Angular Tips

Join us in our way to learning Angular.js

Understanding Service Types

Aug 12th, 2013 | Comments

Last update: June 2014. I have partially rewritten this article to provide more technical details and also to show their differences more clearly.

Angular comes with different types of services. Each one with its own use cases.

Something important that you have to keep in mind is that the services are always singleton, it doesn't matter which type you use. This is the desired behavior.

NOTE: A singleton is a design pattern that restricts the instantiation of a class to just one object. Every place where we inject our service, will use the same instance.

Provider

`Provider` is the parent of almost all the other services (all but constant) and it is also the most complex but more configurable one.

Let's see a basic example:

```
1  app.provider('foo', function() {
2
3    return {
4
5      $get: function() {
6        var thisIsPrivate = "Private";
7        function getPrivate() {
8          return thisIsPrivate;
9        }
10
11       return {
12         variable: "This is public",
13         getPrivate: getPrivate
14       };
```

```
15     }
16
17   };
18
19 });
```

A provider on its simplest form, just needs to return a function called $get which is what we inject on the other components. So if we have a controller and we want to inject this foo provider, what we inject is the $get function of it.

Why should we use a provider when a factory is much simple? Because we can configure a provider in the config function. We can do something like this:

```
1  app.provider('foo', function() {
2
3    var thisIsPrivate = "Private";
4
5    return {
6
7      setPrivate: function(newVal) {
8        thisIsPrivate = newVal;
9      },
10
11     $get: function() {
12       function getPrivate() {
13         return thisIsPrivate;
14       }
15
16       return {
17         variable: "This is public",
18         getPrivate: getPrivate
19       };
20     }
21
22   };
23
24 });
25
26 app.config(function(fooProvider) {
27   fooProvider.setPrivate('New value from config');
28 });
```

Here we moved the thisIsPrivate outside our $get function and then we created a setPrivate function to be able to change thisIsPrivate in a config function. Why do we need to do this? Won't it be easier to just add the setter in the $get? This has a different purpose.

Imagine we want to create a generic library to manage our models and make some REST petitions. If we hardcode the endpoints URLs, we are not making it any generic, so the idea is to be able to configure those URLs and to do so, we create a provider and we allow those URLs to be configured on a config function.

Notice that we have to put nameProvider instead of just name in our config function. To consume it, we just need to use name.

Seeing this we realize that we already configured some services in our applications, like $routeProvider and $locationProvider, to configure our routes and

html5mode respectively.

Providers have two different places to make injections, on the provider constructor and on the $get function. On the provider constructor we can only inject other providers and constants (is the same limitation as the config function). On the $get function we can inject all but other providers (but we can inject other provider's $get function).

Remember: To inject a provider you use: name + 'Provider' and to inject its $get function you just use name

Try it

Factory

Provider are good, they are quite flexible and complex. But what if we only want its $get function? I mean, no configuration at all. Well, in that cases we have the factory. Let's see an example:

Example:

```
1  app.factory('foo', function() {
2    var thisIsPrivate = "Private";
3    function getPrivate() {
4      return thisIsPrivate;
5    }
6
7    return {
8      variable: "This is public",
9      getPrivate: getPrivate
10   };
11 });
12
13 // or..
14
15 app.factory('bar', function(a) {
```

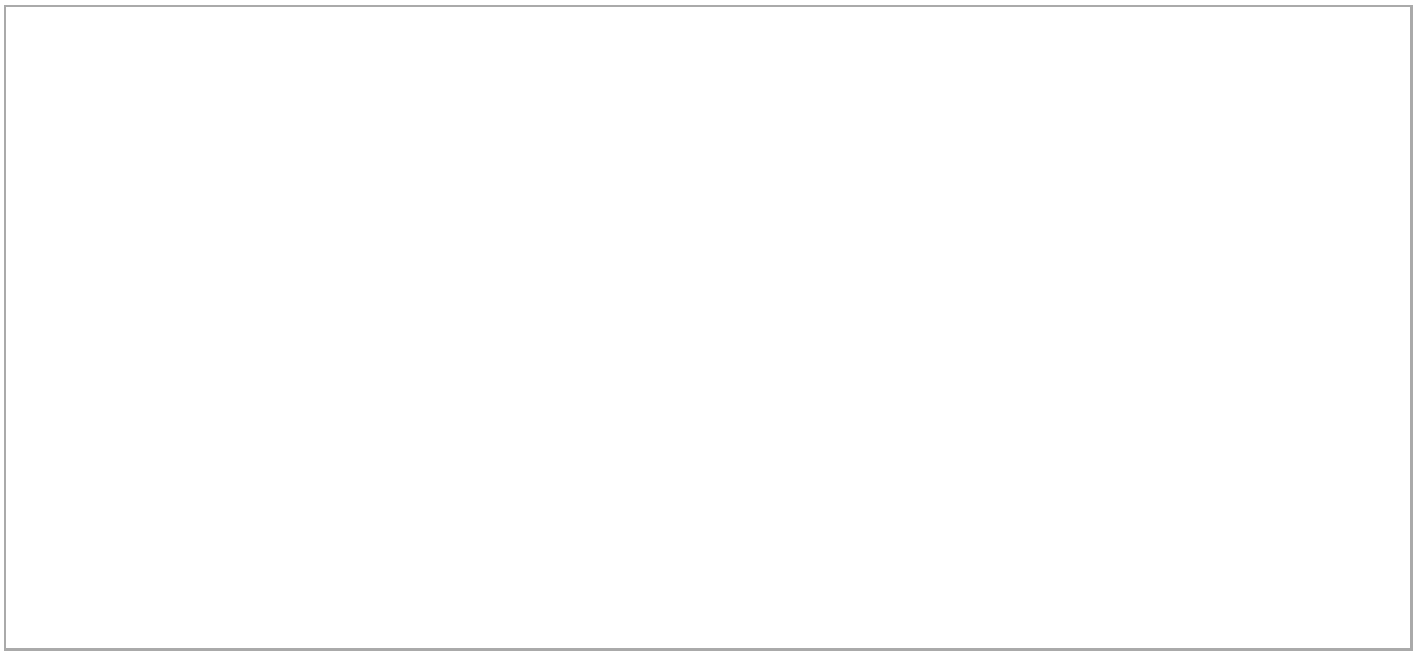
```
16   return a * 2;
17 });
```

As you see, we moved our `provider` `$get` function into a `factory` so we have what we had on the first `provider` example but with a much simpler syntax. In fact, internally a `factory` is a `provider` with only the `$get` function.

As I said before, all types are singleton, so if we modify `foo.variable` in one place, the other places will have that change too.

We can inject everything but providers on a `factory` and we can inject it everywhere except on the `provider` constructor and config functions.

Try it



Value

`Factory` is good, but what if I just want to store a simple value? I mean, no injections, just a simple value or object. Well angular has you covered with the `value` service:

Example:

```
1 app.value('foo', 'A simple value');
```

Internally a `value` is just a factory. And since it is a `factory` the same injection rules applies, AKA can't be injected into `provider` constructor or config functions.

Try it

Service

So having the complex `provider`, the more simple `factory` and the `value` services, what is the `service` service? Let's see an example first:

Example:

```
app.service('foo', function() {
  var thisIsPrivate = "Private";
  this.variable = "This is public";
  this.getPrivate = function() {
    return thisIsPrivate;
  };
});
```

The `service` service works much the same as the `factory` one. The difference is simple: The `factory` receives a function that gets called when we create it and the `service` receives a constructor function where we do a `new` on it (actually internally is uses `Object.create` instead of `new`).

In fact, it is the same thing as doing this:

```
app.factory('foo2', function() {
  return new Foobar();
});


function Foobar() {
  var thisIsPrivate = "Private";
  this.variable = "This is public";
  this.getPrivate = function() {
    return thisIsPrivate;
  };
}
```

Foobar is a constructor function and we instantiate it in our factory when angular processes it for the first time and then it returns it. Like the service, Foobar will be instantiated only once and the next time we use the factory it will return the same instance again.

If we already have the class and we want to use it in our service we can do that like the following:

```
1 app.service('foo3', Foobar);
```

If you're wondering, what we did on foo2 is actually what angular does with services internally. That means that service is actually a factory and because of that, same injection rules applies.

Try it

JS Bin Save | HTML | CSS | JavaScript | Console | Output | Help

```
public variable: {{foo.variable}}
private variable (through getter): {{foo.getPrivate()}}
```

Auto-run JS Run with JS

Constant

Then, you're expecting me to say that a constant is another subtype of provider like the others, but this one is not. A constant works much the same as a value as we can see here:

Example:

```
1 app.constant('fooConfig', {
2   config1: true,
3   config2: "Default config2"
4 });
```

So… what's the difference then? A constant can be injected everywhere and that includes provider constructor and config functions. That is why we use constant services to create default configuration for directives, because we can modify

those configuration on our config functions.

You are wondering why it is called `constant` if we can modify it and well that was a design decision and I have no idea about the reasons behind it.

Try it

JS Bin Save | HTML | CSS | JavaScript | Console | Output | Help

```
config1: {{fooConfig.config1}}
config2: {{fooConfig.config2}}
```

Auto-run JS Run with JS

Bonus 1: Decorator

So you decided that the `foo` service I sent to you lacks a `greet` function and you want it. Will you modify the `factory`? No! You can decorate it:

```
1 app.config(function($provide) {
2   $provide.decorator('foo', function($delegate) {
3     $delegate.greet = function() {
4       return "Hello, I am a new function of 'foo'";
5     };
6
7     return $delegate;
8   });
9 });
```

`$provide` is what Angular uses internally to create all the services. We can use it to create new services if we want but also to decorate existing services. `$provide` has a method called `decorator` that allows us to do that. `decorator` receives the name of the service and a callback function that receives a `$delegate` parameter. That `$delegate` parameter is our original service instance.

Here we can do what we want to decorate our service. In our case, we added a `greet` function to our original service. Then we return the new modified service.

Now when we consume it, it will have the new `greet` function as you will see in the `Try it`.

The ability to decorate services comes in handy when we are consuming 3rd party services and we want to decorate it without having to copy it in our project and then doing there the modifications.

Note: The `constant` service cannot be decorated.

Try it

JS Bin Save | HTML | CSS | JavaScript | Console | Output | Help

```
public variable: {{foo.variable}}
private variable (through getter): {{foo.getPrivate()}}
greet: {{foo.greet()}}
```

Auto-run JS Run with JS

Bonus 2: Creating new instances

Our services are singleton but we can create a singleton factory that creates new instances. Before you dive deeper, keep in mind that having singleton services is the way to go and we don't want to change that. Said that, in the rare cases you need to generate new instances, you can do that like this:

```
1  // Our class
2  function Person( json ) {
3    angular.extend(this, json);
4  }
5
6  Person.prototype = {
7    update: function() {
8      // Update it (With real code :P)
9      this.name = "Dave";
10     this.country = "Canada";
11   }
12 };
13
14 Person.getById = function( id ) {
15   // Do something to fetch a Person by the id
16   return new Person({
17     name: "Jesus",
18     country: "Spain"
19   });
20 };
21
22 // Our factory
```

```
23 app.factory('personService', function() {
24   return {
25     getById: Person.getById
26   };
27 });
```

Here we create a `Person` object which receives some json to initialize the object. Then we created a function in our prototype (functions in the prototype are for the instances of the `Person`) and a function directly in `Person` (which is like a class function).

So we have a class function that will create a new `Person` object based on the id that we provide (well, it will in real code) and every instance is able to update itself. Now we just need to create a service that will use it.

Every time we call `personService.getById` we are creating a new `Person` object, so you can use this service in different controllers and even when the factory in a singleton, it generates new objects.

Kudos to [Josh David Miller] for his example.

Try it

Bonus 3: Coffeescript

Coffeescript can be handy with services since they provide a prettier way to create classes. Let's see an example of the `Bonus 2` using Coffeescript:

```
1 app.controller 'MainCtrl', ($scope, personService) ->
2   $scope.aPerson = personService.getById(1)
3
4 app.controller 'SecondCtrl', ($scope, personService) ->
5   $scope.aPerson = personService.getById(2)
6   $scope.updateIt = () ->
```

```
7      $scope.aPerson.update()
8
9  class Person
10
11   constructor: (json) ->
12     angular.extend @, json
13
14   update: () ->
15     @name = "Dave"
16     @country = "Canada"
17
18   @getById: (id) ->
19     new Person
20       name: "Jesus"
21       country: "Spain"
22
23 app.factory 'personService', () ->
24   {
25     getById: Person.getById
26   }
```

It is prettier now in my humble opinion.

Try it

JS Bin Save | HTML | CSS | CoffeeScript | Console | Output | Help

{{aPerson.name}} - {{aPerson.country}}

{{aPerson.name}} - {{aPerson.country}} Update it

Auto-run JS Run with JS

NOTE: This last one, being Coffeescript seems to fail a little bit with JSbin. Go to the Javascript tab and select Coffeescript to make it work.

Conclusion

Services are one of the coolest features of Angular. We have a lot of ways to create them, we just need to pick the correct one for our use cases and implement it.

If you found any issue or you think that this can be improved, please leave an issue or pull request at Github. In any case, a comment will be appreciated :).

Posted by Jesus Rodriguez Aug 12th, 2013 services

« Removing the unneeded watches my workflow with lineman »

Comments

My Book

Twitter

Follow @Foxandxss

Donate

If you feel I was helpful enough, consider a donation to pay the server.

Recent Posts

- Using Angular 1.x with ES6 and Webpack
- Why will Angular 2 rock?
- My book is done!
- An introduction to Angular 2
- My book is out!

Categories

- $apply (1)
- $digest (1)
- $watch (2)
- advanced (1)
- angular2 (2)
- authentication (3)
- beginners (2)
- Blog (2)
- book (2)
- Book (1)
- directives (3)
- ES6 (1)
- experiment (1)
- filters (1)
- laravel (2)
- plunker (1)
- review (1)
- scopes (1)
- services (2)
- tip (3)
- unit test (8)
- Webpack (1)
- workflow (5)

Tag Cloud

$apply(1) $digest(1) $watch(2) advanced(1) angular2(2) authentication(3) beginners(2) Blog(2) book(2) Book(1) directives(3) ES6(1) experiment(1) filters(1) laravel(2) plunker(1) review(1) scopes(1) services(2) tip(3) unit test(8) Webpack(1) workflow(5)