Angular’s Dependency Injection v2

This is a proposal for changing the current Dependency Injection (DI) in AngularJS. Most of these ideas have been already implemented in either node-di or dart-di. These changes don’t have to be done all at once.

STATUS UPDATE:
This document did a great job but now it’s pretty obsolete. Rest in peace document.

Most of this stuff has already been implemented. Please see the repo. If you have any questions, feedback or ideas, please submit an issue or send a question to angular-dev@googlegroups.com.

Remove the config phase

STATUS: DONE (see the repo)

WHY?

- it is causing a lot of confusion
- imperative configuration does not buy us that much (except the ability to have DSLs)
- make it simple
- simpler to analyze (no imperative $provide.x calls to change the dependency graph)

current:

```
module.config(function($locationProvider) {
  $locationProvider.html5Mode(true);
});
```

proposal:
Remove the config phase and make the configuration declarative rather than imperative:

```
module.value('$http.config', {
  defaultHeaders: {
    // ...
  }
});
```

API changes:
remove module.config -> declarative, provide a config service (value)
remove module.constant -> use value instead
remove module.provider -> convert provider APIs into declarative config properties and replace the provider with a factory
questions:
Is there any use case that we can't cover with declarative configuration ?

Remove the global module registry

STATUS: DONE (see the repo)
We managed to get rid off the concept of a DI module. Any regular (ES6/AMD) module can be loaded. Providers can be loaded directly. Therefore the code examples below are completely obsolete. Please see the repo for actual code examples.

WHY?

- global state is evil
- confuses the style of registering stuff with DI (create a module per file? or use global variable to store reference to a single module? a single file where you register everything with a module? - it's a mess)
- defining a dependency module is hard (you have to do two steps: 1/ define the dependency - a string id of the module; 2/ include the code that defines the module; it is easy to forget one of them; by relying on ES6 modules we gonna make this simpler - if you depend on a module, you just import it)
- impossible to use two modules with the same id (harder to re-use code)

defining a dependency has two steps: 1/ define a dependency (a string id of the module) 2/ include the code of that module
it is easy to forget one of them
by removing the global registry and rather relying on the ES6 module we make this simple - if you depend on a module, you import it.

current:
using `angular.module('x')` is essentially the same as having `var app = angular.module('x', [])` in one file and then using the `app` variable in another file. The only reason for this is to allow automatic bootstrap, where we specify the module to load in the html as a string (`ng-app` attribute).

proposal:
Get rid off the global registry. `angular.module()` always returns a new instance of a module. Use references rather than string ids.

```
var m1 = angular.module();
var m2 = angular.module();

// manual bootstrap
// the second argument is optional
angular.bootstrap([m1, m2, …], document.body);
```

questions:

Can we still keep the auto bootstrap ?
No. Unless we have some sort of convention of a global place where the developer can put the modules, which is not worthy.

Example of using manual bootstrap and ES6 modules (transpiled into Require.js) without global module registry:
https://github.com/vojtajina/dashboard.angularjs.org/blob/es6-modules-di-proposal/app/scripts/main.js
An example of a module with a dependency:
I'm not super happy about this syntax, hopefully we can find something better…
Also, this syntax will be in the same style as "Private Modules" (they need to specify which providers are public/private).

```
// import the dependency module
// this can even be an url to CDN
module requireModule from 'angular-require.js';

// this is an array of modules (dependencies)
// the DI will treat it differently (load it as a module,
// rather than a provider)
var __modules__ = [requireModule];

var createMongoResource = function($resource) {
  return $resource(...);
};

export {__modules__, createMongoResource}
```

Hierarchical Injectors

STATUS: DONE (see the repo)

WHY

- allow non singleton objects
- allow lazy loading of code

current:
Everything is a singleton. If an application needs any better control over the life of a service, the developer has to do it manually. Also, all the code has to be loaded before bootstrap (yes, there are hacks around this).

proposal:
Injector encapsulates a “life scope” - it creates and caches instances with the same life cycle.

There is a hierarchy of injectors. When creating a child injector, it is possible to load additional modules, override any binding or force new instance of any binding. This allows for lazy loading of code during the run-time. For instance, Angular can create a child injector “per route” and load additional modules before the route is activated. A child injector can get instances from its parent (ancestors), but not the other way around.

Creating new child injectors could be transparent to the application developer. Angular could take care of managing the life scopes (creating new child injectors). For instance, there can be “application” scope, or “route” scope and the developer can mark services using “scope” annotations. Angular will basically provide a higher abstraction (named scopes) on the top of injectors.

questions:
Can we mimic the scope / DOM structure and inject everything (elements, directives, ...) ?

Private modules

STATUS: NOT STARTED YET
This feature has been implemented in node-di. At this point I don't think it is as crucial feature and so I decided to wait until we need it. It might be better to modularize on a component (UI) level as Angular will create a child scope per component anyway. I will describe in more detail.

WHY

- better modularization
- allow third party developers to use DI without name collisions

current:
There's only a single flat namespace, which has problems:

- no encapsulation (anything can ask for anything)
- token name collisions (two third party modules can't use the same token)

proposal:
A module can explicitly define/export public bindings. When the module is loaded into an injector, only the public bindings are accessible outside of this module. This can be used for third party modules, but also to modularize a big project.

```
// This is module definition syntax from node-di,
// we can figure out something in current Angular API style
var module = {
  // list of bindings that are public,
  // if not defined, all the bindings are public
  __exports__: ['github'],
  github: ['factory', createGithub],

  // private, only instances from this module can see it
  githubAuth: ['factory', createGithubAuth]
};
```

This also allows re-mapping of tokens.

Property Injection

STATUS: NOT STARTED YET
This feature has been implemented in node-di. It works well with string tokens. I'm however very happy we managed to get rid off the string tokens and so I'm not sure if this makes any sense. Building the flagship app will show us if we need something like this. I think we do. I'm however not sure what is the right implementation.

This is just a syntactic sugar to make it easier to only inject what the service actually needs, without defining every little configuration aspect as a separate service (binding).

proposal:

```
module.value('config', {
  $location: {
    html5Mode: true,
  },
  $http: {
    defaultHeaders: {
      post: {},
      get: {}
    }
  }
});

module.factory('$http', ['config.$http.defaultHeaders',
    function(defaultHeaders) {
  // unless there is a direct binding for
  // "config.$http.defaultHeaders" token,
  // defaultHeaders property of config.$http gets injected
}]);
```

ES6 Modules Integration

STATUS: DONE (see the repo)
This section is obsolete, please see the code examples in the repo. We use annotations and recommend references to concrete functions/classes as tokens rather than strings.

Using a module loader (eg. RequireJS, CommonJS) is a good practice (explicit dependencies, no need to maintain a list of files with the correct order). Typically developers end up defining both DI modules and JS modules, which is extra typing and it is confusing.

WHY?

- simplify using DI together with module loaders
- ES6 is the future
- ES6 can be easily transpiled into RequireJS/CommonJS

The main goal is to minimize the overhead of defining both DI and JS modules.

proposal:
ES6 modules can be consumed as DI modules.

```
// defining the module, in src/http.js
http.$inject = ['$q'];
export function $http(q) {}

// loading the module, in src/main.js
module http from 'http';
angular.bootstrap(element, [http]);
```

I hope this will (together with removing global module registry) remove the current confusion of many different ways of how to register stuff with the DI and will enable us to promote “the” way of doing it.

I have refactored the dashboard app to show this (using ES6 modules, transpiled to Require.js, using helpers to mimic the new DI behavior):
https://github.com/angular/dashboard.angularjs.org/pull/25

Asynchronous Injection

STATUS: NOT STARTED YET
I will update this design doc with a proposal on how to implement asynchronous injection.

TO BE DONE

This is Misko's crazy idea from couple of months ago and I'm becoming brain-washed to like it. I think this is actually very interesting approach that could work, I wanted to try it in node-di, but didn't have any use case. I will try to make a prototype of it.

WHY?

- removing asynchronicity from the application code
- lazy code loading even on smaller scale, not just per route (injector)

https://docs.google.com/document/d/12cX3alzGb75aXW7j5skCXLieTODJ8YBo4vt37kpDbJE/edit?usp=sharing

Annotations

STATUS: DONE (see the repo)
We use annotations such as @Inject, @Provide. Also type annotations can be used for injection. Annotations are not part of ES6, however we provide a simple helper function in the case developers don't want to transpile their code.

Since we need to transpile the source code anyway (because of using ES6 modules), we might use annotations to simplify the syntax:

current:

```
var MyController = function(http) {
  // ...
};
MyController.$providerType = 'controller';
MyController.$inject = ['$http'];
```

proposal:

```
@controllerProvider
@inject('$http')
var MyController = function(http) {
  // ...
};
```

Other Questions

This is a list of common questions or problems we should support. Feel free to add more...

Lazy Code Loading

- child injectors help significantly (we can lazy load modules per injector; eg. per route)
- asynchronous injection might enable even finer lazy loading (eg. per service/binding)

Releasing memory

Currently, an injector keeps references to everything (providers, instances, etc.). As long as somebody is holding a reference to the injector, we can't release any memory. This can cause a significant memory usage.

We might provide some API to explicitly free up some of the providers/instances or all of them. We might also try generating the DI code for production.

Configuring the app

Configuration (eg. for different environments - testing/stagging/production) happens through loading different modules. This can be done either on the client side (eg. with Require.js it is possible to load a module conditionally) or even better on the server side. The client would request `main.js?config=abc` and the returned `main.js` would contain specified modules only.

How would collaborative config be possible? (eg: http interceptors is an array, so my code might just want to add an interceptor without overriding other setup)
I don't have clear answer to this yet.
Decorators could be use (however I'd like to avoid decorators).
Also the injector might have some sort of "getAllX" API, which would return list of registered tokens (or instances). This is something we will need for directives (the compiler needs to know about all the registered directives). We might use it for this kind of configuration - you could basically register multiple interceptors and the $http service would then ask the injector for all the "interceptors".

Other notes IGNORE THIS ;-)
- prebound functions/classes (first x args are bound)
- graph checking (missing providers; static?)
- generate graph data + charts
- module loader syntactic sugar (just import modules adds them into a global inector)
- lazy bindings (inject a fn provider)
- multibindings

-configured binding
  @Inject('Element(input.cls)', 'Logger(logId)')