Provider

Introducció al Provider

Per fi arribem al provider l'últim dels tipus de serveis en AngularJS i amb el qual espero que s'entenguin per fi les diferències entre tots ells.
Un provider és com un factory però permet que es configuri abans de crear el valor del servei. En el tema anterior vam veure l'exemple del servei de hash que es configurava a través d'un value anomenat algoritme. Encara que l'exemple funciona, la veritat és que és una mica barroer ja que aparentment no hi ha relació entre algoritme i hash. La relació entre tots dos queda poc orientada a l'objecte. El provider ve a ajudar creant un objecte previ que permet configurar el factory abans que creí el valor del servei. Aquest nou objecte es diu Provider i en un bloc config podrem accedir-hi per poder configurar el nostre servei.

Un provider està compost de 2 parts:

- El provider que és una classe JavaScript de la qual es crea un únic objecte, el qual permet trucar a un bloc config abans que es truqui al factory-provider i així poder configurar el factory-provider.

- El factory-provider, el qual crea el valor del servei. És pràcticament com la funció factory del tema anterior i l'anomenem factory-provider. Com sempre fem, anem a veure un exemple:

```
1
    function HashProvider() {
2
      var _algoritmo="";
3
4
      this.setAlgoritmo=function(algoritmo) {
5
        _algoritmo=algoritmo;
6
      };
7
8
      this.$get=function() {
9
        var hashFunction;
10
11
        if (_algoritmo==="MD5") {
12
          hashFunction=CryptoJS.MD5;
13
        } else if (_algoritmo==="SHA-1") {
14
          hashFunction=CryptoJS.SHA1;
15
        } else if (_algoritmo==="SHA-2-256") {
16
          hashFunction=CryptoJS.SHA256;
17
```

```
18        } else if (_algoritmo==="SHA-2-512") {
19          hashFunction=CryptoJS.SHA512;
20        } else {
21          throw Error("El tipo de algoritmo no es válido:"+_algoritmo);
22        }
23
24        var hash=function(message) {
25          var objHashResult=hashFunction(message);
26
27          var strHashResult=objHashResult.toString(CryptoJS.enc.Base64);
28
29          return strHashResult;
30        }
31
32        return hash;
33      }
34    }
35
      app.provider("hash",HashProvider);
```

- Línia 1: Definim la classe HashProvider. Posteriorment AngularJS crearà una instància d'aquesta classe.
- Línia 2: Es defineix la propietat privada algoritme la qual contindrà l'algoritme a utilitzar.
- Línia 4: Mètode públic que ens permet establir l'algoritme a utilitzar abans de crear la funció de hash.
- Línia 8: Mètode públic que és realment el factory que crearà el valor del servei. En tota classe Provider és obligatori que existeixi aquest mètode públic anomenat $get. És una obligació que imposa AngularJS perquè ell sàpiga quin és el mètode factory. Podem veure que aquest mètode és exactament igual al del tema anterior de factory excepte que ara fa servir la propietat privada _algoritmo en comptes de trucar al servei algoritmo. Aquest és el mètode que anomenem "factory-provider".
- Línia 35: Definim el provider amb el nom hash i li passem com a argument el nom de la classe HashProvider.Resumiendo, hem creat una classe JavaScript anomenada `HashProvider` amb diferents propietats i mètodes que permetran configurar el factory-provider. El factory-provider es la función `$get` del provider. Por lo tant aquesta funció farà us de les propietats que s'han definit en la classe.

Configurant el provider

Ja tenim definit el provider però ara és necessari poder configurar per establir quin és l'algorisme a utilitzar. Els blocs config són els únics que permeten configurar el provider.
En el bloc config serà necessari injectar el provider, no el factory-provider per poder configurar-lo.

```
1   app.config(["hashProvider",function(hashProvider) {
```

```
2      hashProvider.setAlgoritmo("SHA-1");
3    }]);
```

- Línia 1: Injectem el provider en la funció de config. Cal notar que AngularJS ens obliga en aquest cas a trucar al provider "hashProvider". És a dir que al nom del nostre servei s'afegeix la paraula "Provider" i d'aquesta manera li estem dient que volem l'objecte provider per configurar i no l'objecte final del servei.
- Línia 2: Ara cridem al mètode públic de provider anomenat setAlgoritmo per configurar l'algoritme.

Recorda que en injectar el provider en un bloc config cal incloure en el nom el sufix "Provider".

Per exemple si el servei es diu "login", a l'injectar-lo en un bloc config caldrà posar "loginProvider"

Un cop configurat el provider en el bloc config ja podrem injectar el servei on vulguem, en un controlador, en un altre servei, en un bloc run, etc.

Tornem ara a repassar la diferència entre els blocs config i els blocs run. Un bloc config només existeix per poder configurar un provider i cap dels serveis està encara creat.

Mentre que en un bloc run tots els serveis ja està configurats i es poden utilitzar. Per això el bloc run és més semblant a un mètode Main mentre que el bloc config és més semblant a un tros de codi de preinicialització de l'aplicació.

Injectant dependències

Ja hem dit que un provider està definit per 2 funcions:

- La funció constructora de la classe que permet la configuració, que anomenem provider
- La funció factory que crea el valor del servei, que anomenem factory-provider.

AngularJS ens permet en les dues funcions que puguem injectar dependències encara que en cada un d'ells de tipus diferents. Vegem què podem injectar en cada un d'ells:

- provider: Podem injectar només constant i altres providers però definits en altres mòduls.
- factory-provider: Podem injectar constant, value, service, factory i factory-provider

Vegem ara un petit exemple d'això:

```
1    app.constant("provincia","Madrid");
2    app.factory("municipio",function() {
3      return "Móstoles";
4    });
5
```

```
6    app.provider("direccion",['provincia',function(provincia) {
7      this.$get=['municipio',function(municipio) {
8        return provincia+","+municipio;
9      }]
10   }]);
```

- Línia 6: Definim un provider anomenat "direccion" i injectem la constant província.
- Línia 7: Al factory-provider injectem el factory municipio

L'exemple no té gaire sentit però s'ha posat simplement per veure que es poden injectar dependències en les dues funcions. Una altra cosa interessant d'aquest exemple és que en comptes de crear la funció del Provider a part, s'ha definit com una funció anònima de JavaScript. Són coses que permet el propi llenguatge i no tenen res a veure amb AngularJS.

Millorant la configuració

Encara ens queda un petit canvi per fer, per millorar l'arquitectura del exemple. Al configurar el provider hem posat directament en el bloc config l'algoritme a utilitzar:

```
1   app.config(["hashProvider",function(hashProvider) {
2     hashProvider.setAlgoritmo("SHA-1");
3   }]);
```

- Línia 2: Tenim el text "SHA-1" a pinyó.Com tots sabem no és bona idea posar text fixos en el codi, així que l'ideal és modificar-lo afegint una constant de la següent manera:

```
1   app.constant("algoritmo","SHA-1");
2   app.config(["hashProvider","algoritmo",function(hashProvider,algoritmo) {
3     hashProvider.setAlgoritmo(algoritmo);
4   }]);
```

- Línia 1: Creem la constant amb el valor de l'algoritme, que en aquest cas és "SHA-1".
- Línia 2: Injectem la constant en el bloc config
- Línia 3: Establim l'algoritme que ha d'usar el provider anomenat hash amb el valor de la constant.

Però, ¿no havíem fet tot això per evitar utilitzar la constant? Sí, i seguim sense fer-ho. El que volíem era que el provider no utilitzés directament la constant i segueix sense fer-ho. Aquesta nova constant realment formaria part de la nostra aplicació i no del provider, així que el que estem millorant és la nostra pròpia aplicació i no el provider, que gràcies a ser un provider és perfectament personalitzable i no depèn de cap constant.

Exemple

Podem veure ara el exemple complert veient el ejercici6.html,ejercici6.js, md5.js, sha1.js, sha256.js, sha512.js i enc-base64-min.js.