UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERO TÉCNICO EN INFORMÁTICA DE GESTIÓN

Un Servidor de Autorización OAuth 2.0

Realizado por

Juan Alberto Muñoz Rodríguez

Dirigido por

Antonio Jesús Nebro Urbaneja

Departamento

Lenguajes y Ciencias de la Computación

MÁLAGA, diciembre de 2015

UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERO TÉCNICO EN INFORMÁTICA DE GESTIÓN

Reunido el tribunal examinador en el día de la fecha, constituido por:

Presidente/a Dº/Dª _____

Secretario/a Dº/Dª _____

Vocal Dº/Dª. _____

para juzgar el proyecto Fin de Carrera titulado **Un Servidor de Autorización OAuth 2.0**

realizado por Dº/Dª Juan Alberto Muñoz Rodríguez,

tutorizado por Dº/Dª Antonio Jesús Nebro Urbaneja

y, en su caso, dirigido académicamente por

Dº/Dª _____

ACORDÓ POR _____ OTORGAR LA CALIFICACIÓN DE _____

Y PARA QUE CONSTE, SE EXTIENDE FIRMADA POR LOS COMPARECIENTES DEL TRIBUNAL, LA PRESENTE DILIGENCIA.

Málaga a ____ de _____ de 20__

El/La presidente/a	El/La secretario/a	El/La vocal
Fdo.:	Fdo.:	Fdo.:

Índice

1 Introducción....3
1.1 Objetivos....4
1.2 Contenido del documento....5
2 Conceptos básicos de OAuth 2.0....7
2.1 Introducción....7
2.2 Roles....8
2.3 Flujo del protocolo....9
2.4 Tipos de tokens....10
2.4.1 Token de Acceso....10
2.4.2 Token de Refresco....10
2.5 Clientes....12
2.5.1 Registro de un cliente....12
2.5.2 Tipos de cliente....12
2.5.3 Identificador de cliente....14
2.5.4 Autenticación de cliente....14
2.5.4.1 Contraseña de cliente....14
2.5.4.2 Otros métodos de autenticación....15
2.5.5 Continuación....16
3 Servidor OAuth 2.0....17
3.1 Introducción....17
3.2 Configuración de la arquitectura OWIN....17
3.3 Token Endpoint. Obtención del token de acceso....21
3.3.1 Providers....23
3.3.2 AuthorizationServerProvider....23
3.3.2.1 ValidateClientAuthentication....23
3.3.2.2 GrantResourceOwnerCredentials....26
3.3.2.3 GrantRefreshToken....28
3.3.2.4 TokenEndpoint....29
3.3.3 RefreshTokenProvider....30
3.3.3.1 ReceiveAsync....30
3.3.3.2 CreateAsync....31
4 Servidor de recursos....35
4.1 Introducción....35
4.2 Configuración OWIN....36
4.3 Capa de acceso a datos....38
4.3.1 Diseño de la base de datos....38
4.3.2 Definición de la base de datos. Entity Framework Code First....39
4.3.3 Migrations....41
4.3.4 UserManager y RoleManager....42
4.3.5 Diagrama de clases....43

4.4 Capa de negocio....44
4.5 Capa de presentación....51
5 Aplicación cliente....57
5.1 Introducción....57
5.2 Index.html....59
5.3 Ficheros app*.js....59
5.3.1 Fichero app.js....60
5.3.2 Fichero app.config.route.js....60
5.4 Vista, controlador y servicio....61
5.4.1 Vista....61
5.4.2 Controlador....68
5.4.3 Servicio....75
5.5 Ventanas de la aplicación....79
6 Conclusiones y trabajo futuro....84
7 Apéndice A. Resumen de tecnologías utilizadas....86
8 Apéndice B. Descripción de los paquetes NuGet implementados....87
8.1 Paquetes NuGet .NET....87
8.1.1 Jumuro.Security.Cryptography....87
8.1.2 Jumuro.WebApi.Extensions.ActionResults....89
8.2 Paquetes NuGet AngularJS....89
8.2.1 Jumuro.Angular.CrudREST....89
8.2.2 Jumuro.Angular.Grid....89
8.2.3 Jumuro.Angular.HttpInterceptor....90
8.2.4 Jumuro.Angular.Modal....90
8.2.5 Jumuro.Angular.OAuth....90
8.2.6 Jumuro.Angular.Spinner....95
8.2.7 Jumuro.Angular.Validations....95
8.2.8 Jumuro.Angular.WebApi....95
9 Bibliografía....96

1 Introducción

OAuth es un estándar abierto para autorización. OAuth provee a aplicaciones cliente de un "acceso delegado seguro" a servidores de recursos en nombre del propietario del recurso. Especifica un proceso para que propietarios de recursos autoricen el acceso de terceras partes a sus servidores de recursos sin compartir sus credenciales. Diseñado específicamente para trabajar con HTTP, OAuth esencialmente permite que un servidor de autorización entregue tokens de acceso a aplicaciones cliente, con la aprobación del dueño de los recursos o usuario final (el token de acceso es una cadena especificando un ámbito específico, un tiempo de vida y otros atributos de acceso). La aplicación cliente hace uso del token de acceso para acceder a los recursos protegidos alojados en el servidor de recursos.

OAuth es comúnmente usado como una manera para que los usuarios accedan a sitios web de terceros usando sus cuentas de Microsoft, Google, Facebook, Twitter, etc. sin preocuparse por si sus credenciales de acceso están siendo comprometidas.

OAuth 2.0 es la evolución de OAuth y está enfocado a la simplicidad de desarrollo del cliente mientras provee de flujos de autorización específicos para aplicaciones web, aplicaciones de escritorio, teléfonos móviles y otros dispositivos multimedia.

La especificación del protocolo OAuth 2.0 y sus *Request For Comments* (*RFCs*) asociados son desarrollados por el *Internet Engineering Task Force (IETF) OAuth WG.*

Como se ha dicho antes, el servidor de autorización entrega al cliente tokens de acceso con un tiempo de validez limitado. Si el tiempo de validez del token es muy pequeño, hará que el usuario tenga que estar introduciendo sus credenciales en el servidor de autenticación cada poco tiempo, lo que será bastante incómodo para el usuario e irá en detrimento de la aplicación cliente.

Si el tiempo de validez del token es muy elevado, el uso de la aplicación será más cómodo para el usuario ya que se verá obligado a introducir sus credenciales menos veces, pero un hacker que eventualmente pueda interceptar alguna comunicación, tendrá más tiempo para poder suplantar la identidad del usuario, con los riesgos que esto puede conllevar.

Para evitar estos inconvenientes y algunos más, haremos uso de los llamados tokens de refresco. De este modo tendremos tokens de acceso con un tiempo de vida muy pequeño y

tokens de refresco con un tiempo de vida más elevado que serán usados para obtener nuevos tokens de acceso cada vez que éstos caduquen.

1.1 Objetivos

Para la realización de este proyecto se plantean los siguientes objetivos:

1. El principal objetivo es implementar un servidor de autorización utilizando el estándar OAuth 2.0 que permita a usuarios de aplicaciones de terceros conectarse a dichas aplicaciones sin que éstas tengan la necesidad de conocer ni validar las credenciales de los usuarios.

   La implementación del servidor de autorización OAuth 2.0 se realizará usando el *middleware* OWIN OAuth, desarrollado dentro del proyecto Katana de Microsoft. Por tanto dicho servidor estará desarrollado utilizando el Framework de .NET. Además, el servidor expondrá sus servicios mediante Web API 2 y el acceso a base de datos se realizará utilizando Entity Framework 6. La gestión de los usuarios se hará con ASP.NET Identity 2. Asimismo se realizará utilizando el patrón de diseño Inyección de Dependencias, para lo que se utilizará el contenedor de inversión de control Autofac. Todo ello utilizando el lenguaje de programación C#.

2. Dicho servidor hará también las funciones de servidor de recursos, para permitir la gestión de la configuración de todos los datos referidos a la autorización de usuarios.

3. También se ha pensado en la realización de una aplicación web de una sólo página (*SPA* por sus iniciales en inglés, *Single Page Application*) realizada principalmente con AngularJS, HTML5 y Bootstrap.

   Esta aplicación tendrá varias secciones entre las que se incluyen la gestión de los usuarios, la gestión de las aplicaciones cliente que vayan a hacer uso del servidor de autenticación y la gestión de los tokens de refresco que el servidor haya emitido, lo que permitirá invalidarlos por cualquier motivo, forzando al usuario a introducir nuevamente sus credenciales.

   La aplicación podrá tener usuarios con diferentes niveles de privilegios mediante el uso de roles, cuya implementación será facilitada por la utilización de ASP.NET Identity en el servidor.

La propia aplicación actuará como cliente del servidor de autorización, por lo que sus usuarios harán uso del servidor de autorización para su autenticación.

Como herramienta de control de versiones se utilizará Microsoft Team Foundation a través de Visual Studio Online (ver punto [18] de la bibliografía).

1.2 Contenido del documento

El presente documento se ha estructurado en ocho capítulos. El primero es el actual, en el que se realiza una introducción al proyecto y se describen los objetivos perseguidos con la realización del mismo.

En el segundo capítulo se realiza una introducción al estándar OAuth 2.0 y se definen sus principales conceptos, necesarios para entender mejor la implementación realizada.

En el siguiente capítulo, el número tres, se aborda de lleno la implementación del servidor OAuth 2.0, revisando todos los aspectos técnicos y más prácticos del estándar.

Seguidamente, en el cuarto capítulo, se describe la arquitectura y los detalles de implementación de la parte del servidor destinada a servir los recursos.

El quinto capítulo se ha destinado a la aplicación cliente. Al igual que en los capítulos anteriores se comentan detalles técnicos y de implementación de la misma.

Los capítulos siete y ocho son dos anexos. En el primero de ellos se enumeran todas las tecnologías, protocolos y herramientas utilizadas en la realización de este proyecto. El segundo se utiliza para describir los paquetes *NuGet* desarrollados para la implementación del cliente y el servidor y que pueden ser utilizados en cualquier otro proyecto que requiera alguna de las funcionalidades implementadas.

Finalmente se incluye la bibliografía, compuesta por una serie de enlaces que han sido utilizados como documentación para la realización del proyecto y que pueden ayudar a entender mejor cualquier detalle del mismo.

2 Conceptos básicos de OAuth 2.0

2.1 Introducción

En un modelo de autenticación tradicional cliente/servidor, el cliente accede a un recurso protegido en el servidor autenticándose con las credenciales del propietario del recurso.

Si el dueño del recurso quiere dar acceso a un tercero a dicho recurso, tiene que compartir sus credenciales, lo que conlleva ciertos problemas y limitaciones, principalmente:

- La aplicación de terceros tiene que guardar las credenciales para un uso futuro
- La aplicación de terceros obtiene un acceso total a los recursos, sin que el dueño del recurso tenga opción de restringir la duración del acceso o limitar dicho acceso a una parte de los recursos.
- El dueño del recurso no puede revocar el acceso a una aplicación de terceros sin revocarlo a todas las aplicaciones de terceros, y sólo puede hacerlo cambiando la contraseña.
- Si cualquiera de las aplicaciones de terceros fuese comprometida, implicaría comprometer la contraseña del usuario final y todos los datos protegidos por dicha contraseña.

OAuth salva estos problemas introduciendo una capa de autorización y separando el rol de cliente del de propietario de los recursos. En OAuth, el cliente solicita acceso a los recursos controlados por su propietario y que están almacenados en el servidor de recursos, y se le entregan unas credenciales que son diferentes de las del dueño de los recursos.

En lugar de usar las credenciales del propietario de los recursos para acceder a los recursos protegidos, el cliente obtiene un token de acceso (que como ya adelantamos, es una cadena de texto en la que se define un ámbito concreto, un tiempo de vida y otros atributos de acceso). Los tokens de acceso son entregados a los clientes por un servidor de autorización con la aprobación del propietario de los recursos. El cliente usa el token de acceso para acceder a los recursos protegidos almacenados en el servidor de recursos.

2.2 Roles

OAuth define cuatro roles:

- Propietario de recursos:

  Entidad capaz de conceder acceso a un recurso protegido. Si el propietario del recurso es una persona nos referiremos a él como usuario final.

- Servidor de recursos:

  Es el servidor que almacena los recursos protegidos. Es capaz de aceptar y responder a peticiones de recursos protegidos hechas usando tokens de acceso.

- Cliente:

  Es una aplicación que hace peticiones de recursos protegidos de parte del propietario de los recursos y con su consentimiento. El término *cliente* no implica características de implementación particulares, es decir, no define si la aplicación se ejecuta en un servidor, en un equipo de escritorio o en cualquier otro dispositivo.

- Servidor de autorización

  Es el servidor encargado de entregar los tokens de acceso al cliente tras la correcta autenticación del propietario del recurso y la obtención de la correspondiente autorización.

Para intentar clarificar los roles implicados, en el RFC de OAuth 2.0 se usa el ejemplo de un sitio web de compartición de fotografías (como por ejemplo Flickr) como servidor de recursos y un servicio de impresión (como por ejemplo Snapfish) como cliente.

El usuario final (propietario de un recurso) puede otorgar al servicio de impresión (cliente) acceso de sólo lectura a algunas sus fotos protegidas (en el servidor de recursos) por un tiempo limitado, sin compartir su usuario y contraseña (credenciales) con el servicio de impresión. En su lugar, el cliente se autentica directamente con un servidor de confianza del servicio de compartición de fotografías (servidor de autorización), el cual entrega al servicio de impresión unas credenciales especificas (token de acceso). Incluso el usuario final podría indicar al servidor de autorización que revoque un token de acceso si decide que el cliente ya no debe tener acceso a los recursos.

2.3 Flujo del protocolo

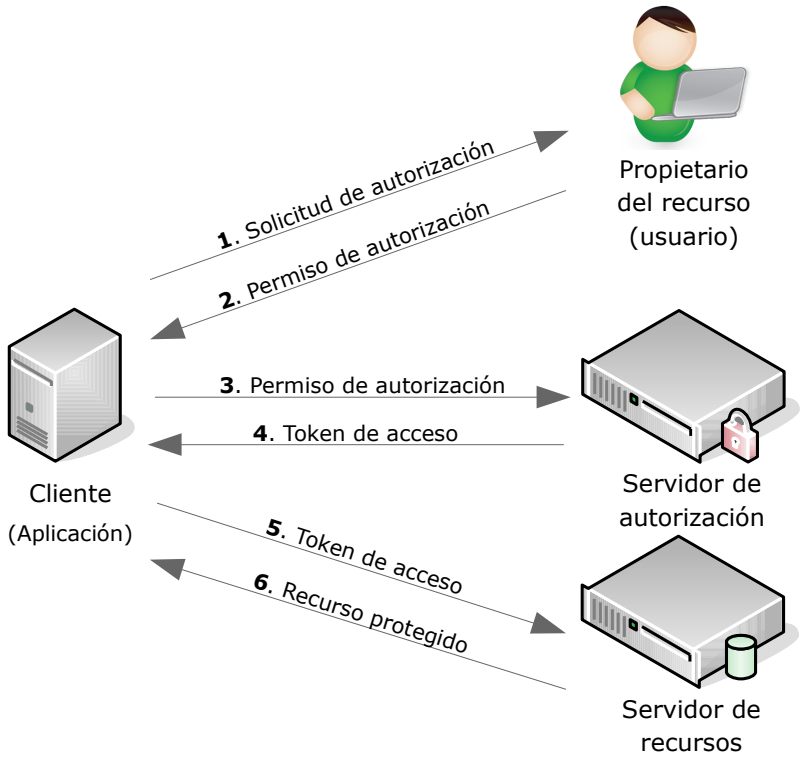


Figura 2.1. Flujo abstracto del protocolo

En la figura 2.1 se muestra el flujo básico del protocolo definido por el estándar. En ella se describe la interacción entre los cuatro roles. A continuación se comentan las diferentes etapas:

1. El cliente solicita autorización del dueño del recurso. La solicitud de autorización puede ser directamente al propietario del recurso (como se muestra) o preferiblemente usando al servidor de autorización como intermediario.
2. El cliente recibe un permiso de autorización, que es una credencial representando la autorización del propietario del recurso.
3. El cliente solicita un token de acceso al servidor de autorización presentando el permiso de autorización recibido en el paso anterior.
4. El servidor de autorización autentica al cliente y valida el permiso de autorización. Si es válido, entrega un token de acceso.

5. El cliente solicita el recurso protegido al servidor de recursos y se autentica presentando el token de acceso.
6. El servidor de recursos valida el token de acceso. Si es válido, sirve el recurso solicitado.

2.4 Tipos de tokens

2.4.1 Token de Acceso

Como ya hemos comentado anteriormente, los tokens de acceso son credenciales usadas para acceder a los recursos protegidos. Un token de acceso es una cadena de texto representando una autorización concedida al cliente. Esta cadena normalmente es opaca al cliente. Estos tokens representan ámbitos y duraciones de acceso específicos, concedidos por el propietario del recurso y regulados por el servidor de recueros y el servidor de autorización.

2.4.2 Token de Refresco

Los tokens de refresco son credenciales usadas para obtener tokens de acceso. Estos tokens son concedidos al cliente por el servidor de autorización y son usados para obtener un nuevo token de acceso cuando el actual pasa a ser inválido o expira, o para obtener tokens de acceso adicionales con igual o menor ámbito (los tokens de acceso pueden tener un período de vida más corto y menos permisos de los autorizados por el propietario del recurso). La concesión de los tokens de refresco es opcional y a discreción del servidor de autorización. Si el servidor de autorización otorga un token de refresco, éste es incluido cuando se otorga el token de acceso (es decir, en el paso 4 de la figura 2.1).

Un token de refresco es una cadena de texto que representa la autorización concedida por el propietario del recurso al cliente. Dicha cadena normalmente es opaca al cliente y denota un identificador utilizado para recuperar la información de autorización. A diferencia de los tokens de acceso, los tokens de refresco solamente son usados con el servidor de autorización y nunca son enviados al servidor de recursos.

En la figura siguiente se muestra el flujo que se produce cuando se refresca un token de acceso caducado:

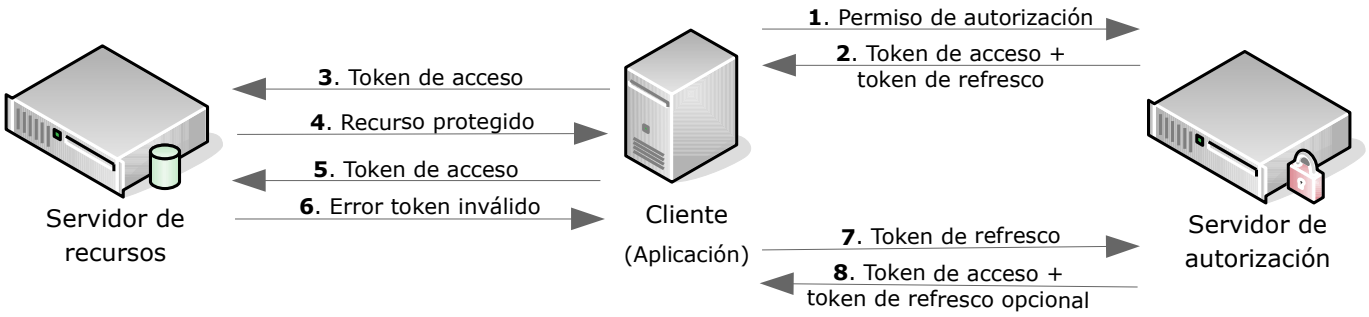


Figura 2.2. Refrescando un token de acceso expirado

El flujo ilustrado incluye los siguientes pasos:

1. El cliente solicita un token de acceso al servidor de autorización presentando el permiso de autorización otorgado.
2. El servidor de autorización autentica al cliente y valida el premiso de autorización. Si es válido, entrega un token de acceso.
3. El cliente solicita el recurso protegido al servidor de recursos y se autentica presentando el token de acceso.
4. El servidor de recursos valida el token de acceso. Si es válido, sirve el recurso solicitado.
5. Los pasos 3 y 4 se repiten hasta que el token de acceso expira. Si el cliente sabe que el token de acceso ha expirado, salta al paso 7, en otro caso hace otra solicitud de recurso protegido.
6. Como el token de acceso es inválido, el servidor de recursos devuelve un error de *token inválido*.
7. El cliente solicita un nuevo token de acceso al servidor de autorización enviando el token de refresco.

8. El servidor de autorización autentica al cliente y valida el token de refresco. Si es válido, entrega al cliente un nuevo token de acceso (y opcionalmente y nuevo token de refresco).

2.5 Clientes

2.5.1 Registro de un cliente

Antes de iniciar el protocolo, el cliente tiene que estar registrado con el servidor de autorización.

Para registrar un cliente, el desarrollador de dicho cliente tiene que realizar las siguientes acciones:

- Especificar el tipo de cliente. Los tipos de cliente los describiremos en el siguiente punto.
- Especificar sus URIs de redirección de cliente. Este punto se explicará más adelante.
- Indicar cualquier otra información requerida por el servidor de autorización, como por ejemplo el nombre de aplicación, el sitio web, una descripción, un logo o la aceptación de los términos legales).

2.5.2 Tipos de cliente

OAuth define dos tipos de clientes, dependiendo de su habilidad para autenticarse de manera segura con el servidor de autorización, o lo que es lo mismo, dependiendo de su habilidad para mantener la confidencialidad de sus credenciales.

Estos tipos son:

- Confidencial

  Es aquel cliente capaz de mantener la confidencialidad de sus credenciales, como por ejemplo siendo implementado en un servidor seguro con acceso restringido a las credenciales de cliente, o capaz de realizar una autenticación segura mediante cualquier otro medio.

- Público

Es aquel cliente incapaz de mantener la confidencialidad de sus credenciales, por ejemplo una aplicación nativa de un dispositivo o una aplicación web, e incapaz de realizar una autenticación seguro mediante cualquier otro medio.

Incluidos en estos tipos de cliente, esta especificación ha sido diseñada pensando en los siguientes perfiles de cliente:

- Aplicación web

  Una aplicación web es un cliente confidencial ejecutándose en un servidor web. El propietario del recurso accede al cliente vía una interfaz de usuario HTML renderizada en un agente de usuario (*user-agent* en inglés) que se ejecuta en el dispositivo usado por el propietario del recurso. Las credenciales del cliente así como cualquier token de acceso entregado al cliente son almacenadas en el servidor web y no son expuestas ni accesibles por el propietario de los recursos.

- Aplicación basada en agente de usuario

  Una aplicación basada en un agente de usuario es un cliente público. El código de cliente es descargado de un servidor web y ejecutado en un agente de usuario (por ejemplo un navegador web) en el dispositivo del propietario de los recursos. Los datos del protocolo y las credenciales son fácilmente accesibles (y a veces visible) por el propietario del recurso. Como estas aplicaciones residen en el agente de usuario, pueden hacer uso de sus capacidades cuando soliciten autorización.

- Aplicación nativa

  Una aplicación nativa es un cliente público instalado y ejecutado en el dispositivo usado por el propietario del recurso. Los datos del protocolo y las credenciales son accesibles por el propietario del recurso. Se asume que las credenciales de autenticación de cliente incluidas en la aplicación podrían ser extraídas. Por otro lado, credenciales entregadas dinámicamente como tokens de acceso o tokens de refresco puede tener un nivel de protección aceptable. Como mínimo estas credenciales pueden ser protegidas de eventuales servidores hostiles con los que la aplicación pudiera interactuar. En algunas plataformas, esta credenciales pueden ser protegidas de otras aplicaciones que residan en el mismo dispositivo.

2.5.3 Identificador de cliente

El servidor de autorización entrega al cliente registrado un identificador de cliente, que es una cadena de texto única representando la información de registro proveída por dicho cliente. El identificador de cliente no es secreto, es expuesto al propietario del recurso y no debe ser usado en solitario para la autenticación del cliente. El identificador de cliente es único en el servidor de autorización.

2.5.4 Autenticación de cliente

Si el tipo de cliente es confidencial, el cliente y el servidor de autorización establecen un método de autenticación adecuado a los requerimientos de seguridad del servidor de autorización.

Típicamente, a los clientes confidenciales se les entrega un conjunto de credenciales de cliente usadas para la autenticación con el servidor de autorización, por ejemplo una contraseña o un par de claves pública/privada.

El cliente no debe utilizar más de un método de autenticación en cada petición.

2.5.4.1 Contraseña de cliente

Los clientes en posesión de una contraseña de cliente deben usar el esquema de autenticación básico HTTP para autenticarse con el servidor de autorización, como se describe en la RFC 2617, ver punto [17] de la bibliografía.

El identificador de cliente es codificado usando el algoritmo de codificación `application/x-www-form-urlencoded` y el valor resultante es utilizado como nombre de usuario.

La contraseña de usuario es codificada usando el mismo algoritmo y el valor resultante es usado como contraseña.

El servidor de autorización debe soportar el esquema de autenticación básico HTTP para autenticar a los clientes a los que fue entregada una contraseña.

Por ejemplo:

```
Authorization: Basic zCzZ4321Sa3F0Mazo3RampaBaenIxS3vEUmJuMuRobUS3
```

Alternativamente, el servidor de autorización puede admitir las credenciales de cliente incluidas en el cuerpo de la petición HTTP usando los siguientes parámetros:

- `client_id`

  OBLIGATORIO. Es el identificador de cliente entregado por el servidor de autorización durante el registro de dicho cliente.

- `client_secret`

  OBLIGATORIO. El secreto de cliente. El cliente puede omitir este parámetro si el secreto de cliente es una cadena vacía.

Incluir las credenciales de cliente en el cuerpo de la petición usando los dos parámetros no es recomendable y debería quedar limitado a clientes incapaces de utilizar directamente el esquema de autenticación básico HTTP (u otro esquema de autenticación HTTP basado en contraseña). Los parámetros sólo podrán ser transmitidos en el cuerpo de la petición y nunca en la URI.

Por ejemplo, una petición de refresco de token de acceso incluyendo los parámetros en el cuerpo quedaría como sigue:

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=refresh_token&refresh_token=tGzv3JOkF0XG5Qx2TlKWIA
&client_id=s6BhdRkqt3&client_secret=7Fjfp0ZBr1KtDRbnfVdmIw
```

El servidor de autorización debe requerir el uso de TLS (*Transport Layer Security*) cuando recibe peticiones que incluya una contraseña.

Ya que este método de autenticación de cliente incluye una contraseña, el servidor de autorización debe proteger cada *endpoint* contra ataques por la fuerza bruta.

2.5.4.2 Otros métodos de autenticación

El servidor de autorización puede dar soporte a cualquier esquema de autenticación HTTP que encaje en sus requerimientos de seguridad. Si se usan otros métodos de autenticación, el servidor de autorización debe definir una correspondencia entre el identificador de cliente y el esquema de autenticación.

2.5.5 Continuación

En este capítulo del documento hemos presentado los conceptos más básicos y teóricos del protocolo. Los demás puntos relevantes los veremos de una manera más práctica y aplicados en el desarrollo del servidor implementado, en el próximo capítulo.

3 Servidor OAuth 2.0

3.1 Introducción

Para la implementación del servidor de autorización me he basado en un proyecto publicado en el blog bitoftech.net, en su entrada referenciada en el punto [4] de la bibliografía.

Este servidor se ha desarrollado siguiendo la especificación OWIN (*Open Web Interface for .NET*, ver puntos [5] y [6] de la bibliografía).

El proyecto Katana es un conjunto de componentes OWIN de código abierto desarrollado por Microsoft. Dentro de Katana se ha desarrollado un *middleware* que implementa el protocolo OAuth 2.0. Este módulo está disponible como paquete NuGet con el nombre `Microsoft.Owin.Security.OAuth` e incluye la clase `OauthAuthorizationServerProvider`, donde se ha llevado a cabo dicha implementación.

Como enseguida veremos, nuestro servidor hace uso de esta clase modificando o extendiendo su funcionalidad en varios puntos. Pero primero hablaremos un poco de OWIN y de su arquitectura dentro de nuestro servidor.

3.2 Configuración de la arquitectura OWIN

Toda aplicación OWIN tiene una clase de arranque donde se especifican los componentes o *middlewares* que forman la tubería de uso de la aplicación.

Hay varias formas de definir esta clase. En nuestro caso y a modo de ejemplo se han utilizado dos de ellas, aunque una de las formas queda anulada por la otra:

1. Convención de nombres: Katana busca una clase llamada `Startup` dentro del espacio de nombres del ensamblado o en el espacio de nombres global.

2. Atributo `OwinStartup`: Es la forma más utilizada por los desarrolladores. El siguiente atributo define la clase de arranque a la clase `Startup` del espacio de nombres `OAuthServer`.

   El atributo `OwinStartup` prevalece sobre la convención de nombres.

La clase de arranque de nuestro servidor es la que se muestra a continuación:

```
using Microsoft.Owin;
using Owin;

[assembly: OwinStartup(typeof(OAuthServer.Startup))]

namespace OAuthServer
{
    public partial class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            //Configure Autofac
            var dependencyResolver = ConfigureAutofac(app);

            //Configure CORS
            ConfigureCors(app);

            //Configure OAuth
            ConfigureOAuth(app, dependencyResolver);

            //Configure Web API
            ConfigureWebApi(app, dependencyResolver);
        }
    }
}
```

Código 3.1. Método `Startup.Configuration()`

En nuestra clase de arranque podemos ver cuatro partes bien diferenciadas:

1. Configuración del contenedor de inversión de control, en nuestro caso Autofac.
2. Configuración del CORS (*Cross-origin resource sharing*). CORS es un mecanismo que permite acceso a recursos restringidos de una página web desde un dominio externo al dominio donde reside dicha página web.
3. Configuración de OAuth.
4. Configuración de Web API.

Cada uno de los métodos de configuración los hemos definido en un fichero de clase parcial dentro de la carpeta de proyecto `App_Start` como se muestra en la siguiente figura:

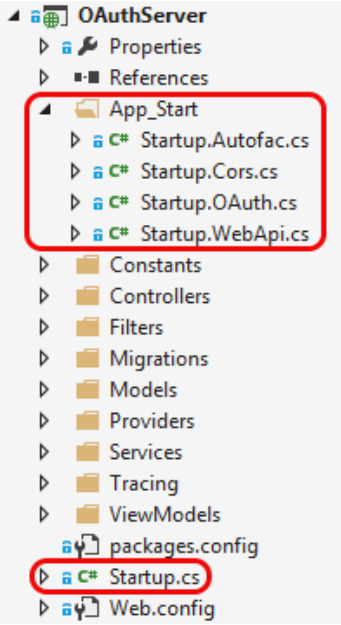


Figura 3.1. Estructura de clase de arranque

Aunque todos los métodos de configuración son interesantes, en el caso que nos ocupa sólo analizaremos la configuración OAuth, cuyo código se muestra a continuación:

```
using System;
using System.Configuration;
using System.Web.Http.Dependencies;
using Microsoft.Owin;
using Microsoft.Owin.Security;
using Microsoft.Owin.Security.Cookies;
using Microsoft.Owin.Security.Infrastructure;
using Microsoft.Owin.Security.OAuth;
using OAuthServer.Constants;
using Owin;

namespace OAuthServer
{
    public partial class Startup
    {
        public void ConfigureOAuth(IAppBuilder app,
                                   IDependencyResolver dependencyResolver)
        {
            // Enable Application Sign In Cookie
            app.UseCookieAuthentication(new CookieAuthenticationOptions
            {
                AuthenticationType = "Application",
                AuthenticationMode = AuthenticationMode.Passive,
                LoginPath = new PathString(Paths.LoginPath),
```

```
                LogoutPath = new PathString(Paths.LogoutPath),
            });

            var OAuthServerOptions = new OAuthAuthorizationServerOptions()
            {
#if DEBUG
                AllowInsecureHttp = true,
#endif
                TokenEndpointPath = new PathString(Paths.TokenPath),
                AccessTokenExpireTimeSpan =
                    TimeSpan.FromMinutes(Convert.ToDouble(
                        ConfigurationManager.AppSettings[
                            "DefaultAccessTokenExpireTime"
                        ]
                    )),
                Provider = (IOAuthAuthorizationServerProvider)
                    dependencyResolver.GetService(
                        typeof(IOAuthAuthorizationServerProvider)),
                RefreshTokenProvider = (IauthenticationTokenProvider)
                    dependencyResolver.GetService(
                        typeof(IAuthenticationTokenProvider))
            };

            var OAuthBearerOptions = new OAuthBearerAuthenticationOptions();

            // Token generation
            app.UseOAuthAuthorizationServer(OAuthServerOptions);
            // Token validation
            app.UseOAuthBearerAuthentication(OAuthBearerOptions);
        }
    }
}
```

Código 3.2. Método `Startup.ConfigureOAuth()`

En el método, primero se añade un *middleware* de autenticación basado en el uso de *cookies* a la tubería de uso de la aplicación mediante la llamada a `app.UseCookieAuthentication()`.

Luego se crea un objeto del tipo `OAuthAuthorizationServerOptions` en el que se definen las opciones del servidor de autorización OAuth:

- `AllowInsecureHttp`: Verdadero para permitir que las peticiones de token y de autorización lleguen a través de una URI http en vez de https. Sólo se pone a verdadero cuando al aplicación está en modo DEBUG.
- `TokenEndpointPath`: Establece el *path* al que las aplicaciones cliente se dirigen para obtener un token de acceso.

- `AccessTokenExpireTimeSpan`: Establece el período de tiempo durante el que el token de acceso es válido desde que es entregado. Por defecto es de 20 minutos, aunque podremos establecer uno diferente para cada cliente configurado en nuestro servidor.
- `Provider`: Establece un objeto, proporcionado por la aplicación, para procesar todos los eventos lanzados por el *middleware* de Servidor de Autorización, y definidos en el estándar OAuth 2.0. Aquí asignaremos una instancia de nuestra clase que hereda de `OauthAuthorizationServerProvider` comentada en la introducción.
- `RefreshTokenProvider`: Establece un objeto que producirá tokens de refresco que podrán ser usados para obtener nuevos tokens de acceso cuando sea necesario.

Con el objeto de opciones creado se realiza una llamada al método `app.UseOAuthAuthorizationServer()` el cual añade las capacidades de Servidor de Autorización OAuth 2.0 a la aplicación OWIN. Este *middleware* lleva a cabo el procesamiento de las peticiones para el *Authorize Endpoint* y *Token Endpoint* definidos por la especificación OAuth 2.0. Este *middleware* gestiona todo lo relacionado con la generación de los tokens.

Por último, en nuestro método de configuración se hace una llamada al método `app.UseOAuthBearerAuthentication()`, lo que hace que se añadan las capacidades de procesamiento de *Bearer tokens* a la tubería de uso de la aplicación OWIN. Este *middleware* entiende los tokens recibidos en la cabecera de cada petición. Este *middleware* gestiona todo lo relacionado con la validación de los tokens.

A continuación explicaremos la implementación de los dos proveedores configurados en este punto.

3.3 *Token Endpoint.* Obtención del token de acceso

Las aplicaciones cliente que quieran obtener un token de acceso lo harán dirigiéndose al *Token Endpoint* cuyo path ha sido configurado en la propiedad `TokenEndpointPath`. En nuestro caso el path configurado es `/token`.

La solicitud de token de acceso se podrá llevar a cabo de dos maneras diferentes:

1. Mediante las credenciales del propietario de los recursos.
2. Mediante token de refresco.

La figura 3.1 muestra el flujo que seguirán ambas solicitudes dentro de las clases *provider*:

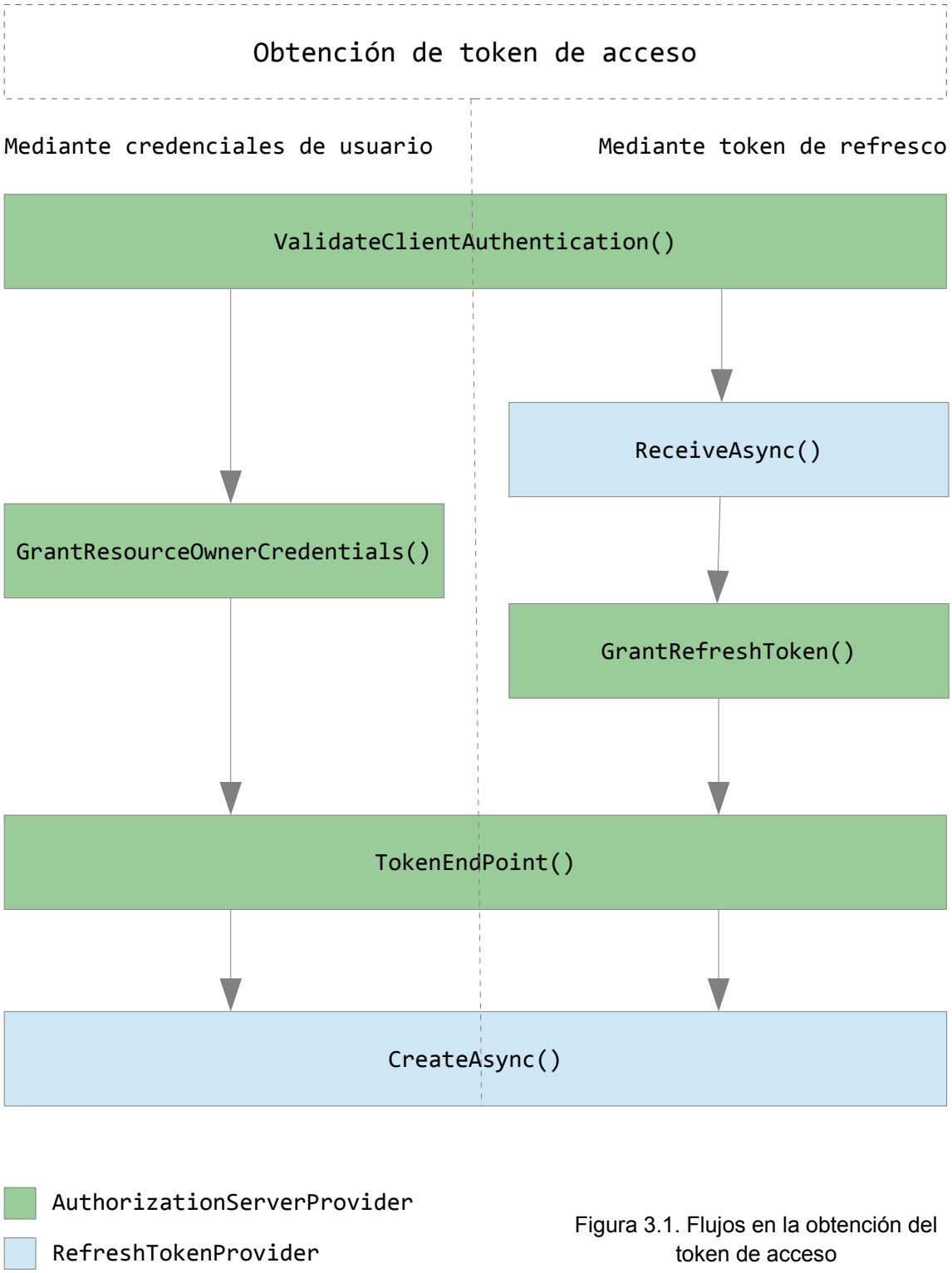


Figura 3.1. Flujos en la obtención del token de acceso

3.3.1 *Providers*

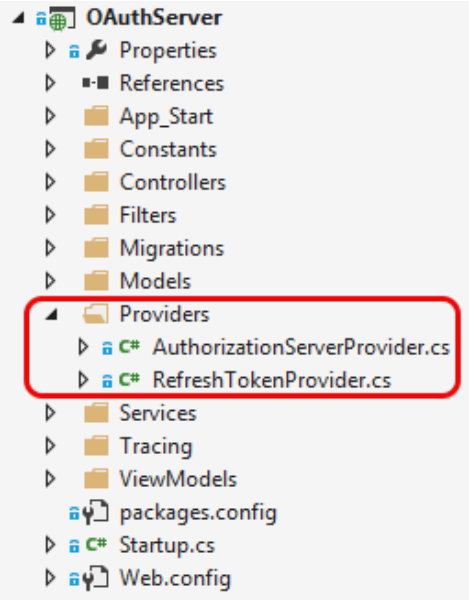


Figura 3.2. *Providers*

Como se observa en la figura 3.2 hemos creado dos clases de *providers* en nuestro proyecto. La primera, `AuthorizationServerProvider`, hereda de `OauthAuthorizationServerProvider`, que es la que implementa el estándar OAuth 2.0 propiamente dicho. Y la segunda `RefreshTokenProvider`, es la encargada de la generación y la recepción de los tokens de refresco. Ambas intervienen en el proceso de concesión del token de acceso.

3.3.2 AuthorizationServerProvider

3.3.2.1 ValidateClientAuthentication

Es llamado para validar que el origen de la petición es un `client_id` registrado, y que sus credenciales están presentes. Si la aplicación web acepta credenciales *Basic authentication*, habrá que llamar al método `context.TryGetBasicCredentials()` para obtener estos valores de la cabecera de la petición. Si la aplicación web acepta `client_id` y `client_secret` con parámetros de POST recibidos en el formulario, habrá que llamar al método `context.TryGetFormCredentials()` para obtener dichos valores del cuerpo de la petición. Si `context.Validated()` no es llamado, la petición no continuará procesándose.

El código completo del método se muestra a continuación:

```
public override async Task
ValidateClientAuthentication(OAuthValidateClientAuthenticationContext context)
{
    base.ValidateClientAuthentication()
    string clientId = string.Empty;
    string clientSecret = string.Empty;

    if (!context.TryGetBasicCredentials(out clientId, out clientSecret))
    {
        context.TryGetFormCredentials(out clientId, out clientSecret);
    }

    if (context.ClientId == null)
    {
        context.SetError("invalid_clientId", "ClientId should be sent.");
        return;
    }

    ClientViewModel client;
    client = await _clientService.GetClientAsync(context.ClientId);

    if (client == null)
    {
        context.SetError("invalid_clientId",
                         string.Format(
                             "Client '{0}' is not registered in the system.",
                             context.ClientId));
        return;
    }

    if (client.ApplicationType.Id == (int)ApplicationTypes.NativeConfidential)
    {
        if (string.IsNullOrWhiteSpace(clientSecret))
        {
            context.SetError("invalid_clientSecret",
                             "Client secret should be sent.");
            return;
        }
        else if (client.Secret !=
                  _hashProvider.GetSHA256Hash(clientSecret).ToBase64String())
        {
            context.SetError("invalid_clientSecret",
                             "Client secret is invalid.");
            return;
        }
    }

    if (!client.IsActive)
    {
        context.SetError("invalid_clientId", "Client is inactive.");
        return;
    }

    context.OwinContext.Set<string>("clientAllowedOrigin",
```

```
                                    client.AllowedOrigin);
        context.OwinContext.Set<string>("clientAccessTokenExpireTime",
                                    client.AccessTokenExpireTime.ToString());
        context.OwinContext.Set<string>("clientRefreshTokenLifeTime",
                                    client.RefreshTokenLifeTime.ToString());

        context.Validated();
    }
```

Código 3.3. Método `ValidateClientAuthentication()`

Como podemos ver en el código, en nuestro caso aceptaremos ambos modos de recepción de las credenciales. Primero se intentan obtener de la cabecera de la petición, si las credenciales no son obtenidas se comprueba si están presentes en el cuerpo del formulario.

Si las credenciales están presentes en la petición, se accede a base de datos para validar el `client_id` recibido.

Tal y como vimos en el capítulo 2, los clientes pueden ser "nativos" o "públicos". Si el cliente en cuestión es nativo hay que comprobar que haya enviado el `client_secret` y validar que sea correcto.

En base de datos se almacena el hash del `client_secret` usando el algoritmo SHA256. Por tanto, para validar el `client_secret` recibido, se compunta su hash y se compara con el valor almacenado.

El método `_hashProvider.GetSHA256Hash()` está implementado en la clase `HashProvider` dentro del paquete NuGet `Jumuro.Security.Cryptography`. En el apéndice B se describen los paquetes NuGet implementados para este proyecto.

Si se superan todas las validaciones anteriores y además el cliente está activo, se añaden una serie de parámetros de configuración al contexto y se llama al método `context.Validated()` para que la petición pueda seguir procesándose.

Los parámetros añadidos al contexto son:

- `clientAllowedOrigin`: Orígenes permitidos para configuración CORS.
- `clientAccessTokenExpireTime`: Tiempo en minutos durante el cual el token de acceso entregado al cliente será válido antes de expirar.

- `clientRefreshTokenLifeTime`: Tiempo en minutos durante el cual el token de refresco entregado al cliente será válido antes de expirar.

3.3.2.2 GrantResourceOwnerCredentials

Es llamado cuando al *Token Endpoint* llega una solicitud con `grant_type = password`. Esto ocurre cuando el usuario final introduce sus credenciales (usuario y contraseña), directamente en la interfaz de usuario de la aplicación cliente, y la aplicación cliente usa estas credenciales para obtener un token de acceso y opcionalmente un token de refresco. Si la aplicación web admite concesión mediante las credenciales del propietario de los recursos, deberá validar el usuario y contraseña recibidos en el contexto (`context.Username` y `context.Password`).

Para poder entregar un token de acceso hay que llamar al método `context.Validated()` pasándole como parámetro un nuevo ticket que incluya los permisos sobre el recurso que serán asociados al token de acceso.

El código fuente escrito para implementar este método es el siguiente:

```
public override async Task
GrantResourceOwnerCredentials(OAuthGrantResourceOwnerCredentialsContext context)
{
    base.GrantResourceOwnerCredentials
    var allowedOrigin = context.OwinContext.Get<string>("clientAllowedOrigin");

    if (allowedOrigin == null) allowedOrigin = "*";

    if (!context.OwinContext.Response.Headers.ContainsKey(
                                                "Access-Control-Allow-Origin"))
    {
        context.OwinContext.Response.Headers.Add("Access-Control-Allow-Origin",
                                                new[] { allowedOrigin });
    }

    UserViewModel user;

    user = await _userService.GetUserAsync(context.UserName,
                                            context.Password,
                                            context.ClientId);

    if (user == null)
    {
        context.SetError("invalid_grant",
                         "The user name, password or clientId is incorrect.");
        return;
    }
```

```
        else if (!user.IsActive)
        {
            context.SetError("invalid_grant", "The user is inactive.");
            return;
        }

        var identity = new ClaimsIdentity(context.Options.AuthenticationType);
        identity.AddClaim(new Claim(ClaimTypes.Name, context.UserName));
        identity.AddClaim(new Claim(ClaimTypes.Sid, user.UserId));

        if (user.Role != null && !string.IsNullOrEmpty(user.Role.Name))
        {
            identity.AddClaim(new Claim(ClaimTypes.Role, user.Role.Name));
        }

        var props = new AuthenticationProperties(new Dictionary<string, string>
        {
            {
                "client_id",
                (context.ClientId == null) ? string.Empty : context.ClientId
            }
        });

        var ticket = new AuthenticationTicket(identity, props);
        context.Validated(ticket);
    }
```

Código 3.4. Método `GrantResourceOwnerCredentials()`

En primer lugar recuperamos del contexto el parámetro de configuración `clientAllowedOrigin` establecido en el método `ValidateClientAuthentication`. Este valor configurado es utilizado para establecer la cabecera de CORS `"Access-Control-Allow-Origin"` en el mensaje de respuesta.

A continuación se validan las credenciales de usuario del propietario de los recursos, y además se valida que dicho usuario esté dado de alta para el cliente que ha enviado las credenciales. También se chequea que el usuario esté activo en nuestra base de datos.

Una vez realizadas las validaciones oportunas, se crea un ticket de autenticación que incluye los datos de usuario contenidos en la base de datos de ASP.NET Identity (en concreto UserId, UserName y Role) y el `client_id`.

Por último se llama al método `context.Validated()` pasándole como parámetro el ticket de autenticación creado. Esto permitirá que la petición pueda seguir procesándose.

3.3.2.3 GrantRefreshToken

Este método es llamado cuando se recibe una petición al *Token Endpoint* con `grant_type = refresh_token`. Esto ocurre si la aplicación entrega un token de refresco junto con el token de acceso el cliente intenta usar el token de refresco para obtener un nuevo token de acceso y posiblemente un nuevo token de refresco. Para que se entrege un token de refresco debe asignarse un `Options.RefreshTokenProvider` para crear el valor que se devuelve. Los permisos y propiedades asociadas al token de refresco están presentes en el `context.Ticket`. La aplicación debe llamar al método `context.Validated()` para indicar al *middleware* de servidor de autorización que entregue un token de acceso con esos permisos y propiedades. Al método `context.Validated()` se le puede pasar un *AuthenticationTicket* o *ClaimsIdentiy* diferentes para controlar qué información fluye del token de refresco al token de acceso. El comportamiento por defecto al usar la clase `OauthAuthorizationServerProvider` es que la información pase del token de refresco al token de acceso sin modificar.

El código fuente de este método es el siguiente:

```
public override Task GrantRefreshToken(OAuthGrantRefreshTokenContext context)
{
    var originalClient = context.Ticket.Properties.Dictionary["client_id"];
    var currentClient = context.ClientId;

    if (originalClient != currentClient)
    {
        context.SetError("invalid_clientId",
                         "Refresh token is issued to a different clientId.");
        return Task.FromResult<object>(null);
    }

    // Change auth ticket for refresh token requests
    var newIdentity = new ClaimsIdentity(context.Ticket.Identity);

    var newTicket = new AuthenticationTicket(newIdentity,
                                             context.Ticket.Properties);
    context.Validated(newTicket);

    return Task.FromResult<object>(null);
}
```

Código 3.5. Método `GrantRefreshToken()`

En primer lugar se comprueba si el cliente al que se le entrogó el token de refresco es el mismo que lo ha enviado, para ello se compara ClientId incluido en el token de refresco con el `context.ClientId`.

Después se crea un ticket de autenticación con las *ClaimsIdentities* y las propiedades incluidas en el ticket recibido en el token de refresco y se llama al método `context.Validated()` pasándole como parámetro el nuevo ticket creado. Esto permitirá que la petición pueda seguir procesándose.

3.3.2.4 TokenEndpoint

Este método es llamado en la última etapa de una petición satisfactoria al *Token Endpoint*. Una aplicación puede implementar esta llamada para hacer alguna modificación final a los permisos usados para los tokens de acceso o de refresco. Este método también puede ser usado para añadir parámetros adicionales al cuerpo de la respuesta *json* del *Token Endpoint.*

El código de nuestro método es:

```
public override Task TokenEndpoint(OAuthTokenEndpointContext context)
{
    // If the client has a specific AccessTokenExpireTime,
    // override the default one with it
    var clientAccessTokenExpireTimeSpan =
        context.OwinContext.Get<string>("clientAccessTokenExpireTime");

    if (!string.IsNullOrEmpty(clientAccessTokenExpireTimeSpan) &&
        Convert.ToDouble(clientAccessTokenExpireTimeSpan) > 0)
    {
        context.Properties.ExpiresUtc = new DateTimeOffset?(
            context.Properties.IssuedUtc.Value.Add(
                System.TimeSpan.FromMinutes(
                    Convert.ToDouble(clientAccessTokenExpireTimeSpan))));
    }

    // Include in the context only "client_id"
    context.AdditionalResponseParameters.Add("client_id",
        context.Properties.Dictionary["client_id"]);

    return Task.FromResult<object>(null);
}
```

Código 3.6. Método `TokenEndpoint()`

Lo único que hacemos en este método es comprobar si en el contexto se ha definido un `AccessTokenExpireTime` particular para el cliente, estableciéndolo en las propiedades del contexto si es así.

Y por último sólo se añade a los parámetros adicionales del cuerpo de la respuesta *json* el `client_id` recibido en las propiedades del contexto.

3.3.3 RefreshTokenProvider

Nuestra clase `RefreshTokenProvider` implementa la interfaz `IAuthenticationTokenProvider` definida dentro del paquete NuGet `Microsoft.Owin.Security`. Como se puede ver en el bloque de código 3.7, esta clase incluye dos métodos que se comentan a continuación.

3.3.3.1 ReceiveAsync

Este método es llamado cuando se recibe una petición al *Token Endpoint* con `grant_type = refresh_token`. Sirve para extraer el ticket de autenticación del token de refresco y cargarlo en el contexto.

El código completo del método implementado es:

```
public async Task ReceiveAsync(AuthenticationTokenReceiveContext context)
{
    string hashedTokenId =
        _hashProvider.GetSHA256Hash(context.Token).ToBase64String();

    var refreshToken =
        await _refreshTokenService.GetRefreshTokenAsync(hashedTokenId);

    // Check that the refresh token exists
    if (refreshToken != null)
    {
        // Get protectedTicket from refreshToken class
        context.DeserializeTicket(refreshToken.ProtectedTicket);

        // Delete the token from database because it is going to be created
        // again in CreateAsync() method that is called after ReceiveAsync()
        var result =
            await _refreshTokenService.DeleteRefreshTokenAsync(hashedTokenId);
    }
}
```

Código 3.7. Método `ReceiveAsync()`

Todos los tokens de refresco generados por el servidor son almacenados en base de datos. En nuestra implementación de este método, en primer lugar se computa el Hash del token de refresco recibido con el algoritmo SHA256, de este modo obtenemos la clave con la que se guardan los datos del token de refresco con base de datos.

A continuación se recupera de base datos el token de refresco cuya clave coincide con el Hash calculado. Si existe, se carga el ticket de autenticación en el contexto y se borra el registro de base de datos, ya que cuando se emita el nuevo token de acceso también se emitirá un nuevo token de refresco.

3.3.3.2 CreateAsync

Es llamado para generar un nuevo token de refresco y para que sea añadido como parámetro adicional al cuerpo de la respuesta *json*.

El código fuente es el siguiente:

```
public async Task CreateAsync(AuthenticationTokenCreateContext context)
{
    var clientId = context.Ticket.Properties.Dictionary["client_id"];

    if (string.IsNullOrEmpty(clientId))
    {
        return;
    }

    var refreshTokenId = Guid.NewGuid().ToString("n");

    // Get the refresh token life time for this client and
    // set to the refresh token ticket
    var refreshTokenLifeTime =
        context.OwinContext.Get<string>("clientRefreshTokenLifeTime");

    if (string.IsNullOrEmpty(refreshTokenLifeTime))
    {
        // Get the default refresh token life time
        refreshTokenLifeTime =
            ConfigurationManager.AppSettings["DefaultRefreshTokenLifeTime"];
    }

    var token = new RefreshToken()
    {
        RefreshTokenId =
            _hashProvider.GetSHA256Hash(refreshTokenId).ToBase64String(),
        ClientId = clientId,
```

```
            UserName = context.Ticket.Identity.Name,
            UserId = context.Ticket.Identity.Claims
                .Where(c => c.Type == ClaimTypes.Sid).Select(c => c.Value)
                .SingleOrDefault(),
            IssuedUtc = DateTime.UtcNow,
            ExpiresUtc =
                DateTime.UtcNow.AddMinutes(Convert.ToDouble(refreshTokenLifeTime))
        };

        // Set the issued and expires times to the ticket
        context.Ticket.Properties.IssuedUtc = token.IssuedUtc;
        context.Ticket.Properties.ExpiresUtc = token.ExpiresUtc;

        // Serialize the refresh token ticket to store it in database
        token.ProtectedTicket = context.SerializeTicket();

        var result = await _refreshTokenService.InsertRefreshTokenAsync(token);

        if (result != null)
        {
            context.SetToken(refreshTokenId);
        }
    }
```

Código 3.8. Método `CreateAsync()`

En primer lugar recuperamos el `client_id` guardado en el ticket, recordemos que el ticket ha sido copiado al contexto en el método `ReceiveAsync()` anteriormente explicado.

Luego se genera el `refreshTokenId` que será la clave del registro de token de refresco guardado en la tabla `RefreshToken` de la base de datos.

A continuación se recupera del contexto el parámetro de configuración `clientRefreshTokenLifeTime`. Si el valor no está presente se establece el `defaultRefreshTokenLifeTime`.

Una vez hecho esto se crea un objeto del tipo de modelo `RefreshToken` para ser guardado en base de datos que incluye el hash del `refreshTokenId` generado, el clientId, el userName, el userId, la fecha de generación y la fecha de expiración del token. Estas últimas fechas también son establecidas en el ticket del contexto. Se serializa el ticket del contexto y el valor obtenido se guarda en el objeto de modelo en el campo `ProtectedTicket`. Ahora que tenemos todos los datos del token de refresco, se guarda en base de datos. Sólo queda llamar al método `context.SetToken()` pasándole como parámetro el `refreshTokenId` generado.

Como podemos observar, el token de refresco que se guarda en el contexto y que posteriormente se enviará al cliente es realmente el `refreshTokenId`, el ticket que contiene toda la información se guarda en base de datos para su posterior recuperación. Como hemos visto es el método `ReceiveAsync()` el encargado de recuperar el ticket de base de datos y copiarlo en el contexto.

4 Servidor de recursos

4.1 Introducción

La parte del servidor dedicada a servidor de recursos ha sido implementada usando Web API 2, lo que nos ha facilitado la tarea de construir nuestros servicios HTTP. Además se ha implementado siguiendo el estándar RESTful.

Podemos dividir la implementación de nuestro servidor en tres capas bien diferenciadas:

- Capa de acceso a datos:

  Como su nombre indica, aquí se implementa el acceso a la base de datos. Dicho acceso se ha implementado utilizando Entity Framework en su vertiente *Code-First*.

- Capa de negocio:

  En esta capa se implementa toda la lógica necesaria para manipular los datos.

- Capa de presentación:

  Esta capa la identificamos con nuestros controladores de Web API, es aquí donde se exponen los servicios proporcionados por nuestro API.

  Aquí se reciben las peticiones HTTP de los clientes, se realizan las validaciones necesarias, se hace uso de la capa de negocio para obtener o modificar los datos y se envía la respuesta a través de HTTP a los clientes.

En la figura 4.1 se puede ver la correspondencia de esta división por capas dentro de la estructura del código.

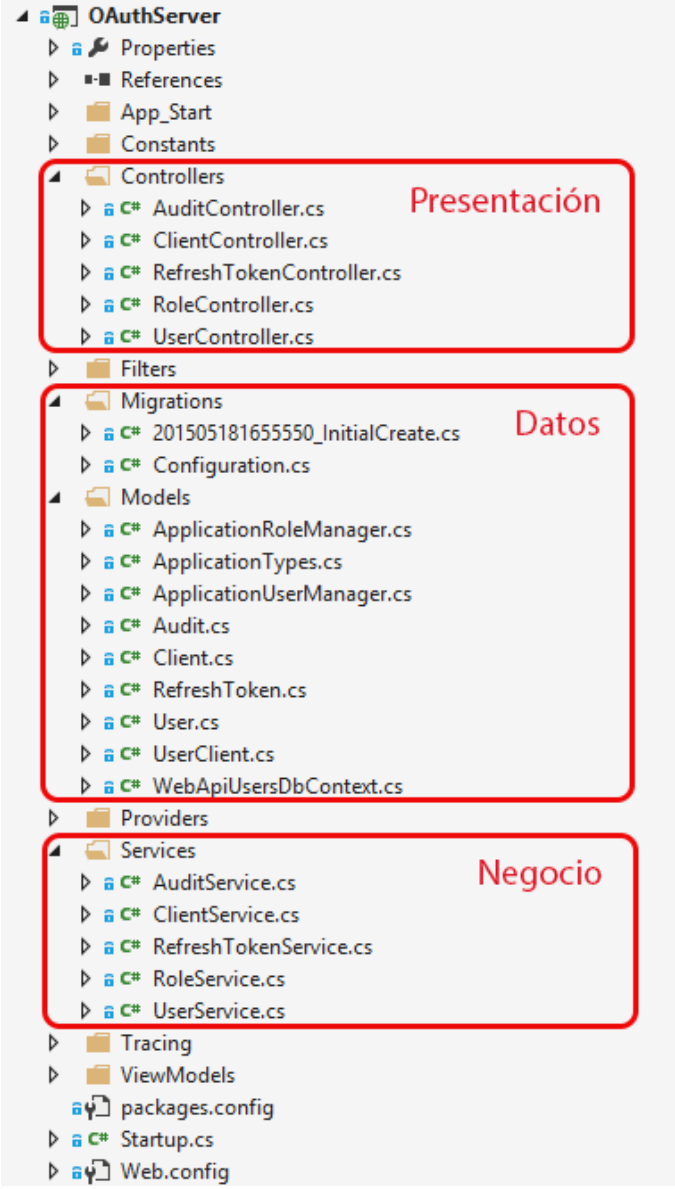


Figura 4.1. División por capas

Pero primero veamos cómo encaja todo esto en nuestra arquitectura OWIN.

4.2 Configuración OWIN

Para poder utilizar Web API en nuestra aplicación se ha tenido que añadir el *middleware* OWIN correspondiente a la tubería de uso de la aplicación. Esta tarea se ha llevado a cabo en el método `ConfigureWebApi()` de la clase `Startup`. El código de este método se muestra a continuación:

```
using System.Linq;
using System.Net.Http.Formatting;
using System.Web.Http;
using System.Web.Http.Dependencies;
using System.Web.Http.ExceptionHandling;
using Newtonsoft.Json.Serialization;
using OAuthServer.Filters.Attributes;
using OAuthServer.Tracing;
using Owin;

namespace OAuthServer
{
    public partial class Startup
    {
        public void ConfigureWebApi(IAppBuilder app,
                                    IDependencyResolver dependencyResolver)
        {
            var config = new HttpConfiguration
            {
                // Assign received dependency resolver for Web API to use.
                DependencyResolver = dependencyResolver
            };
            // Web API routes
            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );
            // Add a global authorization filter, to all ApiControllers
            config.Filters.Add(new ApplicationAuthorizeAttribute());

            // Add a global Exception handler
            config.Services.Replace(typeof(IExceptionHandler),
                                    new GeneralException());

            // Configure Json Media Type Formatter
            var jsonFormatter =
                config.Formatters.OfType<JsonMediaTypeFormatter>().First();
            jsonFormatter.SerializerSettings.ContractResolver =
                new CamelCasePropertyNamesContractResolver();

            app.UseWebApi(config);

            // Make sure the Autofac lifetime scope is passed to Web API.
            app.UseAutofacWebApi(config);
        }
    }
}
```

Código 4.1. Método `Startup.ConfigureWebApi()`

Primero se crea un objeto de tipo `HttpConfiguration` al que se le asigna el objeto de resolución de dependencias, de tipo `AutofacWebApiDependencyResolver` y creado durante la configuración de Autofac (en el método `Startup.ConfigureAutofac()`). Luego se configura la ruta básica para Web API, se añade un filtro de autorización, se añade un servicio para procesar las excepciones no capturadas y se configura el formateador de JSON, encargado de serializar a formato JSON los objetos devueltos por el Web API.

Por último se llama al método `app.UseWebApi()` que es el que añade el *middleware* de Web API a la tubería de uso de la aplicación.

También se llama al método `app.UseAutofacWebApi()` para extender el ámbito de duración de Autofac de OWIN al ámbito de dependencias de Web API.

4.3 Capa de acceso a datos

4.3.1 Diseño de la base de datos

En la figura 4.2 se muestra el diagrama de la base de datos de nuestro servidor.

En el diagrama se pueden apreciar tres bloques de tablas:

- ASP.NET Identiy Database

  Este bloque incluye las tablas de ASP.NET Identity utilizadas para gestionar los usuarios, roles y permisos.

- Application Database

  Son las tablas utilizadas por el servidor para gestionar los clientes, los usuarios de cada cliente, los tokens de refresco y auditoría.

- Entity Framework Migration History Table

  En esta tabla se registra cada cambio realizado en la base de datos desde código. Recordemos que hemos usado Entity Framework en su versión *Code First*, lo que quiere decir que cada creación o modificación de tabla se realiza mediante clases de código. Cada cambio se traslada a base de datos con lo que se llama una *migración*. Es en esta tabla donde se registran todas las migraciones ejecutadas.

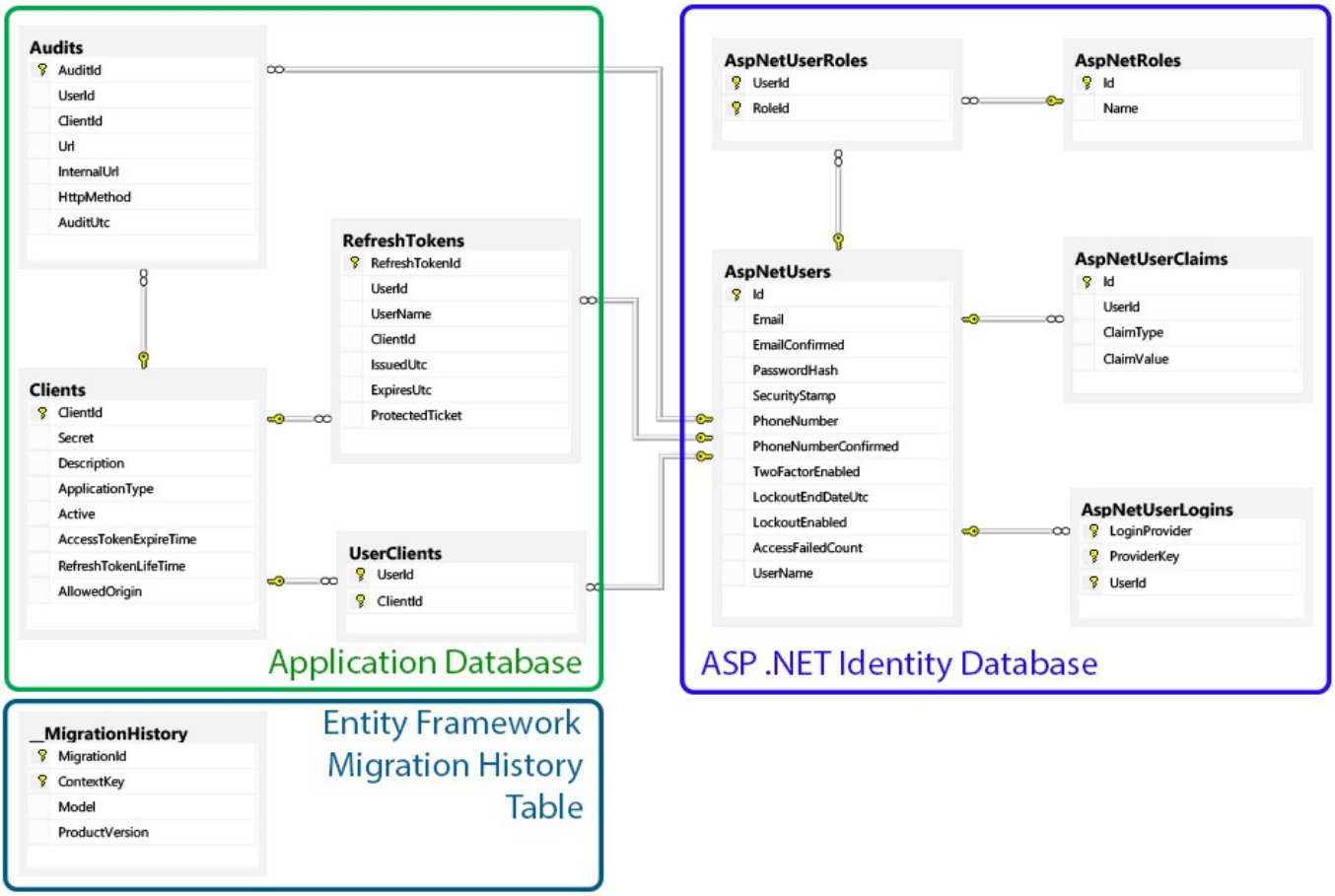


Figura 4.2. Diagrama de base de datos

4.3.2 Definición de la base de datos. Entity Framework Code First

Al utilizar Entity Framework, lo primero que tenemos que definir es una clase que herede de `System.Data.Entity.DbContext`, capaz de interactuar con nuestra base de datos.

Una instancia de `DbContext` representa a una combinación de patrones *Unit Of Work* y *Repository* que pueden ser usados para consultar una base de datos y agrupar cambios que serán escritos de una vez en la misma.

En nuestro caso, como además hacemos uso de ASP.NET Identity, nuestra clase *DbContext* heredará de `Microsoft.AspNet.Identity.EntityFramework.IdentityDbContext<TUser>`, que a su vez es clase descendiente de `DbContext` y donde `TUser` es una clase que hereda de `Microsoft.AspNet.Identity.EntityFramework.IdentityUser`, en nuestro caso será la clase de modelo `User`.

A continuación se muestra el código que define nuestro *DbContext*:

```
using System.Data.Entity;
using AuthServer.Migrations;
using Microsoft.AspNet.Identity.EntityFramework;

namespace OAuthServer.Models
{
    public class OAuthServerDbContext : IdentityDbContext<User>
    {
        public OAuthServerDbContext()
            : base("DefaultConnection")
        {
           Database.SetInitializer<OAuthServerDbContext>(
               new MigrateDatabaseToLatestVersion
                   <OAuthServerDbContext, Configuration>());
        }

        public DbSet<Client> Clients { get; set; }
        public DbSet<RefreshToken> RefreshTokens { get; set; }
        public DbSet<UserClient> UserClients { get; set; }
        public DbSet<Audit> Audits { get; set; }
    }
}
```

Código 4.2. Clase `OAuthServerDbContext`

Lo primero que observamos es que en el constructor se llama al constructor de la clase base pasando como parámetro la cadena `"DefaultConnection"`, que no es más que el nombre de la cadena de conexión a nuestra base de datos definida en el fichero `Web.config`:

```
<configuration>
  ...
  <connectionStrings>
    <add name="DefaultConnection"
     connectionString="Data Source=DELL-INSPIRON\JUMURO;Initial
Catalog=OAuthServer;Persist Security Info=True;Integrated Security=SSPI"
     providerName="System.Data.SqlClient" />
  </connectionStrings>
  ...
</configuration>
```

Código 4.3. Cadena de conexión a base de datos definida en el fichero `Web.config`

Seguidamente, todavía en el constructor, se llama al método `Database.SetInitializer<OAuthServerDbContext>()`, que lo que hace es establecer el *inicializador* de la base de datos para el contexto indicado. El inicializador de la base de datos es llamado cuando el `DbContext` es utilizado por primera vez para acceder a la base de datos. De esta manera además se llama al método `Configuration.Seed()`, que establece los valores iniciales en base de datos.

Finalmente, se define una serie de propiedades de tipo `DbSet<T>` que será traducida en la creación de una tabla en la base de datos con la estructura definida por el modelo de tipo `T` por cada una de las propiedades. En nuestro caso estas propiedades representan a las tablas `Clients`, `RefreshTokens`, `UserClients` y `Audits`.

4.3.3 Migrations

Una vez definida nuestra clase *DbContext*, y definidas todas las clases del modelo, estamos listos para generar la base de datos.

Para ello, desde la consola del Administrador de paquetes de Visual Studio ejecutamos la orden:

```
Enable-Migrations
```

Esto hará que se cree la carpeta `Migrations` en el proyecto y que se genere el fichero `Configuration.cs` con el método `Seed()` comentado anteriormente, en el que incluiremos el código necesario para rellenar nuestra base de datos con los datos iniciales.

Después, para generar la migración inicial que creará la base de datos ejecutamos:

```
Add-Migration InitialCreate
```

Esto habrá creado un fichero llamado `[yyyyMMddhhmmssc]_InitialCreate.cs` con la clase `InitialCreate` y los métodos `Up()`, encargado de trasladar a base de datos los cambios implementados en la migración, y `Down()`, encargado de dar marcha atrás a dichos cambios en caso de ser necesario.

Una vez hecho esto, tenemos dos opciones para que los cambios indicados se trasladen a la base de datos:

- Arrancar la aplicación
- Ejecutar la siguiente orden desde la consola del Administrador de paquetes:

```
Update-Database
```

En ambos casos se ejecutará el método `Up()` y posteriormente el método `Seed()`.

Además, el cambio en base de datos habrá quedado registrado en la tabla `__MigrationHistory`. De esta manera la próxima vez que se ejecute la orden `Update-Database`, Entity Framework sabrá cuál fue la última migración ejecutada y sólo ejecutaría la siguiente, si existiera.

Si posteriormente quisiéramos añadir cambios a la base de datos, modificaríamos la clase que implementa el *DbContext* o alguna de las clases de modelo y crearíamos una nueva migración, ejecutando el siguiente comando:

```
Add-Migration [Nombre Descriptivo]
```

4.3.4 UserManager y RoleManager

`UserManager` y `RoleManager` son dos APIs que proporciona ASP.NET Identity para facilitar la gestión de usuarios y roles.

`UserManager` expone APIs relacionadas con la gestión de usuarios que salvan los datos en el *UserStore* automáticamente. Hemos usado `UserManager` a través de la clase `ApplicationUserManager`:

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;

namespace OAuthServer.Models
{
    public class ApplicationUserManager : UserManager<User>
    {
        public ApplicationUserManager(OAuthServerDbContext dbContext)
            : base(new UserStore<User>(dbContext))
        {
        }
    }
}
```

Código 4.4. Clase `ApplicationUserManager`

`RoleManager` expone APIs relacionadas con la gestión de roles que salvan los datos automáticamente en el *RoleStore*. Hemos usado `RoleManager` a través de la clase `ApplicationRoleManager`:

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;

namespace OAuthServer.Models
{
    public class ApplicationRoleManager : RoleManager<IdentityRole>
    {
        public ApplicationRoleManager(OAuthServerDbContext dbContext)
            : base(new RoleStore<IdentityRole>(dbContext))
        {
        }
    }
}
```

Código 4.5. Clase `ApplicationRoleManager`

4.3.5 Diagrama de clases

Para terminar de revisar la implementación de esta capa, a continuación se muestra un diagrama en el que se muestran todas las clases de la misma, incluyendo las clases de modelo definidas en la aplicación:

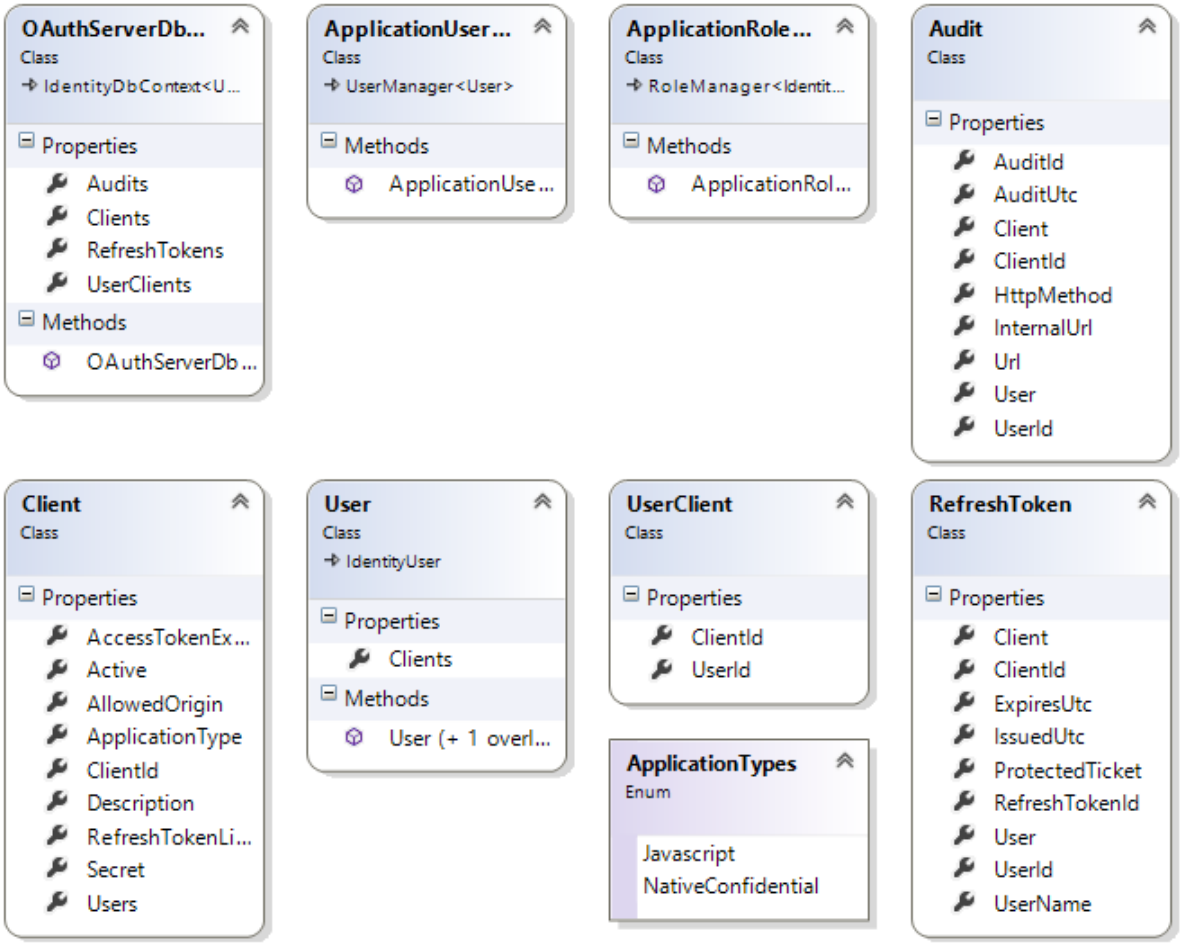


Figura 4.3. Diagrama de clases de capa de acceso a datos

4.4 Capa de negocio

La capa de negocio ha sido implementada en las clases situadas en la carpeta `Services`. Nuestras clases de servicio implementan principalmente las operaciones CRUD de cada tabla, por lo que son bastante sencillas. Como ejemplo analizaremos el código de la clase `ClientService`.

Como ya sabemos, en nuestro proyecto hemos usado el patrón de inyección de dependencias, por tanto, para cada clase que queramos que sus instancias sean inyectadas por el contenedor de inversión de control, definiremos una interfaz que tendrá que implementar.

A continuación se muestra la definición de la interfaz `IclientService` y la parte de la clase `ClientService` que incluye los atributos que albergarán las dependencias a inyectar y el constructor:

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Threading.Tasks;
using OAuthServer.Models;
using OAuthServer.ViewModels;
using Jumuro.Security.Cryptography;
using Jumuro.Security.Cryptography.Extensions;

namespace OAuthServer.Services
{
    public interface IClientService : IDisposable
    {
        Task<IEnumerable<ClientViewModel>> GetClientsAsync();
        Task<ClientViewModel> GetClientAsync(string clientId);
        Task<ClientViewModel> InsertClientAsync(ClientViewModel clientViewModel);
        Task<ClientViewModel> UpdateClientAsync(ClientViewModel clientViewModel);
        Task<ClientViewModel> DeleteClientAsync(string clientId);
        Task<bool> ClientExistsAsync(string clientId);
        Task<bool> ClientHasAssociatedUsersAsync(string clientId);
        Task<ClientsSetupViewModel> GetClientsForSetupAsync();
        Task<IEnumerable<RefreshToken>>
            GetRefreshTokensByClientAsync(string clientId);
    }

    public class ClientService : IClientService
    {
        #region Private Attributes

        /// <summary>
        /// WebApiUsersDbContext object with the connection to database.
        /// </summary>
        private readonly OAuthServerDbContext _dbContext;

        /// <summary>
        /// Object for computing the hash.
        /// </summary>
        private readonly IHashProvider _hashProvider;

        #endregion

        #region Constructor

        /// <summary>
        /// Constructor
        /// </summary>
        public ClientService(OAuthServerDbContext dbContext,
                             IHashProvider hashProvider)
        {
            _dbContext = dbContext;
            _hashProvider = hashProvider;
        }
```

```
        #endregion

        ...
    }
}
```

Código 4.6. Interfaz `IclientService` y clase `ClientService`

Como se puede ver, la interfaz contiene la firma de los métodos que la clase está obligada a implementar. Y la clase tiene dos atributos, uno para la instancia del contexto de base de datos y otro para un objeto con funciones para generar hash, ambas instancias son inyectadas por Autofac en el constructor.

La implementación de los métodos de `IclientService` en `ClientService` es la que sigue:

```
#region GetClientAsync
/// <summary>
/// Gets a client by Id
/// </summary>
/// <param name="clientId"></param>
/// <returns></returns>
public async Task<ClientViewModel> GetClientAsync(string clientId)
{
    var client = await _dbContext.Clients.FindAsync(clientId);
    if (client != null)
    {
        return new ClientViewModel(client);
    }
    else
    {
        return null;
    }
}
#endregion

#region GetClientsAsync
/// <summary>
/// Gets all clients
/// </summary>
/// <returns></returns>
public async Task<IEnumerable<ClientViewModel>> GetClientsAsync()
{
    var clients = await _dbContext.Clients
                                        .OrderBy(c => c.ClientId).ToListAsync();

    return (from c in clients
            select new ClientViewModel(c)
```

```
            ).ToList();
    }
    #endregion

    #region InsertClientAsync
    /// <summary>
    /// Inserts a client.
    /// </summary>
    /// <param name="clientViewModel"></param>
    /// <returns></returns>
    public async Task<ClientViewModel>
        InsertClientAsync(ClientViewModel clientViewModel)
    {
        var insertedClient = _dbContext.Clients
                                .Add(CopyClientViewModelToClient(clientViewModel));

        await _dbContext.SaveChangesAsync();

        return await GetClientAsync(clientViewModel.ClientId);
    }
    #endregion

    #region UpdateClientAsync
    /// <summary>
    /// Updates a client.
    /// </summary>
    /// <param name="clientViewModel"></param>
    /// <returns></returns>
    public async Task<ClientViewModel> UpdateClientAsync(
                                        ClientViewModel clientViewModel)
    {
        var client = await _dbContext.Clients.FindAsync(clientViewModel.ClientId);

        client.Secret = _hashProvider
                            .GetSHA256Hash(clientViewModel.Secret).ToBase64String();
        client.Description = clientViewModel.Description;
        client.ApplicationType =
                            (ApplicationTypes)clientViewModel.ApplicationType.Id;
        client.Active = clientViewModel.IsActive;
        client.AccessTokenExpireTime = clientViewModel.AccessTokenExpireTime;
        client.RefreshTokenLifeTime = clientViewModel.RefreshTokenLifeTime;
        client.AllowedOrigin = clientViewModel.AllowedOrigin;

        // If the client is inactive, delete all tokens in RefreshTokens for it
        if (!clientViewModel.IsActive)
        {
            _dbContext.RefreshTokens.RemoveRange(
                _dbContext.RefreshTokens.Where(
                    rt => rt.ClientId == clientViewModel.ClientId));
        }

        _dbContext.Entry(client).State = EntityState.Modified;
        await _dbContext.SaveChangesAsync();

        return (ClientViewModel)await GetClientAsync(clientViewModel.ClientId);
```

```
}
#endregion

#region DeleteClientAsync
/// <summary>
/// Deletes a client.
/// </summary>
/// <param name="clientId"></param>
/// <returns></returns>
public async Task<ClientViewModel> DeleteClientAsync(string clientId)
{
    Client deletedClient = null;
    Client clientToDelete = await _dbContext.Clients.FindAsync(clientId);

    if (clientToDelete != null)
    {
        deletedClient = _dbContext.Clients.Remove(clientToDelete);

        await _dbContext.SaveChangesAsync();
    }

    return new ClientViewModel(deletedClient);
}
#endregion

#region ClientExistsAsync
/// <summary>
/// Checks if a client id already exists.
/// </summary>
/// <param name="clientId"></param>
/// <returns></returns>
public async Task<bool> ClientExistsAsync(string clientId)
{
    int clientCount = await _dbContext.Clients
                                .CountAsync(c => c.ClientId == clientId);
    return (clientCount > 0);
}
#endregion

#region ClientHasAssociatedUsersAsync
/// <summary>
/// Checks if a client has associated users.
/// </summary>
/// <param name="clientId"></param>
/// <returns></returns>
public async Task<bool> ClientHasAssociatedUsersAsync(string clientId)
{
    int usersCount = await _dbContext.UserClients
                                .CountAsync(uc => uc.ClientId == clientId);
    return (usersCount > 0);
}
#endregion

#region GetRefreshTokensByClientAsync
/// <summary>
```

```
/// Gets all refresh tokens for a client.
/// </summary>
/// <param name="clientId"></param>
/// <returns></returns>
public async Task<IEnumerable<RefreshToken>>
    GetRefreshTokensByClientAsync(string clientId)
{
    var query = from c in _dbContext.Clients
                join rt in _dbContext.RefreshTokens
                    on c.ClientId equals rt.ClientId
                where c.ClientId == clientId
                select rt;

    return await query.ToListAsync();
}
#endregion

#region GetClientsForSetupAsync
/// <summary>
/// Gets all clients for setup
/// </summary>
/// <returns></returns>
public async Task<ClientsSetupViewModel> GetClientsForSetupAsync()
{
    return new ClientsSetupViewModel
    {
        Clients = await GetClientsAsync(),
        ApplicationTypes = (from at in Enum.GetValues(typeof(ApplicationTypes))
                                                .Cast<ApplicationTypes>()
                            select new ApplicationType
                               { Id = (int)at, Description = at.ToString() }
                            ).ToList()
    };
}
#endregion

#region Private Methods

#region CopyClientViewModelToClient
/// <summary>
///
/// </summary>
/// <param name="clientViewModel"></param>
/// <returns></returns>
private Client CopyClientViewModelToClient(ClientViewModel clientViewModel)
{
    if (clientViewModel != null)
    {
        return new Client
        {
            ClientId = clientViewModel.ClientId,
            Secret = _hashProvider
                        .GetSHA256Hash(clientViewModel.Secret).ToBase64String(),
            Description = clientViewModel.Description,
            ApplicationType =
```

```
                        (ApplicationTypes)clientViewModel.ApplicationType.Id,
                Active = clientViewModel.IsActive,
                AccessTokenExpireTime = clientViewModel.AccessTokenExpireTime,
                RefreshTokenLifeTime = clientViewModel.RefreshTokenLifeTime,
                AllowedOrigin = clientViewModel.AllowedOrigin
            };
        }

        return null;
    }
    #endregion

    #endregion
```

Código 4.7. Métodos de la clase `ClientService`

Los métodos implementados son los correspondientes a las operaciones CRUD y alguno más:

- `GetClientsAsync()`: Recupera todos los clientes (cRud).
- `GetClientAsync(string clientId)`: Recupera un cliente por su clientId (cRud).
- `InsertClientAsync(ClientViewModel clientViewModel)`: Inserta un cliente (Crud).
- `UpdateClientAsync(ClientViewModel clientViewModel)`: Actualiza un cliente (cruD).
- `DeleteClientAsync(string clientId)`: Elimina un cliente (cruD).
- `ClientExistsAsync(string clientId)`: Comprueba si existe un cliente con el id recibido.
- `ClientHasAssociatedUsersAsync(string clientId)`: Comprueba si el cliente con el id recibido tiene algún usuario asociado.
- `GetClientsForSetupAsync()`: Recupera todos los clientes devolviendo un *view model* en lugar del modelo.
- `GetRefreshTokensByClientAsync(string clientId)`: Recupera todos los tokens de refresco existentes para el cliente recibido.
- `CopyClientViewModelToClient(ClientViewModel clientViewModel)`: Método privado que crea un objeto de tipo `Client` a partir de otro de tipo `ClientViewModel`.

4.5 Capa de presentación

Como ya hemos adelantado, la capa de presentación la implementan los controladores de Web API, que se sitúan en la carpeta `Controllers`.

Como ejemplo analizaremos el código de la clase `ClientController`.

En el bloque de código 4.8 se muestra la definición de la clase con la sección de atributos y constructor:

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Web.Http;
using OAuthServer.Filters.Attributes;
using OAuthServer.Models;
using OAuthServer.Services;
using OAuthServer.ViewModels;
using Jumuro.WebApi.Extensions;
using Jumuro.WebApi.Extensions.ActionResults;

namespace OAuthServer.Controllers
{
    [ApplicationAuthorize(Roles = "Admin")]
    [RoutePrefix("api/authserver/clients")]
    public class ClientController : ApiController
    {
        #region Private Attributes

        private readonly IClientService _clientService;

        #endregion

        #region Constructor

        public ClientController(IClientService clientService)
        {
            _clientService = clientService;
        }

        #endregion

        ...
    }
}
```

Código 4.8. Clase `ClientController`

Lo primero que nos llama la atención son los atributos de clase `ApplicationAuthorize` y `RoutePrefix`. El primero indica que este controlador sólo será accesible por usuarios que pertenezcan al rol `Admin`. De este modo, nos aseguramos de que cualquier usuario que no pertenezca a este rol no pueda realizar ninguna tarea de administración.

El segundo, establece la URI relativa para todas las acciones del controlador. Los métodos de un controlador son llamados acciones.

Como nuestro controlador hará uso de la clase de servicio `ClientService`, en la sección de atributos se define un objeto que albergará una instancia de esta clase, en realidad de cualquier clase que implemente la interfaz `IclientService`. Y como vemos, dicha instancia será inyectada por Autofac en el constructor.

En el bloque de código 4.9 se muestran las acciones del controlador `ClientController`.

```
#region GetClientsAsync
/// <summary>
/// Gets all clients.
/// </summary>
/// <returns></returns>
[HttpGet]
[Route("")]
public async Task<IHttpActionResult> GetClientsAsync()
{
    var lstClients = await _clientService.GetClientsAsync();

    return Ok<IEnumerable<ClientViewModel>>(lstClients);
}
#endregion

#region GetClientAsync
/// <summary>
/// Gets a client by its id.
/// </summary>
/// <param name="clientId"></param>
/// <returns></returns>
[HttpGet]
[Route("{clientId}")]
public async Task<IHttpActionResult> GetClientAsync(string clientId)
{
    var client = await _clientService.GetClientAsync(clientId);

    if (client == null)
    {
        return this.NotFound(string.Format("Client with Id '{0}' not found.",
```

```
                                clientId));
        }

        return this.Ok<ClientViewModel>(client);
    }
    #endregion

    #region PostClientAsync
    /// <summary>
    /// Inserts a client.
    /// </summary>
    /// <param name="client"></param>
    /// <returns></returns>
    [HttpPost]
    [Route("")]
    public async Task<IHttpActionResult>
        PostClientAsync([FromBody]ClientViewModel clientViewModel)
    {
        if (!ModelState.IsValid)
        {
            return new ModelStateErrorResult(
                ModelState.Keys
                    .SelectMany(k => ModelState[k].Errors)
                    .Select(m => m.ErrorMessage).ToArray(), Request);
        }

        if (await _clientService.ClientExistsAsync(clientViewModel.ClientId))
        {
            return this.Conflict(
                string.Format("A client with the Id '{0}' already exists.",
                              clientViewModel.ClientId));
        }

        var insertedClient = await _clientService.InsertClientAsync(clientViewModel);

        return this.Created<ClientViewModel>(
            Request.RequestUri, insertedClient, "Client created successfully.");
    }
    #endregion

    #region PutClientAsync
    /// <summary>
    /// Updates a client.
    /// </summary>
    /// <param name="client"></param>
    /// <returns></returns>
    [HttpPut]
    [Route("")]
    public async Task<IHttpActionResult>
        PutClientAsync(ClientViewModel clientViewModel)
    {
        if (!ModelState.IsValid)
        {
            return new ModelStateErrorResult(
                ModelState.Keys
```

```
                .SelectMany(k => ModelState[k].Errors)
                .Select(m => m.ErrorMessage).ToArray(), Request);
    }

    if (!await _clientService.ClientExistsAsync(clientViewModel.ClientId))
    {
        return this.NotFound(
            string.Format("A client with the id '{0}' does not exist.",
                          clientViewModel.ClientId));
    }

    var updatedClient = await _clientService.UpdateClientAsync(clientViewModel);

    return this.Ok<ClientViewModel>(clientViewModel,
                                    "Client updated successfully.");
}
#endregion

#region DeleteClientAsync
/// <summary>
/// Deletes a client.
/// </summary>
/// <param name="clientId"></param>
/// <returns></returns>
[HttpDelete]
[Route("{clientId}")]
public async Task<IHttpActionResult> DeleteClientAsync(string clientId)
{
    if (!await _clientService.ClientExistsAsync(clientId))
    {
        return this.NotFound(
            string.Format("A client with the id '{0}' does not exist.",
                          clientId));
    }

    if (await _clientService.ClientHasAssociatedUsersAsync(clientId))
    {
        return this.Conflict(
            string.Format("The client with the id '{0}' has associated users.",
                          clientId));
    }

    var deletedClient = await _clientService.DeleteClientAsync(clientId);

    return this.Ok<ClientViewModel>(deletedClient,
                                    "Client deleted successfully.");
}
#endregion

#region GetClientsForSetupAsync
/// <summary>
/// Gets all clients for setup.
/// </summary>
/// <returns></returns>
[HttpGet]
```

```
[Route("setup")]
public async Task<IHttpActionResult> GetClientsForSetupAsync()
{
    var clients = await _clientService.GetClientsForSetupAsync();

    return Ok<ClientsSetupViewModel>(clients);
}
#endregion

#region GetRefreshTokensByClientAsync
/// <summary>
/// Gets all refresh tokens of a client by its id.
/// </summary>
/// <param name="clientId"></param>
/// <returns></returns>
[HttpGet]
[Route("{clientId}/refreshTokens")]
public async Task<IHttpActionResult>
    GetRefreshTokensByClientAsync(string clientId)
{
    if (!await _clientService.ClientExistsAsync(clientId))
    {
        return this.NotFound(
            string.Format("A client with the id '{0}' does not exist.",
                          clientId));
    }

    var lstRefreshTokens = await _clientService.
                                    GetRefreshTokensByClientAsync(clientId);
    return this.Ok<IEnumerable<RefreshToken>>(lstRefreshTokens);
}
#endregion
```

Código 4.9. Acciones de `ClientController`

En la implementación de nuestro Web API hemos seguido el estándar RESTful (para más información sobre este estándar ver punto [19] de la bibliografía).

Recurso	GET	POST	PUT	DELETE
URI de colección `http://server.com/api/authserver/clients`	**Recupera** una lista de todos los clientes. `GetClientsAsync()`	No implementado	No implementado	No implementado
URI de elemento `http://server.com/api/authserver/clients/{clientId}`	**Recupera** el cliente cuyo Id se recibe en {clientId}. `GetClientAsync()`	**Crea** un nuevo cliente. `PostClientAsync()`	**Actualiza** el cliente cuyo Id se recibe en {clientId}. `PutClientAsync()`	**Borra** el cliente cuyo Id se recibe en {clientId}. `DeleteClientAsync()`

Tabla 4.1. Métodos CRUD de API HTTP RESTful.

Cada acción tiene dos atributos, el verbo HTTP y la ruta. Ambos se corresponden con lo indicado en la tabla 4.1.

Además se pueden observar otras acciones:

- `GetClientsForSetupAsync()`: Recupera todos los clientes para setup (devuelve un view model). Es una operación GET (`HttpGet`), y la ruta es `http://server.com/api/authserver/clients/setup`.
- `GetClientsForSetupAsync()`: Recupera todos los tokens de refresco de un cliente por su Id. Es una operación GET (`HttpGet`), y la ruta es `http://server.com/api/authserver/clients/{clientId}/refreshTokens`.

Las rutas de estas acciones también son RESTful.

5 Aplicación cliente

5.1 Introducción

La aplicación cliente se ha implementado utilizando el framework de JavaScript AngularJS.

Para introducirnos en el mundo AngularJS podemos leer alguno de los enlaces indicados en la bibliografía o alguna de las muchísimas páginas de Internet dedicadas a ello.

Para el desarrollo de la aplicación me he basado en la guía de estilos de Jonh Papa, cuyo enlace podemos ver en el punto número [12] de la bibliografía.

A continuación, en la figura 5.1, se muestra la estructura de directorios de la aplicación en la que se han marcado las partes fundamentales. Más adelante en este capítulo profundizaremos en algunas de ellas.

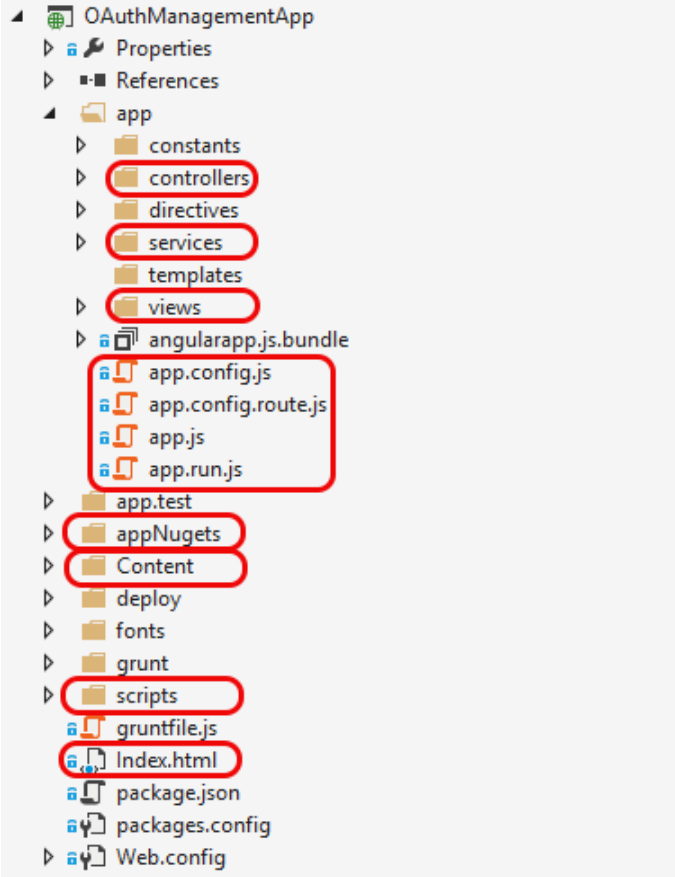


Figura 5.1. Estructura de la aplicación AngularJS

AngularJS es un framework MVC, Modelo Vista Controlador (*Model View Controller*, en inglés).

Componentes MVC en AngularJS:

- Modelo

  Modelos son las propiedades de un ámbito o *scope*. Los *scopes* se conectan al DOM, desde donde las propiedades del *scope* son accedidas a través de enlaces o *bindings.*

- Vista

  La plantilla (HTML con enlaces de datos o *data bindings*) es renderizada en la vista.

- Controlador

  La clase de controlador contiene la lógica de negocio de la aplicación para aderezar el *scope* con funciones y valores.

Mirando la estructura de directorios vemos claramente que los controladores se han implementado en la carpeta `controllers` y las vistas en la carpeta `views`.

Además, parte de la lógica de negocio, principalmente la que tiene que ver con la interacción con el servidor de recursos para manejar los datos, ha sido modularizada y extraída a servicios que se han ubicado en la carpeta `services`.

Aparte de los ya comentados, en la figura 5.1 se han resaltado otros elementos:

- Carpeta `appNugets`. Aquí se han incluido todos los paquetes NuGet AngularJS desarrollados para este proyecto y descritos en el apéndice B.

- Carpeta `scripts`. Contiene todos los ficheros de script externos a la aplicación, incluyendo los scripts de AngularJS, Bootstrap, JQuery, Moment.js, etc.

- Carpeta `Content`. Contiene todos los ficheros de estilos, css y less, utilizados por la aplicación o alguno de los scripts anteriores.

- Ficheros `app.js`, `app.config.js`, `app.config.route.js` y `app.run.js`. Ficheros de script destinados a la definición del módulo principal de nuestra aplicación y configuración y arranque de la misma.

- Fichero `Index.html`. Es la página HTML contenedora de la aplicación AngularJS.

Los dos últimos grupos de ficheros, además de los correspondientes a modelos, vistas y controladores los vamos a ver a continuación en más profundidad.

5.2 Index.html

Como ya sabemos, nuestra aplicación es una SPA (*Single Page Application*), lo que quiere decir que toda la aplicación reside en una sola página HTML.

En nuestro caso esa única página es `Index.html`.

```
<body ng-app="app" ng-controller="AppController">
    ...

    <div ng-view class="container main">
    </div>
    ...
</body>
```

Código 5.1. Extracto del elemento `body` del fichero `Index.html`.

Lo más destacable del bloque de código 5.1 es:

- Atributo `ng-app` del elemento `body`. Carga automáticamente la aplicación angular.

- Atributo `ng-view` de uno de los elementos `div`. Sin entrar en muchos detalles, en este elemento `div` se renderizarán las plantillas de nuestras vistas.

5.3 Ficheros app*.js

Como ya hemos dicho, estos ficheros contienen scripts destinados a la definición del módulo principal de nuestra aplicación y configuración y arranque de la misma. Vamos a comentar los más interesantes.

5.3.1 Fichero app.js

Este fichero contiene la definición del módulo `"app"` indicado en la directiva `ngApp`, que es el módulo principal de nuestra aplicación. Como vemos en el bloque de código 5.2, el segundo parámetro del método `angular.module()` son otros módulos angular utilizados por nuestra aplicación, de esta manera se *inyectan* dichos módulos para que estén disponibles para nuestra aplicación.

```
(function () {
    'use strict';

    // Declares how the application should be bootstrapped. See:
    // http://docs.angularjs.org/guide/module
    //'mgcrea.ngStrap'
    angular.module('app', ['ngRoute', 'ngLocale',
                           'jumuro.crudRest', 'jumuro.oAuth', 'jumuro.spinner',
                           'jumuro.modal', 'jumuro.grid', 'jumuro.validations',
                           'jumuro.webapi']);
})();
```

Código 5.2. Fichero `app.js`

5.3.2 Fichero app.config.route.js

En este fichero se configuran todas las rutas dentro de nuestra aplicación. Como se observa en el bloque de código 5.3, para cada ruta se especifica la url de la plantilla que genera la vista y el controlador que implementa la lógica. Al utilizar `controllerAs`, damos nombre al *scope* del controlador, lo que hace que el *binding* dentro de la vista sea a elementos de un objeto (por ejemplo `vm.name` en vez de `name`), lo que es más contextual, aporta legibilidad y evita hacer llamadas a `$parent` (*scope* padre del actual) en vistas con controladores anidados.

```
(function () {
    'use strict';

    angular.module('app')
        .config(configureRoute);

    configureRoute.$inject = ['$routeProvider'];
```

```
    function configureRoute($routeProvider) {
        $routeProvider
            .when('/login', {
                templateUrl: './app/views/login.html',
                controller: 'LoginController',
                controllerAs: 'vm'
            })
            .when('/role', {
                templateUrl: './app/views/role.html',
                controller: 'RoleController',
                controllerAs: 'vm'
            })
            .when('/client', {
                templateUrl: './app/views/client.html',
                controller: 'ClientController',
                controllerAs: 'vm'
            })
            .when('/refreshToken', {
                templateUrl: './app/views/refreshtoken.html',
                controller: 'RefreshTokenController',
                controllerAs: 'vm'
            })
            .when('/user', {
                templateUrl: './app/views/user.html',
                controller: 'UserController',
                controllerAs: 'vm'
            })
            .when('/home', {
                templateUrl: './app/views/home.html'
            })
            .otherwise({ redirectTo: '/home' });
    }
})();
```

Código 5.3. Fichero `app.config.route.js`

5.4 Vista, controlador y servicio

Para comentar los detalles de implementación de las vistas, los controladores y los servicios, y al igual que hicimos con el servidor de recursos, vamos a tomar como ejemplo la parte de gestión de clientes.

5.4.1 Vista

Para la gestión de clientes disponemos de dos vistas. La principal, definida en el fichero `Client.html`, que contiene una tabla con todos los clientes dados de alta en el sistema, y la de

edición, `ClientPopup.html`, que define una ventana popup para la creación de nuevos clientes o edición de los ya existentes. En los bloques de código 5.4 y 5.5 se muestran ambas.

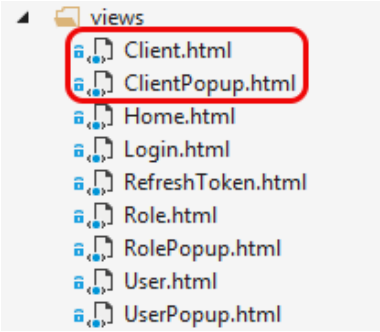


Figura 5.2. Vistas

Como vemos, `Client.html` contiene al elemento `jumuro-grid` que es una directiva implementada en el paquete NuGet `Jumuro.Angular.Grid`, y define un grid con unas opciones de configuración obtenidas del *scope* del controlador por medio del elemento `vm.gridClientsOptions`.

```
<div class="app_containter">
    <div class="container">
        <jumuro-grid jumuro-gridoptions="vm.gridClientsOptions"></jumuro-grid>
    </div>
</div>
```

Código 5.4. Vista `Client.html`

`ClientPopup.html` contiene un formulario completo con todas las propiedades de un cliente, incluyendo validadores para cada una de ellos.

```
<div class="modal-content">
    <div class="modal-header">
        <h3 class="modal-title">{{vm.popupHeaderText}} Client</h3>
    </div>
    <div class="modal-body">
        <form class="form-horizontal" name="form.Client">
            <fieldset>
                <div class="form-group">
                    <label class="col-md-3 control-label">Id</label>
                    <div class="col-md-9"
```

```
                    ng-class=
                        "{ 'has-error' : form.Client.ctrlClientId.$invalid
                                && !form.Client.ctrlClientId.$pristine }">
                <input type="text" class="form-control"
                       ng-maxlength="100"
                       name="ctrlClientId"
                       placeholder="Please enter client id..."
                       ng-model="vm.client.clientId"
                       required ng-disabled="isEdit" />
                <div class="error-group">
                    <div ng-show="form.Client.ctrlClientId.$invalid &&
                                  !form.Client.ctrlClientId.$pristine">
                        <p class="help-block" ng-show=
                           "form.Client.ctrlClientId.$error.required">
                           Client id is required.
                        </p>
                        <p class="help-block" ng-show=
                           "form.Client.ctrlClientId.$error.maxlength">
                           Client id is too long.
                        </p>
                    </div>
                </div>
            </div>
        </div>
        <div class="form-group" ng-hide="isEdit">
            <label class="col-md-3 control-label">Secret</label>
            <div class="col-md-9"
                 ng-class=
                   "{ 'has-error' : form.Client.ctrlSecret.$invalid &&
                                    !form.Client.ctrlSecret.$pristine }">
                <input type="text" class="form-control"
                       name="ctrlSecret"
                       placeholder="Please enter secret..."
                       ng-model="vm.client.secret"
                       required />
                <div class="error-group">
                    <div ng-show="form.Client.ctrlSecret.$invalid &&
                                  !form.Client.ctrlSecret.$pristine">
                        <p class="help-block"
                        ng-show="form.Client.ctrlSecret.error.required">
                           Client secret is required.
                        </p>
                    </div>
                </div>
            </div>
        </div>
        <div class="form-group">
            <label class="col-md-3 control-label">Description</label>
            <div class="col-md-9"
                 ng-class=
                 "{ 'has-error' : form.Client.ctrlDescription.$invalid
                          && !form.Client.ctrlDescription.$pristine }">
                <input type="text" class="form-control"
                       ng-maxlength="100"
                       name="ctrlDescription"
```

```
                        placeholder="Please enter description..."
                        ng-model="vm.client.description"
                        required />
                    <div class="error-group">
                        <div ng-show=
                             "form.Client.ctrlDescription.$invalid &&
                              !form.Client.ctrlDescription.$pristine">
                            <p class="help-block"
                               ng-show=
                              "form.Client.ctrlDescription.$error.required">
                               Description is required.
                            </p>
                            <p class="help-block"
                               ng-show=
                             "form.Client.ctrlDescription.$error.maxlength">
                               Description is too long.
                            </p>
                        </div>
                    </div>
                </div>
            </div>
            <div class="form-group">
                <label class="col-md-3 control-label">
                       Application Type
                </label>
                <div class="col-md-9"
                     ng-class=
                   "{ 'has-error' : form.Client.ctrlApplicationType.$invalid
                          && !form.Client.ctrlApplicationType.$pristine }">
                    <select class="form-control"
                            name="ctrlApplicationType"
                            ng-model="vm.client.applicationType"
                            ng-options="applicationType as
                                        applicationType.description for
                                        applicationType in vm.applicationTypes"
                            required>
                        <option value="">
                                -- Select Application Type –
                        </option>
                    </select>
                    <div class="error-group">
                        <div ng-show=
                             "form.Client.ctrlApplicationType.$invalid &&
                              !form.Client.ctrlApplicationType.$pristine">
                            <p class="help-block" ng-show=
                          "form.Client.ctrlApplicationType.$error.required">
                               Application Type is required.
                            </p>
                        </div>
                    </div>
                </div>
            </div>
            <div class="form-group">
                <label class="col-md-3 control-label">
                       Access Token Expire Time
```

```
                    </label>
                    <div class="col-md-9"
                         ng-class=
                    "{ 'has-error' : form.Client.ctrlAccessTokenExpireTime.$invalid
                && (!form.Client.ctrlAccessTokenExpireTime.$pristine || isEdit) }">
                        <input type="number" class="form-control"
                               name="ctrlAccessTokenExpireTime"
                               placeholder=
                           "Please enter access token expire time in minutes..."
                               ng-model="vm.client.accessTokenExpireTime"
                               required min="1"
                               lower-than="{{vm.client.refreshTokenLifeTime}}"/>
                        <div class="error-group">
                            <div ng-show=
                              "form.Client.ctrlAccessTokenExpireTime.$invalid &&
                              !form.Client.ctrlAccessTokenExpireTime.$pristine">
                                <p class="help-block" ng-show=
                            "form.Client.ctrlAccessTokenExpireTime.$error.required">
                                    Access token expire time is required.
                                </p>
                                <p class="help-block" ng-show=
                             "form.Client.ctrlAccessTokenExpireTime.$error.number">
                                    Access token expire time must be a number.
                                </p>
                                <p class="help-block" ng-show
                              ="form.Client.ctrlAccessTokenExpireTime.$error.min">
                                    Access token expire time min is 1.
                                </p>
                            </div>
                            <div ng-show=
                              "form.Client.ctrlAccessTokenExpireTime.$invalid &&
                              (!form.Client.ctrlAccessTokenExpireTime.$pristine
                               || isEdit)">
                                <p class="help-block" ng-show=
                          "form.Client.ctrlAccessTokenExpireTime.$error.lowerThan">
                                    Access token expire time must be lower than
                                    refresh token life time.
                                </p>
                            </div>
                        </div>
                    </div>
                </div>
                <div class="form-group">
                    <label class="col-md-3 control-label">
                           Refresh Token Life Time
                    </label>
                    <div class="col-md-9"
                         ng-class=
                  "{ 'has-error' : form.Client.ctrlRefreshTokenLifeTime.$invalid &&
                               !form.Client.ctrlRefreshTokenLifeTime.$pristine }">
                        <input type="number" class="form-control"
                               name="ctrlRefreshTokenLifeTime"
                               placeholder=
                            "Please enter refresh token life time in minutes..."
                               ng-model="vm.client.refreshTokenLifeTime"
```

```
                        required min="2" />
                <div class="error-group">
                    <div ng-show=
                       "form.Client.ctrlRefreshTokenLifeTime.$invalid &&
                       !form.Client.ctrlRefreshTokenLifeTime.$pristine">
                        <p class="help-block" ng-show=
                     "form.Client.ctrlRefreshTokenLifeTime.$error.required">
                           Refresh token life time is required.
                        </p>
                        <p class="help-block" ng-show=
                    "form.Client.ctrlRefreshTokenLifeTime.$error.number">
                           Refresh token life time must be a number.
                        </p>
                        <p class="help-block" ng-show=
                      "form.Client.ctrlRefreshTokenLifeTime.$error.min">
                           Refresh token life time min is 2.
                        </p>
                    </div>
                </div>
            </div>
        </div>
        <div class="form-group">
            <label class="col-md-3 control-label">Allowed Origin</label>
            <div class="col-md-9"
                 ng-class=
             "{ 'has-error' : form.Client.ctrlAllowedOrigin.$invalid &&
                           !form.Client.ctrlAllowedOrigin.$pristine }">
                <input type="text" class="form-control"
                       ng-maxlength="100"
                       name="ctrlAllowedOrigin"
                       placeholder="Please enter allowed origin..."
                       ng-model="vm.client.allowedOrigin"
                       required />
                <div class="error-group">
                    <div ng-show=
                         "form.Client.ctrlAllowedOrigin.$invalid &&
                          !form.Client.ctrlAllowedOrigin.$pristine">
                        <p class="help-block"
                           ng-show=
                        "form.Client.ctrlAllowedOrigin.$error.required">
                           Allowed origin is required.
                        </p>
                        <p class="help-block"
                           ng-show=
                       "form.Client.ctrlAllowedOrigin.$error.maxlength">
                           Allowed origin is too long.
                        </p>
                    </div>
                </div>
            </div>
        </div>
        <div class="form-group">
            <label class="col-md-3 control-label">Is Active</label>
            <div class="col-md-9 form-control-static">
                <input type="checkbox" name="ctrlIsActive"
```

```
                                ng-model="vm.client.isActive" />
                        </div>
                    </div>
                </fieldset>
            </form>
        </div>
        <div class="modal-footer">
            <button class="btn btn-primary"
                    ng-click="vm.ok()" data-ng-disabled="form.Client.$invalid"
                    id="btnOkAddClientPopUp">
                    {{vm.popupOkText}}
            </button>
            <button class="btn btn-warning"
                    ng-click="vm.cancel()" id="btnCloseAddClientPopUp">
                    Close
            </button>
        </div>
    </div>
```

Código 5.5. Vista `ClientPopup.html`

Todos los atributos que empiezan por `ng-` son directivas de Angular que se usan para establecer comportamientos o transformar a los elementos del DOM (*Document Object Model*).

Para nombrar las directivas se utiliza la nomenclatura *camel case*, pero para utilizarlas se transforma el nombre *camel case* a *snake case*, por eso para utilizar la directiva `ngClass` se escribe `ng-class` en la plantilla HTML.

Algunas de las directivas utilizadas son:

- `ngClass`: Permite establecer dinámicamente clases CSS a un elemento HTML.
- `ngModel`: Enlaza un control de formulario con una propiedad del *scope*.
- `ngShow`: Muestra u oculta el elemento HTML dependiendo del resultado de la expresión que recibe.
- `ngClick`: Ejecuta una expresión en el evento click del elemento HTML.

Un poco más adelante, en el punto 5.5, podremos ver el aspecto de todas las vistas de la aplicación.

5.4.2 Controlador

Para la administración de los clientes disponemos de dos controladores, uno para cada una de las vistas anteriores, y cada uno implementado en su fichero independiente.

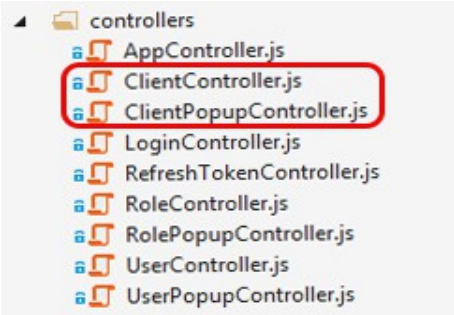


Figura 5.3. Controladores

`ClientController.js` para gestionar la vista `Client.html`, y `ClientPopupController.js` para gestionar la vista `ClientPopup.html`.

En el bloque de código 5.6 podemos ver el código del primero de los controladores indicados.

En la primera parte vemos la declaración de un controlador llamado `'ClientController'` que se crea en el módulo `'app'`. Un controlador es un constructor y por convención lo nombramos empezando por mayúsculas.

A continuación se indican las dependencias que serán inyectadas en el controlador. Aquí se indican los módulos y servicios que serán utilizados por nuestro controlador.

Y por último se define una función que es la que implementa la lógica del controlador.

```
(function() {
    'use strict';

    angular.module('app')
        .controller('ClientController', ClientController);

    ClientController.$inject = ['modalService', '$modal', 'toaster',
                                'clientFactory'];

    function ClientController(modalService, $modal, toaster, clientFactory) {
        var vm = this;

        vm.applicationTypes = [];
```

```
        vm.gridClientsOptions = {};
        vm.openPopup = openPopup;
        vm.deleteClient = deleteClient;

        initialize();

        //#region Scope Methods

        //Open Add/Edit popup
        function openPopup(isEdit, client) {
            var _clientToEdit = angular.copy(client);

            var modalInstance = $modal.open({
                windowClass: 'modalWindow',
                templateUrl: './app/views/ClientPopup.html',
                controller: 'ClientPopupController as vm',
                resolve: {
                    items: function () {
                        return {
                            isEdit: isEdit,
                            client: _clientToEdit,
                            applicationTypes: vm.applicationTypes
                        };
                    }
                }
            });

            modalInstance.result.then(function (result) {
                if (result.isRefresh) {
                    if (result.isEdition) {
                        // Refresh grid after edition
                        for (var i = 0;
                             i < vm.gridClientsOptions.dataList.length;
                             i++) {
                            if (vm.gridClientsOptions.dataList[i].clientId ==
                                result.client.clientId) {
                                vm.gridClientsOptions.dataList[i] =
                                    result.client;
                                break;
                            }
                        }
                    }
                    else {
                        // Refresh grid after insertion
                        vm.gridClientsOptions.dataList.push(result.client);
                    }
                }
            }, function () {

            });
        };

        //Open confirm popup to delete client
        function deleteClient(client, index) {
            //build the details list
```

```
            //check if the user has the requested permission
            var details = [
                {
                    name: "Client Id", value: client.clientId
                },
                {
                    name: "Description", value: client.description,
                }
            ];

            var modalOptions = {
                headerText: 'Please Confirm',
                message: 'Are you sure you want to delete this client?',
                buttons: { ok: 'Yes', cancel: 'No' },
                modalType: 'confirm',
                details: details
            };

            var modal = modalService.modal(modalOptions);
            modal.then(function (result) {
                clientFactory.deleteClient(client.clientId)
                    .then(function (data) {
                        var originalIndex =
                            vm.gridClientsOptions.dataList.indexOf(client);
                        if (originalIndex != -1) {
                            vm.gridClientsOptions.dataList.splice(originalIndex,
                                                                  1);
                        }
                    });
            }, function () {

            });
        }

        //#endregion Scope Methods

        //#region Private Methods

        //Initializes page
        function initialize() {
            configureClientsGrid();
            getClientsForSetup();
        }

        //Set the clients grid configuration
        function configureClientsGrid()
        {
            var columnList = [
                { name: 'clientId', header: 'Client Id',
                  isFilter: false, isOrder: true },
                { name: 'description', header: 'Description',
                  isFilter: false, isOrder: true },
                { name: 'applicationType.description',
                  header: 'Application Type', isFilter: false, isOrder: true },
                { name: 'accessTokenExpireTime',
```

```
                header: 'Access Token Expire Time',
                isFilter: false, isOrder: true },
            { name: 'refreshTokenLifeTime',
                header: 'Refresh Token Life Time',
                isFilter: false, isOrder: true },
            { name: 'allowedOrigin', header: 'Allowed Origin',
                isFilter: true, isOrder: true },
            { name: 'isActive', header: 'Is Active',
                isFilter: true, isOrder: true }
        ];

        vm.gridClientsOptions = {
            columnList: columnList,
            noDataMessage: 'There are no clients',
            animate: true,
            crudOptions: {
                enable: true,
                insert: {
                    security: {
                        route: '/client',
                        action: 'insert'
                    },
                    method: function () {
                        openPopup(false, null);
                    }
                },
                edit: {
                    security: {
                        route: '/client',
                        action: 'edit'
                    },
                    method: function (data) {
                        openPopup(true, data);
                    }
                },
                delete: {
                    security: {
                        route: '/client',
                        action: 'delete'
                    },
                    method: function (data) {
                        deleteClient(data);
                    }
                }
            }
        };
    }

    //Get all clients for setup
    function getClientsForSetup()
    {
        clientFactory.getClientsForSetup()
            .then(function (data) {
                vm.gridClientsOptions.dataList = data.clients;
                vm.applicationTypes = data.applicationTypes;
```

```
                });
        }

        //#endregion Private Methods
    }
})();
```

Código 5.6. Controlador `ClientController.js`

Ya dentro de la función del controlador, en primer lugar se declara una variable `vm` que será la que exponga el *scope* a la vista (ver parámetro `controllerAs` en `app.config.route.js`) y a la que se añadirán todas las variables y métodos que se vayan a utilizar en ella.

Haciéndolo de este modo, de un simple vistazo vemos los miembros enlazables publicados por el controlador a la vista.

Seguidamente se llama al método privado `initialize()` que, como su nombre indica, realiza las tareas para la inicialización.

Debajo se definen las funciones de *scope* indicadas en el objeto `vm`, `openPopup()` y `deleteClient()`. La primera abre el popup de inserción/edición de clientes y evalúa el resultado tras la ejecución del mismo, actualizando la base de datos y el grid en caso necesario. La segunda borra un cliente de base datos y también actualiza el grid. Por supuesto, el acceso a base de datos es a través del servidor de recursos, y la conexión con el mismo se realiza en el servicio `clientFactory`.

Por último se definen los métodos privados. El método `configureClientsGrid()` establece las opciones de configuración del grid y las hace disponibles a la vista a través de la variable de *scope* `vm.gridClientsOptions`. El método `getClientsForSetup()` obtiene los clientes existentes en base de datos. El método `initialize()` llama a los dos métodos anteriores para realizar las tareas de inicialización.

En el bloque de código 5.7 se muestra la implementación del controlador `ClientPopupController.js`. La estructura es similar al anterior.

En la función principal del controlador, lo primero que se hace es inicializar el objeto `vm` con los objetos y funciones que serán usados por al vista, incluyendo los parámetros recibidos en la llamada desde `ClientController`.

```
(function () {
    'use strict';

    angular.module('app')
        .controller('ClientPopupController', ClientPopupController);

    ClientPopupController.$inject = ['$modalInstance', 'webapiConstants',
                                     'jumuroCrudRESTService', 'items',
                                     'webapiAppConfigConstants'];

    function ClientPopupController($modalInstance, webapiConstants,
                                   jumuroCrudRESTService, items,
                                   webapiAppConfigConstants) {
        var vm = this;

        vm.form = {};
        vm.resultPopUp = {
            isRefresh: false,
            client: []
        };
        vm.client = items.client;
        vm.applicationTypes = items.applicationTypes;
        vm.isEdit = items.isEdit;
        vm.popupHeaderText = '';
        vm.popupOkText = '';
        vm.ok = ok;
        vm.cancel = cancel;

        initialize();

        //#region Scope Methods

        function ok() {
            //Mark as refresh
            vm.resultPopUp.isRefresh = true;

            if (items.isEdit) {
                clientFactory.updateClient(vm.client)
                    .then(function (data) {
                        //Mark as edition
                        vm.resultPopUp.isEdition = true;

                        //Set modified client
                        vm.resultPopUp.client = data;

                        //Close popup
                        $modalInstance.close(vm.resultPopUp);
                    });
            }
            else {
                clientFactory.createClient(vm.client)
                    .then(function (data) {
                        //Mark as edition
                        vm.resultPopUp.isEdition = false;
```

```
                    //Set added client
                    vm.resultPopUp.client = data;

                    //Close popup
                    $modalInstance.close(vm.resultPopUp);
                });
            }
        }

        function cancel() {
            $modalInstance.dismiss('cancel');
        }

        //#endregion Scope Methods

        //#region Private Methods

        //Initialize page
        function initialize() {
            if (items.isEdit) {
                // Select current application type of the client
                for (var i = 0; i < vm.applicationTypes.length; i++) {
                    if (vm.client.applicationType.id
                        == vm.applicationTypes[i].id) {
                        vm.client.applicationType = vm.applicationTypes[i];
                        break;
                    }
                }

                vm.popupHeaderText = 'Edit';
                vm.popupOkText = 'Update';
            }
            else {
                vm.popupHeaderText = 'Add';
                vm.popupOkText = 'Add';
            }
        };

        //#endregion Private Methods
    }
})();
```

Código 5.7. Controlador `ClientPopupController.js`

Después tenemos el método de *scope* `ok()`, que realiza la llamada correspondiente al servicio para crear o actualizar un cliente, establece la respuesta que será devuelta y cierra el popup devolviendo dicha respuesta.

El método `cancel()` simplemente cierra el popup.

Y por último el método privado `initialize()` realiza las tareas de inicialización. En caso de actualización de cliente, preselecciona el valor correspondiente del drop de tipos de aplicación y establece los textos de la cabecera y del botón de actualización a "Edit" y "Update" respectivamente. En caso de inserción establece los textos de la cabecera y del botón de actualización a "Add" en ambos casos.

5.4.3 Servicio

En Angular existen tres tipos de servicios: *provider*, *factory* y *service*.

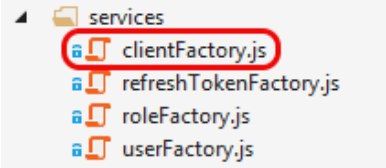


Figura 5.4. Servicios

Las diferencias entre ellos escapan al ámbito de este documento. Lo que sí diremos es que en nuestro caso hemos implementado los servicios siguiendo el patrón de *factory*.

```
(function () {
    'use strict';

    angular
        .module('app')
        .factory('clientFactory', clientFactory);

    clientFactory.$inject = ['webapiConstants', 'jumuroCrudRESTService',
                             'webapiAppConfigConstants'];

    function clientFactory(webapiConstants, jumuroCrudRESTService,
                           webapiAppConfigConstants) {
        var service = {
            getClientsForSetup: getClientsForSetup,
            createClient: createClient,
            updateClient: updateClient,
            deleteClient: deleteClient
        };

        return service;

        function getClientsForSetup() {
            return jumuroCrudRESTService
                .restGet(webapiAppConfigConstants.appConfig.ApiURL +
                         webapiConstants.urls.ApiUrl.clientsSetup, false)
```

```
            .then(getClientsForSetupComplete)
            .catch(getClientsForSetupFailed);

        function getClientsForSetupComplete(data) {
            return data;
        }

        function getClientsForSetupFailed(error) {
            //Log error
            return $q.reject(error);
        }
    }

    function createClient(client) {
        return jumuroCrudRESTService
            .restPost(client, webapiAppConfigConstants.appConfig.ApiURL +
                      webapiConstants.urls.ApiUrl.postClient, false)
            .then(createClientComplete)
            .catch(createClientFailed);

        function createClientComplete(data) {
            return data;
        }

        function createClientFailed(error) {
            //Log error
            return $q.reject(error);
        }
    }

    function updateClient(client) {
        return jumuroCrudRESTService
            .restPut(client, webapiAppConfigConstants.appConfig.ApiURL +
                     webapiConstants.urls.ApiUrl.putClient, false)
            .then(updateClientComplete)
            .catch(updateClientFailed);

        function updateClientComplete(data) {
            return data;
        }

        function updateClientFailed(error) {
            //Log error
            return $q.reject(error);
        }
    }

    function deleteClient(clientId) {
        return jumuroCrudRESTService
            .restDelete(webapiAppConfigConstants.appConfig.ApiURL +
                        webapiConstants.urls.ApiUrl.deleteClient
                        .replace("{clientId}", clientId), false)
            .then(deleteClientComplete)
            .catch(deleteClientFailed);
```

```
            function deleteClientComplete(data) {
                return data;
            }

            function deleteClientFailed(error) {
                //Log error
                return $q.reject(error);
            }
        }
    }
})();
```

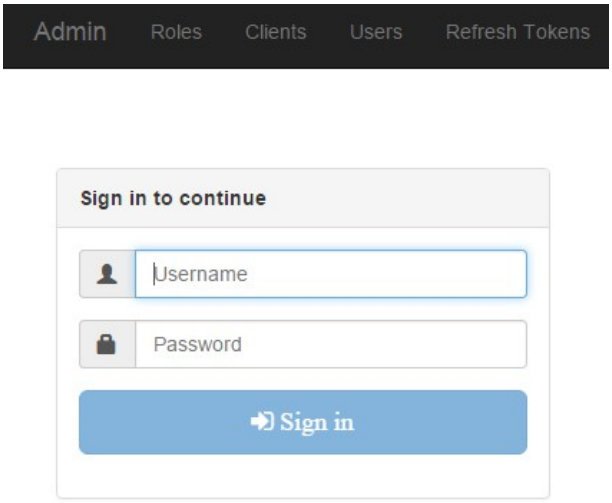
Código 5.8. Servicio `clientFactory.js`

Como podemos ver en el bloque de código 5.8, los métodos de nuestro servicio son muy sencillos, pero abstraen totalmente al controlador de la comunicación con nuestro web API y ayudan bastante a modularizar el código.

Los métodos implementados son `getClientsForSetup()`, `createClient()`, `updateClient()` y `deleteClient()`, que implementan las cuatro operaciones CRUD.

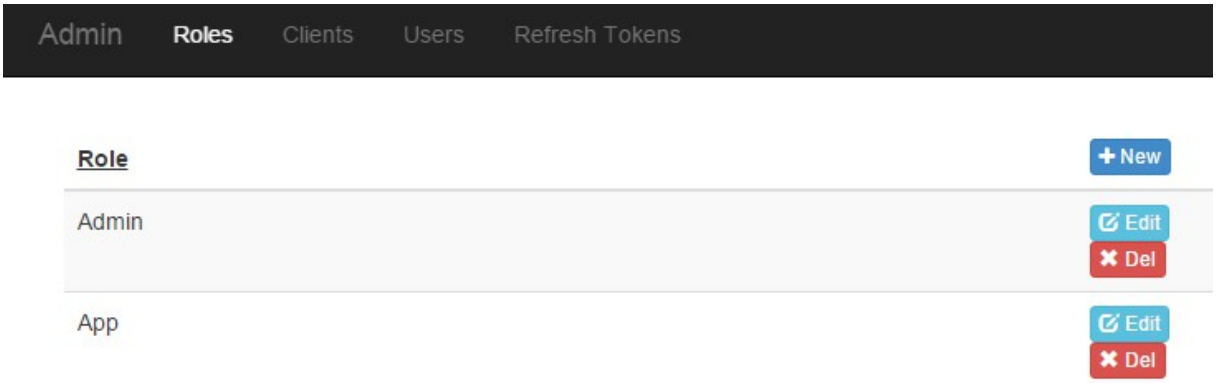
5.5 Ventanas de la aplicación

A continuación se muestran capturas de pantalla de todas las ventanas de la aplicación:

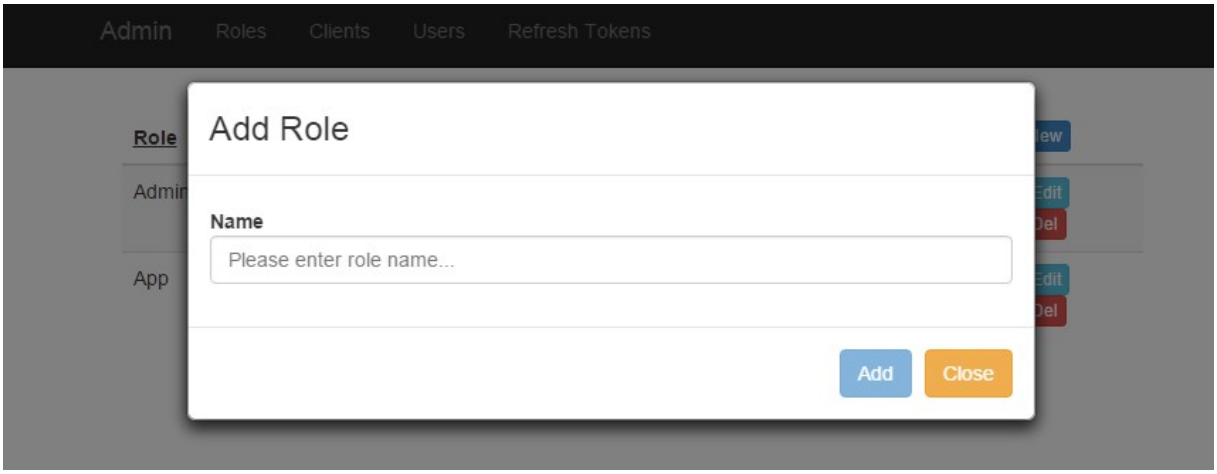


Ventana 1. Login

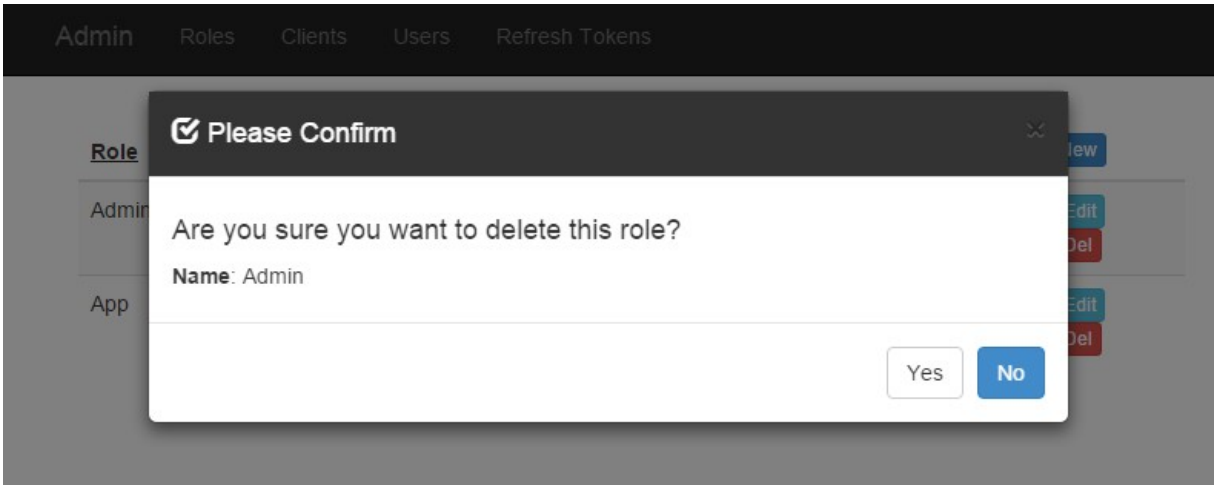
Ventana 2. Home



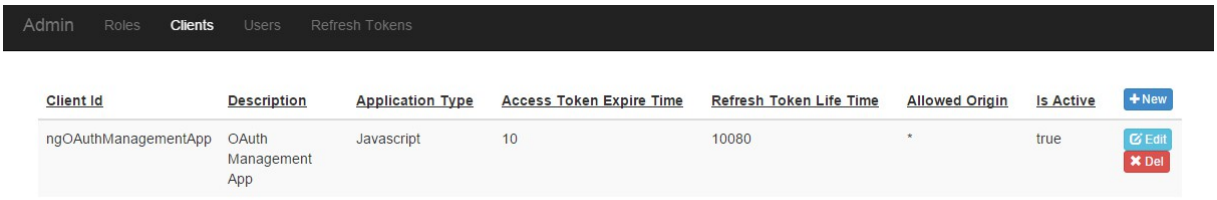
Ventana 3. Roles



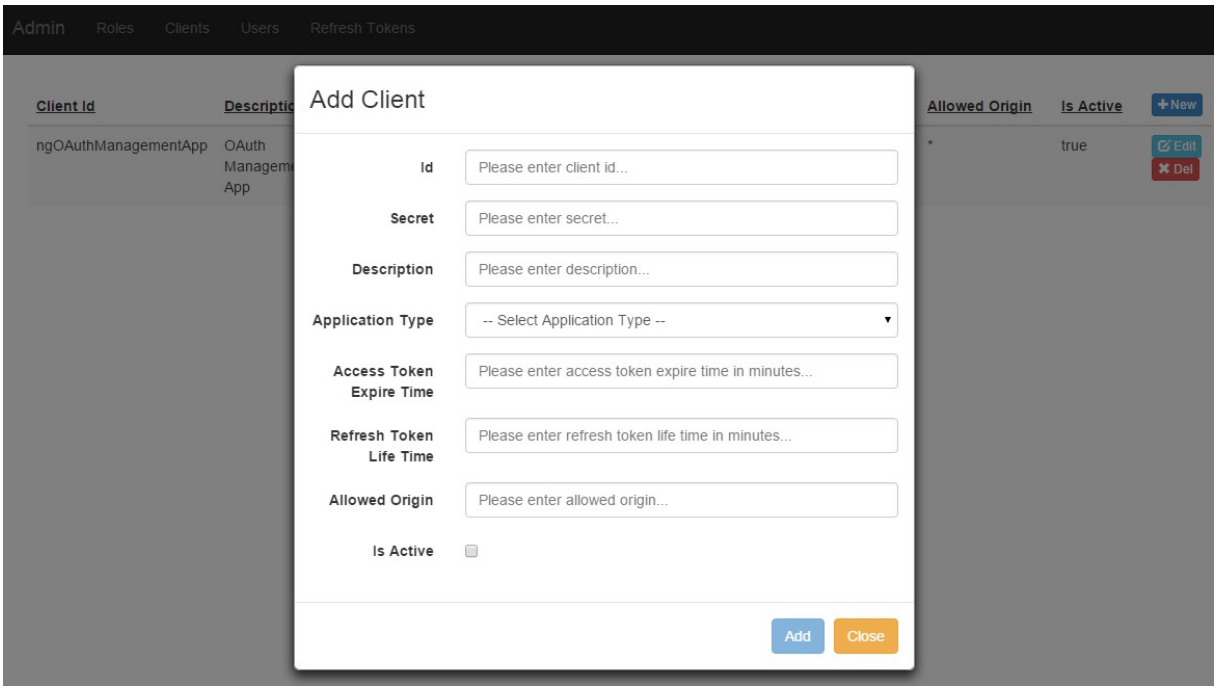
Ventana 4. Añadir rol (similar a editar rol)



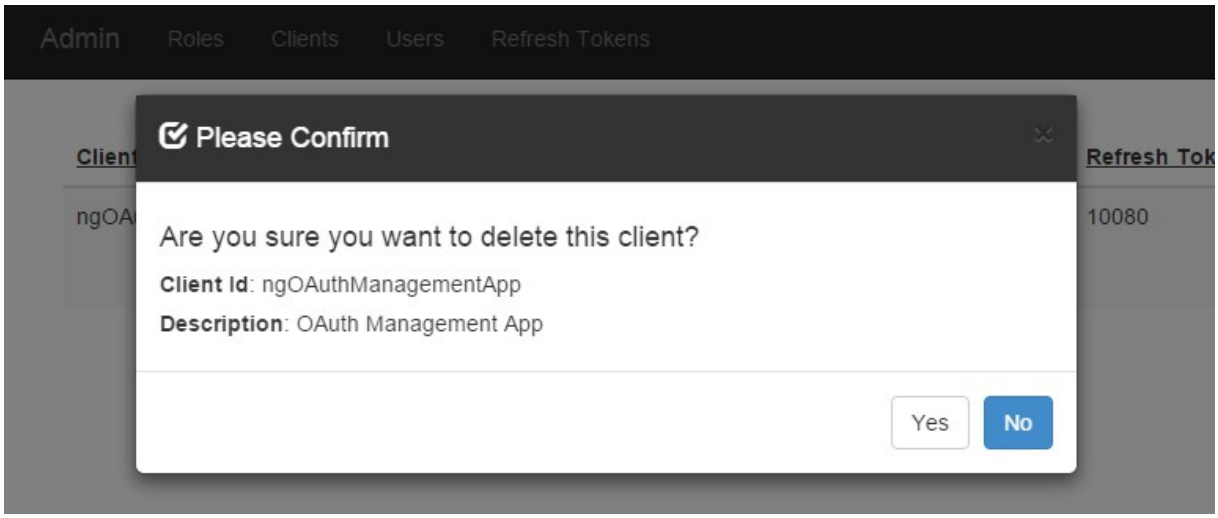
Ventana 5. Borrar rol



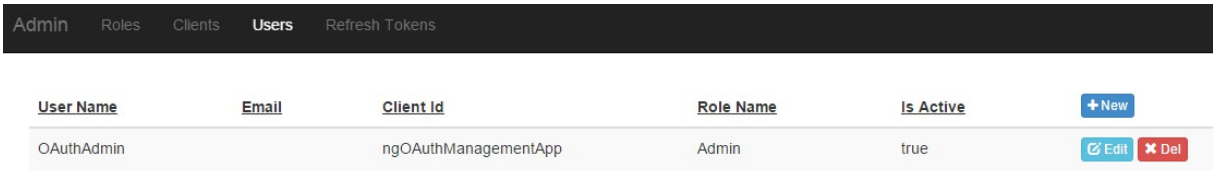
Ventana 6. Clientes



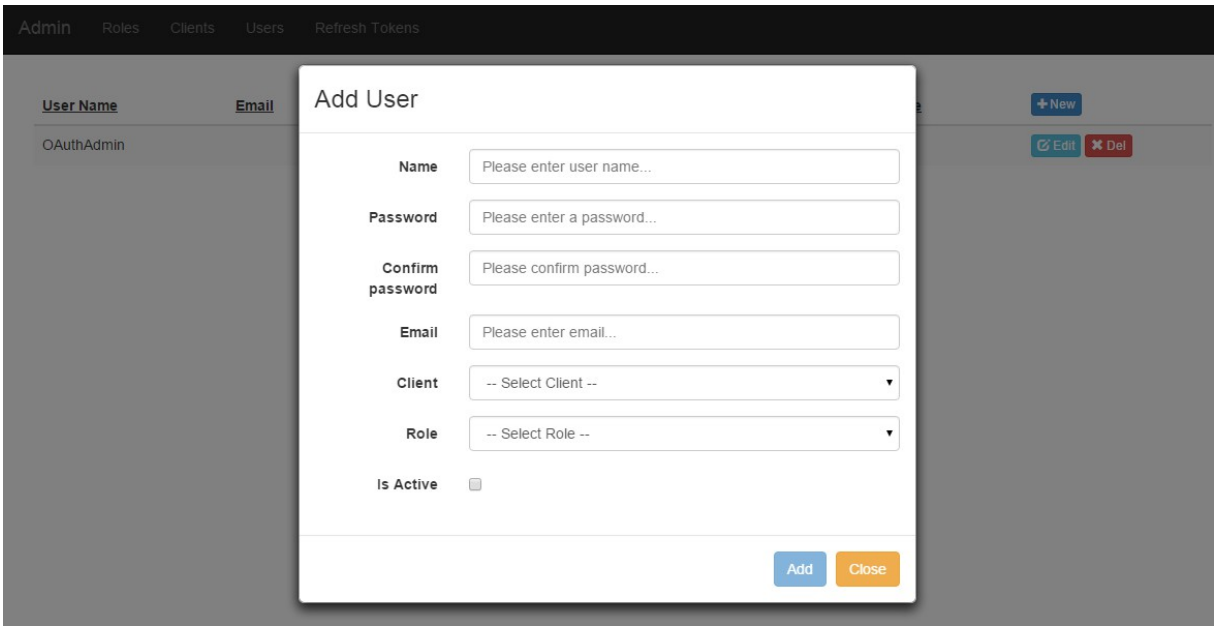
Ventana 7. Añadir cliente (similar a editar cliente)



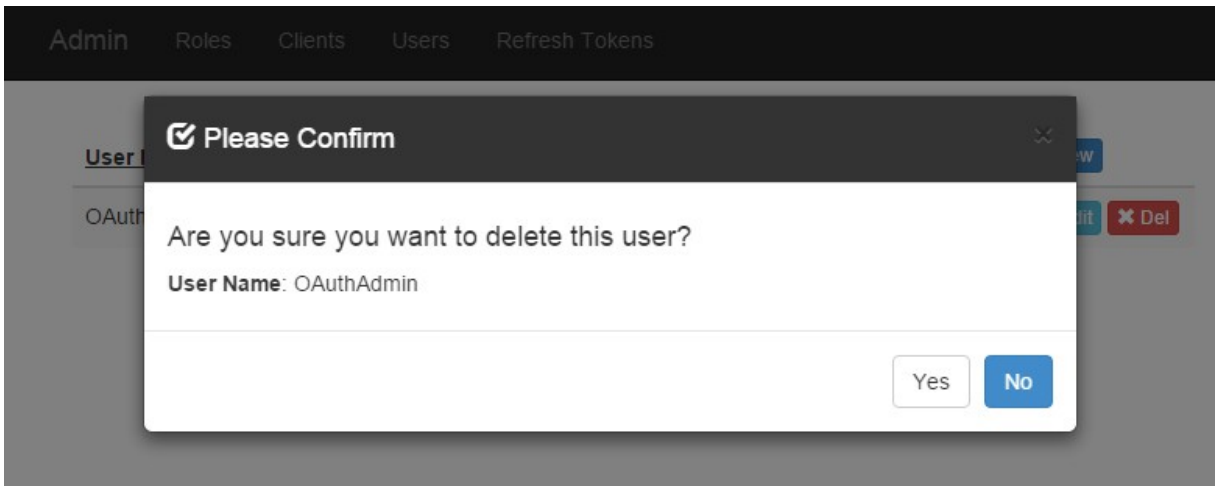
Ventana 8. Borrar cliente



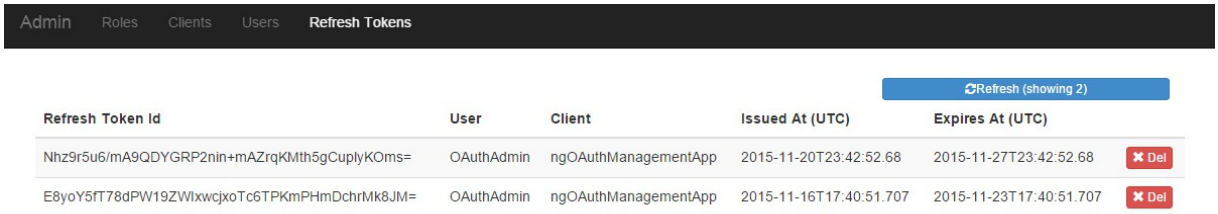
Ventana 9. Usuarios



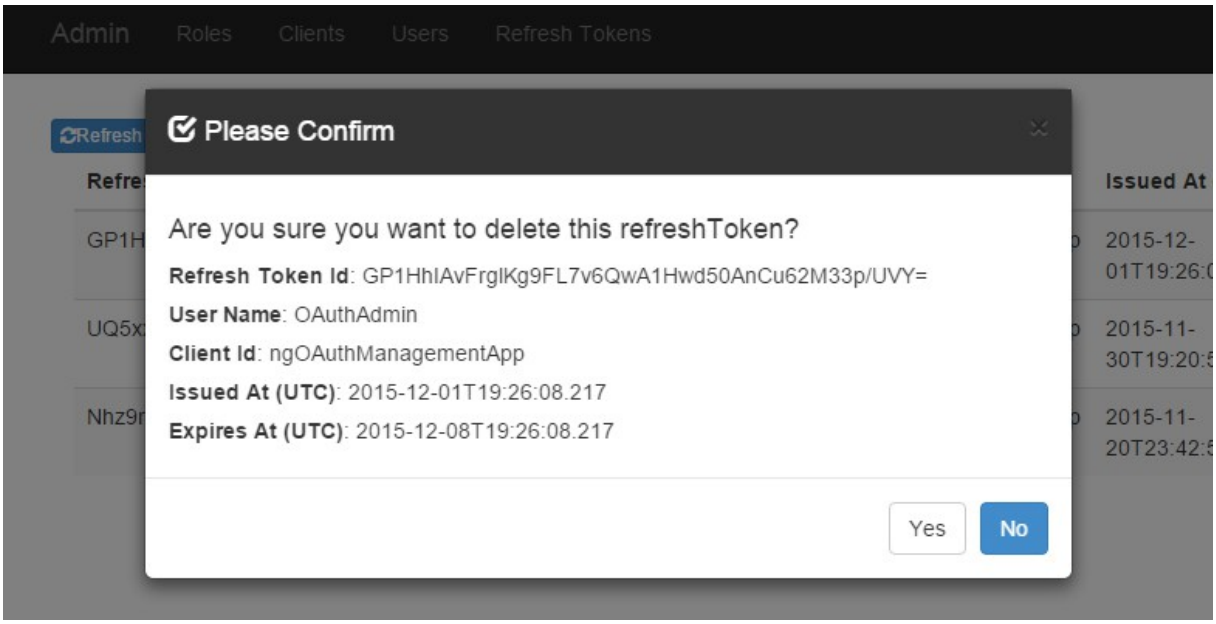
Ventana 10. Añadir usuario (similar a editar usuario)



Ventana 11. Borrar usuario



Ventana 12. Tokens de refresco



Ventana 13. Borrar token de refresco

6 Conclusiones y trabajo futuro

Como conclusión de este documento podemos decir que hemos conseguido desarrollar un servidor de autorización completamente funcional, que junto con la herramienta de administración podríamos implantar en un sistema real.

También hemos conseguido modularizar muchas funcionalidades mediante el uso de paquetes *NuGet*, los cuales podrían ser usados en cualquier otro proyecto tanto de .NET como de AngualrJS.

Aún así existen algunos puntos en los que se podría seguir trabajando:

Al servidor de autorización se le podría añadir al funcionalidad necesaria para permitir la autenticación con servidores OAuth externos, como Google, Facebook, Twitter, StackExchange, etc.

De este modo, nuestro servidor, además de actuar como servidor de autorización tendría el rol de cliente y utilizaría dichos servidores de autorización para obtener la información necesaria de los usuarios que quisieran acceder al sistema.

Para desarrollar el proyecto en este sentido puede ser bastante interesante ver el punto [16] de la bibliografía y como avance a continuación se muestra cómo quedaría el método `ConfigureOAuth` de la clase parcial `Startup`:

```
namespace OAuthServer
{
    public partial class Startup
    {
        public void ConfigureOAuth(IAppBuilder app,
                                   IDependencyResolver dependencyResolver)
        {
            ...

            // Token generation
            app.UseOAuthAuthorizationServer(OAuthServerOptions);
            // Token validation
            app.UseOAuthBearerAuthentication(OAuthBearerOptions);

            //Configure Google External Login
            googleAuthOptions = new GoogleOAuth2AuthenticationOptions()
            {
```

```
                ClientId = "xxxxxx",
                ClientSecret = "xxxxxx",
                Provider = new GoogleAuthProvider()
            };
            app.UseGoogleAuthentication(googleAuthOptions);

            //Configure Facebook External Login
            facebookAuthOptions = new FacebookAuthenticationOptions()
            {
                AppId = "xxxxxx",
                AppSecret = "xxxxxx",
                Provider = new FacebookAuthProvider()
            };
            app.UseFacebookAuthentication(facebookAuthOptions);
        }
    }
}
```

Código 6.1. Método `Startup.ConfigureOAuth()`

Con respecto a la SPA desarrollada, tal y como se explica en el punto 8.2.5 del apéndice B, los tokens se guardan en una cookie sin ningún tipo de seguridad asociada, por lo que sería muy interesante estudiar la manera de mejorar la gestión de los mismos.

Además también se podría añadir una página de registro de usuarios, que permitiera a los mismos crear su propia cuenta en el servidor.

7 Apéndice A. Resumen de tecnologías utilizadas

Para la implementación del servidor de autorización se han utilizado las siguientes tecnologías y protocolos:

- C#
- .NET Framework 4.5.1
- OAuth 2.0
- OWIN y OWIN OAuth 2.0
- ASP.NET Web API 2
- Entity Framework 6
- ASP.NET Identiy 2
- Autofac 3.5
- NuGet
- RESTful

La aplicación cliente se ha implementado utilizando las siguientes tecnologías:

- Javascript
- HTML5
- AngularJS v1.4.7
- CSS 3 y Bootstrap v3.0.0
- Grunt

Como principal herramienta de desarrollo se ha utilizado Microsoft Visual Studio Community 2013 y como herramienta de gestión de base de datos Microsoft SQL Server Management Studio 2012. Para la edición de esta memoria se ha usado Apache OpenOffice 4.

8 Apéndice B. Descripción de los paquetes NuGet implementados

NuGet es el gestor de paquetes para plataformas de desarrollo de Microsoft.

Los paquetes de NuGet pueden incluir ficheros de texto, código fuente o ficheros de ensamblado (dlls) entre otros y mediante su inclusión en un proyecto de Visual Studio se consigue aportar una funcionalidad concreta a dicho proyecto.

Pensando en la modularización y reutilización de código, algunas funcionalidades específicas desarrolladas en este proyecto se han extraído a módulos independientes y con estos módulos se han creado paquetes NuGet que podrán ser reutilizados en cualquier otro proyecto de .NET o AngularJS donde se necesite dicha funcionalidad.

8.1 Paquetes NuGet .NET

8.1.1 Jumuro.Security.Cryptography

Este paquete contiene la interfaz `IHashProvider` que contiene la firma de distintos métodos para general el hash de una cadena de caracteres usando diferentes algoritmos.

También contiene la clase `HashProvider` que implementa la interfaz. Para hacernos una idea a continuación se muestra el código de la interfaz:

```
using System.Security.Cryptography;
using System.Text;

namespace Jumuro.Security.Cryptography
{
    /// <summary>
    /// Represents a Hash provider.
    /// </summary>
    public interface IHashProvider
    {
        /// <summary>
        /// Computes the Hash for data using the received hash algorithm
        /// and <see cref="F:System.Text.Encoding.Default"/>.
        /// </summary>
        /// <param name="data"></param>
        /// <param name="algo"></param>
        /// <returns></returns>
        byte[] GetHash(string data, HashAlgorithm algo);
```

```
        /// <summary>
        /// Computes the Hash for data using the received hash algorithm
        /// and encoding.
        /// </summary>
        /// <param name="data"></param>
        /// <param name="algo"></param>
        /// <param name="encoding"></param>
        /// <returns></returns>
        byte[] GetHash(string data, HashAlgorithm algo, Encoding encoding);

        /// <summary>
        /// Computes the Hash for received data using SHA1 algorithm
        /// and the default Encoding.
        /// </summary>
        /// <param name="data"></param>
        /// <returns></returns>
        byte[] GetSHA1Hash(string data);

        /// <summary>
        /// Computes the Hash for received data using SHA1 algorithm
        /// and the received Encoding.
        /// </summary>
        /// <param name="data"></param>
        /// <param name="encoding"></param>
        /// <returns></returns>
        byte[] GetSHA1Hash(string data, Encoding encoding);

        /// <summary>
        /// Computes the Hash for received data using SHA256 algorithm
        /// and the default Encoding.
        /// </summary>
        /// <param name="data"></param>
        /// <returns></returns>
        byte[] GetSHA256Hash(string data);

        /// <summary>
        /// Computes the Hash for received data using SHA256 algorithm
        /// and the received Encoding.
        /// </summary>
        /// <param name="data"></param>
        /// <param name="encoding"></param>
        /// <returns></returns>
        byte[] GetSHA256Hash(string data, Encoding encoding);

        /// <summary>
        /// Computes the Hash for received data using MD5 algorithm
        /// and the default Encoding.
        /// </summary>
        /// <param name="data"></param>
        /// <returns></returns>
        byte[] GetMD5Hash(string data);

        /// <summary>
        /// Computes the Hash for received data using MD5 algorithm
        /// and the received Encoding.
```

```
        /// </summary>
        /// <param name="data"></param>
        /// <param name="encoding"></param>
        /// <returns></returns>
        byte[] GetMD5Hash(string data, Encoding encoding);
    }
}
```

Código 8.1. Interfaz `IHashProvider`

Además el paquete tiene una clase de extensiones, `CryptographyExtensions`, con dos métodos para transformar un array de bytes en array de caracteres, `ToHexString()` y `ToBase64String()`.

8.1.2 Jumuro.WebApi.Extensions.ActionResults

Este paquete contiene una clase de extensiones, `ApiControllerExtensions`, aplicada sobre objetos de tipo `ApiController` para generar objetos personalizados que implementan la interfaz `IhttpActionResult` que es lo que devuelven las acciones de nuestros controladores de Web API 2. Por ejemplo, entre otras se ha creado una sobrecarga del método `Ok<T>()` que genera una respuesta `Http` con `HttpStatusCode 200` (Ok), un objeto de tipo `T` en el *content* y además añade una cabecera con un mensaje de texto.

8.2 Paquetes NuGet AngularJS

8.2.1 Jumuro.Angular.CrudREST

Contiene una implementación de las llamadas CRUD (acrónimo de las operaciones básicas en inglés: *Create*, *Read*, *Update* y *Delete*) de Http:

- *Create*: `restPost()`
- *Read*: `restGet()`
- *Update*: `restPut()`
- *Delete*: `restDelete()`

8.2.2 Jumuro.Angular.Grid

Implementa una directiva AngularJS para crear un grid con bastante funcionalidad extra.

8.2.3 Jumuro.Angular.HttpInterceptor

Contiene un servicio AngularJS para procesar peticiones y respuestas Http.

8.2.4 Jumuro.Angular.Modal

Implementa un servicio AngularJS para mostrar ventanas modales de confirmación, notificación y error.

8.2.5 Jumuro.Angular.OAuth

Dada la relación directa con el centro de este proyecto, merece la pena que veamos éste paquete con más detalle.

Implementa dos servicios, el primero se llama `oAuthService` y se utiliza para gestionar la solicitud de tokens de acceso mediante el envío de las credenciales de usuario y mediante el envío del token de refresco. Tal y como se describió en el punto "3.3 *Token Endpoint.* Obtención del token de acceso".

Como vemos en el bloque de código 8.2, el método `logIn()` es el encargado de obtener el token de acceso mediante el envío de las credenciales de acceso. Hace una petición a la ruta `/token` de nuestro servidor enviando `grant_type=password` y las credenciales del usuario: `username`, `password` y `client_id`. Una respuesta correcta traerá consigo los tokens de acceso y de refresco, dicha respuesta se guardará en una cookie.

El método `refreshToken()` es el encargado de obtener el token de acceso mediante el envío del token de refresco. Hace una petición a la ruta `/token` de nuestro servidor enviando `grant_type=refresh_token`, el `refresh_token` y el `client_id`. Igual que antes, una respuesta correcta traerá consigo los tokens de acceso y de refresco que se guardarán en la cookie.

El método `logOut()` simplemente borra la cookie que contiene los tokens.

```
'use strict';

angular
    .module('jumuro.oAuth')
    .service('oAuthService', oAuthService);

oAuthService.$inject = ['$http', '$q', '$injector', 'ipCookie',
                        'oAuthConstants', 'oAuthAppConfigConstants',
```

```
                        '$location'];

function oAuthService($http, $q, $injector, ipCookie,
                      oAuthConstants, oAuthAppConfigConstants,
                      $location) {
    var service = {
        refreshToken: refreshToken,
        logOut: logOut,
        logIn: logIn,
        hasCookie: hasCookie,
        getUserInfo: getUserInfo,
        isAuthenticated: isAuthenticated
    };

    return service;

    function refreshToken() {
        var deferred = $q.defer();

        //get the cookie
        var authData = ipCookie(oAuthConstants.oAuthCookieName);

        if (authData) {
            var data = "grant_type=refresh_token&refresh_token=" +
                       authData.refresh_token + "&client_id=" +
                       authData.client_id;

            $http
                .post(oAuthAppConfigConstants.appConfig.oAuthURL, data, {
                    headers: {
                        'Content-Type': 'application/x-www-form-urlencoded'
                    }
                })
                .success(function (response) {
                    ipCookie(oAuthConstants.oAuthCookieName,
                             response, { path: oAuthConstants.appPathName });
                    deferred.resolve(response);
                })
                .error(function (err, status) {
                    logOut();
                    deferred.reject(err);
                });
        }

        return deferred.promise;
    };

    function logIn(postData) {
        var deferred = $q.defer();

        var data = "grant_type=password&username=" + postData.username +
                   "&password=" + postData.password + "&client_id=" +
                   oAuthAppConfigConstants.appConfig.oAuthClientId;

        $http
```

```
            .post(oAuthAppConfigConstants.appConfig.oAuthURL, data, {
                headers: { 'Content-Type': 'application/x-www-form-urlencoded' }
            })
            .success(function (data, status, headers, config) {
                // Create cookie
                ipCookie(oAuthConstants.oAuthCookieName,
                         data, { path: oAuthConstants.appPathName });
                // Store user in a local storage
                localStorage
                    .setItem("login-info",
                             JSON.stringify({ username: postData.username }));

                deferred.resolve(data);
            })
            .error(function (data, status, headers, config) {
                deferred.reject(data);
            });

        return deferred.promise;
    };

    function getUserInfo() {
        var userInfo = JSON.parse(localStorage.getItem("login-info"));

        return userInfo;
    }

    function hasCookie() {
        return ipCookie(oAuthConstants.oAuthCookieName);
    }

    function logOut() {
        // Delete current cookie if it already exsits
        ipCookie.remove(oAuthConstants.oAuthCookieName,
                        { path: oAuthConstants.appPathName });

        localStorage.removeItem("login-info");

        $location.path('/login');
    };

    function isAuthenticated() {
        var ok = ipCookie(oAuthConstants.oAuthCookieName);

        return ok;
    };
}
```

Código 8.2. Servicio `oAuthService`

El segundo servicio incluido en este paquete, `oAuthHttpInterceptor`, es un interceptor Http y se encarga de añadir el token de acceso a la cabecera de cada petición Http y de gestionar los

errores que tienen que ver con el proceso de autorización, como por ejemplo el Http Status Code 401, *Unauthorized*.

Como vemos en el bloque de código 8.3, el método `request()` lee los datos almacenados por el servicio anterior en la cookie y añade la cabecera de autorización `'Bearer [acceso_token]'` a la petición HTTP.

El método `responseError()` procesa una eventual respuesta HTTP errónea. El código de error que más nos interesa es el *401 – Unauthorized*. Previsiblemente, este error se producirá cuando el token de acceso haya expirado, en cuyo caso se leen los datos de la cookie y se realiza una llamada al método `refreshToken()` del servicio `oAuthService`. Además, si la petición de refresco del token es satisfactoria, se vuelve a realizar la llamada que originó el error 401, consiguiendo así que la renovación del token de acceso mediante el token de refresco sea totalmente transparente al usuario.

```
'use strict';

angular
    .module('jumuro.oAuth')
    .factory('oAuthHttpInterceptor', oAuthHttpInterceptor);

oAuthHttpInterceptor.$inject = ['$q', '$injector', 'ipCookie', 'oAuthConstants',
'toaster', '$location'];

function oAuthHttpInterceptor($q, $injector, ipCookie, oAuthConstants, toaster,
$location) {
    var service = {
        request: request,
        responseError: responseError
    };

    return service;

    function request(config) {
        if ($location.path() !== '/login') {
            config.headers = config.headers || {};

            //get the cookie
            var authData = ipCookie(oAuthConstants.oAuthCookieName);

            if (authData) {
                config.headers.Authorization = 'Bearer ' +
                                               authData.access_token;
            }
            else {
```

```
                var authService = $injector.get('oAuthService');
                authService.logOut();
            }
        }

        return config;
    }

    function responseError(rejection) {
        if (rejection.status === 400) {
            if (rejection.data && rejection.data.message) {
                toaster.pop('error', "Error", rejection.data.message);
            }
            else if (rejection.data && rejection.data.error) {
                if (rejection.data.error === 'invalid_grant') {
                    toaster.pop('error',
                                "Error", rejection.data.error_description);
                }
            }
        }
        else if (rejection.status === 401) {
            var authService = $injector.get('oAuthService');
            var authData = ipCookie(oAuthConstants.oAuthCookieName);
            var $http = $http || $injector.get('$http');
            var deferred = $q.defer();

            if (authData) {
                authService.refreshToken().then(function () {
                    //this repeats the request with the original parameters
                    return deferred.resolve($http(rejection.config));
                });
            }
            return deferred.promise;
        }
        else if (rejection.status === 403) {
            var toaster = $injector.get('toaster');
            toaster.pop('error',
                        "Access Denied",
                        "You are not authorized to do this request.");

            //window.location.path = oAuthConstants.appPathName;
        }
        else if (rejection.status === 0 || rejection.status === 500) {
            if (rejection.data && rejection.data.message) {
                toaster.pop('error', "Error", rejection.data.message);
            } else {
                toaster.pop('error', "Error", rejection.data);
            }
        }
        else {
            return $q.reject(rejection);
        }
    }
}
```

Código 8.3. Servicio `oAuthHttpInterceptor`

8.2.6 Jumuro.Angular.Spinner

Implementa una directiva para bloquear la ventana y mostrar un *spinner* durante las peticiones Http. Al realizar la petición se muestra el *spinner* y al recibir la respuesta se oculta.

8.2.7 Jumuro.Angular.Validations

Contiene varias directivas de validación personalizadas.

8.2.8 Jumuro.Angular.WebApi

Contiene dos clases de constantes, una donde la aplicación que incluya este paquete añadirá la/s URL/s de servidor y otra donde se añada la URL relativa a alguna de las URLs incluidas en la otra clase para cada operación del API de servidor.

9 Bibliografía

[1] OAuth 2.0:
http://oauth.net/2/

[2] The OAuth 2.0 Authorization Framework:
http://tools.ietf.org/html/rfc6749

[3] The OAuth 2.0 Authorization Framework: Bearer Token Usage:
http://tools.ietf.org/html/rfc6750

[4] Token Based Authentication using ASP.NET Web API 2, Owin, and Identity:
http://bitoftech.net/2014/06/01/token-based-authentication-asp-net-web-api-2-owin-asp-net-identity/

[5] OWIN and Katana:
http://owin.org/
http://www.asp.net/aspnet/overview/owin-and-katana

[6] OWIN OAuth 2.0 Authorization Server:
http://www.asp.net/aspnet/overview/owin-and-katana/owin-oauth-20-authorization-server

[7] ASP.NET Web API 2:
http://www.asp.net/web-api

[8] Entity Framework:
http://www.asp.net/entity-framework

[9] ASP.NET Identity:
http://www.asp.net/identity

[10] Depency Injection pattern:
http://martinfowler.com/articles/injection.html
http://oscarsotorrio.com/post/2013/02/18/Contenedores-de-inversion-de-control-y-el-patron-inyeccion-de-dependencias.aspx
http://www.variablenotfound.com/2013/02/desacoplando-controladores-aspnet-mvc.html

[11] Autofac
http://autofac.org/

[12] AngularJS:
https://angularjs.org/
https://github.com/johnpapa/angular-styleguide

[13] Bootstrap:
http://getbootstrap.com/

[14] Grunt:
http://gruntjs.com/

[15] Oauth2 with Angular: The right way:
http://jeremymarc.github.io/2014/08/14/oauth2-with-angular-the-right-way/

[16] OAuth for ASP.NET:
http://www.oauthforaspnet.com/

[17] HTTP Authentication: Basic and Digest Access Authentication
http://tools.ietf.org/html/rfc2617

[18] Visual Studio Online:
https://www.visualstudio.com/

[19] RESTful:
https://en.wikipedia.org/wiki/Representational_state_transfer