UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

TRABAJO FIN DE GRADO

Implementación de un proveedor de autorizaciones OAuth 2.0 con Scala

COSITI_96/1314

Álvaro García Delgado

Tutor: Juan José Vázquez Delgado

Ponente: Pablo Varona Martínez

Mayo 2014

Resumen

Implementación de un proveedor de autorizaciones OAuth 2.0 con Scala

En los últimos años se ha podido apreciar una convergencia en el desarrollo de nuevas aplicaciones que tienden a desarrollarse siguiendo la **arquitectura "cloud"**, una alternativa cuyo uso se está expandiendo enormemente gracias a la gran **flexibilidad** que ofrece tanto en el ámbito **económico** como en el **computacional.**

Este tipo de sistemas se construyen mediante una serie de servicios o aplicaciones que hacen un uso exhaustivo de un conjunto base de **APIs**. De este modo la correcta integración de todos esos elementos requiere un mecanismo con un lenguaje independiente a través del se coordina el flujo de información entre todos los elementos involucrados.

Es aquí donde **OAuth** jugará un papel de gran importancia. OAuth es una tecnología **emergente** en el contexto de la **seguridad en la nube**, que cubre los principales requisitos de seguridad aportando **confidencialidad**, **integridad** y **autenticación** a la hora de mantener una **comunicación con la API** y acceder a los recursos que esta ofrece.

OAuth 2.0 es la especificación de un marco de trabajo para la gestión de **autorizaciones** que permite a una **aplicación de terceros** obtener un **permiso** de acceso **limitado** a un **servicio HTTP**. Dicha concesión de acceso puede realizarse en nombre del recurso orquestando un proceso de aprobación entre éste y el servicio HTTP, o bien permitiendo a la aplicación obtener acceso **en su propio nombre**.

El proyecto se basará en el estudio minucioso de la **especificación del estándar** de **OAuth 2.0** y en el aprendizaje del lenguaje de programación **Scala**, entre otras herramientas, de forma que se cumplan los requisitos para poder empezar el proceso de desarrollo del software.

Este documento describe cómo se ha llevado a cabo el **estudio**, **análisis**, **diseño**, **implementación** y **validación** de un proveedor de autorizaciones **OAuth 2.0** en **Scala** para su posterior integración en un entorno real.

Palabras clave: autorización, autenticación, control de acceso, autorización delegada, OAuth, OAuth 2.0, seguridad, API, aplicaciones de terceros, computación en la nube, Scala

Abstract

An OAuth 2.0 Authorization Provider implementation written in Scala

In recent years there has been a convergence in application according the specifics of the so-called **cloud architecture**, an alternative whose use is widely expanding thanks to the big **flexibility** offered in **economic** and **computational** fields.

These systems make extensive use of certain **APIs** and the proper integration of the involved **services** and/or **applications** calls for an **independent language**.

Here is where **OAuth** will play its role. OAuth is an **emerging cloud security tecnology** that covers the main security requirements, providing **confidenciality**, **integrity** and **authentication** when **communicating** with an **API** and accessing shared resources.

OAuth 2.0 is an **authorization framework** specification that enables a **third-party application** to obtain **limited access** to an **HTTP service**, either on behalf of **a resource owner** by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access **on its own behalf**.

The project is based on the meticulous research of the **OAuth 2.0 standard specification** by means of **Scala** programming language. In specific, these elements are used to fulfill the basic requirements in the general concern of software development in the current technological scenario.

This document describes an **OAuth 2.0** authorization provider **research**, **analysis**, **design**, **implementation** and **validation** process using **Scala** so it can be integrated in a real environment.

Keywords: authorization, authentication, access control, delegated authorization, OAuth, OAuth 2.0, security, API, third-party applications, cloud computing, Scala

AGRADECIMIENTOS

Después de haber finalizado mi Trabajo de Fin de Grado,
quiero dar las gracias a aquellos que lo han hecho posible:

A Tecsisa, gracias por proponerme este proyecto y permitirme desarrollarlo.

A David, gracias por haberme ayudado y guiado tanto de forma desinteresada.

Y sobre todo a mis amigos y a mi familia, gracias por vuestro gran apoyo constante.

GLOSARIO

ACTOR (SCALA)	Entidades independientes sin estado que se comunican mediante mensajes sobre los que aplican cierta lógica para generar una respuesta.
API	*Application Programming Interface*. Capa de abstracción en la que una biblioteca recoge toda su funcionalidad para ser utilizada por otro software.
AUTENTICACIÓN	Es el proceso de verificar la identidad de un usuario, es decir, comprobar que es quien dice ser.
AUTENTICACIÓN FEDERADA	Solución por la cual los individuos pueden emplear la misma identificación personal para identificarse en diferentes redes.
AUTHORIZER	Entidad destinada a recoger la autorización de un cliente por parte de un usuario y comunicársela al mismo cliente.
AUTORIZACIÓN	Es el proceso de verificar que a un usuario se le permite llevar a cabo una acción determinada. Comúnmente requiere una identificación (autenticación) previa.
AUTORIZACIÓN DELEGADA	Permitir el acceso a otra persona o aplicación para realizar acciones en tu nombre.
BEARER TOKEN	Un token con la propiedad de que cualquiera que lo posea ("bearer" = portador) puede usarlo de la misma forma que cualquier otro portador.
CLIENTE (OAUTH)	Es la aplicación que quiere acceder al recurso en nombre del usuario.
CLIENTE CONFIDENCIAL	Es la aplicación cliente que es capaz de mantener sus credenciales a salvo a la hora de comunicarse con el servidor de autorización o del recurso (lógica ejecutada en un servidor web).
CLIENTE PÚBLICO	Es la aplicación cliente que no es capaz de mantener sus credenciales a salvo cuando se comunica con el servidor de autorización o del recurso (lógica ejecutada en el navegador, dispositivo móvil, etc.).
CÓDIGO DE AUTORIZACIÓN	Una cadena de caracteres que representa la autorización del cliente por parte del usuario para obtener un token de acceso, que será usado para acceder a sus datos.
CONFIDENCIALIDAD	Propiedad que asegura el acceso a la información únicamente a aquellas personas que cuenten con la debida autorización.

DATA HANDLER	Entidad destinada a gestionar el almacenamiento y recuperación de los datos.
DIRECTIVA (SPRAY)	Bloques de código que correctamente estructurados o anidados permiten construir rutas complejas.
FLUJO DE AUTORIZACIÓN	*Authorization Flow*. Cada uno de los métodos a elegir que constituyen un proceso en que el usuario concede autorización a una aplicación cliente.
FLUJO DE TRABAJO	*Workflow.* Secuencia de operaciones orquestadas y repetibles que constituyen un proceso.
GRANTER	Entidad destinada a conceder un token de acceso a un cliente si su petición se considera válida.
INTEGRIDAD	Propiedad que busca mantener los datos libres de modificaciones no autorizadas.
OBJECT (SCALA)	Es una clase que tiene una única instancia (singleton) y es visible desde cualquier parte del código.
PUNTO DE AUTORIZACIÓN	*Authorize Endpoint*. Representa el punto con el que el usuario puede interactuar para conceder una serie de permisos a un cliente determinado.
PUNTO DE OBTENCIÓN DE TOKEN	*Token Endpoint*. Representa el punto con el que el cliente puede interactuar para obtener o actualizar un token de acceso.
PUNTO FINAL	*Endpoint*. El punto de entrada para un servicio o proceso, representado normalmente por una URL.
RECURSO PROTEGIDO	Acción o datos (imagen, documento, etc.) accesible únicamente si se poseen los permisos requeridos.
RFC	*Request for Comments*. Son una serie de publicaciones del IETF (Internet Engineering Task Force) que describen diversos aspectos del funcionamiento de Internet y otras redes, como protocolos, procedimientos, etc.
RUTA (SPRAY)	Mecanismo de Spray que permite crear un punto final de lógica compleja a base de directivas.
SCOPE	Conjunto de permisos asociados a un token, usuario o cliente.
SERVIDOR DE AUTORIZACIÓN	Es el sistema con el que ha de comunicarse el cliente para que el usuario le autorice el acceso a los recursos. El resultado de autorizar el acceso se representa con el token de acceso.
SERVIDOR DEL RECURSO	Es el sistema que posee los recursos protegidos, permitiendo el acceso únicamente si se proporciona un token de acceso válido con la solicitud.

SGSI	*Sistema de gestión de la seguridad de la información.* Constituye el diseño, implantación y mantenimiento de un conjunto de procesos para gestionar la accesibilidad a la información asegurando la integridad, confidencialidad y disponibilidad de los activos de información.
SINGLE SIGN-ON	Procedimiento de autenticación que habilita al usuario para acceder a varios sistemas con una sola instancia de identificación (Autenticación Federada).
SPRINT (SCRUM)	Bloque de tiempo de un mes o menos durante el cual se crea un incremento de producto "terminado", utilizable y potencialmente desplegable.
SSL	*Secure Sockets Layer*. Protocolo criptográfico que proporciona comunicaciones seguras por una red, comúnmente Internet. Redactado en el RFC 6101. http://tools.ietf.org/html/rfc6101
TLS	*Transport Layer Security*. Protocolo criptográfico sucesor de SSL. Redactado en el RFC 5246. http://tools.ietf.org/html/rfc5246
TOKEN DE ACCESO	Concedido por el servidor de autorización permite que más tarde el servidor del recurso pueda identificar quién está intentando acceder al recurso y en nombre de quién.
TOKEN DE REFRESCO	Concedido por el servidor de autorización, permite renovar el token de acceso.
TRAIT (SCALA)	Similar a una clase abstracta de Java que permite herencia múltiple.
URI	*Uniform resource identifier*. Cadena de caracteres usada para identificar un recurso web por nombre, localización, o ambos.
USUARIO (OAUTH)	Es el usuario que hace uso del recurso de forma convencional.
UUID (JAVA)	*Universally unique identifier*. Tipo de identificador estándar.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. INTRODUCCIÓN

Contenido

1.1. MOTIVACIÓN

En los últimos años se ha podido apreciar una convergencia en el desarrollo de nuevas aplicaciones que según **Gartner** queda resumida en cuatro fuerzas decisivas en el éxito de un sistema, lo que se ha denominado **Nexus of Forces**: **Social**, **Mobile**, **Cloud** e **Information** [1].

Figura 1. Nexus of Forces (Gartner)

La **arquitectura "cloud"** en la que se basan estos sistemas es una alternativa cuyo uso se está expandiendo enormemente gracias a la gran **flexibilidad** que ofrece tanto en el ámbito **económico** como en el **computacional**, ya que permite **redimensionar** de forma eficiente los **recursos** para adaptarlos a sus necesidades y por tanto **controlar su coste**.

Este tipo de sistemas hacen un uso exhaustivo de las **APIs**, ya que los recursos que se desplieguen sobre esta arquitectura se mostrarán de cara al exterior como un **servicio** más. Es por ello que su correcta implantación precisa de un mecanismo con un lenguaje independiente por medio del cual se codificarán todas las comunicaciones entre los **servicios o aplicaciones puedan implicados** en el sistema.

Es aquí donde **OAuth** jugará un papel más que relevante en el actual panorama de las **Tecnologías de la Información y de las Comunicaciones**. En primer lugar efectúa un **control de acceso a activos de información** mediante la gestión de

políticas de **autorización**, y además cubre los **principales requisitos** que han de satisfacer los **Sistemas de Gestión de la Seguridad de la Información (SGSI)**. Así pues, mediante un uso adecuado de las primitivas criptográficas pertinentes y del uso del protocolo **SSL**, OAuth aporta **confidencialidad**, **integridad** y **autenticación** a la hora de mantener una **comunicación** con la **API** y acceder a los recursos que ésta ofrece.

Figura 2. Logo OAuth 2.0

OAuth 2.0 es un protocolo abierto que permite la **autenticación** y **autorización** contra una **API** por parte de aplicaciones web y de escritorio de forma segura mediante un método estándar [2].

Nació como respuesta a la situación en la que un usuario desea **permitir el acceso** de una aplicación **a sus datos** sin tener que compartir sus **credenciales**. En la actualidad, no obstante, actualmente cubre más casos, en su mayoría relacionados con interacciones en redes sociales y control de acceso a documentos.

Es por tanto conocido por su característica de poder llevar a cabo una **Autorización Delegada**, aunque también se utiliza como método de **Autenticación Federada** en armonía con su extensión **OpenID Connect** [3].

OAuth es una tecnología **emergente** en el contexto de la **seguridad en la nube** que cuenta con el apoyo de grandes organizaciones como Facebook o Google y está siendo adoptada en multitud de sistemas. En la **Figura 3** se puede ver el estado de OAuth frente a otras tecnologías relacionadas con la seguridad en la nube.

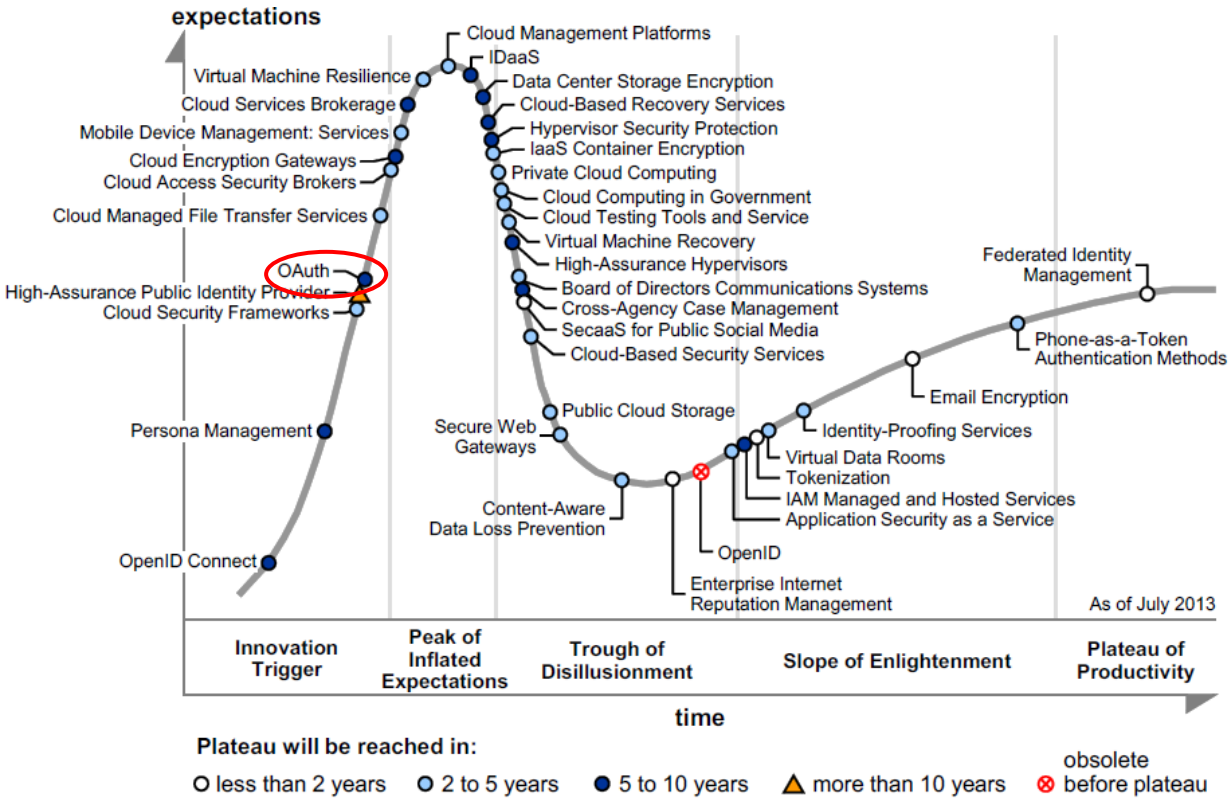


Figura 3. Hype Cycle: Cloud Security (Gartner, 2013)

Esta gráfica sitúa a **OAuth** en la fase de innovación, justo antes de llegar al pico de expectativas sobredimensionadas que sería el momento en el que la tecnología está "de moda" porque se piensa que es la solución a todos los problemas.

La siguiente etapa representa el **abismo de la desilusión**, después del cual empieza a consolidarse con unas expectativas más realistas hasta convertirse en una práctica común. Este proceso no tiene por qué completarse siempre, pues por ejemplo en el caso de **OpenID** se puede observar como con la aparición de **OpenID Connect** (y por lo tanto OAuth) queda **obsoleto**.

La especificación de OAuth no establece limitaciones en cómo implementarlo, por lo que se puede diseñar fácilmente de forma que pueda ser **extensible**, **escalable** y adecuado para integrarse con un sistema distribuido en la **nube**.

Figura 4. Logo Scala

Se ha escogido el lenguaje de programación **Scala** para llevar a cabo la implementación ya que como lenguaje funcional garantiza que una operación, para unas entradas determinadas, siempre devolverá el **mismo resultado**, lo que permite **paralelizar** esta misma operación al no depender de un **estado**. Además de poder llevar a cabo las operaciones de forma **concurrente** o **distribuida** por su carácter funcional, también está diseñado para **aprovechar el potencial** de esta técnica, sobre todo haciendo uso de una de las herramientas más potentes de este lenguaje: los **actores**, entidades independientes **sin estado** que se comunican mediante **mensajes** sobre los que aplican cierta lógica para generar una **respuesta**.

El proyecto vinculado a este Trabajo de Fin de Grado tendrá por eje central la **implementación de un proveedor de autorizaciones OAuth 2.0**, el cual se ajustará a la especificación del **estándar** denfinido en [4] y será desarrollado en **Scala**.

Entre los motivos que justifican la realización de este proyecto se encuentran:

- **Necesidad en un proyecto real**: la empresa en la que me encuentro desempeñando las prácticas ha iniciado un proyecto que está utilizando tecnologías recientes con una arquitectura basada en la nube que hacen de este un contexto idóneo para el uso de OAuth 2.0.

- **Reciente especificación del estándar**: lo que hace que sea un buen momento para llevar a cabo una propuesta de implementación definitiva que se ajustará a lo que realmente se debería usar.

- **Implementaciones existentes pobres**: sobre todo en **Scala**, un lenguaje de programación optimizado para este tipo de servicios en la nube, principalmente a causa de lo expuesto en el punto anterior.

- **Interés personal por un tema tan actual**: ya que tiene una relación directa con el desarrollo de aplicaciones que hacen uso de redes sociales como Facebook, Instagram, Twitter, Google, etc. y otras herramientas como Dropbox o Github.

1.2. OBJETIVOS

Los objetivos principales que entran dentro del alcance de este proyecto son:

- **Estudio detallado del estándar OAuth 2.0**: antes de empezar a desarrollar habrá sido necesario estudiar y comprender cómo funciona OAuth, de qué elementos se compone, qué soluciones propone, etc.
- **Familiarización con el lenguaje Scala y otras herramientas:** hasta alcanzar un nivel apropiado para poder aprovechar las características que éstas ofrecen.
- **Estudio y diseño de la implementación OAuth 2.0:** una vez se tenga el conocimiento necesario del lenguaje Scala y de OAuth, se podrá proceder a realizar un diseño para su posterior implementación.
- **Implementación de OAuth 2.0 en Scala**
- **Desarrollo de aplicaciones de ejemplo:** que muestren la forma de integración, uso y funcionamiento de este sistema de autorización.

Las principales herramientas seleccionadas para llevar a cabo este proyecto son las siguientes:

- **Lenguaje de programación Scala[1] (Scala IDE for Eclipse):** las propiedades que posee Scala como lenguaje funcional hacen que sea una muy buena decisión para implementar un sistema de autorización que está pensado para componentes distribuidos.
- **Scala Simple Build Tool[2]**: que se encargará de resolver las dependencias, generar un proyecto de Eclipse, compilar el código y realizar los tests implementados.
- **Spray Toolkit[3]:** como solución a la necesidad de una librería para crear servicios HTTP, que hace uso del sistema de actores de **Akka**[4]. Esta implementación de OAuth 2.0 estará dirigida a este marco de trabajo concreto, aunque se podrá transportar fácilmente a otro contexto.
- **Play Framework for Scala[5]:** que se utilizará para implementar una aplicación web cliente de prueba.
- **JQuery[6]:** se utilizará en los clientes de prueba para implementar la lógica ejecutada en el navegador del usuario.

[1] http://www.scala-lang.org
[2] http://www.scala-sbt.org
[3] http://spray.io
[4] http://akka.io
[5] http://www.playframework.com
[6] http://jquery.com

1.3. PLANIFICACIÓN DEL PROYECTO

El proyecto para el que se ha desarrollado esta implementación sigue el **marco de gestión y desarrollo** de software conocido como **Scrum** [5], basado en un proceso **iterativo** e **incremental** orientado al **desarrollo ágil** de software. Por este motivo el desarrollo del proyecto de **OAuth 2.0** se ha adaptado a esta técnica, dividiendo su implementación en **varios sprints**, bloques de tiempo inferiores a un mes donde al final de cada uno se intenta **llegar a un producto "utilizable" con la funcionalidad** esperada.

Se puede consultar el listado de tareas y sprints en los que se ha dividido el proyecto, sus fechas de comienzo y fin estimadas y sus dependencias en el **Diagrama de Gantt** que figura en el **Anexo** del presente documento.

1.4. ORGANIZACIÓN DEL DOCUMENTO

El documento estará dividido en los siguientes apartados:

- **Estado del arte:** consiste en una breve explicación de varios métodos de autorización delegada que existían antes de OAuth y que sirvieron de inspiración a la hora de crear el estándar.
- **OAuth 2.0:** una explicación más detallada de lo que hace OAuth 2.0, mostrando los diferentes flujos propuestos que soportará la implementación de una forma sencilla e ilustrativa.
- **Análisis de requisitos:** los requisitos del proyecto que tendrán que cumplir las implementaciones de los flujos de autorización del apartado previo, explicados de una forma más concreta.
- **Diseño:** la solución propuesta para desarrollar el proyecto que cumpla los requisitos anteriores, enfocada a su implementación en Scala.
- **Implementación:** la explicación de las clases resultantes haciendo hincapié en la problemática encontrada.
- **Integración y uso:** una descripción de cómo integrar la solución a un servicio que desee hacer uso de OAuth 2.0 como proveedor de autorizaciones.
- **Pruebas y ejemplos:** las pruebas llevadas a cabo para demostrar el cumplimiento de los requisitos y buen funcionamiento de la solución. También se explicarán los ejemplos desarrollados.
- **Conclusión:** una conclusión sobre OAuth 2.0, su implementación y el proyecto en general.

2. ESTADO DEL ARTE

Contenido

El surgimiento de las **estrategias** de desarrollo de aplicaciones **"cloud computing"** ha hecho que sea necesario **replantear la forma de gestionar las identidades** de los usuarios con el fin de **evitar** cualquier **limitación** que pueda perjudicar la filosofía de estos sistemas. Esto se ha traducido en un soplo de aire **fresco** en las tecnologías de **Single Sign-On** e **Identidad Federada** (OpenID, SAML, etc.), ya que siguen unas normas que las convierten en idóneas para su integración en un sistema en la nube.

Estas tecnologías permiten que un **usuario** pueda acceder a **distintos servicios** auténticandose con las **mismas credenciales**. Esto se consigue centralizando la gestión de identidades en un **Proveedor de Identidades**, de forma que varios servicios que **confíen** en este proveedor puedan acudir a este punto para obtener información de los usuarios que quieran acceder a los mismos.

Una de las claves de la identidad federada reside en que la **autenticación** se **abstrae** de la **autorización:** autorización y autenticación se articulan de modo solidario a través de dominios de implantación disjuntos, esto es, el contexto de autorización implica de modo implícito la autenticación, pero no incorpora de modo explícito su implementación [6]. Por este motivo **no se cubre la autorización** en el contexto en el que se autenticó el usuario.

OAuth nace para solucionar este problema, pues es un estándar libre que permite realizar el proceso de **autorización** de forma **segura**. Con OAuth, un usuario puede permitir a un **sitio A** que acceda a su **información** en el **sitio B** sin tener que prestarle sus credenciales al **sitio A**. Antes de que se redactara la especificación del estándar, muchos sitios web de la industria eran conscientes del problema y de la necesidad de abordarlo, por lo que **diseñaron** e **implementaron** sus **propios protocolos.** En la medida que dichos protocolos representan el estudio preliminar que condujo a **OAuth 1.0** [7], y finalmente a **OAuth 2.0**.

A continuación se procede a reseñar algunos de los mismos con objeto de subrayar los elementos finales constituyentes de la versión más reciente del estándar OAuth.

2.1. AUTHSUB

AuthSub [8] es un proceso de autorización desarrollado por **Google** para aplicaciones web que tuvieran la necesidad de acceder a **servicios protegidos** por un usuario de Google. Este proceso permite a la aplicación obtener acceso al servicio sin llegar a manejar las credenciales del usuario. Actualmente sigue en uso pero se recomienda abandonarlo y migrar a **OAuth 2.0**.

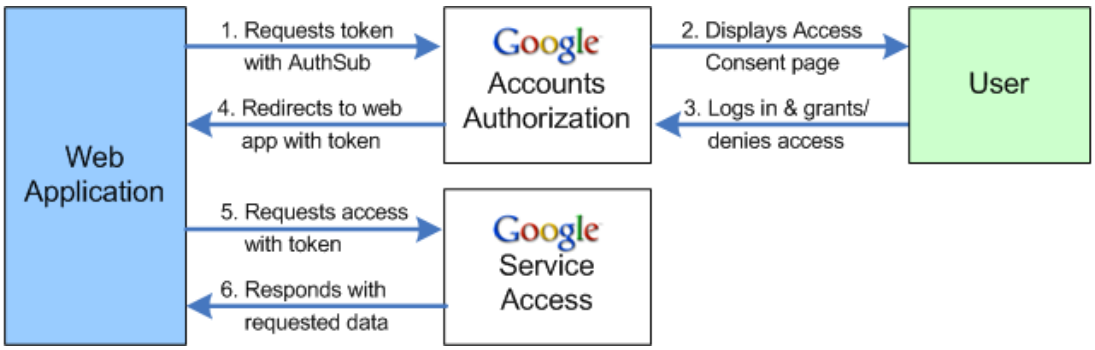


Figura 5. Proceso de Autorización AuthSub

En la **Figura 5** se puede observar el proceso de autorización seguido, que posee los siguientes pasos:

1. Cuando la aplicación necesita acceder a un servicio de un usuario de Google, hace una llamada utilizando AuthSub al **Servicio de Autorización de Google**.

2. El Servicio de Autorización responde mostrando una página de Google donde se le notifica de la **petición de acceso a sus servicios**. Para ello el usuario ha de estar **autenticado**.

3. El usuario **acepta o rechaza la petición** y se le redirige a la aplicación web inicial o a Google, según su decisión.

4. En la redirección a la aplicación web inicial se incluye un **token de autorización** de un solo uso, que se podrá cambiar por un token de más duración.

5. La aplicación web **adjunta el token** recibido al enviar la **petición al servicio de Google**, para poder actuar en nombre del usuario.

6. Si el servicio de Google reconoce el token, **devuelve la información** solicitada.

2.2. BBAUTH

Browser-Based Authentication (BBAuth) [9] es un proceso que permite a las aplicaciones pedir permiso a los usuarios de Yahoo! para poder acceder a sus datos.

También implementa un proceso de **Single Sign-On** para poder autenticar a los usuarios en aplicaciones externas. Aunque Yahoo! propone el uso de este método, también da soporte a la especificación estándar de OAuth 1.0.

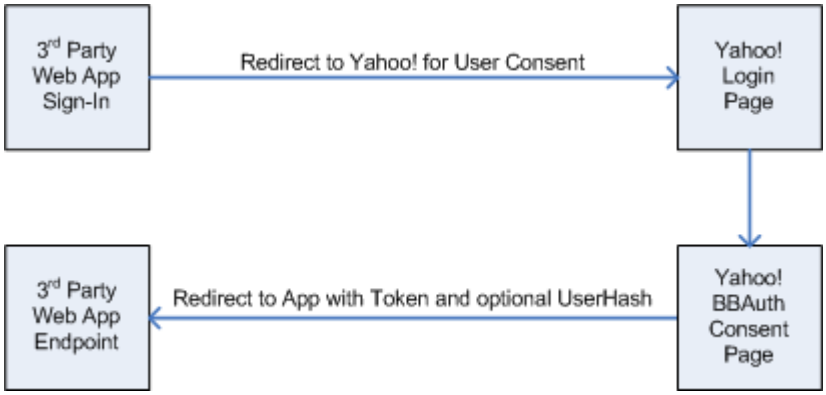


Figura 6. Proceso BBAuth

En la **Figura 6** se puede observar el flujo que implementa BBAuth:

1. Tras haber registrado la aplicación de terceros en Yahoo!, se **redirige al usuario** a una URL de Yahoo! específica.
2. Si el usuario no se había autenticado en Yahoo!, lo hace y pasa a una página donde **puede elegir si conceder los permisos a la primera aplicación**.
3. Se **redirige a la aplicación inicial** incluyendo los datos necesarios para que ésta **pueda acceder a la información que solicitaba** desde el principio.

2.3. OAUTH 1.0

OAuth 1.0 [7] es un protocolo abierto que permite **autorización segura de una API** de modo estándar y simple para aplicaciones de escritorio, web y móvil. Como los anteriores métodos de autorización, permite a una aplicación de terceros acceder al recurso protegido de un usuario, con su permiso.

Debido a problemas de seguridad, se trabajó en una **versión 1.0a** no estándar que soluciona estos problemas, que actualmente utiliza por ejemplo **Twitter**, aunque su uso ha decaído debido a la aparición de **OAuth 2.0**. Los pasos de este proceso, representados en la **Figura 7**, se explican a continuación:

A. La aplicación solicita una **petición de token**.
B. El proveedor de servicios se la concede.
C. La aplicación dirige al **usuario** al proveedor de servicios, donde **se autentica**.
D. El **usuario permite a la aplicación acceder** al servicio protegido y es dirigido a la aplicación cliente.
E. La aplicación cliente, con la autorización del usuario, pide un **token de acceso**.

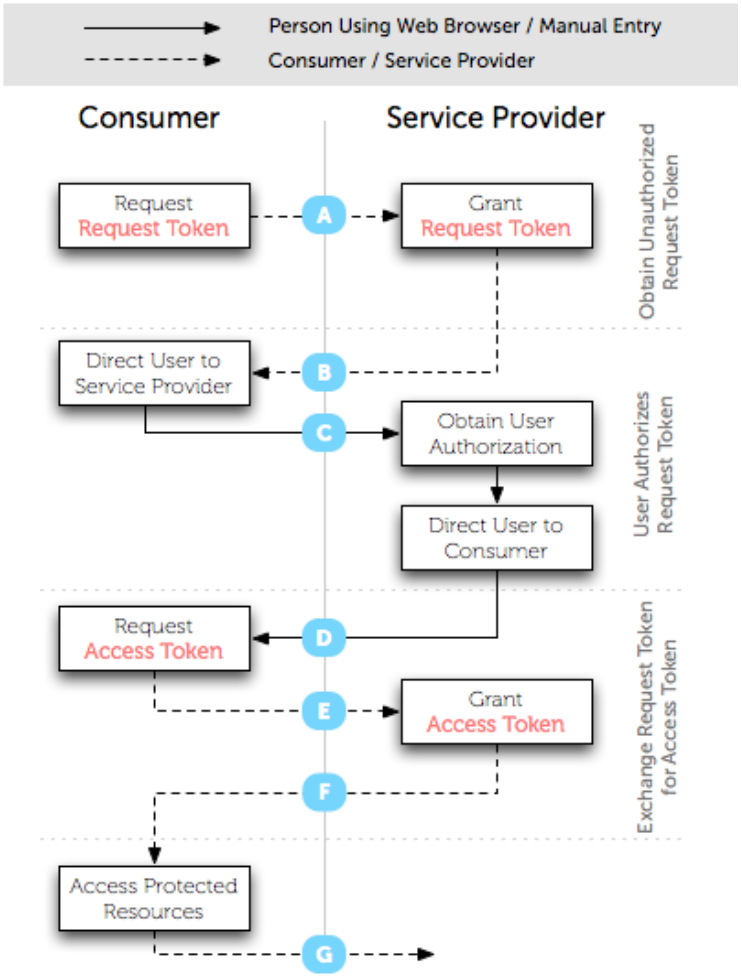


Figura 7. Proceso de autorización OAuth v1.0a

F. El proveedor de servicios le **proporciona el token de acceso** a la aplicación.

G. Ahora la aplicación puede **acceder a los servicios protegidos**.

2.4. OTROS

Cuando se inventó **OAuth,** en Internet convivían varios mecanismos para proporcionar una capa de seguridad a las **APIs**. Algunas de estas soluciones siguen utilizándose (las mencionadas anteriormente) aunque siempre proponen como alternativa el uso de OAuth.

Otras soluciones como **OpenAuth**, **FlickrAuth** y **FacebookAuth** se han retirado y se han sustituido por OAuth, pero todas ellas han contribuido a construir lo que es OAuth actualmente, como bien explican los autores en su página web, ya que tienen muchos puntos en común y que acabara apareciendo un método estándar era inminente, como se explica en [7].

OAuth 2.0 [10] aparece con el fin de reemplazar a OAuth 1.0 enfocándose en la simplificación de los flujos de autorización para aplicaciones. Se abordará de forma más detallada en el resto del documento.

3. OAUTH V2.0

Contenido

Antes de entrar en la descripción del trabajo realizado es necesario explicar en qué consiste OAuth 2.0.

Como se ha expuesto, su objetivo es **añadir una capa de seguridad** a los **servicios** que ofrece una **API**, a la vez que protege los **datos de los usuarios**, pudiendo un **usuario** conceder un **acceso limitado a aplicaciones** de terceros. OAuth propone proteger estos servicios y datos (a partir de ahora **recursos**) pidiendo un código (**token de acceso**) del que se puede deducir la identidad de **quién** accede y **en nombre de quién** accede.

Teniendo en cuenta la definición anterior es necesario definir los **elementos** que juegan un papel en el proceso:

Usuario	Es el usuario que hace uso del recurso de **forma convencional**.
Cliente	Es la aplicación que quiere acceder al recurso **en nombre del usuario**.
Servidor de autorización	Es el sistema con el que ha de comunicarse el **cliente** para que el **usuario** le autorice el acceso a los **recursos**. El resultado de autorizar el acceso se representa con el **token de acceso**.
Token de acceso	Concedido por el **servidor de autorización** como respuesta a una solicitud correcta del **cliente**, permite que más tarde el **servidor del recurso** pueda **identificar quién** está intentando acceder al recurso y en nombre de quién. Tiene un **tiempo de vida** específico.
Servidor del recurso	Es el sistema que posee los **recursos protegidos**, permitiendo el acceso únicamente si se proporciona un **token de acceso válido** con la solicitud.

OAuth 2.0 propone diferentes **flujos de trabajo**, entendiendo como flujo la **secuencia de operaciones a seguir** para obtener el token de acceso, en los cuales cambia la forma de la que interactúan los elementos anteriores. Estos flujos siempre se han de llevar a cabo bajo una comunicación segura, por ejemplo añadiendo **SSL** o **TLS**.

A continuación se explican los diferentes flujos de forma detallada.

3.1. AUTHORIZATION CODE GRANT

Mediante este flujo un usuario puede permitir el acceso de un cliente a sus recursos. Está pensado para **clientes confidenciales**, es decir, que son capaces de ocultar sus credenciales cuando se autentican en el servidor de autorización. En esta modalidad, como bien se muestra en la **Figura 8**, hay dos interacciones con el servidor de autorización.

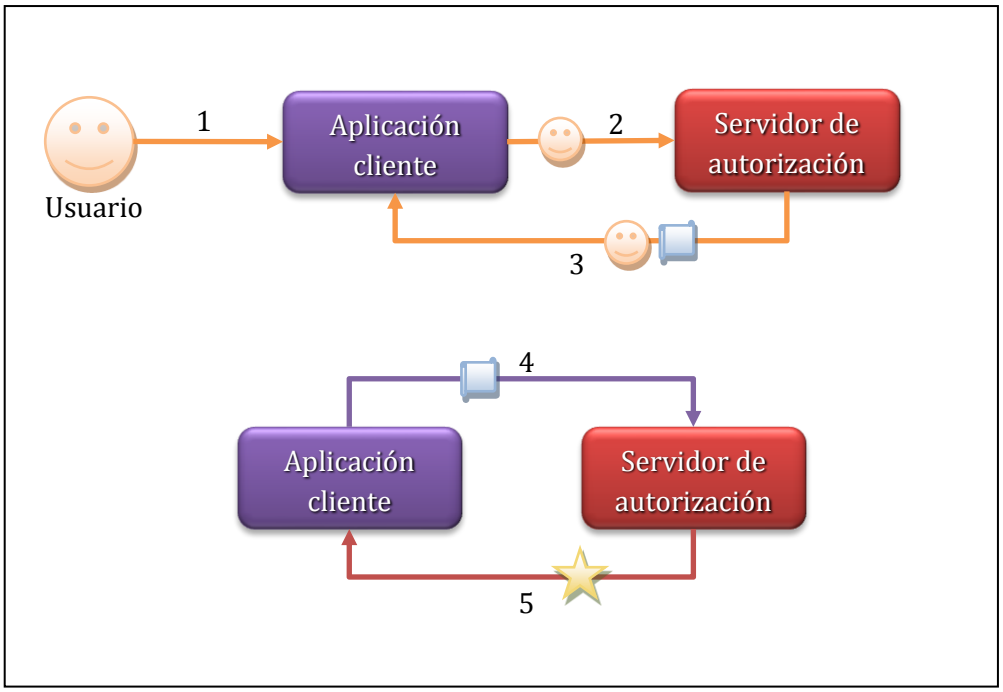


Figura 8. Authorization Code Grant

1. El **usuario** accede a la aplicación cliente que querrá acceder a sus datos.
2. La aplicación cliente envía al **usuario** al **punto final de autorización** con los parámetros adecuados. Si el usuario no estaba autenticado, lo hace.
3. El **usuario** elige si **autoriza** o no a la aplicación cliente. Si el usuario acepta, se le devuelve a la aplicación cliente con un **código de autorización**.
4. El **cliente** realiza una petición en el **punto final de solicitud de token**.
5. Si con la petición ha enviado un **código de autorización válido** se le devuelve un **token de acceso**.

3.2. IMPLICIT GRANT

Mediante este flujo un usuario puede permitir el acceso de un cliente a sus recursos. Está pensado para **clientes públicos**, es decir, que no serían capaces de ocultar sus credenciales si se autenticaran en el servidor de autorización.

En esta modalidad hay una sola interacción con el servidor de autorización, como se puede ver en la **Figura 9**.

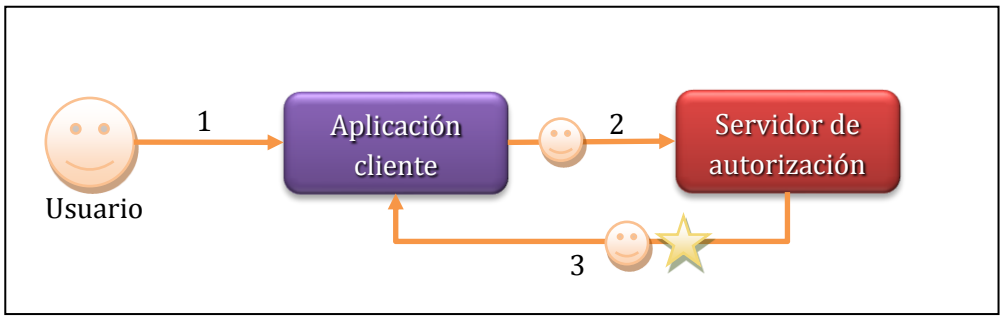


Figura 9. Implicit Grant

1. El **usuario** accede a la aplicación cliente que querrá acceder a sus datos.
2. La aplicación cliente envía al **usuario** al **punto final de autorización** con los parámetros adecuados. Si el usuario no estaba autenticado, lo inicia y ejecuta un proceso de autenticación.
3. El **usuario** elige si **autoriza** o no a la aplicación cliente. Si el usuario acepta, se le devuelve a la aplicación **cliente** con el **token de acceso**.

3.3. RESOURCE OWNER PASSWORD CREDENTIALS GRANT

Este flujo es adecuado para cuando el **usuario** tiene una **relación de confianza** con el **cliente**, pues le va a **prestar sus credenciales**, donde queda implícita la **autorización** por parte del usuario. También se utiliza para convertir las credenciales en un token de acceso cuando se piensa almacenar en un dispositivo, **evitando el robo** de las mismas.

En esta alternativa hay una sola interacción con el servidor de autorización, pero esta vez sólo por parte de la aplicación cliente. Queda representado en la **Figura 10**.

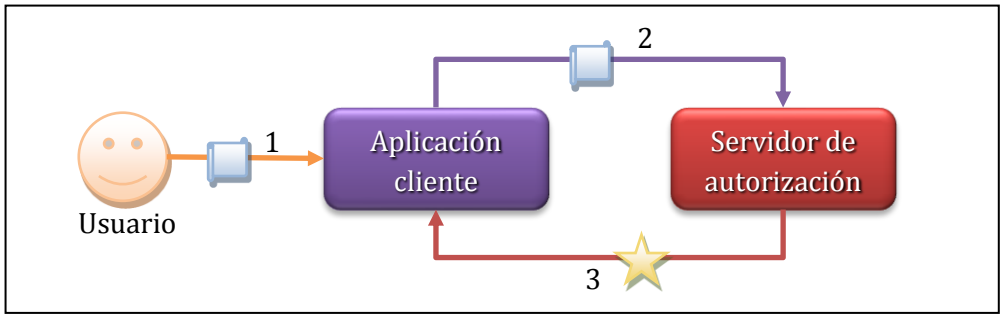


Figura 10. Resource Owner Password Credentials Grant

1. El **usuario** le proporciona sus **credenciales** a la aplicación cliente.
2. La aplicación **cliente** realiza la solicitud en el **servidor de autorización** proporcionando las **credenciales del usuario** en nombre del cual quiere operar.
3. Si las **credenciales del usuario** son válidas, se le devuelve el **token de acceso** al **cliente**.

3.4. CLIENT CREDENTIALS GRANT

Este flujo permite que el **cliente** con sus **credenciales** pueda solicitar un **token de acceso**. En este caso accedería **en su propio nombre** y no en el de otro usuario, pudiendo hacer uso de los servicios que le estén permitidos. Este sencillo proceso es el que se sigue en la **Figura 11**.

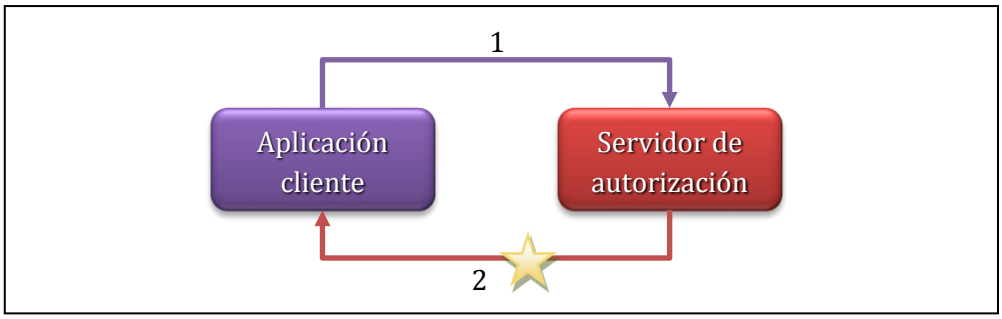


Figura 11. Client Credentials Grant

1. El **cliente** realiza una **solicitud en el servidor de autorización** utilizando sus propias **credenciales**.
2. Si las **credenciales** son **válidas**, el servidor de autorización le responde con un **token de acceso** que no estará asociado a un usuario.

3.5. ACTUALIZAR TOKEN DE ACCESO

Ya que el token de acceso tiene un tiempo de vida después del cual caduca, en varios de los flujos anteriores se da la opción de devolver un **token de refresco.** Con él se podrá obtener un nuevo token de acceso una vez haya caducado (**Figura 12**).

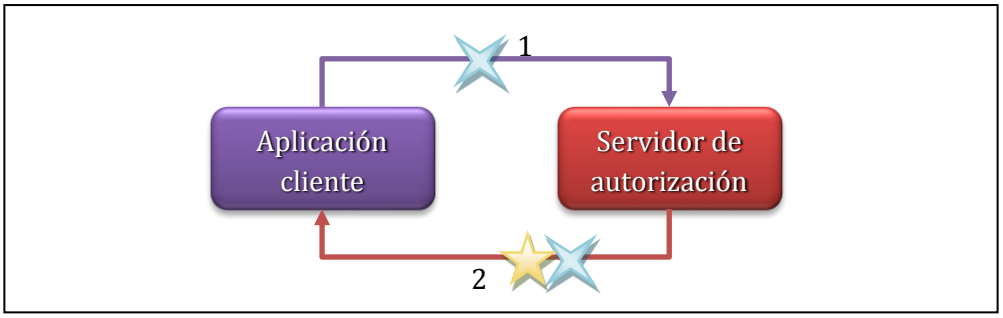


Figura 12. Actualizar token de acceso

1. El **cliente** realiza una **solicitud en el servidor de autorización** proporcionando el  **token de refresco**.

2. Si el  **token de refresco** es válido, se devuelve un  **token de acceso** y  **token de refresco** nuevos (este último de forma opcional).

3.6. ACCEDER AL RECURSO

Una vez se obtenga el **token de acceso** se podrá utilizar para acceder a un recurso proporcionándolo en la petición del mismo.

Este token de acceso poseerá un **conjunto de permisos** que representan los datos o tareas que puede realizar, lo que en OAuth se denomina **scope**. El cliente podrá, bien especificar una serie de **permisos al solicitar el token** o bien **no especificarlo**, con lo que se supondrá que se solicitan los **permisos por defecto**.

Si el cliente intenta acceder a un recurso con un token que **no tenga los permisos requeridos** por ese recurso, se le **denegará el acceso**.

4. ANÁLISIS DE REQUISITOS

Contenido

4.1. REQUISITOS FUNCIONALES

Ya que la funcionalidad queda dividida por las diferentes situaciones explicadas en el apartado anterior, se listarán los requisitos funcionales con una estructura similar.

4.1.1. REQUISITOS FUNCIONALES GLOBALES

RF1: El cliente podrá dirigir al usuario al **punto de autorización**, que se encontrará en una URL a la que se accederá con diferentes parámetros **GET** que representan la petición del cliente. Esa implementación ha de ser capaz de interpretar la URL y procesar sus parámetros. Los parámetros de la petición se describen en la **Tabla 1**.

Tabla 1. Entradas de punto de autorización

response_type	El tipo de respuesta esperada por OAuth. Se mostrarán las alternativas en siguientes requisitos.
client_id	El identificador del cliente.
[redirect_uri]	La dirección a la que se redirigirá al usuario después de decidir si autorizar o no al cliente. Si no se proporciona se obtendrá de los datos asociados al cliente.
[scope]	Permisos que desea obtener la aplicación. Si no se especifica se supondrán los permisos por defecto.
[state]	Un valor utilizado por el cliente para mantener el estado desde que se hace la petición hasta que se obtiene una respuesta. El servidor de autorización devolverá este mismo parámetro en la respuesta.

RF2: Si no ha habido ningún problema tras la anterior solicitud, el servidor será capaz de mostrar al usuario una **pantalla donde decidir si autorizar** o no al cliente.

RF3: El cliente podrá realizar una petición en el **punto de solicitud de token**, pudiendo interpretar y procesar los distintos parámetros que acompañarán a una petición **POST**. Estos parámetros cambiarán según el flujo a seguir.

RF4: Cuando el **cliente** interactúe con el servidor de autorización directamente, este tendrá que poder **autenticarse** con las **credenciales** que se le hayan asignado, mediante una de estas dos alternativas:

- Mediante **HTTP Basic Authentication**:

```
Authorization: Basic dGVzdDp0ZXN0
```

- Con los parámetros **client_id** y **client_secret** en el **cuerpo** de la **petición HTTP** (sólo para clientes que no puedan autenticarse de otra forma):

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

client_id=s6BhdRkqt3&client_secret=7Fjfp0ZBr1KtDRbnfVdmIw
```

RF5: El servidor de autorización será capaz de **crear y almacenar un token** de acceso si se le proporcionan los parámetros adecuados.

RF6: El servidor de autorización será capaz de generar una **respuesta con el token generado**, la cual estará compuesta por los siguientes campos:

Tabla 2. Respuesta con token de acceso

access_token	El token de acceso generado por el servidor.
token_type	El tipo del token proporcionado.
expires_in	El tiempo de vida del token de acceso, en segundos.
[refresh_token]	Token de refresco.
[scope]	Permisos concedidos por el servidor. Opcional si es idéntico al pedido por el cliente. Si no, requerido.
[state]	El mismo valor proporcionado por el cliente al realizar la petición en caso de que este parámetro estuviera presente.

RF7: En caso de error, el servidor de autorización será capaz de generar una respuesta de error con los campos de la **Tabla 3**.

Tabla 3. Respuesta con error

error	El código de error.
[error_description]	Información adicional explicando la causa del error.
[error_uri]	Una URI identificando una página web en la que se pueda ver la descripción del error.
[state]	El mismo valor proporcionado por el cliente al realizar la petición en caso de que este parámetro estuviera presente.

4.1.2. AUTHORIZATION CODE GRANT

RF7: El servidor de autorización soportará una petición del flujo "Authorization Code Grant" en el punto de autorización, que tendrá la estructura expuesta en la **Tabla 4**.

Tabla 4. Entradas de punto de autorización para Authorization Code Grant

response_type	Su valor en este caso será "**code**".
client_id	El identificador del cliente.
[redirect_uri]	La dirección a la que se redirigirá al usuario después de decidir si autorizar o no al cliente. Si no se proporciona se obtendrá de los datos asociados al cliente.
[scope]	Permisos que quiere obtener la aplicación. Si no se especifica se supondrán los permisos por defecto.
[state]	Un valor utilizado por el cliente para mantener el estado desde que se hace la petición hasta que se obtiene una respuesta. El servidor de autorización devolverá este mismo parámetro en la respuesta.

Por ejemplo (saltos de línea añadidos para facilitar comprensión):

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3
&state=xyz&redirect_uri=https://cliente.com/callback HTTP/1.1
Host: server.oauth.com
```

RF8: Si el usuario acepta la petición en la pantalla del **RF2**, se ha de poder crear un código de autorización asociado al mismo y al cliente que lo solicita.

RF9: Tras lo anterior, si no hay ningún problema, el sistema ha de ser capaz de redirigir al usuario a la URL especificada por el cliente, con los parámetros GET correspondientes.

Estos parámetros se detallan en la **Tabla 5**.

Tabla 5. Respuesta punto de autorización para Authorization Code Grant

code	El código de autorización generado por el servidor.
[state]	El mismo valor proporcionado por el cliente al realizar la petición en caso de que este parámetro estuviera presente.

Por ejemplo:

```
HTTP/1.1 302 Found
Location: https://cliente.com/callback
?code=SplxlOBeZQQYbYS6WxSbIA&state=xyz
```

RF10: En caso de error, el **usuario ha de ser informado** sin ser redirigido directamente a la aplicación cliente, ya que se le dará la opción de hacerlo manualmente, **añadiendo a la URL de redirección los parámetros** especificados en el **RF7**.

Por ejemplo, el servidor de autorización redirigiría al usuario con la siguiente respuesta HTTP:

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?error=access_denied
&state=xyz
```

RF11: El cliente accederá al **punto de solicitud de token**, donde, ya autenticado, podrá realizar una petición POST con los parámetros adecuados para obtener uno. Los parámetros son los listados en la **Tabla 6**.

Tabla 6. Entradas de punto de solicitud de token para Authorization Code Grant

grant_type	En este caso será "**authorization_code**".
code	Código de autorización concedido por el servidor.
[redirect_uri]	Necesario si la URI fue incluida en la solicitud del código de autorización. Ha de ser la misma.
[state]	Un valor utilizado por el cliente para mantener el estado desde que se hace la petición hasta que se obtiene una respuesta. El servidor de autorización devolverá este mismo parámetro en la respuesta.

Por ejemplo:

```
POST /token HTTP/1.1
Host: server.oauth.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded
grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https://cliente.com/callback
```

RF12: El servidor ha de ser capaz de procesar los parámetros de la petición anterior, y si estos son válidos generará un token y lo proporcionará en la respuesta, que tendrá la estructura especificada en el **RF6** en formato JSON.

Por ejemplo, el servidor de autorización respondería con:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache
{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "token_type":"example",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
  "state":"xyz"
}
```

RF13: Si hay algún problema o dato incorrecto durante la creación del token, se responderá con un mensaje de error que contendrá los mismos parámetros que **RF7**, pero representándolos en JSON. Por ejemplo:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache
{
  "error":"invalid_request"
}
```

4.1.3. IMPLICIT GRANT

RF14: El servidor de autorización soportará una petición del flujo "Implicit Grant" en el punto de autorización, que tendrá la estructura de la **Tabla 7**.

Tabla 7. Entradas de punto de autorización para Implicit Grant

response_type	Su valor en este caso será "**token**".
client_id	El identificador del cliente.
[redirect_uri]	La dirección a la que se redirigirá al usuario después de decidir autorizar o no al cliente. Si no se proporciona se obtendrá de los datos asociados al cliente.
[scope]	Permisos que quiere obtener la aplicación. Si no se especifica se supondrán los permisos por defecto.
[state]	Un valor utilizado por el cliente para mantener el estado desde que se hace la petición hasta que se obtiene una respuesta. El servidor de autorización devolverá este mismo parámetro en la respuesta.

Por ejemplo:

```
GET /authorize?response_type=token&client_id=s6BhdRkqt3
&state=xyz&redirect_uri=http://cliente.com/callback HTTP/1.1
Host: server.oauth.com
```

RF15: Si el usuario acepta la petición en la pantalla del **RF2**, se ha de poder crear un token de acceso asociado al mismo y al cliente que lo solicita. **No** se deberá proporcionar un **token de refresco**.

RF16: Tras lo anterior, si no hay ningún problema, el sistema ha de ser capaz de redirigir al usuario a la URL especificada por el cliente, con los parámetros correspondientes. Estos parámetros son los especificados en el **RF6**, y se incluirán en la parte **hash** de la URL. Por ejemplo:

```
HTTP/1.1 302 Found
Location: http://cliente.com/callback#access_token=2YotnFZF21
&state=xyz&token_type=example&expires_in=3600
```

RF17: En caso de error, el usuario ha de ser informado sin redirigírsele directamente a la aplicación cliente, ya que se le dará la opción de hacerlo manualmente, con un mensaje de error que contendrá los mismos parámetros que **RF7**, pero incluyéndolos en el **hash** de la URL de redirección. Por ejemplo:

```
HTTP/1.1 302 Found
Location:
http://cliente.com/callback#error=access_denied&state=xyz
```

4.1.4. RESOURCE OWNER PASSWORD CREDENTIALS GRANT

RF18: El cliente accederá al **punto de solicitud de token**, donde, ya autenticado, podrá realizar una petición POST con los parámetros adecuados para obtener uno. Los parámetros se detallan en la **Tabla 8**.

Tabla 8. Entradas punto de solicitud de token para Resource Owner Password Credentials Grant

grant_type	En este caso será "**password**".
username	Nombre del usuario.
password	Contraseña del usuario.
[state]	Un valor utilizado por el cliente para mantener el estado desde que se hace la petición hasta que se obtiene una respuesta. El servidor de autorización devolverá este mismo parámetro en la respuesta.

Por ejemplo:

```
POST /token HTTP/1.1
Host: server.oauth.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=password&username=johndoe&password=A3ddj3w
```

RF19: El servidor ha de ser capaz de procesar los parámetros de la petición anterior, y si estos son válidos generará un token y lo proporcionará en la respuesta, que tendrá la estructura especificada en el **RF6** en formato JSON.

RF20: Si hay algún problema o dato incorrecto durante la creación del token, se responderá con un mensaje de error que contendrá los mismos parámetros que **RF7**, pero representándolos en JSON.

4.1.5. CLIENT CREDENTIALS GRANT

RF21: El cliente accederá al **punto de solicitud de token**, donde, ya autenticado, podrá realizar una petición POST con los parámetros adecuados para obtener uno. Los parámetros son los siguientes, explicados en la **Tabla 9**.

Tabla 9. Entradas de punto de solicitud de token para Client Credentials Grant

grant_type	En este caso será "**client_credentials**".
[state]	Un valor utilizado por el cliente para mantener el estado desde que se hace la petición hasta que se obtiene una respuesta. El servidor de autorización devolverá este mismo parámetro en la respuesta.

Por ejemplo:

```
POST /token HTTP/1.1
Host: server.oauth.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials
```

RF22: El servidor ha de ser capaz de procesar los parámetros de la petición anterior, y si estos son válidos generará un **token** y lo proporcionará en la respuesta, que tendrá la estructura especificada en el **RF6** en formato JSON. Esta modalidad **no proporcionará** un **token de refresco**.

RF23: Si hay algún problema o dato incorrecto durante la creación del token, se responderá con un mensaje de error que contendrá los mismos parámetros que **RF7**, pero representándolos en JSON.

4.1.6. ACTUALIZAR TOKEN DE ACCESO

RF24: El cliente accederá al **punto de solicitud de token**, donde, ya autenticado, podrá realizar una petición POST con los parámetros adecuados para obtener uno. En la **Tabla 10** se especifican los parámetros.

Tabla 10. Entradas para punto de solicitud de token para actualizar Token de Acceso

grant_type	En este caso será "**refresh_token**".
refresh_token	Token de refresco proporcionado anteriormente por el servidor de autorización.
[state]	Un valor utilizado por el cliente para mantener el estado desde que se hace la petición hasta que se obtiene una respuesta. El servidor de autorización devolverá este mismo parámetro en la respuesta.

RF25: El servidor ha de ser capaz de procesar los parámetros de la petición anterior, y si estos son válidos generará un **token** y lo proporcionará en la respuesta, que tendrá la estructura especificada en el **RF6** en formato JSON.

RF26: Si hay algún problema o dato incorrecto durante la creación del token, se responderá con un mensaje de error que contendrá los mismos parámetros que **RF7**, pero representándolos en JSON.

4.1.7. ACCEDER AL RECURSO PROTEGIDO

RF27: El servidor del recurso será capaz de extraer el token de la petición del cliente. El token de acceso se puede proporcionar sólo una de las siguientes formas:

- En la cabecera **Authorization** de la petición HTTP**:**

```
POST /resource HTTP/1.1
Host: server.example.com
Authorization: Bearer mF_9.B5f-4.1JqM
```

- Como **parámetro en el cuerpo** de la petición HTTP:

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

access_token=mF_9.B5f-4.1JqM
```

- Como **parámetro en la URL** del recurso accedido:

```
GET /resource?access_token=mF_9.B5f-4.1JqM HTTP/1.1
Host: server.example.com
```

4.2. REQUISITOS NO FUNCIONALES

RNF1: La implementación no está dirigida a un tipo de servicio concreto, por lo que ha de tener una estructura flexible que se pueda acomodar a diferentes situaciones y que pueda extenderse sin demasiada complicación.

RNF2: Esta implementación concreta estará dirigida para su uso en Spray, pero se ha de poder cambiar de contexto fácilmente.

RNF3: El diseño seguido no deberá imponer límites en la escalabilidad del servicio en el que se utilice.

RNF4: El proveedor de autorizaciones OAuth tendrá una estructura que le permitirá ser un servidor independiente o bien acoplarse al servidor del recurso.

RNF5: El sistema de gestión de datos de usuarios, clientes y tokens de acceso elegido por quien utilice esta implementación no deberá afectar a la funcionalidad del servidor de autorización.

RNF6: El sistema dará soporte a los flujos explicados anteriormente, permitiendo añadir flujos personalizados.

RNF7: El sistema será capaz de manejar los diferentes errores que puedan ocurrir en los procesos en los que intervenga OAuth, incluyendo en los códigos de error uno de los listados en la **Tabla 11**, según la naturaleza del problema.

Tabla 11. Códigos de error de OAuth 2.0

Código error	Estado HTTP	Naturaleza
invalid_request	400 Bad Request	Se ha recibido una petición mal formada, faltan parámetros, etc.
invalid_client	401 Unauthorized	El cliente no se está autenticando, no se reconoce o está utilizando un método de autenticación no soportado.
invalid_grant	400 Bad Request	Los parámetros propios del flujo elegido son incorrectos (usuario/contraseña no válidos, código de autorización desconocido, token de refresco expirado, etc.).
unauthorized_client	400 Bad Request	El cliente se ha reconocido pero no se le permite utilizar ese método de solicitud de token.
access_denied	400 Bad Request	El usuario o el servidor de autorización han denegado la petición.
unsupported_response_type	400 Bad Request	El servidor no soporta el método de autorización especificado en el parámetro response_type.
unsupported_grant_type	400 Bad Request	El servidor no soporta el método de concesión de token de acceso especificado en el parámetro grant_type.

Código error	Estado HTTP	Naturaleza
invalid_scope	400 Bad Request	Se han especificado permisos no válidos en el parámetro scope de la petición de token o de autorización.
server_error	500 Server Error	El servidor de autorización encontró un problema que no le permitió completar la petición.
temporarily_ unavaible	503 Service Unavailable	El servidor no puede atender la petición debido a una sobrecarga temporal o por labores de mantenimiento.
insufficient_ scope	403 Forbidden	El recurso al que se está intentando acceder requiere más permisos de los que tiene el token de acceso utilizado.
invalid_token	401 Unauthorized	El token que se está utilizando no es válido o ha caducado.

RNF8: La implementación de OAuth cubrirá las amenazas de seguridad que aparezcan como consecuencia del diseño seguido tanto como le sea posible, delegando el resto en el servicio que más tarde incorpore OAuth (por ejemplo la necesidad de uso de SSL).

RNF9: El código estará organizado y comentado de forma que favorezca su continuación en un futuro. Los comentarios estarán en inglés.

5. DISEÑO

Contenido

En este apartado se expondrá una **descripción de la solución** encontrada para los **requisitos** listados anteriormente. El diseño está **orientado a solucionar la validación de los parámetros de una petición** empezando por los más generales y acabando por los más concretos, llevando a cabo las operaciones de comprobación con **las herramientas que le haya proporcionado el desarrollador** que quiera integrar OAuth en su servicio. Para lograr esto, se han tenido muy presentes los siguientes patrones de diseño [11]:

- **Inversión de control:** sucede cuando es la biblioteca la que **invoca el código del usuario**. El desarrollador tendrá que implementar, por ejemplo, una clase que se encargará de manejar los datos sobre los que opera OAuth.

- **Método plantilla (Template Method):** se utiliza cuando una **superclase** implementa un **proceso** en el que se realizan una **serie de pasos** siempre de la misma forma y las **clases herederas redefinen** estos pasos. Por ejemplo, cuando se llame al método ***process*** de **Granter**, se realizará una validación y si es correcta se llamará a un método más específico implementado por la **clase heredera** (PasswordGranter, etc.).

- **Inyección de dependencias:** la secuencia de operaciones principal necesitará de **clases ya instanciadas** desde el punto más externo de la aplicación, las cuales se **'inyectarán'**, aunque gracias a Scala se hará de forma **implícita** (al validar los datos, se necesita un DataHandler, para crear un token, un TokenFactory, etc.). Además Scala tiene su propia forma de seguir este patrón con su conocido **Cake Pattern** [12].

- **Singleton:** Scala aborda este patrón de diseño con el uso de **Objects**, que son clases con una **única instancia**. Esta estructura es utilizada en distintos puntos del proyecto.

A continuación se explicará el diseño atendiendo a la estructura funcional del proyecto.

5.1. PUNTO DE SOLICITUD DE TOKEN DE ACCESO

El punto por el que se ha comenzado a diseñar la aplicación ha sido por la parte más importante de OAuth: el flujo en el que éste **proporciona un token de acceso**. Un diseño simplificado del que se pensó es el que se muestra en la **Figura 13**.

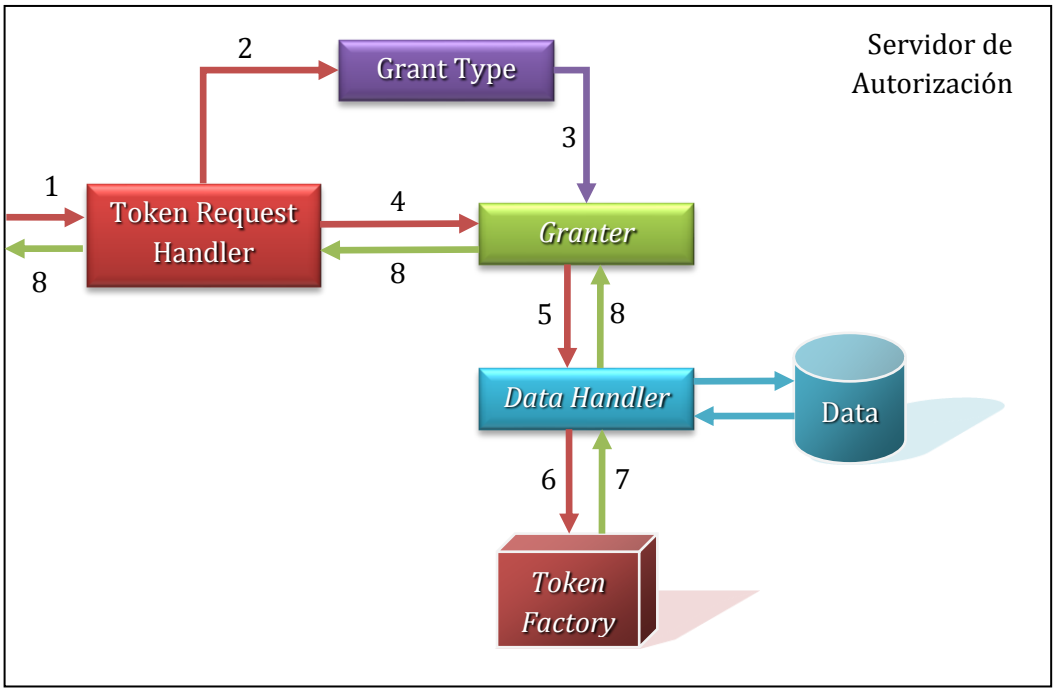


Figura 13. Diseño del flujo de solicitud de token de acceso

1. Llega la petición a **TokenRequestHandler**, donde se realizará un primer análisis de la misma. Se comprueba el parámetro **grant_type**. Si no se encuentra se devolverá un error.

2. Se envía el parámetro **grant_type** a **GrantType**, un objeto que recoge todas las formas de solicitud de token soportadas y comprobará si es válido.

3. Según el valor del parámetro, **GrantType** devolverá un **Granter** u otro, correspondiente al método seguido.

4. **TokenRequestHandler** le pasa los parámetros de la petición al **Granter** específico, para que este termine de resolverla.

5. **Granter** continúa con el análisis de la petición, comprobando que no faltan parámetros. Las validaciones de los datos recibidos, como la id y secreto del cliente, nombre y contraseña del usuario, etc., se realizarán contra **DataHandler**, una implementación propia de quien esté utilizando OAuth que contiene los métodos necesarios para acceder a los datos.

6. Una vez que se ha comprobado la validez de los datos proporcionados, se le pide a **TokenFactory** que genere un token de acceso asociado al cliente y usuario proporcionados.

7. **TokenFactory** genera el Token y se lo devuelve a **DataHandler**, que lo almacenará en sus datos de la forma que se haya especificado.

8. La respuesta con el token de acceso recorre el flujo de forma inversa hasta llegar al punto de salida, liberando los datos ocupados por las diferentes instancias.

Este diseño no interfiere en los requisitos funcionales, pues no limita a las implementaciones de los diferentes **Granter** permitiendo también añadir nuevas, y da la posibilidad de utilizar cualquier forma de gestionar los datos mientras soporte las operaciones requeridas por **DataHandler**. Tampoco limita el tipo de token utilizado, pues se le pedirá a una factoría abstracta.

Se implementará un Granter por cada uno de los flujos en los que interviene el punto de solicitud de token:

- Authorization Code Granter
- Password Granter
- Client Credentials Granter
- Refresh Token Granter

5.2. PUNTO DE AUTORIZACIÓN

Una vez pensado el diseño del punto de solicitud de token es fácil encontrar un **paralelismo** con el punto de autorización siguiendo la misma filosofía.

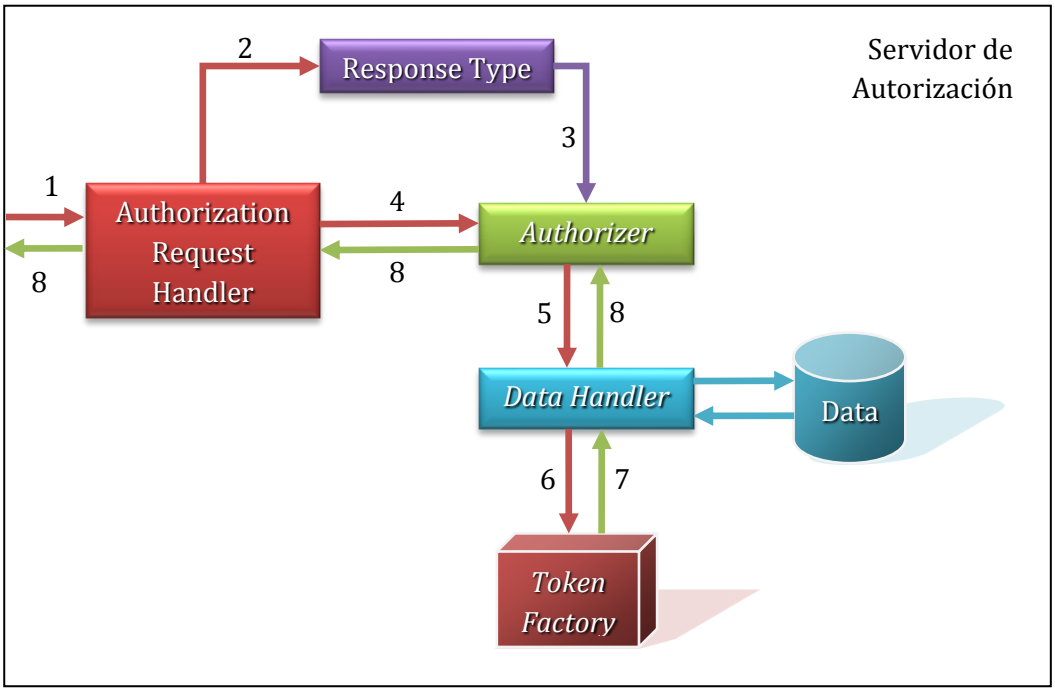


Figura 14. Diseño del flujo del punto de autorización

1. Llega la petición a **AuthorizationRequestHandler**, donde se realizará un primer análisis de la misma. Se comprueba el parámetro **response_type**. Si no se encuentra se devolverá un error.

2. Se envía el parámetro **response_type** a **ResponseType**, un objeto que recoge todas las formas de solicitud de autorización soportadas y comprobará si es válido.

3. Según el valor del parámetro, **ResponseType** devolverá un **Authorizer** u otro, correspondiente al método de autorización seguido.

4. **AuthorizationRequestHandler** le pasa los parámetros de la petición al **Authorizer** específico, para que este termine de resolverla.

5. **Authorizer** continúa con el análisis de la petición, comprobando que no faltan parámetros. Las validaciones de los datos recibidos, como la id del cliente, URI de redirección, métodos de autorización permitidos para el cliente, etc., se realizarán contra **DataHandler**, una implementación propia de quien esté utilizando OAuth que contiene los métodos necesarios para acceder a los datos.

6. Una vez que se ha comprobado la validez de los datos proporcionados, se le pide a **TokenFactory** que genere un código de autorización asociado al cliente y usuario proporcionados en el caso de **Authorization Code** o un token de acceso en el caso de **Implicit Grant**.

7. **TokenFactory** genera el Token o el código y se lo devuelve a **DataHandler**, que lo almacenará en sus datos de la forma que se haya especificado.

8. La respuesta con el **token de acceso** o **código** recorre el flujo de forma inversa hasta llegar al punto de salida, liberando los datos ocupados por las diferentes instancias.

Se implementará un **Authorizer** por cada uno de los flujos en los que interviene el punto de autorización:

- Code Authorizer
- Implicit Authorizer

5.3. ACCESO AL RECURSO PROTEGIDO

La forma de autorizar el acceso a un recurso ha de ser independiente al mismo, por lo que se piensa que esta parte ha de diseñarse como si fuera una capa superior que en caso de no estar autorizado rechace la petición. Se puede ver en la **Figura 15**.

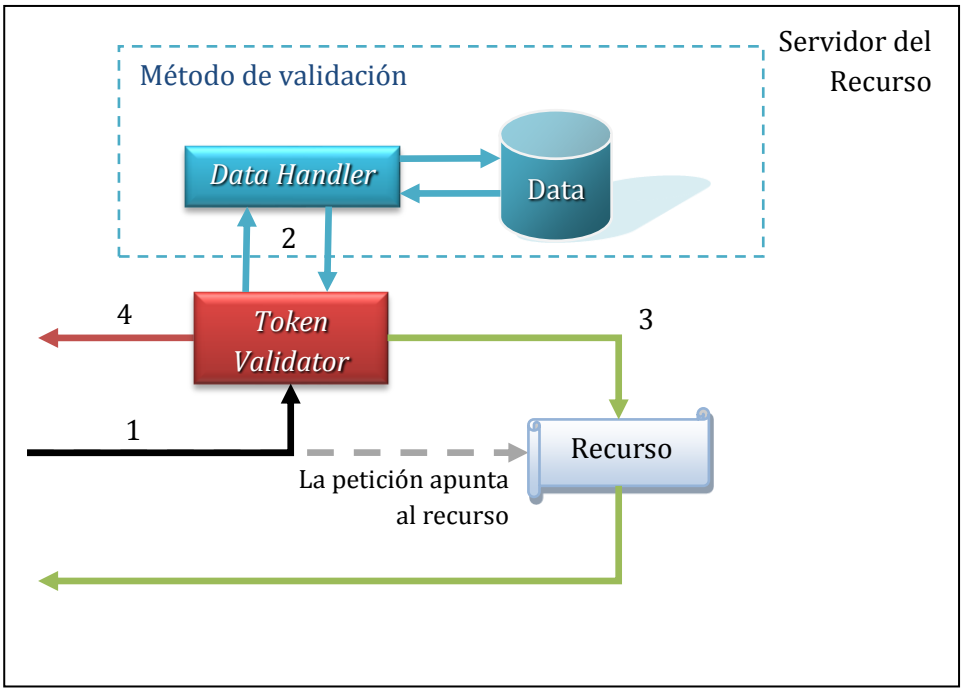


Figura 15. Diseño del acceso al recurso protegido

1. Se envía la petición para acceder al recurso, pero como requiere estar autorizado **TokenValidator** ha de validar el token de acceso proporcionado.
2. Mediante el método de validación especificado (en este ejemplo una comprobación utilizando el **DataHandler**), se comprueba la validez del token, sus permisos y si aún no ha expirado.
3. Si el token es válido, se permite el acceso al recurso y se le devuelve al solicitante.
4. Si el token no es válido se rechaza la petición con el error correspondiente.

Este diseño también permite utilizar una implementación de **TokenValidator** propia, que permitiría escoger entre varios métodos, por ejemplo:

- Comprobar que el **token existe** en los datos porque se ha creado anteriormente.
- Realizar una **petición HTTP** sobre un posible **punto de validación de token** en el servidor de autorización que devuelva los datos del cliente y del usuario asociados al token.
- En el caso de que el **token** contenga sus propios **datos asociados cifrados**, se procede a descifrarlos y extraerlos.

5.4. MODELOS

Previamente se han explicado las clases que actúan en los diferentes procesos seguidos, pero no las clases sobre las que se opera. Hay dos tipos de modelos: modelos que OAuth podrá mostrar como **respuesta** y modelos que se utilizarán únicamente en el **funcionamiento interno**.

5.4.1. MODELOS DE RESPUESTA

El modelo más importante, el que representa al **token de acceso**, queda representado por los campos de la **Tabla 12**.

Tabla 12. Clase AccessToken

AccessToken		
token	Token	Datos del token de acceso.
client	Client	Datos del cliente asociado al token de acceso.
creationDate	DateTime	Fecha de creación de este token.
firstCreationDate	DateTime	Fecha de creación del primer token utilizado, diferente de creationDate si se han obtenido más tokens con el token de refresco.
refreshToken	Option[String]	Token de refresco, si es que el método de obtención de token proporcionó uno.
scope	Scope	Permisos asociados al token.
user	Option[User]	Datos del usuario asociado al token de acceso, si es que lo hay.

Este modelo, en virtud del método **toResponse(*state*)** podrá convertirse en el modelo **TokenResponse**, con los mismos campos que los que se especificaron en la **Tabla 2**. Además, el token, podrá convertirse en JSON con el método **toJson**, o en un mapa con el método **toMap** (para añadirlo a la URL en el caso de Implicit Grant). Con el método **isExpired** se comprobará si el token ha expirado.

Otro de los modelos más importantes es el que representa el **código de autorización**, cuyos campos se describen en la **Tabla 13**.

Tabla 13. Clase AuthorizationCode

AuthorizationCode		
code	String	Código de autorización.
client	Client	Datos del cliente asociado al código.
user	User	Datos del usuario asociado al código.
creationDate	DateTime	Fecha de creación de este código.
scope	Scope	Permisos asociados al código.
redirect_uri	Option[String]	URI de redirección asociada a la petición, si es que se especificó alguna.

Este, con el método **toResponse(*state*)** podrá convertirse en el modelo **AuthorizationResponse**, con los mismos campos que los que se especificaron en la **Tabla 5**. Este a su vez se podrá convertir en un mapa con el método **toMap** para poder añadirlo a la URL de redirección. Se comprobará si ha caducado con el método **isExpired**.

Y por último, el modelo **ErrorResponse**, que contendrá los mismos campos que los especificados en la **Tabla 3**, y tendrá los métodos para mostrarse como JSON y como mapa igual que los anteriores.

5.4.2. MODELOS INTERNOS

Aquí están los modelos User, Client, Scope y Token: **User** es un trait (en este caso como una interfaz en Java) que deberá implementar:

Tabla 14. Trait User

User		
username	String	Identificador del usuario.
perm	List[String]	Lista de permisos.

Client será una clase que represente al cliente, que está pensada para ser construida después de haberlo validado y sus campos son:

Tabla 15. Clase Client

Client		
name	String	Nombre del cliente.
id	String	Identificador del cliente (client_id).
scope	Scope	Permisos por defecto del cliente.
allowedGrants	Set[String]	Conjunto de métodos de solicitud de token.
redirect_uri	String	URI de redirección por defecto o por la que han de empezar las especificadas en las peticiones.

Scope será una clase que contendrá un conjunto de permisos y a través de sus métodos podrá operar con otras clases Scope o Strings que representen un Scope.

Tabla 16. Clase Scope

Scope		
scope	Set[String]	Conjunto de permisos.
text	String	Conjunto de permisos como texto.

Token será un trait pensado para implementar cualquier tipo de Token:

Tabla 17. Trait Token

Token		
ttype	String	Tipo del token.
code	String	Código que representa al token de acceso.

5.5. OTROS

Para acomodar estas implementaciones al servidor de Spray, crearíamos varias clases más. El servicio HTTP que quiera utilizar OAuth tendrá que extender de **OAuthService**, el cual podrá incorporar los distintos **puntos finales** (conectados con las clases correspondientes que atenderán las peticiones) extendiendo a su vez de las clases que los representan.

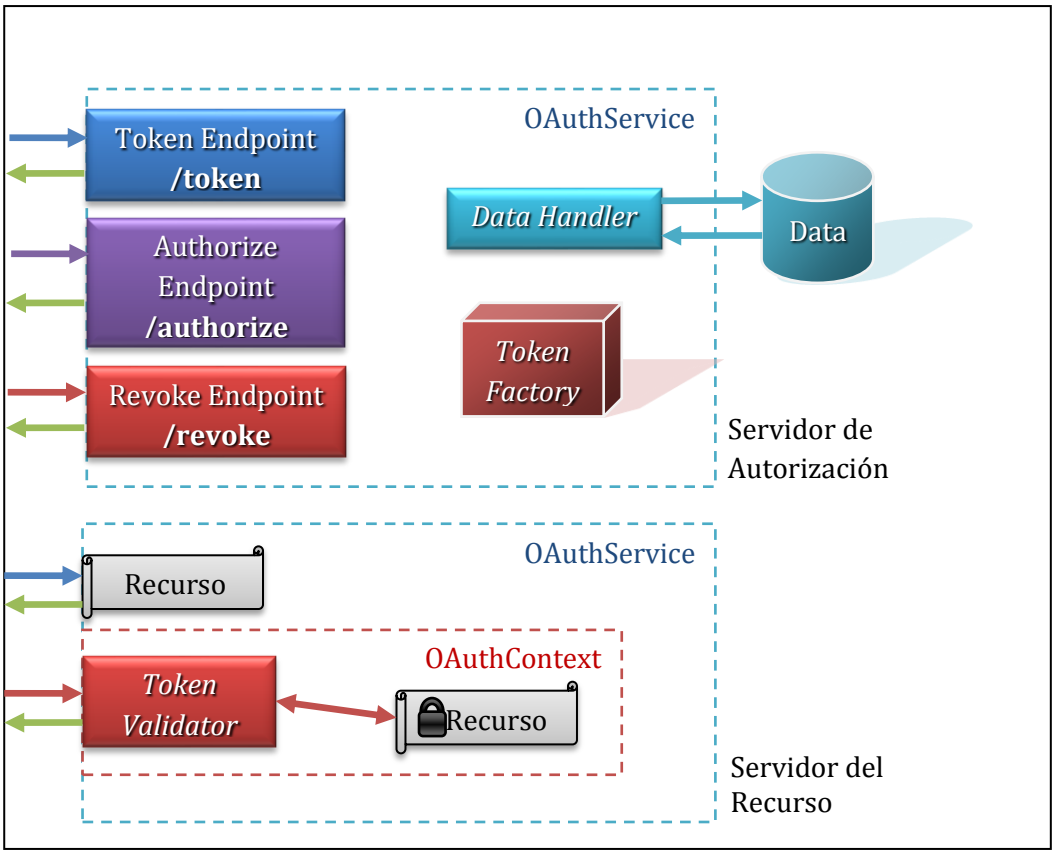


Figura 16. Diposición de OAuth 2.0 en Spray

Si se desea proteger un recurso, también habría que hacerlo en un servicio HTTP de Spray que extienda de **OAuthService**, haciendo así posible utilizar las directivas que requieran autenticarse o un permiso concreto. La clase **OAuthContext** contendrá los **datos asociados a un token** para poder decidir si responder con un recurso accedido en nombre de un usuario determinado o con un error de autorización.

Se ha considerado conveniente añadir un punto final opcional para destruir un token de acceso: **Revoke Endpoint**.

También se crearán objetos que recogen información que debería ser accesible desde cualquier punto del código. Estos objetos son:

- **OAuthParam:** contiene el **texto que representa a cada parámetro** en las peticiones que se realizan en los diferentes puntos de entrada.

- **OAuthError:** contiene los diferentes **códigos de error** que representan a los errores que pueden tener lugar. También incluye descripciones detalladas para errores concretos.

- **OAuthConfig:** carga un **archivo de configuración** con datos como el tiempo de vida del token de acceso, el tiempo de vida del token de refresco, clientes por defecto, etc.

En la especificación del estándar no se expone cómo tratar internamente el scope, es decir, los permisos para acceder a los recursos. En esta implementación se tratarán los permisos como se especifica en la **Figura 17**.

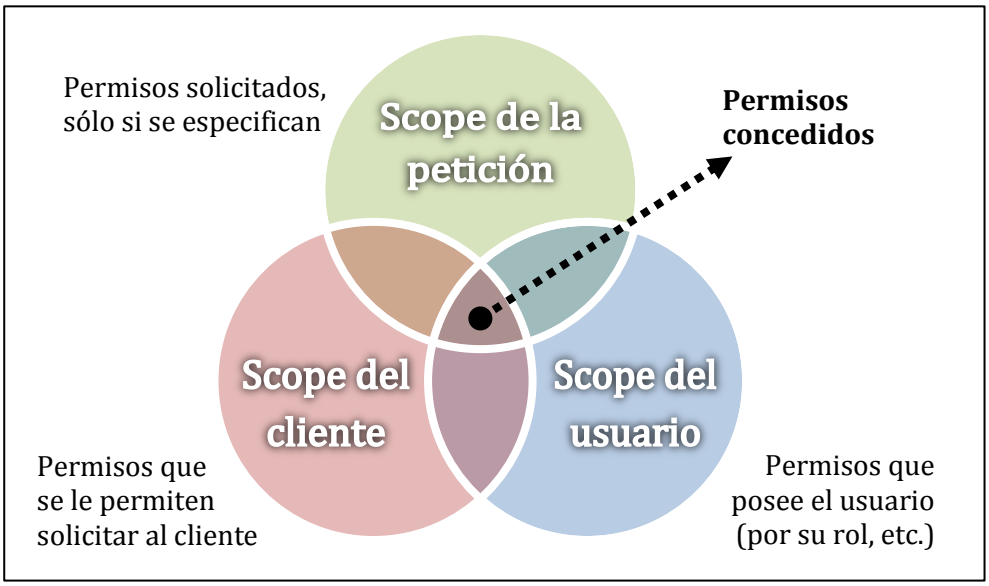


Figura 17. Interpretación del Scope

El cliente, el usuario y la petición de autorización tendrán sus propios permisos de los que dependerán los permisos finalmente concedidos al cliente.

- En el caso del **cliente** estos permisos representarán aquellos que podrá solicitar.

- En el ámbito del **usuario**, serán los permisos que posee el usuario en el sistema, es decir, las acciones que puede llevar a cabo.

- Los permisos de la **petición** son aquellos que el cliente ha solicitado, si no se especifican no se tienen en cuenta.

Los permisos que se le concedan al cliente serán la intersección de todos estos, generando un error en caso de que resulte un conjunto vacío.

5.5.1. DIAGRAMAS DE CLASES Y DE SECUENCIA

Debido a la extensión del Diagrama de clases se ha dividido la representación siguiendo un criterio de organización en paquetes y funcionalidad en el **Anexo B**. Asimismo se han elaborado los diagramas de secuencia expuestos en el **Anexo C**.

6. IMPLEMENTACIÓN

Contenido

Este apartado profundizará en la parte de **codificación del proyecto**, exponiendo **cómo se han abordado los problemas** que han surgido en cada parte siguiendo el orden de la organización del código y la función de cada clase.

6.1. OAUTH/API

6.1.1. OAUTHSERVICE

Este trait representa la **capa más alta de la implementación** de OAuth 2.0. Es el que podrá establecer los **puntos finales** para la obtención del token de acceso y también **validarlo**. Por ello espera que se le definan los componentes que utilizará en estos procesos de forma implícita:

```
implicit val dataHandler: DataHandler
implicit val tokenFactory: TokenFactory
implicit val tokenValidator: TokenValidator
```

También tiene la función de convertir una respuesta del sistema de OAuth en una **respuesta HTTP**, añadiendo de forma implícita un nuevo método toHttpResponse a las respuestas de OAuth.

La directiva **requireAuth** extraerá de las cabeceras el token de acceso proporcionado. Si no lo encuentra, lo buscará como parámetro GET en la dirección, y

si no, como parámetro en el cuerpo de la petición. Si no hay token de acceso, o si no es válido, devolverá un mensaje de error como respuesta.

Si es válido, almacenará el **OAuthContext** en una variable, para poder acceder con el resto de directivas: **requireScope** y **requireUser**. Estas directivas solamente harán comprobaciones sobre los **datos obtenidos por requireAuth**.

6.1.2. OAuthContext

Contiene la definición de la clase **OAuthContext**, que recoge los datos asociados a un token de acceso (token, usuario, cliente y scope).

6.1.3. Endpoints

6.1.3.1. AuthorizeEndpoint

Tiene la función de extraer los parámetros de la petición y convertirlos en un mapa clave-valor para pasárselo a **AuthorizationRequestHandler**.

Representa la ruta **/authorize** de OAuth, que presenta a su vez dos entradas: mediante método **GET**, que será la página a la que llegará el **usuario a través del cliente**, y mediante método **POST**, que supondrá **la autorización del cliente** por parte del usuario con los mismos parámetros que con los que se acompañó el método GET.

Durante la implementación de esta clase se encontró un problema de seguridad que obligó a añadir un **token de corta duración** que confirmara que el proceso había seguido el curso esperado. La situación se refleja en la siguiente figura:

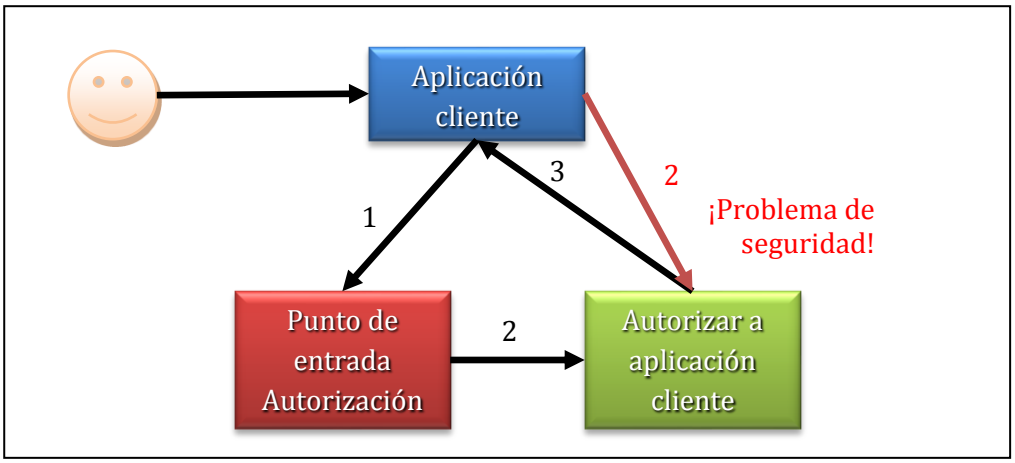


Figura 18. Vulnerabilidad del punto de autorización

Se espera que el cliente envíe al usuario al punto de entrada (1) y ya no haga nada hasta que este regrese (3) de aceptar o denegar la solicitud (2). El problema aparece cuando el cliente envía al usuario a la primera página y más tarde a la segunda **sin pasar por el punto en el que el usuario acepta la solicitud** (2).

Para asegurar que se ha seguido el proceso de forma correcta se ha añadido un **token de autenticidad** que se proporciona al entrar en el punto de autorización (1) y

lo requiere el punto en el que se autoriza a la aplicación cliente, de forma que el cliente al **no conocer este dato** no pueda validar su propia petición.

6.1.3.2. TOKENENDPOINT

Tiene la función de extraer los parámetros de la petición y convertirlos en un mapa clave-valor para pasárselo a **TokenRequestHandler**. Representa la ruta **/token** de OAuth, a la que se accederá mediante una petición **POST**.

La necesidad de mostrar la respuesta en formato JSON fue el problema principal de este punto, el cual se solucionó utilizando las utilidades de JSON que ofrece la librería de Spray.

6.1.3.3. REVOKEENDPOINT

Presentando una estructura similar al punto de solicitud de token, este punto final esperará un **token de refresco para eliminar los datos asociados** al mismo.

Atendiendo a la ruta **/revoke**, se esperará el parámetro '**refresh_token**' para enviárselo directamente al **DataHandler**, que será el encargado de buscar y **eliminar** los datos del token.

6.1.4. REQUEST

6.1.4.1. AUTHORIZATIONREQUESTHANDLER

Esta clase se instanciará en el **punto de autorización** y recibirá los parámetros de la petición de ese momento. Tiene dos métodos que se corresponden con el tipo de petición que haya llegado, que en ambos casos devolverán una **respuesta** que se podrá convertir en un **mapa** para su inclusión en la **URL de redirección**.

- Si es **GET** se llamará al método **validate(*user*, *params*)**, para únicamente validar los datos recibidos y preparar la creación del código de autorización o token.
- Si es **POST** se llamará al método **process(*user*, *params*)**, para crear el código de autorización o token.

6.1.4.2. TOKENREQUESTHANDLER

Esta clase es similar a la anterior pero se instanciará en el **punto de solicitud de token**. Con los parámetros de la petición **POST** se llamará a **process(*params*)**, que delegará la validación de los datos y creación del token en el **Granter** correspondiente.

6.1.5. RESPONSE

Contiene el trait **OAuthResponse** que representa una respuesta de OAuth que se puede devolver como resultado de una petición. También contiene a la clase OAuthResponseWithStatus, que envuelve una respuesta y le asocia un código de estado Http.

Entre las posibles respuestas están:

- **TokenResponse**: representa la respuesta que contiene un token de acceso.
- **ErrorResponse**: representa la respuesta que contiene un error.
- **AuthorizationResponse**: representa la respuesta que contiene un código de autorización.
- **ValidAuthorizationResponse**: representa la respuesta de una validación correcta de una petición en el punto de autorización.

6.2. OAUTH/AUTHORIZER

6.2.1. AUTHORIZER

Es el trait en el que se **delegará la autorización** a partir de los parámetros que reciba. Contiene los **métodos** que tienen en común los **autorizadores** como validar el cliente, validar la URI, el método de obtención de token, etc. Tiene un método sin definir: **processAuthRequest(*user*, *client*, *params*)** que dependerá del autorizador utilizado.

6.2.2. CODEAUTHORIZER

Después de la validación hecha por la clase **Authorizer**, no tiene más que hacer que pedir al **DataHandler** un código de autorización con el que responder.

6.2.3. IMPLICITAUTHORIZER

Después de la validación hecha por la clase **Authorizer**, no tiene más que hacer que pedir al **DataHandler** un token de acceso con el que responder.

6.3. OAUTH/GRANTERS

6.3.1. GRANTER

Es el trait en el que se **delegará la creación del token** a partir de los parámetros que reciba. Contiene los **métodos** que tienen en común los **granters** como validar el cliente, el método de obtención de token, crear token, etc. Tiene un método sin definir: **processGrantRequest(*client*, *params*)** que dependerá del granter utilizado.

6.3.2. AUTHORIZATIONCODEGRANTER

Añade al **Granter** un paso en el que se **valida el código de autorización** proporcionado. Realiza la operación con el **DataHandler** de **OAuthService**.

6.3.3. PASSWORDGRANTER

Añade al **Granter** un paso en el que se **valida el usuario y contraseña** proporcionados. Realiza la operación con el **DataHandler** de **OAuthService**.

6.3.4. CLIENTCREDENTIALSGRANTER

Pasa directamente a **crear el token de acceso**, ya que el cliente ya ha sido validado por **Granter**. **Deshabilita** el token de **refresco**.

6.3.5. REFRESHTOKENGRANTER

Añade al **Granter** un paso en el que se **valida el token de refresco** proporcionado. Realiza la operación con el **DataHandler** de **OAuthService**.

6.4. OAUTH/VALIDATOR

6.4.1. TOKENVALIDATOR

Recibe un parámetro de tipo **Token** a partir del cuál ha de poder obtener un **OAuthContext** (usuario, cliente y scope) que pueda proporcionar a **OAuthService** para las diferentes directivas.

6.5. OAUTH/DATA

6.5.1. DATAHANDLER

Es un trait que recoge todas las operaciones de gestión de datos: validación de cliente, de usuario, creación y consulta de tokens, códigos, etc.

La idea de reunir toda las operaciones dependientes del sistema con el que se va a integrar está viene de la implementación en Java de OAuth 2.0 *oauth2-server*[7].

6.5.2. MODEL

Contiene las clases sobre las que se opera.

6.5.2.1. AUTHORIZATIONCODE

Clase que representa el código de autorización y sus datos asociados.

6.5.2.2. CLIENT

Clase que representa al cliente y sus datos.

[7] https://github.com/yoichiro/oauth2-server

6.5.2.2. USER

Trait del que se espera que extienda el usuario que se vaya a utilizar.

6.5.2.3. SCOPE

Clase que representa al Scope y contiene unas pocas operaciones del mismo.

6.5.2.4. TOKEN / ACCESSTOKEN

Contiene la clase que representa al token de acceso, al token como código, y la factoría de tokens.

6.5.2.5. TOKEN / BEARERTOKEN

Especifica la definición de Token para que cumpla la especificación de un token de tipo **Bearer**. Implementa una factoría de tokens en la que los códigos se generan a partir de la herramienta de generación de identificadores únicos de **java UUID.**

6.6. OAUTH/UTILS

6.6.1. RESPONSETYPE

Hace accesibles desde cualquier punto de la aplicación los valores que puede tener el campo '**response_type**' de la solicitud de autorización. También proporciona un método a partir del cual, según el valor de '**response_type** devuelve el **Authorizer que tenga asignado**, si es que tiene uno.

6.6.2. GRANTTYPE

Hace accesibles desde cualquier punto de la aplicación los valores que puede tener el campo '**grant_type**' de la solicitud de token. También proporciona un método a partir del cual, según el valor de '**grant_type** devuelve el **Granter que tenga asignado**, si es que tiene uno.

6.6.3. OAUTHCONFIG

Carga un archivo de configuración y hace que ciertos valores sean accesibles desde cualquier punto de la aplicación. Se utiliza el sistema de configuración que ofrece Typesafe, que permite expresar una configuración como un fichero JSON.

6.6.4. OAUTHERROR

Hace accesibles los diferentes códigos de error desde cualquier punto de la aplicación. También proporciona descripciones para errores concretos.

6.6.5. OAUTHPARAM

Recoge todas las cadenas de texto que representan el nombre de los parámetros de cada una de las solicitudes.

7. INTEGRACIÓN Y USO

Contenido

Ya con la implementación terminada, el siguiente paso es integrarlo con la API en la que se vaya a utilizar mediante los siguientes pasos: publicar los puntos finales en la API para permitir la obtención de tokens, proteger los recursos de la API y personalizar las implementaciones de más bajo nivel.

7.1. INCLUIR PUNTOS FINALES OAUTH 2.0

Como se explicó anteriormente, para hacer que un punto final esté disponible hay que implementar un Servicio Http (HttpService) que extienda de OAuthService y del punto final que se quiera añadir.

Por ejemplo, si se quieren habilitar los puntos de autorización y solicitud de token:

```
class Service extends HttpServiceActor with OAuthService
              with TokenEndpoint with AuthorizeEndpoint
```

Esto permitirá usar las rutas correspondientes al construir el conjunto de rutas de la API:

```
pathPrefix("oauth") {
    tokenRoute ~ authorizeRoute
}
```

Esto hará que estén disponibles las rutas: **/oauth/token** y **/oauth/authorize**.

Las correspondencias Trait → Método → Ruta son las siguientes:

Trait	Método	Ruta
TokenEndpoint	tokenRoute	/token
AuthorizeEndpoint	authorizeRoute	/authorize
RevokeEndpoint	revokeRoute	/revoke

Tabla 18. Correspondencias trait-método-ruta de los puntos finales

El punto de autorización funciona sobre el fichero authorize.html y error.html ubicados en src/main/resources/html/.

El fichero authorize.html ha de tener un aspecto similar a este:

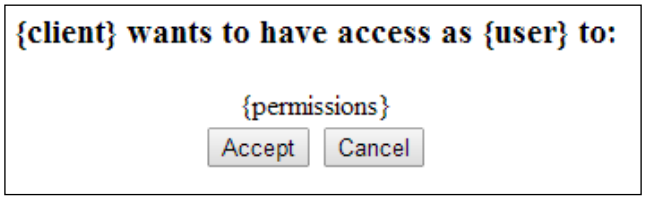


Figura 19. Página de autorización

AuthorizeEndpoint será el encargado de presentar un nuevo fichero html donde el texto con la estructura {valor} será sustituido por los valores correspondientes, listados a continuación en la **Tabla 19**.

Tabla 19. Campos fichero html de Authorize

{client}	Campo name de la clase Client.
{user}	Campo username de la clase User.
{permissions}	Lista de permisos, mostrados como lista desordenada.
{action}	URL sobre la que se tendrá que hacer POST.
{auth_token}	Token de autenticidad que habrá que enviarse en la petición dentro del parámetro **authenticity_token**, por ejemplo en un campo de tipo hidden.

También será necesario incluir los botones de aceptar y cancelar, con los nombres **accept** y **cancel** respectivamente.

En el caso de la página de error sólo tendremos los campos de la **Tabla 20**.

Tabla 20. Campos fichero html de Error

{error}	Código de error de OAuth 2.0.
{description}	Descripción detallada del error.
{previous_uri}	Dirección a la que volver para informar del error al cliente.

7.2. PROTEGER RECURSOS

Una vez establecidos los puntos para obtener el token de acceso se ha de proceder a añadir la capa de seguridad en los recursos a proteger. Con un servicio Http extendiendo de OAuthService podremos utilizar las siguientes directivas:

- **requireAuth { auth => ... }:** Hace que sea **necesario un token de acceso** para acceder al recurso, sin tener en cuenta los permisos. Los datos asociados al token vienen dados en "**auth**".

```
path("secureAPI") {
    requireAuth { auth =>
        ... //Zona protegida
    }
}
```

- **requireScope("scope1", "scope2", ...) { ... }:** Dentro de una directiva requireAuth, puede utilizarse para una vez autenticado comprobar si el token proporcionado tiene los **permisos requeridos**.

```
path("secureAPI") {
    requireAuth { auth =>
        requireScope("read_resource") {
            ... //Zona protegida con permiso read_resource
        }
    }
}
```

- **requireUser { user => ... }:** Dentro de una directiva requireAuth, puede utilizarse para una vez autenticado comprobar si el token proporcionado está **asociado a un usuario**. En caso de no estarlo se pasará a comprobar la siguiente directiva, pudiendo dar entonces dos funcionamientos distintos a una misma ruta según si hay usuario o no.

```
path("secureAPI") {
    requireAuth { auth =>
        requireUser { user =>
            ... //Zona protegida, sólo para usuarios
        } ~ ... //Zona protegida, sólo para clientes
    }
}
```

Estas directivas aplican la seguridad sobre **la directiva que las engloba**, por lo que si se coloca otra directiva al mismo nivel que una de estas, esta nueva directiva también estará protegida, aunque no se recomienda este uso.

7.3. CONFIGURACIÓN

Para gestionar diferentes valores globales se hace uso de **OAuthConfig** que funciona sobre el fichero **/src/main/resources/oauth.conf** (**Tabla 21**).

Tabla 21. Propiedades fichero de configuración

Propiedad	Valor	Uso
token.expiration_time	Long	Tiempo de vida de un token de acceso, en segundos.
token.code_expiration_time	Long	Tiempo de vida de un código de autorización, en segundos.
token.refresh_expiration_time	Long	Tiempo de vida de un token de refresco, en segundos.
clients	Array	Clientes que podrán hacer uso de OAuth. Ver tabla 22.

Los clientes que se pueden especificar en el fichero de configuración tendrán a su vez los siguientes campos expuestos en la **Tabla 22**.

Tabla 22. Propiedades de cliente en fichero de configuración

Propiedad	Valor	Uso
name	String	Nombre que mostrará el cliente.
client_id	String	Identificador del cliente.
client_secret	String	Contraseña del cliente.
scope	String	Scope por defecto del cliente, tal y como se representaría en una petición de token.
grants	String	Métodos de obtención de token permitidos para un cliente. Tienen el mismo formato que Scope.
redirect_uri	String	Directorio base sobre el que se realizarán las redirecciones desde el servidor de autorización.

Un ejemplo sería:

```
oauth {
  token {
    expiration_time = 3600
    code_expiration_time = 300
    refresh_expiration_time = 86400
  }
  clients = [{
      name = Cliente de prueba
      client_id = testclient
      client_secret = testclientsecret
      scope = "get_photos post_comments"
      grants = "refresh_token client_credentials code"
      redirect_uri = "https://www.prueba.com/test"
    }]
}
```

7.4. IMPLEMENTACIONES PROPIAS

Como este proyecto está pensado para su integración en cualquier tipo de servicio se han definido interfaces que permite modificar fácilmente la forma de la que se opera. Estas interfaces son:

- **TokenFactory**, que define la forma de crear un token de acceso, de refresco, código de autorización, etc.
- **DataHandler**, que define cómo se valida un usuario, un cliente, cómo se almacena un token, etc.
- **TokenValidator**, que define cómo se obtiene la información asociada al token de acceso a partir del token que se le proporcione.

7.4.1. TOKENFACTORY

Los métodos a implementar son los siguientes (**Tabla 23**).

Tabla 23. Métodos de TokenFactory

Método	Argumentos	Descripción
next	*atoken*: AccessToken	A partir de los datos que forman un token de acceso, rellena el campo token de la instancia de AccessToken recibida.
nextRefresh	-	Devuelve una cadena que será el código que represente un token de refresco.
nextCode	-	Devuelve una cadena que será el código que represente un código de autorización.
read	*str*: String	Extrae del string 'str' un token de acceso. Este string a su vez es el valor de la cabecera Authorization de la petición Http.
create	*code*: String	Dado un código devuelve una instancia del token de acceso con este código.

7.4.2. TOKENVALIDATOR

El único método a implementar es el siguiente:

Tabla 24. Métodos de TokenValidator

Método	Argumentos	Descripción
validateAccessToken	*token*: Token	A partir del token recibido (normalmente instanciado por el método create de tokenFactory) se obtiene la información asociada al token de acceso y se representa en una instancia de la clase OAuthContext.

7.4.3. DATAHANDLER

Los métodos a implementar se encuentran en la **Tabla 25**.

Tabla 25. Métodos de DataHandler

Método	Argumentos	Descripción
validateClient	*clientId*: String	Utilizando la id del cliente **devuelve** los **datos del cliente** si es que existe. La respuesta es por tanto un **Option[Client].**
validateClient	*clientId*: String *clientSecret*: String	Utilizando la id y secreto del cliente, **devuelve** los **datos del cliente** si es que son correctos. La respuesta es por tanto un **Option[Client]**.
validateUser	*username*: String *password*: String	Utilizando el username y password de un usuario **devuelve** los **datos del usuario** si es que existe. La respuesta es de tipo Option[**User**].
validateCode	*client*: Client *code*: String *redirectUri*: Option[String]	**Comprueba** que el **código de autorización** proporcionado existe y pertenece al cliente que se indicó al crearlo. También comprueba la URI de redirección. Devuelve un **Option[AuthorizationCode].**
validateRefresh Token	*client*: Client *refreshToken*: String	Comprueba que el **token de refresco** existe y pertenece al cliente indicado. Devuelve un **Option[AccessToken].**
createAccess Token	*client*: Client *user*: Option[User] *scope*: Option[String] *refreshEnabled*: Boolean *refreshing*: Boolean	**Crea** un **token de acceso** asociado al cliente y usuario indicados con el scope dado. El resto de parámetros se usarán para las opciones de creación. *RefreshEnabled* indica si el método permite usar token de refresco, y *refreshing* indica si se está refrescando en ese momento. Devuelve un **AccessToken** si todo va bien.
validateAccess Token	*token*: Token	**Devuelve** los datos del token de acceso asociados al código proporcionado. Devuelve un **Option[AccessToken].**
revokeToken	*refreshToken*: String	**Elimina** el **token de acceso** cuyo **token de refresco** coincide con el proporcionado. Devuelve los datos del token de acceso si es que existe, como **Option[AccessToken].**

Método	Argumentos	Descripción
create Authorization Code	*user*: User *client*: Client *scope*: Option[String] *redirectUri*: Option[String]	**Crea** un **código de autorización** asociado al cliente y usuario indicados, con el scope y URI de redirección especificados. Si no hay problemas, devuelve un **AuthorizationCode**.
getCodeInfo	*code*: String	**Recupera** la **información** asociada a un **código de autorización**, si es que existe. Devuelve por lo tanto un **Option[AuthorizationCode]**.
prepareGrant	*user*: User *client*: Client *scope: Option[String]*	**Crea** un **código** asociado al usuario y cliente con el scope indicado que se validará cuando se solicite el token o el código de autorización. Devuelve un **AuthorizationRequest**.
validate Authorization Request	*user:* Option[User] *client:* Option[Client] *authToken*: String	**Comprueba** que el **código** pertenece a uno de los creados con el método anterior y lo elimina aunque el usuario o cliente no sea válido.

8. PRUEBAS Y EJEMPLOS

Contenido

En este apartado se listarán todas las pruebas (unitarias, de integración, etc.) llevadas a cabo y finalmente se expondrán varios ejemplos (una API y dos clientes) desarrollados para terminar de validar el proyecto.

8.1. PLAN DE PRUEBAS

A continuación se listan las pruebas diseñadas y que se han llevado a cabo durante cada una de las fases de desarrollo del proyecto. La implementación desarrollada ha pasado de forma satisfactoria todas estas pruebas.

8.1.1. PRUEBAS UNITARIAS

8.1.1.1. PRUEBAS DE DATAHANDLER

Las pruebas serán de **caja negra**, ya que la implementación que se vaya a utilizar de forma definitiva es desconocida. En cambio las entradas y resultados sí están definidos (**Tablas 26-31**).

Tabla 26. Prueba Unitaria 1 DataHandler

PU1	**Creación de token de Acceso**
El objetivo de esta prueba es comprobar que la simple creación de un token de acceso no da ningún problema desde este punto.	
Entradas	Resultado
→ Cualquier cliente → Cualquier usuario → Cualquier scope	Nuevo token de acceso

Tabla 27. Prueba Unitaria 2 DataHandler

PU2	**Validación de token de Acceso**
El objetivo de esta prueba es comprobar que los resultados del intento de validar un token de acceso son los esperados según las entradas proporcionadas.	
Entradas	Resultado
→ Código del token anterior	Un token de acceso con el mismo cliente, usuario y scope
→ Cualquier código distinto al anterior	Ningún token
→ Código de un token eliminado	Ningún token

Tabla 28. Prueba Unitaria 3 DataHandler

PU3	**Validación de token de Refresco**
El objetivo de esta prueba es comprobar que los resultados del intento de validar un token de refresco son los esperados según las entradas proporcionadas.	
Entradas	Resultado
→ Mismo cliente → Mismo token de refresco	El token de acceso asociado al token de refresco
→ Cualquier otro cliente → Mismo token de refresco	Ningún token de acceso
→ Mismo cliente → Token de refresco inexistente	Ningún token de acceso

Tabla 29. Prueba Unitaria 4 DataHandler

PU4	**Revocar token de Acceso**
El objetivo de esta prueba es comprobar si la eliminación de un token se lleva a cabo correctamente.	
Entradas	Resultado
→ Token de refresco asociado al token	Información del token de acceso eliminado.
→ Token de refresco asociado al token	Ningún token, ya que ha sido eliminado.
→ Cualquier otro token de refresco	Ningún token.

Tabla 30. Prueba Unitaria 5 DataHandler

PU5	**Creación de código de autorización**
El objetivo de esta prueba es comprobar que la simple creación de un código de autorización no da ningún problema desde este punto.	
Entradas	Resultado
→ Cualquier cliente → Cualquier usuario → Cualquier scope → Cualquier URI	Nuevo código de autorización

Tabla 31. Prueba Unitaria 6 DataHandler

PU6	**Validación de código de autorización**
El objetivo de esta prueba es comprobar que los resultados del intento de validar un código de autorización son los esperados según las entradas proporcionadas.	
Entradas	Resultado
→ Mismo cliente → Mismo código → Misma URI	Código de autorización el mismo usuario y scope que los solicitados.
→ Mismo cliente → Mismo código → Misma URI	Ningún código de autorización: al validarse se ha invalidado.
Habiendo creado un nuevo código… → Otro cliente → Mismo código → Misma URI	Ningún código de autorización.
Habiendo creado un nuevo código… → Mismo cliente → Mismo código → Otra URI	Ningún código de autorización.

8.1.1.2. PRUEBAS AUTHORIZER

Estas pruebas serán de **caja blanca**, ya que conocemos la forma de la que está implementado y querremos probar **todos los flujos posibles**. Como hacen uso de un DataHandler, se ha diseñado uno muy simple de prueba que pasa todos los tests del apartado anterior, de forma que estas pruebas puedan seguir considerándose unitarias (**Tablas 32-33**).

Tabla 32. Prueba Unitaria 7 Authorizer

PU7	**Validar solicitud con Authorizer**
La finalidad de esta prueba es comprobar si se responde con un código de autenticidad o no según los parámetros de entrada.	
Entradas	Resultado
→ Cualquier usuario → Identificador de un cliente al que se le permite utilizar este método → La URI de este cliente	Nuevo código de autenticidad
→ Cualquier usuario → Identificador de un cliente inexistente → Una URI	Error: invalid_client
→ Cualquier usuario → Identificador de un cliente al que no se le permite utilizar este método → La URI de este cliente	Error: unauthorized_client

PU7	**Validar solicitud con Authorizer**
→ Cualquier usuario → Identificador de un cliente al que se le permite usar este método → Una URI que no pertenece a este cliente	Error: unauthorized_client

Tabla 33. Prueba Unitaria 8 Authorizer

PU8	**Procesar solicitud con Authorizer**
La finalidad de esta prueba es comprobar si se responde con un código de autorización/token o no según los parámetros de entrada.	
Entradas	Resultado
→ Usuario anterior → Identificador del cliente de la prueba anterior → La URI de este cliente → Código de autenticidad concedido	Nuevo código de autorización o token de acceso (según extienda ImplicitAuthorizer o CodeAuthorizer)
Habiendo creado un nuevo código… → Usuario anterior → Identificador de un cliente inexistente → Una URI → Código de autenticidad concedido	Error: invalid_client
Habiendo creado un nuevo código… → Cualquier usuario → Identificador de un cliente al que no se le permite utilizar este método → La URI de este cliente → Código de autenticidad concedido	Error: unauthorized_client
Habiendo creado un nuevo código… → Cualquier usuario → Identificador de un cliente al que se le permite usar este método → Una URI que no pertenece a este cliente → Código de autenticidad concedido	Error: unauthorized_client
Sin crear un nuevo código… → Usuario anterior → Identificador del cliente de la prueba anterior → La URI de este cliente → Código de autenticidad concedido	Error: invalid_request
→ Usuario anterior → Identificador del cliente de la prueba anterior → La URI de este cliente → Código de autenticidad desconocido	Error: invalid_request

8.1.1.3. PRUEBAS GRANTER

Estas pruebas serán de **caja blanca**, ya que conocemos la forma de la que está implementado y querremos probar **todos los flujos posibles**. Hacen uso del mismo **DataHandler** de prueba (**Tablas 34-37**).

Tabla 34. Prueba Unitaria 9 Granter

PU9	**Procesar solicitud con Granter**
La finalidad de esta prueba es comprobar si se llama al método del Granter específico o se rechaza la petición directamente, según el cliente que esté realizando la solicitud, cuyas credenciales están en los parámetros de entrada.	
Entradas	Resultado
→ Credenciales de cualquier cliente que pueda usar el método especificado	Llama al método de procesamiento específico de la clase que extiende de esta.
→ Credenciales de cualquier cliente que no pueda usar el método especificado	Error: unauthorized_client
→ Credenciales de un cliente inexistente	Error: invalid_client

Tabla 35. Prueba Unitaria 10 AuthorizationCodeGranter

PU10	**Procesar solicitud con AuthorizationCodeGranter**
La finalidad de esta prueba es comprobar si se responde con un token de acceso o no según los parámetros de entrada.	
Entradas	Resultado
→ Mismo cliente → Código de autorización concedido → Misma URI que la que se especificó en la solicitud del código	Nuevo token de acceso.
→ Mismo cliente → Código de autorización concedido → Una URI distinta de la que se especificó en la solicitud del código	Error: invalid_grant
→ Mismo cliente → Código de autorización inexistente → Misma URI que la que se especificó en la solicitud del código	Error: invalid_grant
→ Otro cliente → Código de autorización concedido → Misma URI que la que se especificó en la solicitud del código	Error: invalid_grant

Tabla 36. Prueba Unitaria 11 PasswordGranter

PU11	**Procesar solicitud con PasswordGranter**
La finalidad de esta prueba es comprobar si se responde con un token de acceso o no según los parámetros de entrada.	
Entradas	Resultado
→ Mismo cliente → Usuario y contraseña que pertenezcan a un usuario real (el de prueba)	Nuevo token de acceso.
→ Mismo cliente → Usuario y contraseña que no pertenezcan a nadie	Error: invalid_grant

Tabla 37. Prueba Unitaria 12 RefreshTokenGranter

PU12	**Procesar solicitud con RefreshTokenGranter**
La finalidad de esta prueba es comprobar si se responde con un token de acceso o no según los parámetros de entrada.	
Entradas	Resultado
→ Mismo cliente → Token de refresco de un token de acceso de este cliente → Mismo scope (opcional)	Nuevo token de acceso.
→ Mismo cliente → Token de refresco de un token de acceso de este cliente → Otro scope	Error: invalid_scope
→ Mismo cliente → Token de refresco inexistente	Error: invalid_grant
→ Otro cliente → Token de refresco de un token de acceso de este cliente → Mismo scope (opcional)	Error: invalid_grant

8.1.2. PRUEBAS DE INTEGRACIÓN

Estas pruebas de integración se van a aplicar sobre capas que sólo añaden ciertas comprobaciones que pueden rechazar la petición.

En caso de aceptarla, pasarán la petición a un nivel inferior que corresponde a las pruebas realizadas anteriormente, las cuáles se han repetido en este nivel, pero sólo se van a especificar las nuevas pruebas añadidas para no repetir información no relevante. Ver **Tablas 38-41**.

8.1.2.1. PRUEBAS DE AUTHORIZATIONREQUESTHANDLER

Tabla 38. Prueba de Integración 1 AuthorizationRequestHandler

PI1	**Procesar solicitud con AuthorizationRequestHandler**
La finalidad de esta prueba es comprobar si se responde con un token de acceso, código de autorización o error según los parámetros de entrada.	
Entradas	Resultado
→ Credenciales de un cliente → Tipo de respuesta esperada: code	Ver Prueba Unitaria 8 como CodeAuthorizer
→ Credenciales de un cliente → Tipo de respuesta esperada: token	Ver Prueba Unitaria 8 como ImplicitAuthorizer
→ Credenciales de un cliente → Tipo de respuesta esperada desconocida	Error: unsupported_response_type
→ Credenciales de un cliente → Tipo de respuesta esperada no especificada	Error: invalid_request

8.1.2.2. PRUEBAS DE TOKENREQUESTHANDLER

Tabla 39. Prueba de Integración 2 TokenRequestHandler

PI2	**Procesar solicitud con TokenRequestHandler**
La finalidad de esta prueba es comprobar si se responde con un token de acceso o no según los parámetros de entrada.	
Entradas	Resultado
→ Credenciales de un cliente → Tipo de obtención de token: authorization_code	Ver Prueba Unitaria 10
→ Credenciales de un cliente → Tipo de obtención de token: password	Ver Prueba Unitaria 11
→ Credenciales de un cliente → Tipo de obtención de token: client_credentials	Ver Prueba Unitaria 9 como ClientCredentialsGranter
→ Credenciales de un cliente → Tipo de obtención de token: refresh_token	Ver Prueba Unitaria 12
→ Credenciales de un cliente → Tipo de obtención de token desconocido	Error: unsupported_grant_type
→ Credenciales de un cliente → Tipo de obtención de token no especificado	Error: invalid_request

8.1.2.3. PRUEBAS DE OAUTHSERVICE

Tabla 40. Prueba de Integración 3 OAuthService

PI3	**Crear un token de acceso a partir de una petición Http**
La finalidad de esta prueba es comprobar si se responde con un token de acceso o no según la petición Http.	
Entradas	Resultado
→ Credenciales de un cliente → Tipo de obtención de token: cualquiera que se le permita al cliente → Datos asociados a la obtención del token	JSON con los datos del token de acceso

Tabla 41. Prueba de Integración 4 OAuthService

PI4	**Solicitar un recurso protegido mediante una petición Http**
La finalidad de esta prueba es comprobar las respuestas del servidor según la forma de acceder al recurso protegido.	
Entradas	Resultado
→ Token de acceso no válido → Ubicación de recurso protegido	Error: invalid_token
→ Token de acceso válido → Ubicación de recurso protegido	Recurso obtenido
→ Token de acceso válido → Ubicación de recurso protegido por un scope que posee el token	Recurso obtenido
→ Token de acceso válido → Ubicación de recurso protegido por un scope que no posee el token	Error: insufficient_scope
→ Token de acceso válido tras eliminarlo → Ubicación de recurso protegido	Error: invalid_token

8.1.3. PRUEBAS FUNCIONALES

8.1.3.1. PRUEBAS AUTOMATIZADAS

Todas las pruebas realizadas anteriormente, además de confirmar la **correcta gestión de errores**, demuestran que **se cumple la funcionalidad** requerida en los módulos abordados, por lo que las mismas servirían como pruebas funcionales.

8.1.3.2. PRUEBAS MANUALES

Hay que tener en cuenta que no siempre va a interactuar una aplicación con la implementación de OAuth 2.0 ya que la mayoría de veces un usuario tendrá que navegar dentro del punto de autorización, el cual se ha pensado que sería correcto probar de forma manual para ponerse en su lugar.

Al entrar en el punto de autorización, en la implementación de ejemplo se requerirá que el usuario se autentique mediante la cabecera de autenticación, o bien en la ventana que aparecerá si la petición no tiene la cabecera.

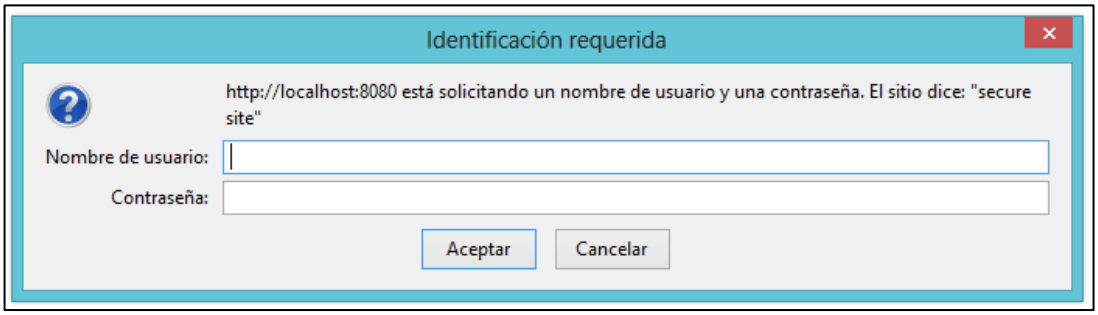


Figura 20. Ventana de autenticación

Si se introduce un usuario no válido volverá a aparecer la ventana, y si se cancela aparecerá un mensaje de error.

Al introducir un usuario válido llegaremos a la siguiente pantalla, para un cliente válido al que se le permite utilizar el método de autorización usado, habiendo expresado correctamente los parámetros en la dirección:

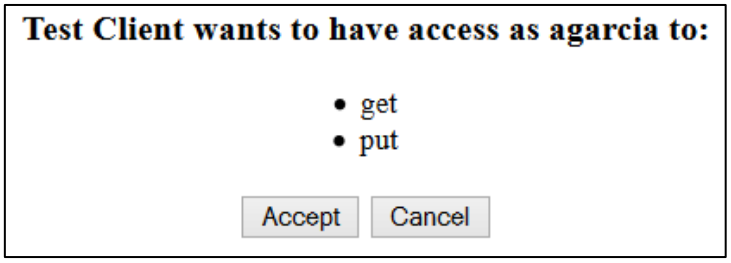


Figura 21. Pantalla de punto de autorización

Se mostrará el nombre del cliente, el scope solicitado (el del cliente por defecto) y los botones para aceptar o cancelar.

Si ocurre algún error, como un parámetro no válido o un cliente no autorizado a usar este método, se mostrará una pantalla como la siguiente:

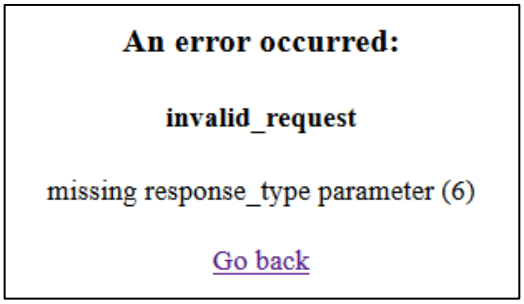


Figura 22. Pantalla de error en punto de autorización

En esta pantalla se le dará la opción al usuario de informar al cliente siguiendo un enlace cuya URL contiene el código y descripción del error correspondiente.

Con estas pantallas, que permiten saber cómo ha ido la solicitud, se han realizado de forma manual las pruebas descritas en las **Tablas 42-44**.

Tabla 42. Prueba Manual 1 Punto de autorización

PM1	**Acceder al punto de autorización**
La finalidad de esta prueba es comprobar si sólo se alcanza la pantalla de autorización con los datos esperados.	
Entradas	Resultado
→ Identificador de un cliente que soporta el método 'code' → URI correcta del cliente (opcional) → Scope válido (opcional)	Aparece la pantalla de autorización.
→ Identificador de un cliente que soporta el método 'token' → URI correcta del cliente (opcional) → Scope válido (opcional)	Aparece la pantalla de autorización.
→ Identificador de un cliente que soporta el método 'token' → URI incorrecta del cliente → Scope válido (opcional)	Error: unauthorized_client
→ Identificador de un cliente que soporta el método 'token' → URI correcta del cliente (opcional) → Scope no válido (uno que el cliente no posea)	Error: invalid_scope
→ Identificador de un cliente que no posea el método de autorización especificado	Error: unauthorized_client
→ Identificador de un cliente inexistente	Error: invalid_client
→ Identificador de un cliente → Ningún tipo de respuesta	Error: invalid_request
→ Ningún identificador de un cliente → Tipo de respuesta	Error: invalid_client

Tabla 43. Prueba Manual 2 Punto de autorización

PM2	**Cancelar solicitud en el punto de autorización**
La finalidad de esta prueba es comprobar qué ocurre al pulsar sobre el botón de cancelar.	
Entradas	Resultado
→ Desde pantalla de autorización 'code' → Pulsar cancelar	Error: access_denied
→ Desde pantalla de autorización 'token' → Pulsar cancelar	Error: access_denied

Tabla 44. Prueba Manual 3 Punto de autorización

PM3	**Aceptar solicitud en el punto de autorización**
La finalidad de esta prueba es comprobar si sólo se le concede el token de acceso o código de autorización al cliente en el caso oportuno.	
Entradas	Resultado
→ Desde pantalla de autorización 'code' → Pulsar aceptar	Se redirecciona a la página: http://test/?code=61e3c60ee2374dca955f11beecf8a8d670eddb44fa00454f9ffed87761bf1f82
→ Desde pantalla de autorización 'token' → Pulsar aceptar	Se redirecciona a la página: http://test/#token_type=Bearer&access_token=5834f994-82c1-4917-a9b3-f616009ef6c0&expires_in=3600&scope=get+put
Habiendo pulsado cancelar, volver atrás y sobre la misma solicitud: → Pulsar aceptar	Error: invalid_request
Habiendo pulsado cancelar, volver atrás y sobre la misma solicitud: → Pulsar aceptar	Error: invalid_request
→ Enviar al usuario al punto de autorización → Enviar al usuario con método post sobre el punto de autorización (es decir, aceptar desde otra página)	Error: invalid_request

8.2. EJEMPLOS

Para probar la implementación en un contexto más real se ha creado una serie de ejemplos relacionados entre sí, representados en la siguiente figura:

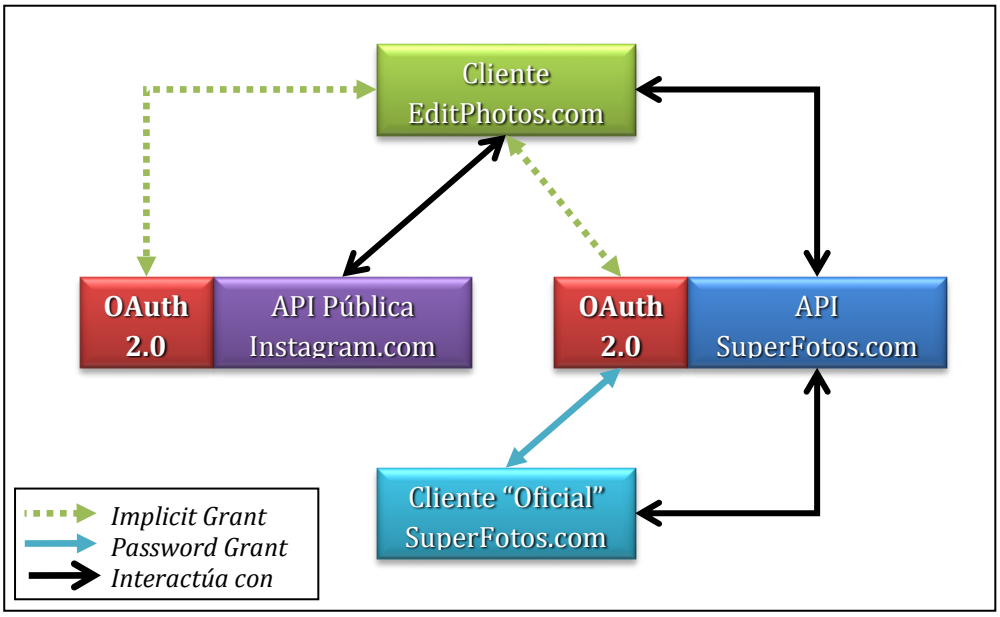


Figura 23. Ejemplos creados y sus relaciones

Los elementos de la figura son:

- **API SuperFotos.com:** una **API** que simula un servicio de subida, almacenamiento y publicación de fotos con usuarios que pueden interaccionar entre sí.
- **Cliente "Oficial" SuperFotos.com:** representa a un **cliente confidencial** y de confianza. El usuario lo utiliza sin saber que por debajo se usa el método de **Resource Owner Password Credentials Grant** para acceder a las partes de la API que requieren autorización.
- **API Instagram:** la API pública real de Instagram, la cual utiliza su propia implementación de OAuth 2.0 pero siguiendo el estándar, por lo que se podrá utilizar de la misma forma que la de ejemplo.
- **Cliente EditPhotos.com:** un **cliente público** que mediante el método de **Implicit Grant** accederá a la API de Instagram o de SuperFotos.com para obtener las fotos de ese usuario, y tras aplicarles un ligera edición, permitirá publicar el resultado en SuperFotos.com (la API pública de Instagram no ofrece la funcionalidad de realizar publicaciones).

8.2.1. EJEMPLO DE API SPRAY CON OAUTH 2.0: API "SUPERFOTOS.COM"

Esta API, después de obtener el token de acceso, permite realizar una serie de acciones, las cuales se podrán realizar sólo si el token posee el scope requerido.

Tabla 45. Operaciones API SuperFotos.com

Operaciones API SuperFotos.com	
Ver información del usuario	
Operación	GET /me/
Permisos	view_user_info

Ver información de otro usuario	
Operación	GET /users/{id}/
Permisos	view_friend_info

Ver imágenes del usuario	
Operación	GET /me/pictures/
Permisos	view_user_pictures

Ver imágenes de otro usuario	
Operación	GET/users/{id}/pictures/
Permisos	view_friend_pictures

Publicar imagen	
Operación	POST /me/pictures/
Paráms.	url, title
Permisos	perform_actions

Eliminar Imagen	
Operación	DELETE /me/pictures/{id}/
Permisos	delete_pictures

Ver a quién le ha gustado una imagen	
Operación	GET {picture}/stars/
Permisos	view_user_pictures / view_friend_pictures

Ver quién ha compartido una imagen	
Operación	GET {picture}/shares/
Permisos	view_user_pictures / view_friend_pictures

Marcar como favorita una imagen	
Operación	POST {picture}/stars/
Permisos	perform_actions

Compartir una imagen	
Operación	POST {picture}/shares/
Permisos	perform_actions

Ver publicaciones recientes de los usuarios seguidos	
Operación	GET /me/timeline
Permisos	view_friend_pictures

Ver seguidores de un usuario	
Operación	GET {user}/followers/
Permisos	view_user_info / view_friend_info

Ver a quién sigue un usuario	
Operación	GET {user}/following/
Permisos	view_user_info / view_friend_info

Seguir a un usuario	
Operación	POST /me/following/{user}/
Permisos	follow_user

En cuanto a OAuth 2.0, las rutas correspondientes se encuentran en /oauth/. El punto de solicitud de token y de eliminación de token son los mismos, pero el punto de autorización ha sido modificado para utilizar otro método de autenticación (mediante un formulario).

Figura 24. Pantalla de autenticación en OAuth 2.0 de SuperFotos.com

En el ejemplo de EditPhotos.com se mostrará cómo es la pantalla en la que se muestran los permisos que solicita la aplicación cliente.

8.2.2. EJEMPLO DE CLIENTE CONFIDENCIAL: SUPERFOTOS.COM

Este cliente representa a uno en el que el usuario confía plenamente, ya que le cede su nombre y contraseña para entrar en el sistema al utilizar el método de obtención de token Resource Owner Password Credentials Grant. Se ha creado utilizando Play Framework, que permite crear una aplicación web con Scala.

Figura 25. Pantalla de login SuperFotos.com

Tras hacer login exitosamente se accederá a la aplicación, donde se podrán realizar diferentes acciones: ver el propio perfil, marcar como favoritas o compartir las fotos, ver a quién se sigue, las fotos de ese usuario, las últimas fotos de todos los usuarios a los que sigues, etc.

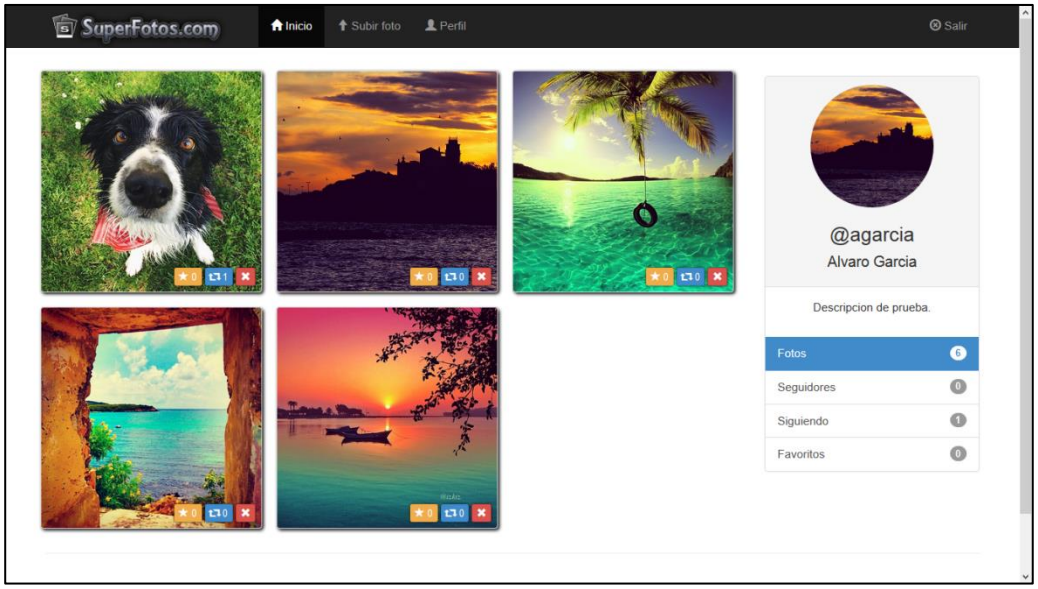


Figura 26. Pantalla de aplicación SuperFotos.com

Esta aplicación almacena el token de acceso en una cookie, de forma que el robo de este token no será tan perjudicial como robar las credenciales, ya que tiene un tiempo de vida limitado.

8.2.3. EJEMPLO DE CLIENTE PÚBLICO: EDITPHOTOS.COM

Este cliente representa aquel en el que el usuario no confía, por lo que querrá evitar que este maneje sus credenciales y a su vez que los permisos concedidos sean los mínimos posibles. La lógica en la que interviene la autorización se lleva a cabo en el navegador del usuario, ya que en su mayoría está hecho utilizando JQuery. Las páginas web las sirve una aplicación de Spray. Esta sería la pantalla principal:

Figura 27. Pantalla principal EditPhotos.com

Se pueden observar las dos alternativas de aplicaciones para importar las fotos. Elegiremos **Instagram**, lo que nos llevará al punto de autorización de OAuth 2.0:

```
https://instagram.com/oauth/authorize/?
client_id=a52b75f31ce64945b030e29e26d23328
&redirect_uri=http://localhost:9090/startig.html
&response_type=token
```

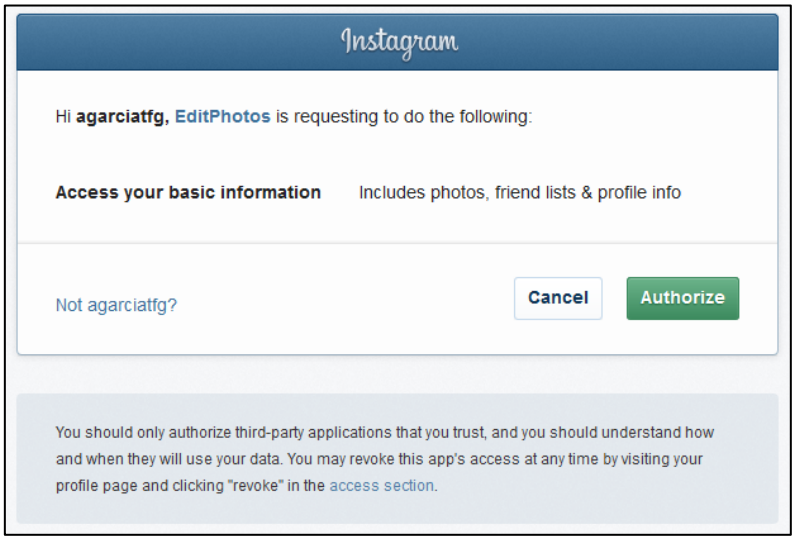


Figura 28. Punto de autorización de la API de Instagram

En esta página se listan los **permisos** que requiere la aplicación además de algún mensaje de advertencia.

Cuando se pulsa aceptar nos redirecciona a EditPhotos de nuevo, con **el token de acceso** incluido en la **parte hash** de la URL:

```
http://localhost:9090/startig.html#access_token=1277192651.a5
2b75f.041a015056a2450eb612771d108bd6dc
```

El token creado podrá extraerse mediante código **Javascript** ejecutado en el navegador del usuario. Tras seleccionar una foto, una capa y crear la foto podremos publicarla en SuperFotos.com utilizando el botón que aparece en la siguiente figura.

Figura 29. Pantalla de edición de fotos de EditPhotos.com

Al pulsar en el botón nos llevará al punto de autorización de SuperFotos.com, donde habrá que autenticarse en la pantalla de login ya mostrada.

```
http://localhost:8080/oauth/authorize?
state=photo6181.png
&scope=view_user_info+perform_actions
&redirect_uri=http://localhost:9090/sharesf.html
&client_id=editphotos
&response_type=token
```

En este caso el parámetro **state** se utiliza para saber que la imagen que querremos publicar corresponde a la que tiene ese identificador para así **mantener la continuidad** de la operación.

Tras autenticarse se verá la pantalla de autorización con los **permisos** que requiere la aplicación y un pequeño mensaje de advertencia. Si se pulsa aceptar se creará el token y se devolverá en la URL seguida en la redirección.

Figura 30. Punto de autorización de la API de SuperFotos.com

Al aceptar se volverá a la página de EditPhotos.com, incluyendo el token en la URL como en el caso de Instagram:

```
http://localhost:9090/sharesf.html#
access_token=f9549a48-0296-49d9-9624-d8bd3e5a9672
&state=photo6181.png
&expires_in=3600
&scope=view_user_info+perform_actions
&token_type=Bearer
```

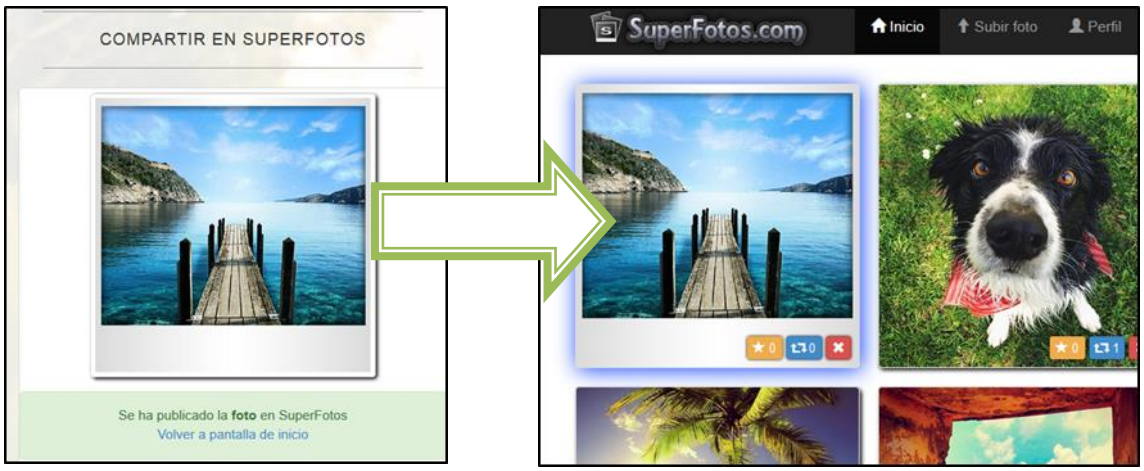


Figura 31. Publicar foto en SuperFotos.com desde EditPhotos.com

9. CONCLUSIONES Y TRABAJO FUTURO

Contenido

Este apartado constituye una **valoración** del proyecto y su posterior **conclusión** en todos los aspectos, tanto en el ámbito **académico** como en el **personal**, además de las posibles **mejoras** e ideas que por ciertas causas se han establecido fuera del alcance del proyecto.

9.1. RESULTADO DEL PROYECTO

La implementación de un proveedor de autorizaciones OAuth 2.0 en Scala se ha llevado a cabo de **forma satisfactoria**, lo que queda demostrado tras **superar las pruebas** descritas anteriormente, **integrar OAuth 2.0 en un servicio** de ejemplo, y lo más importante, **utilizarlo en un proyecto real**.

Es posible que no sea la solución definitiva, es algo muy difícil para el trabajo de una sola persona, porque es **imposible conocer todos los contextos** en los que es posible utilizar OAuth y por ello se planteó desde el principio hacer **público** el **código** permitiendo que otros usuarios interesados por el tema puedan **aportar** sus ideas y mejoras.

9.2. APORTACIONES PERSONALES

La realización de este proyecto conllevaba la **familiarización con muchas herramientas** que hasta el momento me eran desconocidas (como Scala, Spray, Play, etc.) las cuales me han aportado **conocimientos muy valiosos** que han conseguido que me vea capaz y motivado para repetir esta forma de trabajo en el futuro. El hecho de que sean **herramientas** relativamente **recientes** ha provocado que sea mucho más **complicado** encontrar soluciones para problemas poco frecuentes, pero que finalmente se han alcanzado.

El estudio pormenorizado OAuth 2.0 me ha servido para que resulte, a partir de ahora, muy sencillo interactuar con una API que esté protegida con este sistema, lo cual ha supuesto una motivación constante.

9.3. VALORACIÓN DE OAUTH 2.0

OAuth 2.0 nace para **simplificar** la tan compleja especificación de OAuth 1.0, lo que a mi parecer **se consigue** (aunque sigue siendo compleja) pero a cambio de un **coste en seguridad** que ha sido muy **criticado** pero aún así adaptado por organizaciones bastante importantes, aunque algunas también han decidido mantener la implementación de **OAuth 1.0** o **1.0a** (Figura 32). Este problema tan recurrente que enfrenta a la seguridad con la usabilidad queda perfectamente representado en la **Figura 33**.

Figura 32. Uso de especificaciones OAuth (Anexo D)

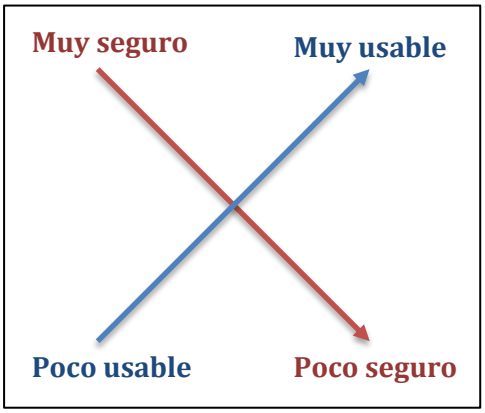


Figura 33. Usabilidad frente a seguridad

OAuth 2.0 **elimina** la necesidad de **firmar los mensajes** a la hora de realizar la comunicación entre el cliente y el servidor de autorización, lo que hace que se requiera una capa adicional de seguridad como **SSL** o **TLS**.

Este punto, entre otros, ha sido criticado incluso por un editor de la misma especificación, **Eran Hammer-Lahav**, que **abandonó el grupo de trabajo** señalando que se estaban alejando de lo que OAuth realmente debería ser [13]. Aunque por otra parte, también dice que **con cuidado** se puede realizar una implementación de **OAuth 2.0**, siempre siendo consciente de los **riesgos de seguridad** que deben cubrirse. En este punto **Hammer-Lahav** pone como ejemplo los casos de implementación de **Facebook** y **Google**.

La **reciente vulnerabilidad** que se hizo pública de la **implementación de SSL** que utilizaban gran cantidad de servicios, es un ejemplo de los problemas que acarrea que OAuth 2.0 delegue gran parte de lo referente a seguridad en otro protocolo, ya que **si este no es seguro**, **OAuth 2.0 tampoco** lo será.

En cualquier caso, el conjunto de potencialidades de **OAuth 2.0** abordadas a lo largo de este trabajo llevan a concluir que ha marcado **un antes y un después** en el ámbito de las **TIC**, lo que ha motivado el análisis detallado de todos los componentes, ventajas y restricciones de OAuth, entendiendo finalmente que sólo después de un estudio tan profundo es posible hacer frente a la implantación en un entorno real.

9.4. TRABAJO FUTURO

Esta implementación permite a su usuario **adaptar OAuth 2.0** a la **arquitectura de su sistema** y a la **forma** que posee de **gestionar los datos**. Una mejora sería realizar **implementaciones base** para cada una de estas arquitecturas de forma que a la hora de integrarlo sólo haya que hacer unos pocos ajustes seleccionando la opción que más se ajuste, en vez de implementar toda la lógica.

En cuanto a la funcionalidad, se podrían añadir mejoras aunque innecesarias ya que se **sigue un estándar** que **ya se está cumpliendo**, así que otra opción sería **optimizar el código** en términos de **eficiencia** tras un estudio exhaustivo.

Tras presentar este Trabajo de Fin de Grado, se ha determinado que el código de la implementación estará disponible en el siguiente **repositorio de GitHub**:

PUBLIC REPOSITORY algd / **oauth2.0-scala**

https://github.com/algd/oauth2.0-scala

BIBLIOGRAFÍA

[1] "The Nexus of Forces: Social, Mobile, Cloud and Information." [Online]. Available: https://www.gartner.com/doc/2049315#a-482575219. [Accessed: 31-Mar-2014].

[2] R. Boyd, *Getting started with OAuth 2.0*. 2012.

[3] "OpenID Connect | OpenID." [Online]. Available: http://openid.net/connect/. [Accessed: 04-May-2014].

[4] D. Hardt, "The OAuth 2.0 Authorization Framework," 2012. [Online]. Available: http://tools.ietf.org/html/rfc6749. [Accessed: 28-Mar-2014].

[5] J. Sutherland, "The Scrum Guide ™," no. July, 2013.

[6] D. Rountree, *Federated Identity Primer*. 2012.

[7] "OAuth Community Site." [Online]. Available: http://oauth.net/. [Accessed: 02-Apr-2014].

[8] "AuthSub for Web Applications - Google Accounts Authentication and Authorization — Google Developers." [Online]. Available: https://developers.google.com/accounts/docs/AuthSub?hl=es. [Accessed: 01-Apr-2014].

[9] "Browser-Based Authentication (BBAuth) - YDN." [Online]. Available: http://developer.yahoo.com/bbauth/. [Accessed: 01-Apr-2014].

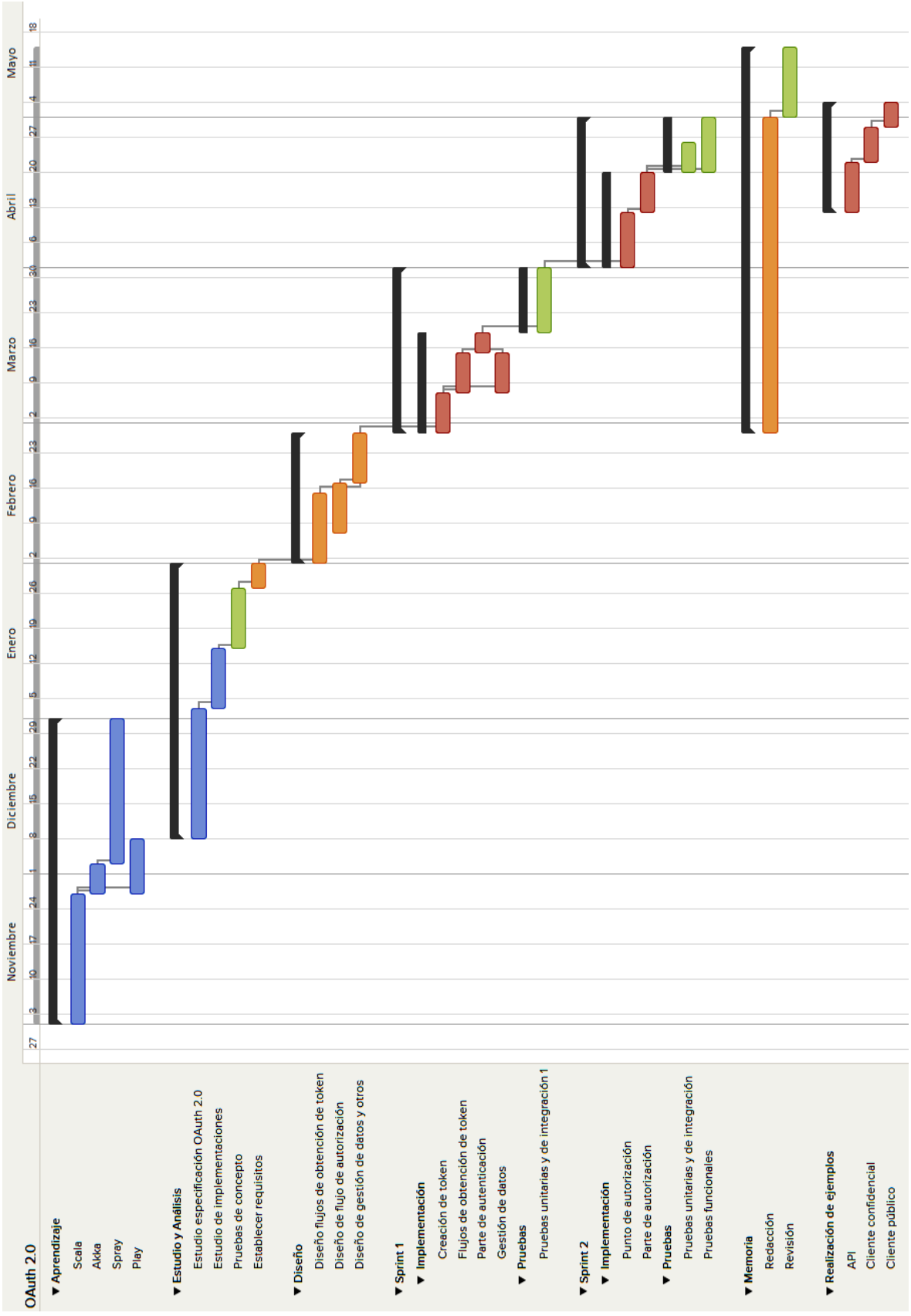
[10] "OAuth 2.0 — OAuth." [Online]. Available: http://oauth.net/2/. [Accessed: 02-Apr-2014].

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995, p. 395.

[12] "Real-World Scala: Dependency Injection (DI) ←." [Online]. Available: http://jonasboner.com/2008/10/06/real-world-scala-dependency-injection-di/. [Accessed: 15-May-2014].

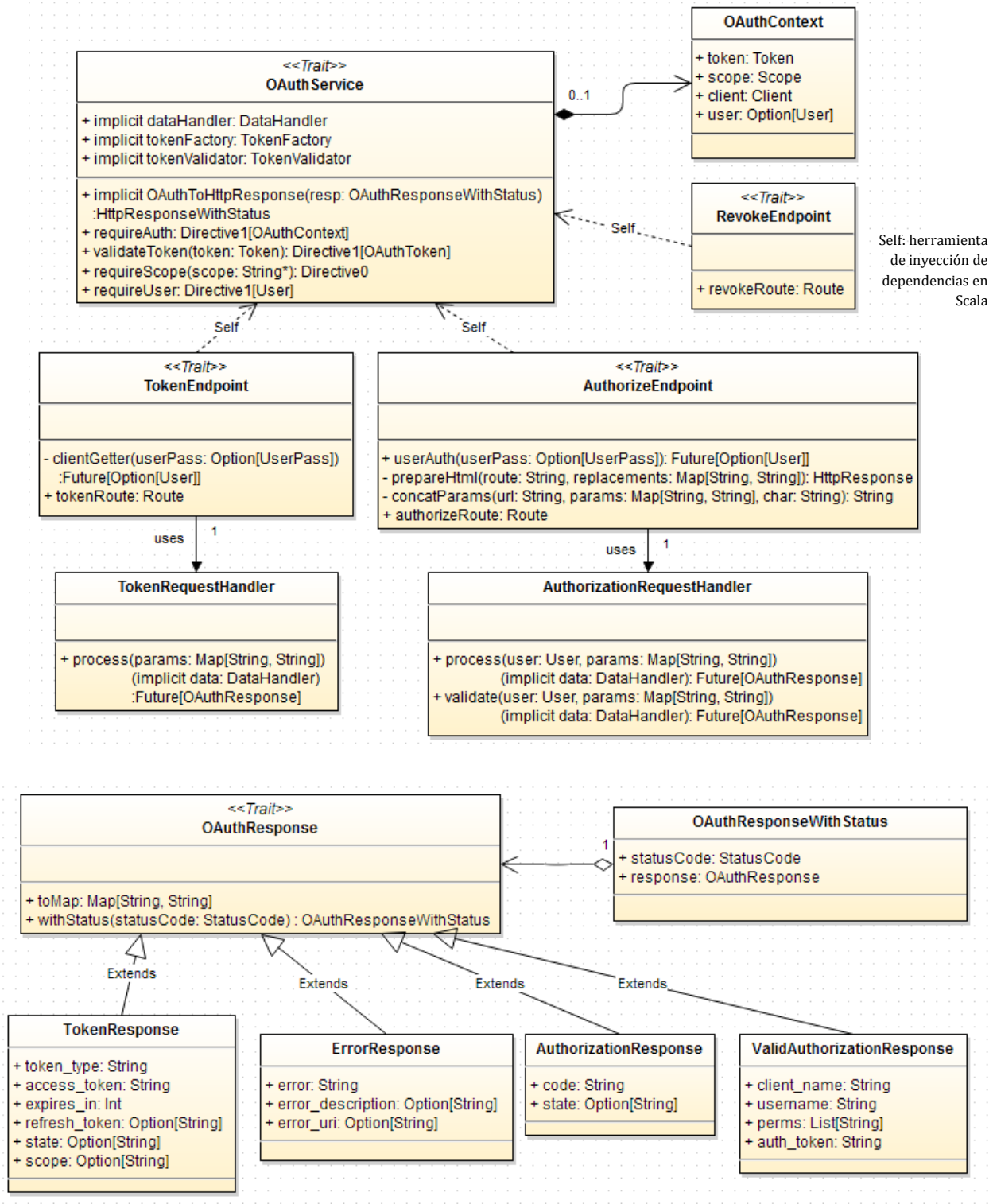
[13] "OAuth 2.0 and the Road to Hell | hueniverse on WordPress.com." [Online]. Available: http://hueniverse.com/2012/07/26/oauth-2-0-and-the-road-to-hell/. [Accessed: 06-May-2014].

ANEXO A. DIAGRAMA DE GANTT



ANEXO B. DIAGRAMA DE CLASES

OAUTH API



Self: herramienta de inyección de dependencias en Scala

Figura 34. Diagrama de clases: OAuth API

OAUTH AUTHORIZER

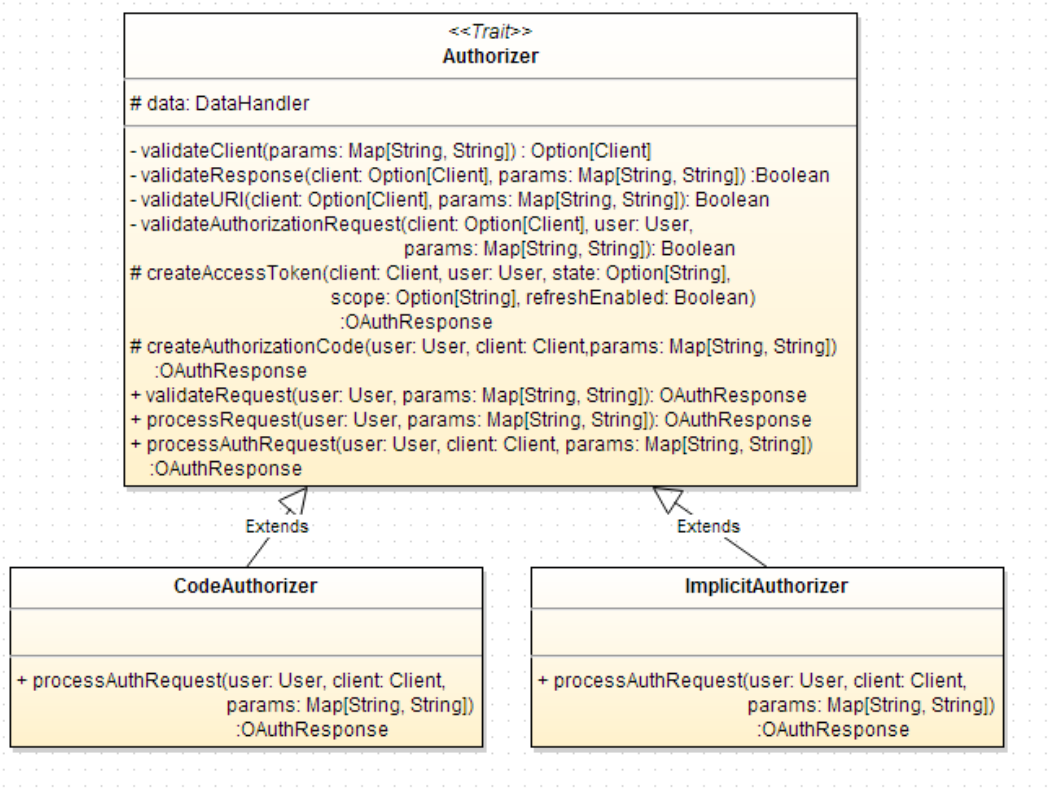


Figura 35. Diagrama de clases: OAuth Authorizer

OAUTH GRANTERS

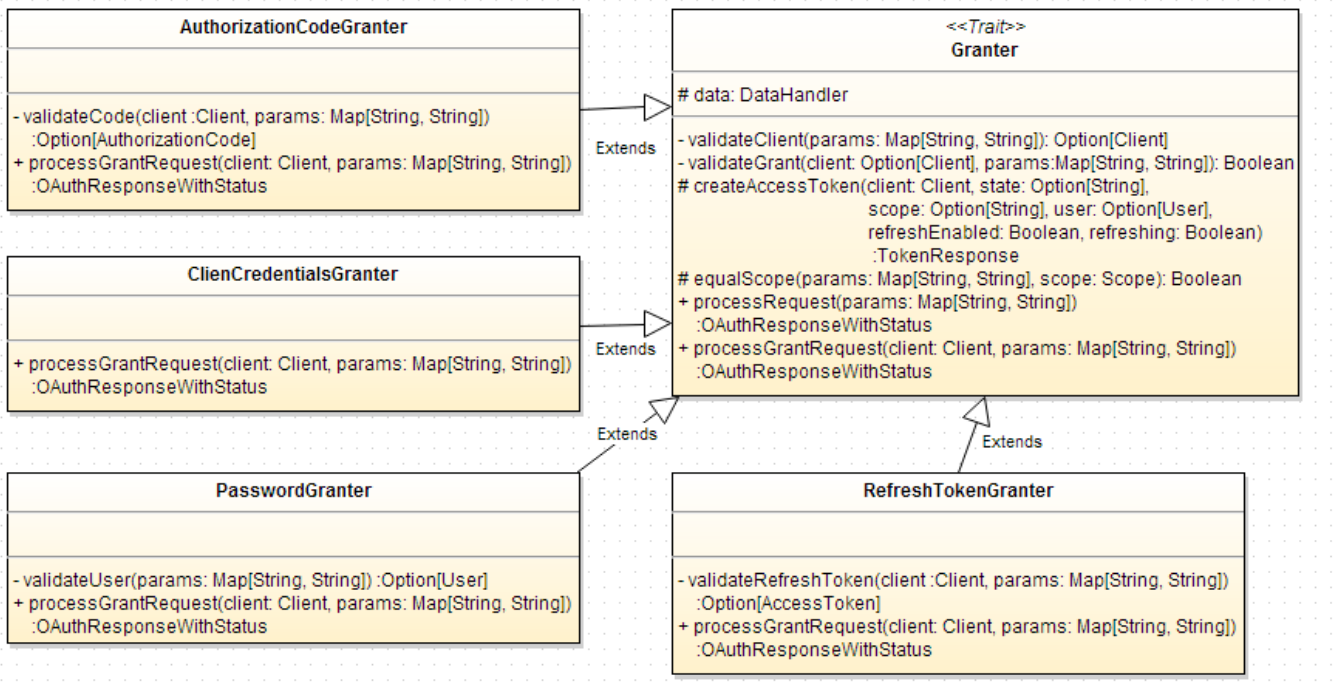


Figura 36. Diagrama de clases: OAuth Granters

OAUTH DATA MODEL

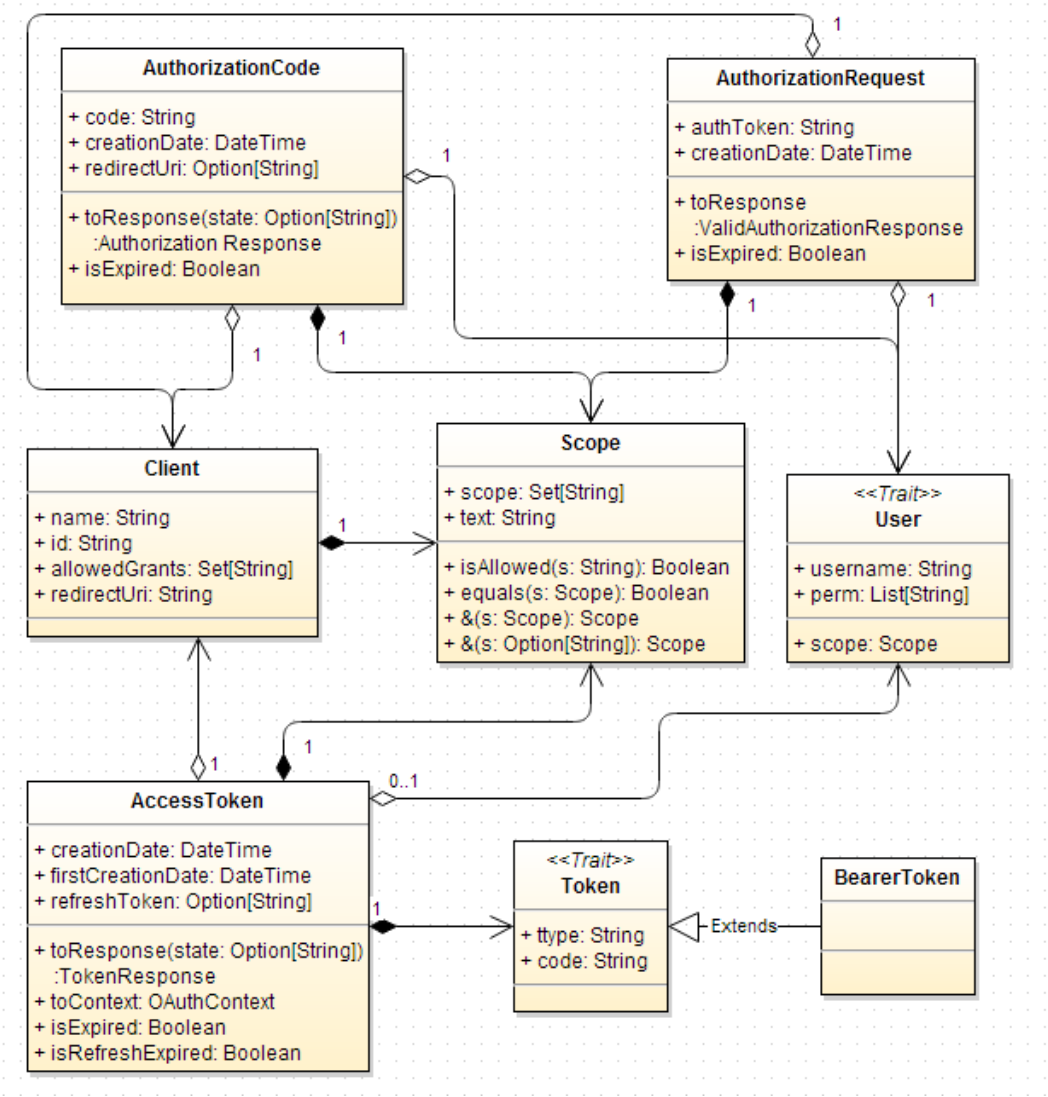


Figura 37. Diagrama de clases: OAuth Data Model

OAUTH UTILS

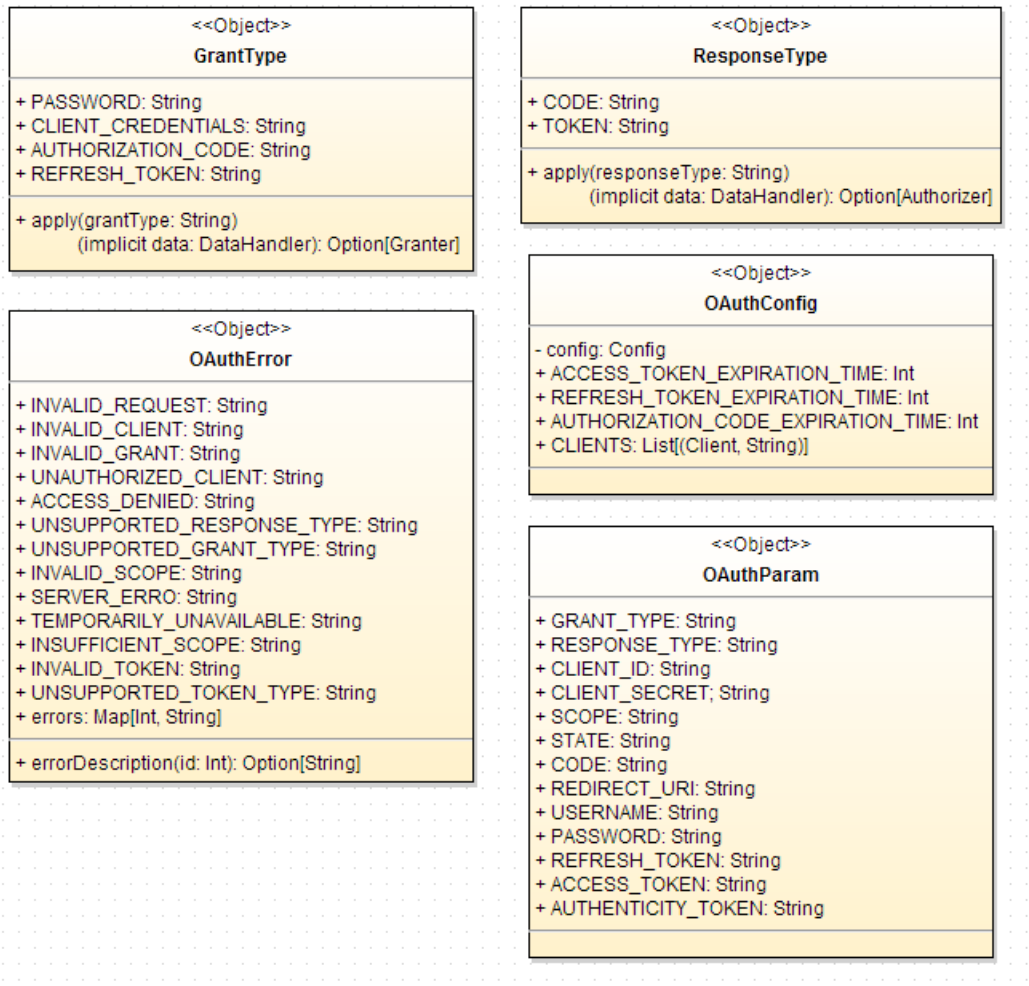


Figura 38. Diagrama de clases: OAuth Utils

OAUTH TRAITS: DATAHANDLER, TOKENVALIDATOR, TOKENFACTORY

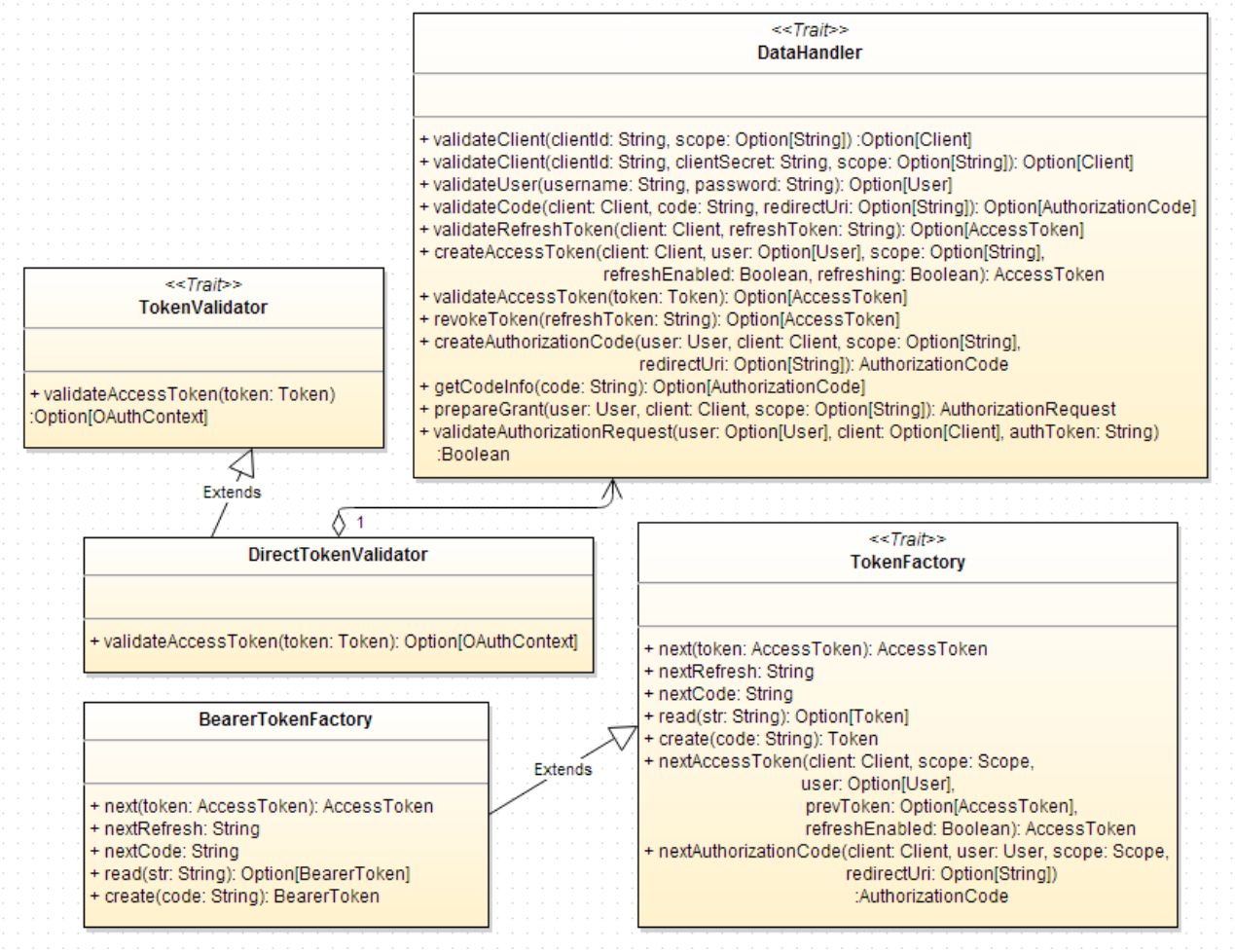


Figura 39. Diagrama de clases: OAuth Traits: DataHandler, TokenValidator, TokenFactory

ANEXO C. DIAGRAMAS DE SECUENCIA

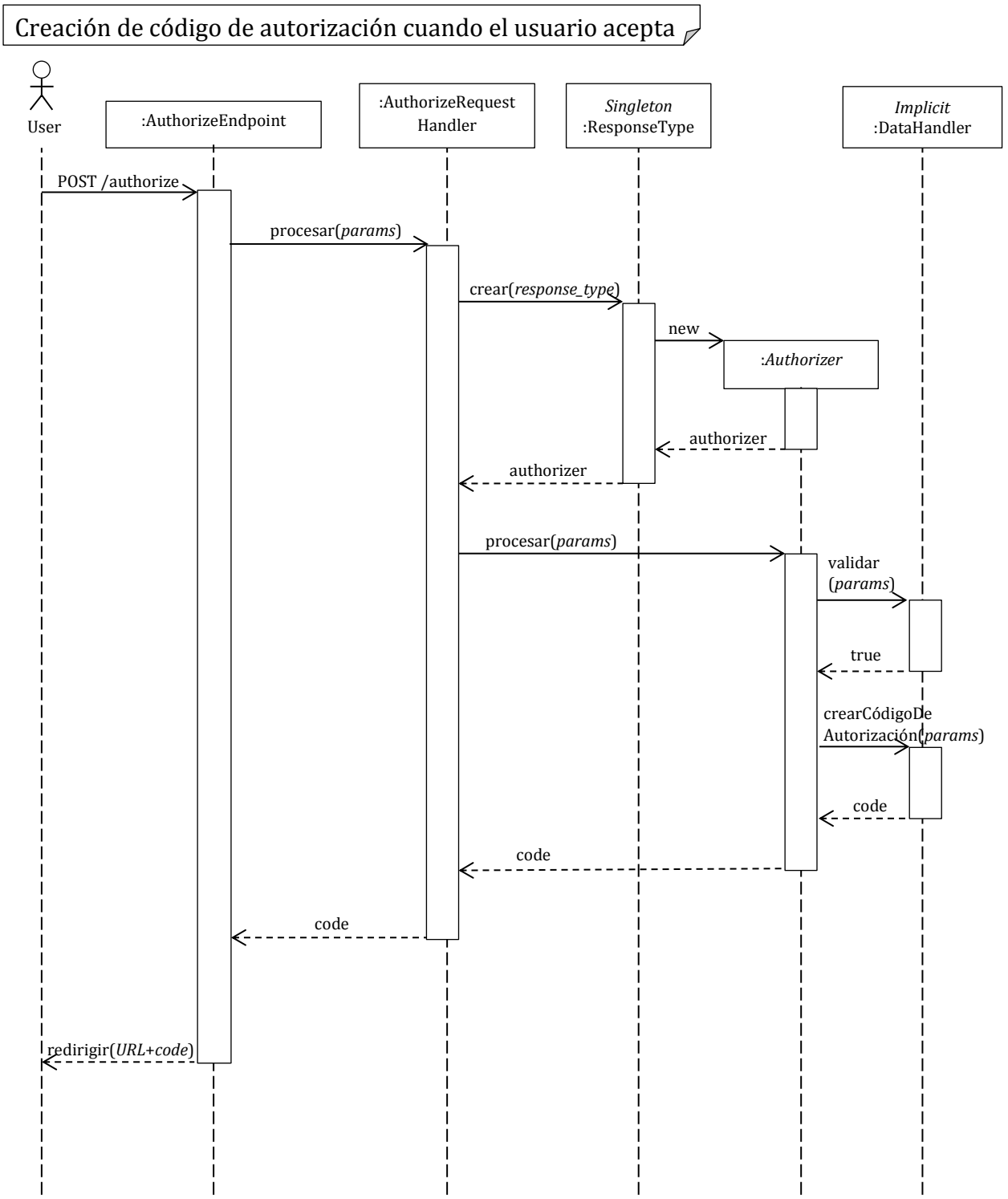


Figura 40. Diagrama de secuencia: creación código de autorización

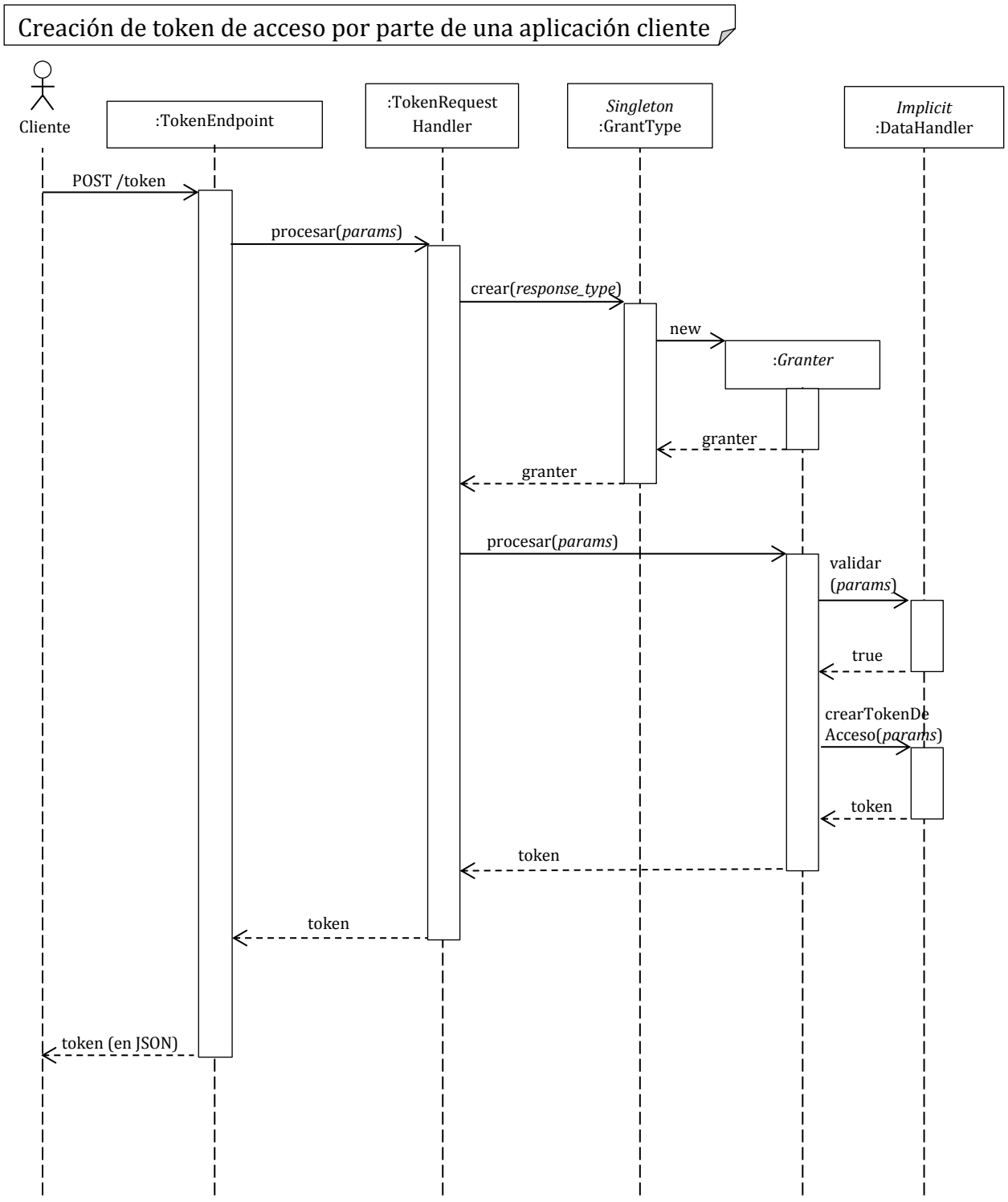


Figura 41. Diagrama de secuencia: creación token de acceso

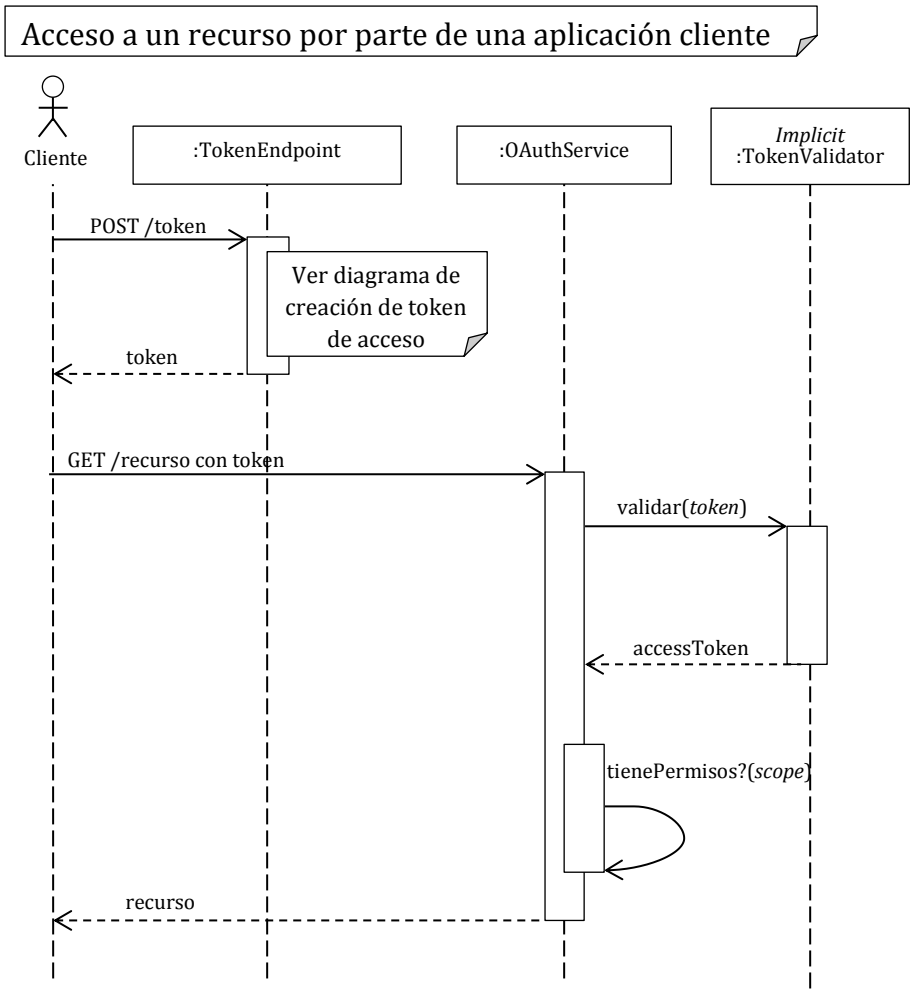


Figura 42. Diagrama de secuencia: acceso a recurso protegido

ANEXO D. PROVEEDORES OAUTH

Proveedor OAuth	Versión
AllPlayers.com	1.0
Amazon	2.0
AOL	2.0
Basecamp	2.0
Bitbucket	1.0a
bitly	2.0
blueKiwi software	2.0
Box	2.0
ciValidator	1.0
cosm	2.0
deviantART	2.0
Deezer	2.0
Discogs	1.0a
Dropbox	1.0, 2.0
Evernote	1.0
Facebook	2.0
Fitbit	1.0
Flickr	1.0a
Formstack	2.0
Foursquare	2.0
GitHub	2.0
Goodreads	1.0
Google	2.0
Google App Engine	1.0a
Groundspeak	1.0
HnyB.me	2.0
Huddle	2.0
Instagram	2.0
Intel Cloud Services	2.0
Jive Software	1.0a, 2.0
LinkedIn	1.0a, 2.0
Microsoft	2.0
Mixi	1.0
Moves	2.0

Proveedor OAuth	Versión
MySpace	1.0a
Netflix	1.0a
OpenLink Data Spaces	1.0
Noosh	2.0
OpenTable	1.0a
PayPal	2.0
Plurk	1.0a
RealPeepz	1.0
Reddit	2.0
Salesforce.com	1.0a, 2.0
SARE	2.0
SensioLabs Connect	2.0
Sina Weibo	2.0
StatusNet	1.0a
Strava	2.0
Stripe	2.0
Tent.io	2.0
Tumblr	1.0a
Twitter	1.0a, 2.0
Ubuntu One	1.0
Unbounce	2.0
Veevop	2.0
Viadeo	2.0
Vimeo	1.0a
VK	2.0
Withings	1.0
Xero	1.0a
XING	1.0
Yahoo!	1.0a
Yammer	2.0
Yandex	2.0
Yelp	1.0a
Zendesk	2.0