Jason Dobry

Husband. Father. Software Engineer. Open Source Hacker. Always Learning.

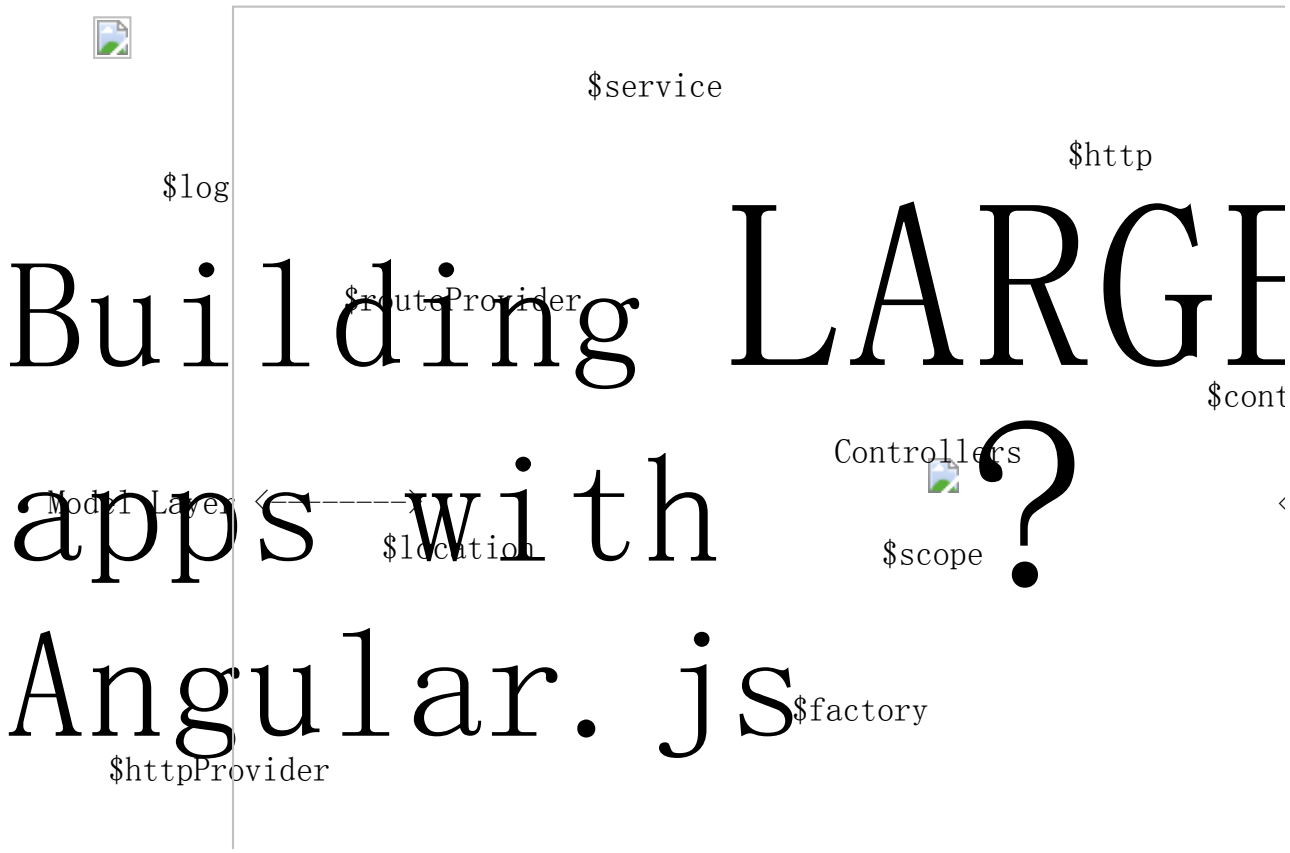
Pseudobry

01 Aug 2013 on javascript | angular.js

Building large apps with Angular.js

This post came out of my presentation at a UtahJS meetup. You can even go see my slides.

New to AngularJS? Confused by directive mumbo-jumbo? Giddy over Angular's magic and your new productivity? Jaded by gotchas you didn't see coming? Enjoying the zen of Angular?



Building LARGE apps with Angular.js ?

Whatever your experience with Angular and whatever size of project you're working on, there will come a day when you encounter Angular in the wild being used on a large project. Does that sound like a dream come true or are you still wary? Either way, you will inevitably run into a problem that Angular does not solve for you. Will you survive?

If you've needed to:
- Organize your files - Name your files - Lazy-load your code - Serialize application state to the URL - Derive application state from the URL - Manage a Model layer - Do smart caching - Work with a RESTful interface (for real) - Use a 3rd-party lib with Angular

Then you've already met the beast.

Angular is not the framework to end all frameworks, however, never before have I derived so much pleasure from web development. Not with NodeJs, not with NoSQL, not with BackboneJS. To misquote @ryanflorence: "I'm Sick of the Web: [Angular]'s Healing Balm".

My first few weeks with AngularJS I was like a kid on Christmas morning, unwrapping presents as fast as I could and running around showing everybody. Then my new race car broke after 10 loops around the track, I found that none of my previous toys played nice with my new ones, and I realized I would need a Ph.D. to understand my new book "Directives for Dummies". After a few months of hate/love, things started to click. Directives became my best friends, Angular best practices got me up and running, and I began derive real satisfaction from my work.

Now before you think it's all fun and games, here are some points to consider as you embark on your journey building large apps with Angular:

Convention vs Configuration

When a framework embraces conventions, developers familiar with the framework can move from project to project with minimal "ramp up" time in order to become familiar with each project. It basically comes down to familiarizing oneself with the project's business domain. The purpose of conventions is to improve:

"Conventionless" frameworks result in fragmentation, meaning that one developer's use of a framework can differ wildly from another's. Example: BackboneJS. An excellent tool in and of itself—but decidedly lacking in enforced conventions. Have you ever seen two Backbone developers do something the exact same way? They might look at each others' project and have no idea what is going on. Freedom has its downsides.

In regard to convention, AngularJS is a little on the light side. AngularJS puts itself forth as a "MVW" framework loosely based on a traditional MVC framework. Angular Controllers setup and add behavior to `$scope` objects, and Angular's templates and two-binding make for a nifty View layer, but Angular has virtually no opinion when it comes to your Model layer. You can do whatever you want.

The path to successfully building a large, fast, and stable application with AngularJS lies in finding and following conventions for you and your dev team. The AngularJS documentation doesn't help much with that, but many developers are becoming experienced with Angular and

have shared the conventions that have helped them find success. You can't commit to writing your large application with AngularJS without paying attention to the mailing list, the GitHub project, and the opinions of Angular's core contributors.

Model Layer

Large applications often have a lot of data to deal with, so designing a solid Model layer is essential to building and maintaining a stable Angular application. Angular Services provide a useful tool for separating your Model layer or business logic from your View and Controller layers. `$rootScope` and `$scope` are useful as the Model layer for small applications, but can quickly become a mass of messy prototypal inheritance chains that may or may not work the way you think they will. For example:

```
$scope.fruit = 'banana';
var $newScope = $scope.$new();

$scope.fruit; // 'banana'
$newScope.fruit; // 'banana'

$newScope.fruit = 'apple';
$newScope.fruit; // 'apple'

$scope.fruit; // 'banana' // Did you think it would be
```

```
'apple'?
```

In a large AngularJS application it is important to have a source of truth for your data. Keep your data in your Angular Services and use your Controllers to expose your data to your views.

Your services own your data:

```
app.service('MyService', function ($http, $q,
$angularCacheFactory) {
    var _dataCache = $angularCacheFactory('dataCache', {
        maxAge: 3600000 // items expire after an hour
    });
    /**
     * @class MyService
     */
    return {
        manipulateData: function (input) {
            var output;
            // do something with the data
            return output;
        },

        getDataById: function (id) {
            var deferred = $q.defer();
            if (_dataCache.get(id)) {
                deferred.resolve(_dataCache.get(id));
            } else {
```

```
                // Get the data from the server and
populate cache
            }
            return deferred.promise;
        }
    };
});
```

Your controllers and directives consume your data:

```
$scope.$watch(function () {
    return MyDataService.getMyData();
}, function () {
    $scope.myData = MyDataService.getMyData();
});
```

Ultimately, you'll want your Model layer to support the needs of a large client-side application. Many tools exist. The point is to not let Angular do your Model layer for you, because you'll be disappointed. Look into something like $resource, restangular, BreezeJS, or even using BackboneJS Models/Collections. Treat your data with respect.

File Organization

Found at the heart of many an argument is the issue of file organization and naming. Where do we put

everything? Do we organize by type, form, function, or feature? PascalCase or camelCase? Some frameworks make this decision for you by enforcing a convention. Ruby on Rails, for example, expects to find your controllers in `myApp/app/controller/` and your views in `myApp/app/views/`. This leaves little room for argument or ambiguity. Some frameworks also use naming conventions to find your files. Here is an example taken from the EmberJS website:

```
App.Router.map(function() {
    this.route("about", { path: "/about" });
    this.route("favorites", { path: "/favs" });
});
```

When the user visits `/`, Ember.js will render the `index` template. Visiting `/about` renders the `about` template, and `/favs` renders the `favorites` template.
Note that you can leave off the path if it is the same as the route name. In this case, the following is equivalent to the above example:

```
App.Router.map(function() {
    this.route("about");
    this.route("favorites", { path: "/favs" });
});
```

The AngularJS framework itself does not provide any such conventions. It's all up to you. Below are some

possible solutions.

Monolithic files

The angular-seed project recommends something like the following for project file organization:

```
partials/
    home.html
    login.html
    users.html
    orders.html
js/
    controllers.js
    directives.js
    filters.js
    services.js
    app.js
```

This works great for small, focused applications, but quickly becomes untenable with large applications.

Monolithic folders

A step in the right direction, but can quickly become ridiculous. Imagine searching through the `controllers/` folder when it contains 100+ files, trying to find the code for some obscure piece of functionality that your product manager wants to tweak.

```
js/
    controllers/
        homeController.js
        loginController.js
    directives/
        usersDirective.js
        ordersDirective.js
    filters/
    services/
        userService.js
        orderService.js
        loginService.js
partials/
    home.html
    login.html
    users.html
    orders.html
app.js
```

Organize by feature

I personally like to be able to visually map what I see on the screen to the source files that make it happen. If I am looking at the "users" page then I expect to be able to find all of the code for it in the "users" package/folder. Modifying the "users" page now means making changes to files in one package/folder, instead of performing "shotgun surgery".

```
orders/
    directives/
        orders.html
        ordersDirective.js
    services/
        orderService.js
users/
    directives/
        users.html
        usersDirective.js
    services/
        userService.js
home/
    controllers/
        home.html
        homeController.js
        login.html
        loginController.js
    services/
        loginService.js
shared/
    services/
        i18nService.js
    filters/
        i18nFilter.js
app.js
```

Modules and lazy

loading application components

Angular Modules are good for nothing...so far. Except for loading 3rd-party angular code into your app and mocking during testing. Other than that, there is no reason to use more than one module in your app. Misko said of multiple modules: "[you] should group by view since views will be lazy loaded in near future". I'd love for someone on the Angular team to enlighten us on what their plans are for that.

Angular does not officially support lazy loading of components. While different workarounds exist, they can result in undefined behavior because Angular expects everything to be loaded during the bootstrap phase. Kudos to Ifeanyi on a lazy-loading solution.

Here's the basic idea:

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

app.config(function ($controllerProvider,
$compileProvider, $filterProvider, $provide,
$animationProvider) {
    // save references to the providers
    app.lazy = {
        controller: $controllerProvider.register,
```

```
        directive: $compileProvider.directive,
        filter: $filterProvider.register,
        factory: $provide.factory,
        service: $provide.service,
        animation: $animationProvider.register
    };

    // define routes, etc.
});
```

Lazy-load dependencies on a per-route basis:

```
$routeProvider.when('/items', {
    templateUrl: 'partials/items.html',
    resolve: {
        load: ['$q', '$rootScope', function ($q,
$rootScope) {
            var deferred = $q.defer();
            // At this point, use whatever mechanism you
want
            // in order to lazy load dependencies. e.g.
require.js
            // In this case, "itemsController" won't be
loaded
            // until the user hits the '/items' route
            require(['itemsController'], function () {
                $rootScope.$apply(function () {
                    deferred.resolve();
                });
```

```
            });
            return deferred.promise;
        }]
    }
});
```

itemsController.js

```
// reference the saved controller provider
app.lazy.controller('ItemsController', function ($scope)
{
    // define ItemsController like normal
});
```

items.html

```
<section data-ng-controller="ItemsController">;
    <!-- a bunch of awesome html here -->;
</section>;
```

Understand the $scope life cycle

$scope is the lifeblood of AngularJS. You cannot build a large application with AngularJS and avoid the intricacies of the $scope life-cycle. The $rootScope of your application and all of its child $scopes contain

$watch expressions. There is a $watch expression for everything you put onto $scope. This is the magic of Angular's two-way data-binding.

Angular continuously runs what's called a $digest loop that checks each $watch expression to see if the value being watched has changed. If it has changed, than Angular executes all of the listeners for that variable. This can happen many times in a second. It is important for your watch expressions to be fast and idempotent. Performance can become an issue if you attach functions as $watch expressions to your $scope. These functions will be executed many times and can slow down performance. If any of the listeners fired by changing $watch expressions create yet more changes, which kick off more $digest loops which end up firing *yet more* listeners, then performance can be adversely affected.

3rd-party integration with AngularJS

If you've spent any real amount of time with Angular than you've probably run into problems trying to make 3rd-party libraries work with Angular. If you're building a large application, chances are you need to integrate several such libraries with your app. Angular thrives on its $digest loop, giving you the two-way data-binding

magic. 3rd-party libraries don't know what Angular does. If your 3rd-party library changes something in the DOM, or returns you a value via an AJAX call, Angular doesn't know about it and continues doing its thing, none the wiser. The keys to 3rd-party integration with Angular are $scope.$apply(), $scope.$evalAsync, $q.when(), and $timeout.

When something happens with a 3rd-party library, then you'll probably need to manually kick off a $digest loop via `$scope.$apply()`. This way Angular can react to whatever your 3rd-party library did. Angular's $q promise library is a very useful tool for asynchronously resolving the result of whatever your 3rd-party is going to do. Doing something like:

```
$scope.$apply(function () {
    // do something with 3rd-party lib
});
```

or

```
$timeout(function () {
    // do something with 3rd-party lib
}, 0);
```

$timeout causes Angular to automatically do a `$scope.$apply()` after the anonymous function is executed (which isn't executed until the *next* browser event loop).

Learning AngularJS is a roller coaster ride, but ultimately results in increased productivity and flexibility in building large applications.

Jason Dobry

Husband. Father. Software Engineer. Open Source Hacker. Always Learning.

Utah, USA

Share this post

23 Comments Pseudobry dongguangming

Recommended 6 Share Sort by Best

Join the discussion…

Latoof • 2 years ago

Thank you. Many best practices !

1 • Reply • Share ›

Spencer Allen Gardner • 2 years ago

Nice post! Good high-level stuff for creating quality software.

1 • Reply • Share ›

Ansuman Bebarta • 7 months ago

Awesome and very informative. I learned a lot. Thanks

• Reply • Share ›

Juan Biscaia • a year ago

Hi, great article, helped me a lot, but still I have a question about this great amount of files (controllers, directives, services, etc...). Following your recommendations in folder structure i'll have about 20 to 30 files in my project and i'm using the angular-seed way of routing and working (with the ng-view). My question is: I will need to include ALL my files (controllers, filters, directives, etc...) in the index.html file using the <script> tag?

I'm ok with that, but this just looks a bit messy for me...

• Reply • Share ›

Jason Dobry Mod ↗ Juan Biscaia • a year ago

No, you should certainly have some sort of build step that concatenates your many files into a single file. This way the browser only has to make a single request for all of your javascript. Take a look at Grunt or Gulp for a build process.

▲ | ▼ • Reply • Share ›

Juan Biscaia ↗ Jason Dobry • a year ago

Ty for the answer! I'll check Grunt.

▲ | ▼ • Reply • Share ›

Ivo Santiago • a year ago

Not sure if I would go with the $apply solution for 3rd-party integration. I think if you wrap it on a service it would be better. Except for that all the practices are great! Thanks Jason

▲ | ▼ • Reply • Share ›

Jason Dobry Mod ↗ Ivo Santiago • a year ago

Wrapping 3rd-party (not angular plugins) libraries in a service is a good idea, but optional. Calling $apply around 3rd-party library calls and callbacks is not only a good idea, but often required for it to even work. Often the only thing the service wrapper does is call $apply for you, so you don't have to do it elsewhere in your application. Take the $timeout service as a perfect example. It wraps the "3rd-party" setTimeout library with a service, and calls $apply for you, because otherwise Angular wouldn't know to re-run its digests when the timeout completes. You don't have to use $timeout, but if you use setTimeout, you must call $scope.$apply, or it won't work.

My post covers Angular gotchas, with the 3rd-party library $apply situation being an important one.

▲ | ▼ • Reply • Share ›

Daniel Quinn • a year ago

Thanks for posting this. Its great to see best practices from others implementing large scale Angular application. My team really struggles implementing the model layer for our applications. Specifically with how they integrate with the service API. Our models our complex and sometimes represented by deeply nested graphs. We've tried building normalized(flat, e.g GET:/team returns {name:"Cleveland Browns"} vs denormalized GET:/team returns {name:"Cleveland Browns", players: [{name: "Johnny Manziel"}]}) REST endpoints to grab all the data for maximum flexibility, but thats leads to a really chatty app. We also have issues with persisting these large objects. With normalized endpoints each item in the graph has to be persisted to different endpoints. However

they all need to happen successfully otherwise the application will be out of sync. There doesn't seem to be a good way to handle that yet in the front-end. Your information here is helpful, but do you know of any good resources that dive into best practices for handling complex models in an SPA like those build with AngularJS?

• Reply • Share ›

Jason Dobry Mod ➚ Daniel Quinn • a year ago

I'm currently struggling with complex model layers myself. I am currently working on an Ember-Data-like analog for Angular called angular-data (http://angular-data.codetrain....). I'm currently in the design phase for implementing nested RESTful models (relations/associations). It's an interesting and difficult problem. BreezeJS is a very robust (albeit complex) solution–not for the faint of heart. Restangular is better than ngResource, but still insufficient in my opinion.

• Reply • Share ›

Daniel Quinn ➚ Jason Dobry • a year ago

Yes, it is a very interesting and difficult problem. Also, one that I'd assume most teams using AngularJS would encounter. Am I alone in being surprised this isn't being talked about more, or are others just slogging their way thru it too?

Your project looks interesting. I'm looking forward to digging into it.

• Reply • Share ›

Jason Dobry Mod ➚ Daniel Quinn • a year ago

The angular team is discussing the model layer problem, but a solution coming from their end wouldn't happen until angular 2.x.x. See https://docs.google.com/docume...

• Reply • Share ›

Yoorek • a year ago

'MyService' - shouldn't it be app.factory instead app.service? I mean with service you need provide constructor function and instance is created with 'new' as i remember. When you return object i think it should be factory... Or i forgot sth in messy factory-service-provider chaos?

• Reply • Share ›

Jason Dobry Mod ➚ Yoorek • a year ago

Angular services demystified: https://gist.github.com/geddsk...

• Reply • Share ›

Yoorek ➚ Jason Dobry • a year ago

So? The sample confirms exactly what I wrote - for 'service" you should provide constructor function, for factory - return object or function... your code is still not 'right'... I mean - your 'MyService' should be factory not service.

▲ | ▼ • Reply • Share ›

Jason Dobry Mod ➚ Yoorek • a year ago

My example works the same as the service example in the gist, with the added benefit of private variable scope.

1 ▲ | ▼ • Reply • Share ›

Yoorek ➚ Jason Dobry • a year ago

No it's not. This is gist:

```
//define a "class", will be instanciated
app.service('monkey', function(){
this.name = "joe";
this.rand = Math.random();
});
```

This is your example:

```
app.service('MyService', function ($http, $q,
$angularCacheFactory) {
var _dataCache =
$angularCacheFactory('dataCache', {
maxAge: 3600000 // items expire after an hour
});
/**
* @class MyService
*/
```

see more

▲ | ▼ • Reply • Share ›

Jason Dobry Mod ➚ Yoorek • a year ago

That's all very nice. Thank you for pointing out my mistakes.

▲ | ▼ • Reply • Share ›

Macha • 2 years ago

Hey, how do I write a filter? using the above?

▲ | ▼ • Reply • Share ›

Jason Dobry Mod ➚ Macha • 2 years ago

Try this: http://docs.angularjs.org/guid...

^ | v • Reply • Share ›

Macha ➚ Jason Dobry • 2 years ago

Thanks for the response.

The below is my DateObject.js Filter

```
define(['app'], function(app) {

app.lazy.filter('DateObject', [ function () {

console.log('DateObject FILTER')

return function(dateString) {

if(dateString == null) return;

var p = dateString.split(" ");

var d = p[0].split("-");

var t = p[1].split(":");

var date = new Date();
```

see more

^ | v • Reply • Share ›

Jason Dobry Mod ➚ Macha • 2 years ago