This is a preview-quality chapter of our continuously deployed eBook on AngularJS for .NET developers. You can read more about the project at http://henriquat.re. You can also follow us on twitter (Project: @henriquatreJS, Authors: @ingorammer and @christianweyer)

Services and Dependency Injection in AngularJS

As you've seen in Controllers and $scope *(Please note: this chapter has not yet been released)* AngularJS makes heavy use of Dependency Injection (DI) to retrieve references to components. In this chapter, we'll show you how AngularJS uses DI and how you can inject your own objects into controllers and services for your applications.

— A 30 Second Intro to Dependency Injection —

DI is a pattern which is often used in infrastructure components and which ensures that one particular component does not directly create references to other components. Instead of direct instantiation, every component will receive references to required other components like *helpers*, *services*, etc. as parameters to their constructor.

(Yes, we realize that this is a very simplified definition as there is more than just *constructor injection*, but for the scope of AngularJS, this should be sufficient for now. If you're interested in DI in the broader scope for .NET developers, we absolute recommend the book 'Dependency Injection in .NET' by Mark Seemann.)

Angular's Dependency Injection allows you to write code like the following without taking into account where `$scope` comes from.

```
function UserController($scope) {

   $scope.currentUser = {
      firstName: "John",
      lastName: "Doe"
   };

}
```

In this case, `$scope` gets injected by Angular whenever this controller is instantiated. *We'll explain in a second how the dependency injector resolves the correct values.*

Angular uses this approach because it allows for a clear separation of concerns and for a higher degree of focused unit-testing. You can for example configure the DI framework to use *mock-objects* for underlying components instead of real services during your unit tests. This approach allows you to focus on testing only *one particular piece of functionality* - one unit - instead of testing all the underlying services as well. You can read more about Angular's ideas around testing at Unit Testing withing AngularJS Applications *(Please note: this chapter has not yet been released)*.

An additional benefit of Angular's internal reliance on DI is that you can replace nearly every part of AngularJS itself with your own implementation. This includes all built-in services and directives. (Not purely DI-related, but if you for example don't like how `ng-click` works, you could simply provide your own implementation!).

— Well-Known Services —

Your first contact with dependency injection is quite likely the `$scope` parameter in a controller as shown above. But AngularJS offers more services than these.

Other base services which you'll encounter are for example:

SERVICE	DESCRIPTION
`$http`	Allows promise-based access to HTTP requests (API Reference: ng.$http). You can read more about promises in Promises instead of Callbacks *(Please note: this chapter has not yet been released)*.
`$resource`	Provides a higher-level abstraction for accessing REST-style services (API Reference: ng.$resource)
`$document`	Is a jQuery (lite)-wrapped reference to `window.document` (API Reference: ng.$document)
`$window`	Is a jQuery (lite)-wrapped reference to `window` (API Reference: ng.$window)
`$timeout`	This is a wrapped version of `window.setTimeout`. In tests, you can use `$timeout.flush()` to synchronously execute all scheduled timeouts. (API Reference: ng.$timeout)
`$parse`	Parses Angular expressions (for example, for binding with `ng-model` or similar) and provides accessor functions (API Reference: ng.$parse)
`$cacheFactory`	Usually used by other services whenever they need a scoped cache of elementes (API Reference: ng.$cacheFactory)
`$filter`	Provides programmatic access to a filter function (API Reference: ng.$filter)

In Angular's API Documentation, you can find a list of additional services in the left-side navigation bar.

— Using $http to Retrieve Data —

To look at the use of one of these services in more detail, let's just examine an example in which we use the `$http` service to look up geographical information for the location of our company's HQ. We're going to use the free/open service at http://www.geonames.org to perform this lookup.

You can access this web service with an URL like http://api.geonames.org/postalCodeLookupJSON?postalcode=XXXXX&country=YY&username=ZZZZZ. In this URL, XXXXX is the postal code to lookup, YY is the country code and ZZZZ is your *free* user name. An example would be this request, for which the service returns the following piece of information.

```
{"postalcodes": [{
  "adminCode3": "09677",
  "adminName2": "Unterfranken",
  "adminName3": "Main-Spessart",
  "adminCode2": "096",
  "postalcode": "97845",
  "adminCode1": "BY",
  "countryCode": "DE",
  "lng": 9.5675,
  "placeName": "Neustadt am Main",
  "lat": 49.9308,
  "adminName1": "Bayern"
}]}
```

Alternatively, if all credits for one particular account have been used up (currently, 30,000 requests per day are allowed for free users), you would receive a message similar to the following from the web service:

```
{"status": {
  "message": "the deaily limit of 30000 credits for demo has been exceeded. Please use an application specific account. Do not use the demo account for your application.",
  "value": 18
}}
```

In our particular case - to avoid cross-domain restrictions - we're going to retrieve this data as JSONP by specifying a *callback function*. For JSONP, this particular service expects the temporary callback-method's name to be passed as a parameter named `callback` in the URL.

To facilitate this, `$http` can dynamically generate a JSONP-compatible callback function behind the scenes, and then uses *macro*-replacement in the URL by adding the method's name in place of the string *JSONCALLBACK*. So basically we just have to append `"&callback=JSON_CALLBACK"` to the above URL to retrieve the data using JSONP.

Postal Code: 97845 Country: DE Lookup
Result: Not yet retrieved

Edit in JSFiddle

In the markup for this simple demo, we're binding the two text boxes to values of `$scope.query` and invoke the method `performPostcodeLookup()` when the user clicks the button. In addition, we display a result string which we retrieve from the scope's `lookupResult`-object which we'll create in the controller.

```
<div ng-app="demo" ng-controller="DemoController">
    Postal Code: <input type=text ng-model="query.postalCode">
    Country: <input type=text ng-model="query.countryCode">
    <button ng-click="performPostcodeLookup()">Lookup</button><br>
    Result: {{lookupResult.displayValue}}
</div>
```

In this controller, we first populate the `$scope` with default values for query and response and after this add the method `performPostcodeLookup` to `$scope` so that it can be called from the view.

```
var demoModule = angular.module("demo", []);

function DemoController($scope, $http) {
   $scope.query = {
      postalCode: "97845",
      countryCode: "DE"
   };

   $scope.lookupResult = {
      displayValue: "Not yet retrieved"
   };

   $scope.performPostcodeLookup = function () {
      var url = "http://api.geonames.org/postalCodeLookupJSON?" +
         "postalcode=" + $scope.query.postalCode +
         "&country=" + $scope.query.countryCode +
         "&username=henriquatre" +
         "&callback=JSON_CALLBACK";

      $http.jsonp(url).then(function (response) {
         var data = response.data;
         if (data.status) {
            $scope.lookupResult.displayValue = "Error: " + data.status.message;
         } else {
            if (data.postalcodes.length === 0) {
               $scope.lookupResult.displayValue = "No data found";
            } else {
               $scope.lookupResult.displayValue = data.postalcodes[0].placeName;
            }
         }
      }, function (data, status) {
         $scope.lookupResult.displayValue = "HTTP Error - " + status;
      });
   }

}
```

As you can see here, we've simply added a parameter called `$http` to the constructor of the controller and we're later using this object to call a method which retrieves the JSONP payload from the service.

— But Where Does $http Come From? —

So the question is: *where does the value for `$http` come from*? The simple answer is: AngularJS injects *all parameters* whenever a controller is instantiated.

This is performed using Angular's `$injector` (API Reference: angular.$injector) which - by default - simply looks at *the parameters' names*. This is very important, because it is rather atypical for most current application development environments (apart from Objective-C) that the name of a parameter has *any* significance at all.

With Angular's default behavior, the parameter's name is however *very* important. If you'd rename this from parameter from `$http` to `$httpAndMore`, it would cease to work. (This - of course - has big implications for minification of JavaScript, a common process which decreases the size of codebase by dynamically shortening variable and parameter names. We'll talk about how to solve this problem a bit later in this chapter).

— Resolving Dependencies —

As you can imagine, Angular's dependency resolver needs a mapping between these parameter names and the corresponding factory methods for them. You can easily try this by adding a custom parameter - which does not exist within Angular - to one of your controllers.

Let's assume that we want to extend the small sample above and make the default *postcode* and *country* configurable in the user's profile. To do this in *the Angular way*, we would inject an object which allows us to access the current user's configuration to initialize the `$scope` with the matching values.

Please note: an injection-parameter for custom objects should *not* be prefixed with `$`. The dollar-sign is reserved for the default services of AngularJS.

```
function DemoController($scope, $http, userProfile) {
   $scope.query = {
      postalCode: userProfile.defaultPostalCode,
      countryCode: userProfile.defaultCountryCode
   };

   $scope.lookupResult = {
      displayValue: "Not yet retrieved"
   };

   $scope.performPostcodeLookup = function () {
      $scope.lookupResult.displayValue = "Just a demo.";
   }
}
```

If you'd run this example without any further changes, you would see the following in your browser's debug console:

```
Error: Unknown provider: userProfileProvider <- userProfile
```

Translated to plain English this basically means "I couldn't find the provider `userProfileProvider` which would be necessary to create the instance for `userProfile`". Ok, that makes sense: after all, Angular can't yet know how we want to fill the parameter `userProfile`.

To register a factory for this service, we will usually first need to specify the root module for our application using `ng-app="moduleName"` in the top-level markup. This is necessary, because all providers are later registered with a particular module.

```
<div ng-app="demo" ng-controller="DemoController">
    Postal Code: <input type=text ng-model="query.postalCode">
    Country: <input type=text ng-model="query.countryCode">
    <button ng-click="performPostcodeLookup()">Lookup</button><br>
    Result: {{lookupResult.displayValue}}
</div>
```

After this, we can create the module and register the `factory` with it:

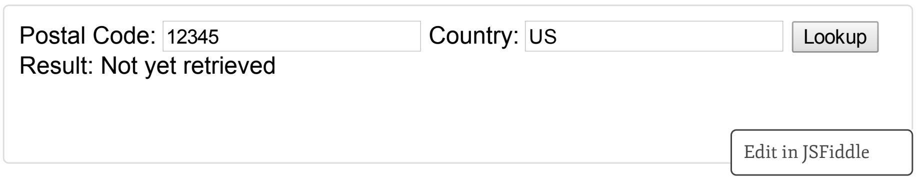
```
var demoModule = angular.module("demo", []);

demoModule.factory("userProfile", function () {
   // return a hardcoded profile for this example
   return {
      defaultPostalCode: "12345",
      defaultCountryCode: "US"
   }
});
```

As you can see in this example, a factory is registered with a module by using its `factory()` method. There are additional ways apart from `factory()` to configure certain injected values: `provider()`, `constant()`, `value()` and more. You can find out more about these in Angular's reference - they are all methods defined on `Module` (API Reference: angular.Module).

The first parameter to `factory()` is the the name of the object-to-be-injected. The second parameter is the factory method itself. (Please note: this method itself also supports injected parameters: if you'd for example need http access in your service, you can simply add a parameter called `$http` to this factory method.)

When running this sample, you can see that the defaults are now initialized from this externally injected `$userProfile`-service.


Postal Code: 12345 Country: US Lookup
Result: Not yet retrieved

Please note: the return value of a factory method is cached for the complete execution time of an application. The factory method is therefore called only once and provides a *singleton* instance of the requested service.

— Injection in Angular's Core —

Now you might wonder how and where this injection behavior is used by the AngularJS core itself. The good news is: it's used nearly *everywhere*.

Internally, AngularJS defines - amonst others - a module called `ng` which contains the core of Angular's functionality. You can read more about modules in general, and how to use them in your code in Modularizing AngularJS Applications. The `ng` module internally also depends on another module called `ngLocale`, which simply provides the current locale ID (for example `en-us`) to Angular components.

Each of Angular's default providers, factories, services, values, constants, filters and directives are then added to this `ng` module. When you for example use the `$http` service, Angular does actually not use a hardcoded reference to the default implementation of this service. Instead, even Angular internally always uses the regular dependency injection mechanism to resolve a service's implementation.

Whenever you create an application (using the `ng-app` attribute on an HTML element for example), the module `ng` will be an implicitly required module. Depending on the type of provider (for example, whether we are looking at a *directive* or at a *service*), you can override the default implementation in different ways.

In the first sample of this chapter (here), we've used the `$http` service's method `jsonp()` to retrieve location data from a public service. If we'd like to replace `$http` to provide a different implementation for this method, we could simply register the replacement factory with our own module:

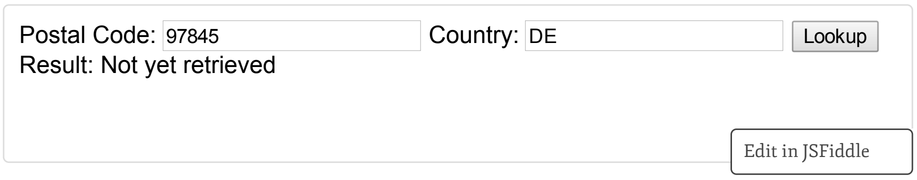
```
var demoModule = angular.module("demo", []);

demoModule.factory("$http", function ($q) {
   return {
      jsonp: function () {
         alert("This is our replacement function");

         var dummyResponse = {data: { postalcodes: [
            {placeName: "Testing"}
         ]}};


         var deferred = $q.defer();
         deferred.resolve(dummyResponse);
         return deferred.promise;
      }
   }
});
```

After running this demo (with the unchanged controller code from above), you can see that we are now getting access to *our* own version of `$http`.


Postal Code: 97845 Country: DE Lookup
Result: Not yet retrieved

Please note: in the example above, we're making use of Angular's Q-based `promises`. If you are a .NET developer, these constructs are comparable to `Task<T>` as they provide an API on top of asynchronous interactions. In a lot of cases, this API allows you to write code which most developers find more readable than the classic callback-passing style of invoking JavaScript, which you might know for example from jQuery. You can read a lot more about these constructs at Promises instead of Callbacks *(Please note: this chapter has not yet been released)*.

— Using $injector Manually —

When using an Angular application and a controller, the `$injector` will be used transparently behind the scenes. But nothing prevents you from using it manually (really, don't you love all the flexibility Angular provides?).

To do so, you would first simply call Angular's `angular.injector` factory method and pass it a list of modules which you'd like to base your injector on. (In the following demo, we're just passing Angular's `ng` module, which contains the majority of the default implementations and providers for Angular). You can then use the returned `$injector` instance and for example call `get()` on it to resolve a manually specified dependency. You can see this in the following sample:

```
var myInjector = angular.injector(["ng"]);
var $http = myInjector.get("$http");
```

After this, you can use `$http` just like before.

But `$injector` is not *just* a service locator; it can also be used as a dependency injector for object instantiations or even for regular functions. In particular, it can be used to *call methods* including *constructors*. In fact, this is the usual way in which Angular internally uses `$injector`.

```
function doSomething($http) {
   if ($http) {
      document.write("Looks good: We received a $http instance");
   } else {
      document.write("Doesn't look that great ...")
   }
}
```

You can then call this method by using the injector's `invoke` method as shown in the next fragment:

```
var myInjector = angular.injector(["ng"]);
myInjector.invoke(doSomething);
```

```
Looks good: We received a $http instance
```

Edit in JSFiddle

Now, please keep in mind that this is not the *usual Angular way*, but it knowing about this way to access the injector might sometimes come in handy when integrating AngularJS with an existing codebase.

— Dependency Injection and Testability —

As you've seen in the previous examples, it is relatively easy to inject custom elements into existing controllers - and other components - without changing a single line of code in the objects which use the injected element.

The reason for doing this is quite simple: it allows for code bases with clear dependency separation, and it provides great support for unit testing. Angular's reliance on DI provides you with a framework which is very flexible for mocking (and in fact, provides ready-made mocks for services like `$http`) and is built from the ground-up to be testable at the *unit* level as well as at the *integration* level. You can read more about testing Angular applications at Unit Testing withing AngularJS Applications *(Please note: this chapter has not yet been released)*.

— Dependency Injection and Minification —

Of course, the sky is not always *just* blue: there *are* drawbacks to the particular DI approach which has been chosen for Angular. The main issue is that most JavaScript minification toolkits do not really like the idea of 'important' parameter names. Instead, they try to make the parameter names as small as possible (usually singlecharacter names) to squeeze as much code as possible into as few bytes as possible.

This leaves you with two options: you can either disable minification for your application code (and only use it for framework code and libraries) *or* alternatively, you can use Angular's more verbose DI-syntax which is compatible with minification because it does not rely on the parameter names. Disabling minification is usually not the preferred solution, because it not only increases your code-size, but it also immediately exposes all the internals of your application. (In a lot of cases, minifaction is used as a substitute or in conjunction with obfuscation.)

To illustrate this, let's run the popular minification tool UglifyJS on the following fragment of code:

```
function DemoController($scope, $http) {
   $http.jsonp("http://example.com").then(function(response) {
      console.log(response);
   });
}
```

After running this tool, we receive the following version of the code:

```
function DemoController(a,b){b.jsonp("http://example.com").then(function(a){console.
log(a)})}
```

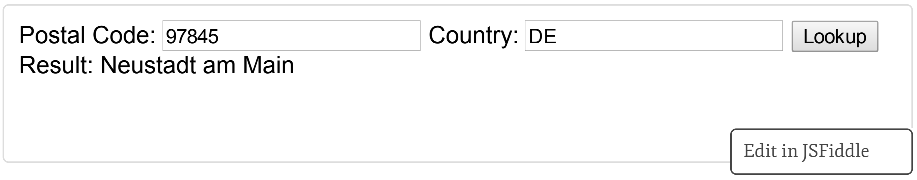
As you can see, in terms of their actual implementations, the two methods are comparable. The only thing which should really bother you is that `$scope` and `$http` have been renamed to `a` and `b`. If we'd run this code within Angular right now, it would tell us that it's looking for an `aProvider` and a `bProvider` -- neither of which would make sense in our application.

To support minification, we therefore have to provide some additional information to AngularJS in a way which preserves the *name* for the object to be injected.

There are two basic ways. The first one is by using `$inject`. This function allows you to specify the injected service names in a string array:

```
function DemoController($scope, $http) {
...
}

DemoController.$inject = ["$scope", "$http"];
```


Postal Code: 97845 Country: DE Lookup
Result: Neustadt am Main

Edit in JSFiddle

The alternative form, which is often preferred for service registration, is to use *inline annotation* for parameters. In this case, instead of explicitly calling `$inject`, you can pass the to-be-injected service names as parameters to the module's registration functions.

Instead of passing the factory function to `module.factory("name", function (...){})`, you pass an array consisting first of the services-to-be-injected and the factory function as the last parameter. The call to `factory` would therefore look like the following:

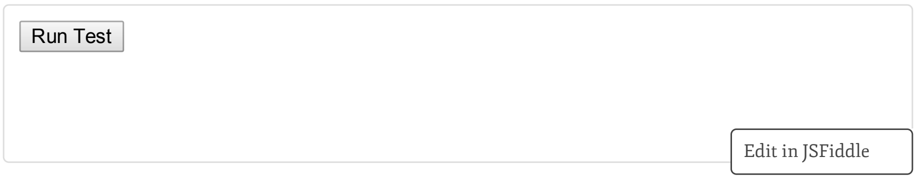
```
module.factory("name",["service1", ..., "service-n",
     function(service1, ..., serviceN) {...})]);


var demoModule = angular.module("demo", []);

demoModule.factory("someCustomService", ["$http",
   function ($http) {
      return {
         test: function () {
            alert("This is our dummy function. $http is provided: " + (!!$http));
         }
      }
   }]);

demoModule.controller("DemoController", ["$scope", "someCustomService",
   function ($scope, someCustomService) {
      $scope.runTest = function () {
         someCustomService.test();
      }
   }]);
```

When running this example (with a button with `ng-click="runTest()"`, you'll see the following behavior even after minification):



In this case, the parameter names become irrelevant. You could now easily rename the parameters to different values.

But one word of caution: this can lead to some serious confusion if the actual parameters do not match the string-array of requested elements. Angular does not complain if you would register a `factory` like the following, with different sequences of parameters in the method and in the string array:

```
demoModule.factory("test", ["$http", "someService",
   function(someService, $http) {
           $http.get("...").then(function() { /* ... */});
   }
});
```

In this case, the `$http` object will contain the instance of someService and vice versa!

— Summarizing —

In this chapter, you've seen how AngularJS relies on Dependency Injection for its internal components. You've also learned how you can add your own objects (providers/services) to the injection chain and how you can override existing injection configurations to provide mock-objects to your controllers and services.

In the next chapter, Introduction to Directives, we'll look into the creation of *directives* which allow you to provide domain-specific extensions to HTML on top of these underlying services.

This was a preview-quality chapter of our continuously deployed eBook on AngularJS for .NET developers. If you enjoyed this chapter, you can read more about the project at http://henriquat.re. You can also follow us on twitter (Project: @henriquatreJS, Authors: @ingorammer and @christianweyer)

© Copyright thinktecture, 2013 - contact & imprint