Angular .service() or .factory(), the actual answer

Mar 11, 2016

6 mins read

Edit post

I was giving a workshop earlier this week, and as the workshop came to a close and people were leaving, an attendee asked if I could explain the difference between a factory and service. Everyone seemed to pause and stop packing up their things, and sat back down to listen to the additional 15 minute showdown on the API differences. So in my words this the answer to `.factory()` and `.service()` .

Table of contents

So, you may be thinking once again we know the answer already to this, however the masses of answers online simply show how you can do the same thing with `.factory()` and `.service()` , without explaining what you can do with the APIs. They also paste in the same code snippet from the Angular source saying that a Factory is just a Service and Service is just a Factory but there's one line of code different and so on, this is meaningless to teaching the developer how to actually use

the APIs - so let's give them a little perspective that answer questions I've been asked online and in person.

Service

Let's start with `.service()`. First, a quick example of a Service with a basic `$http.get()` implementation:

```
function InboxService($http) {
  this.getEmails = function getEmails() {
    return $http.get('/emails');
  };
}
angular
  .module('app')
  .service('InboxService', InboxService);
```

We can then inject this into a Controller and fetch some emails:

```
function InboxController(InboxService) {
  InboxService
    .getEmails()
    .then(function (response) {
      // use response
    });
}
angular
  .module('app')
  .controller('InboxController', InboxController);
```

Delightful. We all know this, don't we.

So, what is a service?

A Service is just a function for the business layer of the application, it's just a simple function. It acts as a `constructor` function and is invoked once at runtime with `new`, much like you would with plain JavaScript (Angular is just calling a `new` instance under the hood for us).

Think about it being just a constructor method, you can add public methods to it and that's pretty much it. It's a simple API, don't overthink it.

When should you use it?

Use a `.service()` when you want to just create things that act as public APIs, such as the `getEmails` method we defined above. Use it for things you would use a `constructor` for. Unfortunately if you don't like using constructors in JavaScript, there isn't much flexibility if you want to just use a simple API pattern. However, you can achieve identical results by using the `.factory()` method, but the `.factory()` method is *much* different, let's take a look.

Factory

Next, the confusing `.factory()` method. A factory is not just "another way" for doing services, anyone that tells you that is wrong. It *can* however, give you the same capabilities of a `.service()`, but is much more powerful and flexible.

A factory is not just a "way" of returning something, a factory is in fact a design pattern. Factories create Objects, that's it. Now ask yourself: what type of Object do I want? With `.factory()`, we can create various Objects, such as new Class instances (with `.prototype` or ES2015 Classes), return Object literals, return functions and closures, or even just return a simply String. You can create whatever you like, that's the rule.

Let's take a look at some examples that usually get shown with these "factory versus service" articles:

```
// Service
function InboxService($http) {
  this.getEmails = function getEmails() {
    return $http.get('/emails');
  };
}
angular
  .module('app')
  .service('InboxService', InboxService);

// Factory
function InboxService($http) {
  return {
    getEmails: function () {
      return $http.get('/emails');
    }
  };
}
angular
  .module('app')
  .factory('InboxService', InboxService);
```

Yes yes, we can inject both versions into a Controller and call them in identical ways by using `InboxService.getEmails();` . This is *usually* where all explanations end, but come on, you don't *need* to stop there. A Factory is flexible, we can return anything. A factory*creates* things, or at least can do. With the above example it doesn't really create anything, it just returns an Object literal. Let's add a further example below then walkthrough the other things we can do with the `.factory()` method:

Factory: Object Literals

As above, we can define some functions on an Object and return them. This is just basic module pattern implementation:

```
function InboxService($http) {
  return {
```

```
    getEmails: function () {
      return $http.get('/emails');
    }
  };
}
angular
  .module('app')
  .factory('InboxService', InboxService);
```

We can also use the revealing module pattern as a variant:

```
function InboxService($http) {
  function getEmails() {
    return $http.get('/emails');
  }
  return {
    getEmails: getEmails
  };
}
angular
  .module('app')
  .factory('InboxService', InboxService);
```

I much prefer this syntax as it allows for nicer code indentation and better comments above the function definitions.

Factory: Closures

We can return a closure from a `.factory()` also, which we may want to do to purely expose a single function call.

```
function InboxService($http) {
  return function () {
```

```
    return $http.get('/emails');
  };
}
angular
  .module('app')
  .factory('InboxService', InboxService);
```

We would use it like so inside the Controller:

```
function InboxController(InboxService) {
  InboxService().then(function (response) {
      // use response
    });
}
angular
  .module('app')
  .controller('InboxController', InboxController);
```

Using the `$http` example here isn't a great example, however demonstrates what we can return from a `.factory()` call. As we're returning a closure, we can also return other things inside the closure that make using a closure the *sensible* options:

```
function InboxService($http) {
  return function (collection) {
    return function (something) {
      collection.forEach(function (item) {
        // manipulate each collection `item` and also
        // use the `something` variable
      });
    };
  };
}
angular
```

```
  .module('app')
  .factory('InboxService', InboxService);
```

Some example usage:

```
function InboxController(InboxService) {
  var myService = InboxService([{...},{...}]);
  var callTheClosure = myService('foo');
}
angular
  .module('app')
  .controller('InboxController', InboxController);
```

The above Controller example passes in a collection, then return a closure to pass further information into. There are use cases for this you've likely come across.

Factory: Creating Constructors/instances

We've learned that `.service()` *is* the `constructor` function, but is *only* invoked once, however that doesn't mean we cannot create `constructor` Objects elsewhere that we can call with `new` . We can rock this pattern inside a `.factory()` :

```
function PersonService($http) {
  function Person() {
    this.foo = function () {

    };
  }
  Person.prototype.bar = function () {

  };
  return Person;
```

```
}
angular
  .module('app')
  .factory('PersonService', PersonService);
```

Now, we have awesome power. Let's inject our `PersonService` into a Controller and call `new PersonService();` to get a new instance:

```
function PersonController(PersonService) {
  // new instance!
  var newPerson = new PersonService();
  console.log(newPerson);
}
angular
  .module('app')
  .controller('PersonController', PersonController);
```

The output of the `newPerson` gives us access to the `constructor` `foo` property and also `__proto__` where we can access `bar: function () {}` . This is what factories do, they create new Objects in ways you want them to.

Conclusion

Both `.service()` and `.factory()` are *both* singletons as you'll only get one instance of each Service regardless of what API created it.

Remember that `.service()` is just a Constructor, it's called with `new` , whereas `.factory()` is just a function that returns a value.

Using `.factory()` gives us much more power and flexibility, whereas a `.service()` is essentially the "end result" of a `.factory()` call. The `.service()` gives us the returned value by calling `new` on the function,

which can be limiting, whereas a `.factory()` is one-step before this compile process as we get to choose which pattern to implement and return.

Get updates