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

Mar 11, 2016 5 mins read

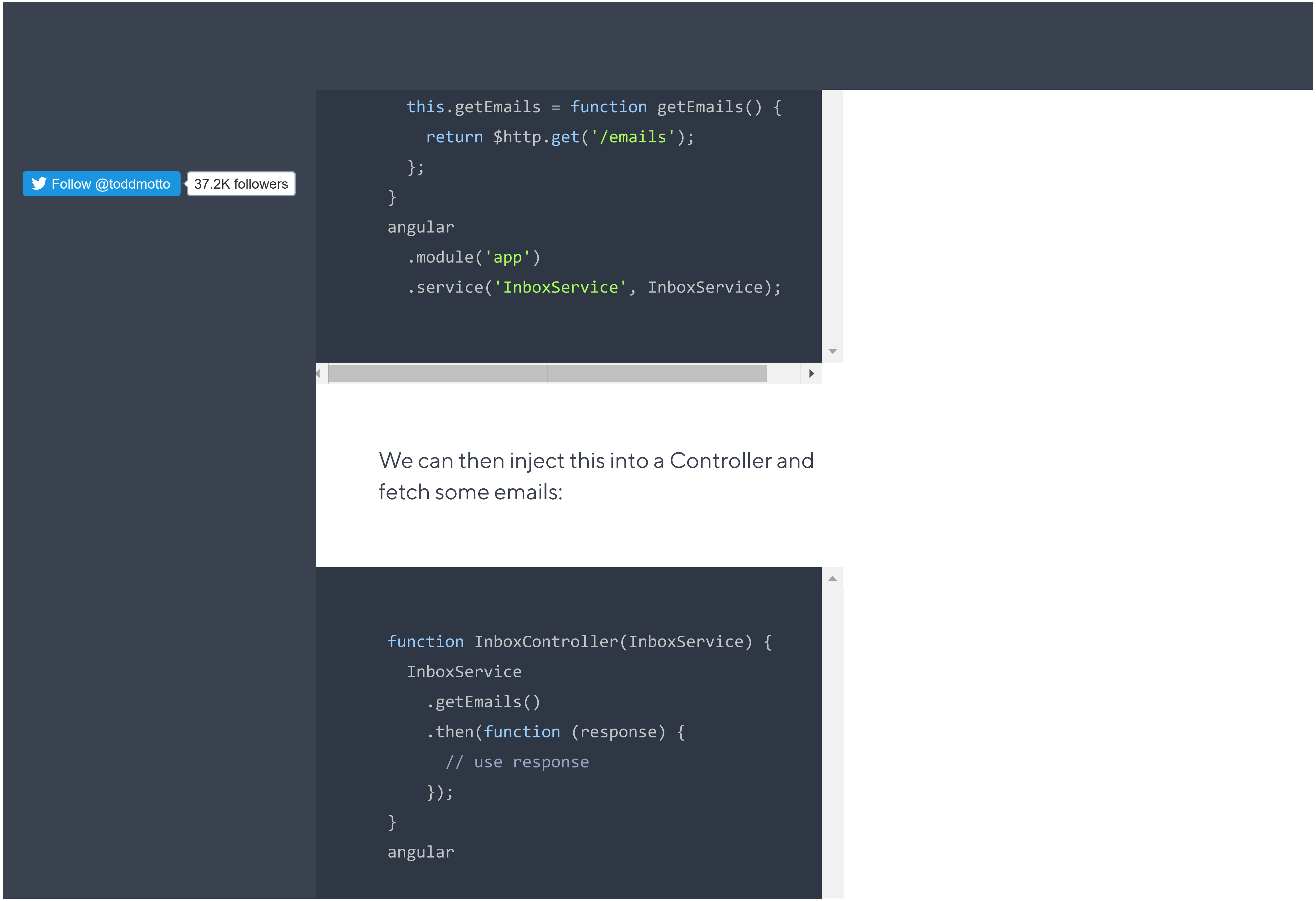
I was giving a workshop earlier this week, and as the workshop came to a close and people were leaving, an attendee asked if I could explain the difference between a factory and service. Everyone seemed to pause and stop packing up their things, and sat back down to listen to the additional 15 minute 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:



```
    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
```

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.

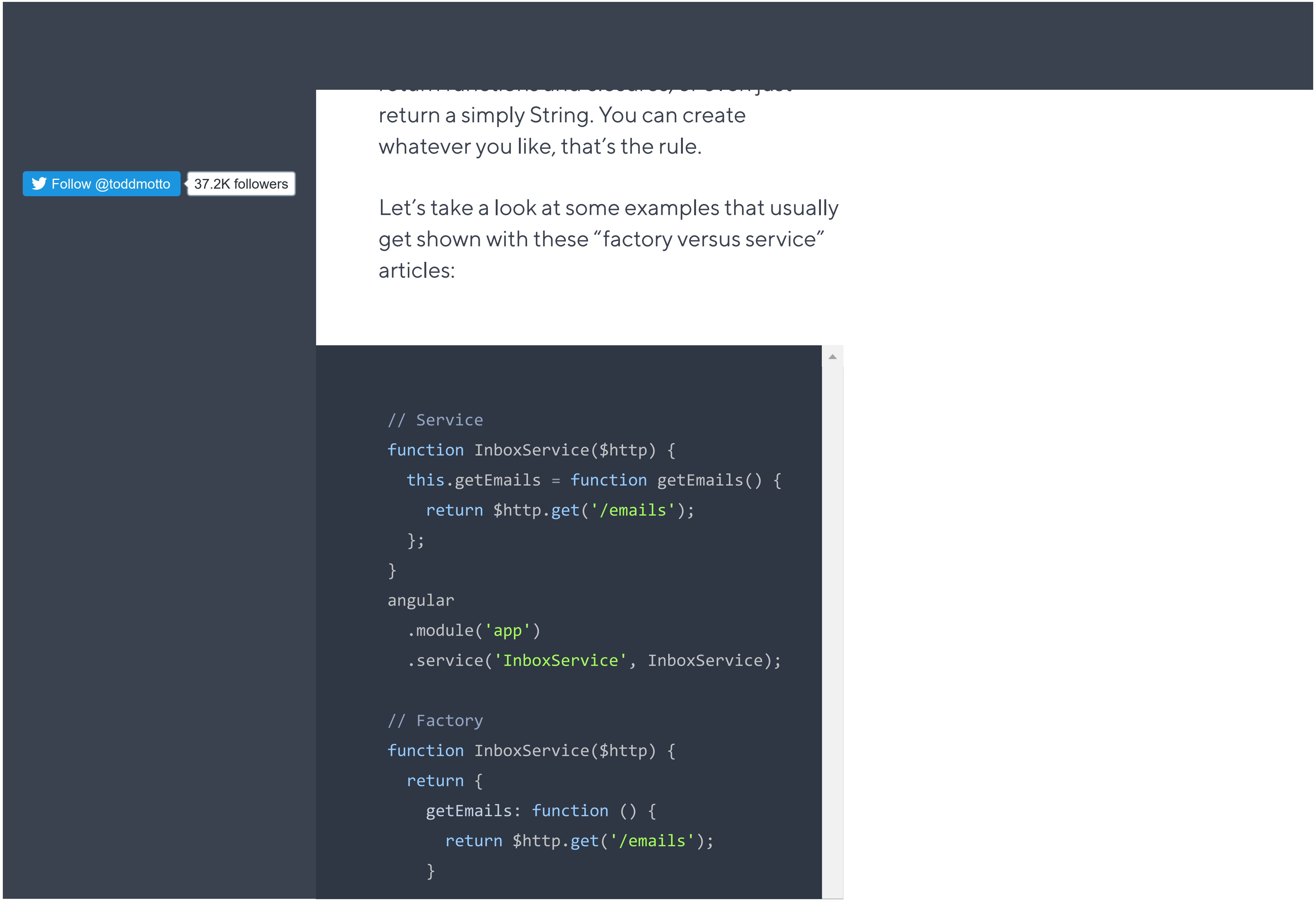
When should you use it?

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,



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');
    }
  .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:

```
  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() {
```

Search posts

```
  }
  return {
    getEmails: getEmails
  };
```

Todd Motto
Owner, Ultimate Angular

```
    .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
```

We would use it like so inside the Controller:

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

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

```
function InboxService($http) {
  return function (collection) {
    return function (something) {
      collection.forEach(function (item) {
        // manipulate each collection `item`
        // use the `something` variable
      });
    };
  };
}
angular
  .module('app')
  .factory('InboxService', InboxService);
```

Some example usage:

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

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 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:

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

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

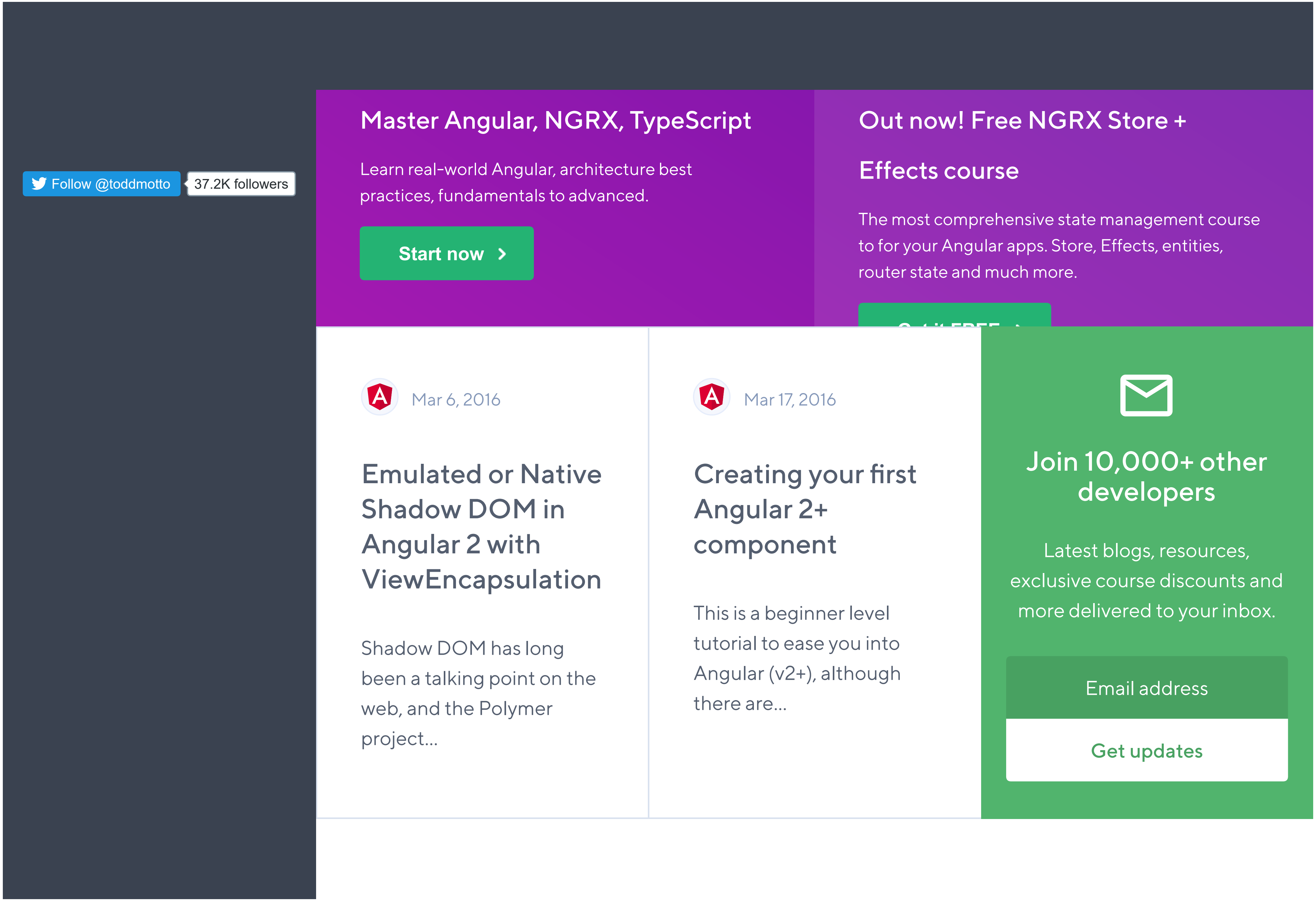
Both `.service()` and `.factory()` are *both* singletons as you'll only get one instance of each Service regardless of what API created it.

`.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.

Enjoy this post? Share it!

Lots of time and effort goes into all my blogs, resources and demos, I'd love if you'd spare a moment to share them!



Master Angular, NGRX, TypeScript

Learn real-world Angular, architecture best practices, fundamentals to advanced.

Out now! Free NGRX Store + Effects course

The most comprehensive state management course to for your Angular apps. Store, Effects, entities, router state and much more.

Mar 6, 2016

Emulated or Native Shadow DOM in Angular 2 with ViewEncapsulation

Shadow DOM has long been a talking point on the web, and the Polymer project...

Mar 17, 2016

Creating your first Angular 2+ component

This is a beginner level tutorial to ease you into Angular (v2+), although there are...

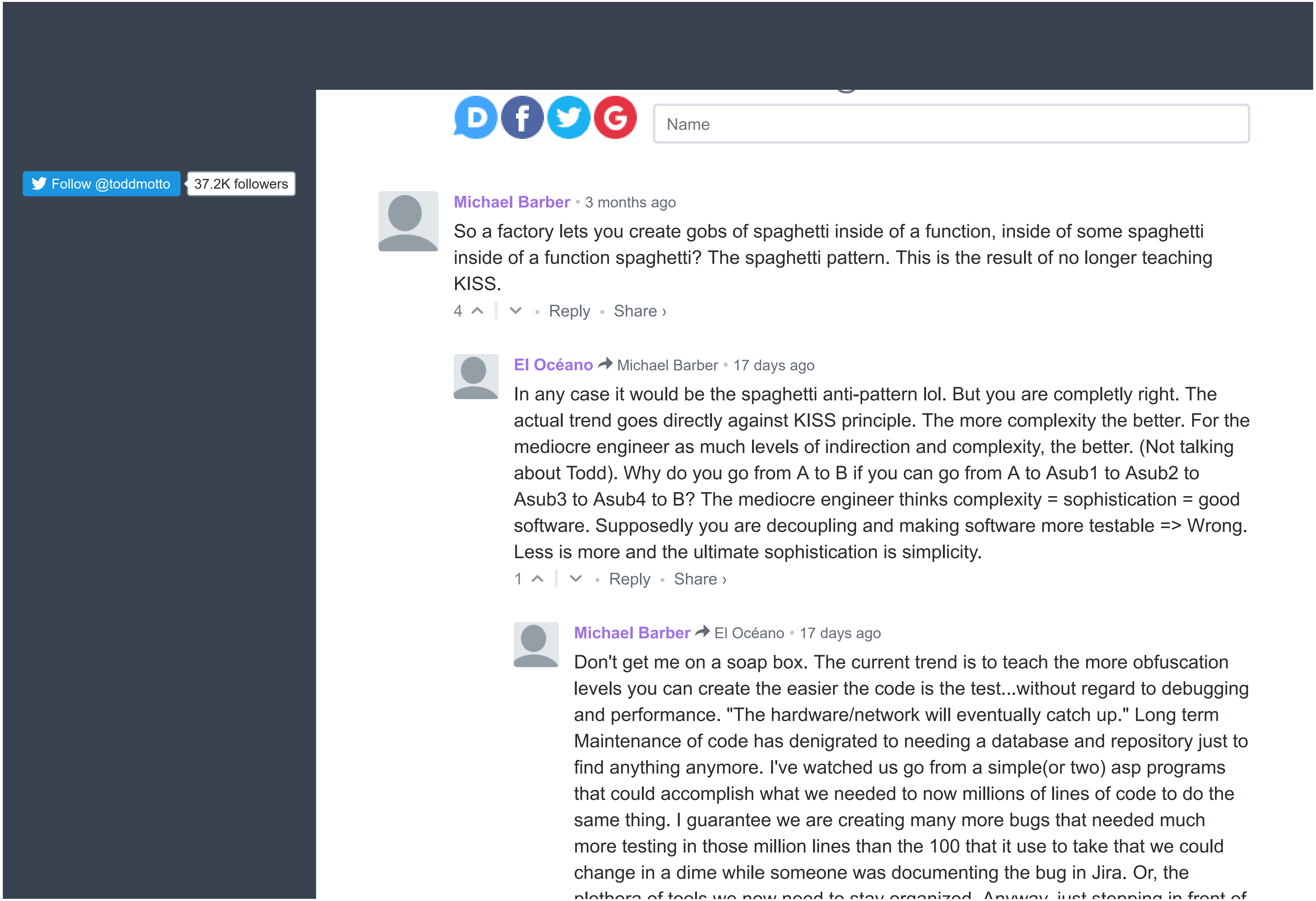
Join 10,000+ other developers

Latest blogs, resources, exclusive course discounts and more delivered to your inbox.

Follow @toddmotto 37.2K followers

27 Comments Todd Motto 1 Login

♡ Recommend 24 Share Sort by Newest



Michael Barber • 3 months ago

So a factory lets you create gobs of spaghetti inside of a function, inside of some spaghetti inside of a function spaghetti? The spaghetti pattern. This is the result of no longer teaching KISS.

El Océano → Michael Barber • 17 days ago

In any case it would be the spaghetti anti-pattern lol. But you are completly right. The actual trend goes directly against KISS principle. The more complexity the better. For the mediocre engineer as much levels of indirection and complexity, the better. (Not talking about Todd). Why do you go from A to B if you can go from A to Asub1 to Asub2 to Asub3 to Asub4 to B? The mediocre engineer thinks complexity = sophistication = good software. Supposedly you are decoupling and making software more testable => Wrong. Less is more and the ultimate sophistication is simplicity.

Michael Barber → El Océano • 17 days ago

Don't get me on a soap box. The current trend is to teach the more obfuscation levels you can create the easier the code is the test...without regard to debugging and performance. "The hardware/network will eventually catch up." Long term Maintenance of code has denigrated to needing a database and repository just to find anything anymore. I've watched us go from a simple(or two) asp programs that could accomplish what we needed to now millions of lines of code to do the same thing. I guarantee we are creating many more bugs that needed much more testing in those million lines than the 100 that it use to take that we could change in a dime while someone was documenting the bug in Jira. Or, the plethora of tools we now need to stay organized. Anyway, just stepping in front of

Follow @toddmotto 37.2K followers

place. I think we have basically knee jerked from one gigantic main program with 500 goto statements to the opposite of that.... I'm NOT advocating we go back to one program to do everything. However, you also have some basic laws of physics in play here- Entropy (3rd law of thermodynamics which I believe is actually the fundamental law of the universe. It is in essence good -vs- bad in it's most element form, light vs dark, hot vs cold, creation vs destruction, and lastly the most important in what we do --> order vs disorder. KISS is just the that expression of Entropy.

1 ^ | ⌄ • Reply • Share ›

Louai Drissi • 3 months ago

Thank you for the article and it really gave me a better perspective and when to use factory and service.

^ | ⌄ • Reply • Share ›

bathos • a year ago

I don’t agree with the conclusions here personally. In JS, constructors are themselves a type of object factory that implicitly (with `function`) or more explicitly (with `class`) involves a free prototype to build up. If a constructor is only ever going to be instantiated once (or especially if, as in the first example here, it doesn’t make use of its own prototype at all), I’d argue that its being a constructor/class is an unhelpful obfuscation. The less complex pattern of returning a simple object from a callable function is a more direct (and cheaper) way to express the same effective result.

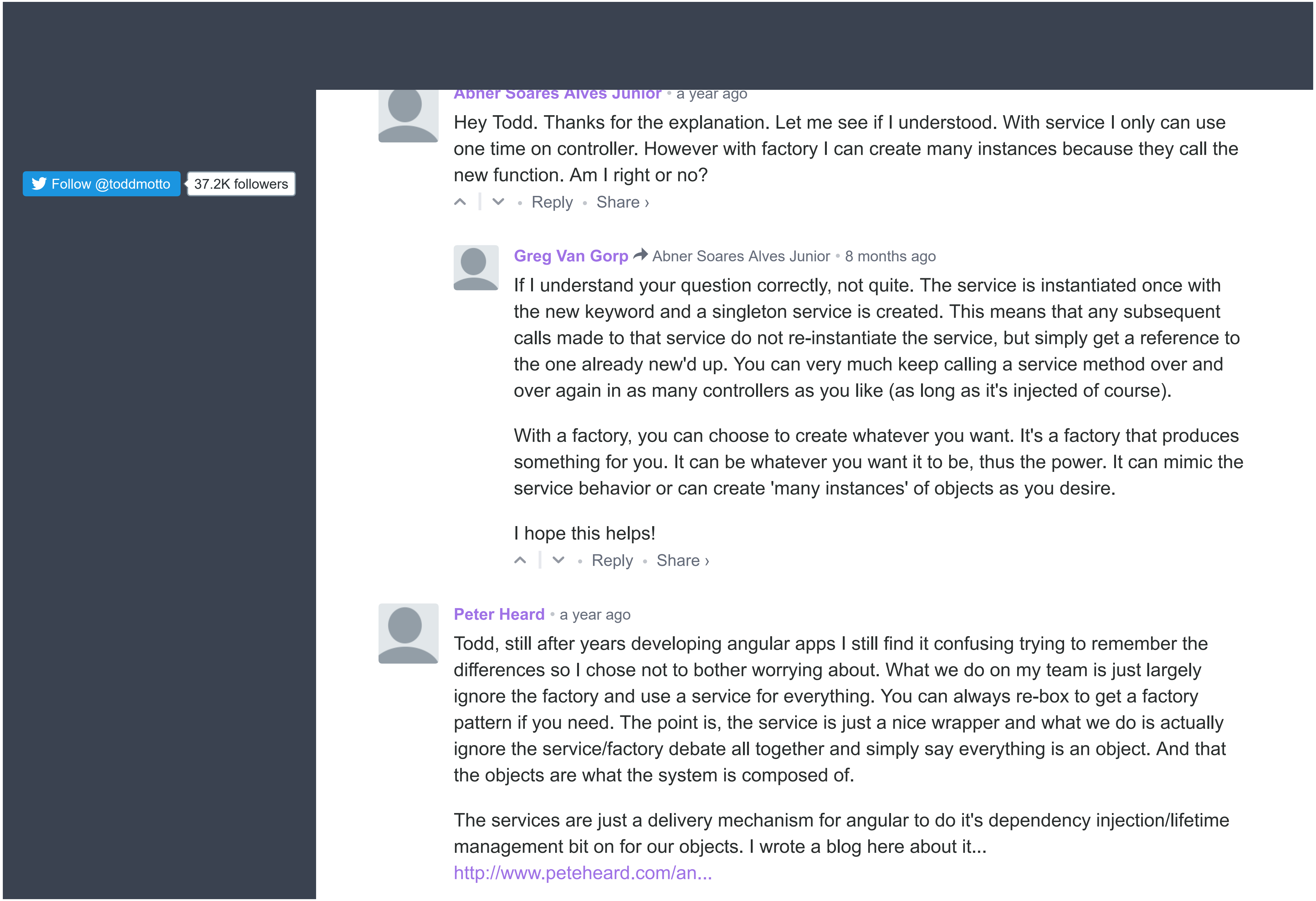
59 ^ | ⌄ • Reply • Share ›

Sahil Lakhwani • a year ago

Thanks for this. The explanation is as clear as the UI of this page.

1 ^ | ⌄ • Reply • Share ›

Michal • a year ago



Abner Soares Alves Junior • a year ago

Hey Todd. Thanks for the explanation. Let me see if I understood. With service I only can use one time on controller. However with factory I can create many instances because they call the new function. Am I right or no?

^ | ˅ • Reply • Share ›

Greg Van Gorp ➦ Abner Soares Alves Junior • 8 months ago

If I understand your question correctly, not quite. The service is instantiated once with the new keyword and a singleton service is created. This means that any subsequent calls made to that service do not re-instantiate the service, but simply get a reference to the one already new'd up. You can very much keep calling a service method over and over again in as many controllers as you like (as long as it's injected of course).

With a factory, you can choose to create whatever you want. It's a factory that produces something for you. It can be whatever you want it to be, thus the power. It can mimic the service behavior or can create 'many instances' of objects as you desire.

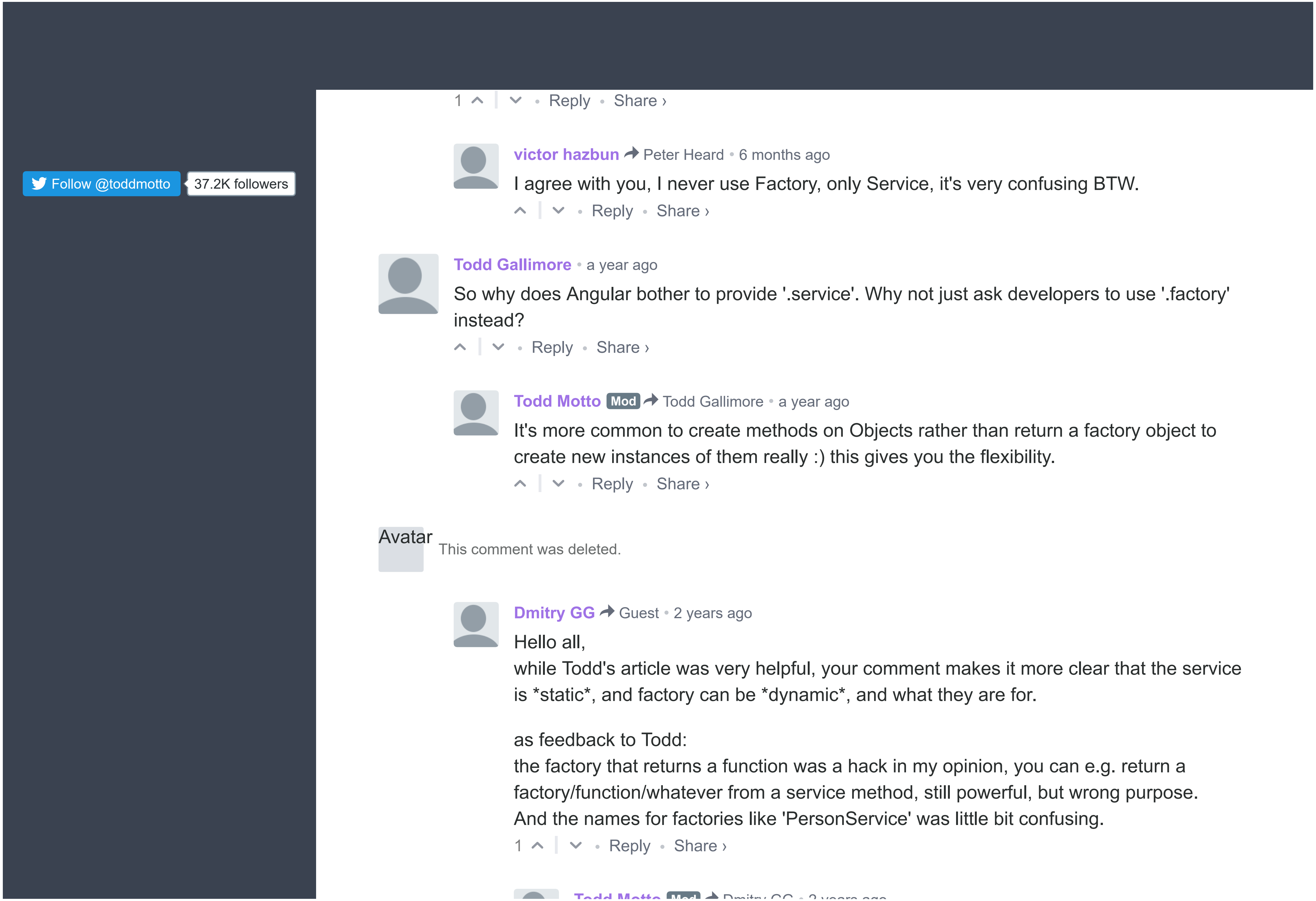
I hope this helps!

^ | ˅ • Reply • Share ›

Peter Heard • a year ago

Todd, still after years developing angular apps I still find it confusing trying to remember the differences so I chose not to bother worrying about. What we do on my team is just largely ignore the factory and use a service for everything. You can always re-box to get a factory pattern if you need. The point is, the service is just a nice wrapper and what we do is actually ignore the service/factory debate all together and simply say everything is an object. And that the objects are what the system is composed of.

The services are just a delivery mechanism for angular to do it's dependency injection/lifetime management bit on for our objects. I wrote a blog here about it...
http://www.peteheard.com/an...



Follow @toddmotto 37.2K followers

1 ^ | v • Reply • Share ›

victor hazbun ➦ Peter Heard • 6 months ago

I agree with you, I never use Factory, only Service, it's very confusing BTW.

^ | v • Reply • Share ›

Todd Gallimore • a year ago

So why does Angular bother to provide '.service'. Why not just ask developers to use '.factory' instead?

^ | v • Reply • Share ›

Todd Motto Mod ➦ Todd Gallimore • a year ago

It's more common to create methods on Objects rather than return a factory object to create new instances of them really :) this gives you the flexibility.

^ | v • Reply • Share ›

Avatar This comment was deleted.

Dmitry GG ➦ Guest • 2 years ago

Hello all,
while Todd's article was very helpful, your comment makes it more clear that the service is *static*, and factory can be *dynamic*, and what they are for.

as feedback to Todd:
the factory that returns a function was a hack in my opinion, you can e.g. return a factory/function/whatever from a service method, still powerful, but wrong purpose.
And the names for factories like 'PersonService' was little bit confusing.

1 ^ | v • Reply • Share ›

Todd Motto Mod ➦ Dmitry GG • 2 years ago

Ryan K. Davidson ➦ Todd Motto • 2 years ago

Anything wrong with "PersonFactory"? I feel like any other word besides "Service" is good... using that completely complicates the ideas you had simplified.
IMHO you should change the last example code... undermines the great point of this article!

I also like the idea of `PersonFactory.create(...)`, so the factory is responsible for creating `new` People objects, gives it the opportunity to do caching or object pooling etc. A singleton to produce instances of other things, I guess.

3 ^ | ˅ • Reply • Share ›

Ruslan Stelmachenko ➦ Ryan K. Davidson • a year ago

I agree. Creating person instances using `new PersonService()` is confusing API. :) this should be:

1. `new Person()` (the factory itself should be named `Person` and return constructor function of Person objects)

or

2. `personFactory.create()` (the factory should be named `personFactory` and return a factory instance instead of constuctor function).

1 ^ | ˅ • Reply • Share ›

Martin Spierings • 2 years ago

I still find the examples with the http calls to be unneeded for these kinds of explanations.

Its just simple: a service is not persistent, factories are.

Follow @toddmotto 37.2K followers

Factories are really handy to store data between pages, directives and such. You could store the user-data in a factory or keep data from a multi-page form. Where in the service you would probably set the $http calls to submit the data as that doesn't need to be persistent (but is handy to have all calls in a single file for maintenance).

6 ⌃ | ⌄ • Reply • Share ›

Raymon Schouwenaar ➦ Martin Spierings • a month ago

This is the most simply and clear explanation I have ever found 🙏👍

1 ⌃ | ⌄ • Reply • Share ›

kittybytes ➦ Martin Spierings • 9 months ago

holy mother of god, it only took 5 hours of researching this topic to find this short blurb which literally is the only thing I have read so far to help clarify what the damn difference is between factory and service

⌃ | ⌄ • Reply • Share ›

Saurabh Saxena ➦ kittybytes • 8 months ago

Hi I tried following code but both service and factory incrementing the values.

```
app.service("TestService", function ()
{
var x = 5;

this.checkvalService = function() {
x = x + 5;
return (x)
}
})

app.factory("TestFactory", function ()
{
```

```
return (y)
```

see more

^ | ˅ • Reply • Share ›

sminutoli_etermax ➦ Saurabh Saxena • 8 months ago

Your code must be throwing an exception, like here... https://plnkr.co/edit/VoqVg...

A factory must return an object, this is used only in Services.

The main difference between a Service and a Factory is that Angular calls a Service via apply() providing a context as this and the Factory invoking the plain function. That´s all.

Isn´t about closures. Isn´t about nothing but sugar syntax for a common pattern: you want to get a Single Object directly, using `this` inside a function (Service) or doing some calc in the middle step, avoiding `this` (Factory).

A working example:
https://plnkr.co/edit/rZjuk...