize that both frameworks supposedly perform dependency injection (DI).

And if AngularJS and RequireJS both support DI, then developers quickly start to ask these types of questions:

1. How is RequireJS DI used with AngularJS ?
2. Is RequireJS dependency injection the same as AngularJS dependency injection ?
3. How are RequireJS injected values used to prepare AngularJS injected instances ?

Here is a video of a presentation I gave at ng-Conf 2014:

For those who want to continue reading my responses to the questions above, let's first explore how Javascript objects/classes can be packaged into discrete, manageable files.

The Module Pattern

The Module pattern is a common Javascript coding pattern widely used when developers want discrete JS files/modules.

All of the code that runs inside the function lives in a closure, which provides privacy and state throughout the lifetime of our application. To avoid the use of implicit globals, the Module pattern can specify parameters to the anonymous function.

```
(function (define, angular) {

// ...

}(window.define, window.angular));
```

By passing globals as parameters to our anonymous function, we import them into our code, which is both clearer and faster than implied globals.

> Unfortunately, the V8 engine currently manifests Javascript breakpoint bugs with Module patterns and parameters. As such, parameters are **NOT** currently recommend for development purposes. In addition, since the parameters we would pass are in fact global (define, angular), while cleaner it's not a necessity to function properly.

Module Exports

Sometimes you don't just want to use globals, but you want to declare them. We can easily do this by exporting them, using the anonymous function's return value.

```
windows.utils = (function ()
{
    // Publish a `log()` utility function
    return {
        log : function( message ) {
           console.log( message );
        }
    };

}());
```

The example above shows how the anonymous function publishes a *global* `utils` registry with a `log()` function.

> Module Exports are not used within our project; as such approaches require explicit assignment to external variables or storage. Instead the **AMD** pattern is used to manage module exports and dependencies.

Asynchronous Module Definition Pattern

The AMD pattern allows developer to

- Define discrete JS modules...modules that may have 1..n external dependencies for other modules, and
- Load these modules asynchronously (e.g. at some later time)

Using the RequireJS AMD pattern, let's see how we could define a Logger class:

```
/**
* Log provides an abstract method to encapsulate logging message
* Filename: `myApp/utils/logger.js`
*/
var dependencies = [ ]; // list of JS file/Module names to be lo

define( dependencies, function()
{
    // Publish a `log()` utility function
    var logger = {
            log : function( message ) { console.log( message );
            debug : function( message ) { console.debug( message
        };
```

```
    return logger;

});
```

It is important to realize that the `define()` does **not** return a value. Rather the anonymous function specified **within** `define()` returns a value to RequireJS... a value that is registered/cached for later use/injection.

Developers can think of the above code as follows:

- Check if the dependencies are loaded (in this case none)
- Execute the anonymous function (usually done asynchronously after all the defines() have been loaded and their dependency tree has been constructed.
- Store the return value in a *definition* registry associated with the `myApp/utils/logger` key
- Inject the stored *value* into functions that declare `logger` as a dependency

Recall that the AMD pattern must return a value... a value that will be stored in the definition registry [for future injection]. The important thing to note is that value can be **anything**: object, string, array, or even a function.

The specific type of the value returned is entirely dependent on **how** or **what** the developer wants to inject into dependent definitions.

RequireJS AMD and Dependency Injection

In addition to the management of import dependencies and registering return values, the AMD pattern has one (1) other very important feature: **Dependency Injection**.

When the `logger` utility is needed within another module, you simply *define* a dependency to load the `logger` module and RequireJS will **inject** a reference to the `logger` value in the AMD definition registry.

> A definition is said to be loaded when the anonymous function [with its parameters] is invoked and a value is returned;

```
/**
* Greeter
* FileName: `myApp/Greeter.js`
*/
define( [ 'myApp/utils/logger' ], function( logger )
{
    // Publish Greeter instance
```

```
    return {
        welcomeUser : function( user ) {
            logger.debug( "Welcome back " + user );
        }
    };

});
```

Notice that the `logger` argument in the anonymous function above will now reference the `logger` instance that is stored in our definition registry... and this *value* is generated as the return value from the anonymous function in the `myApp/utils/logger` module itself.

Since the dependency chain specifies that the Greeter class cannot be *defined* until the `logger` definition has been loaded, this effectively means that RequireJS has **INJECTED** the logger value into the Greeter class (during runtime instantiation).

Fully-protected AMD Usages

Finally let's wrap the RequireJS AMD pattern within the Module pattern:

```
(function())
{
    var numGreetings = 0;
    var dependencies = [
        // tell RequireJS to inject the value returned from Logg
        'myApp/utils/logger'
    ];

    define( dependencies, function( logger )
    {
        // Publish Greeter instance
        return {
            welcomeUser : function( user )
            {
                numGreetings += 1;
                logger.debug( "Welcome back " + user );
            },
            reportGreetings : function ()
            {
                logger.debug( "Number of greetings issued = " +
            }
        };

    });

}( ));
```

The Module pattern usage (above) has enabled the developer to:

- Protect the internals of how the Greeter class is defined
- Encapsulate and manage the private global state `numGreetings`

> This approach of defining a RequireJS AMD pattern within a Module pattern is used within every class in my HTML5 project(s).

Developers should note, however, that the RequireJS AMD dependency injection (aka DI) is **NOT** an AngularJS injection process. RequireJS AMD DI is used only to inject functionality that is external and separate of any AngularJS instance constructions.

In subsequent sections, we will see how Angular DI differs and how both DIs can be used together in synergistic solutions.

AngularJS Dependency Injection

AngularJS provides a unique and powerful support for **dependency injection** that is distinct from RequireJS AMD dependency injections.

> Think of AngularJS DI as injecting `instances` and RequireJS DI as injecting `classes` or `constructors`

Similar to RequireJS, AngularJS provides mechanisms for building a registry of instances managed by keywords: `factory()`, `service()`, and `controller()`. All three (3) functions take:

- keyword: consider this the variable instance name; which is used to inject as a constructor parameter into other instance constructions.
- function: consider this the construction function.

```
angular
    .factory( "session", function()
    {
        return Session();
    })
    .service( "authenticator", function()
    {
        return new Authentication();
    })
    .controller( "loginController", function( session, authentic
    {
        return new LoginController( session, authenticator );
    });
```

Using a construction function allows AngularJS to internally defer invocation until the instance is actually needed.

But notice how the construction function for `loginController` requires the `session` and `authenticator` instances as function parameters ? These instances are injected into the function invocation by AngularJS when AngularJS instantiates an instance of the LoginController.

Using AngularJS Annotations for Injection

Let's reconsider the AngularJS registration of the `/myApp/controllers/LoginController.js` module as a controller:

```
(function()
{

    define( function()
    {
        var LoginController = function( session, authenticator )
        {
            // Publish instance with simple login() and logout()
            return {
                login : function( userName, password )
                {
                    authenticator.login( userName, password );
                },
                logout : function()
                {
                    authenticator.logout().then( function()
                    {
                        session.clear();
                    });
                }
            };
        };

        // Publish the constructor function
        return LoginController;
    });

}());
```

Here the module published the *expected* construction function... where the `LoginController` reference is actually a `Function` .

As I mentioned earlier, AngularJS injects parameter values as constructor arguments. But how does AngularJS know which instances/values to inject? JavaScript does not have annotations, and annotations are needed for dependency injection.

Well, you have to tell AngularJS the names of the instances to be injected. The following are all valid ways of annotating functions with injection arguments and are equivalent:

- inferred – Use the `Function.toString()` method to parse and determine the argument names. Note: *this only works if the code is not minified/obfuscated*
- annotated – Adding an `$inject` property to the function where the parameters are specified as elements in an array
- inline – as an array of injection names, where the last item of the array is the function to call

> Both the **annotated** and **inline** techniques also continue to work properly even when we use Javascript minification to obfuscate and compress our production application code. And for developers concerned about performance, the AngularJS engine actually scans for the annotation method in the following priority order: $inject, inline, inferred.

AngularJS ***inline*** support is an elegant technique where developers can use an array where each element of the array is a string whose names corresponds to an argument of the function to be invoked AND the last element is the actual constructor function. I personally call this technique a `constructor array`.

I find the built-in support for either a constructor function or an array is an incredibly powerful, ingenious solution. Here is how I use this inline, `constructor array` technique as part of an AngularJS configuration:

```
angular
    .controller( "LoginController", [ "session", "authenticator"
    {
        return new LoginController( session, authenticator );
    }]);
```

This works... but developers should immediately note two (2) concerns with this solution:

1. The AngularJS registration of the “LoginController” code is much messier using the **inline**, construction array.
2. If the LoginController implementations changes and the constructor arguments are different, developers must remember to also modify/update the construction array above.

> As mentioned earlier, Angular supports the **$inject** annotation technique (that also supports minification issue). Instead of using constructor arrays, developers can attach a `$inject` property to the constructor function. The value of this property is an array of string names of the parameters used as constructor arguments:

```
LoginController.$inject = ["session", "authenticator"];
```

Note: This technique has been deprecated; since we want to use **inline**, Constructor Arrays with RequireJS DI.

Using Constructor Arrays within AngularJS

We can easily refactor our code to both resolve the above two issues and support both annotation and minification... by using **inline**, constructor `Arrays` instead of constructor `Functions`. The *trick* is to have RequireJS inject the **inline**, constructor array into the module where we configure AngularJS.

The code below will demonstrate. First, let's update the LoginController module to publish an **inline**, **constructor array** instead of a function:

```
(function()
{

    define( function()
    {
        var LoginController = function( session, authenticator )
        {
            // Publish instance with simple login() and logout()
            return {
            login : function( userName, password ) { /***/ },
            logout : function() { /***/ }
            };
        };

        // Publish the constructor/construction array
        return [ "session", "authenticator", LoginController];
    });

}());
```

Now our AngularJS startup configuration code is great simplified:

```
(function()
{

    var dependencies = [
        'myApp/model/Session',
        'myApp/services/Authenticator',
        'myApp/controllers/LoginController'
    ];

    define( dependencies, function( Session, Authenticator, Logi
```

```
    {
        angular
            .factory( "session", Session )
            .service( "authenticator", Authenticator )
            .controller( "loginController", LoginController );
    });

}());
```

Developers should note several interesting aspects to the above code.

Here we have leveraged the power of the RequireJS AMD return values which are injected in as parameters to the `define()` anonymous function. These parameters look like class references... but are actually AngularJS **constructor arrays**. And the result configuration code is **SIGNIFICANTLY** easier to maintain. In fact, now the configuration code indicates nothing about annotations or minification-safety or argument-name specifications... it just works!

The AMD injected parameters are then used within AngularJS to support dynamic construction and subsequent dependency injection. AngularJS will detect that constructor arrays are specified, determine the appropriate DI values that must be injected as constructor arguments, and then invoke the constructor function. And finally the returned value from the constructor function is stored within an Angular registry; by its variable name.

> I must emphasize that when the constructor function is invoked, the published or `return` value MUST still be an appropriate value or object instance. This value/instance will be cached by AngularJS for subsequent future injections.

Note that the registration keywords [used in the sample code above] `session`, `authenticator`, and `loginController` are lowercase because their respective values are **instances** created from the invocation of their constructor functions. In contrast, upper camel case names indicate types that are actually **classes**, **constructors**, or **functions**... type references that have been injected by RequireJS.

AMD Injection Considerations

As stated above, in most cases the RequireJS AMD dependency allows us to inject classes or `constructor arrays`. Sometimes, it is expedient to also have RequireJS inject a function that is NOT intended for internal AngularJS consumption.

As one example, let's consider a `supplant` function that provides features to build complex strings with tokenized parameters. In our example below, we want to build strings and log messages independent of the AngularJS $log mechanisms and processes.

So let's inject the `supplant` function [published in the myApp/utils/supplant.js Module]... along with our other classes: `Session`, `Authenticator`, `LoginController`:

```
(function()
{
    var dependencies = [
        'myApp/utils/supplant',
        'myApp/model/Session',
        'myApp/services/Authenticator',
        'myApp/controllers/LoginController'
    ];

    define( dependencies, function( supplant, Session, Authentic
    {
        console.log( supplant (
            "Added dependency `supplant()` from the {0} module."
            dependencies
        ));

        angular
            .factory( "session", Session )
            .service( "authenticator", Authenticator )
            .controller( "loginController", LoginController );
    });

}());
```

Here the `supplant` function is used only with the `console.log()` and is not used with AngularJS at all.

This simple example shows how easy it is to use RequireJS to inject both (a) functions for general use and (b) Classes (constructor functions or constructor arrays) for AngularJS instantiation and DI usages.

Recommended Module Coding Standards

A condensation of this entire exploration can be summarized as this:

> Think of AngularJS DI as injecting `instances` and RequireJS AMD DI as injecting `Classes` or `constructors`

When defining classes and packages, I suggest that the following heuristics should be considered universal HTML5 development standards:

- All classes/modules should have a root Module wrapper; without any *global imports*.
- Use RequireJS AMD to establish/define module dependencies and invocations orders

- Use RequireJS AMD to inject module values (eg Class, function, constructor arrays)
- If the Module values are to be used within AngularJS, then those modules should publish a **constructor array** value.

Tags: amd, angularjs, dependency injection, html5, javascript, modules, requirejs

Related Posts:

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

← Function Currying in Javascript & AS3

Enhancing AngularJS Logging using

Decorators →

DEPENDENCY INJECTION USING REQUIREJS & ANGULARJS

26 Responses

I think this is one of the most important info for me.

And i'm glad reading your article. But should remark on few
general things, The web site style is wonderful, the articles is really nice :
D. Good job, cheers

สักคิ้ว 3 มิติ
July 20, 2014 at 9:18 pm #

Reply

Hi Thomas,
This is a great article and I have been trying to use your approach for my next project. However, I am running into trouble while running e2e tests with Protractor js. The issue seems to be with the manual bootstrapping that needs to be done to load the application. It appears that Protractor does not work with a manually bootstrapped application. Do you have any fixes / workarounds for this?

Thanks
Aneesh

Aneesh
July 10, 2014 at 1:13 am #

Reply

Hi Thomas,

thank you for this great article !! I've been trying to make these two guys (AngularJS and RequireJS) work together but I couldn't come up with a solution that satisfies me !
Your approach is very clear and I like the way you handle the asynchronous

loading with head.js.
I looked at the sources on GitHub and especially at the build process and there are some things which are not very clear to me.
It seems that you never use the vendorFiles object which is defined in the build.config.js. As it is, all the content form the vendors directories is copied in the build directory... so where and how would you make use of these vendorFiles data ? In a copy task or concat task ? or something like that ?

Do you have any feedback about head.js performances or any caveats it could have ?
Thank your for your time and yours explanations !

Boris.

Boris
June 12, 2014 at 9:59 am #

Reply

The `vendorFiles` is not used in Gruntfile.js build processes; that is a breadcrumb from another project where the full deployment would concatenate the vendor files and deploy the entire SPA to a target **deploy** directory. To make the Grunt script more understandable, I removed the concatenation of the vendor files; only `copy:build_vendorjs` is used.

Please be sure that you use my fork of`Head.js`... as my fork supports the `notify` callback:

```
head.notify( function(name, size, loaded, total) {
// for splash preloader
});
```

See Angular-ZZa-Mean with it's boot.js for details.

Thomas Burleson
June 12, 2014 at 10:39 am #

Reply

Thank you! I learned a great deal from this post and your presentation, abbreviated though it was

You seem to strongly recommend “All classes/modules should have a root Module wrapper” and I am very curious why. In a recent project, I wrote literally 100’s of AMD modules (for Dojo, sadly) but until I read your article I never thought of wrapping my defines in a module pattern. It seems to add verbosity without any benefit. But I feel sure I am missing something.

Again, many thanks for a very lucid explanation.

Mark Florence
May 5, 2014 at 12:00 pm #

Reply

If you are using AMD-style modules, then I recommend the Module pattern wrapper. @see http://briancray.com/posts/javascript-module-pattern
BTW, I am less convinced of the benefits of this pattern, however, if you are using CommonJS format for NodeJS server-tier code.

Burleson Thomas
May 5, 2014 at 12:12 pm #

Reply

Hi, here is the approach I use : http://codrspace.com/thaiat/angularjs-requirejs/

let me know what u think...

Avi Haiat
February 26, 2014 at 12:15 pm #

Reply

Is there a working exmple of this? I’m running into errors and wonder what I’m missing. I found the code for the Quizzler application, but was confused as to why in addition to requirejs there is a head.js loading dependencies.

Brendan Fagan
January 28, 2014 at 10:59 am #

Reply

Quizzler has a live demo of the code using RequireJS @ Live Demo.
I use HeadJS for two reasons:

1) Allows me to parallel load my vendor libraries and others scripts before AngularJS or RequireJS are used…
This is important if I want to show a Splash loader while AngularJS is initializing.

2) Allows me to NOT use RequireJS's internal "lazy loading" feature to load my class files on-demand.
For production I want to concatenate/uglify my AMD scripts into a single script and load all my code before AngularJS or require() are triggered,

If these are not important for your needs, then you do not need to use HeadJS… sorry for the confusion.

Thomas Burleson
January 29, 2014 at 10:38 am #

Reply

Thanks for the detailed explanation! The approach seems very elegant, but I'm still left with two questions though:

1. how would you create a class or constructor function that makes use of an Angular service (e.g. `$http` or `$q`)?
2. How would you test one of these **AMD** classes?

David Q
January 22, 2014 at 6:47 pm #

Reply

Simply define a function that specifies $http and $log as arguments; see sample @QuizDelegate; lines 42 and 115.
Make sure you use RequireJS with your AngularJS test code also; @see Quizzler Testing for samples.

Here is a compressed example of a constructor function implemented within a `define( )`

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

    define([
            'utils/supplant'
        ],
        function ( supplant )
        {
            var QuizDelegate = function ( $http, $q, $lo
                {
                   $log = $log.getInstance( "QuizDelegat

                };

            return [ "$http", "$q", "$log", QuizDelegate
        });

}( define ));
```

Thomas Burleson
January 25, 2014 at 11:46 am #

Reply

I wish I read this article a week ago, it would save me quite some time to come up to the same conclusions
Very nice article, thanks.

In my company I use Typescript to generate AMD modules so my code is not polluted with AMD definition ugliness:

```
// ************************
// users_module.ts
// ************************

var _usersModule = angular.module('Users', [
]);
```

```
import userPref = require('users/user-preferences-service')
import userSrv = require('users/user-service');
import userProf = require('users/user-profile-controller');

_usersModule.service("userPreferenceService", userPref.User
_usersModule.service("userService", userSrv.UserService);
_usersModule.controller('UserProfileController', userProf.U

// ************************
// user_controller.ts:
// ************************

import userSrv = require('users/user-service');

export class UserProfileController {

    static $inject = ['$scope', 'userService'];

    constructor(private $scope, private userService: userSr
    ///....
    };
}
```

konstantin raev
October 15, 2013 at 6:14 pm #

Reply

I like how detailed article about this problem you wrote. But what about moving that angular.controller(), angular.factory() constructs into AMD modules itself? To me, it is much cleaner if in Controller module we define service/factory modules as AMD deps. Can this stack support this?

srigi
October 10, 2013 at 2:16 am #

Reply

I agree that organizing your code into dependent-modules can help with scaling and maintenance.

Personally, I do not organize my code into **modules-by-type** (e.g. controllers modules, services, modules, etc.) Instead, I organize **modules-by-context**: authentication module, twitter module, routes modules, etc. There is one (1) exception to that hueristic: I do like to create a distinct, centralized `Dataservices` module for all my delegates.

Notice the code snippet in my reply to @Damian.

```
/**
 * Specify main application dependencies...
 *
 */
var dependencies = [
        'myApp/modules/DataServices',
        'myApp/modules/Routes',
    ];

require( dependencies, function ( DataServices, Routes )
{
    angular.module( 'myCustomApp.Application',  [ "ngRou

});
```

The module `myCustomApp.Application` has dependencies on the modules `Routes` and `DataServices`; each of which internally has registered services, factories, and controllers.

Thomas Burleson
October 10, 2013 at 9:17 am #

Reply

I think you're wanting to wrap the AMD pattern of require around angular itself to handle dependencies.

You can do this with shims. Something like:

"`javascript
shim:{
'angular': {
deps: [],
exports: 'angular'
}
}

```
//then in any define block:
define([
'angular',
'something'
],
function ( angular, something )
```

Unfortunately angular submodules: angular-route, angular-sanitize aren't AMD supported. So you'll have to get creative on how to load those using similar shims, likely and extra define block around the main app.

Kris
April 7, 2014 at 9:01 pm #

Reply

Interesting idea... I think, however, of AngularJS as a **framework** library.

And I think that frameworks are expected to be globally available and – as such – do not need a dependency management.
I use only use RequireJS `define()` to both load-on-demand my own custom Class(es) and to establish a dependency tree.

Burleson Thomas
April 15, 2014 at 10:36 am #

Reply

Great write up, this would have came in handy for me when I attempted this a few days ago.
This article, however, does not speak to how your main module definition would look. For a large (even mid-sized) app with many modules and vendor libraries I saw better gains using bundling over requirejs (granted I didn't run require's optimization) as angular seems to want to know all its module dependencies up front. Thoughts? (I'm pretty green on this so I may just be doing it wrong).

Damian Reeves
October 4, 2013 at 6:38 am #

Reply

@Damian

You are correct that Grunt will bundle or concatenate your JS files.
And you are correct that the current versions of AngularJS does not support lazy, on-demand loading... so all JS code that your AngularJS application will use must be loaded prior to bootstrapping.

> Dan Wahlin wrote an interesting article Dynamically Loading Controllers and Views with AngularJS and RequireJS that provides support for on-demand loading. And while occasionally requested... lazy, on-demand loading is not yet part of the AngularJS core feature set.

Using RequireJS AMD, however, will eliminate the need to define the concatenation **order** for those JS files. And you can configure RequireJS to load all the JS modules before you configure and bootstrap/start your AngularJS application.

```
/**
 * Specify main application dependencies...
 * one of which is the Authentication module.
 *
 * @type {Array}
 */
var dependencies = [
        'myApp/modules/DataServices',
        'myApp/modules/Routes',
        'myApp/controller/LoginController'
    ],

    appName = 'myCustomApp.Application';

/**
 * Now let's start our AngularJS app...
 * which uses RequireJS to load the sxm packages and cod
 *
 */
require( dependencies, function ( DataServices, Routes,
{
    /**
     * Start the main application
     *
     * We manually start this bootstrap process; since n
     * ( necessary to allow Loader splash pre-AngularJS
     */

    angular.module( appName,  [ "ngRoute", Routes, DataS
            .controller( "LoginController", LoginControll
```

```
    // Since I am not using ng-app in my HTML, I must ma
    // the application using the application name config

    angular.bootstrap( $("html"), [ appName ]);

});
```

And you can still use Grunt to bundle all the RequireJS define() modules into a single JS file… and the above code will still work.

Thomas Burleson
October 4, 2013 at 8:33 am #

Reply

What about integrating ngmin into the mix?

Wouldn't that help remove the need to maintain the constructor arrays during development? Just run the code through ngmin before minimizing and it can help add the annotations purely based off the function arguments.

Nathan
October 3, 2013 at 9:13 am #

Reply

Certainly you could use NgMin or Grunt (or other tools) to achieve similar purposes.

What I was trying to articulate, however, was how dependency injection works between RequireJS and AngularJS.
The **inline**, constructor array solution was an add-on solution that makes your code VERY clean and terse.
Best of all, the code tersity simplifies the maintenance of your code.

Thomas Burleson
October 3, 2013 at 12:11 pm #

Reply

Really cool post I enjoyed it very much thank you
So now we have a way to integrate elegantly AngularJS and RequireJS, but I think and please correct me if i am wrong, Angular bootstrapping mechanism requires that all the dependencies are present at initialisation and the benefit of RequireJS is the lazy loading of the dependencies. So how do we integrate the lazy loading of AngularJS Modules via RequireJS in the running AngularJS App ?

Bretto
October 4, 2013 at 1:03 am #

Reply

@Bretto,

I use `ng-cloak` and do NOT use `ng-app` in my main HTML file, since I want to **manually** bootstrap my AngularJS application… after all my modules are loaded and ready. See my reply to @Damian for a code snippet.

Thomas Burleson
October 4, 2013 at 8:40 am #

Reply

Yes I have been doing the same with ng-cloak and manual bootstrapping… (I have actually used ng-include to dynamically at runtime load other ng-app in the “main” or “master” ng-app I am more curious about the loading of AngularJS Modules via RequireJS in a running AngularJS App… So loading additional modules at runtime in an already running app… like a swf loading a swf Have you been doing any R&D on that front ?

Bretto
October 6, 2013 at 3:34 am #

TRACKBACKS/PINGBACKS

1. **AngularJS-Learning | Nisar Khan** - May 1, 2014
   [...] Using RequireJS with AngularJS DI [...]

2. **AngularJS Highlights: Week Ending 6 October 2013 | SyntaxSpectrum** - October 8, 2013
   [...] Dependency Injection using RequireJS & AngularJS [...]

3. **Enhance $log using AngularJS Decorators | The Solution Optimist** - October 7, 2013
   [...] 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 [...]

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.