

AngularJS: Factory vs Service vs Provider

When you first get started with Angular, you'll naturally find yourself flooding your controllers and scopes with unnecessary logic. It's important to realize early on that your controller should be very thin; meaning, most of the business logic and persistent data in your application should be taken care of or stored in a service. I see a few questions a day on Stack Overflow regarding someone trying to have persistent data in his or her controller. That's just not the purpose of a controller. For memory purposes, controllers are instantiated only when they are needed and discarded when they are not. Because of this, every time you switch a route or reload a page, Angular cleans up the current controller. Services however provide a means for keeping data around for the lifetime of an application while they also can be used across different controllers in a consistent manner.

Angular provides us with three ways to create and register our own service.

1. Factory
2. Service
3. Provider

TL;DR

1. When you're using a **Factory** you create an object, add properties to it, then return that same object. When you pass this service into your controller, those properties on the object will now be available in that controller through your factory.

```
app.controller('myFactoryCtrl', function ($scope, myFactory) {
  $scope.artist = myFactory.getArtist()
})

app.factory('myFactory', function () {
  var _artist = '';
  var service = {}

  service.getArtist = function () {
    return _artist
  }

  return service;
})
```

2. When you're using **Service**, it's instantiated with the 'new' keyword. Because of that, you'll add properties to 'this' and the service will return 'this'. When you pass the service into your controller, those properties on 'this' will now be available on that controller through your service.

```
app.controller('myServiceCtrl', function ($scope, myService) {
  $scope.artist = myService.getArtist();
});

app.service('myService', function () {
  var _artist = '';
  this.getArtist = function () {
    return _artist;
  }
});
```

3. **Providers** are the only service you can pass into your .config() function. Use a provider when you want to provide module-wide configuration for your service object before making it available.

```
app.controller('myProviderCtrl', function ($scope, myProvider) {
  $scope.artist = myProvider.getArtist();
  $scope.data.thingFromConfig = myProvider.thingOnConfig;
});

app.provider('myProvider', function () {
  this._artist = '';
  this.thingFromConfig = '';

  //Only the properties on the object returned from $get are available in the controller
  this.$get = function () {
    var that = this;
    return {
      getArtist: function () {
        return that._artist;
      },
      thingonConfig: that.thingFromConfig
    }
  }
});

app.config(function (myProviderProvider) {
  myProviderProvider.thingFromConfig = 'This was set in config()';
});
```

```
  })
```

NON TL;DR

In order to extensively show the difference between a Factory, Service, and Provider, we're going to build the same service in three separate ways. The services are going to utilize the iTunes API as well as promises with \$q.

1) Factory

Factories are the most popular way to create and configure a service. There's really not much more than what the TL;DR said. You just create an object, add properties to it, then return that same object. Then when you pass the factory into your controller, those properties on the object will now be available in that controller through your factory. A more extensive example is below.

First we create an object, then return that object like so.

```
app.factory('myFactory', function () {  
  var service = {};  
  return service;  
});
```

Now whatever properties we attach to 'service' will be available to us when we pass 'myFactory' into our controller.

Now let's add some 'private' variables to our callback function. These won't be directly accessible from the controller, but we will eventually set up some getter/setter methods on 'service' to be able to alter these 'private' variables when needed.

```
app.factory('myFactory', function ($http, $q) {  
  var service = {};  
  var baseUrl = 'https://itunes.apple.com/search?term=';  
  var _artist = '';  
  var _finalUrl = '';  
  
  var makeUrl = function () {  
    _artist = _artist.split(' ').join('+');  
    _finalUrl = baseUrl + _artist + '&callback=JSON_CALLBACK';  
    return _finalUrl;  
  }  
  
  return service;  
});
```

Here you'll notice we're not attaching those variables/function to 'service'. We're simply creating them in order to either use or modify them later.

- **baseUrl** is the base URL that the iTunes API requires
- **_artist** is the artist we wish to lookup
- **_finalUrl** is the final and fully built URL to which we'll make the call to iTunes
- **makeUrl** is a function that will create and return our iTunes friendly URL.

Now that our helper/private variables and function are in place, let's add some properties to the 'service' object. Whatever we put on 'service' we'll be able to directly use in whichever controller we pass 'myFactory' into.

We are going to create setArtist and getArtist methods that simply return or set the artist. We are also going to create a method that will call the iTunes API with our created URL. This method is going to return a promise that will fulfill once the data has come back from the iTunes API. If you

haven't had much experience using promises in Angular, I highly recommend doing a deep dive on them.

- **setArtist** accepts an artist and allows you to set the artist
- **getArtist** returns the artist
- **callItunes** first calls `makeUrl()` in order to build the URL we'll use with our `$http` request. Then it sets up a promise object, makes an `$http` request with our final url, then because `$http` returns a promise, we are able to call `.success` or `.error` after our request. We then resolve our promise with the iTunes data, or we reject it with a message saying 'There was an error'.

```
app.factory('myFactory', function ($http, $q) {
  var service = {};
  var baseUrl = 'https://itunes.apple.com/search?term='
  var _artist = '';
  var _finalUrl = '';

  var makeUrl = function () {
    _artist = _artist.split(' ').join('+');
    _finalUrl = baseUrl + _artist + '&callback=JSON_CALLBACK';
    return _finalUrl;
  }

  service.setArtist = function (artist) {
    _artist = artist;
  }

  service.getArtist = function () {
    return _artist;
  }

  service.callItunes = function () {
    makeUrl()
    var deferred = $q.defer();
    $http({
```

```
        method: 'JSONP',
        url: _finalUrl
    }).success(function (data) {
        deferred.resolve(data);
    }).error(function () {
        deferred.reject('There was an error');
    })

    return deferred.promise;
}

return service;
});
```

Now our factory is complete. We are now able to inject 'myFactory' into any controller and we'll then be able to call our methods that we attached to our service object (setArtist, getArtist, and callItunes).

```
app.controller('myFactoryCtrl', function ($scope, myFactory) {
    $scope.data = {};
    $scope.updateArtist = function () {
        myFactory.setArtist($scope.data.artist);
    }

    $scope.submitArtist = function () {
        myFactory.callItunes()
            .then(function (data) {
                $scope.data.artistData = data;
            }, function (data) {
                alert(data);
            })
    }
})
```

In the controller above we're injecting in the 'myFactory' service. We then set properties on our \$scope object that are coming from data from 'myFactory'. The only tricky code above is if you've never dealt with

promises before. Because `callItunes` is returning a promise, we are able to use the `.then()` method and only set `$scope.data.artistData` once our promise is fulfilled with the iTunes data. You'll notice our controller is very 'thin'. All of our logic and persistent data is located in our service, not in our controller.

2) Service

Perhaps the biggest thing to know when dealing with creating a Service is that that it's instantiated with the 'new' keyword. For you JavaScript gurus this should give you a big hint into the nature of the code. For those of you with a limited background in JavaScript or for those who aren't too familiar with what the 'new' keyword actually does, let's review some JavaScript fundamentals that will eventually help us in understanding the nature of a Service.

To really see the changes that occur when you invoke a function with the 'new' keyword, let's create a function and invoke it with the 'new' keyword,

then let's show what the interpreter does when it sees the 'new' keyword. The end results will both be the same.

First let's create our Constructor.

```
var Person = function (name, age) {  
    this.name = name;  
    this.age = age;  
}
```

This is a typical JavaScript constructor function. Now whenever we invoke the Person function using the 'new' keyword, 'this' will be bound to the newly created object.

Now let's add a method onto our Person's prototype so it will be available on every instance of our Person 'class'.

```
Person.prototype.sayName = function () {  
    alert('My name is ' + this.name)  
}
```

Now, because we put the sayName function on the prototype, every instance of Person will be able to call the sayName function in order alert that instance's name.

Now that we have our Person constructor function and our sayName function on its prototype, let's actually create an instance of Person then call the sayName function.

```
var tyler = new Person('Tyler', 23);  
tyler.sayName() // alerts 'My name is Tyler'
```

So all together the code for creating a Person constructor, adding a function to it's prototype, creating a Person instance, and then calling the function on its prototype looks like this.

```
var Person = function (name, age) {  
    this.name = name;  
    this.age = age;  
}  
  
Person.prototype.sayName = function () {  
    alert('My name is ' + this.name)  
}  
  
var tyler = new Person('Tyler', 23);  
tyler.sayName() // alerts 'My name is Tyler'
```

Now let's look at what actually is happening when you use the 'new' keyword in JavaScript. First thing you should notice is that after using 'new' in our example, we're able to call a method (sayName) on 'tyler' just as if it were an object — that's because it is. So first, we know that our Person constructor is returning an object, whether we can see that in the code or not. Second, we know that because our sayName function is located on the prototype and not directly on the Person instance, the object that the Person function is returning must be delegating to its prototype on failed lookups. In more simple terms, when we call tyler.sayName() the interpreter says "OK, I'm going to look on the 'tyler' object we just created, locate the sayName function, then call it. Wait a minute, I don't see it here — all I see is name and age, let me check the prototype. Yup, looks like it's on the prototype, let me call it."

Below is code for how you can think about what the 'new' keyword is actually doing in JavaScript. It's basically a code example of the above

paragraph. I've put the 'interpreter view' or the way the interpreter sees the code inside of notes.

```
var Person = function (name, age) {  
    // The next line creates an object that will  
    // delegate to the person's prototype on failed lookups.  
  
    //var obj = Object.create(Person.prototype);  
  
    // The next line sets 'this' to the newly created object  
    // this = obj;  
  
    this.name = name;  
    this.age = age;  
  
    // return this;  
}
```

Now having this knowledge of what the 'new' keyword really does in JavaScript, creating a Service in Angular should be easier to understand now.

The biggest thing to understand when creating a Service is knowing that Services are instantiated with the 'new' keyword. Combining that knowledge with our examples above, you should now recognize that you'll be attaching your properties and methods directly to 'this' which will then be returned from the Service itself. Let's take a look at this in action.

Unlike what we originally did with the Factory example, we don't need to create an object then return that object because, like mentioned many times before, we used the 'new' keyword so the interpreter will create that object, have it delegate to it's prototype, then return it for us without us having to do the work.

First things first, let's create our 'private' and helper function. This should look very familiar since we did the exact same thing with our factory. I won't explain what each line does here because I did that in the factory example, if you're confused, re-read the factory example.

```
app.service('myService', function ($http, $q) {  
  var baseUrl = 'https://itunes.apple.com/search?term=';  
  var _artist = '';  
  var _finalUrl = '';  
  
  var makeUrl = function () {  
    _artist = _artist.split(' ').join('+');  
    _finalUrl = baseUrl + _artist + '&callback=JSON_CALLBACK';  
    return _finalUrl;  
  }  
});
```

Now, we'll attach all of our methods that will be available in our controller to 'this'.

```
app.service('myService', function ($http, $q) {  
  var baseUrl = 'https://itunes.apple.com/search?term=';  
  var _artist = '';  
  var _finalUrl = '';  
  
  var makeUrl = function () {  
    _artist = _artist.split(' ').join('+');  
    _finalUrl = baseUrl + _artist + '&callback=JSON_CALLBACK';  
    return _finalUrl;  
  }  
  
  this.setArtist = function (artist) {  
    _artist = artist;  
  }  
  
  this.getArtist = function () {
```

```
        return _artist;
    }

    this.callItunes = function () {
        makeUrl()
        var deferred = $q.defer();
        $http({
            method: 'JSONP',
            url: _finalUrl
        }).success(function (data) {
            deferred.resolve(data);
        }).error(function () {
            deferred.reject('There was an error');
        })

        return deferred.promise;
    }

});
```

Now just like in our factory, `setArtist`, `getArtist`, and `callItunes` will be available in whichever controller we pass `myService` into. Here's the `myService` controller (which is almost exactly the same as our factory controller).

```
app.controller('myService', function ($scope, myService) {
    $scope.data = {};
    $scope.updateArtist = function () {
        myService.setArtist($scope.data.artist);
    }

    $scope.submitArtist = function () {
        myService.callItunes()
            .then(function (data) {
                $scope.data.artistData = data;
            }, function (data) {
                alert(data);
            })
    }
});
```

```
    }  
  });
```

Like I mentioned before, once you really understand what 'new' does, Services are almost identical to factories in Angular.

3) Provider

The biggest thing to remember about Providers is that they're the only service that you can pass into the app.config portion of your application. This is of huge importance if you're needing to alter some portion of your service object before it's available everywhere else in your application. Although very similar to Services/Factories, there are a few differences which we'll discuss.

First we set up our Provider in a similar way we did with our Service and Factory. The variables below are our 'private' and helper function.

```
app.provider('myProvider', function ($http, $q) {  
  var baseUrl = 'https://itunes.apple.com/search?term=';  
  var _artist = '';  
  var _finalUrl = '';  
  
  // Going to set this property on the config fn below  
  this.thingFromConfig = '';  
  
  var makeUrl = function () {  
    _artist = _artist.split(' ').join('+');  
    _finalUrl = baseUrl + _artist + '&callback=JSON_CALLBACK';  
    return _finalUrl;  
  }  
});
```

*Again if any portion of the above code is confusing, check out the Factory
tion where I explain what it all does in greater details.*

You can think of Providers as having three sections. The first section is the 'private' variables/functions that will be modified/set later (shown above). The second section is the variables/functions that will be available in your app.config function and are therefore available to alter before they're available anywhere else (also shown above). It's important to note that those variables need to be attached to the 'this' keyword. In our example, only 'thingFromConfig' will be available to alter in the app.config. The third section (shown below) is all the variables/functions that will be available in your controller when you pass in the 'myProvider' service into that specific controller.

When creating a service with Provider, the only properties/methods that will be available in your controller are those properties/methods which are returned from the \$get() function. The code below puts \$get on 'this' (which we know will eventually be returned from that function). Now, that \$get function returns all the methods/properties we want to be available in the controller. Here's a code example.

```
this.$get = function ($http, $q) {  
  return {  
    callItunes: function () {  
      makeUrl()  
      var deferred = $q.defer();  
      $http({  
        method: 'JSONP',  
        url: _finalUrl  
      }).success(function (data) {  
        deferred.resolve(data);  
      }).error(function () {  
        deferred.reject('There was an error');  
      })  
    }  
  }  
  return deferred.promise;  
}
```

```

    setArtist: function (artist) {

        _artist = artist;
    },
    getArtist = function () {
        return _artist;
    },
    thingonConfig: this.thingFomConfig
}
}

```

Now the full Provider code looks like this

```

app.provider('myProvider', function ($http, $q) {
    var baseUrl = 'https://itunes.apple.com/search?term=';
    var _artist = '';
    var _finalUrl = '';

    // Going to set this property on the config fn below
    this.thingFromConfig = '';

    var makeUrl = function () {
        _artist = _artist.split(' ').join('+');
        _finalUrl = baseUrl + _artist + '&callback=JSON_CALLBACK';
        return _finalUrl;
    }

    this.$get = function ($http, $q) {
        return {
            callItunes: function () {
                makeUrl()
                var deferred = $q.defer();
                $http({
                    method: 'JSONP',
                    url: _finalUrl
                }).success(function (data) {
                    deferred.resolve(data);
                }).error(function () {
                    deferred.reject('There was an error');
                })
            }
        }
    }
}

```



```

        return deferred.promise;
    },
    setArtist: function (artist) {
        _artist = artist;
    },
    getArtist = function () {
        return _artist;
    },
    thingonConfig: this.thingFomConfig
    }
}
});

```

Now just like in our factory and Service, setArtist, getArtist, and callItunes will be available in whichever controller we pass myProvider into. Here's the myProvider controller (which is almost exactly the same as our factory/Service controller).

```

app.controller('myProviderCtrl', function ($scope, myProvider) {
    $scope.data = {};
    $scope.updateArtist = function () {
        myProvider.setArtist($scope.data.artist);
    }

    $scope.submitArtist = function () {
        myProvider.callItunes()
            .then(function (data) {
                $scope.data.artistData = data;
            }, function (data) {
                alert(data);
            })
    }

    $scope.data.thingFromConfig = myProvider.thingOnConfig;
});

```

As mentioned before, the whole point of creating a service with Provider is to be able to alter some variables through the `app.config` function before the final object is passed to the rest of the application. Let's see an example of that.

```
app.config(function (myProviderProvider) {  
  // Note that NG appends 'Provider' to the end of the provider name.  
  myProviderProvider.thingFromConfig = 'This sentence was set in app.config. Provi  
  })
```

Now you can see how `'thingFromConfig'` is as empty string in our provider, but when that shows up in the DOM, it will be `'This sentence was set...'`.

