[2013.01.07]

The AngularJS documentation is great for getting started and for looking up specific API calls. However, it doesn't really tell you how to organize and manage your app as it grows to tens or hundreds of thousands of lines of code. I've collected here some of my observations and best practices for how to manage your sprawling application. First we'll take a look at organization, then move on to some tips on improving performance, and conclude with a brief summary on tools, servers, and build processes. While this post will focus on big apps in particular, there's a great video on YouTube from the December 2012 AngularJS meetup on best practices that's also worth checking out.

Don't Write a Huge App

The best advice about huge apps is not to make them. Write small, focused, modular parts, and progressively combine them into bigger things to make your app. (This advice brought to you by node.js hacker and all around cool dude @substack)

Organization

Probably the biggest question with large apps is where to put all of that code. On your toolbelt of organizational tools, you've got files, directories, modules, services, and controllers. For a quick overview of good AngularJS project structure, check out the Angular Seed on Github. However I'd like to go a bit more in-depth and offer some additional advice on project structure. Let's start with directories and work our way down the list.

Directories

This is the typical folder layout that I recommend:

```
root-app-folder
 ├─── index.html
 ├─── scripts
 │    ├─── controllers
 │    │    └─── main.js
 │    │    └─── ...
 │    ├─── directives
 │    │    └─── myDirective.js
 │    │    └─── ...
 │    ├─── filters
 │    │    └─── myFilter.js
 │    │    └─── ...
 │    ├─── services
 │    │    └─── myService.js
 │    │    └─── ...
 │    ├─── vendor
 │    │    ├─── angular.js
 │    │    ├─── angular.min.js
 │    │    ├─── es5-shim.min.js
 │    │    └─── json3.min.js
 │    └─── app.js
 ├─── styles
 │    └─── ...
 └─── views
      ├─── main.html
      └─── ...
```

As you add more files, it might make sense to create subdirectories to further organize controllers and services. For instance, I often find myself making a `models` directory inside of `services`. My rule of thumb is to only further sort files into directories if there is some rational hierarchy by which you can organize the files.

Files

Each file should have one "thing" in it, where a "thing" is a controller, directive, filter, or service. This makes for small, focused files. It also helps create a litmus test for how APIs are doing. If you find yourself flipping back and forth through files too frequently, this is an indication that your APIs are too complex. You should rethink, refactor, and simplify.

I would make an exception for closely related directives. For example, if you have an `<pane>` directive that requires `<panel>` to be a parent, those should be in the same file.

Modules

Define and configure all modules in `app.js`:

```
angular.module('yourAppName', ['yourAppDep']);
angular.module('yourAppDep');
```

Define controllers, services, etc. on modules like this:

```
angular.module('yourAppDep').controller('MyCtrl', function () {
  // ...
});
```

Although we (the AngularJS team) have discussed being able to lazy-load at the module granularity, it's not yet on the roadmap for the next version of Angular. There was a good discussion on Google+ about using multiple top-level AngularJS apps to produce a lazy-loading-like effect. I haven't tried it or seen it done, but if you desperately need to reduce payload size, that's certainly one way to do it.

The only remaining question is how to subdivide controllers, directives, services, and filters into modules. The Angular Seed puts filters, services, and directives into separate modules, but that seems a bit silly to me. Depending on the app, I'd be more inclined to organize modules by page/route. From a performance perspective, it doesn't really matter how you organize your modules, so choose whatever method best suits your project.

Dependencies

In general, services, controllers, directives, etc. should have as few dependencies as possible. This is good software development practice in general, but it's worth mentioning. This also will help in testing.

APIs should be layered. Controllers especially should not be synthesizing different levels of abstraction.

Directives

Use an app-specific prefix for directives. This is useful for avoiding collisions with 3rd party components. On the subject of 3rd party components, there's a growing community site called ngmodules that looks promising.

For example, if your app is called "The Best Todo List App Ever," you might prefix your directives with "btla."

```
angular.module('yourAppDep').directive('btlaControlPanel', function () {
  // ...
});
```

You might worry about directive names becoming too long, but I haven't seen it become problematic. You'll never really see a performance hit from long directive names, thanks to gzip compression.

Services

```
angular.module('yourAppDep').service('MyCtrl', function () {
  // ...
});
```

Models

AngularJS is unique among JavaScript frameworks in that it gives you complete control over your model layer. I think this is one of the great strengths of Angular, because at the core of your application is your data, and data varies wildly from app to app. My biggest recommendation is to consider what data you will be using, and how you will store it.

If you're using a NoSQL datastore like CouchDB or MongoDB, you might be content working with plain-old JavaScript Objects (POJO) and functional helpers. If you're using a relational database like MySQL, you might want psuedo-classes with methods for mutating, serializing, and deserializing the data. If your server provides a RESTful interface, `$resource` might be a good place to start. These are just suggestions; any of these approaches might be useful outside of the situations I described. With so many options, this is sometimes a challenging decision to make. But thinking hard about your data will pay off.

In general, you'll find many great utilities for dealing with models in Underscore.js, the utility library that also powers Backbone.js.

Controllers

By convention, controller names should start with a capital letter and end with "Ctrl".

```
angular.module('yourAppDep').controller('MyCtrl', function () {
  // ...
});
```

Remember that controllers can be reused. This seems obvious, but I've caught myself reimplementing features in different controllers. For example, consider a user control panel that allows a user to change application settings and a dialog box prompting a user to change a setting. Both could share a common controller.

Performance

AngularJS apps are generally very, very fast. Most apps really don't need any sort of special optimization, so unless you're experiencing poor performance, your time is better spent improving your app in other ways. For these exceptional cases Angular provides some excellent ways to address performance issues. First though, it's important to identify the cause of performance degradation. For that, I highly recommend the AngularJS Batarang Chrome extension or Chrome's built-in CPU profiling.

Optimizing the Digest Cycle

AngularJS uses dirty checking in its "digest cycle." For the uninitiated, you can read more about the digest cycle in the official docs and in this StackOverflow answer.

Sometimes, you want to be able to avoid a digest cycle. One common situation in real-time WebSocket'ed apps is that you don't always want to digest when you receive a message. Consider a real-time game where messages are sent from the server 30 or more times per second.

```
app.factory('socket', function ($rootScope) {
  var socket = io.connect();
  return {
    on: function (eventName, callback) {
      socket.on(eventName, function () {  // this could fire many times a
second
        var args = arguments;
        $rootScope.$apply(function () {
          callback.apply(socket, args);
        });
      });
    }
    // ...
  };
});
```

One great way to deal with this is to "throttle" the requests so a digest only happens a few times a second. Underscore.js provides such a function, but the implementation is pretty tiny, so I've reproduced it inside the `socket` service below:

```
app.factory('socket', function ($rootScope) {

  // Underscore.js 1.4.3
  // http://underscorejs.org
  // (c) 2009-2012 Jeremy Ashkenas, DocumentCloud Inc.
  // Underscore may be freely distributed under the MIT license.

  // _.throttle
  // https://github.com/documentcloud/underscore/blob/master/underscore.js#L626
  // Returns a function, that, when invoked, will only be triggered at most
once
  // during a given window of time.
  var throttle = function (func, wait) {
    var context, args, timeout, result;
    var previous = 0;
    var later = function() {
      previous = new Date();
      timeout = null;
      result = func.apply(context, args);
```

```
    };
    return function() {
      var now = new Date();
      var remaining = wait - (now - previous);
      context = this;
      args = arguments;
      if (remaining <= 0) {
        clearTimeout(timeout);
        timeout = null;
        previous = now;
        result = func.apply(context, args);
      } else if (!timeout) {
        timeout = setTimeout(later, remaining);
      }
      return result;
    };
  };

  var socket = io.connect();
  return {
    on: function (eventName, callback) {
      socket.on(eventName, throttle(function () {  // limit to once every 500ms
        var args = arguments;
        $rootScope.$apply(function () {
          callback.apply(socket, args);
        });
      }, 500));
    }
    // ...
  };
});
```

Other times, you know an incoming change will only affect certain scopes, and might want to just dirty check those. For those cases, you can call `$scope.$digest` instead of `$scope.$apply`. `$digest` will only run the digest cycle on the scope it's called on, and all of that scope's children.

Finally, to keep digest cycles short, watch expressions in `$scope.$watch` should be as fast as possible. Wherever possible, avoid deep comparisons. Remember, you only need to compare things that affect the view.

Filters

Filters are called at least twice during every digest cycle. For that reason, it's best if they are lightweight.

In cases where data is loaded and displayed, but not modified, moving the transformations done in filters to run on the data when it's fetched might be better. This is actually relatively simple, since AngularJS exposes filters to be used programmatically through `$filter`.

For instance, you load a list of names in lower-case, but need title-case, so you have a filter that does the transformation:

```
{{someModel.name | titlecase}}
```

It's pretty easy to move this to your controller and change the name to title-case when it's loaded.

```
angular.module('myApp').controller('MyCtrl', function ($scope, $http, $filter)
{
  $http.get('/someModel')
    .success(function (data) {
      $scope.someModel = data;
      // run the same "titlecase" filter inside the controller after loading
the data
      $scope.someModel.name = $filter('titlecase')($scope.someModel.name);
    });
});
```

For cases that you can't transform data upon retrieval, memoization is a great way to speed up your expensive filters without much work. Addy Osmani has a pretty great article on memoization in JavaScript that's worth a read. As for implementation, Underscore.js provides an excellent memoization function that you can use. This technique should not be used blindly, however. Memoization only helps when you're calling a filter frequently with the same, unchanged models. For rapidly changing models that have many different values throughout your application's runtime, you're better off not memoizing filters that act on that data.

Testing

Testing is immensely important for large projects. Tests allow you to refactor with confidence, which is essential in keeping code clean on a large project. Large apps should have both unit and end-to-end (E2E) tests. Unit tests are great in helping you pinpoint problems, and E2E tests ensure that the whole app works as expected. Each controller, service, filter, and directive should have a set of unit tests. Each feature of your app should have an E2E test.

This is another topic that deserves more attention. Fortunately, the AngularJS documentation has more to say on both unit and E2E tests. My colleage Vojta Jina also spoke recently about testing directives. The video is on YouTube, and definitely worth a watch.

Tools

I've been doing a lot of work on Yeoman to try to consolidate best practices and good project structure, and make what little boilerplate AngularJS has automagically generated. I highly recommend checking it out.

The AngularJS Batarang is another of my projects, and is great for both debugging and finding performance bottlenecks.

Server

As you're aware, you can use any server you want with AngularJS. It is strictly a client-side library. My recommendation and prefered setup is to use Node.js alongside nginx. I use nginx to server static files, and Node to create a RESTful API and/or socketed app. Node.js hits a sweet spot between ease of use and speed. For instance, it's relatively easy to spawn worker processes or create a webserver that can use all of your fancy server's cores.

As for cloud providers, I've used both Nodejitsu and Linode to great success. Nodejitsu is great if you're sticking strictly to node. It makes deploying your app easy, and you don't have to worry about your server environment. You can spawn additional node processes as needed to scale and handle bigger loads. If you need more control over your server environment, Linode gives you root to a fleet of VMs. Linode also provides a nice API for managing VMs. And there are plenty of other great cloud server providers that I haven't had a chance to look at yet myself.

Configuring and scaling a backend is a subject worthy of its own article, and there's no shortage of great advice elsewhere for that.

Build Process

Admittedly, this is one thing Angular needs to be better at, and one of my huge aims for 2013 is to help on this front. I've released ngmin, a tool that I hope will ultimately solve the problem of minimizing AngularJS apps for production.

For now, I think your best bet is concatenating your JavaScript files with `app.js` first, then using `ngmin` to annotate DI functions, and finally minifying with Closure Compiler with the flag `--compilation_level SIMPLE_OPTIMIZATIONS`. You can see an example of this at work in the build process for angular.js.

I don't recommend using RequireJS with AngularJS. Although it's certainly possible, I haven't seen any instance where RequireJS was beneficial in practice.

Conclusion

AngularJS is one of the most suitable JS frameworks for writing large apps. Out-of-the-box, it's very fast and greatly helps structure your app. But hopefully this advice is useful as you push the boundaries for what's possible with AngularJS.

Have some advice for scaling Angular apps? Tweet, email, or send a pull request on Github.