Capítulo 5
Promesas y peticiones HTTP

1. Introducción

En un sitio web clásico, el rol del servidor es proveer páginas que mostrará el navegador. En una SPA, su rol es diferente, dado que toda la gestión de la navegación, de la visualización de los datos y de las páginas se gestiona del lado cliente, con JavaScript, y se libera al servidor de estas responsabilidades, convirtiéndose en una API web. Su rol es devolver los datos, generalmente en formato JSON.

Las nociones de promesas, que permiten encadenar acciones asíncronas, y los mecanismos de peticiones HTTP que provee AngularJS se describirán en este capítulo.

2. Promesa, el fin de los callbacks

JavaScript es un lenguaje monohilo, por diseño, y se basa en un modelo de eventos no bloqueante. Esto implica que este lenguaje utiliza mucho el mecanismo de acciones asíncronas y se basa en el principio de callback para gestionar este asincronismo.

Una función no asíncrona devuelve un valor o un objeto.

```
var resultado = suma(4, 8)
```

El método `suma` es síncrono y devuelve el resultado de la suma entre sus dos parámetros. La variable `resultado` vale, a continuación, 12.

Una función JavaScript asíncrona no devuelve ningún resultado, sino que recibe como parámetro una función de callback que se invocará cuando el procesamiento haya terminado. Esta función de callback recibe, por su parte, como parámetro los resultados del procesamiento.

```
step1(function(value1) {
    // Procesamiento que se realiza cuando termina la ejecución de la función
});
```

El método `step1` es asíncrono y recibe como parámetro una función de callback. Esta se invoca cuando termina la ejecución de la acción asíncrona y recibe como parámetro el valor resultante de la acción.

Si bien este mecanismo funciona, resulta algo difícil de utilizar cuando es necesario manipular varias funciones asíncronas, por ejemplo para realizar llamadas encadenadas o en paralelo, o bien cuando es deseable implementar una gestión de errores globales a varias llamadas de funciones asíncronas.

```
step1(function (value1) {
    step2(value1, function(value2) {
        step3(value2, function(value3) {
            step4(value3, function(value4) {
                // Procesamiento cuando la ejecución de las cuatro
                // funciones asíncronas ha terminado
            });
        });
    });
});
```

El código anterior encadena la ejecución de varios métodos asíncronos. El método `step2` se invoca en el callback del método `step1`, el método `step3` en el callback del método `step2` y el método `step4` en el callback del método `step3`.

La anidación de callbacks, como hemos visto en el ejemplo anterior, es una sintaxis que se denomina Pyramid of Doom y que replica una sintaxis que se encuentra con frecuencia cuando una aplicación JavaScript utiliza mucho asincronismo. El problema de esta sintaxis proviene de su estructuración, que la vuelve difícilmente legible, resultando menos fácil de mantener.

Para obtener un funcionamiento más flexible, más sencillo de implementar y más legible, se crea la noción de promesa (promise).

2.1 Promise

Una promise, o promesa en español, representa la espera del resultado de una acción asíncrona. Esta noción permite ignorar los callbacks, haciendo que las funciones asíncronas no los reciban como parámetro y en su lugar devuelvan una promesa.

```
var todoId = 5;
var promise = recuperarTodo(todoId);
```

La función `recuperarTodo` no recibe ningún callback como parámetro, sino que devuelve una promesa, que representa la espera del resultado de la función.

Una promesa puede tener tres estados. Cuando la acción asíncrona está en curso, la promesa estará en un estado de espera del resultado. Una vez que la acción asíncrona haya terminado, la promesa estará en un estado de éxito o bien en un estado de fallo. Una vez esté en estado de éxito o de fallo, la promesa ya no podrá cambiar de estado.

Un objeto promise se caracteriza por un método `then` que permite reaccionar en caso de que se modifique su estado.

```
var promise = step1()
    .then(function(value1) {
       // Promesa resuelta con éxito
    }, function(error) {
       // Promesa resuelta con error
    });
```

El método `then` de la promesa devuelve una nueva promesa. Esto permite encadenar varias acciones asíncronas.

```
var promise = step1()
    .then(function(value1) {
       return step2(value1);
    })
    .then(function(value2) {
       return step3(value2);
```

```
})
.then(function(value3) {
   return step4(value3);
})
.then(function (value4) {
    // Procesamiento cuando la ejecución de las cuatro funciones
    // asíncronas ha terminado
});
```

El mecanismo del objeto promise no es nativo de JavaScript y no está vinculado a AngularJS. La mayoría de frameworks JavaScript implementan su propio sistema de promesas, como jQuery o WinJS.

AngularJS implementa también su propio sistema de promesas, integrado en su arquitectura.

Todo el mecanismo de promesas de AngularJS está encapsulado en el servicio `$q`.

2.2 Creación de una promesa

El servicio `$q` provee un método `defer` que permite crear un objeto que representa una acción asíncrona.

```
var deferred = $q.defer();
```

La propiedad `promise` del objeto devuelto por el método `defer` permite acceder a la promesa.

```
var deferred = $q.defer();

var promise = deferred.promise;
```

El objeto devuelto por el método `defer` expone métodos que permiten modificar el estado de la promesa. El método `resolve` resuelve la promesa haciéndola pasar al estado de éxito. Recibe como parámetro un objeto JavaScript que se devolverá como resultado de la promesa.

```
var recuperarTodo = function(id) {
    var deferred = $q.defer();

    setTimeout(function() {
           deferred.resolve({ id : id, name : "Todo " + id});
    }, 2000);

   return deferred.promise;
}
```

La función anterior permite recuperar una tarea en función de su id. Crea una promesa mediante el servicio `$q` y la devuelve como resultado de la función. La llamada a la función `setTimeout` permite simular un procesamiento asíncrono. Pasados dos segundos, la función resolverá la promesa proporcionando la tarea recuperada.

El método `reject` permite, por el contrario, rechazar una promesa haciéndola pasar al estado de error.

```
var recuperarTodo = function(id) {
    var deferred = $q.defer();

    setTimeout(function() {
           if(id > 0) {
                  deferred.resolve({ id : id, name : "Todo " + id});
           } else {
                  deferred.reject("Id de tarea inválido");
           }
    }, 2000);

    return deferred.promise;
}
```

La función `recuperarTodo` se ha modificado para que la promesa se rechace si el id proporcionado como parámetro no es superior a 0. En este caso, se devuelve un mensaje de error como resultado de la promesa.

El último método expuesto por el objeto devuelto por el método `defer` es `notify`. Este método permite enviar información relativa a la ejecución de la función asíncrona manteniendo la promesa en el estado de espera de resultado.

```
var uploadItem = function(item) {
    var deferred = $q.defer();

    var progress = 0;
    var interval = $interval(function() {
```

```
            if (progress >= 100) {
                $interval.cancel(interval);
                deferred.resolve('Carga terminada');
            }

            progress += 10;
            deferred.notify(progress + '% realizado');
    }, 100);

    return deferred.promise;
}
```

La función anterior simula la carga asíncrona de un elemento. El método `notify` de la promesa permite enviar un estado del avance de la carga.

El servicio `$q` provee también un método `all` que permite sincronizar la ejecución de varias promesas en una única. La promesa resultante se resolverá cuando todas las promesas se hayan resuelto. Si al menos una de las promesas se rechaza, la promesa resultante también se rechazará.

```
$q.all([recuperarTodo(5), recuperarTodo(14)]);
```

La promesa devuelta por el método `all` se resolverá cuando terminen las llamadas al método `recuperarTodo`. Si alguno de los casos fracasa, la promesa devuelta por el método `all` también estará en estado de error.

La última función expuesta por el servicio `$q` es la función `when`. Esta función permite encapsular un objeto JavaScript en una promesa. Si el objeto que se desea encapsular es una promesa, el método `when` devuelve, simplemente, esta misma promesa.

```
var promise = $q.when(recuperarTodo(5));
```

La función `when` devuelve la promesa devuelta por la función `recuperarTodo`.

Si el objeto que se ha de encapsular no es una promesa, la función `when` devolverá una nueva promesa, que tendrá como resultado el objeto encapsulado.

```
var promise = $q.when({ id: 0, name: "Nuevo todo"});
```