Dependency Injection

In general, there are only three ways an object can get a hold of its dependencies:

1. We can create it internally to the dependent.
2. We can look it up or refer to it as a global variable.
3. We can pass it in where it's needed.

With dependency injection, we're tackling the third way (the other two present other difficult challenges, such as dirtying the global scope and making isolation nearly impossible). Dependency injection is a design pattern that allows for the removal of hard-coded dependencies, thus making it possible to remove or change them at run time.

This ability to modify dependencies at run time allows us to create isolated environments that are ideal for testing. We can replace *real* objects in production environments with mocked ones for testing environments.

Functionally, the pattern *injects* depended-upon resources into the destination when needed by automatically looking up the dependency in advance and providing the destination for the dependency.

As we write components dependent upon other objects or libraries, we will describe its dependencies. At run time, an `injector` will create instances of the dependencies and pass them along to the *dependent* consumer.

```
// Great example from the Angular docs
function SomeClass(greeter) {
  this.greeter = greeter;
}
SomeClass.prototype.greetName = function(name) {
  this.greeter.greet(name)
}
```

It is never a good idea to create a controller on the global scope like we've done in the sample code above. We're doing it only as an example for simplicity's sake.

At runtime, the `SomeClass` doesn't care *how* it gets the `greeter` dependency, so long as it gets it. In order to get that `greeter` instance into `SomeClass`, the creator of `SomeClass` is responsible for passing in the `SomeClass` dependencies when it's created.

Angular uses the `$injector` for managing lookups and instantiation of dependencies for this reason. In fact, the `$injector` is responsible for handling all instantiations of our Angular components, including our app modules, directives, controllers, etc.

When *any* of our modules boot up at run time, the `injector` is responsible for actually instantiating the instance of the object and passing in any of its required dependencies.

For instance, this simple app declares a single module and a single controller, like so:

```
angular.module('myApp', [])
.factory('greeter', function() {
  return {
    greet: function(msg) { alert(msg); }
  }
})
.controller('MyController',
  function($scope, greeter) {
    $scope.sayHello = function() {
      greeter.greet("Hello!");
    };
});
```

At run time, when Angular instantiates the instance of our module, it looks up the greeter and simply passes it in naturally:

```
<div ng-app="myApp">
  <div ng-controller="MyController">
    <button ng-click="sayHello()">Hello</button>
  </div>
</div>
```

Behind the scenes, the Angular process looks like:

```
// Load the app with the injector
var injector = angular.injector(['ng', 'myApp']);
// Load the $controller service with the injector
var $controller = injector.get('$controller');
var scope = injector.get('$rootScope').$new();
// Load the controller, passing in a scope
// which is how angular does it at runtime
var MyController = $controller('MyController', {$scope: scope})
```

Nowhere in the above example did we describe how to find the greeter; it simply *works*, as the injector takes care of finding and loading it for us.

AngularJS uses an `annotate` function to pull properties off of the passed-in array during instantiation. You can view this function by typing the following in the Chrome developer tools:

```
> injector.annotate(function($q, greeter) {})
["$q", "greeter"]
```

In every Angular app, the `$injector` has been at work, whether we know it or not. When we write a controller without the `[]` bracket notation or through *explicitly* setting them, the `$injector` will *infer* the dependencies based on the name of the arguments.

Annotation by Inference

Angular assumes that the function parameter names are the names of the dependencies, if not otherwise specified. Thus, it will call `toString()` on the function, parse and extract the function arguments, and then use the `$injector` to *inject* these arguments into the instantiation of the object.

The injection process looks like:

```
injector.invoke(function($http, greeter) {});
```

Note that this process will *only* work with non-minified, non-obfuscated code, as Angular needs to parse the arguments intact.

With this JavaScript inference, order is **not** important: Angular will figure it out for us and inject the right properties in the "right" order.

JavaScript minifiers generally change function arguments to the minimum number of characters (along with changing white spaces, removing new lines and comments, etc.) so as to reduce the ultimate file size of the JavaScript files. If we do not explicitly describe the arguments, Angular will not be able to infer the arguments and thus the required injectable.

Explicit Annotation

Angular provides a method for us to explicitly define the dependencies that a function needs upon invocation. This method allows for minifiers to rename the function parameters and still be able to inject the proper services into the function.

The injection process uses the `$inject` property to annotation the function. The `$inject` property of a function is an array of service names to inject as dependencies.

To use the `$inject` property method, we set it on the function or name.

```
var aControllerFactory =
  function aController($scope, greeter) {
    console.log("LOADED controller", greeter);
    // ... Controller
  };
aControllerFactory.$inject = ['$scope', 'greeter'];
// Greeter service
var greeterService = function() {
  console.log("greeter service");
}
// Our app controller
angular.module('myApp', [])
  .controller('MyController', aControllerFactory)
  .factory('greeter', greeterService);
// Grab the injector and create a new scope
var injector  = angular.injector(['ng', 'myApp']),
  controller  = injector.get('$controller'),
  rootScope   = injector.get('$rootScope'),
  newScope    = rootScope.$new();
// Invoke the controller
controller('MyController', {$scope: newScope});
```

With this annotation style, order is important, as the `$inject` array must match the ordering of the arguments to inject. This method of injection *does* work with minification, because the annotation information will be packaged with the function.

Inline Annotation

The last method of annotation that Angular provides out of the box is the inline annotation. This syntactic sugar works the same way as the `$inject` method of annotation from above, but allows us to make the arguments inline in the function definition. Additionally it affords us the ability to not use a temporary variable in the definition.

Inline annotation allows us to pass an array of arguments instead of a function when defining an Angular object. The elements inside this array are the list of injectable dependencies as strings, the last argument being the function definition of the object.

For instance:

```
angular.module('myApp')
  .controller('MyController',
    ['$scope', 'greeter',
      function($scope, greeter) {

  }]);
```

The inline annotation method *works* with minifiers, as we are passing a list of strings. We often refer this method as the bracket or array notation `[]`.

$inject API

Although it's relatively rare that we'll need to work directly with the `$injector`, knowing about the API will give us some good insight into how exactly it works.

annotate()

The `annotate()` function returns an array of service names that are to be injected into the function when instantiated. The `annotate()` function is used by the injector to determine which services will be injected into the function at invocation time.

The `annotate()` function takes a single argument:

- fn (function or array)

The `fn` argument is either given a function or an array in the bracket notation of a function definition.

The `annotate()` method returns a single array of the names of services that will be injected into the function at the time of invocation.

```
1 var injector = angular.injector(['ng', 'myApp']);
2 injector.annotate(function($q, greeter) {});
3 // ['$q', 'greeter']
```

Try it in your Chrome Debugger.

get()

The `get()` method returns an instance of the service and takes a single argument.

- name (string)

The `name` argument is the name of the instance we want to get.

`get()` returns an instance of the service by `name`.

has()

The `has()` method returns true if the injector knows that a service exists in its registry and false if it does not. It takes a single argument:

- name (string)

The string is the name of the service we want to look up in the injector's registry.

instantiate()

The `instantiate()` method creates a new instance of the JavaScript type. It takes a constructor and invokes the `new` operator with all of the arguments specified. It takes two arguments:

- Type (function)

This function is the annotation constructor function to invoke.

- locals (object – optional)

This optional argument provides another way to pass argument names into the function when it is invoked.

The `instantiate()` method returns a new instance of `Type`.

invoke()

The `invoke()` method invokes the method and adds the method arguments from the `$injector`.

This `invoke()` method takes three arguments:

- fn (function)

This function is the one to invoke. The arguments for the function are set with the function annotation.

- self (object – optional)

The `self` argument allows for us to set the `this` argument for the invoke method.

- locals (object – optional)

This optional argument provides another way to pass argument names into the function when it is invoked.

The `invoke()` method returns the value that the `fn` function returns.

ngMin

With the three methods of defining annotations from above, it's important to note that these options all exist when defining a function. In production, however, it is often less convenient to explicitly concern ourselves with order of arguments and code bloat.

The `ngMin` tool allows us to alleviate the responsibility to define our dependencies explicitly. `ngMin` is a *pre-minifier* for Angular apps. It walks through our Angular apps and sets up dependency injection for us.

For instance, it will turn this code:

```
angular.module('myApp', [])
.directive('myDirective',
  function($http) {
})
.controller('IndexController',
  function($scope, $q) {
});
```

into the following:

```
angular.module('myApp', [])
.directive('myDirective', [
  '$http',
  function ($http) {
  }
]).controller('IndexController', [
  '$scope',
  '$q',
  function ($scope, $q) {
  }
]);
```

`ngMin` saves us a lot of typing and cleans our source files significantly.

Installation

To install `ngMin`, we'll use the `npm` package manager:

```
$ npm install -g ngmin
```

If we're using Grunt, we can install the `grunt-ngmin` Grunt task. If we are using Rails, we can use the Ruby gem `ngmin-rails`.

Using ngMin

We can use `ngMin` in standalone mode at the CLI by passing two arguments: the input.js and the output.js files or via stdio/stdout, like so:

```
$ ngmin input.js output.js
# or
$ ngmin < input.js > output.js
```

where input.js is our source file and output.js is the annotated output file.

How It Works

At its core, `ngMin` uses an Abstract Syntax Tree (AST) as it walks through the JavaScript source. With the help of `astral`, an AST tooling framework, it rebuilds the source with the necessary annotations and then dumps the updated source using `escodegen`.

`ngmin` expects our Angular source code to consist of logical declarations. If our code uses syntax similar to the code used in this book, `ngMin` will be able to parse the source and pre-minify it.