# Cliente y Servidor

En este ejemplo discutiremos la implementación de una aplicación web completa con cliente y servidor.

Para ello, tendremos que introducir los conceptos de **Servicios de Angular**, **Promises** y **Interceptores HTTP**, además de introducir dos nuevos middlewares para nuestro servidor con Express, **`compression`** y **`express.static`**.

## Servidor

### Compression

Compression es un **middleware** útil para **comprimir los datos que se envían al cliente** en respuesta a una petición. Es un módulo externo y por lo tanto se debe instalar con:

```
npm install --save compression
```

El hecho de comprimir los datos aligera mucho la transmisión y la hace **más rápida**, lo cual nos va a ser útil si enviamos una gran cantidad de información, como pueden ser ficheros de código de librerías externas como Angular o Bootstrap, que como ahora veremos, los serviremos junto con todos los demás ficheros del cliente desde el servidor donde tenemos la API.

Lo podemos usar de esta forma, añadiendo la siguiente línea antes de tratar cualquier ruta de la que queramos que se compriman los datos (típicamente todas, con lo cual arriba del todo de nuestro fichero del servidor):

```
app.use(compression());
```

Donde app es el nombre que le habéis dado a la variable que almacena vuestra aplicación de express (`var app = express();`).

### Express.static

Este es un middleware que nos da express (no hay que instalar nada) y que utilizaremos para servir los ficheros del cliente desde nuestro servidor, de modo que no tendremos que hacer `http-server .` para crear un servidor que sirva los ficheros, sino que lo haremos nosotros desde nuestra api.

Si no utilizaramos este middleware, deberíamos definir una ruta para cada fichero que tuviésemos en el cliente, uno para cada imagen, para cada fichero js, para cada fichero html, etc. y hacer `res.sendFile(<path_fichero>)` con el camino del fichero.

Esto sería inviable si tenemos muchos ficheros y vamos añadiendo más a menudo, por lo que este middleware nos facilita el trabajo definiendo las rutas por nosotros, de modo que nosotros le pasamos la carpeta donde tengamos los ficheros del cliente que queremos servir, y el automáticamente genera rutas para cada fichero.

Lo podemos utilizar de la siguiente forma:

```
app.use(express.static(<path_carpeta_cliente>));
```

Cuidado porque si tenéis alguna ruta en esa carpeta que sea igual a una ruta que hayáis definido en vuestra API váis a tener un conflicto de rutas y va a servir la que esté primero en el código.

Notad también que si queréis que el path de la carpeta sea correcto independientemente del `current working directory` (directorio desde el que hacéis `node server.js`) deberéis especificar el path absoluto:

```
app.use(express.static(__dirname + "/carpeta_cliente")
```

Donde `/carpeta_cliente` es el path relativo a la carpeta del cliente, relativo a la carpeta donde está el fichero en el que programáis.

De este modo, si en la carpeta de mi aplicación tengo el fichero `server.js` y la carpeta `client` donde tengo todos los ficheros del cliente (`index.html`, controladores, partials, librerías externas, etc.), usaré:

```
app.use(express.static(__dirname + "/client"));
```

Y si tengo en `client/img/imagen_bonita.png` una imagen, una vista, o el fichero que sea, yendo al navegador a la url:

```
http://localhost:8080/img/imagen_bonita.png
```

podré ver la imagen.

## Inclusión dinámica de modelos y routers

En este ejemplo, hemos implementado una inclusión dinámica de ficheros.
Si os fijáis, antes para incluir todos los modelos desde models/index.js,
indicábamos un array con los paths relativos de los ficheros de los modelos,
y haciamos un `.forEach()` sobre el array que hacía el trabajo de incluir todos los modelos.

Ahora, para evitarnos tener que declarar el array y para obtenerlo automáticamente,
hacemos lo siguiente:

1. Definimos la convención que todos los ficheros js de los modelos van a tener la extensión `.model.js`.
2. Mediante el módulo `fs`, leemos todos los ficheros de la carpeta `models` (no su contenido, sino su nombre), y los guardamos en un array.
3. Filtramos el array de modo que dejamos sólo aquellos nombres de fichero que tengan el substring `.model`, dado que hemos definido la convención que esos son ficheros js de modelos que se deben incluir.
4. Hacemos un `.forEach()` del mismo modo que lo hacíamos anteriormente, con el array filtrado.

Lo mismo hacemos con los routers, con la diferencia que con ellos, en vez
de simplemente hacerles un `require(..)()` los requerimos y el resultado del
require lo guardamos en el objeto `module.exports`, asignandolo a la propiedad
con nombre igual al nombre del fichero sin extensión, de modo que si tenemos:

`routes/usuarios.router.js`

entonces `routes/index.js` hará

```
exports['usuarios'] = require('./usuarios.router.js')
```

Y si luego hacemos `var routers = require('./routes/')`
en `routers.usuarios` tendremos el router definido en `usuarios.router.js`.

# Cliente

A modo de introducción rápida, los servicios son un componente de angular que utilizaremos principalmente para reutilizar código entre controladores y para comunicarnos con el servidor.

Las promises nos servirán para manejar la asincronía en el lado del cliente, para cuando nos comuniquemos con el servidor desde los servicios.

Los HTTP Interceptors nos servirán para poner el token en el header de las peticiones HTTP que enviemos, de forma automática para todas las peticiones.

## Servicios

Los servicios son otro componente que nos ofrece Angular, además de los controladores, las directivas y otros.

Se declaran del mismo modo que los controladores, y funcionan como clases de C++ o Java, con la particularidad que son singletons, es decir, sólo se instancian la primera vez que se piden.

Los utilizaremos principalmente para:

- **Reutilizar código** en nuestra aplicación de Angular.
- Hacer **comunicación entre controladores**.
- **Guardar información** que se mantendrá mientras el usuario navegue por nuestra página, y que se perderá cuando se recargue la página o se abra de nuevo.
- **Comunicarnos con el servidor**.

**NO** los utilizaremos para:

- Comunicarnos con la **vista**.

Los servicios pueden depender de (**utilizar**) **otros servicios**, lo cual se indica en su declaración.

**Cómo se declaran?** Igual que los controladores, salvo que debemos usar la función `.service()`,
y no `.controller()`:

```
var MyService = function(Dep1, Dep2) {
    // Código del servicio
}

angular.module('myApp').service('MyService', ['Dep1', 'Dep2', MyService]);
```

Los servicios, del mismo modo que los módulos de node, pueden declarar que es lo que quieren que sea
público y que no. En este caso, **todo lo que queramos que sea público se debe asignar como propiedad
del objeto `this`** de la función del servicio. Por ejemplo:

```
var MyService = function(Dep1, Dep2) {
    this.funcionPublica = function() {

    }

    this.valorPublico = 4;

    var funcionPrivada = function() {

    }
}

angular.module('myApp').service('MyService', ['Dep1', 'Dep2', MyService]);
```

Todo aquello que no esté como propiedad del objeto `this` **será privado**.

**Cómo** puedo **utilizar un servicio** desde un controlador/otro servicio?

De hecho, ya lo hemos visto. **Del mismo modo en que requerimos los servicios $scope o $state desde
un controlador**. Lo único que debemos tener en cuenta es de cambiar el nombre ($scope o $state) por
el que hayamos indicado nosotros al crear el servicio (en este caso MyService).

Por ejemplo:

```
var GeneralService = function(MyService) {
    MyService.funcionPublica(); // Funciona!

    MyService.funcionPrivada(); // Error: funcionPrivada es privada (undefined
is not a function)
}

angular.module('myApp').service('GeneralService', ['MyService',
GeneralService]);
```

Como ya habréis podido ver, los servicios que nos da angular o librerias externas normalmente tienen
un nombre que empieza por el símbolo $. Esto es útil para **diferenciarlos fácilmente de nuestros servicios**.

## Promises

Las promises son un mecanismo para **tratar fácilmente con la asincronía**.

Se pueden utilizar tanto en el cliente como en el servidor, pero en este caso las usaremos en el cliente
para **recibir y devolver resultados asíncronamente**, típicamente dentro de un servicio o

controlador de Angular.

Por ejemplo, imaginemos que tenemos un servicio, TareasService, que nos sirve para hacer operaciones
con el recurso Tareas de nuestro servidor. Con este servicio podemos listar, crear, actualizar y borrar
tareas.

```
var TareasService = function() {
    // Código del servicio
}

angular.module('myApp').service('TareasService', [TareasService]);
```

Por el momento, este servicio no tiene dependencias y no está implementado. Vamos a ver como podemos implementar la operación de listar tareas, que va a ser pública. Antes que nada,
necesitaremos poner como dependencias los servicios `$http` y `$q` de angular.

`$http` es un servicio que nos **ofrece por defecto angular** y que utilizaremos para **enviar peticiones**
**al servidor y recibir las correspondientes respuestas**.

`$q`es un servicio que también nos ofrece por defecto angular y que utilizaremos para **crear promises**
**y devolver resultados asíncronamente**. Pero porqué tenemos que devolver resultados asíncronamente? Simplemente porque la petición al servidor va a ser asíncrona, por lo tanto la función para listar
las tareas será también asíncrona, y deberá devolver los resultados asíncronamente mediante una promise.

```
var TareasService = function($http, $q) {
    // Código del servicio
}

angular.module('myApp').service('TareasService', ['$http', '$q',
TareasService]);
```

Ahora ya tenemos los dos servicios como dependencias. Vamos a implementar la petición al servidor:

```
var TareasService = function($http, $q) {
    var URL = 'http://localhost:8080/tareas';

    this.getTasks = function() {
        $http.get(URL).then(
            function(respuestaOkay) {
                // El servidor no ha devuelto ningún código de error,
                // Todo okay
            },
            function(respuestaError) {
                // Algo ha ido mal
            }
        );
    };
};

angular.module('myApp').service('TareasService', ['$http', '$q',
TareasService]);
```

Como véis, hemos usado la función `get` del servicio `$http`. Este servicio nos **ofrece funciones para hacer peticiones de diferentes tipos**: `get`, `post`, `patch`, `delete`... etc.

Esta función recibe como parámetro la **URL a la que queremos hacer la petición**. Esta url debe incluir
el host (localhost en este caso), el puerto (8080 en este caso) sólo necesario si es diferente de 80,
y el path (/tareas). Notad que si tuviésemos el servidor colgado en una página web deberíamos sustituir localhost por la página correspondiente, por ejemplo
`http://www.apptareas.com/tareas`.

Como resultado, devuelve una **promise**. Una promise es un objeto Javascript, y **tiene una función**
**`then()`** que recibe como parámetro **dos funciones**. La primera es la que se va a ejecutar cuando
se tenga el resultado y **todo haya ido bien**, y la segunda es la que se va a ejecutar cuando se tenga el resultado y **algo haya ido mal**.

Ahora ya sabemos hacer la petición al servidor, pero nos falta **devolver el resultado** de alguna manera,
para que el controlador pueda obtenerlo. No podemos utilizar un return dentro de las funciones que
le pasamos al then, ya que estamos implementando una función asíncrona, por lo tanto, debemos
utilizar las promises:

```
var TareasService = function($http, $q) {
    var URL = 'http://localhost:8080/tareas';

    this.getTasks = function() {
        var q = $q.defer();

        $http.get(URL).then(
            function(respuestaOkay) {
                // El servidor no ha devuelto ningún código de error,
                // Todo okay

                q.resolve(respuestaOkay);
            },
            function(respuestaError) {
                // Algo ha ido mal

                q.reject(respuestaError);
            }
        );

    return q.promise;
    };
};

angular.module('myApp').service('TareasService', ['$http', '$q',
TareasService]);
```

Para crear una promise, lo primero que hacemos es usar la función `$q.defer()`, que nos devuelve un objeto con funciones para devolver resultados asíncronamente,
y devolver la promise síncronamente con un return. Fijaos que como la llamada a `$http.get` va a ser asíncrona, immediatamente la función `getTasks()` **devolverá el objeto q.promise como valor de retorno**. Este objeto va a tener la función `then()` que ya hemos visto, por lo tanto para recoger
el resultado cuando llamemos a la función `getTasks()` que estamos implementado lo tendremos que hacer
del mismo modo que con la llamada a `$http.get()`, es decir, tendremos que hacer algo del estilo:
`TareasService.getTasks().then(funcionTodoCorrecto, funcionAlgoHaIdoMal);`.

Sólo nos falta **indicar** cuando se va a llamar la primera función del `.then()` (`funcionTodoCorrecto`), y cuando la segunda (`funcionAlgoHaIdoMal`). Para ello, usamos las funciones **`resolve()`** y **`reject()`** del objeto
q. Cuando llamemos a **`resolve(param1, param2, ..., paramN)` se va a llamar la función `funcionTodoCorrecto`** del `.then` de la promise que devolvemos, con exactamente los mismos parámetros que le pasamos a la función resolve (`param1, param2, ..., paramN`). **Lo mismo para `.reject()` pero en este caso se va a llamar la segunda función (`funcionAlgoHaIdoMal`)**.

De este modo, podemos obtener las tareas desde el controlador usando el servicio `TareasService`:

```
var TareasCtrl = function(TareasService) {
    $scope.tasks = [];

    TareasService.getTasks().then(
    function(respuestaOkay) {
            // En respuestaOkay.data tenemos lo que hayamos enviado en el
.send() o .json()
            // Como respuesta a la petición.
        // Si hemos hecho res.send("Hola mundo"); entonces respuestaOkay.data
será "Hola mundo"
        $scope.tasks = respuestaOkay.data;
    },
    function(respuestaError) {
        // En respuestaError.status tenemos el código de status
        // del error, y en respuestaError.data lo que hayamos enviado en el
.send() o .json()
    }
    );
};

angular.module('myApp').controller('TareasCtrl', ['TareasService',
TareasCtrl]);
```

Del mismo modo podríamos implementar la creación de tareas. Sólo tenemos que tener en cuenta
que **los métodos `post`, `patch` y `put` de `$http` reciben un segundo parámetro, la información
que queremos enviar como *body* de la petición**, y que podremos leer en el servidor accediendo a `req.body`. Por ejemplo:

```
$http.post('http://localhost:8080/tareas', { titulo: 'Fregar' }).then(...)

// En el servidor req.body será el objeto { titulo: 'Fregar' }.
```

## Autenticación y HTTP Interceptors

Hemos visto como podemos enviar peticiones al servidor y recibir sus respuestas.
Pero, y si queremos **autenticarnos, recibir el token del servidor, y enviar peticiones con** este **token**?

Para **autenticarnos**, deberemos **hacer una petición a nuestro servidor**, y este nos devolverá el token, por ejemplo:

```
$http.post('http://localhost:8080/authenticate/', { username: 'albert',
password: '1234' })
    .then(
        function(respuestaOkay) {
            // En respuestaOkay.data.token tenemos el token,
            // ya que en el servidor hacemos res.json({ token: tokenCreado })
            var token = respuestaOkay.data.token;
        },
        function(respuestaError) {
            // Probablemente usuario o contraseña incorrectas
        }
    )
```

Hasta aquí bien, pero como podemos guardar el token que hemos recibido?
Para ello, utilizaremos el servicio `$window` de angular, que deberemos **poner como dependencia**
(típicamente este código iría en un servicio llamado LoginService) y su objeto `sessionStorage`.

```
$http.post('http://localhost:8080/authenticate/', { username: 'albert',
password: '1234' })
    .then(
        function(respuestaOkay) {
            // En respuestaOkay.data.token tenemos el token,
            // ya que en el servidor hacemos res.json({ token: tokenCreado })
            var token = respuestaOkay.data.token;

            // Recordad que habrá que poner $window como dependencia, ya que es
un servicio!!
            $window.sessionStorage.token = token;
        },
        function(respuestaError) {
            // Probablemente usuario o contraseña incorrectas
        }
    )
```

**Los datos que guardemos en el objeto `$window.sessionStorage` se mantendrán hasta que el usuario cierre el navegador**. Por lo tanto, el estado de login se mantendrá mientras el usuario navegue por nuestra página web, y aunque la cierre sin cerrar el navegador.

Ahora nos falta ver como podemos enviar el token que hemos recibido con aquellas peticiones que lo requieran.

Una posible opción sería poner el token en el header cada vez que hiciesemos una petición. Esta opción sería poco viable si hacemos muchas peticiones, ya que deberíamos replicar el código cada vez que hiciesemos la petición.

Otra opción mucho mejor es usar un HTTP `Interceptor`. Los HTTP Interceptors nos sirven para **indicar que se ejecute cierto código cada vez que se envíe cualquier petición HTTP**. De este modo, lo utilizaremos para que cada vez que enviemos una petición HTTP, se añada el token en el header de la petición.

Veamos un ejemplo de como implementarlo:

```
angular.module('TareasApp').factory('authInterceptor', ['$window', '$injector',
function ($window, $injector) {
    return {
        request: function (config) {
            config.headers = config.headers || {};
            if ($window.sessionStorage.token) {
                config.headers.Authorization = 'Bearer ' +
$window.sessionStorage.token;
            }
            return config;
        },
        response: function (response) {
            if (response.status === 401)
                $injector.get('$state').transitionTo('login');

            return response;
        }
    };
}]);

angular.module('TareasApp').config(['$httpProvider', function ($httpProvider) {
    $httpProvider.interceptors.push('authInterceptor');
}]);
```

Como véis, un interceptor no es más que una factory. **Una factory es lo mismo que un servicio, lo único que cambia es que lo que va a ser público del servicio es lo que devuelve la función, en vez del objeto `this` de la función**.

Esta factory tiene dos funciones públicas, `request` y `response`. **`request`** es la función que se ejecutará **justo antes de enviar cualquier petición HTTP al servidor**. En cambio, **`response`** es la función que **se ejecutará justo después de recibir la respuesta del servidor**.

La primera la usamos para añadir el token guardado en `$window.sessionStorage.token` en el header `Authorization` de la petición, y la segunda la usamos para redireccionar a la página de login cuando el servidor responde con un error de autenticación (401).

De este modo, cada vez que hagamos una petición HTTP cualquiera, se va a añadir el token como header de la petición, y si el servidor responde con un error de autenticación, ya sea porque ha expirado el token, o porque no había ninguno guardado, redireccionaremos a la página de login.

Pero hasta ahora sólo hemos creado una factory. No hemos indicado en ningún momento que queremos que esta factory sea un HTTP Interceptor. Para hacerlo, **debemos configurar el servicio HTTP mediante un `.config`**, en el que añadiremos el nombre de la factory que hemos creado al array de interceptors HTTP.
HTTP Provider es simplemente un servicio (en este caso provider) que nos permite configurar el servicio `$http`.

## Bower

Bower es el **equivalente a npm para el cliente**. Es un **gestor de librerías** que utilizaremos para instalar y borrar librerías en el cliente.

Ejemplos de librerías de cliente son **Angular, Bootstrap, Angular-material, Jquery, etc.** Bower nos simplifica el trabajo de instalación de estas librerías, y su funcionamiento es muy similar que con npm y los `package.json`.

Para **instalar Bower** en nuestro ordenador, ejecutaremos:

```
[sudo] npm install -g Bower
```

Una vez instalado, nos situaremos en nuestra carpeta del cliente (típicamente /client), y ejecutaremos `bower init`, para crear el fichero `bower.json`, que será similar al `package.json` del servidor, y contendrá información sobre nuestra aplicación y sus dependencias.

Para **instalar una nueva librería**, ejecutaremos:

```
bower install --save <nombre_libreria>
```

Algunos ejemplos:

```
bower install --save angular

bower install --save angular-material

bower install --save bootstrap

bower install --save jquery
```

La librería **se instalará en la carpeta `bower_components`**, dentro de la carpeta en la que estemos cuando ejecutamos el comando, en este caso `/client`, y se añadirá como dependencia al fichero `bower.json`, de modo que luego ejecutando `bower install`, se instalarán las dependencias indicadas, si no están ya instaladas.

Luego, si queremos **incluir la librería en nuestra aplicación**, procederemos tal y como lo hacíamos anteriormente.

En el fichero html pondremos el path del fichero de la librería que queramos incluir.
Por ejemplo, en el caso de angular, pondríamos la siguiente línea dentro del tag <head>:

```
<script src="/bower_components/angular/angular.js"></script>
```