Services

Angular services are substitutable objects that are wired together using [dependency injection (DI)](https://docs.angularjs.org/guide/di). You can use services to organize and share code across your application.

Services provide a method for us to keep data around for the lifetime of the app and communicate across controllers in a consistent manner.

Angular services are:

* Lazily instantiated – Angular only instantiates a service when an application component depends on it.
* Singletons – Each component dependent on a service gets a reference to the single instance generated by the service factory.

Angular offers several useful services (like [$http](https://docs.angularjs.org/api/ng/service/$http)), but for most applications you'll also want to [create your own](https://docs.angularjs.org/guide/services#creating-services).

**Note:** Like other core Angular identifiers, built-in services always start with $ (e.g. $http).

## Using a Service

To use an Angular service, you add it as a dependency for the component (controller, service, filter or directive) that depends on the service. Angular's [dependency injection](https://docs.angularjs.org/guide/di) subsystem takes care of the rest.

<!DOCTYPE html>

<html ng-app="IG">

<head lang="en">

<meta charset="UTF-8">

<title>Services in AngularJS</title>

<script src="Scripts/Vendor/angular.js"></script>

<script src="Scripts/external.js"></script>

</head>

<body>

<div ng-controller="MyController">

<p>Let's try this simple notify service, injected into the controller...</p>

<input ng-init="message='test'" ng-model="message" >

<button ng-click="callNotify(message);">NOTIFY</button>

<p>(you have to click 2 times to see an alert)</p>

</div>

</body>

</html>

//defining module

var app = angular.module('IG', [])

// Controller , where we inject notify services

app.controller('MyController', ['$scope','notify', function ($scope, notify) {

$scope.callNotify = function(msg) {

notify(msg);

};

}]);

//Services using Factory

app.factory('notify', ['$window', function(win) {

var msgs = [];

return function(msg) {

msgs.push(msg);

if (msgs.length == 2) {

win.alert(msgs.join("\n"));

msgs = [];

}

};

}]);

## Creating Services

There are multiple way to create services in Angular.

While the most common method for registering a service with our Angular app is through the factory() method, there are some other APIs we can take advantage of in certain situations to shorten our code.

The five different methods for creating services are:

1. factory()
2. service()
3. constant()
4. value()
5. provider()

**factory() 🡺**

As we’ve seen, the factory() method is a quick way to create and configure a service.

The factory() function takes two arguments:

• name (string)

This argument takes the name of the service we want to register.

• getFn (function)

This function runs when Angular creates the service.

Note: It will return function return value

var app = angular.module('IG', [])

app.factory('myService', function () {

return {

'username': 'auser'

}

});

The getFn will be invoked once for the duration of the app lifecycle, as the service is a singleton

object. As with other Angular services, when we define our service, getFn can take an array or a

function that will take other injectable objects.

The getFn function can return anything from a primitive value to a function to an object (similar to

the value() function).

var app = angular.module('IG', [])

app.factory('myService', ['$http', function ($http) {

return {

getUserEvents: function (username) {

// ...

}

}

} ]);

**service() 🡺**

If we want to register an instance of a service using a constructor function, we can use service(),

which enables us to register a constructor function for our service object.

The service() method takes two arguments:

• name (string)

This argument takes the name of the service instance we want to register.

• constructor (function)

Here is the constructor function that we’ll call to instantiate the instance.

The service() function will instantiate the instance using the new keyword when creating the

instance.

Note: It will return actual function value.

var app = angular.module('IG', [])

app.service('personService', Person);

var Person = function ($http) {

this.getName = function () {

return $http({

method: 'GET',

url: '/api/user'

});

};

this.SaveName = function () {

return $http({

method: 'GET',

url: '/api/user'

});

};

};

**constant() 🡺**

It’s possible to register an existing value as a service that we can later inject into other parts of our app as a service. For instance, let’s say we want to set an apiKey for a back-end service. We can store that constant value using constant().

The constant() function takes two arguments:

• name (string)

This argument is the name with which to register the constant.

• value (constant value)

This argument gives the value to register as the constant.

The constant() method returns a registered service instance.

var app = angular.module('IG', [])

app.constant('apiKey', '123123123')

We can now inject this value into a configuration function just like any other service:

var app = angular.module('IG', [])

app.controller('MyController',

function ($scope, apiKey) {

// We can use apiKey as a constant

// string as 123123123 set from above

$scope.apiKey = apiKey;

});

**value() 🡺**

If the return value of the $get method in our service is a constant, we don’t need to define a fullblown

service with a more complex method. We can simply use the value() function to register the service.

The value() method accepts two arguments:

• name (string)

Once again, this argument gives the name with which we want to register the value.

• value (value)

We’ll return this value as the injectable instance.

The value() method returns a registered service instance for the name given.

var app = angular.module('IG', [])

app.value('apiKey', '123123123');

When to Use Value or Constant

The major difference between the value() method and the constant() method is that you can inject a constant into a config function, whereas you cannot inject a value.

Conversely, with constants, we’re unable to register service objects or functions as the value.

Typically, a good rule of thumb is that we should use value() to register a service object or function, while we should use constant() for configuration data.

var app = angular.module('IG', [])

app.constant('apiKey', '123123123')

.config(function (apiKey) {

// The apiKey here will resolve to 123123123

// as we set above

})

app.value('FBid', '231231231')

.config(function (FBid) {

// This will throw an error with

// Unknown provider: FBid

// because the value is not accessible by config

});

**provider() 🡺**

These factories are all created through the $provide service, which is responsible for instantiating these providers at run time. A provider is an object with a $get() method. The $injector calls the $get method to create a new instance of the service. The $provider exposes several different API methods for creating a service, each with a different intended use.

At the root of all the methods for creating a service is the provider method. The provider() method

is responsible for registering services in the $providerCache.

Technically, the factory() function is shorthand for creating a service through the provider() method wherein we assume that the $get() function is the function passed in.

The two method calls are functionally equivalent and will create the same service.

Note: It will provide $get function

var app = angular.module('IG', [])

app.factory('myService', function () {

return {

'username': 'auser'

}

});

// This is equivalent to the above use of factory

app.provider('myService', {

$get: function () {

return {

'username': 'auser'

}

}

});

Why would we ever need to use the .provider() method when we can just use the .factory() method?

The answer lies in whether we need the ability to externally configure a service returned by the .provider() method using the Angular .config() function. Unlike the other methods of service creation, we can inject a special attribute into the config() method.

Let’s say we want to configure our githubService with our URL in advance of the application starting up:

// register the service using `.provider`

var app = angular.module('IG', [])

app.provider('githubService', function($http) {

// default, private state

var githubUrl = 'https://github.com'

setGithubUrl: function(url) {

// change default via .config

if (url) { githubUrl = url }

}

method: JSONP, // override me, if you want

$get: function($http) {

self = this;

return $http({

method: self.method,

url: githubUrl +

'/events'

});

}

});

The idea here is that, by using the .provider() method, we have more flexibility when using our service in more than one app, when sharing our service across applications, or when sharing with the community.

With the example above, the provider() method creates an additional provider with the string ‘Provider’ appended to it that can be injected into the config() function.

var app = angular.module('IG', [])

app.config(function (githubServiceProvider) {

githubServiceProvider.setGithubUrl("git@github.com");

});

If we want to be able to configure the service in the config() function, we must use provider() to define our service.

The provider() method registers a provider for a service. It takes two arguments:

• name (string)

The name argument is a string that we use as the key in the providerCache. This argument makes the name + Provider available as the provider for the service. The name will be used as the name of an instance of the service.

For instance, if we define a service as githubService, then the provider will be available as githubServiceProvider.

• aProvider (object/function/array)

The aProvider argument can take a few different forms:

If the aProvider argument is a function, then the function is called through dependency injection and is responsible for returning an object with the $get method.

If the aProvider argument is an array, then it’s treated just like a function with inline dependency injection annotation. It will expect the last argument to be a function that returns an object with the $get method.

If the aProvider argument is an object, then it is expected to have a $get method.

The provider() function returns an object that is a registered provider instance. The most raw method of creating a service is by using the provider() API directly:

// Example of creating a provider directly on the module object.

var app = angular.module('IG', [])

app.provider('UserService', {

favoriteColor: null,

setFavoriteColor: function (newColor) {

this.favoriteColor = newColor;

},

// the $get function can take injectables

$get: function ($http) {

return {

'name': 'Ari',

getFavoriteColor: function () {

return this.favoriteColor || 'unknown';

}

}

}

});

Creating a service in this way, we must return an object that has the $get() function defined; otherwise, it will result in an error. We can instantiate the service with the injector (although it’s unlikely that we’ll ever instantiate it directly, as Angular apps do it on their own.

var app = angular.module('IG', [])

// Get the injector

var injector = app.injector();

// Invoke our service

injector.invoke(

['UserService', function (UserService) {

// UserService returns

// {

// 'name': 'Ari',

// getFavoriteColor: function() {}

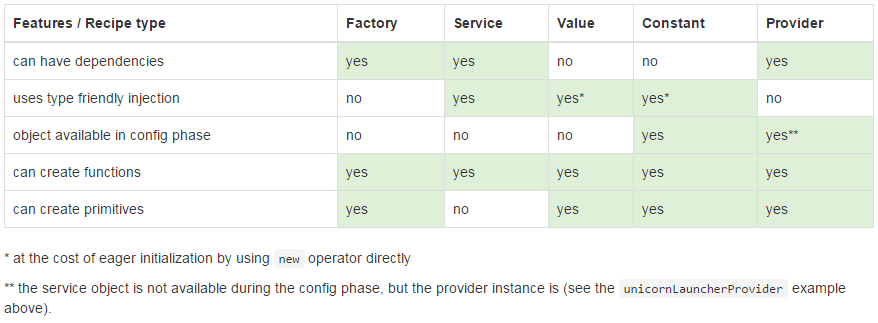
// }

} ]);

Using .provide() is very powerful and gives us the ability to use and share our services across our applications.

It is also important for us to know about the constant() and value() methods when creating services.

Conclusion



 AngularJS, services are reusable singleton objects that are used to organize and share code across your app. They can be injected into controllers, filters, directives. AngularJS provides you three ways : service, factory and provider to create a service.

## Factory

A factory is a simple function which allows you to add some logic before creating the object. It returns the created object.

### Syntax

1. **app.factory('serviceName',function(){ return serviceObj;})**

### Creating service using factory method

1. **<script>**
2. ***//creating module***
3. **var app = angular.module('app', []);**
5. ***//define a factory using factory() function***
6. **app.factory('MyFactory', function () {**
8. **var serviceObj = {};**
9. **serviceObj.function1 = function () {**
10. ***//TO DO:***
11. **};**
13. **serviceObj.function2 = function () {**
14. ***//TO DO:***
15. **};**
17. **return serviceObj;**
18. **});**
19. **</script>**

### When to use

It is just a collection of functions like a class. Hence, it can be instantiated in different controllers when you are using it with constructor function.

## Service

A service is a constructor function which creates the object using new keyword. You can add properties and functions to a service object by using this keyword. Unlike factory, it doesn’t return anything.

### Syntax

1. **app.service('serviceName',function(){})**

### Creating service using service method

1. **<script>**
2. ***//creating module***
3. **var app = angular.module('app', []);**
5. ***//define a service using service() function***
6. **app.service('MyService', function () {**
7. **this.function1 = function () {**
8. ***//TO DO:***
9. **};**
11. **this.function2 = function () {**
12. ***//TO DO:***
13. **};**
14. **});**
15. **</script>**

### When to use

It is a singleton object. Use it when you need to share a single object across the application. **For example**, authenticated user details.

## Provider

A provider is used to create a configurable service object. It returns value by using $get() function.

### Syntax

1. **//creating a service**
2. **app.provider('serviceName',function(){});**
4. **//configuring the service**
5. **app.config(function(serviceNameProvider){});**

### Creating service using provider method

1. **<script>**
2. ***//define a provider using provider() function***
3. **app.provider('configurableService', function () {**
4. **var name = '';**
5. **this.setName = function (newName) {**
6. **name = newName;**
7. **};**
8. **this.$get = function () {**
9. **return name;**
10. **};**
11. **});**
13. ***//configuring provider using config() function***
14. **app.config(function (configurableService) {**
15. **configurableService.setName('www.dotnet-tricks.com');**
16. **});**
17. **</script>**

### When to use

When you need to provide module-wise configuration for your service object before making it available.

