Universidad del País Vasco
Euskal Herriko Unibertsitatea

Facultad de Informática

Grado de Ingeniería Informática

▪ Proyecto Fin de Grado ▪

Ingeniería de Software

Sistema de sincronización entre servicios/modelo de un portal Liferay contra entornos NoSQL

Yongliang Zhan Zheng

Junio 2016

Director: Oscar Díaz García

Resumen

En este proyecto fin de grado se ha desarrollado una solución de sincronización de datos entre un portal Liferay y Firebase, un servicio de base de datos NoSQL en la nube.

La solución desarrollada ha sido enfocada en resolver el problema para una aplicación en particular, pero se ha logrado una serie de bloques de carácter reusable, de tal forma que pueden emplearse para facilitar el desarrollo de futuras aplicaciones de carácter similar.

Para alcanzar la solución, ha sido necesario un aprendizaje mínimo de múltiples tecnologías; Liferay, Node.js, REST y JSON-RPC por mencionar algunas.

Índice

Índice de Figuras y Tablas

FIGURAS

TABLAS

Índice de extractos de código y comandos

EXTRACTOS

1

Introducción

En este documento se presenta el trabajo realizado por Yongliang Zhan en la empresa INDABA, bajo la supervisión y dirección de Aritz Galdos por parte de la empresa, y Oscar Díaz por parte de la Universidad del País Vasco.

El proyecto de fin de grado realizado forma parte de un proyecto mayor descrito más adelante, y busca dar solución a un problema de alcance limitado que surge a partir de dicho proyecto.

En este capítulo se hace una introducción a los inicios del proyecto y la motivación del mismo.

1.1. INDABA

INDABA Consultores S.L. es una empresa del grupo LKS. Apuesta por el software libre y pretende ser el referente del grupo LKS en tecnologías *Open Source*.

Surge en 1999 a partir de un grupo de investigación de la UPV/EHU, fundada por Ainhoa Uzkudun y Patricia Hernández junto con un tercer socio. Inicialmente, centra su actividad en el desarrollo de aplicaciones web basadas en tecnologías de código abierto.

En 2007 se incorpora al grupo LKS como una participada al 100%, con el objetivo de ser la marca asociada al software *Open Source* en el grupo.

Actualmente se dedica al diseño e implantación de sistemas de información basadas en tecnologías web con implementaciones flexibles y escalables, desarrollos basados en estándares evitando dependencia de proveedores y soluciones multiplataforma.

Además, también realiza aplicaciones en tecnología móvil con desarrollos para Android.

Las tecnologías más usadas en los proyectos de INDABA son Liferay como portal de gestión de contenidos, y MySQL como sistema de gestión de bases de datos, aunque siempre es capaz de emplear otros tipos de tecnologías siempre que sea necesario, como el framework Jboss Seam o sistemas de gestión de bases de datos Oracle.

1.2. LostAndFound Virtual Offices

LostAndFound Virtual Offices es un proyecto ideado por Aritz Galdos Otermin, ganador del premio regional en el concurso Galileo Masters Gipuzkoa 2014.

LostAndFound Virtual Offices pretende ser una red colaborativa de búsqueda y gestión de objetos perdidos. A las oficinas de objetos perdidos se les ofrece un servicio centralizado, pero a la vez personalizable, en forma de portal web en el que publicar los objetos perdidos que gestiona; y a los usuarios, una aplicación móvil con la capacidad de enviar alertas de objetos perdidos y visualizar otras alertas u objetos perdidos de la zona.

Los datos del sistema estarán sustentados principalmente en dos sistemas de bases de datos diferentes. El primero, será una base de datos MySQL, la cual dará soporte a los datos necesarios para implementar el portal web.

Por otro lado, y por su idoneidad para dar acceso a los datos a las aplicaciones móviles que estarán disponibles para los usuarios finales, será necesario un sistema de gestión de bases de datos no relacional. En este caso, y tras barajar distintas opciones, se ha seleccionado Firebase para este cometido.

Esta estrategia trae consigo una problemática que necesita ser solventada de la forma más eficiente posible. El hecho de trabajar con dos sistemas de bases de datos presenta una situación en la que se hará necesaria la gestión de la redundancia de los datos.

2

Objetivos del proyecto

El proyecto de fin de grado se ha enmarcado dentro del proyecto LostAndFound Virtual Offices presentado en el capítulo anterior. En concreto, se busca la solución al problema de emplear dos sistemas de bases de datos, de tal manera que la información entre los dos entornos descritos previamente puedan integrarse y sincronizarse.

2.1. Alcance del proyecto

El alcance del proyecto puede ser dividido en dos apartados.

Principalmente y como requisito mínimo, se busca resolver el problema de sincronización del modelo de datos que emplea LostAndFound Virtual Offices.

Para esto, se prevé la necesidad de alcanzar varios objetivos transversales:

- Aprender sobre el desarrollo para portales Liferay.
- Aprender el servicio que provee Firebase.
- Investigar los métodos de sincronización entre bases de datos.
- Identificar posibles problemas que pueden surgir en la comunicación entre las bases de datos.

Como requisito deseable o secundario, se busca el análisis y posible desarrollo de un módulo genérico que permita la sincronización con facilidad de futuras aplicaciones que empleen las mismas tecnologías que este proyecto, Liferay y Firebase, independientemente de las entidades que se buscase sincronizar.

Eso implicará, probablemente, aplicar los métodos de refactorización de código estudiados en la asignatura de Ingeniería de Software a la solución resultante anterior.

2.2. Exclusiones del alcance

Se excluyen de este proyecto de fin de grado los siguientes puntos:

- El desarrollo de las aplicaciones que componen el proyecto LostAndFound.
- La puesta en marcha de la solución en un entorno de producción.
- Aspectos de seguridad en la solución.

3

Herramientas y tecnologías

En este capítulo se describen de forma selectiva las principales herramientas o tecnologías empleadas en el proyecto para aportar el contexto tecnológico que forma parte del proyecto.

3.1. Liferay

Liferay es un portal de gestión de contenidos de código abierto escrito en Java. Entre otras formas, permite crear los portales mediante el añadido de portlets. Un portlet es un componente web diseñado específicamente para ser agregado en páginas web compuestas.[1]

Al desarrollo de aplicaciones en Liferay podría dedicarse un libro completo; a continuación se describe uno de los aspectos que más influye en este proyecto: el acceso a datos en una aplicación Liferay.

Aunque el acceso a datos a una aplicación Liferay puede construirse de la forma que se desee, generalmente, es recomendable utilizar Service Builder.

Service Builder es una herramienta de generación de código que permite al desarrollador definir las entidades de su aplicación. Genera, a partir de un fichero XML, las capas de modelo, persistencia y servicio, implementando las operaciones de lectura y escritura más sencillas.

El acceso a datos de una aplicación Liferay puede dividirse en tres capas (aunque realmente sólo una accede a la base de datos):

- La capa del modelo, formada por una serie de clases e interfaces que representan la entidad.
- La capa de servicio es el intermediario entre el controlador a la capa de persistencia.
- La capa de persistencia consta de los métodos necesarios para leer, insertar, actualizar y eliminar los objetos en la base de datos.

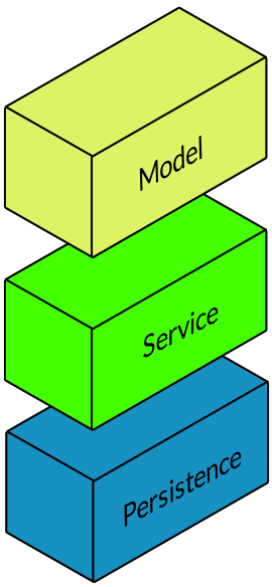

Figura 2.1. Capas de acceso a datos en aplicaciones Liferay
(imagen obtenida de la web de documentación de Liferay)

Así pues, la lógica de negocio de una aplicación Liferay, en cuanto al acceso a datos, se implementa en las clases que forman la capa de servicio. Estos servicios, además de ser métodos invocables desde el controlador, pueden publicarse como servicios web, pudiendo consumirse a través de peticiones HTTP.

3.2. Firebase

Firebase es una plataforma que facilita el desarrollo de aplicaciones, proporcionando a sus usuarios diversas funcionalidades. En este proyecto interviene únicamente el servicio de bases de datos. Se trata una base de datos NoSQL en la nube que almacena los datos en formato JSON, accesible mediante diversas APIs.[2]

3.2.1. Estructura de datos

Firebase almacena los datos como objetos JSON en forma de árbol. A diferencia de las bases de datos SQL, no hay tablas ni tuplas; cuando se añade un dato, se convierte en un nodo en el árbol. Por tanto, para poder recuperar los datos de forma eficiente, es indispensable definir una estructura. Con este fin, Firebase facilita una guía de buenas prácticas.

Una de las recomendaciones a tener en cuenta es la de crear referencias recíprocas en los datos relacionados de forma bidireccional. Esto es importante porque en las bases de datos relacionales es común justo lo contrario, es decir, que las referencias entre entidades sólo existan una vez por relación.[3]

Por poner un ejemplo, supongamos las entidades *user* y *group* (del inglés usuario y grupo, respectivamente), de tal forma que un *user* puede pertenecer a varios *group* y un *group* puede tener más de un *user.*

En un modelo relacional se recomienda emplear una relación o tabla que combine las referencias (Figura 3.2).

Sin embargo, en Firebase es más eficiente que cada grupo contenga las referencias a los usuarios que contiene y viceversa debido a las limitaciones en cuanto a consultas que presenta (Figura 3.3).

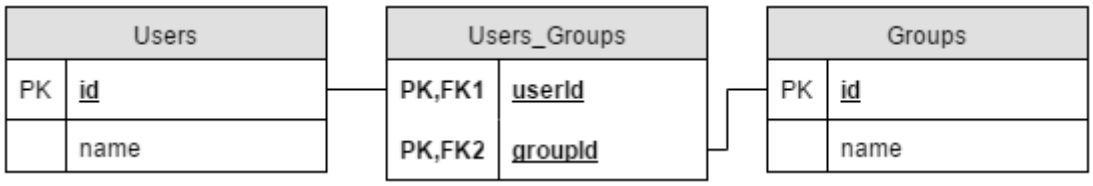

Figura 3.2. Ejemplo de esquema relacional

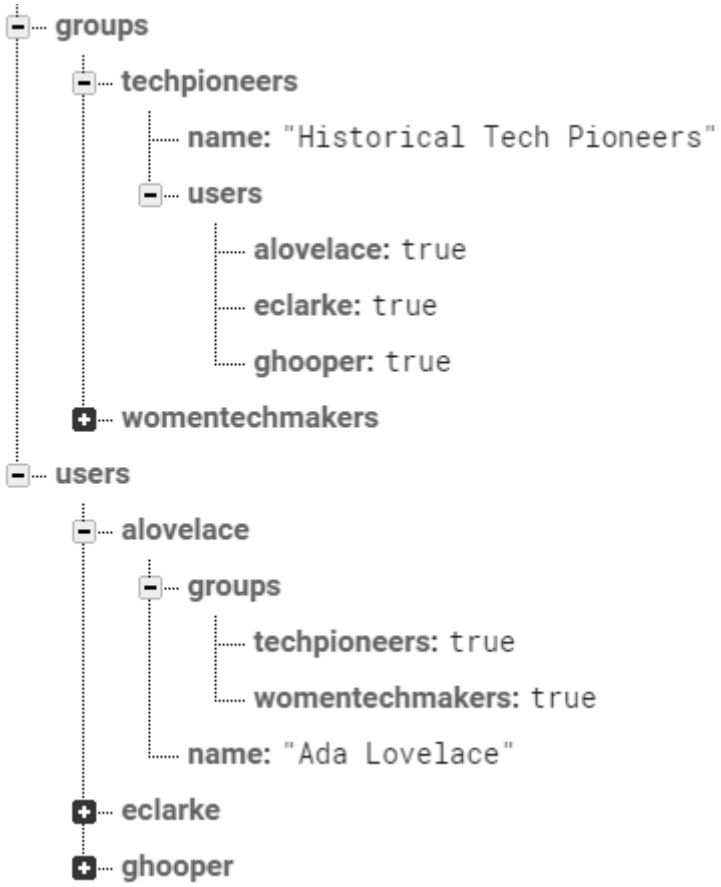

Figura 3.3. Ejemplo de estructura de datos Firebase

3.2.1. API de Firebase

Firebase proporciona múltiples APIs, además de diversas librerías, según la plataforma de desarrollo. Ofrece APIs con las mismas funcionalidades para desarrollo web (JavaScript), iOS y Android, y para el resto de plataformas ofrece la API REST, con ligeras diferencias en la funcionalidades.

Como la estructura de almacenamiento de datos es un árbol, cualquier nodo puede representarse con una URL. La API REST funciona enviando peticiones HTTP a dichas rutas, empleando los métodos PUT, GET, PATCH y DELETE para realizar las operaciones de escritura, lectura, actualización y borrado respectivamente. Por tanto, no es posible capturar eventos mediante esta API, y cualquier cambio en la base de datos debería comprobarse manualmente.

```
curl https://samplechat.firebaseio-demo.com/users.json
```

Extracto 3.1. Ejemplo de consulta Firebase por API REST mediante cURL

En cambio, las API de JavaScript, iOS y Android funcionan precisamente mediante manejadores de eventos; se agregan funciones *callback* a ciertos eventos, de tal forma que cuando ocurren dichos eventos se lanza la función correspondiente.

```
firebase.database().ref('/users/').once('value').then(
function(snapshot) {
  var users = snapshot.val();
  // ...
});
```

Extracto 3.2. Ejemplo de consulta Firebase por API JavaScript

3.3. Node.js

Node.js es un intérprete Javascript del lado del servidor que cambia la noción de cómo debería trabajar un servidor. Su meta es permitir a un programador construir aplicaciones altamente escalables y escribir código que maneje decenas de miles de conexiones simultáneas en una sólo una máquina física.[4]

En este proyecto, se emplea una aplicación Node.js para escuchar, de forma contínua, eventos de la base de datos Firebase y realizando acciones en consecuencia a éstos. Para esto, se hace uso de la API JavaScript de Firebase.

3.3. XSLT

XSLT es un lenguaje para transformar documentos XML en otros documentos XML, otros formatos como HTML, o texto plano. No modifica el fichero original, sino que crea un documento nuevo (o varios) a partir del contenido del mismo.[5]

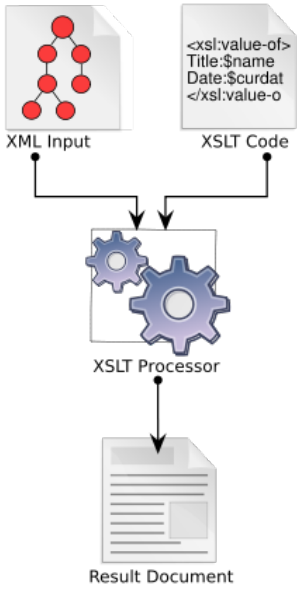

Figura 3.4. Transformación XSLT
(Imágen obtenida del artículos sobre XSLT en Wikipedia)

El uso más común de XSLT es para presentar de forma más visual los contenidos de ficheros XML particulares. En este proyecto, sin embargo, se emplea como medio de generación de código fuente, facilitando así la construcción de un tipo de aplicaciones en concreto.

4

Sistema de sincronización

Este capítulo presenta el desarrollo del trabajo realizado en el proyecto. Se comienza con una aglomeración del análisis realizado en las distintas fases del proyecto y, finalmente, se describe el proceso seguido para el desarrollo e implementación de la solución.

4.1. Soluciones existentes

El primer paso antes de desarrollar cualquier cosa es buscar soluciones existentes. En el caso del proyecto, no hay una solución ideal existente; no obstante, conocer la arquitectura de herramientas de propósito similar puede ser de gran ayuda. De entre distintas soluciones encontradas, la más interesante ha sido SymmetricDS por ser de código abierto e independiente de la aplicación de las bases de datos.

4.2.1. SymmetricDS

SymmetricDS es un software de código abierto para la sincronización de bases de datos y ficheros, con soporte para replicación multi-master, sincronización filtrada y transformación. Emplea tecnologías web y de bases de datos para replicar cambios en los datos como operaciones programadas o en tiempo-real.[6]

Aunque parece el software idóneo para resolver el problema, no es capaz de sincronizar datos en bases de datos no relacionales, por no mencionar que Firebase es un servicio en la nube y, por tanto, sólo se puede acceder al mismo mediante la API proporcionada.

Lo más interesante de esta herramienta es su arquitectura; a diferencia de otras arquitecturas que funcionan en un esquema maestro-esclavo, cada base de datos es responsable de una parte del movimiento de datos. Los pasos más relevantes en el proceso de sincronización son los siguientes:

1. Capturar los cambios en la base de datos origen.
2. Transformar los datos para que se adapten a esquema destino.
3. Enviar los datos a la base de datos destino.

4.2. Análisis y diseño

Aunque el análisis se realizó de forma incremental, siguiendo las fases descritas en la planificación, en este apartado se agrupan las características a considerar en la solución identificadas.

4.2.1. Modelo de datos

Antes de nada, es imprescindible conocer los datos que van a emplearse en la aplicación. Inicialmente, se definió un modelo, susceptible a cambios. En la figura 4.2 se presenta una versión simplificada de este modelo inicial que describe únicamente las entidades y sus relaciones.

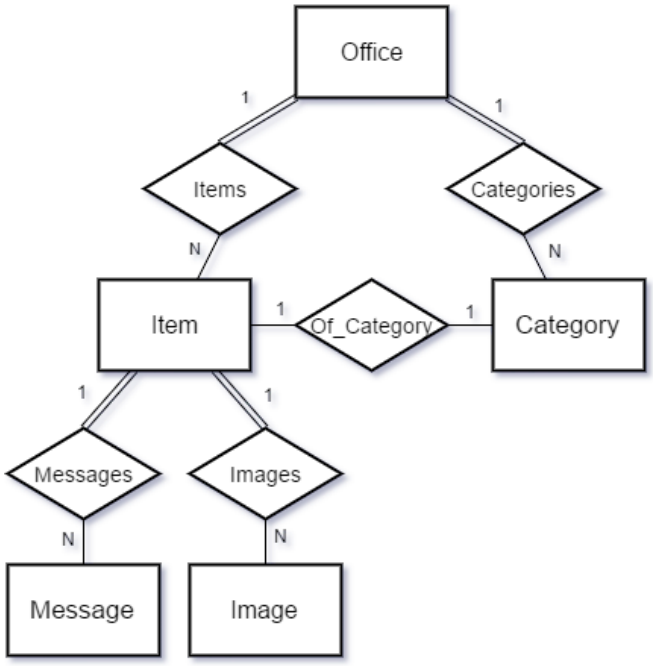

Figura 4.1. Diagrama ER del modelo de datos de LostAndFound VirtualOffices

Un aspecto importante del modelo es que para el desarrollo de los portales web, algunas de las entidades del modelo se representan mediante clases definidas por el propio sistema Liferay. En concreto, se trata de las entidades Message, Category y Office (denominada Group en Liferay). La implicación de este hecho es que para estas entidades, el esquema no puede modificarse y, además, estarán definidos los métodos de acceso a datos desde la aplicación.

Por otra parte, las entidades *Item*, *Message* e *Image* pueden ser modificadas de forma bidireccional, es decir, que se pueden modificar tanto en el portal web como en las aplicaciones móviles. Generalmente, los esquemas de sincronización más comunes tratan de evitar estas situaciones, limitando las capacidades de escritura a una única base de datos.

4.2.2. Estrategia de sincronización

Para desarrollar un sistema de sincronización entre bases de datos, es necesario conocer las herramientas que cada una de ellas ofrece.

En el caso de la base de datos relacional MySQL, se ha optado no hacer uso directo de ella con el fin de evitar conflictos con Liferay Portal. Por tanto, todo el acceso a esta base de datos se realiza mediante los servicios que ofrece Liferay Portal, descritos en el apartado 3.1.

En cuanto a la base de datos no relacional de Firebase, como se menciona en el apartado 3.2.2., ofrece distintas APIs con funcionalidades similares según el entorno de desarrollo.

Siguiendo la arquitectura de SymmetricDS, descrita en el apartado anterior, se busca un modelo de sincronización en el que cada base de datos capture los cambios realizados en la misma y, posteriormente, los envíe a la base de datos destino. Además, este modelo será también en tiempo real, es decir, se replicarán los cambios en el momento que se capturen. Para este fin, necesitamos tanto en Liferay como en Firebase algún mecanismo que capture los cambios realizados.

Aprovechando el hecho de que se emplea Liferay como medio de acceso a la base de datos MySQL, se emplea la propia aplicación Liferay como entidad sincronizadora. Para esto, sería necesario poder capturar los cambios en ambas bases de datos.

Los cambios en MySQL pueden capturarse fácilmente, ya que siempre se realizarán a través de la propia aplicación. En cuanto a Firebase, aunque se pueden leer los datos gracias a la API REST, no es posible capturar los eventos de escritura desde la aplicación Liferay. Por este motivo, se decidió desarrollar una aplicación Node.js dedicado a la captura de cambios en Firebase y el envío de estos a Liferay. En adelante se hará referencia a estos elementos como módulo de sincronización Liferay y módulo de sincronización Node.js.

La siguiente figura describe los participantes en la sincronización; las flechas indican la dirección en la que pueden enviarse los datos.

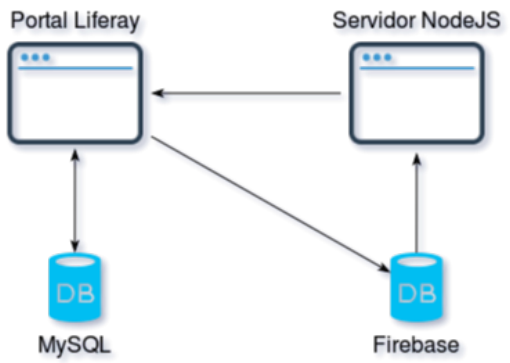

Figura 4.2. Flujo de datos en la solución

4.2.3. Sincronización bidireccional

Otro aspecto a considerar son los casos de sincronización bidireccional. Los datos que se deban sincronizar en ambos sentidos puede resultar en un ciclo infinito: al replicar el cambio en una base de datos destino, se detecta y replica otra vez en la base de datos origen, que otra vez podría detectarlo y replicarlo, creando un ciclo infinito. Para evitar esto, debemos tener algún método para identificar, tanto en Liferay como en el servidor Node.js, el origen de los cambios que detectan y decidir según esto si el cambio debería replicarse.

Para identificar el origen de las modificaciones en Firebase, podemos aprovechar que no exige una estructura fija en los datos para añadir un campo cuando los datos provienen de Liferay. En el caso de Liferay, es posible emplear uno de los parámetros que toma parte en la capa de servicios de la aplicación, que es el objeto *ServiceContext*, que permite, en la capa de servicios, identificar las peticiones remotas.

También hay que considerar la transformación entre entidades. En concreto, hay que considerar cómo gestionar las claves primarias de las entidades. El objetivo final es crear dos aplicaciones independientes; no debería existir una limitación de funcionalidad en ninguna de las aplicaciones cuando una de ellas deja de funcionar. Por tanto, hay que optar por mantener, en cada sistema, una gestión independiente de estas claves primarias.

Con esto surge un problema: si las entidades utilizan claves distintas para identificar a las instancias, ¿cómo se relaciona una instancia de entidad con su correspondiente réplica en cada base de datos?. Para resolver esto, otra vez se aprovecha el hecho de que Firebase no requiere un esquema, pudiendo añadir como atributo la clave que cada objeto emplea en Liferay. Así, una instancia en Firebase conoce la clave primaria que emplea la instancia correspondiente en Liferay, mientras que desde Liferay se puede obtener la clave primaria de cada instancia Firebase mediante una consulta.

4.3. Desarrollo del sistema

La implementación se realizó en una serie de fases incrementales, de naturaleza similar a las iteraciones de procesos de desarrollo ágil; cada fase resulta en una solución parcial al problema de sincronización. A continuación se describen los problemas resueltos en cada una de las fases.

4.3.1. Sincronización de una entidad

Como objetivo inicial, se buscó la forma de poder sincronizar, de forma bidireccional, entidades entre las dos aplicaciones. En concreto, se comenzó con la entidad *Item* mencionada en el apartado 4.2.1. Como se ha mencionado en el apartado anterior, esto supone, por un lado, crear un módulo Node.js que capture eventos Firebase, y por otro, añadir a la capa de servicio de Liferay la replicación de los objetos.

4.3.1.1. Modificando la capa de servicio

La creación del módulo de sincronización Liferay se realiza sobre el propio portlet de la aplicación LostAndFound Virtual Office. La entidad Item es una entidad propia, por lo que la capa de servicio se crea manualmente.

Así pues, tras cada uno de los métodos creados, se invoca un método que convierte el objeto de la entidad en JSON y realiza una llamada REST a Firebase. Para facilitar la llamada REST, se empleó una librería existente, *firebase4j*.

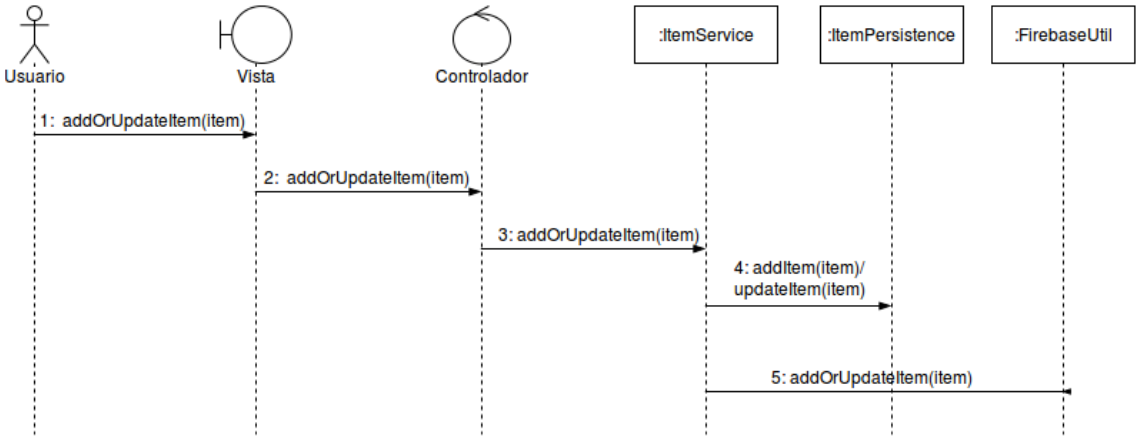

Figura 4.3. Diagrama de secuencia (simplificado) del CU insertar/actualizar *item* desde la aplicación Liferay

Para evitar que el módulo de sincronización Node.js repitiese el cambio realizado, al objeto convertido se le añadió un atributo especial para que dicho módulo pudiera tratarlo de forma especial al capturar el evento.

Además, se crearon unas funciones adicionales como servicios remotos, para que el módulo Node.js pudiera consumirlos. Estas funciones realizaban las mismas operaciones de inserción, modificación y eliminado, pero sin realizar las llamadas REST a Firebase.

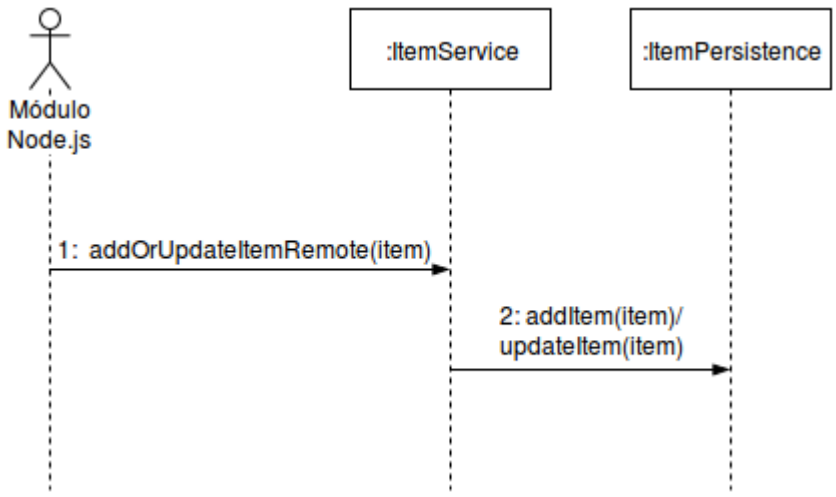

Figura 4.4. Diagrama de secuencia (simplificado) del CU insertar/actualizar *item* desde el módulo Node.js

Así pues, quedaron definidas en las clase de la capa de servicio 4 métodos:

- addOrUpdateItem(…): para ser usado en la propia aplicación Liferay, inserta o actualiza un *item* tanto en la base de datos de Liferay como en Firebase.
- deleteItem(…): para ser usado en la propia aplicación Lfieray, elimina un objeto tanto de la base de datos de Liferay como en Firebase.
- addOrUpdateItemRemote(…): para ser consumido por el módulo Node.js, inserta o actualiza un *item* en la base de datos de Liferay.
- deleteItemRemote(…): para ser consumido por el módulo Node.js, elimina un *item* en la base de datos de Liferay.

4.3.1.2. Creando el módulo Node.js

La construcción del módulo Node.js fue relativamente sencilla; con la API de Firebase para Node.js se adjuntan funciones callback a eventos de escritura en la rama correspondiente a la entidad y, según el evento, se realiza una llamada al servicio correspondiente de Liferay.

Para la consumición del servicio ofrecido por Liferay se hizo uso del protocolo JSON-RPC sobre HTTP.

Las funciones de la API capturan todos los eventos, incluyendo los cambios realizados desde el módulo Liferay. Como se ha mencionado, se emplea un atributo especial para identificar los objetos que inserta o modifica el módulo Liferay; así pues, en la función callback correspondiente se comprueba que el objeto de la escritura capturado sólo se replique en Liferay si no contiene dicho atributo. En caso de tenerlo, se elimina para que futuros cambios no sean ignorados.

4.3.2. Sincronización de referencias entre entidades

Tras lograr replicar las entidades en los dos sentidos, se trató de aplicar la misma estrategia a todas las entidades de la aplicación.

Como se ha mencionado previamente, algunas de las entidades son especiales, ya que son propias de Liferay. Para estas entidades, surge un problema al intentar repetir el método de replicación usado inicialmente: la capa de servicios ya está implementada. Por suerte, Liferay permite personalizar estos métodos, sin tener que modificar las entrañas del sistema, mediante un *hook*. Un *hook* permite sobreescribir las funciones de las clases de servicio propias de Liferay. Así, se puede aplicar otra vez la misma estrategia otra vez, realizando llamadas REST a Firebase tras invocar el método original.

Sin embargo, el *hook* no permite añadir métodos, por lo que no es posible crear las funciones adicionales para el uso en el módulo Node.js. El módulo Node.js tiene que consumir los mismos servicios que la aplicación Liferay, y éstos replican las operaciones en Firebase, generando un rebote indeseado. La forma de evitar este rebote reside en un parámetro común a casi todos los métodos de las clases del servicio: el *ServiceContext*. Éste es un objeto que

proporciona información sobre el contexto de la consumición del servicio y, por medio del mismo, es posible identificar las peticiones realizadas por el módulo de sincronización Node.js.

Por otro lado, las entidades que se sincronizan en un único sentido (*category, office)* también se tratan de un modo especial, ya que la gestión de estos objetos se lleva sólo en una aplicación. Para estas entidades, se emplea una única clave identificadora para ambas bases de datos. Además, no es necesario realizar una gestión de rebotes, ya que sólo un módulo tratará las modificaciones del mismo.

Finalmente, está la representación de relaciones en Firebase. En esta fase se trató añadiendo las referencias necesarias al objeto JSON antes de enviar la petición REST; antes de añadir, modificar y eliminar cualquier entidad, se obtenían las entidades relacionadas y se añadían referencias a ellas en una representación JSON. En casos necesarios, se obtenían desde Firebase las referencias antiguas, para poder eliminar las referencias recíprocas antiguas.

Por parte del módulo Node.js también hay que realizar unos cambios, además de añadir los capturadores de eventos correspondientes, es necesario resolver posibles referencias a otras entidades ya que, en casos de entidades bidireccionales, emplean claves primarias distintas, generadas por su propio sistema, como se ha mencionado en el apartado 5.1.

4.3.3. Refactorización del sistema

La última iteración del proyecto trata de refactorizar el código para extraer clases genéricas, de tal modo que pudiera crearse una librería de clases para un uso posterior.

En Liferay, se ha creado una clase encargada de replicar cada entidad de la aplicación en Firebase. Estas clases son extremadamente similares, así que el primer paso en la refactorización trata de emplear la programación genérica, de tal modo que cada una de las clases pasa a ser una instancia de una clase genérica. Evidentemente, hay aspectos que difieren entre estas clases de utilidad; los parámetros como nombre de la entidad o URL de Firebase destino se solucionan definiéndose en la función constructora, pero los distintos comportamientos se solucionan mediante el patrón de diseño Template Method. En concreto, se identifican dos tipos de clases con comportamientos distintos: las de sincronización bidireccional y las unidireccional. Estas resultan en una subclase cada una y, para facilitar la instanciación, se sigue el patrón Factory.

Una versión simplificada (obviando múltiples métodos y parámetros) del diagrama de clases es el siguiente:

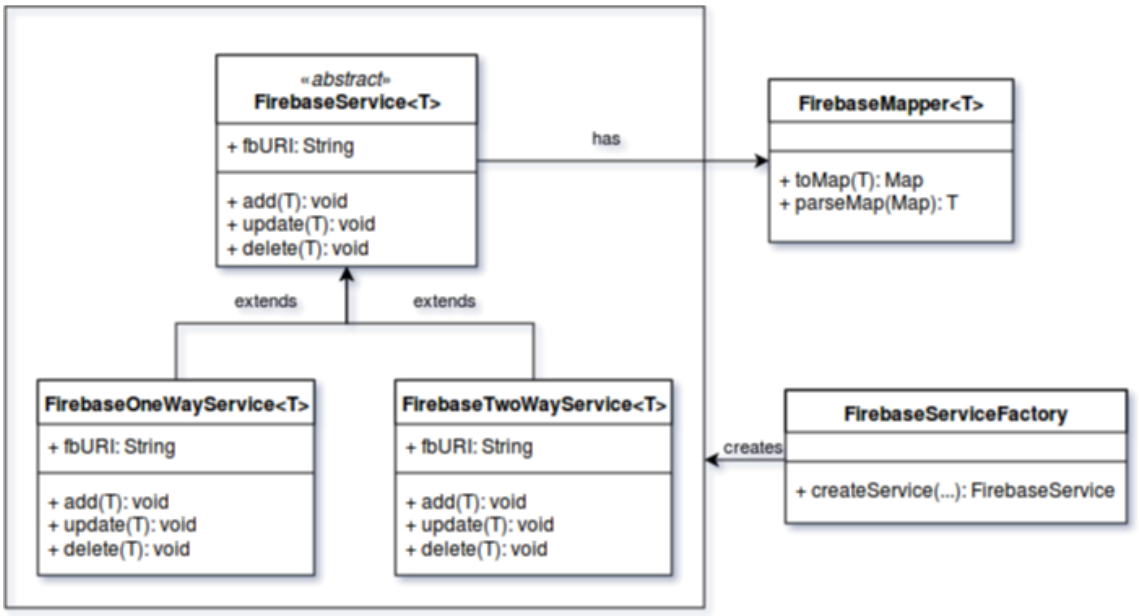

Figura 4.5. Diagrama de clases Liferay simplificado

En el módulo Node.js también se realiza una refactorización similar, empleando siempre el patrón de diseño Prototype, ya que JavaScript no es un lenguaje orientado a objetos. Se han definido dos "clases"; una para facilitar las llamadas a los servicios de Liferay y otra encargada de capturar y tratar los eventos de Firebase.

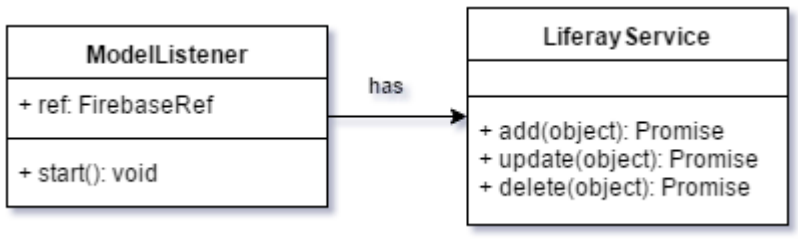

Figura 4.6. Diagrama de clases Node.js

4.4. Generación de código

Uno de los objetivos a alcanzar en el proyecto era lograr una solución reutilizable. Por ello, se han creado clases Java genéricas para facilitar la sincronización de los datos desde Liferay a Firebase y clases prototipo JavaScript para la construcción del módulo Node.js.

Sin embargo, se optó por dar un paso más, empleando XSLT para generar el código de cualquier sistema de sincronización Liferay-Firebase, a partir de un fichero XML.

Inicialmente, se pretendía generar el código tanto del módulo Node.js como del módulo Liferay; sin embargo, la generación del módulo Liferay resultó ser muy impráctica por varios motivos: por un lado, Java es un lenguaje fuertemente tipado y, por otro, la creación de un portlet Liferay depende de otros componentes generadores de código. Así pues, considerando además que en INDABA disponen de amplia experiencia de desarrollo con Liferay pero no con Node.js, se tomó la decisión de limitar la generación de código al módulo Node.js.

Para definir el esquema del documento XML se siguió un proceso inspirado en la creación de DSL: primero se identificaron los conceptos mediante un diagrama de características y, a partir de éste, se definió un XSD.

4.4.1. Diagrama de características

En principio, el diagrama de características se define para capturar los conceptos y variabilidades del dominio; en este caso, sincronización de datos desde una base de datos Firebase a un portal Liferay.

La definición de conceptos fue relativamente sencilla, gracias al hecho de tener el modulo Node.js definido; extraer los conceptos básicamente trata de identificar las clases y variables que definen el módulo.

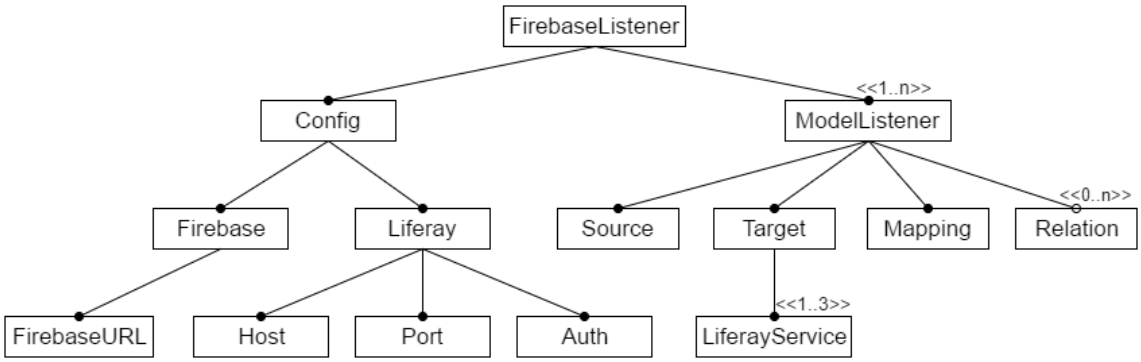

Figura 4.7. Diagrama de características FirebaseListener

Una característica notable en el diagrama de características (Figura 4.7) es la falta de componentes opcionales. Esto representa una carencia de variabilidad en el dominio y fue el principal motivo de emplear XML como lenguaje de definición del dominio, en lugar de crear un lenguaje específico de dominio (DSL) como se pretendía inicialmente.

4.4.2. Definición de esquema XML

Una vez de definido el diagrama de características, el siguiente paso fue definir el esquema de los documentos XML. Se escogió emplear el formato XSD (XML Schema Definition) para esta tarea, ya que aporta varias funcionalidades deseables como los tipos de datos URI o restricción de ocurrencias de elementos.

El esquema definido básicamente replica los conceptos definidos en el diagrama de características, empleando los tipos de datos necesarios para facilitar la representación. El resultado es un fichero bastante complejo, por lo que se ha decidido no incluirlo en esta memoria. En cualquier caso, está disponible en el repositorio GitHub correspondiente al proyecto firebase-listener.

4.4.3. Definición de la transformación

Con el esquema definido, el último paso para la generación del código fue crear el fichero XSL que realizase la transformación del fichero XML a los fichero de código fuente necesarios.

El objetivo era generar código fuente, sabiendo que aplicaciones del mismo dominio serían muy similares. Al tener definida una aplicación, el proceso se hizo bastante sencillo; bastaba con emplear el código disponible como plantilla y completar con los datos proporcionados en el fichero XML.

Un aspecto deseable en el código generado es que esté modularizado en ficheros distintos. Con las primeras versiones de XSLT, no hubiera sido posible lograr este objetivo, pero XSLT 2.0 define una nueva instrucción *result-document* que permite definir los ficheros de salida en la transformación.

Finalmente, para realizar la transformación en sí, fue necesario obtener el procesador XSLT. Se escogió para este fin el procesador XSLT Saxon, por ser una solución gratuita de código abierto.

5

Gestión del proyecto

En este capítulo se detalla la gestión del proyecto.

5.1. Planificación inicial

Debido a la falta de conocimiento sobre el contexto del proyecto, la planificación inicial se realizó de forma abstracta. El objetivo era seguir, sin mucha rigidez, una metodología ágil,

revisando y modificando la planificación de forma periódica, a medida que se identifican nuevas tareas.

Así pues, en lugar de un diagrama Gantt, como suele ser de costumbre, se definieron únicamente la EDT y los hitos deseados.

5.1.1. EDT

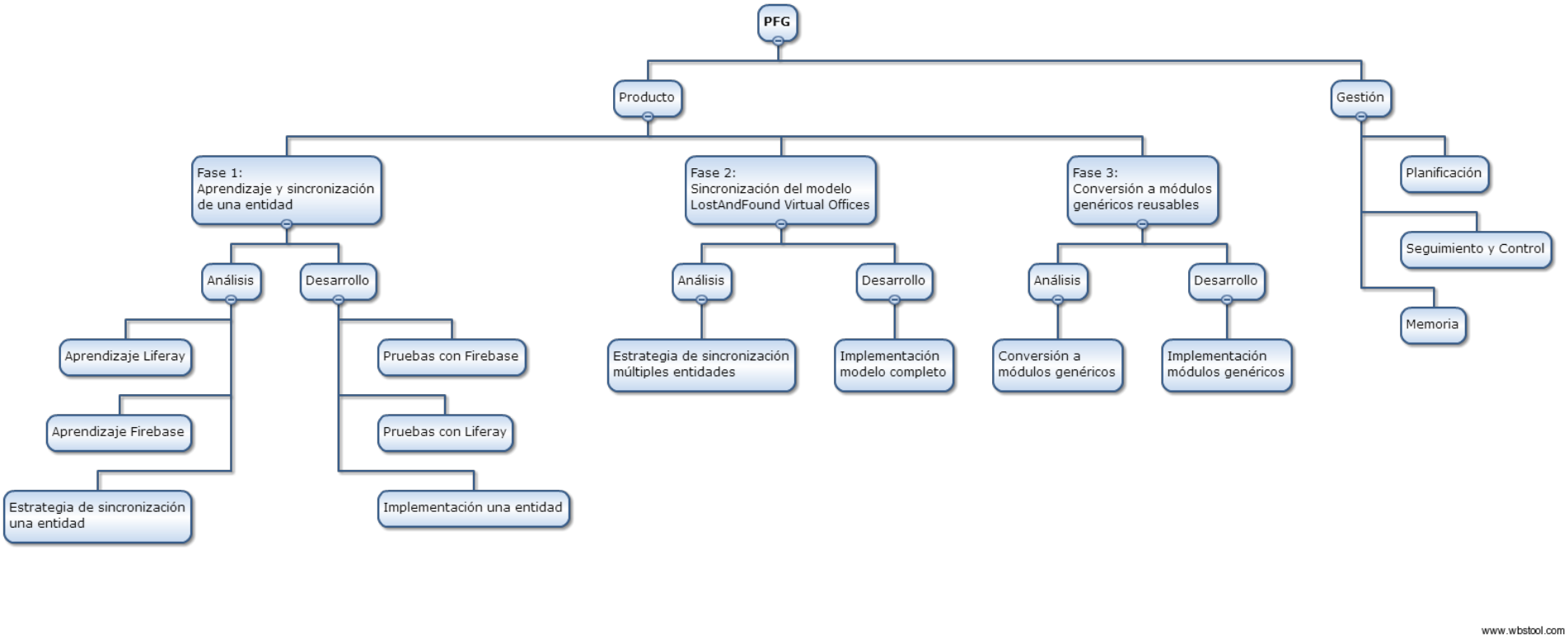

Figura 5.1. Estructura de Descomposición del Trabajo

5.1.2. Hitos

Los hitos definidos al inicio del proyecto se realizaron basándose en una estimación del trabajo de coste decreciente (se preveía la disminución de costes en cada fase consecutiva) y suponiendo una dedicación semanal de 20 horas.

- 1 de Febrero: Inicio del proyecto, Fase 1
- 28 de Marzo: Fin Fase 1, inicio Fase 2
- 18 de Abril: Fin Fase 2, inicio Fase 3
- 9 de Mayo: Fin Fase 3, Inicio Memoria
- 16 de Mayo: Fin de Proyecto

5.2. Seguimiento y control

Desde el inicio del proyecto, se preveía desviaciones en los plazos de significativas con respecto a la planificación inicial.

Los hitos propuestos fueron sorprendentemente alcanzados en una gran medida, pese al hecho de que no se pudo lograr la dedicación establecida inicialmente.

Hito	Fecha prevista	Fecha real
Inicio del proyecto	01/02/2016	01/02/2016
Fin de Fase 1	28/03/2016	15/03/2016
Fin de Fase 2	18/04/2016	17/04/2016
Fin de Fase 3	09/05/2016	24/05/2016
Fin de Proyecto	16/05/2016	14/05/2016

Tabla 5.1. Comparativa entre hitos establecidos y logrados

El principal causante del enorme desvío en los últimos hitos se debe a la subestimación de los costes de tareas de tipo analítico; se esperaba poder terminar este tipo de trabajos en un menor tiempo en comparación a las tareas de implementación.

La realidad fue exactamente la contraria: las tareas de desarrollo e implementación, concentradas principalmente en las fases 1 y 2, requirieron mucho menos tiempo del previsto, mientras que las de mayor análisis e investigación, la fase 3 y la redacción de esta memoria, resultaron ser muy costosas.

Paquete de trabajo	Coste (horas)
Fase 1	90
Fase 2	60
Fase 3	120
Memoria	50
Total	320

Tabla 5.2. Costes de tiempo por paquete de trabajo

Hay que tener en cuenta que en estas tablas no se han considerado muchas horas dedicadas a la gestión, como la planificación y replanificación, el seguimiento o las reuniones llevadas a cabo, puesto que no se consideraron siquiera.

5.3. Conclusiones sobre la gestión

Como se puede apreciar en los apartados anteriores, la gestión del proyecto ha dejado mucho que desear.

Para empezar, un error grave cometido fue no establecer un método de trabajo en cuanto a la gestión. En retrospectiva, hubiera beneficiado mucho dedicar, desde el principio, media hora al día para recoger lo trabajado en el día, a modo de diario.

También ha tenido que ver en las desviaciones que en la planificación no se considerara la semana festiva en abril, aunque al final no supuso un retraso porque el coste estimado para las fases iniciales resultó ser menor.

Por último, algo sorprendente ha sido la falta de exigencia en cuanto a metodologías de desarrollo y gestión por parte de la empresa. Después de cursas las asignaturas de Gestión de Proyectos y Calidad del Software, no es inconcebible pensar que, en un nivel profesional, lo más común sería implantar métodos como los que se describían en las asignaturas mencionadas. Sin embargo, resulta que no es lo más común encontrarse con conceptos como *sprints* ni realizar todos los pasos en la gestión de proyectos como se describían en Gestión de Proyectos.

6

Conclusiones

En este capítulo se reúnen las conclusiones obtenidas tras la realización del proyecto.

Por un lado, están las conclusiones personales en cuanto al proyecto, lo que se ha aprendido; y por otro, las conclusiones en cuanto a la solución desarrollada, el grado en el que se han alcanzado los objetivos del proyecto y posibles mejoras.

6.1. Conocimientos adquiridos

Si hay algún aspecto ha abundado en este proyecto, es sin duda el gran número de nuevas tecnologías que han intervenido, ya sea superficial como profundamente.

Para empezar, está Liferay. Una herramienta de la cual no había oído hablar en absoluto antes de comenzar el proyecto, y resulta que es algo muy usado para crear los sitios web de entidades empresariales. Incluso los sitios web de la UPV/EHU están creados con Liferay.

Aunque en el proyecto no ha intervenido mucho el desarrollo de *portlets* Liferay, inevitablemente he aprendido, de forma superficial, sobre el proceso. Especialmente, me ha llevado a aprender sobre el núcleo de Liferay el hecho de que, en el proyecto, surgieran algunos problemas inesperados a la hora de consumir cierto servicio web ofrecido por el portal por defecto. Este problema llevó a la necesidad de indagar en el código fuente de Liferay, disponible en un repositorio público de GitHub, para poder determinar una solución.

Por otro lado, el desarrollo en Node.js ha supuesto la necesidad de aprender sobre el infame *callback hell*, que surge debido a la naturaleza orientada a eventos del sistema.

Finalmente, sobre la generación de código; aunque se decidió utilizar XSLT, en un momento se planteó la idea de emplear una estrategia de desarrollo dirigido por modelos, con el fin de desarrollar un DSL. Evidentemente, no se siguió adelante con eso por distintos motivos, pero el proceso de considerarlo supuso una tarea de aprendizaje considerable.

6.2. Experiencia personal

Realizar un proyecto en un entorno profesional es sin duda una experiencia diferente a la que un estudiante puede tener trabajando con otros alumnos.

LKS no exigió una dinámica de trabajo regular, permitiendo un grado de libertad que, probablemente, otras empresas no darían. Sin embargo, ha habido suficiente contacto con la empresa como para que pudiera tener una toma de contacto beneficiosa con el mundo profesional del desarrollo de software.

Además, al ser el proyecto fin de grado parte de un proyecto en desarrollo más grande, fue necesaria la coordinación en varias ocasiones con otros miembros de la empresa para acordar aspectos como el modelo o la estructura de datos.

En general, considero que el proyecto ha aportado, sin duda, una experiencia gratificante.

6.3. Líneas futuras y propuestas de mejora

Considerando el alcance establecido al inicio del proyecto, puede decirse que se han logrado todos los objetivos propuestos, de una manera satisfactoria.

Sin embargo, como todo producto de software, este también puede ser mejorado, por no mencionar que debería ser mantenido. Aunque el proyecto fin de grado ha concluido, no se puede decir lo mismo sobre el proyecto en el que ha formado parte, LostAndFound Virtual Offices.

El primer aspecto mejorable del proyecto sería el de la definición de pruebas, preferiblemente unitarias, de los módulos desarrollados. La solución desarrollada ha funcionado sin problemas en un entorno local, pero con un número de pruebas limitado. Si en un futuro surgiera un error en el entorno de producción, la detección del problema y la corrección debería hacerse de forma manual.

Otro aspecto importante que se ha excluido del proyecto ha sido el de la seguridad. En la solución actual el acceso al servicio de bases de datos de Firebase se realiza con permisos de administración y eso muy probablemente supone problemas de seguridad.

Relacionado con lo anterior, en el modelo de datos de la aplicación sincronizada no se han tenido en cuenta los usuarios, pero es innegable que deberían formar parte del mismo y, por tanto, también debería sincronizarse. Ya se comentó este problema superficialmente en alguna ocasión, pero se optó por apartarlo porque se hace uso de los sistemas de autenticación provistos por Liferay y Firebase, en lugar de crearlos.

Además, hay que mencionar que tanto Liferay como Firebase son proyectos activos y seguro que en el futuro publican actualizaciones. De hecho, a lo largo del proyecto, se ha actualizado en varias ocasiones la versión de Liferay, ya que se comenzó con una versión beta. En cuanto a Firebase, recientemente fue adquirida por Google y actualizaron las API. Algo que afecta al proyecto actualmente es la nueva API para Java que han publicado, que podría permitir realizar la sincronización a través de un único módulo en forma de *portlet* de Liferay.

Por último, está el aspecto de generación de código, que no se había planteado al inicio del proyecto. La idea surgió debido a la frecuencia de repetición del código y, como se ha mencionado previamente, inicialmente se pretendía crear un DSL. La idea fue brillante, pero, lamentablemente, la ejecución fue muy apresurada y el resultado deja algo que desear. Hubiera sido conveniente realizar una investigación más profunda sobre las posibilidades de generación de código, para determinar si sería posible hacer la generación para todos los módulos que participan en la sincronización.

Apéndice A:
Código fuente

Para el desarrollo de este proyecto se ha hecho uso de la herramienta de control de versiones Git. Todo el código generado durante el proyecto está disponible públicamente en dos repositorios GitHub.

El repositorio que contiene el código fuente del *portlet* Liferay, incluyendo la parte correspondiente a la sincronización de datos es la siguiente:

https://github.com/yzhan94/lfvo-portlet.git

El código del módulo Node.js y el componente de generación de código mediante XSLT está disponible en el siguiente repositorio:

https://github.com/yzhan94/lfvo-firebase-listener.git

Apéndice B:
Gestión de dependencias

A la hora de desarrollar software, es extremadamente común emplear librerías creadas por terceros para facilitar el trabajo. Cuando el trabajo desarrollado es en sí mismo una librería y se quiere ofrecer al uso público, es necesario indicar, de alguna forma, las dependencias que tiene el software desarrollado.

En este proyecto se ha dado precisamente este caso, al emplear la librería *firebase4j*, para facilitar la comunicación con la API REST de Firebase desde Liferay. Para usar *firebase4j*, es necesario adquirir también otras librería de las cuales ésta depende y, a su vez, probablemente esas librerías también dependan de otras. Esta jerarquía de dependencias esta gestionada con Maven.

Por otro lado, la API JavaScript de Firebase también requiere de ciertas librerías (o módulos, como se les llama en Node.js) y, en este caso, están manejadas por el gestor de paquetes de Node.js NPM.

A continuación se describen superficialmente las herramientas mencionadas y su uso como gestores de dependencias.

B.1. Maven

Apache Maven es una herramienta para construir y gestionar proyectos Java. Permite construir los proyectos a partir del POM (del inglés *project object model*, objeto modelo del proyecto) de una forma común para los proyectos que integren Maven, facilitando así el proceso. [7]

Si un proyecto está integrado con Maven y tiene un fichero pom.xml, la instalación de dicho proyecto se puede realizar fácilmente mediante la instrucción presentada en el extracto B.1. (suponiendo que se emplea la CLI de Maven). Esto generaría el comprimido JAR en un directorio local, permitiendo usarlo en proyectos Maven propios.

```
mvn install
```

Extracto B.1. Comando instalación de proyecto mediante Maven

Este es precisamente el caso de *firebase4j*; es un proyecto de código abierto que emplea Maven para su construcción. Descargando el proyecto de su repositorio en GitHub [https://github.com/bane73/firebase4j], se puede instalar utilizando el comando previamente mostrado.

Sin embargo, por simplificar, este proyecto de fin de grado no integró Maven para simplificar el desarrollo. En su lugar, tras instalar la librería, se copiaron manualmente todas las dependencias desde el repositorio local de Maven (por defecto el directorio .m2 en la carpeta personal del usuario), incluyéndolas en el proyecto desarrollado.

B.2. NPM

NPM facilita a los desarrolladores JavaScript compartir el código que han creado para resolver problemas particulares, y a otros desarrolladores reutilizar ese código en sus propias aplicaciones.

Estos fragmentos de código se conocen como paquetes o módulos. Se tratan, sencillamente, de un directorio con uno o más ficheros entre los cuales está el *package.json* que describe el módulo.[8] En este fichero también se especifican las dependencias del módulo y, mediante la CLI de NPM, instalar dichas dependencias (por defecto en un directorio local llamado node_modules).

```
npm install
```

Extracto B.2. Comando instalación de módulo Node.js

De hecho, para emplear la API JavaScript de Firebase, ha sido necesario definir un fichero *package.json* para el módulo de sincronización Node.js.

Apéndice C:
XSLT mediante SaxonHE

Para el desarrollo del apartado de generación de código se empleó la herramienta Oxygen XML Editor para definir tanto la definición de esquema XML como el fichero de transformación XSL..

Oxygen XML Editor es una herramienta que facilita mucho el desarrollo pero, desgraciadamente, es de pago. Por tanto, fue necesario buscar una alternativa gratuita para la generación del código. Una búsqueda rápida resultó en el procesador XSLT Saxon, que dispone de una versión gratuita *Home Edition*.

Para realizar la transformación XSLT, Saxon ofrece diversas maneras; en este proyecto se empleó la forma más sencilla: ejecutar XSLT a través de la línea de comandos.

```
java -jar dir/saxon9he.jar -s:source.xml -xsl:style.xsl
```

Extracto C.1. Comando para ejecutar XSLT en la línea de comandos

Con el comando presentado en el Extracto C.1 se realiza la transformación del fichero fuente *source.xml* según el fichero de transformación *style.xsl*. Si el resultado de la transformación tuviera una salida no definida mediante la función *result-document*, se imprimiría en consola.

Referencias

[1] O'Reilly Media. (7 jun. 2016). *What Is a Portlet* [en línea]. Available: http://www.onjava.com/pub/a/onjava/2005/09/14/what-is-a-portlet.html

[2] Firebase. (7 jun. 2016). *Firebase Realtime Database* [en línea]. Available: https://firebase.google.com/docs/database/

[3] Firebase. (7 jun. 2016). *Structure Your Database* [en línea]. Available: https://firebase.google.com/docs/database/web/structure-data

[4] IBM. (7 jun. 2016). *¿Simplemente qué es Node.js?* [en línea]. Available: https://www.ibm.com/developerworks/ssa/opensource/library/os-nodejs/

[5] W3C (7 jun. 2016). Transformation [en línea]. Available: https://www.w3.org/standards/xml/transformation

[6] SymmetricDS. (7 jun. 2016). *User Guide – Introduction* [en línea]. Available: http://www.symmetricds.org/doc/3.7/html/user-guide.html#_introduction

[7] Maven. (13 jun. 2016). *What is maven?* [en línea]. Available: https://maven.apache.org/what-is-maven.html

[8] npm. (13 jun. 2016). *What is npm?* [en línea]. Available: https://docs.npmjs.com/getting-started/what-is-npm

Glosario

API Del inglés, Application Programming Interface. El conjunto de funciones que ofrecen acceso a servicios o funcionalidades particulares, sin necesidad de conocer la implementación.

Árbol En ciencias de la computación, una estructura de datos de naturaleza jerárquica. Está compuesta por una raíz de la cual cuelgan hijos que, a su vez, pueden tener múltiples hijos de forma recursiva.

Callback Una función que se emplea como argumento de otra función. Cuando se ejecuta la segunda función, esta realizará en algún momento la llamada a la función pasada como argumento.

CLI Del inglés, command line interface. Herramientas que se usan desde la línea de comandos.

Controlador Componente del patrón de arquitectura software Modelo-Vista-Controlador (MVC). Recibe las acciones del usuario, accediendo al modelo o la vista para realizar las operaciones de lógica de negocio necesarias.

DSL Del inglés, Domain Specific Language. Lenguaje de programación de uso específico para dominios de alcance limitado; por ejemplo, SQL.

Entidad En Bases de Datos, algo que representa un concepto del mundo real.

Framework Estructura conceptual y tecnológica de soporte definido que puede servir de base para la organización y el desarrollo de software.

Git Software de control de versiones diseñado por Linus Torvalds.

GitHub Plataforma de desarrollo colaborativo para alojar proyectos mediante Git.

JSON Del inglés, JavaScript Object Notation. Formato de texto ligero para la transmisión de datos que consisten en pares de atributos-valores.

Open Source Expresión con la que se conoce al software distribuido y desarrollado libremente. Usado de forma intercambiable con el término *de código abierto*.

MySQL Sistema de Gestión de Bases de Datos Relacional de código abierto desarrollado por Oracle Corporation.

NoSQL Hace referencia a los Sistemas de Gestión de Bases de Datos que no emplean SQL como el lenguaje principal de consultas.

REST Del inglés, Representational State Transfer. Actualmente se usa para describir cualquier interfaz entre sistemas que utilice el protocolo HTTP para obtener o realizar operaciones sobre datos.

SQL Del inglés, Structured Query Language. Es el lenguaje de programación empleado para gestionar los datos en los Sistemas de Gestión de Bases de Datos Relacionales.

XML Del inglés, Extensible Markup Language. Es un lenguaje de marcas utilizado para almacenar los datos de forma legible tanto para las personas como para las computadoras.

XSD Del inglés, XML Schema Definition. Recomendación del W3C que especifica cómo describir los elementos que puede contener un fichero XML.

XSL Del inglés, Extensible Stylesheet Language. Se refiere a la familia de lenguajes empleados para la visualización de ficheros XML. Generalmente, se emplea como la extensión de ficheros que contienen transformaciones XSL (XSLT).