Servicios y Módulos

Anxo Fole
afole@plainconcepts.com

Servicios

Servicio para AngularJs

- Objeto genérico que ofrece funcionalidad encapsulada
  - Inyectada mediante Inyección de Dependencias
  - Sólo es instanciado si algún componente depende de él
  - Singleton. Una única instancia compartida durante todo el ciclo de vida
- AngularJs incluye una serie de servicios por defecto
  - Utilizan el prefijo $ (no utilizar este prefijo para nuestros servicios!)
- Normalmente crearemos servicios personalizados
  - Encapsular lógica reutilizable de nuestra aplicación
  - ¡Modularización!

Registro de Servicios

- Se registran en el servicio ***$provide***. Hay un acceso a este servicio en ***angular.module***, que es el que normalmente usaremos
  - constant
  - value
  - service
  - factory
  - provide
- La mayoría de las veces utilizaremos **Factory**
- En realidad son atajos para definir ***providers*** con menos complejidad

Registro de Servicios

- Usar IIFE’s

```
(function () {
    'use strict';
    var serviceId = 'MyFactory';
    angular.module('app').factory(serviceId, ['$http', MyFactory]);
    function MyFactory($http) {
        // Define the functions and properties to reveal.
        var service = {
            getData: getData
        };
        return service;
        function getData() {
        }
    }
})();
```

constant

- Registran un servicio de constantes en el **$injector**
- string, number, array, object o function
- No se puede modificar
- Inyectable en:
  - controller
  - service
  - ***config*** y run de un módulo

```
var myModule = angular.module('myModule', []);

myModule.constant('myConst', 'Constant value!');
```

value

- Registran un servicio de valores en el **$injector**
- string, number, array, object o function
- Inyectable en:
  - controller
  - service
  - run de un módulo

```
var myModule = angular.module('myModule', []);

myModule.value('myValue', 'A value!');
```

service

- Registra una función constructora que será invocada con un ***new*** para crear la instancia del servicio
- Inyectable en:
  - controller
  - service
  - run de un módulo

```
//service style, probably the simplest one
myApp.service('helloWorldFromService', function () {
    this.sayHello = function () {
        return "Hello, World!"
    };
});
```

factory

- Registra una función factoría que será invocada para crear la instancia del servicio
- Revealing Module Pattern
- Inyectable en:
  - controller
  - service
  - run de un módulo

```
//factory style, more involved but more sophisticated
myApp.factory('helloWorldFromFactory', function () {
    return {
        sayHello: function () {
            return "Hello, World!"
        }
    };
});
```

provider

- Registra una función de proveedor en el **$injector**. Es una función de constructor cuyas instancias son responsables de proporcionar una factoría para la creación del servicio
- Pueden tener otros métodos que permiten añadir configuraciones
- El proveedor se llama como el nombre del servicio más el sufijo ***Provider***
- Inyectable como **servicio** en:
  - controller
  - service
  - run de un módulo
- Inyectable como **Provider** en:
  - ***config***

Provider

```
//provider style, full blown, configurable version
myApp.provider('helloWorld', function () {
    // In the provider function, you cannot inject any
    // service or factory. This can only be done at the
    // "$get" method.
    this.name = 'Default';
    this.$get = function () {
        var name = this.name;
        return {
            sayHello: function () {
                return "Hello, " + name + "!"
            }
        }
    };
    this.setName = function (name) {
        this.name = name;
    };
});
//hey, we can configure a provider!
myApp.config(function (helloWorldProvider) {
    helloWorldProvider.setName('World');
});
```

Características

Features / Recipe type	Factory	Service	Value	Constant	Provider
can have dependencies	yes	yes	no	no	yes
uses type friendly injection	no	yes	yes*	yes*	no
object available in config phase	no	no	no	yes	yes**
can create functions	yes	yes	yes	yes	yes
can create primitives	yes	no	yes	yes	yes

* at the cost of eager initialization by using new operator directly

** the service object is not available during the config phase, but the provider instance is (see the unicornLauncherProvider example above).

Factory, service, provider

- Service, Factory y provider hacen lo mismo. Proporcionan funcionalidades a través de la aplicación.
- Son singletons.
- Factoria:
    - Usar siempre.
    - Puedes ejecutar lógica compleja antes de devolver el objecto.
- Services:
    - Solo cuando queremos acceder a datos comunes o funcionalidades.
    - Servicios simples.
- Provider:
    - Para configurar modulo.

Módulos

Ciclo de vida

Características

- Contenedores de diferentes elementos/partes de la aplicación

- Recomendado
  - Un modulo por cada característica
  - Un modulo por cada component reusable (sobre todo directivas y filtros)
  - Un modulo a nivel de aplicación que depende de los otros dos (y de terceros) y contiene código de inicialización

Fases

- Para dar soporte al uso de los proveedores, el ciclo de vida de Angular se divide en dos fases:
  1. Fase de configuración
  2. Fase de ejecución

Fase de configuración

- En esta fase se pueden configurar las variables de los providers
  - Los que sean configurables
- Las dependencias que se reciban deben tener el nombre del servicio a configurar más el sufijo ***Provider***
- Sólo se pueden inyectar servicios registrados como ***module.constant***

```
myMod.config(function (notificationsServiceProvider) {
    notificationsServiceProvider.setMaxLen(5);
});
```

Fase de ejecución

- Podríamos considerarlo como el método ***Main*** de la aplicación
- Se ejecuta código que ha de ser inicializado antes de que el bootstrapping de la aplicación termine.

```
angular.module('upTimeApp', []).run(function ($rootScope) {
    $rootScope.appStarted = new Date();
});
```

Servicios inyectables por fase

		Fase de configuración	Fase de ejecución
Constant	Valor constante	Sí	Sí
Variable	Valor variable	-	Sí
Service	Nuevo objeto creado por por una función constructora	-	Sí
Factory	Nuevo objeto devuelto por una función de factoría	-	Sí
Provider	Un nuevo objeto creado por la función de factoría $get	Sí (provider)	Sí (servicio)

Múltiples config y run

- Puede haber más de un método **config** y **run** por cada módulo
- Sistema muy flexible pero que hay que mantener controlado
- Buenas prácticas:
  - Es habitual usar config
  - Un método config por provider que se quiera configurar
  - Un único método run si es que se necesita
    - Si podemos evitarlo mejor. Dificulta el testeo

Inyección de dependencias

Inyección de Dependencias

- Angular contiene un subsistema basado en el componente **$injector** que le permite crear instancias de objetos que definen dependencias.

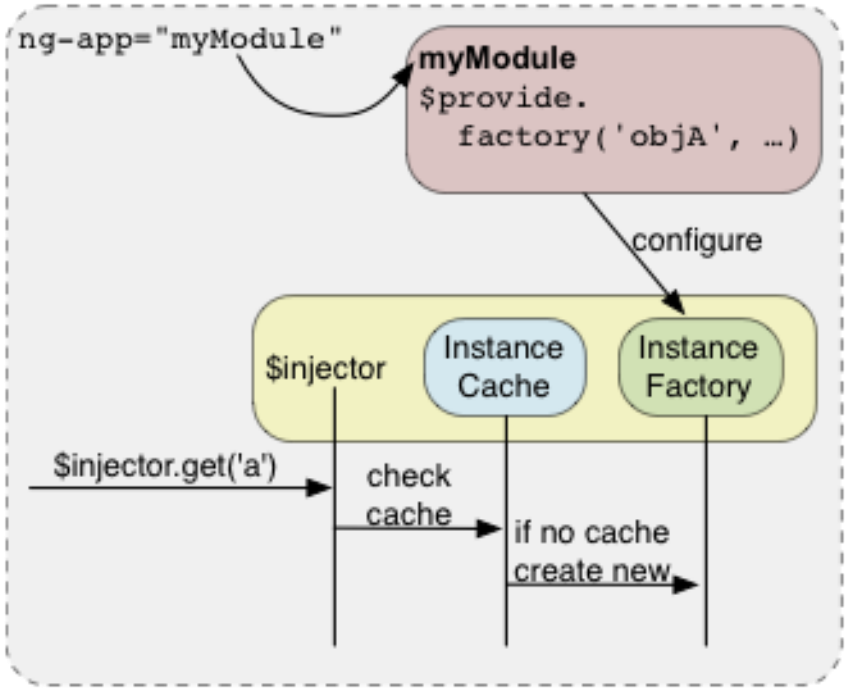
```
var myModule = angular.module('myModule', []);
myModule.factory('greeter', function ($window) {
    return {
        greet: function (text) {
            $window.alert(text);
        }
    };
});
function MyController($scope, greeter) {
    $scope.sayHello = function () {
        greeter.greet('Hello World');
    };
}
```

$injector

- Implementación de un Service Locator

```
var $injector = angular.injector();
```

- Solamente una instancia en toda la aplicación
- Cada módulo registra sus servicios, controladores, etc.
  - Un módulo superior puede depender de un servicio de un módulo dependiente.
  - Incluso de módulos hermanos ***¡evitar!***
- Si hay un conflicto de nombres, el último gana



Inyección de Dependencias

- La raíz de composición será el controlador casi siempre

```
myModule.controller('MyController', function($location) { ... });
```

- Su uso es general a nivel de angular y se usará en:
  - Factorías de servicios
  - Funciones run y config de un módulo
  - Controladores

ID: Factorías de servicios

```
angular.module('myModule', [])
  .factory('serviceId', ['depService', function(depService) {
      ...
  }])
  .directive('directiveName', ['depService', function(depService) {
      ...
  }])
  .filter('filterName', ['depService', function(depService) {
      ...
  }]);
```

ID: Funciones **run** y **config** de un módulo

```
angular.module('myModule', [])
  .config(['depProvider', function(depProvider){
      ...
  }])
  .run(['depService', function(depService) {
      ...
  }]);
```

ID: Controladores

```
someModule.controller('MyController', ['$scope', 'dep1', 'dep2',
function($scope, dep1, dep2) {
    ...
    $scope.aMethod = function() {
        ...
    }
    ...
}]);
```

Inyección de Dependencias

- Cuidado con la minificación
  - Las variables serán renombradas y AngularJS no será capaz de inyectar las dependencias
  - El orden de las dependencias en el array es el mismo que los parámetros en la función de factoría

```
myModule.controller('MyController', function($location) { ... });
```

```
myModule.controller('MyController', ['$location', function($location) { ... }]);
```

Formas de utilizar ID

```
var $injector = angular.injector();

// implicit (only works if code not minified/obfuscated)
$injector.invoke(function (serviceA) { });

// annotated
function explicit(serviceA) { };
explicit.$inject = ['serviceA'];
$injector.invoke(explicit);

// inline
$injector.invoke(['serviceA', function (serviceA) { }]);
```

Anotaciones automáticas con grunt o gulp

- Delegar en los task runners el “trabajo sucio”

```
/* @ngInject */
function SidebarController($state, routerHelper) {}
```

  - gulp-ng-annotate | grunt-ng-annotate

- Usar ngStrictDi en ng-app

```
<body ng-app="todo" ng-strict-di>
```