ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN

Curso Académico 2010/2011

Proyecto de Fin de Carrera

Librería Cliente del Protocolo de Datos de Google en Qt "Google Data Qt Client"

Autor: Fernando Jiménez Moreno
Tutor: Gregorio Robles Martínez

Resumen

A lo largo de los últimos años, la publicación de APIs se ha ido convirtiendo en una práctica cada vez más usual para *web startups* y otras empresas más tradicionales de la red con el fin de expandir el alcance de sus productos - y, en especial, de sus datos - e introducir con mayor profundidad su impacto en la red.

La práctica de publicar APIs permite que las comunidades web creen una arquitectura abierta para compartir contenido y datos con otros servicios y aplicaciones. De este modo, el contenido creado en un lugar se puede recuperar, colocar y/o actualizar de forma dinámica en varias ubicaciones de la web. Hoy en día es más que común la aparición de un producto web que disponga de un API para que otros productos online puedan integrar sus servicios en nuevas aplicaciones y utilidades. De este modo, las APIs públicas le proporcionan a un producto web la posibilidad de disponer de miles de puntos de presencia en distinto productos, en lugar de en un único lugar.

Google es un gran ejemplo en cuanto a la publicación de APIs se refiere. La mayor parte de los productos ofertados por Google disponen de un API[1] pública para que productos de terceros puedan implementar funcionalidad basada en el acceso a estos servicios y datos publicados. Servicios tan utilizados como Youtube, Maps, Picasa, Blogger y otros muchos, proporcionan APIs que implementan el denominado *Google Data Protocol,* basado en REST, que permite la lectura, escritura y modificación de los datos almacenados en los distintos servicios de Google.

Junto a estas APIs, Google proporciona una serie de librerías cliente que propocionan una capa de abstracción y las herramientas necesarias para interactuar con los servicios de Google a través de su API de datos sin necesidad de tener que construir peticiones HTTP o realizar el procesado a mano de las respuestas de las mismas. Este Proyecto de Fin de Carrera surge con la idea de implementar una de estas librerías cliente del Protocolo de Datos de Google, en un lenguaje aún no empleado para tal fin: C++. Concretamente, se pretende utilizar el framework de desarrollo Qt.

[1] http://code.google.com/more/

Índice general

Capítulo 1

Introducción

En este primer capítulo introductorio, haremos un repaso al marco general de desarrollo de este proyecto y de las tecnologías en él empleadas. Se dará una explicación superficial a cerca del Protocolo Google Data, junto a una breve descripción de las APIs que lo implementan. Del mismo modo, se proporcionará una visión general de las tecnologías REST y el Protocolo Atom, sobre los que está basado Google Data. Finalmente, se presentará una breve introducción sobre el framework de desarrollo Qt.

1.1. Motivación del proyecto

Al inicio de este proyecto, no existía ninguna librería cliente que implementara el Protocolo Google Data en lenguajes de uso común dentro del mundo GNU/Linux. Había clientes para desarrollo web en PHP y JavaScript, clientes para plataformas Microsoft en .NET y, finalmente, un cliente Java para diversos usos. Todos y cada uno de ellos podían ser empleados para el desarrollo de aplicaciones sobre entorno GNU/Linux, sin embargo, ninguno de ellos facilitaba un desarrollo "natural" en esta plataforma como lo habría hecho un cliente en C/C++ o Python, lenguajes más comunes en dicho entorno. Google Data Qt Client se presentaba como una posible solución a este problema.

Google Data Qt Client surge en respuesta a una propuesta de proyecto, basada en las necesidades anteriormente expuestas, para el Google Summer of Code [1] del año 2010 enviada a una de las listas de correo de la comunidad de desarrollo Qt [2]. Dicha propuesta contemplaba la posibilidad de implementar en C++ para Qt una librería cliente para el Protocolo de Datos de Google, desarrollada ya en otros lenguajes y plataformas. A pesar de lo atractivo de la propuesta, ésta no recibió tutorización o aprobación alguna y quedó, por tanto, descartada para la ocasión. Fue entonces cuando, tras comprobar que efectivamente no se iba a poner en marcha la propuesta de proyecto, comencé a diseñar e implementar Google Data Qt Client.

[1] http://code.google.com/soc/
[2] http://groups.google.com/group/qt-gsoc/web/project-ideas?version=6&pli=1

1.2. El Protocolo Google Data

Hoy en día, cualquier persona que haya tenido un mínimo contacto con internet conoce el nombre de Google. Quien más y quien menos ha realizado búsquedas, navegado a través de mapas o leído emails a través de sus servicios.

Google nació de las manos de Larry Page [3] y Sergey Brin [4] que, como resultado de su tesis doctoral y con el objetivo de conseguir información relevante a partir de una importante cantidad de datos, crearon el motor de búsqueda que poco después se convertiría en el buscador de buscadores y principal producto de la empresa.

En palabras del propio gigante informático, la misión de Google es la de organizar la información del mundo, haciendola universalmente accesible y útil. Esto incluye proporcionar acceso a dicha información en contextos más amplios que un simple navegador web y a servicios fuera del entorno de Google. El ***Protocolo Google Data*** **[1]** ofrece un medio seguro para que cualquier desarrollador pueda escribir nuevas aplicaciones para el acceso y modificación de los datos almacenados por muchos de los productos de Google.

El *Protocolo Google Data* es una tecnología basada en REST que permite consultar, añadir y modificar información en la red. Google pone a disposición de los desarrolladores una serie de APIs que implementan su protocolo de satos, a través de las cuales, es posible acceder a los datos de muchos de los servicios ofrecidos por la empresa. Del mismo modo, se ponen a disposición del desarrollador una serie de librerías cliente, como la que se pretende implementar en este Proyecto de Fin de Carrera, para el acceso a dichos servicios en varios lenguajes de programación. Para cada uno de estos lenguajes se proporcionan herramientas y una capa de abstracción con las que construir peticiones y manejar respuestas sin tener que crear peticiones o respuestas HTTP a mano. A día de hoy, Google ha desarrollado librerías cliente para su protocolo de datos en los siguientes lenguajes:

- Java
- JavaScript
- .NET
- PHP
- Python (recientemente desarrollada)
- Objective-C

Existe también una librería cliente para Apex[5] desarrollada por la empresa Developer Force[6].

[3] http://en.wikipedia.org/wiki/Larry_Page
[4] http://en.wikipedia.org/wiki/Sergey_Brin
[5] http://apex.oracle.com/i/index.html
[6] http://wiki.developerforce.com/index.php/Google_Data_API_Toolkit

El Protocolo Google Data utiliza los formatos de publicación Atom 1.0 y RSS 2.0 y, por tanto, está basado en el Protocolo de Publicación Atom (APP), posteriormente analizado. Google Data amplía la funcionalidad de dichos estándares, usando los mecanismos de extensión que estos mismos proporcionan. Los elementos que conforman el protocolo de datos se atienen a los formatos de publicación Atom o RSS. Mientras que el modelo de datos cumple el estándar establecido por el Protocolo de Publicación Atom.

1.2.1. Directorio de APIs

Los siguientes servicios de Google proporcionan APIs que implementan el Protocolo de Datos.

Tabla APIs

A modo de resumen, en la siguiente tabla se muestran las APIs que implementan el Protocolo de Datos de Google.

API	Descripción breve
Google Analytics Data Export API	API de acceso a los datos estadísticos de sitios web
Google Apps API	Acceso a los datos de las aplicaciones cloud de Google
Google Base Data API	Ahora Google Merchant. Permite subir contenido indexable para búsquedas
Blogger Data API	API de acceso al servicio de blogs Blogger
Google Booksearch Data API	API para realizar búsquedas en el repositorio de libros de Google
Google Calendar Data API	API para el acceso a los datos de Google Calendar
Google Code Search API	Permite realizar búsquedas de código fuente de los repositorios de Google
Google Contacts API	Proporciona acceso a los datos de los contactos de un usuario
Google Document List API	Permite acceder y manipular documentos almacenados por el usuario
Google Project Hosting Issue Tracker API	Permite añadir, ver y modificar issues en el sistema de bugs de Google
Google SideWiki Data API	Permite modificar la barra lateral de Google en Firefox
Google Site Data API	Permite manejar el contenido del CMS para crear web de Google
Google Spreadsheet Data API	Proporciona acceso a las hojas de cálculo almacenadas por el usuario
Google Translator Toolkit Data API	Actualmente cerrado. Acceso al traductor de Google
Google Webmaster Tools Data API	Herramientas para desarrolladores de sitios web
Youtube Data API	Acceso a los contenidos del servicio de streaming de video Youtube

Google Analytics Data Export API.

Google Analytics[7] es una solución de analítica web para empresas que proporciona información muy valiosa sobre el tráfico del sitio web y la eficacia del plan de marketing. Gracias a unas funciones potentes, flexibles y fáciles de usar, permite ver y analizar el tráfico desde una perspectiva totalmente distinta. Es útil a la hora de diseñar anuncios más orientativos o mejorar las iniciativas de marketing de sitios webs.

[7] http://www.google.com/analytics/

Google Apps API

Google Apps[8]es un conjunto de aplicaciones en la nube que incorpora un gestor de correo basado en gmail, un calendario, gdocs, sites, y la posibilidad de añadir aplicaciones de terceros. En definitiva es un contenedor de aplicaciones muy útiles para cualquier empresa, con la que resulta muy fácil establecer vínculos de colaboración entre los empleados.

Google Base Data API

Google Base[9] es un servicio gratuito para enviar todo tipo de contenidos a Google para que los almacene y los haga localizables online. Permite a lo proveedore de contenidos subir información estructurada a Google, encontrarla a través de las propiedades de búsqueda, y publicarla a través de apis, gadgets y anuncios. En Junio de 2011, y por tanto, para la lectura de este PFC, Google Base pasará a ser *deprecated*, para dar paso definitivo a Google Merchant[10].

Blogger API

Blogger[11]es un servicio creado por Pyra Labs [12] y, posteriormente, adquirido por Google, para crear y publicar una bitácora en línea. El usuario no tiene que escribir ningún código o instalar programas de servidor o de scripting. Blogger acepta para el alojamiento de las bitácoras su propio servidor (Blogspot) o el servidor que el usuario especifique (FTP o SFTP).

Google Booksearch Data API

Google Books[13] (previously known as Google Book Search and Google Print) is a service from Google that searches the full text of books that Google has scanned, converted to text using optical character recognition, and stored in its digital database. Results from Google Book Search show up in both general web search at google.com and through the dedicated Google Books site (books.google.com). Up to three results from the Google Books index may be displayed, if relevant, above other search results in the Google Web search service (google.com).

Google Calendar Data API

Google Calendar[14], cuyo nombre código anterior era CL2, es una agenda y calendario electrónico desarrollado por Google. Permite sincronizarlo con los contactos de Gmail de manera que podamos invitarlos y compartir eventos.

[8] http://www.google.com/apps/intl/es/business/index.html
[9] http://www.google.com/base/
[10] http://www.google.es/merchants
[11] http://blogger.com/
[12] http://www.pyra.com/
[13] http://books.google.com/
[14] http://www.google.com/calendar

Google Code Search Data API

Google Code Search[15] permite buscar código fuente de acceso público alojado en Internet con el fin de encontrar definiciones de funciones y código muestra. Entre otras funciones permite:

- Utilizar expresiones regulares para realizar búsquedas más precisas.
- Restringir la búsqueda por lenguaje, licencia o nombre de archivo.
- Ver el archivo fuente con vínculos que enlazan con todo el paquete y con la página web de donde proviene.

Google Contacts Data API

El API de datos de Google Contacts permite a las aplicaciones cliente ver y actualizar contenido de Google Contacts como feeds del API de datos de Google. La aplicación cliente puede solicitar una lista de contactos de un usuario, así como editar, eliminar y consultar contenido de contactos existentes. Entre otras funciones, Google Contacts permite:

- Sincronizar contactos de Google con contactos de un dispositivo móvil.
- Mantener relaciones entre usuarios en aplicaciones sociales.
- Ofrecer a los usuarios la posibilidad de comunicarse directamente con sus amigos desde aplicaciones externas a través de teléfono, correo electrónico y mensajería instantánea.

Google Document List Data API

El API de Google Documents[16] permite a las aplicaciones cliente pedir una lista con los documentos del usuario, consultar el contenido de los mismos, subir o descargar documentso de la lista, modificar los permisos de compartición, ver el historial de revisiones y almacenar ficheros en carpetas.

Google Finance Portfolio Data API

Google Finance[17] es un espacio de Google dedicado exclusivamente a mostrar todo tipo de información relacionada con el mundo financiero: cotizaciones bursátiles, datos económicos de las empresas y noticias relacionadas con ellas, provenientes de diferentes fuentes (medios tradicionales, blogs o foros de discusión). Los datos de cotizaciones provienen de los mercados estadounidenses Nasdaq, AMEX y NYSE, los canadienses TSE, TSX, y el europeo Euronext.

El API de Google Finance Portfolio permite a las aplicaciones cliente consultar y actualizar contenido financiero en forma de feeds de Google Data. Los clientes pueden usar el API de Google

[15] http://www.google.com/codesearch
[16] http://docs.google.com
[17] http://www.google.com/finance

Finance para crear nuevos portfolios y entradas de transacciones, solicitar una lista de dichas entradas, modificarlas o borrarlas.

Google Health Data API

Google Health[18] es un servicio de información personal centralizado enfocado a la sanidad (también conocido como Historial Clínico Electrónico de Google). El servicio permite a los usuarios de Google registrar voluntariamente su Historial Clínico, ya sea manualmente o con la cuenta de los servicios sanitarios asociados al sistema de Google Health, con lo que se permite la fusión de los historiales médicos, que puedan estar dispersos, en un único perfil de Google Health centralizado. La información voluntariamente añadida puede incluir condiciones de salud, medicamentos, alergias y resultados de laboratorio. Una vez introducido, Google Health usa la información para proporcionar al usuario un registro clínico centralizado, información sobre las condiciones y las posibles contraindicaciones entre medicamentos, condiciones y alergias.

El API de Google Health Data permite a las aplicaciones cliente consultar y actualizar el contenido de Google Health como feeds de Google Data. Dependiendo del tipo de aplicacion (web o instalada), se puede postear, editar o borrar contenido en el perfil de un usuario, obtener una lista de las entradas existentes y realizar consultas de información específica sobre temas de salud.

Picasa Web Albums Data API

Picasa Web Albums es un servicio de Google que permite subir y compartir fotos a través de un sitio web.

El API de Picasa Web Albums permite a páginas web y otras aplicaciones integrarse con Picasa, crear albums, subir y descargar imágenes, comentar en fotografías, etc. Entre otras muchas aplicaciones, el API de Picasa permitido a algunos desarrolladores:

- Crear aplicaciones para fácilmente subir fotos desde dispositivos, aplicaciones de escritorio y otros servicios web.
- Crear clientes móviles con toda la funcionalidad para navegar y subir contenido a Picasa Web Album.
- Integrar Picasa Web Album con sistemas de blogging para permitir mostrar fácilmente álbumes y fotografías.
- Usar Picasa Web Album para proveer de imágenes a marcos digitales.

Google Project Hosting Issue Tracker API

Google Project Hosting proporciona un servicio de hosting open source rápido y fiable con las siguientes características:

[18] http://www.google.com/health/

- Creación de proyectos instantánea sobre cualquier tema.
- Hosting Subversion y Mercurial con 2 gigabytes de espacio y servicio de descargas con el mismo espacio en disco.
- Navegador de código integrado y herramientas de revisión de código.
- Un gestor de bugs y una wiki para cada proyecto.
- Permite marcar y seguir la evolución de proyectos publicados por otros.

El API del gestor de incidencias de Google Project Hosting permite a aplicaciones cliente consultar y modificar bugs en forma de feeds de Google Data. Permite crear nuevas incidencias y comentarios sobre las mismas, obtener una lista de todos los bugs, pedir comentarios sobre una incidencia en concreto, editar incidencias existentes y consultar bugs según unos serie de criterios de búsqueda.

Google Sidewiki Data API

Google Sidewiki es una barra lateral del navegador que permite ver, añadir y compartir comentarios en cualquier página de Internet. Está disponible como una función de la barra de Google. Actualmente, Sidewiki sólo está disponible en la barra Google para Firefox.

El API de Google Sidewiki permite a la aplicaciones clientes ver contenido de Google Sidewiki en forma de feeds de Google Data, permitiendo obtener una lista de entradas, específicas de una web o un autor, y consultar dicha lista según unos criterios de búsqueda particulares.

Google Sites Data API

Google Sites es una aplicación online gratuita que permite crear un sitio web o una intranet de una forma tan sencilla como editar un documento. Con Google Sites los usuarios pueden reunir en un único lugar y de una forma rápida información variada, incluidos vídeos, calendarios, presentaciones, archivos adjuntos y texto. Además, permite compartir información con facilidad para verla y editarla por un grupo reducido de colaboradores o con toda su organización, o con todo el mundo.

El API de Google Sites permite:

- Obtener, crear, modificar y borrar páginas y contenidos.
- Subir y descargar attachments.
- Revisar el historial de versiones de un sitio web.
- Mostrar la información de actividad reciente de un usuario.

Google Spreadsheets Data API

Se trata de un servicio vía web de hojas de cálculo, realizado en tecnología AJAX. Con ella se puede realizar la mayoría de las funciones que dejan las aplicaciones de hojas de cálculos de los programas ofimáticos, como realizar operaciones entre celdas con diferentes tipos de funciones (matemáticas, financieras, lógicas, de fechas, de búsquedas, estadísticas, con cadenas e informativas), ordenar columnas, manejar diferentes hojas dentro de cada fichero, manejas ficheros del tipo xls y csv, etc. Aunque no ofrece la funcionalidad de realizar gráficas a partir de los datos de las tablas.

El API de Google Spreadsheets permite acceder y manipular la información almacenada en Google Spreadsheets. Permite obtener las hojas de cálculo que concuerdan con un ID o un título específico, tratar una hoja de trabajo como una lista de filas, manejar y explorar las hojas de trabajo dentro de una hoja de cálculo, etc.

Google Webmaster Tools Data API

Google Webmaster Tools permite a los creadores de páginas web comprobar el estado de la indexación de sus sitios en internet por el buscador y optimizar su visibilidad. Entre otras funciones, permite la creación y emisión de *sitemaps*, la comprobación y ajuste de la frecuencia de indexación, la enumeración de enlaces de páginas internas y externas al sitio web, etc. El API de datos de Google Webmaster Tools proporciona, a grandes rasgos, herramientas para visualizar sitios webs en la cuenta de Webmaster Tools del usuario, añadir y eliminar webs de la misma, verificar su propiedad y enviar y eliminar *sitemaps*.

Youtube Data API

YouTube es un sitio web en el cual los usuarios pueden subir y compartir vídeos. Fue creado por tres antiguos empleados de PayPal en febrero de 2005. En noviembre de 2006 Google Inc. lo adquirió por 1650 millones de dólares, y ahora opera como una de sus filiales. YouTube usa un reproductor en línea basado en Adobe Flash para servir su contenido. Es muy popular gracias a la posibilidad de alojar vídeos personales de manera sencilla. Aloja una variedad de clips de películas, programas de televisión, vídeos musicales, a pesar de las reglas de YouTube contra subir vídeos con derechos de autor, este material existe en abundancia, así como contenidos amateur como videoblogs. Los enlaces a vídeos de YouTube pueden ser también puestos en blogs y sitios electrónicos personales usando API o incrustando cierto código HTML.

El API de datos de Youtube permite a aplicaciones cliente realizar muchas de las operaciones disponibles en la web de Youtube. Es posible realizar búsquedas de videos, obtener feeds estándar, y ver contenido relacionado. Una aplicación puede incluso autenticarse como usuario para subir videos, modificar las listas de reproducción de los usuarios y realizar otras acciones sobre su cuenta.

1.3. Servicios web RESTful

Tal y como se indicó anteriormente, el Protocolo de Datos de Google está construido alrededor de la arquitectura software REST.

La **Transferencia de Estado Representacional** (Representational State Transfer) o **REST** [5] es una técnica de arquitectura software para sistemas hipermedia distribuidos como la World Wide Web. El término se originó en el año 2000, en una tesis doctoral sobre la web escrita por Roy Fielding[19], uno de los principales autores de la especificación del protocolo HTTP y ha pasado a ser ampliamente utilizado por la comunidad de desarrollo.

Si bien el término *REST* se refería originalmente a un conjunto de principios de arquitectura —descritos más abajo—, en la actualidad se usa en el sentido más amplio para describir cualquier interfaz web simple que utiliza XML y HTTP, sin las abstracciones adicionales de los protocolos basados en patrones de intercambio de mensajes como el protocolo de servicios web SOAP. Es posible diseñar sistemas de servicios web de acuerdo con el estilo arquitectural REST de Fielding y también es posible diseñar interfaces XMLHTTP de acuerdo con el estilo de llamada a procedimiento remoto pero sin usar SOAP[20]. Estos dos usos diferentes del término REST causan cierta confusión en las discusiones técnicas, aunque RPC[21] no es un ejemplo de REST.

Los sistemas que siguen los principios REST se llaman con frecuencia *RESTful*; los defensores más acérrimos de REST se llaman a sí mismos *RESTafaris*.

REST afirma que la web ha disfrutado de escalabilidad como resultado de una serie de diseños fundamentales clave:

- Un **protocolo cliente/servidor sin estado**: cada mensaje HTTP contiene toda la información necesaria para comprender la petición. Como resultado, ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre mensajes. Sin embargo, en la práctica, muchas aplicaciones basadas en HTTP utilizan cookies y otros mecanismos para mantener el estado de la sesión (algunas de estas prácticas, como la reescritura de URLs, no son permitidas por REST).

- Un conjunto de **operaciones bien definidas** que se aplican a todos los recursos de información: HTTP en sí define un conjunto pequeño de operaciones, las más importantes son **POST, GET, PUT y DELETE**. Con frecuencia estas operaciones se equiparan a las operaciones CRUD que se requieren para la persistencia de datos, aunque POST no encaja exactamente en este esquema.

- Una **sintaxis universal** para identificar los recursos. En un sistema REST, cada recurso es direccionable únicamente a través de su URI.

- El **uso de hipermedios**, tanto para la información de la aplicación como para las transiciones de estado de la aplicación: la representación de este estado en un sistema REST

[19] http://es.wikipedia.org/wiki/Roy_Fielding
[20] http://en.wikipedia.org/wiki/SOAP
[21] http://en.wikipedia.org/wiki/Remote_procedure_call

son típicamente HTML o XML. Como resultado de esto, es posible navegar de un recurso REST a muchos otros, simplemente siguiendo enlaces sin requerir el uso de registros u otra infraestructura adicional.

1.4. El Protocolo de Publicación Atom

El Protocolo de Publicación Atom (AtomPub) [4, 8, 2] es un protocolo a nivel de aplicación para la publicación y edición de recursos web utilizando HTTP [RFC2616] y XML 1.0 [REC-xml]. El protocolo soprta la creación de recursos web y provee de medios para:

- Colecciones: sets de recursos, que pueden ser obtenidos como un total o por partes:
- Servicios: Descubrimiento y descripción de colecciones.
- Edición: Creación, edición y borrado de recursos.

El Protocolo de Publicación Atom será estudiado más profundamente como anexo a esta memoria.

1.5. Qt Framework

Qt [7, 6]es un framework para el desarrollo de aplicaciones multiplataforma creado por la compañía noruega Trolltech (anteriormente **Q**uasar **T**echnologies, de ahí el nombre) y que actualmente es propiedad de Nokia y Digia [22], esta última poseedora de la licencia comercial de Qt desde Marzo del 2011 y, previsiblemente, futura poseedora del total de la plataforma y principal responsable de la continuación del desarrollo y mantenimiento de la misma.

Qt, por sus inicios y sencillez, es mayormente conocida como herramienta para el desarrollo rápido de interfaces gráficas, sin embargo, su funcionalidad no se limita ni mucho menos a esta tarea, proporcionando un amplio abanico de librerías para el desempeño de otras muchas tareas como el manejo de sockets, programación multithread, comunicación con bases de datos, programación de sistemas embebidos y un largo etcétera.

Qt utiliza C++ de manera nativa, pero ofrece soporte para otros lenguajes en forma de *bindings* tales como Python mediante PyQt [23], Java con QtJambi [24] o C# con Qyoto [25].

1.5.1. *Code less, Create more, Deploy everywhere*

Bajo este eslogan, el propósito de Qt, semejante al de otras plataformas como .NET de Microsoft, y su alternativa libre Mono, o Swing de Java, es permitir a los desarrolladores construir

[22] http://www.digia.com/C2256FEF0043E9C1/0/405002251
[23] http://www.riverbankcomputing.co.uk/software/pyqt/intro
[24] http://qt-jambi.org/
[25] http://techbase.kde.org/Development/Languages/Qyoto

aplicaciones multiplataforma a partir de una misma base de código de manera rápida y sencilla, añadiendo la gran ventaja de la eficiencia, notablemente mayor, que proporciona el código nativo C++.

1.5.2. Módulos

Qt está compuesto por una serie de módulos que proveen funcionalidad específica a través de una biblioteca de clases multiplataforma. Aunque también existen algunos módulos específicos para cada plataforma, por ejemplo, *QtDBUS* para comunicación entre procesos, exclusiva de Unix o *QtAxContainer* y *QtAxServer* para construir y utilizar componentes *ActiveX*, exclusiva de Windows. Alguno de los módulos que componen Qt son:

- El módulo ***QtCore*** contiene el núcleo de las clases *non-GUI*, es decir, aquellas no usadas específicamente para el desarrollo de interfaces gráficas, inluyendo el bucle de eventos y el mecanismo de *signals* y *slots* de Qt. Este módulo incluye igualmente abstracciones independientes de la plataforma para Unicode, threads, mapeo de ficheros, memoria compartida, expresiones regulares y configuraciones de usuario y aplicación.

- El módulo **QtGui** contiene la mayor parte de las librerías dedicadas a la creación de interfaces gráficas. Entre ellas se incluyen una serie de clases tabla, árbol y lista basadas en el patrón de diseño Modelo Vista Controlador. También incluye un sofisticado *widget canvas* 2D capaz de almacenar miles de elementos, incluyendo otros widgets ordinarios.

- El módulo **QtMultimedia** implementa funcionalidad multimedia a bajo nivel.

- El módulo **QtNetwork**, ámpliamente usado en el desarrollo de este Proyecto de Fin de Carrera, contiene clases para implementar clientes y servidores UDP y TCP. Incluye, también, clases que implementan clientes HTTP y FTP y que soporta búsquedas DNS. Los eventos de red están integrados con el bucle de eventos, facilitando el desarrollo de aplicaciones de red.

- El módulo **QtOpenGL** contiene clases que permiten el uso de OpenGL [26] para el renderizado gráfico en 3D.

- El módulo **QtOpenVG** es un plugin que proporciona soporte para OpenVG [27].

- El módulo **QtScript** es un motor de scripting basado en Emacs[28].

- El módulo **QtScriptTools** proporciona componentes adicionales para aplicaciones que hagan uso de QtScript.

[26] http://www.opengl.org/
[27] http://www.khronos.org/openvg/
[28] http://www.gnu.org/software/emacs/

- El módulo **QtSql** contiene clases que integran con bases de datos open-source y propietarias. Incluye un modelo de datos editable para tablas de bases de datos que pueden ser usadas con clases GUI. Del mismo modo, incluye una implementación de SQLite [29].
- El módulo **QtSvg** contiene clases para mostrar el contenido de ficheros SVG. Soporta las características estáticas de SVG 1.2 Tiny.
- El módulo **QtWebKit** proporciona un motor de navigación web basado en WebKit [30] del mismo modo que provve de clases para el renderizado y la interacción con contenido web.
- El módulo **QtXml** implementa las interfaces SAX [31] y DOM [32] para el parser XML del Qt.
- El módulo **QtXmlPatterns** proporciona soporta para la validación de XPath [33], XQuery [34], XSLT [35] y XML Schema [36].
- El API multimedia **Phonon** provee un control multimedia simple.
- El módulo **Qt3Support** proporciona clases que permiten portar fácilmente aplicaciones de Qt3 a Qt4.
- El módulo **QtDeclarative** es un framework declarativo para construir interfaces gráficas fluidas en QML.
- El módulo **QtDBus**, específico de sistemas Unix, es una librería para la comunicación entre procesos (IPC) usando el protocolo D-Bus [37].
- El módulo **QAxContainer**, específica de plataformas Windows, es una extensión para acceso a controles ActiveX y objetos COM.
- El módulo **QAxServer** es una librería estática con la que convertir un binario Qt en un servidor COM. También específica de Windows.

1.5.3. Qt en la industria

Qt se utiliza en una amplia variedad de dispositivos y es empleado en los desarrollos de importantes empresas como Google, Intel o Dreamworks.

La lista de aplicaciones desarrollada con Qt (y ahora también con QML) es interminable. Entre las más notables podríamos mencionar Google Earth, Skype, VirtualBox, VLC, Last.fm e

[29] http://www.sqlite.org/
[30] http://www.webkit.org/
[31] http://www.saxproject.org/
[32] http://www.w3.org/DOM/
[33] http://www.w3.org/TR/xpath/
[34] http://www.w3.org/XML/Query/
[35] http://www.w3.org/TR/xslt
[36] http://www.w3.org/XML/Schema
[37] http://www.freedesktop.org/wiki/Software/dbus

incluso el editor con el que se está escribiendo este documento, LyX. Mención especial en esta lista de ejemplos de desarrollo con Qt tienen el entorno de escritorio KDE, con toda su suite de aplicaciones (Kopete, KOffice, Konqueror y un larguísimo etcetera) y el sistema operativo MeeGo.

1.5.4. Licenciamiento

En todo momento, Qt ha estado disponible bajo licencia comercial que permite el desarrollo de aplicaciones propietarias sin restricciones de licencia. Además de eso, Qt ha ido gradualmente incrementando el número de licencias libres bajo las que aparece disponible. Hoy por hoy, Qt está disponible bajo licencia LGPL (GNU Lesser General Public License), que permite su uso tanto en software libre como en propietario.

Hasta la versión 1.45, el código fuente de Qt era liberado bajo licencia FreeQt, lo cual no estaba acorde con el principio de código libre de la Open Source Initiative [38] ni con la definición de software libre de la Free Software Foundation (FSF) [39], porque, a pesar de tener el fuente disponible, no permitía su redistribución modificada.

La controversia surgió alrededor de 1998 cuando quedó claro que la KDE Software Compilation [40] iba a convertirse en una de las principales plataformas de escritorio del sistema operativo Linux. Al estar basada en Qt, miembros del mundo del software libre comenzaron a temer que una de las piezas esenciales de su sistema operativo mayoritario podría ser propertaria.

Con el lanzamiento de la version 2.0 del framework, la licencia cambio a Q Public License (QPL) [41], una licencia de software libre, pero considerada por la FSF como incompatible con la GPL. KDE y Trolltech se comprometieron a que Qt no fuese licenciada bajo términos más restrictivos que QPL, incluso en el caso de que Trolltech fuese comprada o quebrase. Lo cual conllevó a la creación de la KDE Free Qt Foundation [42], que garantizaba que Qt sería liberado bajo licencia BSD o similar en los casos anteriormente mencionados de compra o quiebra por parte de Trolltech.

En 2002, miembros de KDE en el proyecto Cygwin comenzaron a portar el código de Qt/X11, licenciado GPL, a Windows, en respuesta a la negativa por parte de Trolltech de licenciar Qt/Windows bajo licencia GPL basándose en que Windows no era software libre. Dicho código adquirió un éxito razonable aunque nunca alcanzo las suficiente calidad para ponerlo en producción.

Este tema fue resuelto por Trolltech cuando, en Junio de 2005, liberó Qt/Windows 4 bajo licencia GPL. Qt4 ahora soporta el mismo abanico de plataformas en la versión libre y en la versión propietaria, con lo que es posible crear aplicaciones con licencia libre GPL usando Qt en todas las plataformas soportadas. Posteriormente se introdujo una excepción a la licencia GPLv3

[38] http://www.opensource.org/
[39] http://www.fsf.org/
[40] http://www.kde.org/community/whatiskde/softwarecompilation.php
[41] http://en.wikipedia.org/wiki/Q_Public_License
[42] http://www.kde.org/community/whatiskde/kdefreeqtfoundation.php

[43] que permitía licenciar aplicaciones bajo varias licencias libres incompatibles con GPL, tales como la Mozilla Public License [44].

Tal y como se anunció en Enero del 2009, la versión 4.5 de Qt añade otra opción, la licencia LGPL, que hace a Qt incluso más atractiva para proyectos de código cerrado.

[43] http://doc.trolltech.com/4.4/license-gpl-exceptions.html
[44] http://www.mozilla.org/MPL/MPL-1.1.html

Capítulo 2

Objetivos

Una vez presentado el marco general de desarrollo de este proyecto y las tecnologías en él empleadas, en este capítulo abordaremos los objetivos que se han perseguido durante la elaboración del proyecto.

2.1. Descripción del problema

Al inicio de este proyecto, no existía ninguna librería cliente que implementara el Protocolo Google Data en lenguajes de uso común dentro del mundo GNU/Linux. Había clientes para desarrollo web en PHP y JavaScript, clientes para plataformas Microsoft en .NET y, finalmente, un cliente Java para diversos usos. Todos y cada uno de ellos podían ser empleados para el desarrollo de aplicaciones sobre entorno GNU/Linux, sin embargo, ninguno de ellos facilitaba un desarrollo "natural" en esta plataforma como lo habría hecho un cliente en C++ o Python, lenguajes más comunes en dicho entorno. Google Data Qt Client se presentaba como una posible solución a este problema.

Google Data Qt Client surge en respuesta a una propuesta de proyecto, basada en las necesidades anteriormente expuestas, para el Google Summer of Code [1] del año 2010 enviada a una de las listas de correo de la comunidad de desarrollo Qt [2]. Dicha propuesta contemplaba la posibilidad de implementar en C++ para Qt una librería cliente para el Protocolo de Datos de Google, desarrollada ya en otros lenguajes y plataformas. A pesar de lo atractivo de la propuesta, ésta no recibió tutorización o aprobación alguna y quedó, por tanto, descartada para la ocasión. Fue entonces cuando, tras comprobar que efectivamente no se iba a poner en marcha la propuesta de proyecto, comencé a diseñar e implementar Google Data Qt Client.

[1] http://code.google.com/soc/
[2] http://groups.google.com/group/qt-gsoc/web/project-ideas?version=6&pli=1

2.2. Objetivo principal

Una vez identificada la problemática, fijé como objetivo principal de mi Proyecto Final de Carrera el diseño e implementación en C++, utilizando el framework de desarrollo Qt, de una librería cliente REST para el acceso a las APIs que implementan el Protocolo de Datos de Google.

2.3. Requisitos

Dada la magnitud de la tarea, de cara a mi Proyecto de Fin de Carrera, decidí acotar los requisitos para el mismo en los siguientes puntos:

- Diseño e implementación del núcleo de la librería cliente basada en REST para el acceso a las APIs de los servicios de Google que implementan su Protocolo de Datos.
- Diseño e implementación de los clientes para dos de las APIs que implementan dicho Protocolo de Datos empleando el núcleo anteriormente desarrollado.
- Desarrollo de uno o varios demostradores del uso de la librería.
- Documentación Doxygen[3] del código fuente. Al menos de la parte pública del API de la librería y en inglés. Así como la elaboración de los diagramas UML de herencia y colaboración.
- Desarrollo de test unitarios para verificar la consistencia y buen funcionamiento de la librería desarrollada. En un principio se intentó seguir la metodología de desarrollo TDD (Test Driven Development) [4], pero finalmente, por la naturaleza del core de la librería y especialmente por falta de experiencia y tiempo, se optó por cambiar la metodología a TAD (o POUT, Plain Old Unit Testing)[5].

2.4. Metodología empleada

El desarrollo de cualquier producto software se realiza siguiendo una determinada metodología o modelo de desarrollo, de manera que se realizan una serie de tareas entre la idea inicial y el resultado obtenido. El modelo de desarrollo escogido establece el orden en el que se han de afrontar las tareas en el desarrollo del proyecto, y nos provee de requisitos de entrada y salida para cada una de las actividades.

El desarrollo de este proyecto se ha basado parcialmente en un modelo de desarrollo en espiral. En este tipo de desarrollo software los productos son creados a través de múltiples iteraciones del proceso de ciclo de vida.

[3] http://www.stack.nl/~dimitri/doxygen/index.html
[4] http://en.wikipedia.org/wiki/Test-driven_development
[5] http://stephenwalther.com/blog/archive/2009/04/08/test-after-development-is-not-test-driven-development.aspx

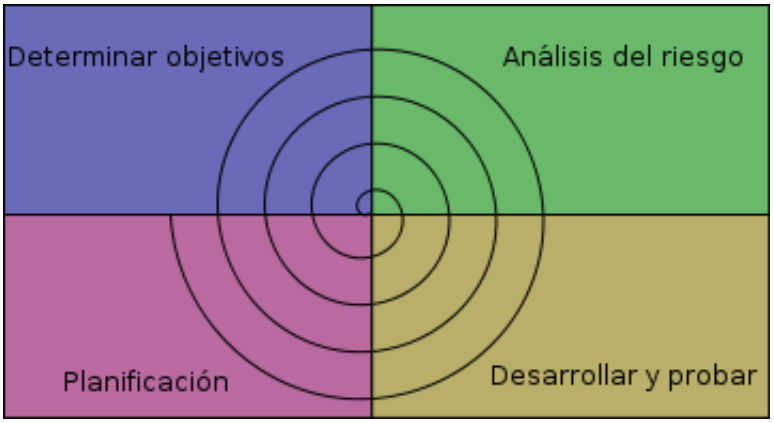


Figura 2.4.1: Tareas del desarrollo en espiral

El conjunto de actividades en que se puede dividir el modelo de desarrollo escogido comprende las siguientes fases:

- **Determinar objetivos.** El cliente especifica los objetivos del ciclo de la espiral. En este caso, el papel del cliente es desempeñado por el tutor del proyecto y, en menor parte, por el autor del mismo. Ambos son los encargados tanto de definir los objetivos y su alcance como de acotar los requisitos del proyecto.

- **Análisis del riesgo.** Se realizan las tareas necesarias para evaluar los riesgos técnicos y de gestión del proyecto referentes a la iteración actual. Los riesgos son estudiados y evaluados y conforme al resultado de dicho análisis ciertas actividades son puestas en vigor para reducir los riesgos claves.

- **Desarrollo y validación.** El sistema se desarrolla y se valida usando casos de prueba que comprueban que los requisitos especificados se cumplen de manera satisfactoria.

- **Planificación.** En esta fase se define el plan de acción para la siguiente iteración repasando el último ciclo de desarrollo del proyecto y se realizan tareas inherentes a la definición de los recursos, tiempo y otra información relacionada con el proyecto.

Este modelo proporciona una serie de ventajas vitales para el tipo de desarrollo que se pretende realizar. En primer lugar, proporciona gran flexibilidad a la hora de reconducir el desarrollo y adaptarse a los posibles cambios generados, por ejemplo, por errores en el diseño inicial, variación de los requisitos o cambios en el entorno del sistema. Dado que la implementación de una librería cliente de Google Data es totalmente dependiente de la especificación del Protocolo de Datos de Google y ya que no tenemos control alguno sobre dicho protocolo, es necesario contemplar la posibilidad de que éste varíe en mayor o menor medida y que dicha variación tenga un impacto negativo en nuestro desarrollo. Por otra parte, la falta de experiencia, como la que se presenta

en este caso, es también un gran motivo a tener en cuenta a la hora de escoger una metodología de desarrollo adaptable a cambios y modificaciones.

Otra de las ventajas que este modelo de desarrollo proporciona es la de proveer de puntos de control al final de cada iteración. Estos puntos de control están representados, en este caso, por los test unitarios desarrollados junto a la librería. Con estos tests se pretende comprobar la validez y consistencia de cada una de las iteraciones del desarrollo. Es importante destacar que los tests han de comprobar no solo la corrección de la iteración recientemente desarrollada, si no del total de las mismas desarrolladas hasta el momento.

Capítulo 3

Diseño e Implementación

3.1. Arquitectura general

Como se vio anteriormente, el Protocolo Google Data está basado en REST y por tanto en una arquitectura cliente-servidor, donde los clientes inician las peticiones HTTP a los servidores, que procesan dichas peticiones y retornan a los clientes respuestas apropiadas para cada caso. Las peticiones y respuestas están construidas alrededor de la transferencia de representaciones de recursos. Un recurso puede ser esencialmente cualquier concepto u objeto direccionable con significado coherente. La representación de un recurso es típicamente un documento que captura el estado actual o pretendido de un recurso.

En el caso de Google Data, la figura del servidor es desempeñada por Google, quien será el receptor de las peticiones de los clientes y el generador de las respuestas a los mismos. Los clientes deberán implementar su Protocolo de Datos, basado como ya se mencionó, en los protocolos Atom y RSS 2.0, que emplea XML para la representación de los recursos a intercambiar. La arquitectura de Google Data Qt Client, está basada en la de uno de estos clientes. Un cliente que implemente el Protocolo de Datos de Google ha de ser capaz de transformar su representación interna de objetos en XML (o JSON como método de representación alternativo) y viceversa, es decir, necesita disponer de un serializador de objetos del cliente a XML y de un parseador que transforme XML en objetos manejables por el cliente. Por otra parte, necesita de la figura de un conector HTTP que le permita enviar y recibir peticiones HTTP para realizar la comunicación con los servidores de datos de Google, los cuales requerirán de cierta información de seguridad para realizar la autenticación del usuario que realiza las peticiones. En la siguiente figura se puede observar una representación gráfica de la arquitectura citada anteriormente.

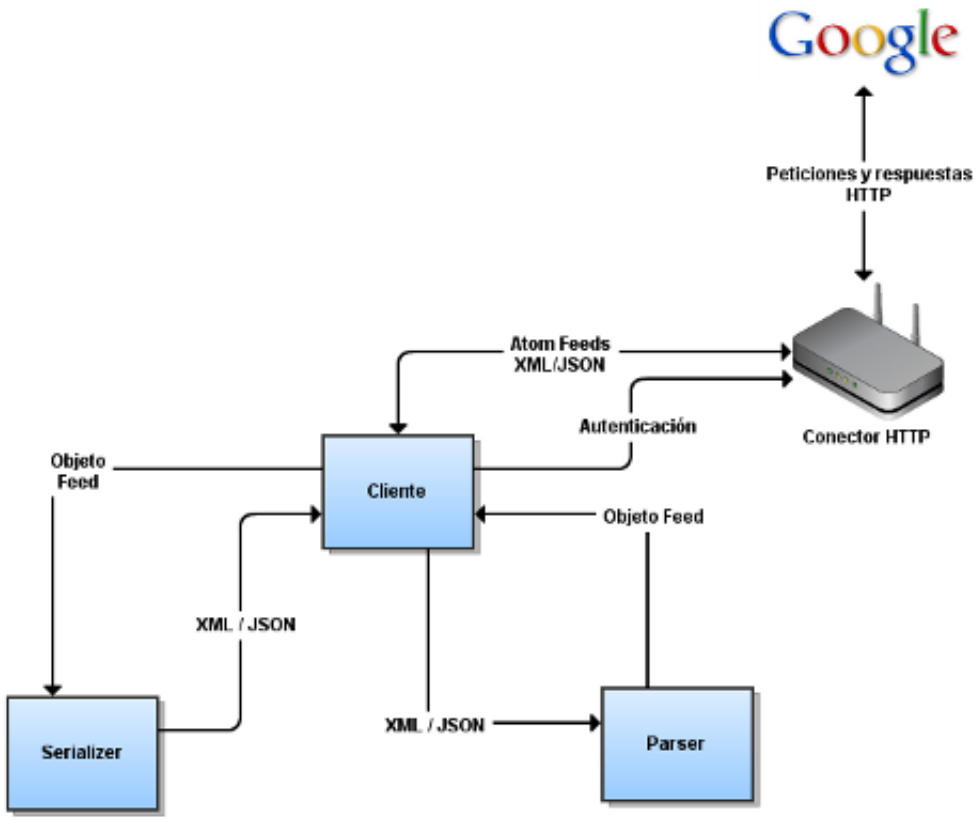


Figura 3.1.1: Arquitectura cliente-servidor de alto nivel del Protocolo de Datos de Google

Esta arquitectura de cliente, conector, serializador y parseador, representa el *core* de Google Data Qt Client y se ha implementado buscando la mayor genericidad posible con el fin de poder reutilizar la mayor cantidad de código a la hora de implementar las especializaciones de cada uno de los clientes. Es más, a lo largo del desarrollo de la librería, tal y como se verá más adelante en el apartado de conclusiones, me di cuenta de que con una mínimas modificaciones, el núcleo de la librería podría hacerse tan genérico que podría convertirse fácilmente en una librería para desarrollar cualquier cliente REST en Qt de manera rápida y sencilla.

En las siguientes secciones se describirá el proceso de desarrollo de cada una de las partes de la librería.

3.2. Iteración 0: Estudio previo y preparación del entorno de trabajo

3.2.1. Análisis de la arquitectura de otros clientes REST

Tras la fase de documentación a cerca de las tecnologías a emplear durante el desarrollo de este Proyecto de Fin de Carrera - que incluyó, entre otros temas, el estudio de la arquitectura REST, el Protocolo de Publicación Atom y el framework de desarrollo Qt - para llevar a cabo el diseño de Google Data Qt Client se realizó un estudio y análisis previo de la estructura y el diseño de algunas de las librerías cliente REST para el acceso a servicios web a través de API existentes en el momento del inicio del proyecto.

Como era lógico, las primeras librería estudiadas fueron las propias desarrolladas por Google para el acceso a su Protocolo de Datos. Concretamente se profundizó en el análisis de el cliente de Java, en cuya estructura se basó remotamente el diseño de Google Data Qt Client.

La librería cliente Java provee un set de clases que representan los elementos usados por el Protocolo de Datos de Google. Un ejemplo de ello es la clase *Feed* ,que tiene correspondencia con el elemento *<atom:entry>* de Atom . Al igual que en Google Data Qt Client, la librería puede parsear contenido Atom automáticamente y asignar los valores de dichos elementos Atom a sus representaciones en forma de objetos Java. Por ejemplo, el método *getFeed* obtiene un feed Atom, lo parsea y retorna un objeto Feed con los valores resultantes. Para enviar un feed o una entrada a un servicio, es necesario crear un objeto *Feed* o *Entry*, realizar la llamada a uno de los métodos de la librería (por ejemplo, el metodo *insert*) para realizar la traducción automática del objeto a XML para, posteriormente, realizar su envío. Del mismo modo, se puede serializar y/o generar XML; la manera más sencilla de hacerlo, al contrario que en Google Data Qt Client que implementa internamente un serializador, es mediante librerías de terceros como Rome[1].

Paralelamente, se realizó el estudio de los recientemente publicados SDKs (Software Development Kit) de la iniciativa BlueVia[2] de Telefónica. Dichos SDKs incluyen librerías cliente REST para el acceso a las APIs de los servicios que la operadora pone a disposición de terceros. Concretamente, se realizó el análisis del SDK para la plataforma móvil Android, cuya estructura de clientes y conector es muy similar a la de Google Data Qt Client. Por el contrario, la arquitectura de los mecanismos de serializado y parseado es bastante diferente, ya que, en el caso de este SDK, se realiza el serializado y parseado manual y específicamente para cada tipo de feed Atom enviado o recibido, mientras que en el caso de Google Data Qt Client el mecanismo es genérico y automático para ambos casos.

[1] http://java.net/projects/rome/
[2] http://www.bluevia.com

3.2.2. Herramientas de desarrollo

Para poder realizar el desarrollo de este Proyecto de Fin de Carrera se han requerido una serie de herramientas detalladas a continuación.

QtCreator

QtCreator es el entorno de desarrollo por excelencia de la plataforma Qt y el IDE escogido para el desarrollo de Google Data Qt Client. Posee un avanzado editor de código C++ y soporte para otros lenguajes como C#, Python, Ada, Pascal, PHP y Perl, entre otros. Por otra parte, proporciona una GUI integrada y un diseñador de formularios para la creación rápida de interfaces de usuario, al igual que un avanzado depurador visual y una herramienta para la detección de *memory leaks*.

Doxygen

Doxygen ha sido la herramienta escogida para desarrollar la documentación interna de la librería. Doxygen es un sistema generador de documentación multiplataforma que puede ser usado para diversos lenguajes como C/C++, D, Objective-C, Python, IDL, Fortran, VHDL, PHP y C#. Básicamente lo que hace es buscar en los archivos fuente de un directorio que especifiquemos cierto código el cual le dice qué y cómo documentar. El resultado final puede ser un documento html navegable, un documento pdf (usando LaTeX), un documento RTF o uno de tipo XML.

Git

Git es un software de control de versiones diseñado por Linus Torvalds[3], pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente. Al principio, Git se pensó como un motor de bajo nivel sobre el cual otros pudieran escribir la interfaz de usuario o front end como Cogito[4] o StGIT[5]. Sin embargo, Git se ha convertido desde entonces en un sistema de control de versiones con funcionalidad plena. Hay algunos proyectos de mucha relevancia que ya usan Git, por ejemplo, el grupo de programación del núcleo Linux o el sistema operativo móvil Android.

El desarrollo de Google Data Qt Client se ha realizado con Git como sistema de control de versiones. Existen dos repositorios con el código fuente del proyecto. Uno en Github[6] y, otro, mirror del primero, en Gitorious[7], dos portales que permiten el alojamiento y gestión web de repositorios Git. En dichos repositorios se puede observar la evolución del desarrollo del proyecto, incluyendo las diferentes ramas (*branches*) e hitos (*tags*) que han ido surgiendo. Del mismo modo, se pueden ver otras estadísticas más triviales sobre los lenguajes empleados o gráficos sobre los momentos de mayor actividad del proyecto.

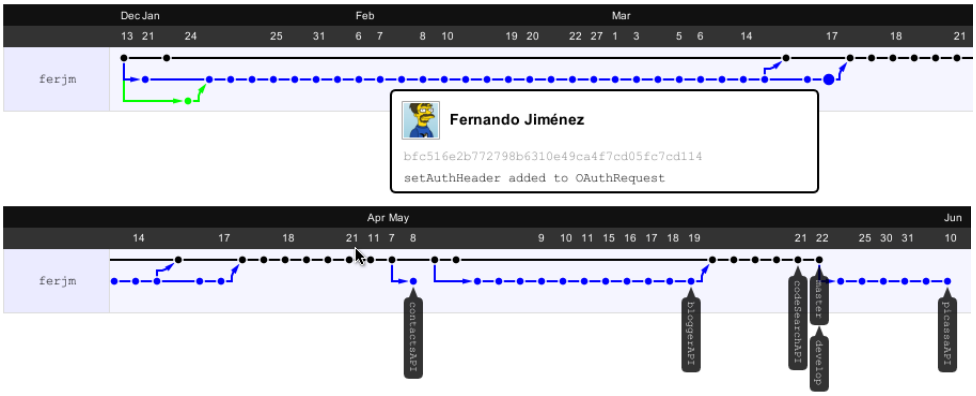


Figura 3.2.1: Gráficos del repositorio Git alojado en Github de Google Data Qt Client

[3] http://en.wikipedia.org/wiki/Linus_Torvalds
[4] http://git.or.cz/cogito/
[5] http://www.procode.org/stgit/
[6] https://github.com/ferjm/Google-Data-Qt-Client/
[7] https://gitorious.org/google-data-qt-client

LyX

Para la elaboración de esta documentación se ha escogido la herramienta LyX[8], un procesador de textos que permite generar código LaTeX a través de una interfaz gráfica de fácil uso.

3.3. Iteración 1: Diseño e implementación del Parser y el Serializer XML

3.3.1. Clase *IEntity*

Para hablar del diseño e implementación del parser y el serializer XML, es necesario hablar en primer lugar de la clase IEntity.

Físicamente, un documento XML está compuesto por unidades llamadas *entidades*. Una entidad contiene el tipo y valor o valores de cada una de las partes de un documento XML y puede hacer referencia a otra u otras entidades, causando que éstas se incluyan en el documento. Cada documento comienza con una entidad *documento*, también llamada *raíz*, y continúa con una serie de entidades, anidadas e incluidas dentro de la raíz, que pueden ser *vacías* (si solo presentan etiqueta), *simples* (si solo contienen un valor textual) o *compuestas* (si contienen una o más entidades anidadas en ella). Cada entidad puede, a su vez, disponer de una serie de atributos.

Esta composición en entidades está representada en Google Data Qt Client mediante la clase *IEntity*.

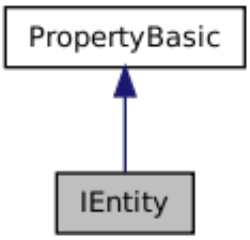


Figura 3.3.1: Diagrama de colaboración y herencia de la clase *IEntity*

La clase *IEntity* desciende de la clase *PropertyBasic,* que contiene las propiedades básicas de cada entidad, esto es, su nombre, identificador numérico, atributos y valores. Respecto al identificador numérico de cada entidad, Google Data Qt Client implementa un sistema de "pseudo-arrays" asociativos, entendiendo como array asociativo aquellos que en vez de estar organizados con índices numéricos en función de su posición dentro del array, están organizados por claves no numéricas. En este caso las claves sí son numéricas, pero identificadas por el nombre de contantes enumeradas, y por tanto indexables, en cierto modo, por nombre. El motivo de dicha

[8] http://www.lyx.org/

representación de identificadores es, en primer lugar, asegurar la validez de cada uno de los identificadores de las entidades, comprobando que se envían y reciben entidades válidas con nombres iguales a los que esperan y envían los servidores de Google. En segundo lugar, al realizar la comprobación entre enteros, se asegura una mejora significativamente mayor que al realizar la comprobación entre cadenas de caracteres. La implementación de este sistema se realiza a través del siguiente código:

```
#define Ids(...) typedef enum {__VA_ARGS__} tipo; \
    const QString sAux(#__VA_ARGS__);

namespace Id {
    Ids(
    NULLID,
    mockId,
    //-- atom ids
    feed,
    id,
    category,
    link,
    [...] )
}
namespace AttributeId {
    Ids(
    NULLID,
    mockId,
    [...] )
}
namespace NamespaceId {
    Ids(
    NULLID,
    mockId,
    [...])
}
```

Como se puede observar, la macro definida al inicio del código genera un *QString* a partir del contenido del enumerado *Ids* que se está definiendo y que se utiliza en cada uno de los namespaces. Cada enumerado corresponde con el nombre de una de las propiedades, atributos o espacios de nombres de los XML soportados por los servidores de Google. Finalmente, en la implementación de la clase *Qtgdata*, analizada posteriormente, se realiza el procesado de la cadena generada por la macro anterior, almacenando cada parte de la misma en las variables miembro de la clase *Qtdata* que corresponda en cada momento.

3.3.2. Clase Qtgdata

La clase *Qtgdata* mencionada anteriormente, representa el contenedor de los datos y la funcionalidad común al resto de clases de la librería. Como se describió anteriormente, contiene la lista de identificadores de las propiedades, atributos y espacios de nombres de las entidades que

se intercambiarán entre el cliente y los servidores de Google. Por otra parte, en un futuro, esta clase podrá contener más datos y funcionalidad común al resto de la librería, como ,por ejemplo, el objeto flyweight que se comentaba anteriormente en la descripción del cliente genérico.

Patrón de diseño *Singleton*

Por contener datos accesibles globalmente y por la necesidad de disponer de una única instancia de estos datos, la clase *Qtgdata* implementa el patrón de diseño *Singleton [9]*. Dicho patrón está concebido para restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella. Para ello, se implementa creando en la clase Singleton un método que crea una instancia del objeto sólo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor (con atributos como protegido o privado), tal y como se puede observar en el siguiente listado de código.

```
class Qtgdata {
    [...]
    /** Instancia única */
    static Qtgdata qtgdata;
    /*** Constructor */
    Qtgdata();
    public:
    /*** Destructor */
    ~Qtgdata();
    /*** Obtiene una referencia de la instancia única */
    static Qtgdata* getInstance();
    [...]
};
```

3.3.3. Clases ISerializer y XMLSerializer

La clase IEntity es el tipo de datos con el que trabajan tanto el serializer como el parser. En el caso del serializer, su tarea consiste en realizar el procesado y conversión de una instancia de la clase IEntity en un formato entendible por los servidores de Google Data. De cara a este Proyecto de Fin de Carrera solo se ha implementado el procesado y conversión de IEntity a XML, sin embargo, el diseño del serializer permite especializarlo para cualquier formato.

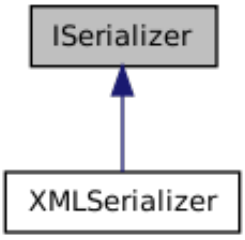


Figura 3.3.2: Diagrama de herencia de la clase *ISerializer*

La clase *ISerializer* declara la interfaz que cualquier clase serializadora ha de implementar. Dicha interfaz cuenta únicamente con el método abstracto puro *serialize*, que recibe como parámetro un objeto *IEntity* y retorna una cadena con el objeto serializado.

```
class ISerializer {
public:
        virtual QString serialize(const IEntity *obj) = 0;
};
```

XMLSerializer es, por el momento, la única implementación de la interfaz declarada por *ISerializer*. Utiliza la clase *QXmlStreamWriter* del módulo *QtXml*[9] para realizar el serializado **genérico** de la instancia de la clase *IEntity*.

3.3.4. Clases IParser y XMLParser

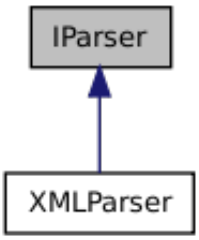


Figura 3.3.3: Diagrama de herencia de la clase *IParser*

Si el serializer se encargaba de realizar la conversión del objeto IEntity a un formato entendible por los servidores de Google Data, el parser realiza la función inversa, a partir de una cadena de carácteres con la respuesta del servidor, que se espera que venga en XML bien formado, crea un objeto IEntity, con el que la librería podrá realizar el resto de acciones oportunas.

El diseño de clases es muy similar al del serializer. En este caso, IParser es quien declara la interfaz que todos los parseadores deberán de implementar.

```
class IParser {
```

[9] *http://doc.qt.nokia.com/latest/qtxml.html*

```
public:
        virtual IEntity* parse(void *data) = 0;
};
```

XMLParser emplea *QDomDocument*[10] y el resto de clases del paquete *QtXml* relacionadas con la representación DOM[11] de documentos XML. Al igual que en el caso del serializer, el parseo se realiza de manera **genérica**.

3.4. Iteración 2: Diseño e implementación del Conector HTTP

El conector HTTP es el encargado de realizar el envío de las peticiones generadas por los clientes a los servidores de Google Data y la recepción de las respuestas que estos últimos generen de vuelta. La tarea del conector ha de ser lo más transparente posible para con los datos. Su cometido se ha de limitar al envío y recepción de los mismos, sin alterar en ningún momento su estructura o contenido. Actualmente, cada cliente instancia su propio conector dedicado.

3.4.1. Clases *HttpRequest* y *OAuthRequest*

El conector recibe como parámetros de entrada objetos de la clase *HttpRequest.* Esta clase supone una representación de lo que sería físicamente una petición HTTP. Contiene como miembros el tipo de petición HTTP (`GET, PUT, POST, DELETE o HEAD`), la URL del servidor al que se pretende enviar la petición, la lista de cabeceras HTTP y, finalmente, el cuerpo de la petición.

A parte de ser el contenedor de los parámetros de la petición HTTP, la clase *HttpRequest* proporciona la definición del método abstracto *setAuthHeader*, que deberá ser implementado en cada una de las clases hijas y que deberá proporcionar la funcionalidad para la generación de la cabecera de autenticación específica de cada petición. En el caso de Google Data Qt Client, el único método de autenticación implementado es OAuth, estudiado en el apéndice A.

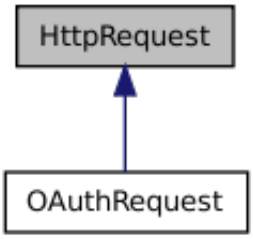


Figura 3.4.1: Diagrama de herencia de la clase *HttpRequest*

La clase *OAuthRequest* supone la especialización de la clase *HttpRequest,* que añade el conjunto de parámetros y la funcionalidad necesaria para implementar el protocolo de autenticación OAuth. Entre estos parámetros adicionales se encuentran:

[10] *http://doc.qt.nokia.com/latest/qdomdocument.html*
[11] http://es.wikipedia.org/wiki/Document_Object_Model

- *oauthConsumerKey* y *oauthConsumerSecretKey*, que son los valores alfanuméricos de las claves que Google proporciona para cada aplicación que haga uso del Protocolo Google Data. Por lo general, el proceso OAuth implementado por Google soportará el valor *anonymous* para estos dos parámetros en aplicaciones instaladas, no siendo así para aplicaciones web.
- *oauthToken* y *oauthTokenSecret*, que contienen los valores de las claves generadas por el servidor en respuesta a la petición *getAccessToken* de la clase OAuth.
- *oauthSignatureMethod*, enumerado para el tipo de firmado de la petición. Puede ser `PLAINTEXT`, `HMAC-SHA1` o `RSA-SHA1`.
- *oauthCallbackUrl*, que contendrá el valor de la URL de la aplicación que deberá ejecutarse tras completar el proceso de verificación del *request token*.
- *oauthVersion*, de momento, siempre 1.
- oauthVerifier, obtenido en el proceso de verificación del *request token* y necesario para finalizar el proceso OAuth mediante el método *getAccessToken*.

La implementación del proceso OAuth de Google Data Qt Client está ligeramente basada en la proporcionada por la librería KQOAuth de Johan Paul[12].

3.4.2. Clase *HttpConnector*

La clase *HttpConnector* es un wrapper[13] de la clase *QNetworkAccessManager*[14] del módulo *QtNetwork,* que permite a las aplicaciones enviar peticiones y recibir respuestas HTTP.

El método principal de esta clase es *httpRequest*.

```
void HttpConnector::httpRequest(const HttpRequest *request) {
        replyData.clear();
        QNetworkRequest networkRequest(request->getRequestEndpoint());
        QList<QPair<QByteArray,QByteArray> > httpHeaders = request->getHttpHeaders();
        QList<QPair<QByteArray,QByteArray> >::const_iterator it = httpHeaders.begin();
        for(;it != httpHeaders.end(); it++)
                networkRequest.setRawHeader((*it).first,(*it).second);
        QEventLoop *loop = new QEventLoop;
        HttpRequest::RequestHttpMethod httpMethod = request->getHttpMethod();
        switch(httpMethod)
        {
        case HttpRequest::GET:
                reply = manager->get(networkRequest);
                break;
        case HttpRequest::POST: {
                const QByteArray body = request->getRequestBody();
```

[12] http://gitorious.org/kqoauth/kqoauth
[13] http://es.wikipedia.org/wiki/Adapter_(patr%C3%B3n_de_dise%C3%B1o)
[14] http://doc.qt.nokia.com/4.7-snapshot/qnetworkaccessmanager.html

```
                reply = manager->post(networkRequest,body);
                break;
        }
        case HttpRequest::PUT: {
                const QByteArray body = request->getRequestBody();
                reply = manager->put(networkRequest,body);
                break;
        }
        case HttpRequest::HEAD:
                reply = manager->head(networkRequest);
                break;
        case HttpRequest::DELETE:
                reply = manager->deleteResource(networkRequest);
                break;
        }
        connect(reply, SIGNAL(readyRead()), this, SLOT(readyRead()));
        connect(manager,
                SIGNAL(finished(QNetworkReply*)),
                this,
                SLOT(finished(QNetworkReply*)));
        connect(manager,
                SIGNAL(finished(QNetworkReply *)),
                loop,
                SLOT(quit()));
        connect(reply,
                SIGNAL(error(QNetworkReply::NetworkError)),
                this,
                SLOT(error(QNetworkReply::NetworkError)));
        loop->exec();
}
```

3.5. Iteración 3: Diseño e implementación del Cliente

El cliente genérico es el eje central de la librería y es quien, junto a los clientes que heredan de él, implementa la mayor parte de la lógica para con los datos. Es también la interfaz pública con la que actuarán las aplicaciones que empleen la librería.

A grandes rasgos, el cliente es el encargado de recibir las peticiones de los usuarios (aplicaciones) a través de sus funciones públicas, a las que se les proporcionarán los parámetros para componer los objetos que representarán la información a intercambiar con los servidores de Google. Es, del mismo modo, el encargado de instanciar, en los casos que sea necesario, tanto al serializer como al parser para realizar la traducción objeto-XML y viceversa, y el encargado de proporcionarle los datos necesarios, incluidos los de seguridad, al conector HTTP para que éste realice la emisión de los datos a los servidores de Google y la recepción de las respuestas de los mismos.

Patrón de diseño *Facade*

El cliente genérico y todas las clases que de él descienden y, por tanto, lo especializan, implementan el patrón de diseño *Facade [9]*, que se utiliza para proporcionar una interfaz unificada de alto nivel para un conjunto de clases en un subsistema, haciéndolo más fácil de usar. Simplifica el acceso a dicho conjunto de clases, ya que el usuario de esas clases sólo se comunica con ellas a través de una única interfaz.

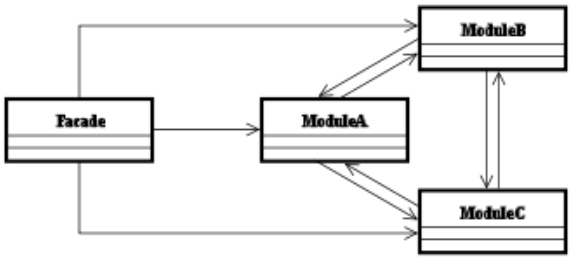


Figura 3.5.1: Estructura del patrón de diseño *Facade*

En el caso de Google Data Qt Client, el cliente es la interfaz común que está por encima del conjunto de clases formado por serializer, parser y conector.

3.5.1. Clase *QtgdataClient*

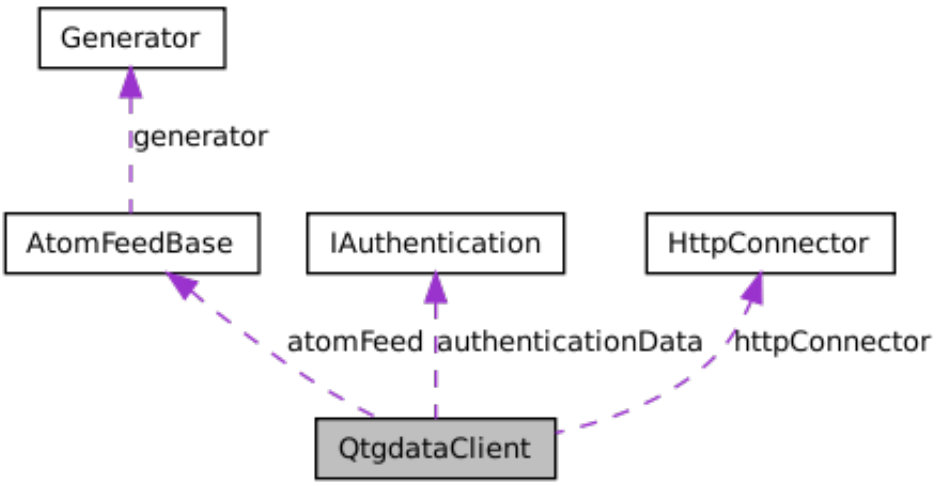


Figura 3.5.2: Diagrama de colaboración de la clase *QtgdataClient*

De cara al código, el cliente genérico está representado por la clase abstracta *QtgdataClient*. Dicha clase mantiene una instancia del objeto conector de tipo HttpConnector, analizado posteri-

ormente. Este conector será el que se utilizará en la implementación de cada una de las primitivas de las APIs de Google Data para enviar los datos compuestos por el cliente a los servidores de Google y para recibir las respuestas que estos últimos envíen de vuelta.

Por otra parte, la clase *QtgdataClient* contiene un miembro del tipo *AtomFeedBase*, que contendrá la información referente al último feed recibido en la última respuesta del servidor, en caso de ser una respuesta satisfactoria. Dicho objeto podrá ser de cualquiera de las especializaciones de la clase AtomFeedBase, esto es, a día de hoy *AtomFeed*, *CodeSearchFeed* y *PicasaFeed*. Esta instancia de la clase AtomFeedBase será la que se emitirá como dato de la señal[15] que deberán de capturar las aplicaciones que utilicen la librería para obtener los datos de respuesta de la última petición realizada.

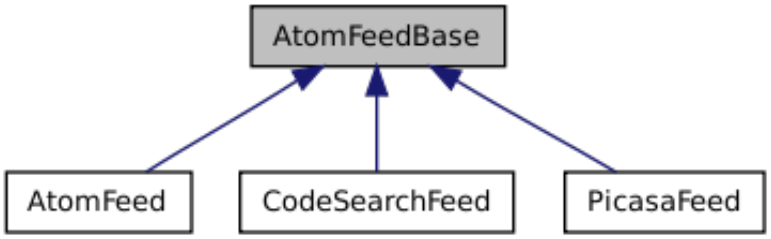


Figura 3.5.3: Diagrama de herencia de la clase *AtomFeedBase*

Finalmente, la clase *QtgdataClient* contiene los datos de autenticación del tipo *IAuthentication*, que se emplearán para proporcionarle al conector la cabecera de autenticación necesaria en cada caso. Por el momento, el único metodo de autenticación soportado por Google Data Qt Client es OAuth 1.0[16], descrito posteriormente. La clase *IAuthentication* es una clase abstracta de la cual heredarán las diferentes especializaciones dependiendo del tipo de autenticación empleada, en este caso, la única clase que hereda de *IAuthentication* es *OAuthData*, que contiene los parámetros necesarios para realizar el proceso OAuth, esto es, el *Consumer Key*, *Consumer Secret, Token Token Secret.* Dichos parámetros serán almacenados en la tabla hash heredada de la clase *IAuthentication.*

[15] http://doc.qt.nokia.com/4.7/signalsandslots.html
[16] http://tools.ietf.org/html/rfc5849

3.5.2. Clases *QtgdataBloggerClient*, *QtgdataCodeSearchClient* y *QtgdataPicasaClient*

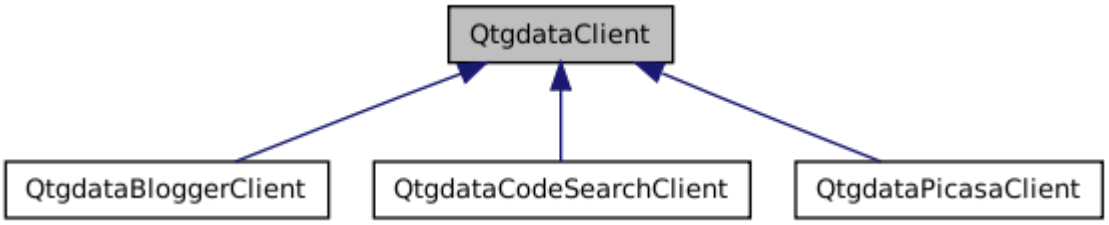


Figura 3.5.4: Diagrama de herencia de la clase *QtgdataClient*

De la clase QtgdataClient, descrita anteriormente, heredan cada una de las especializaciones de la misma que representarán e implementarán la funcionalidad específica de cada una de las APIs del Protocolo Google Data. Para la presentación de este PFC, se han implementado tres especializaciones. La primera de ellas para el servicio de blogging *Blogger*, implementada en la clase ***QtgdataBloggerClient***. Contiene toda la funcionalidad del API que permite:

- obtener la lista de blogs del usuario a través de la función *retrieveListOfBlogs*
- obtener la lista de posts de cada uno de esos blogs mediante la función *retrieveListOfPosts*, que recibe como parámetro, entre otros, el identificador numérico del blog a consultar.
- crear un post en uno de los blogs del usuario mediante la función *createPost*.
- actualizar el contenido de un post mediante la función *updatePost*.
- eliminar un post de un blog usando la función *deletePost*.
- obtener la lista de comentarios de todo un blog o de un post específico mediante la función *retrieveListOfComments*.
- crear comentarios en posts con la función *createComment*.
- eliminar comentarios con la función *deleteComment*.

La estructura algorítmica de cada una de estas funciones, y de las del resto de especializaciones de QtgdataClient, es muy similar entre ellas. Comienzan con la validación de los parámetros de entrada, en caso de tenerlos, y la composición de un objeto IEntity, del cual se hablará más adelante, con el valor de dichos parámetros. Esta instancia de la clase IEntity se le pasará al serializador, del tipo que sea, en este caso de XML, para que genere lo que será, en caso de tenerlo, el cuerpo de la petición al servidor. Una vez obtenido el cuerpo, se crearán las cabeceras necesarias para la petición HTTP, sin incluir las de seguridad, y se compondrá la URL del servicio a atacar. Finalmente, se realizará una llamada a la función *sendClientRequest*, implementada

en *QtgdataClient* y base de todas las peticiones al conector lanzadas desde clientes. Dada su importancia, vamos a analizar brevemente el código de esta función.

```
void QtgdataClient::sendClientRequest(HttpRequest::RequestHttpMethod method,
                QUrl endpoint,
                QList<QPair<QByteArray,QByteArray> > &headers,
                QByteArray *body,
                bool oauth) {
HttpRequest *request = authenticatedRequest();
request->setRequestEndpoint(endpoint);
request->setHttpMethod(method);
for(int i=0; i < headers.size(); i++)
    request->setHeader(headers.at(i).first,headers.at(i).second);
if(oauth)
    request->setAuthHeader();
if(body != NULL)
    request->setRequestBody(*body);
if((method == HttpRequest::POST) || (method == HttpRequest::PUT))
request->setHeader(QByteArray("Content-Type"),
           QByteArray("application/atom+xml"));
connect(&this->httpConnector,SIGNAL(requestFinished(QByteArray)),
    this,SLOT(onAtomFeedRetrieved(QByteArray)));
this->httpConnector.httpRequest(request);
delete request;
}
```

La función sendClientRequest, como se puede apreciar en el listado anterior, recibe como parámetros el método HTTP de tipo ***RequestHTTPMethod***, que puede tomar los valores de `GET`, `PUT`, `POST`, `DELETE` y `HEAD`, dependiendo del tipo de petición HTTP que se requiera. Del mismo modo, recibe la URL del endpoint que contiene el servicio del API que se pretende usar, una lista con las cabeceras HTTP que se van a incluir en la petición, el cuerpo de la misma (vacío por defecto) en formato XML (***hardcodeado,*** de momento) y, finalmente, un *flag* booleano indicando si la petición requiere cabecera de autenticación o no.

El algoritmo de la función comienza con la llamada al método *authenticatedRequest* que proporciona una instancia de una clase que especialice a la clase HttpRequest, explicada posteriormente, según el tipo de autenticación soportada, en este caso OAuth y, por tanto, *OAuthRequest.* Este objeto será el que reciba el conector HTTP para realizar la solicitud al servidor. En las siguientes líneas de código se le dan valores a los parámetros de la petición. Se introduce la URL del servidor, el método HTTP, las cabeceras HTTP, incluyendo la de autenticación, en caso de ser necesaria, y la de *Conten-Type* si la petición es de tipo `POST` o `PUT`. Finalmente se realiza la conexión entre la señal *requestFinished,* emitida en el momento en el que el conector finalice la petición, y el manejador *onAtomFeedRetrieved,* función muy importante, analizada tras el siguiente listado de código.

```
void QtgdataClient::onAtomFeedRetrieved(QByteArray reply)
```

```
{
[...]
XMLParser parser;
try {
    IEntity *entity = parser.parse(reply,reply.size());
    if((entity != NULL)&&(entity->getId() != Id::NULLID)){
        if(entity->getId() == Id::feed){
            atomFeed = createAtomFeed();
            IEntity::itConstEntities begin,end;
            entity->getEntityList(begin,end);
            if(begin != end){
                for(IEntity::itConstEntities it = begin; it != end; it++){
                    IEntity *sit = dynamic_cast<IEntity *>(*it);
                    switch(sit->getId()){
                    case Id::id:
                        atomFeed->id = sit->getValue();
                        break;
                    [...]
                    case Id::entry: {
                        AtomEntry *atomEntry = createAtomEntry();
                        [...]
                        int id = entry->getId();
                        switch(id) {
                        [...]
                        default:
                            parseEntry(id,atomEntry,entry);
                            break;
                        }
                   }
                   appendEntry(atomEntry);
                }
                default:
                    parseFeed(sit);
                    break;
                }
            }
        }
        emitAtomFeedRetrieved();
    } // if feed
 [...]
}
```

Esta función se encargará de realizar el procesado de la respuesta del servidor. Dicha respuesta la recibirá en forma de *QByteArray*[17] y deberá parsearla utilizando la clase *XMLParser*, que retornará una instancia de la clase *IEntity*. Dicha entidad deberá ser analizada y procesada para crear con sus datos el objeto *AtomFeedBase* contenido en la clase *QtgdataClient* y contenedor de los datos de la última respuesta del servidor. Dicho objeto es creado a través de la función

[17] http://doc.qt.nokia.com/latest/qbytearray.html

abstracta *createAtomFeed,* que es implementada en cada una de las clases hijas de *QtgdataClient* y que se encarga de destruir el feed existente, en caso de haber uno previo, y crearlo con el tipo necesario en función del tipo de cliente. Así, la clase *QtgdataCodeSearchClient*, por ejemplo, creará un objeto de tipo *CodeSearchFeed.* Una vez creado el objeto feed del tipo correspondiente, se procesa cada una de las entidades contenidas en el cuerpo de la respuesta del servidor, según el identificador de las mismas (sobre las propiedades de la clase IEntity se hablará más adelante) y se añade el valor extraído de la respuesta al miembro de la clase feed que corresponda. En el caso de encontrarse con una entidad de tipo *Entry,* se procesarán las entidades que ésta misma contenga, del mismo modo que anteriormente, con la diferencia de que por cada entidad contenida en una de tipo Entry se creará un objeto de una clase hija de la clase *AtomEntry*, creada a través de la función abstracta *createAtomEntry* implementada, del mismo modo que *createAtomFeed*, en cada una de las especializaciones del cliente abstracto. Estos objetos AtomEntry serán añadidos a la instancia de la clase AtomFeed contenida en el cliente, dando como resultado final, en la misma, una representación de la respuesta del servidor, recibida en XML y transformada en objeto AtomFeed. Finalmente, se realizará la llamada al método abstracto *emitAtomFeedRetrieved*, implementado en cada una de las clases hijas de *QtgdataClient* y encargado de realizar la emisión de la señal que expondrá a las aplicaciones que usen el API el objeto AtomFeed recientemente creado.

3.5.3. Patrón de diseño *Flyweight*

Durante el desarrollo de este Proyecto de Fin de Carrera, se identificó la posibilidad de emplear el patrón de diseño *Flyweight* [8] para la implementación del cliente. El patrón Flyweight sirve para eliminar o reducir la redundancia cuando tenemos gran cantidad de objetos que contienen información idéntica, además de lograr un equilibrio entre flexibilidad y rendimiento (uso de recursos).

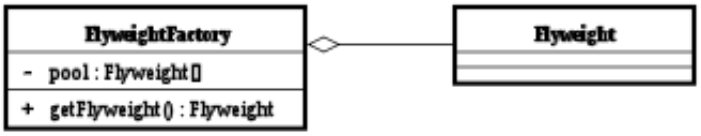


Figura 3.5.5: Arquitectura del patrón Flyweight

Actualmente, cada cliente contiene un objeto de la clase *HttpConnector*, un objeto de clase *IAuthentication*, un objeto de la clase *AtomFeedBase* y un entero indicando la versión del protocolo implementada por el cliente. La información de autenticación, el feed y la versión serán exclusivos de cada cliente, sin embargo, el conector podría ser común, ya que no requiere ninguna especialización derivada de cada cliente. Dicho conector podría ser una clase *flyweight* , compartida por cada uno de los clientes. Este *refactor* se realizará en futuras versiones de la librería.

3.6. Iteración 4: Desarrollo de los demostradores y ejemplos de uso

De cara a la presentación de este Proyecto de Fin de Carrera consideraba necesario realizar una prueba de concepto del uso de Google Data Qt Client. Para ello se realizó la implementación de dos aplicaciones demostradores que hacen uso de la librería, instanciando los clientes específicos de cada servicio utilizado por cada una de las aplicaciones y realizando las llamadas al API correspondientes a cada uno de ellos. Concretamente, por ser los únicos clientes totalmente funcionales en el momento de la presentación de este PFC, se realizaron los demostradores de los clientes de los servicios Blogger y Code Search.

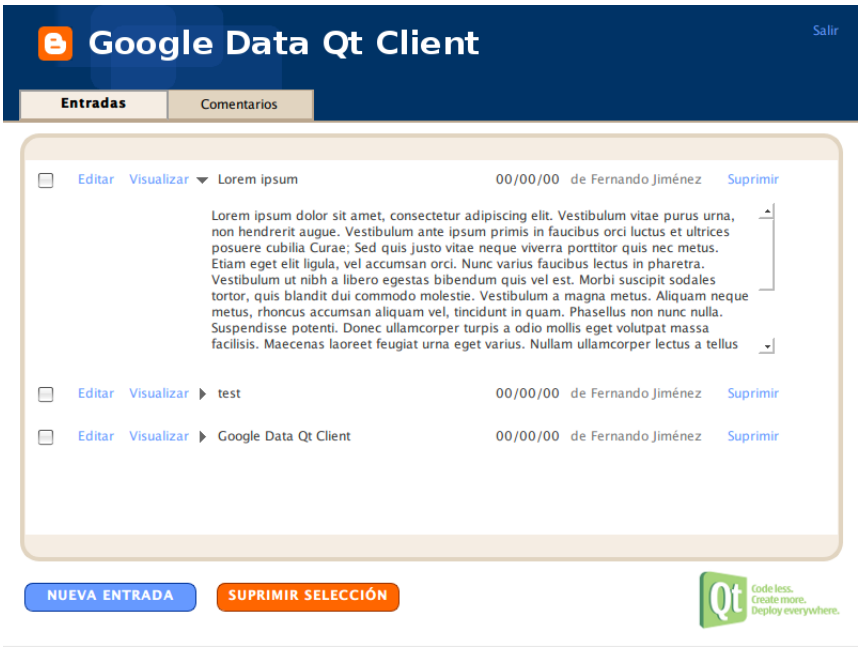


Figura 3.6.1: Screenshot del demostrador del cliente Blogger

El proceso de creación de aplicaciones utilizando Google Data Qt Client es relativamente sencillo. Para poder describir este proceso con mayor eficacia, realicé en un entorno recién instalado, diferente al que empleé para el desarrollo del código fuente de este proyecto, la simulación de los pasos que debería realizar un desarrollador para poder implementar una aplicación cliente utilizando la librería.

3.6.1. Instalación de Google Data Qt Client

El primer paso, evidentemente, es disponer de la librería compilada accesible a través de la aplicación a desarrollar. Para ello hay varias alternativas. Es posible realizar la instalación de

Google Data Qt Client a través de uno de los paquetes específicos de la plataforma objetivo, *.deb* o *.rpm*, publicados en SourceForge[18]. Por otra parte, también es posible obtener el código fuente de los repositorios existentes en Github[19] y Gitorious[20], con el fin de realizar la compilación y posterior instalación de la librería.

En el caso de querer realizar la instalación mediante uno de los paquetes publicados en SourceForge, es necesario descargar el específico de la plataforma sobre la que se pretende desarrollar la aplicación cliente. Si dicha plataforma utiliza el sistema de paquetería de Debian, el paquete adecuado será el *.deb* y su instalación se realizará mediante el comando *dpkg -i libqtgdata.deb*. Si por el contrario la distribución emplea el sistema de paquetería RPM, el comando a ejecutar será *rpm -i libqtgdata.rpm*. Hay que tener en cuenta que para implementar aplicaciones utilizando Google Data Qt Client es necesario instalar los paquetes de desarrollo (*dev*) de la misma.

Para realizar la compilación del código, es necesario obtener el mismo a través de uno de los repositorios Git publicados. Por ejemplo, en el caso de Github, sería necesario clonar el repositorio con el comando *git clone https://github.com/ferjm/Google-Data-Qt-Client.git*. Una vez obtenido el código fuente, los pasos para realizar su compilación son los siguientes:

- Google Data Qt Client depende de QtCore, QtNetwork y QtXml, incluidos por defecto en el framework Qt, con lo que es necesario asegurarse de que se cumplen todas las dependencias antes de comenzar la compilación.
- Ejecutar *qmake* en el directorio raíz del proyecto, donde deberá encontrarse el fichero con las especificaciones del mismo, *qtgdata.pro*.
- Ejecutar *make*, que realizará la compilación de la librería.
- Finalmente, ejecutar, con permisos de administrador, make install para realizar la instalación de la librería en el sistema. En el caso de querer enlazar estáticamente la librería desde la aplicación cliente, este último paso no sería necesario.

En ambos casos, tanto si se instala con uno de los paquetes *dev* como si instala a través del código fuente, la librería quedará alojada en */usr/local/lib/qtgdata* y los archivos de cabecera en */usr/local/include/qtgdata*.

3.6.2. Preparación del proyecto

Una vez que se dispone de la librería instalada en el sistema, el siguiente paso es la creación de un proyecto Qt. En ese caso, se empleó, nuevamente, como entorno de desarrollo, Qt Creator.

Para incluir la librería en el nuevo proyecto de aplicación, es necesario editar el fichero *.pro* de dicho proyecto. Se deben añadir las dependencias del módulo network y la ruta de inclusión del binario de la librería, así como de sus ficheros de interfaces. Tras realizar dichas modificaciones, el

[18] http://sourceforge.net/projects/qtgdataclient/
[19] https://github.com/ferjm/Google-Data-Qt-Client
[20] https://gitorious.org/google-data-qt-client

fichero de configuración del proyecto ha de quedar parecido al siguiente, empleado en el desarrollo del demostrador del cliente del servicio Blogger:

```
QT += core gui network
TARGET = DemoBlogger
TEMPLATE = app
SOURCES += main.cpp\
        mainwindow.cpp \
        postframe.cpp
HEADERS  += mainwindow.h \
        postframe.h
FORMS    += mainwindow.ui \
        postframe.ui
RESOURCES += \     bloggerres.qrc
INCLUDEPATH += /usr/local/include
LIBS += -L/usr/local/lib -lqtgdata
```

3.6.3. Proceso de autenticación

Una vez que se ha preparado el entorno de desarrollo, se puede comenzar la implementación del cliente. Como se explicó anteriormente, las aplicaciones necesitan pasar un proceso de autenticación para poder interactuar con los servicios de Google. De las opciones de autenticación que define el Protocolo de Datos de Google, en esta librería se implementa el de OAuth, explicado en detallo en el Anexo C. Para realizar dicho proceso, por ser una aplicación de escritorio, no es necesario registrar la misma para obtener el *Consumer Key* y el *Consumer Secret*. Basta con el valor *anonymous* para ambas claves. Por otra parte, es necesario indicar el ámbito (*scope*) de la aplicación de cara a los servicios de Google, en otras palabras, hay que indicarle las rutas de los endpoints para los cuales se quiere obtener el acceso. Por ejemplo, para el caso del demostrador del servicio Code Search, el ámbito es *https://www.google.com/codesearch/feeds/*. Con estos datos es posible comenzar el proceso de autenticación mediante la petición de un *request token* temporal. Para ello es necesario instanciar un objeto de la clase OAuth y realizar la conexión de la señal *temporaryTokenReceived* con un manejador que debemos de implementar, por ejemplo, *onTokensReceived*.

```
OAuth oauth("anonymous","anonymous",this);
connect(&oauth,
        SIGNAL(temporaryTokenReceived(QString,QString,QUrl)),
        this,
        SLOT(onTokensReceived(QString,QString,QUrl)));
```

La petición del request token se realiza mediante el método getRequestToken de la clase OAuth que recibe como parámetros una lista con los ámbitos anteriormente citados.

```
QList<QUrl> scope;
scope.append(QUrl("http://www.blogger.com/feeds/"));
oauth.getRequestToken(scope);
```

Tras realizar esta llamada, la librería se encargará de realizar la petición y obtener la respuesta del servidor, que devolverá junto a la emisión de la señal *temporaryTokenReceived* anteriormente citada. Una vez obtenido el token temporal, es necesario proceder a la autenticación del mismo a través de la URL obtenida en el manejador de la señal. Dicha URL llevará a una web como la siguiente.

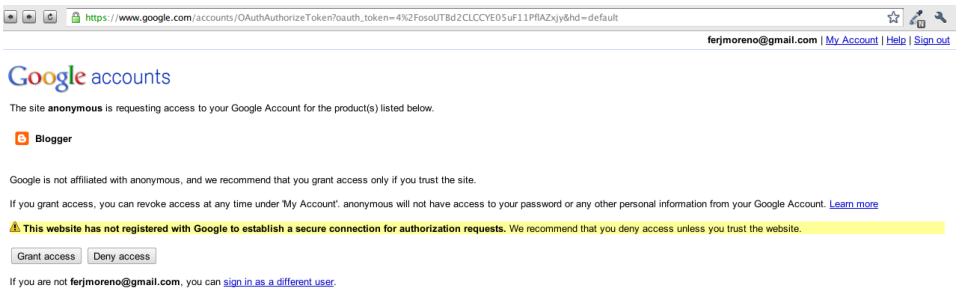


Figura 3.6.2: Primer paso para la verificación del *Request Token*

Si concedemos acceso a la aplicación, se nos mostrará una página como la siguiente, de la cual se puede obtener el código de verificación del Request Token obtenido anteriormente.

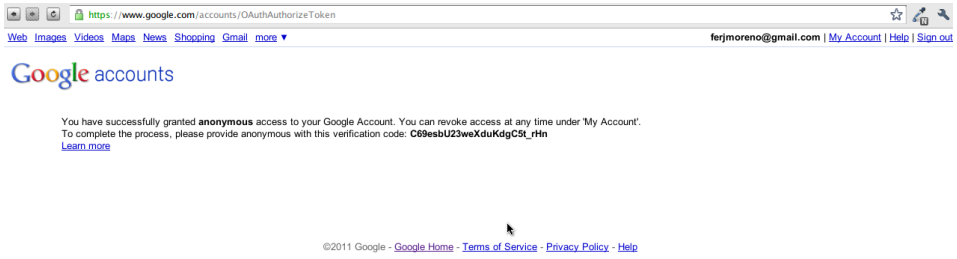


Figura 3.6.3: *Request Token* verificado

Con dicho código y los datos del token, se puede proceder a realizar la petición del *Access Token* final, que servirá para realizar las llamadas autenticadas contra los servicios de Google para los cuales se ha solicitado acceso. Esta petición se ha de realizar mediante la función de la clase OAuth *getAccessToken*, que recibirá como parámetros el código de verificación y los datos del request token. Previamente a dicha llamada, será necesario realizar nuevamente una conexión entre la señal accessTokenReceived y un manejador que deberá de implementar el desarrollador,

por ejemplo, *onAccessTokenReceived*, que recibirá los datos del token devueltos por el servidor.

```
connect(&oauth,
        SIGNAL(accessTokenReceived(QString,QString)),
        this,
        SLOT(onAccTokensReceived(QString,QString)));
oauth.getAccessToken("xxxxxxxxxxxxxxxxxxxxxxxx",
                "xxxxxxxxxxxxxxxxxxxxxxxx",
                "C69esbU23weXduKdgC5t_rHn");
```

Finalmente, si todo el proceso se ha completado correctamente, en el manejador de la señal anteriormente mencionada se dispondrá del Access Token final, con el cual se podrán realizar peticiones a los endpoints solicitados.

3.6.4. Instanciando clientes y realizando peticiones

Una vez que se han obtenido los datos de autenticación para la aplicación, es posible realizar la instanciación del cliente del API que requiera nuestro programa. En el caso del demostrador del servicio Blogger, se realiza la instanciación de un cliente de tipo *QtgdataBloggerClient*. El constructor de todos los clientes recibe como parámetro un puntero a un objeto de tipo OAuth que será necesario construir previamente.

```
OAuthData *auth = new OAuthData();
auth->setConsumerKey("anonymous");
auth->setConsumerSecret("anonymous");
auth->setToken("xxxxxxxxxxxxxxxxxxxxxxxxxxxx");
auth->setTokenSecret("xxxxxxxxxxxxxxxxxxxxxxxx");
QtgdataBloggerClient *bloggerClient = new QtgdataBloggerClient(auth);
```

A partir de este momento, es posible comenzar a realizar las peticiones específicas de cada cliente empleado. Por ejemplo, en el caso del demostrador del servicio Blogger, al comenzar la aplicación es necesario solicitar la lista de posts asociada a uno de los blogs del usuario (en este caso, yo mismo). Para ello se emplea la función *retrieveListOfPosts*. Si la respuesta del servidor es satisfactoria, dicha función emite la señal *listOfPostsRetrieved* junto a un objeto de tipo *AtomFeed* conteniendo la lista de entradas del blog especificado. Por tanto, es necesario realizar la conexión de esta señal con un manejador para la misma, en este caso, *onPostsRetrieved*.

```
connect(bloggerClient,
        SIGNAL(listOfPostsRetrieved(AtomFeed*)),
        this,
        SLOT(onPostsRetrieved(AtomFeed*)));
bloggerClient->retrieveListOfPosts(BLOG_ID);
```

Capítulo 4

Conclusiones

Una vez concluido el análisis del desarrollo de Google Data Qt Client, terminamos esta memoria resumiendo los principales logros alcanzados, así como las limitaciones del software desarrollado y las líneas futuras de desarrollo.

4.1. Logros alcanzados

Tras el desarrollo de este Proyecto de Fin de Carrera se ha conseguido el objetivo principal que consistía en el diseño e implementación en C++, utilizando el framework de desarrollo Qt, de una librería cliente REST para el acceso a las APIs de los servicios de Google que implementen su Protocolo de Datos, cubriendo de este modo el vacío existente al respecto.

El resultado del trabajo efectuado cumple todos y cada uno de los requisitos que se fijaron al inicio del proyecto, y extiende los mismos con logros adicionales:

- El núcleo de la librería proporciona una manera sencilla, rápida y segura para desarrollar clientes que implementen el Protocolo de Datos de Google. Es más, durante el desarrollo, se identificó la posibilidad de modificar mínimamente el núcleo para convertirlo en sí mismo en una librería genérica para la implementación rápida de clientes para servicios REST.

- En el momento de la entrega de este Proyecto de Fin de Carrera se proporcionan dos clientes totalmente funcionales para acceder a los datos de los servicios Blogger y Code Search de Google y parte de la funcionalidad de un cliente para el servicio Picasa Web.

- Del mismo modo, se han desarrollado y se entregan junto a la librería, dos demostradores de uso de la misma. Uno para cada uno de los clientes de las APIs cuya funcionalidad ha sido enteramente implementada.

- Por otra parte, tal y como se especificó en los requisitos, la librería se entrega perfectamente documentada con la herramienta de documentación de código Doxygen, con descripción

de las interfaces tanto públicas como privadas de todas las clases que conforman tanto el núcleo de la librería como los clientes de la misma. Del mismo modo, se proporcionan los diagramas UML de clases y de interacción de todas las clases desarrolladas.

- Con el objetivo de garantizar el buen funcionamiento y la efectividad del código desarrollado para la librería, se han implementado y se entregan una serie de test unitarios que cubren la mayor parte de la misma.

4.1.1. Publicación de Google Data Qt Client en SourceForge

Con motivo de la presentación de este Proyecto de Fin de Carrera, se ha decidido liberar la versión 0.1. Para ello se ha creado un proyecto en SourceForge[1], donde se han incluido paquetes instalables .deb y .rpm junto con un tarball de la actual versión del código fuente.

4.1.2. Inclusión en los Grupos de estudio de Telefónica I+D

Como logro adicional de este Proyecto de Fin de Carrera, he de mencionar la inclusión de Google Data Qt Client en el plan de trabajo de uno de los grupos de estudio de Telefónica I+D, concretamente el de Desarrollo Multiplataforma. La intención de dicho grupo es colaborar en el desarrollo e implementación de clientes para más APIs de Google Data empleando el core de Google Data Qt Client del mismo modo en que se han implementado los clientes de los servicios Blogger, Code Search y Picasa. Del mismo modo, la idea de modificar el core para facilitar la creación de clientes REST de manera genérica también se contempla como posible desarrollo en los planes de dicho grupo.

4.1.3. Resultados del análisis del software desarrollado con SLOCCount

SLOCCount[2] es una aplicación libre que toma como entrada recursivamente el código fuente de un proyecto especifíco y extrae información sobre los lenguajes utilizados y el número de líneas (físicas) de código en cada lenguaje. A partir de ese número de líneas, SLOCCount usa el método COCOMO para estimar esfuerzo y dedicación y, por tanto, coste de producción. A continuación se muestra la salida resultado de ejecutar SLOCCount sobre el código fuente desarrollado en este proyecto.

```
$ sloccount qtgdata qtgdatatest
Creating filelist for qtgdata
Creating filelist for qtgdatatest
Categorizing files.
Finding a working MD5 command....
Found a working MD5 command.
```

[1] http://sourceforge.net/projects/qtgdataclient/
[2] http://www.dwheeler.com/sloccount/

```
Computing results.

SLOC    Directory       SLOC-by-Language (Sorted)
3780    qtgdata         cpp=3780
279     qtgdatatest     cpp=279

Totals grouped by language (dominant language first): cpp:  4059 (100.00%)

Total Physical Source Lines of Code (SLOC)  = 4,059
Development Effort Estimate, Person-Years (Person-Months) = 0.87 (10.45)
(Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months) = 0.51 (6.10)
(Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule)  = 1.71 Total
Estimated Cost to Develop        = $ 117,620
(average salary = $56,286/year, overhead = 2.40).
SLOCCount, Copyright (C) 2001-2004 David A. Wheeler
SLOCCount is Open Source Software/Free Software, licensed under the GNU GPL.
SLOCCount comes with ABSOLUTELY NO WARRANTY, and you are welcome to redistribute
it under certain conditions as specified by the GNU GPL license; see the
documentation for details. Please credit this data as "generated using
David A. Wheeler's 'SLOCCount'."
```

4.1.4. Datos obtenidos mediante la herramienta de análisis de repositorios CVSAnaly 2.0

CVSAnalY es una herramienta que permite extraer información estadística de los logs de repositorios de código (CVS, Subversion o Git) y transformar la misma en una base de datos en formato SQL.

Estos son algunos de los datos extraídos con dicha herramienta del log del repositorio de Google Data Qt Client.

Como se puede apreciar en la primera gráfica, el proceso de desarrollo de Google Data Qt Client comenzó en Diciembre del año 2010. Concrétamente, el primer commit realizado fue el 13 de ese mes. Del mismo modo, se puede observar que el mayor pico de trabajo, sobre el código de la librería, ocurrió entre marzo y abril del 2011, tras un parón considerable en febrero de ese mismo año.

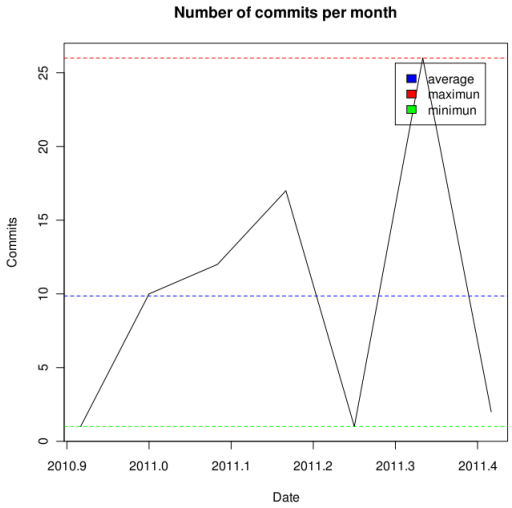


Figura 4.1.1: Número de commits al mes

Esta gráfica de actividad es contrastable con la generada por GitHub, donde se pueden apreciar los mismos picos de trabajo, con el añadido del total de líneas de código afectadas. Tal y como se puede observar en la siguiente figura, a lo largo del desarrollo de Google Data Qt Client se han realizado un total de 11764 inserciones y 7737 borrados de líneas de código.

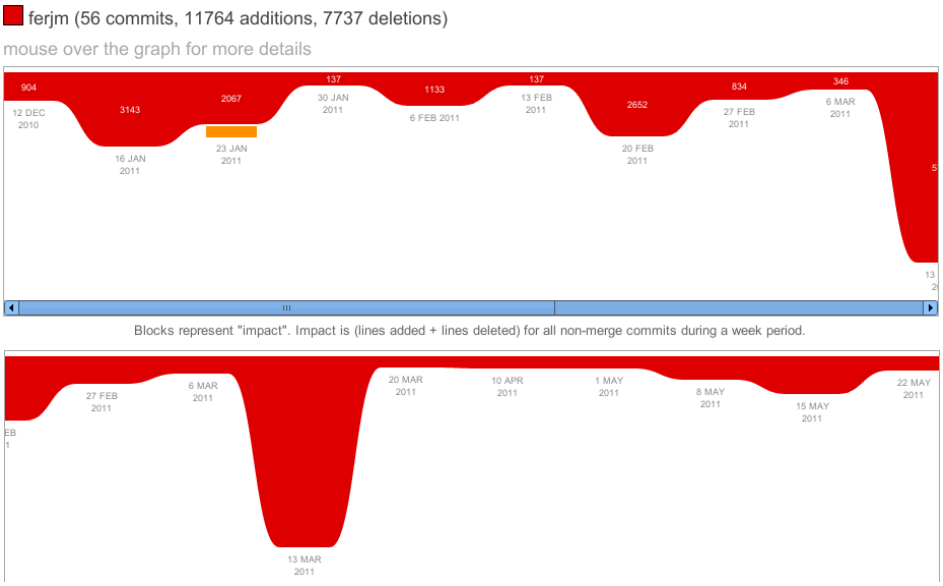


Figura 4.1.2: Número de commits al mes (Github)

Otro gráfico interesante es el del número de commits acumulados cada mes. En dicho gráfico

se puede observar el crecimiento del total de commits del proyecto, con la disminución en la velocidad de dicho crecimiento en los meses anteriormente mencionados de menor actividad productiva. Un dato que no parece cuadrar con los proporcionados por Github es el del número total de commits, situado por este último en 56.

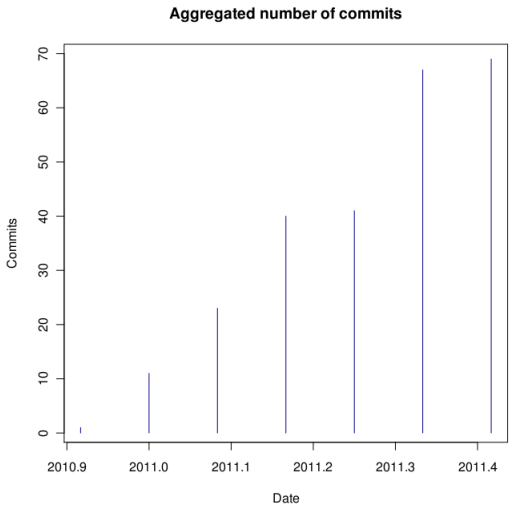


Figura 4.1.3: Total de commits acumulado

Los últimos gráficos generados a partir de los datos obtenidos por CVSAnaly2 muestran, nuevamente, la actividad del proyecto en términos de número de commits mensuales y anuales por autor, en este caso, se muestra que, efectivamente, un solo autor ha realizado commits al repositorio.

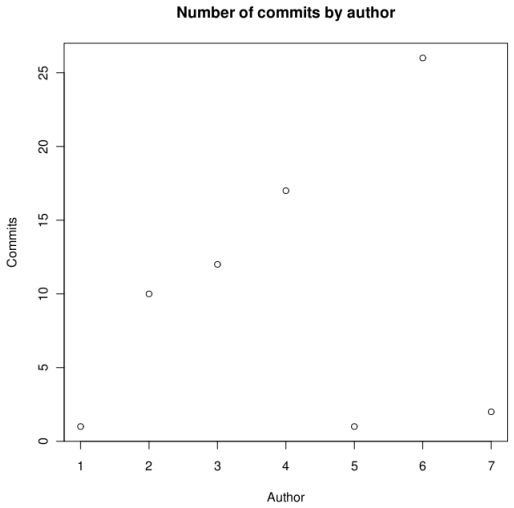


Figura 4.1.4: Número de commits por autor/mes

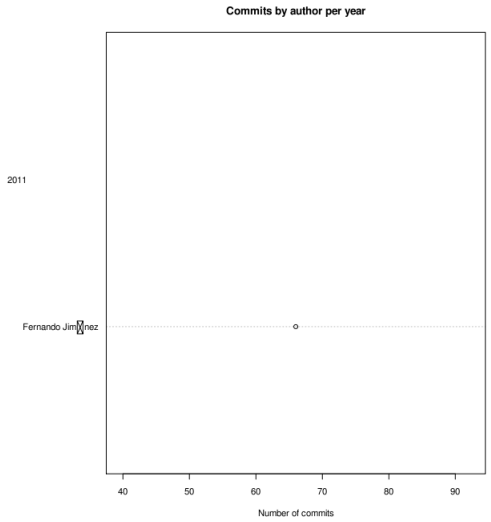


Figura 4.1.5: Número de commits por autor/año

4.2. Conocimientos adquiridos

Tras haber realizado todos estos desarrollos y tareas, puedo resumir los conocimientos adquiridos de su realización en los siguientes puntos:

- Aprendizaje y aplicación de la arquitectura de servicios **REST**, en cuyas bases se fundamenta la librería cliente desarrollada.

- Aprendizaje y aplicación del **Protocolo de Datos de Google** y, por tanto, del **Protocolo de Publicación Atom** y **RSS 2.0**. Del mismo modo, he adquerido bastantes conocimientos sobre las tecnologías **XML** y, en menor medida, **JSON**.
- Aprendizaje y aplicación del Protocolo de seguridad **OAuth**,
- Ampliación de mis conocimientos sobre la plataforma **Qt** y el lenguaje de programación **C++**. Durante este desarrollado he podido hacer uso de módulos de librerías Qt tales como **QtNetwork** y **QtXml**.
- Aplicación real de **Patrones de diseño**. Durante este desarrollo he podido poner en práctica los patrones de diseño **Flyweight**, **Facade** y **Adapter**.
- Ampliación de mis conocimientos respecto al sistema de control de versiones **Git**.
- Aprendizaje del uso de LaTeX, herramienta con la que se ha desarrollado esta memoria.

4.3. Trabajo futuro

Este Proyecto de Fin de Carrera supuso el inicio del desarrollo de Google Data Qt Client, sin embargo, su presentación no supondrá el fin de dicho desarrollo. Como siguientes tareas a realizar en un futuro próximo, están los desarrollos de los clientes para el resto de APIs que implementan el Protocolo de Datos de Google. Dicho desarrollo se realizará durante los próximos meses y estará abierto a contribuciones.

Por otra parte, a lo largo del desarrollo de Google Data Qt Client, se han identificado una serie de posibles mejoras en la arquitectura de la librería. Por ejemplo, como ya se comentó anteriormente, se está estudiando la posibilidad de implementar el patrón de diseño *Flyweight* para que los clientes compartan un único objeto conector, en lugar de disponer de uno exclusivo y dedicado para cada uno de ellos, con el aumento del *footprint* en memoria que esto supone. Otro punto a mejorar es el relacionado al manejo de errores, actualmente no hay ninguna manera unificada para proporcionar el feedback necesario respecto al buen o mal funcionamiento de las peticiones. Este es punto importante a reforzar en futuras versiones.

Finalmente, y tal y como se comentó anteriormente, durante el desarrollo de la librería se indentificó la posibilidad de, con una serie de mínimas modificaciones, adaptar el núcleo de la librería para proporcionar una librería autocontenida para la creación de clientes REST, no solo para el Protocolo de Datos de Google, sino para cualquier protocolo que implemente una arquitectura REST.

Apéndice A

Transferencia de Estado Representacional (REST)

La Transferencia de Estado Representacional (REST - Representational State Transfer) [2] fue ganando amplia adopción en toda la web como una alternativa más simple a SOAP y a los servicios web basados en el Lenguage de Descripción de Servicios Web (Web Services Descripcion Language - WSDL). Ya varios grandes proveedores de Web 2.0 están migrando a esta técnología, incluyendo, por supuesto, a Google, Yahoo o Facebook, quienes marcaron como obsoletos a sus servicios SOAP y WSDL y pasaron a usar un modelo más facil de usar, orientado a los recursos. Veamos los principios de REST para entender más esta tecnología.

A.1. Presentando REST

REST define un set de principios arquitectónicos por los cuales se diseñan servicios web haciendo foco en los recursos del sistema, incluyendo cómo se accede al estado de dichos recursos y cómo se transfieren por HTTP hacia clientes escritos en diversos lenguajes. REST emergió en los últimos años como el modelo predominante para el diseño de servicios. De hecho, REST logró un impacto tan grande en la web que prácticamente logró desplazar a SOAP y las interfaces basadas en WSDL por tener un estilo bastante más simple de usar.

REST no tuvo mucha atención cuando Roy Fielding lo presentó por primera vez en el año 2000 en la Universidad de California, durante la charla académica "Estilos de Arquitectura y el Diseño de Arquitecturas de Software basadas en Redes"[1], la cual analizaba un conjunto de principios arquitectónicos de software para usar a la Web como una plataforma de Procesamiento Distribuido. Ahora, años después de su presentación, existen múltiples frameworks REST y es parte integral de Java 6 a través de Java API for RESTful Web Services (JAX-WS) [2].

[1] http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
[2] http://www.oracle.com/technetwork/articles/javase/index-137171.html

A.1.1. Los 4 principios de REST

Una implementación concreta de un servicio web REST sigue cuatro principios de diseño fundamentales:

- Utiliza los métodos HTTP de manera explícita.
- No mantiene estado.
- Expone URIs con forma de directorios.
- Transfiere XML, JavaScript Object Notation (JSON), o ambos.

A continuación vamos a ver en detalle estos cuatro principios, y explicaremos porqué son importantes a la hora de diseñar un servicio web REST.

A.1.2. REST utiliza los métodos HTTP de manera explícita

Una de las caraterísticas claves de los servicios web REST es el uso explícito de los métodos HTTP, siguiendo el protocolo definido por RFC 2616. Por ejemplo, HTTP GET se define como un método productor de datos, cuyo uso está pensado para que las aplicaciones cliente obtengan recursos, busquen datos de un servidor web, o ejecuten una consulta esperando que el servidor web la realice y devuelva un conjunto de recursos.

REST hace que los desarrolladores usen los métodos HTTP explícitamente de manera que resulte consistente con la definición del protocolo. Este principio de diseño básico establece una asociación uno-a-uno entre las operaciones de crear, leer, actualizar y borrar y los métodos HTTP. De acuerdo a esta asociación:

- Se usa POST para crear un recurso en el servidor.
- Se usa GET para obtener un recurso.
- Se usa PUT para cambiar el estado de un recurso o actualizarlo.
- Se usa DELETE para eliminar un recurso.

Una falla de diseño poco afortunada que tienen muchas APIs web es el uso de métodos HTTP para otros propósitos. Por ejemplo, la petición del URI en un pedido HTTP GET, en general identifica a un recurso específico. O el string de consulta en el URI incluye un conjunto de parámetros que definen el criterio de búsqueda que usará el servidor para encontrar un conjunto de recursos. Al menos, así como el RFC HTTP/1.1 describe al GET.

Pero hay muchos casos de APIs web poco elegantes que usan el método HTTP GET para ejecutar algo transaccional en el servidor; por ejemplo, agregar registros a una base de datos. En estos casos, no se utiliza adecuadamente el URI de la petición HTTP, o al menos no se usa "a la manera REST". Si el API web utiliza GET para invocar un procedimiento remoto, seguramente se verá algo como esto:

```
GET /agregarusuario?nombre=Zim HTTP/1.1
```

Este no es un diseño muy atractivo porque el método aquí arriba expone una operación que cambia estado sobre un método HTTP GET. Dicho de otra manera, la petición HTTP GET de aquí arriba tiene efectos secundarios. Si se procesa con éxito, el resultado de la petición es agregar un usuario nuevo (en el ejemplo, Zim) a la base de datos. El problema es básicamente semántico. Los servidores web están diseñados para responder a las peticiones HTTP GET con la búsqueda de recursos que concuerden con la ruta (o el criterio de búsqueda) en la URI de la petición, y devolver estos resultados o una representación de los mismos en la respuesta, y no añadir un registro a la base de datos. Desde el punto de vista del protocolo, y desde el punto de vista de servidor web compatible con HTTP/1.1, este uso del GET es inconsistente.

Más allá de la semántica, el otro problema con el GET es que al ejecutar eliminaciones, modificaciones o creación de registros en la base de datos, o al cambiar el estado de los recursos de cualquier manera, provova que las herramientas de caché web y los motores de búsqueda (crawlers) puedan realizar cambios no intencionales en el servidor. Una forma simple de evitar este problema es mover los nombres y valores de los parámetros en la petición del URI a tags XML. Los tags resultantes, una representación en XML de la entidad a crear, pueden ser enviados en el cuerpo de un HTTP POST cuyo URI de petición es el padre de la entidad.

Antes:

```
GET /agregarusuario?nombre=Zim HTTP/1.1
```

Después:

```
POST /usuarios HTTP/1.1
Host: miservidor
Content-type: application/xml
<usuario>
   <nombre>Zim</nombre>
</usuario>
```

El método anterior es un ejemplo de una petición REST: hay un uso correcto de HTTP POST y la inclusión de los datos en el cuerpo de la petición. Al recibir esta petición, la misma puede ser procesada de manera de agregar el recurso contenido en el cuerpo como un subordinado del recurso identificado en el URI de la petición; en este caso el nuevo recurso debería agregarse como hijo de /usuarios. Esta relación de contención entre la nueva entidad y su padre, como se indica en la petición del POST, es análoga a la forma en la que está subordinado un archivo a su directorio. El cliente indica esta relación entre la entidad y su padre y define el nuevo URI de la entidad en la petición del POST.

El PUT arriba mostrado también tiene el efecto de renombrar al recurso Zim a Dib, y al hacerlo cambia el URI a */usuarios/Dib*. En un servicio web REST, las peticiones siguientes al recurso que apunten a la URI anterior van a generar un error estándard "404 Not Found".

Como un principio de diseño general, ayuda seguir las reglas de REST que aconsejan usar sustantivos en vez de verbos en las URIs. En los servicios web REST, los verbos están claramente definidos por el mismo protocolo: POST, GET, PUT y DELETE. Idealmente, para mantener una interfaz general y para que los clientes puedan ser explícitos en las operaciones que invocan, los servicios web no deberían definir más verbos o procedimientos remotos, como ser */agregarusuario* y */actualizarusuario*. Este principio de diseño también aplica para el cuerpo de la petición HTTP, el cual debe usarse para transferir el estado de un recurso, y no para llevar el nombre de un método remoto a ser invocado.

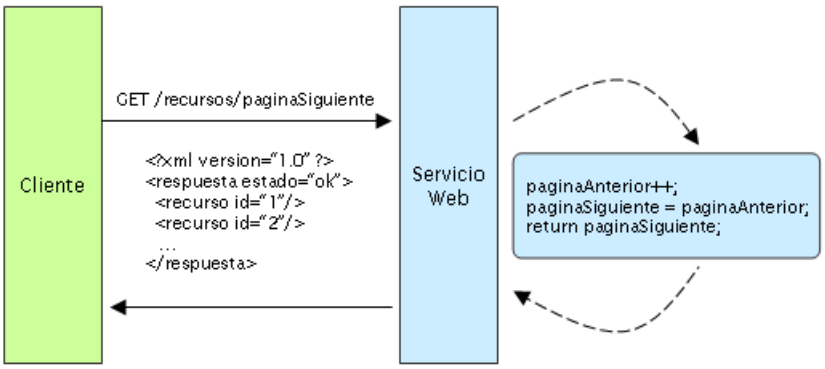
A.1.3. REST no mantiene estado

Los servicios web REST necesitan escalar para poder satisfacer una demanda en constante crecimiento. Se usan clusters de servidores con balanceadores de carga y alta disponibilidad, proxies, y gateways de manera de conformar una topología serviciable, que permita transferir peticiones de un equipo a otro para disminuir el tiempo total de respuesta de una invocación al servicio web. El uso de servidores intermedios para mejorar la escalabilidad hace necesario que los clientes de servicios web REST envien peticiones completas e independientes; es decir, se deben enviar peticiones que inlcuyan todos los datos necesarios para cumplir el pedido, de manera que los componentes en los servidores intermedios puedan redireccionar y gestionar la carga sin mantener el estado localmente entre las peticiones.

Una petición completa e independiente hace que el servidor no tenga que recuperar ninguna información de contexto o estado al procesar la petición. Una aplicación o cliente de servicio web REST debe incluir dentro del encabezado y del cuerpo HTTP de la petición todos los parámetros, contexto y datos que necesita el servidor para generar la respuesta. De esta manera, el no mantener estado mejora el rendimiento de los servicios web y simplifica el diseño e implementación de los componentes del servidor, ya que la ausencia de estado en el servidor elimina la necesidad de sincronizar los datos de la sesión con una aplicación externa.

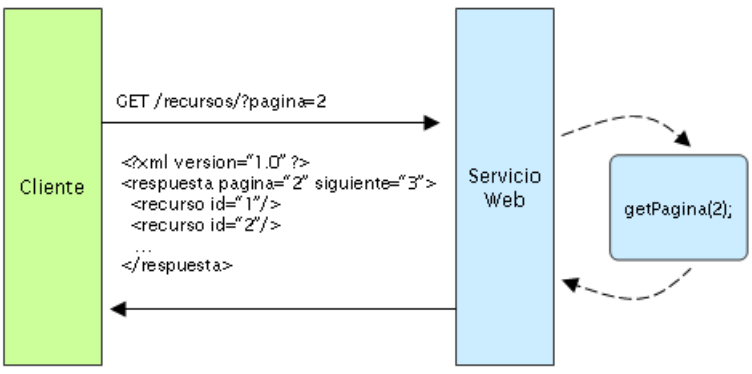
Servicios con estado vs. sin estado

La siguiente ilustración nos muestra un servicio con estado, del cual una aplicación realiza peticiones para la página siguiente en un conjunto de resultados múlti-página, asumiendo que el servicio mantiene información sobre la última página que pidió el cliente. En un diseño con estado, el servicio incrementa y almacena en algún lugar una variable *paginaAnterior* para poder responder a las peticiones siguientes.



Los servicios con estado tienden a volverse complicados. En la plataforma Java Enterprise Edition (Java EE)[3], un entorno de servicios con estado necesita bastante análisis y diseño desde el inicio para poder almacenar los datos eficientemente y poder sincronizar la sesión del cliente dentro de un cluster de servidores. En este tipo de ambientes, ocurre un problema que le resulta familiar a los desarrolladores de servlets/JSP y EJB, quienes a menudo tienen que revolver buscando la causa de una *java.io.NotSerializableException* cuando ocurre la replicación de una sesión. Puede ocurrir tanto sea en el contenedor de Servlets al intentar replicar la *HttpSession* o por el contenedor de EJB al replicar un EJB con estado; en todos los casos, es un problema que puede costar mucho esfuerzo resolver, buscando el objeto que no implementa *Serializable* dentro de un grafo complejo de objetos que constituyen el estado del servidor. Además, la sincronización de sesiones es costosa en procesamiento, lo que impacta negativamente en el rendimiento general del servidor.

Por otro lado, los **servicios sin estado** son mucho más simples de diseñar, escribir y distribuir a través de múltiples servidores. Un servicio sin estado no sólo funciona mejor, sino que además mueve la responsabilidad de mantener el estado al cliente de la aplicación. En un servicio web REST, el servidor es responsable de generar las respuestas y proveer una interfaz que le permita al cliente mantener el estado de la aplicación por su cuenta. Por ejemplo, en el mismo ejemplo de una petición de datos en múltiples páginas, el cliente debería incluir el número de página a recuperar en vez de pedir "la siguiente", tal como se muestra en la siguiente figura:



[3] http://download.oracle.com/javaee/

Un servicio web sin estado genera una respuesta que se enlaza a la siguiente página del conjunto y le permite al cliente hacer todo lo que necesita para almacenar la página actual. Este aspecto del diseño de un servicio web REST puede descomponerse en dos conjuntos de responsabilidades, como una separación de alto nivel que clarifica cómo puede mantenerse un servicio sin estado.

Responsabilidad del servidor

- Genera respuestas que incluyen enlaces a otros recursos para permitirle a la aplicación navegar entre los recursos relacionados. Este tipo de respuestas tiene enlaces embebidos. De la misma manera, si la petición es hacia un padre o un recurso contenedor, entonces una respuestsa REST típica debería también incluir enlaces hacia los hijos del padre o los recursos subordinados, de manera que se mantengan conectados.

- Genera respuestas que indican si son susceptibles de caché o no, para mejorar el rendimiento al reducir la cantidad de peticiones para recursos duplicados, y para lograr eliminar algunas peticiones completamente. El servidor utiliza los atributos Cache-Control y Last-Modified de la cabecera en la respuesta HTTP para indicarlo.

Responsabilidades del cliente de la aplicación

- Utiliza el atributo Cache-Control del encabezado de la respuesta para determinar si debe cachear el recurso (es decir, hacer una copia local del mismo) o no. El cliente también lee el atributo Last-Modified y envia la fecha en el atributo If-Modified-Since del encabezado para preguntarle al servidor si el recurso cambió desde entonces. Esto se conoce como GET Condicional, y ambos encabezados van de la mano con la respuesta del servidor 304 (No Modificado) y se omite al recurso que se había solicitado si no hubo cambios desde esa fecha. Una respuesta HTTP 304 significa que el cliente puede seguir usando la copia local de manera segura, evitando así realizar las peticiones GET hasta tanto el recurso no cambie.

- Envia peticiones completas que pueden ser serviciadas en forma independiente a otras peticiones. Esto implica que el cliente hace uso completo de los encabezados HTTP tal como está especificado por la interfaz del servicio web, y envia las representaciones del recurso en el cuerpo de la petición. El cliente envia peticiones que hacen muy pocas presunciones sobre las peticiones anteriores, la existencia de una sesión en el servidor, la capacidad del servidor para agregarle contexto a una petición, o sobre el estado de la aplicación que se mantiene entre las peticiones

Esta colaboración entre el cliente y el servicio es esencial para crear un servicio web REST sin estado. Mejora el rendimiento, ya que ahorro ancho de banda y minimiza el estado de la aplicación en el servidor.

A.1.4. REST expone URIs con forma de directorios

Desde el punto de vista del cliente de la aplicación que accede a un recurso, la URI determina qué tan intuitivo va a ser el web service REST, y si el servicio va a ser utilizado tal como fue pensado al momento de diseñarlo. La tercera característica de los servicios web REST es justamente sobre las URIs.

Las URI de los servicios web REST deben ser intuitivas, hasta el punto de que sea facil adivinarlas. Pensemos en las URI como una interfaz auto-documentada que necesita de muy poca o ninguna explicación o referencia para que un desarrollador pueda comprender a lo que apunta, y a los recursos derivados relacionados.

Una forma de lograr este nivel de usabilidad es definir URIs con una estructura al estilo de los directorios. Este tipo de URIs es jerárquica, con una única ruta raiz, y va abriendo ramas a través de las subrutas para exponer las áreas principales del servicio. De acuerdo a esta definición, una URI no es sólamente una cadena de caracteres delimitada por barras, sino más bien un árbol con subordinados y padres organizados como nodos. Por ejemplo, en un servicio de hilos de discusiones que tiene temas varios, se podría definir una estructura de URIs como esta:

```
http://www.miservicio.org/discusion/temas/{tema}
```

La raiz, `/discusion`, tiene un nodo /temas como hijo. Bajo este nodo hay un conjunto de nombres de temas (como ser tecnologia, actualidad, y más), cada uno de los cuales apunta a un hilo de discusión. Dentro de esta estructura, resulta facil recuperar hilos de discusión al tipear algo después de /temas/.

En algunos casos, la ruta a un recurso encacja muy bien dentro de la idea de "estructura de directorios". Por ejemplo, tomemos algunos recursos organizados por fecha, que son muy prácticos de organizar usando una sintáxis jerárquica.

El siguiente ejemplo es intuitivo porque está basado en reglas:

`http://www.miservicio.org/discusion/2008/12/23/{tema}`

El primer fragmento de la ruta es un año de cuatro dígitos, el segundo fragmento es el mes de dos dígitos, y el tercer fragmento es el día de dos dígitos. Puede resultar un poco tonto explicarlo de esta manera, pero es justamente el nivel de simpleza que buscamos. Tanto humanos como máquinas pueden generar estas estructuras de URI porque están basadas en reglas. Como vemos, es facil llenar las partes de esta URI, ya que existe un patrón para crearlas:

`http://www.miservicio.org/discusion/{año}/{mes}/{dia}/{tema}`

Podemos también enumerar algunas guías generales más al momento de crear URIs para un servicio web REST:

- ocultar la tecnología usada en el servidor que aparecería como extensión de archivos (.jsp, .php, .asp), de manera de poder portar la solución a otra tecnología sin cambiar las URI.
- mantener todo en minúsculas.
- sustituir los espacios con guiones o guiones bajos (uno u otro).

- evitar el uso de strings de consulta.
- en vez de usar un 404 Not Found si la petición es una URI parcial, devolver una página o un recurso predeterminado como respuesta.

Las URI deberían ser estáticas de manera que cuando cambie el recurso o cambie la implementación del servicio, el enlace se mantenga igual. Esto permite que el cliente pueda generar "favoritos" o bookmarks. También es importante que la relación entre los recursos que está explícita en las URI se mantenga independiente de las relaciones que existen en el medio de almacenamiento del recurso.

A.1.5. REST transfiere XML, JSON, o ambos

La representación de un recurso en general refleja el estado actual del mismo y sus atributos al momento en que el cliente de la aplicación realiza la petición. La representación del recurso son simples "fotos" en el tiempo. Esto podría ser una representación de un registro de la base de datos que consiste en la asociación entre columnas y tags XML, donde los valores de los elementos en el XML contienen los valores de las filas. O, si el sistema tiene un modelo de datos, la representación de un recurso es una fotografía de los atributos de una de las cosas en el modelo de datos del sistema. Estas son las cosas que serviciamos con servicios web REST.

La última restricción al momento de diseñar un servicio web REST tiene que ver con el formato de los datos que la aplicación y el servicio intercambian en las peticiones/respuestas. Acá es donde realmente vale la pena mantener las cosas simples, legibles por humanos, y conectadas.

Los objetos del modelo de datos generalmente se relacionan de alguna manera, y las relaciones entre los objetos del modelo de datos (los recursos) deben reflejarse en la forma en la que se representan al momento de transferir los datos al cliente. En el servicio de hilos de discusión anterior, un ejemplo de una representación de un recurso conectado podría ser un tema de discusión raiz con todos sus atributos, y links embebidos a las respuetas al tema.

```
<discusion fecha="{fecha}" tema="{tema}">
        <comentario>{comentario}</comentario>
        <respuestas>
                <respuesta de=" gaz@mail.com"
                href="/discusion/temas/{tema}/gaz"/>
                <respuesta de=" gir@mail.com"
                href="/discusion/temas/{tema}/gir"/>
        </respuestas>
</discusion>
```

Por último, es bueno construir los servicios de manera que usen el atributo HTTP Accept del encabezado, en donde el valor de este campo es ul tipo MIME. De esta manera, los clientes pueden pedir por un contenido en particular que mejor pueden analizar. Algunos de los tipos MIME más usados para los servicios web REST son:

MIME-Type	Content-Type
JSON	application/json
XML	application/xml
XHTML	application/xhtml-xml

Esto permite que el servicio sea utilizado por distintos clientes escritos en diferentes lenguajes, corriendo en diversas plataformas y dispositivos. El uso de los tipos MIME y del encabezado HTTP Accepto es un mecanismo conocido como negociación de contenido, el cual le permite a los clientes elegir qué formato de datos puedan leer, y minimiza el acoplamiento de datos entre el servicio y las aplicaciones que lo consumen.

A.1.6. Conclusión

No siempre REST es la mejor opción. Es una alternativa para diseñar servicios web con menos dependencia en middleware propietario (por ejemplo, un servidor de aplicaciones), que su contraparte SOAP y los servicios basados en WSDL. De algún modo, REST es la vuelta a la Web antes de la aparición de los grandes servidores de aplicaciones, ya que hace énfasis en los primeros estándares de Internet, URI y HTTP. Como examinamos anteriormente, XML sobre HTTP es una interfaz muy poderosa que permite que aplicaciones internas, como interfaces basadas en JavaScript Asincrónico + XML (AJAX) puedan conectarse, ubicar y consumir recursos. De hecho, es justamente esta gran combinación con AJAX que generó esta gran atención que tiene REST hoy en día.

Resulta muy flexible el poder exponer los recursos del sistema con un API REST, como medio de brindar datos a distintas aplicaciones, formateados en distintas maneras. REST ayuda a cumplir con los requerimientos de integración que son críticos para construir sistemas en donde los datos tienen que poder combinarse fácilmente (mashups) y extenderse. Desde este punto de vista, los servicios REST se convierten en algo mucho más grande.

Apéndice B

El Protocolo de Publicación Atom

El Protocolo de Publicación Atom [3][4][5] es una aproximación basada en HTTP para la creación y edición de recursos Web. Fue diseñada fundamentalmente con la idea de emplear las operaciones básicas provistas por el protocolo HTTP (tales como GET, PUT y DELETE) para intercambiar instancias de documentos *Atom 1.0 Feed* y *Entry*, que representan datos como entradas en blogs, podcasts, páginas wiki, calendarios, etc.

B.1. Visión general

Común al Protocolo de Publicación Atom es el concepto de colecciones de recursos editables que están representados por los documentos *Atom 1.0 Feed* y *Entry*. Una colección tienen un URI único. Lanzando una petición HTTP GET contra la URL de dicho recurso retornará un documento Atom Feed. Para crear nuevas entradas en dicho feed, los clientes deberán enviar peticiones HTTP POST a la URI de la colección. A estas entradas recientemente creadas se le asignarán una URI propia de edición. Para modificarlas, el cliente simplemente tendrá que obtener el recurso de la colección, realizar las modificaciones oportunas y devolverlo a la misma URI. Eliminar una entrada de un feed es tan simple como enviar una petición HTTP DELETE a la URI de edición apropiada. Todas estas operaciones se realizan usando simples peticiones HTTP y normalmente pueden ser efectuadas con un simple editor de texto y una línea de comandos.

Listing B.1: Interaccionando con un endpoint de publicación Atom usando el cliente HTTP libre curl

```
curl -s -X POST --data-binary @entry.xml http://example.org/atom/entries
curl -s -X GET http://example.org/atom/entries/1
curl -s -X PUT --data-binary @entry.xml http://example.org/atom/entries/1
curl -s -X DELETE http://example.org/atom/entries/1
```

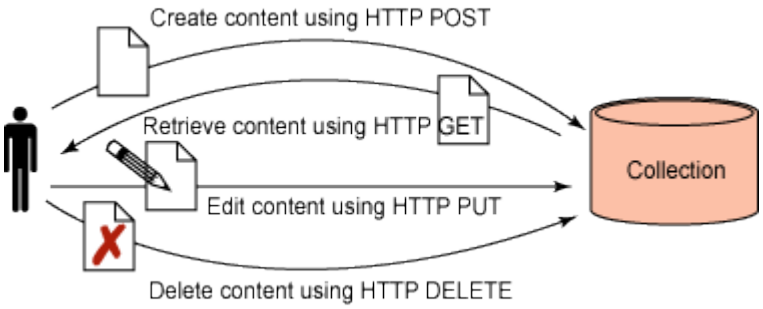


Figura B.1.1: El Protocolo Atom emplea simples peticiones HTTP para publicar y modificar contenido

B.2. Descubriendo qué colecciones se encuentran disponibles

El primer paso para usar cualquier servicio que soporte Atom es determinar qué colecciones están disponibles y qué tipo de recursos pueden contener dichas colecciones. El protocolo Atom especifica un formato XML conocido como un documento del servicio que un cliente puede usar para inspeccionar el endpoint. Para obtener dicho documento, es necesario enviar una petición HTTP GET a la URI correspondiente al servicio.

```
GET /servicedocument HTTP/1.1
Host: example.org
```

El servidor debe responder con un documento que enumere las colecciones disponibles para el cliente, tal y como muestra el siguiente listado.

```
HTTP/1.1 200 OK
Date: ...
Content-Type: application/atomserv+xml;
charset=utf-8
Content-Length: nnn
<service xmlns="..." xmlns:atom="http://www.w3.org/2005/Atom">
        <workspace>
                <atom:title>Google Data Qt Client</atom:title>
                <collection href="http://www.ejemplo.org/blog/entries">
                        <atom:title>Entradas</atom:title>
                        <accept>entry</accept>
                </collection>
                <collection href="http://www.ejemplo.org/blog/photos">
                        <atom:title>Fotos</atom:title>
                        <accept>image/*</accept>
                </collection>
        </workspace>
</service>
```

Cada elemento de la colección listado representa un contenedor dentro del cual alguna pieza de contenido puede ser almacenada. Los elementos del workspace del documento sirven únicamente

para agrupar colecciones realicionadas en conjuntos lógicos. Por ejemplo, un usuario puede tener múltiples cuentas para un servicio de blogging dado que provee diferentes contenedores para entradas de blogs, archivos subidos, marcadores, etc. Cada servicio puede ser representado como un workspace separado. El elemento de la colección provee de la dirección de la misma (el atributo *href*) y un listado de los tipos de contenidos que pueden ser añadidos a la colección (identificados por el mime type en el elemento *accepts*). El ejemplo del anterior XML tiene dos colecciones, una que acepta únicamente documentos Atom Entry y otra que únicamente acepta archivos de imagen (tales como PNG, GIF, JPEG y otros).

B.3. Añadiendo una entrada a la colección

Una vez que está disponible la dirección de la colección, es posible usar HTTP POST para añadir un nuevo recurso, tal y como se muestra en el siguiente listado.

```
POST /blog/entries HTTP/1.1
Host: www.ejemplo.org
Content-Type: application/atom+xml;
charset=utf-8 Content-Length: nnn
<?xml version="1.0" ?>
<entry xmlns="http://www.w3.org/2005/Atom">
        <title>Google Data Qt Client</title>
        <link href="http://ejemplo.org/2011/05/27/atom03"/>
        <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>
        <updated>2011-05-27T18:30:02Z</updated>
        <author>
                <name>Fernando Jiménez</name>
        </author>
        <summary>Ejemplo...</summary>
</entry>
```

En el ejemplo anterior, se añade una entrada Atom a la colección localizada en *http://ejemplo/blog/entries*. La URI de la colección es adquirida a través del documento de servicio obtenido anteriormente. Hay que tener en cuenta que la entrada *posteada* debe ser válida, esto es, debe tener los elementos *id*, *author* y *updated* a pesar de que muchos servidores ATOM ignoraran y sobreescribiran los valores provistos por el cliente. Una respuesta satisfactoria a la petición de POST, ilustrada en el siguiente listado de código, provee al cliente de dos pedazos críticos de información: el estado de la petición (el código de respuesta HTTP) y la dirección del recurso recientemente creado, contenido en la cabecera *Location*.

```
HTTP/1.1 201 Created
Date: nnnn
Content-Type: application/atom+xml; charset=utf-8
Content-Location: /blog/entries/1
Location: /blog/entries/1
ETag: "/blog/entries/1?1"
Last-Modified: Fri, 27 May 2011 13:40:03 GMT
```

```
<?xml version="1.0" ?>
<entry xmlns="http://www.w3.org/2005/Atom" >
        <id>tag:ejemplo.org,2011:/blog/entries/1</id>
        <title>Google Data Qt Client</title>
        <link href="http://ejemplo.org/2011/05/27/atom03"/>
        <link rel="edit" href="http://ejemplo.org/blog/entries/1" />
        <updated>2011-05-27T13:40:03Z</updated>
        <author>
                <name>Fernando Jiménez</name>
        </author>
        <summary>Ejemplo...</summary>
</entry>
```

Algunos servidores ATOM pueden modificar varias aspectos clave de una entrada (tales como los elementos *id*, *author* y *updated*), por lo que la respuesta del servidor puede incluir una copia de la entrada que ha sido realmente añadirda a la colección. Esto proporciona al cliente un medio para contrastar y reconciliar la entrada enviada al servidor con la que realmente se ha creado.

B.4. Listando las entradas en una colección

Una vez que una entrada ha sido añadida a una colección, los clientes pueden obtener una lista de los recursos miembro realizando una petición GET a la URI de la colección, de este modo:

```
GET /blog/entries HTTP/1.1
Host: ejemplo.org
```

La respuesta a esta petición será un documento Atom Feed cuya lista de entradas representará exactamente los recursos de la colección, tal y como se ilustra en el siguiente código.

```
HTTP/1.1 200 OK
Date: ...
Content-Type: application/atom+xml; charset=utf-8
Content-Length: nnn
ETag: "/blog/entries?132"
Last-Modified: Fri, 27 May 2011 13:40:03 GMT
<feed xmlns="http://www.w3.org/2005/Atom"
        xml:base="http://ejemplo.org/blog/entries">
        <id>http://ejemplo.org/blog/entries</id>
        <title>Google Data Qt Client</title>
        <updated>2011-05-27T13:40:03Z</updated>
        <link rel="self" href="/blog/entries" />
        <link href="http://blog.ejemplo.org"/>
        <entry>
            <id>tag:ejemplo.org,2006:/blog/entries/1</id>
                <title>Google Data Qt Client</title>
                <link href="http://ejemplo.org/2003/12/13/atom03"/>
                <link rel="edit" href="http://ejemplo.org/blog/entries/1" />
```

```
                <updated>2011-05-27T13:40:03Z</updated>
                <author><name>Fernando Jiménez</name></author>
                <summary>Ejemplo...</summary>
        </entry>
        <entry>
                ...
        </entry>
        ...
</feed>
```

Se podría tomar el *feed* retornado por la colección como un índice de la misma, algo así como el resultado de realizar un comando *ls* o *dir* en un sistema de ficheros. La entradas mismas están ordenadas según el valor del campo updated de cada una de ellas, estando las más recientes en primer lugar. Adicionalmente, el listado de entradas puede abarcar múltiples feeds enlazados usando los denominados *paging links.*

```
<feed xmlns="http://www.w3.org/2005/Atom"
        xml:base="http://example.org/blog/entries?page2">
        <link rel="next" href="entries?page3" />
        <link rel="previous" href="entries?page1" />
        ...
```

Los *paging links* proveen un manera de fragmentar posibles listados de tamaño excesivo en partes más reducidas y, por tanto, más sencillas de manejar y procesar.

B.5. Editando una entrada

Para editar una entrada, en primer lugar, el cliente necesita obtener una representación editable de la misma. Para ello, es necesario realizar una petición GET a la URI de edición, tal y como se muestra en el siguiente listado de código. Esencialmente, esta acción es análoga a la de abrir un documento local en un editor de texto con la intención de editarlo.

```
GET /blog/entries/1 HTTP/1.1
Host: example.org
```

La respuesta a esta petición debería ser una entrada Atom como la siguiente.

```
HTTP/1.1 200 OK
Date: nnn
Content-Type: application/atom+xml; charset=utf-8
Content-Length: nnn
ETag: "/blog/entries/1?1"
Last-Modified: Fri, 27 May 2011 13:40:03 GMT
<?xml version="1.0" ?>
        <entry xmlns="http://www.w3.org/2005/Atom" >
        <id>tag:ejemplo.org,2011:/blog/entries/1</id>
        <title>Google Data Qt Client</title>
        <link href="http://ejemplo.org/2011/05/27/atom03"/>
        <link rel="edit" href="http://ejemplo.org/blog/entries/1" />
```

```
        <updated>2011-05-27T13:40:03Z</updated>
        <author>
                <name>Fernando Jiménez</name>
        </author>
        <summary>Ejemplo...</summary>
</entry>
```

Una vez que la representación editable es recibida, el cliente puede, por lo general, hacer cualquier modificación sobre la entrada que quiera, tras lo que realizará una petición PUT de vuelta a la URI del recurso para realizar la actualización del mismo.

```
PUT /blog/entries/1 HTTP/1.1
Host: example.org
Content-Type: application/atom+xml; charset=utf-8
Content-Length: nnnn
If-Match: "/blog/entries/1?1"
If-Unmodified-Since: Fri, 27 May 2011 13:40:03 GMT
<?xml version="1.0" ?>
        <entry xmlns="http://www.w3.org/2005/Atom" >
        <id>tag:example.org,2011:/blog/entries/1</id>
        <title>Google Data Qt Client</title>
        <link href="http://ejemplo.org/2011/05/27/atom03"/>
        <link rel="edit" href="http://ejemplo.org/blog/entries/1" />
        <updated>2011-05-27T13:40:03Z</updated>
        <author>
                <name>Fernando</name>
        </author>
        <summary>Ejemplo de edición...</summary>
</entry>
```

Del anterior listado, es destacable el uso de las cabeceras *If-Match* y *If-Unmodified-Since* en la petición PUT. Opcionalmente, el uso de estas cabeceras permite a las implementaciones del Protocolo Atom proteger contra modificaciones de sobreescritura que otros clientes podrían haber realizado en un miembro del recurso. Si ninguna de estas dos condiciones se cumple , el servidor debe rechazar la petición y notificar al cliente a cerca de el posible conflicto con el recurso que pretende modificar. Si las condiciones se cumplen y el servidor considera que las modificaciones emitidas por el cliente son aceptables, el primero enviará la respuesta satisfactoria correspondiente.

B.6. Eliminando entradas

Para que un cliente elimine un recurso de una colección, ha de enviar una petición DELETE a la URI de edición de la misma de este modo.

```
DELETE /blog/entries/1 HTTP/1.1
Host: ejemplo.org
```

Tras una petición DELETE satisfactoria, la entrada no debería volver a aparecer en la colección Atom y, por tanto, dejará de ser accesible o editable.

B.7. Añadiendo recursos multimedia a la colección

Es posible añadir recursos multimedia tales como fotografías, documentos, archivos de sonido, etc. a las colecciones ATOM. Tales recursos son denominados por la especificación del Protocolo de Publicación Atom *media-link entries* por el hecho de que al ser añadidos a la colección, el servidor crea una entrada Atom enlazada al recurso multimedia enviado por el cliente. Aunque, originalmente, fue diseñado con la idea de permitir únicamente a autores de Weblogs subir objetos multimeda como contenido de sus posts, el Protocolo de Publicación Atom soporta una larga variadad de recursos multimedia, haciéndolo ideal para un extenso rango de aplicaciones incluyendo:

- Podcasting
- Video blogging
- Repositorios fotográficos.
- Wikis
- Gestión documental.
- Repositorios XML
- Distribución de software.
- Aplicaciones de productividad, tales como Office Suites.
- Y un largo etc.

Para crear una entrada media-link, el cliente debe realizar un petición de POST a la URI de la colección, incluyendo, en lugar de una entrada Atom, una representación del recurso multimedia a enlazar, de este modo.

```
POST /blog/photos HTTP/1.1
Host: ejemplo.org
Content-Type: image/png
Content-Length: nnnn
Slug: Viaje a la playa...
{datos binarios de la imagen}
```

Si la colección puede almacenar el tipo de recurso multimedia creado por el cliente, lo almacena y crea una entrada Atom enlazando al recurso multimedia, tal y como se muestra en el próximo listado de código. La cabecera *Slug* contenida en la petición es una nueva cabecera HTTP introducida por la Especificación Atom y usada para asociar un nombre simple con el

recurso, que puede ser usado para una serie de propósitos varios a la hora de crear y manejar los recursos. Por ejemplo, el servidor puede usar el valor del *slug* al crear la URI del recurso o al configurar el valor del título de la entrada Atom. Esta cabecera puede ser empleada al postear entradas Atom o recursos multimedia, siendo este último el caso más común.

```
HTTP/1.1 201 Created
Date: nnnn
Content-Location: /blog/photos/viaje_playa
Location: /blog/photos/viaje_playa
Content-Type: application/atom+xml; charset=utf-8
Content-Length: nnnn
Slug: Viaje a la playa
ETag: "/blog/photos/viaje_playa?1"
Last-Modified: Fri, May 27 2011 14:11:04 GMT
<?xml version="1.0"?>
        <entry xmlns="http://www.w3.org/2005/Atom">
        <id>tag:example.org,2006:/blog/photos/viaje_playa</id>
        <title>Viaje a la palya</title>
        <link rel="edit"
        href="http://ejemplo.org/blog/photos/viaje_playa" />
        <link rel="edit-media"
        type="image.png"
        href="http://ejemplo.org/blog/photos/viaje_playa?media" />
        <updated>2011-05-27T14:11:04Z</updated>
        <author>
                <name>Fernando Jiménez</name>
        </author>
        <summary>Viaje a la playa</summary>
        <content type="image/png"
        src="http://blog.ejemplo.org/photos/viaje_playa" />
</entry>
```

Las entradas *Media-link* contendrán siempre un elemento cuyo atributo *src* proveerá la URI del recurso multimedia recientemente creado, que podrá ser utilizada públicamente para hacer referencia al mismo. Con el enlace *edit-media* separado, se puede identificar la URI que puede ser utilizada para actualizar el recurso multimedia.

B.8. Editando recursos multimedia

El método para editar un recurso multimedia posteado en una colección es, por lo general, identico al empleado para editar una entrada Atom. El primer paso es obtener una version editable del recurso mediante una petición GET a la URI especificada en el enlace *edit-media.*

```
GET /blog/photos/viaje_playa?media HTTP/1.1
Host: ejemplo.org
```

Una vez que la representacion editable es obtenida, el cliente realizará las modificaciones que crea convenientes, tras lo cual las enviará de vuelta mediante una petición PUT a la URI de

edición.

```
PUT /blog/photos/viaje_playa?media HTTP/1.1
Host: ejemplo.org
Content-Type: image/png
Content-Length: nnn
{nuevos datos binarios de la imagen}
```

B.9. Protegiendo colecciones

Mientras que el Protocolo de Publicación Atom no requiere que la implementación use autenticación alguna, es altamente recomendable que la incluye con el fin de prevenir que clientes malintencionados puedan crear o modificar objetos de la colección. Como mínimo, las implementaciones deberían ser capaz de usar autenticación HTTP básica y conexión TLS/SSL. En cualquier caso, en la práctica, los clientes Atom pueden soportar gran variedad de mecanismos de autenticación. Independientemente de dicho mecanismo empleado, de cualquier modo, los servidores deberían emplear desafío tipo HTTP para identificar el tipo de autenticación seleccionada. Por ejemplo, si un servidor recibe una petición no autorizada por parte de un cliente, el servidor debería responder con una respuesta *401 Unauthorized* incluyendo la cabecera *WWW-Authenticate*, de este modo.

```
HTTP/1.1 401 Unauthorized
Date: nnn
WWW-Authenticate: Basic realm="mi blog"
```

El cliente puede entonces reenviar la petición con la cabecera de autenticación apropiada.

```
POST /entries/blog HTTP/1.1
Host: ejemplo.org:443
Authorization: Basic SmFtZXM6bm90IG15IHJlYWwgcGFzc3dvcmQgOi0p
...
```

En el caso de Google Data, las autenticaciones soportadas, tal y como se verá más adelante en profundidad, son:

- OAuth 1.0 y, recientemente, 2.0 [1]
- OpenID [2]
- AuthSub [3]
- ClientLogin [4]

[1] http://oauth.net/
[2] http://openid.net/
[3] http://code.google.com/apis/gdata/docs/auth/authsub.html
[4] http://code.google.com/apis/accounts/docs/AuthForInstalledApps.html

Apéndice C

Autenticación

Google Data Qt Client implementa uno de los mecanismos de autenticación soportados por el Protocolo de Datos de Google para aplicaciones instaladas. Este mecanismo de autenticación está basado en OAuth 1.0 *[3]*, un estándar abierto para la autorización del uso de datos en aplicaciones. Las aplicaciones instaladas han de registrarse previamente con Google para poder usar OAuth.

C.1. El proceso de autenticación OAuth

¿Qué es OAuth?

Como bien resume Google, el protocolo OAuth proporciona una manera estándar de acceder a los datos protegidos de diferentes sitios web. O dicho de otro modo, OAuth es un protocolo abierto y estándar que permite a un sitio web (o aplicación) `X` acceder de un modo seguro, previa autorización del usuario, a datos de acceso restringido de dicho usuario albergados en otro sitio web Y mediante una API que soporta OAuth y que `Y` pone a disposición de X.

El aspecto más importante en cuanto a la seguridad que ofrece OAuth es que el sitio web `X` (OAuth Consumer) no almacena los credenciales de acceso (usuario y contraseña) que el usuario utiliza en su cuenta en el sitio web `Y` (Service Provider).

Usuarios, OAuth Consumer y OAuth Service Provider

Para poder hablar de OAuth, se necesitan tres partes, un servidor o proveedor de servicios, usuarios y un consumidor:

- *OAuth Service Provider* o proveedor de servicios OAuth: Sitios o servicios web que contienen información de usuarios cuyo acceso es restringido. Estos proveedores ponen a disposición de los desarrolladores una API que soporta el protocolo de autenticación OAuth. En el caso que tenemos entre manos, el proveedor de servicios es Google.

- *Usuarios*: Sin los usuarios, no existiría OAuth. Por usuario se entiende cualquier persona que tiene una cuenta de usuario en un Service Provider.

- *OAuth Consumer* o consumidor OAuth: Cualquier sitio o aplicación web, móvil o de escritorio que solicita permiso a un usuario para acceder a sus datos de acceso restringido que alberga un Service Provider. El usuario puede autorizar o denegar el acceso del consumer a sus datos. Es necesario que el consumer soporte el protocolo HTTP y que utilice la versión de OAuth que el Service Provider haya implementado.

API Key y Callback

Cada OAuth Service Provider debe proporcionar un API Key (un string de letras y numeros) para identificar que las peticiones que recibe mediante su API vienen de un OAuth Consumer autorizado, es decir, la aplicación cliente que pretende hacer uso de los servicios provistos.

A su vez, cada OAuth Service Provider deberá requerir una Callback URL, es decir, una dirección URL que apunte a la parte de la aplicación cliente que se encargará de procesar la respuesta de autorización (o desautorización) de acceso a los datos de la cuenta del Usuario en el OAuth Service Provider.

Diagrama de flujo

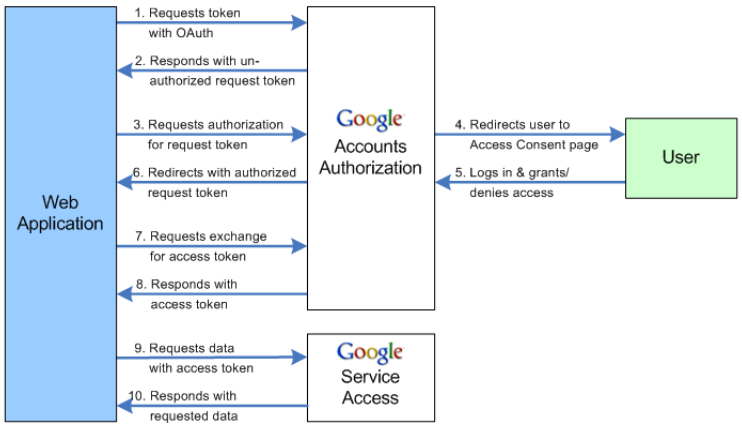


Figura C.1.1: Diagrama de flujo del proceso OAuth

Podemos resumir el flujo de autenticación OAuth 1.0 en los siguientes pasos:

- Partiendo de que la aplicación cliente ha de disponer de un Consumer Key y un Consumer Secret obtenido previamente a través del registro de la aplicación con Google, el primer paso del proceso es la obtención de un *Request Token*. De cara a Google Data Qt Client, la obtención de dicho token se realiza a través de la función *getRequestToken* de la clase *OAuth*, que recibe como parámetros una lista de las URLs de los servicios para los cuales se

pide autorización. En el caso de aplicaciones de escritorio, es posible realizar la llamada a dicha función sin haber registrado previamente la misma con Google. En tal caso, se usará el Consumer Key y el Consumer Secret por defecto: *anonymous*.

- Una vez obtenido el Request Token, es necesario solicitar, empleando dicho token, autorización al usuario para acceder a los datos de su cuenta. Para ello, el prooveedor del servicio ha de proveer una URL de login. En el caso de Google, dicha URL es *https://www.google.com/accounts/OAuthAuthorizeToken?oauth_token=* ,a la cual se le ha de concatenar el token anteriormente obtenido. A través de esta URL se accederá a una web como la siguiente.

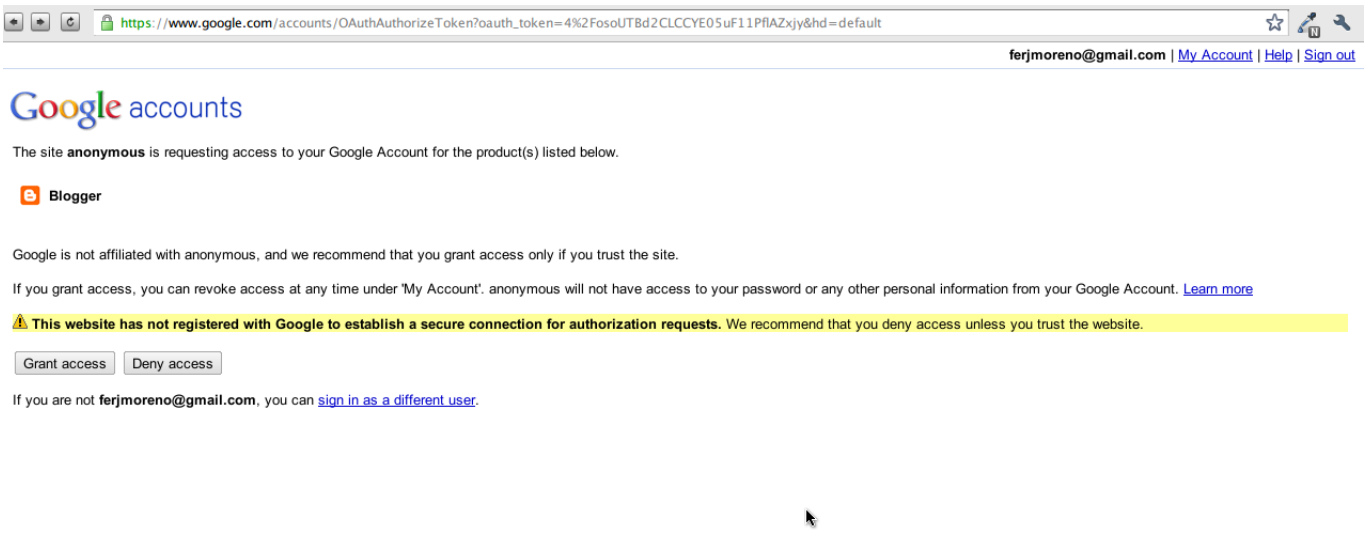


Figura C.1.2: Proceso de verificación de Request Token

- Una vez autorizado el acceso de la aplicación a los datos del usuario, se proporcionará un código de verificación, con el cual, se podrá solicitar el Access Token final para, finalmente, poder realizar las llamadas autorizadas a los diferentes servicios del proveedor. La petición del Access Token se realiza a través de la función *getAccessToken* de la clase *OAuth* de Google Data Qt Client.

Apéndice D

Contenido del CDROM y repositorio de código

D.1. Contenido del CDROM

En el CDROM anexo en el presente documento se incluye:

- Código fuente de la librería Google Data Qt Client, incluyendo el núcleo de la misma, los clientes, los tipos de datos y el archivo de proyecto Qt compilable.
- Documentación Doxygen en formato HTML.
- Código fuente de los tests unitarios.
- Código fuente de los demostradores del cliente del servicio Blogger y el servicio Google Code Search.
- Videos del funcionamiento de los demostradores.
- Copia digital en pdf de esta memoria.

D.2. Repositorio de código

El desarrollo de Google Qt Data Client se ha realizado empleando el sistema de control de versiones Git[1]. Existe un repositorio de código fuente público en Github[2], con mirror en Gitorious[3], desde donde se puede ver y descargar todo el código fuente de la librería, las pruebas unitarias y los demostradores de uso. Del mismo modo, en dicho repositorio se puede ver la línea temporal del desarrollo del proyecto, así como sus diferentes ramas.

[1] http://git-scm.com/
[2] https://github.com/ferjm/Google-Data-Qt-Client
[3] https://gitorious.org/google-data-qt-client

D.3. Licencias

Licencia del código

Todo el código fuente desarrollado en este proyecto, incluyendo las pruebas unitarias y los demostradores se distribuyen bajo licencia pública GNU General Public License version 3 (GPLv3). En todos los ficheros de código fuente publicados y distribuidos con el CDROM que acompaña a esta documentación se encuentra la siguiente cabecera que indica la licencia de dicho código fuente:

```
Copyright (C) 2010-2011 Fernando Jiménez Moreno <ferjmoreno@gmail.com>
This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 3 of the License, or
(at your option) any later version.
This program is distributed in thehope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.
You should have received a copy of the GNU General Public License
along with this program; if not, write to the
Free Software Foundation, Inc.,
59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
```

Licencia de la memoria

Esta memoria se distribuye bajo licencia Creative Commons (CC-BY-SA), que se resume en:

```
Usted es libre de:
* copiar, distribuir y comunicar públicamente la obra.
* Remezclar - transformar la obra.
* Hacer un uso comercial de esta obra
Bajo las siguientes condiciones:
* Reconocimiento: Debe reconocer los créditos de la obra de la manera especificada
     por el autor o el licenciador (pero de una manera que sugiera que tiene su
    apoyo o apoyan el uso que hace de su obra).
* Compartir bajo la misma licencia: Si altera o transforma esta obra, o genera una
     obra derivada, sólo puede distribuir la obra generada bajo una licencia
    idéntica a ésta.
Entiendo que:
* Renuncia: alguna de estas condiciones puede no aplicarse si se obtiene el
    permiso del titular de los derechos de autor.
* Dominio Público: cuando la obra o alguno de sus elementos se halle en el dominio
     público según la ley vigente aplicable, esta situación quedará afectada por
    la licencia.
Otros derechos: los derechos siguientes no quedan afectados por la licencia de
    ninguna manera:
* Los derechos derivados de usos legítimos u otras limitaciones reconocidas por
    ley no se ven afectados por lo anterior.
```

```
* Los derechos morales del autor.
* Derechos que puedan ostentar otras personas sobre la propia obra o su uso, como
    por ejemplo derechos de imagen o de privacidad.
```

La versión íntegra de la licencia puede descargarse en español de la web de Creative Commons[4].

[4] http://creativecommons.org/licenses/by-sa/2.5/es/legalcode.es

Bibliografía

[1] Google. Documentación del protocolo de datos, 2011.

[2] IETF. Especificación del protocolo de publicación atom.

[3] IETF. Especificación del protocolo oauth 1.0.

[4] Dave Johnson. *RSS and Atom in Action. Web 2.0 Building Blocks.* Manning, 2006.

[5] Leonard Richardson and Sam Ruby. *Restful Web Services.* O'Reilly, 2007.

[6] Ray Rischpater and Daniel Zucker. *Beginning Nokia Apps Development.* Apress, 2010.

[7] Mark Summerfield. *Advanced Qt Programming. Creating Great Software with C++ and Qt4.* Prentice Hall, 2010.

[8] Heinz Wittenbrink. *RSS and Atom. Understanding and Implementing Content Feeds and Syndication.* Packt, 2009.

[9] Erich Gamma y otros (Gang of Four). *Design Patterns: Elements of Reusable Object-Oriented Software.* Adison-Wesley, 1995.