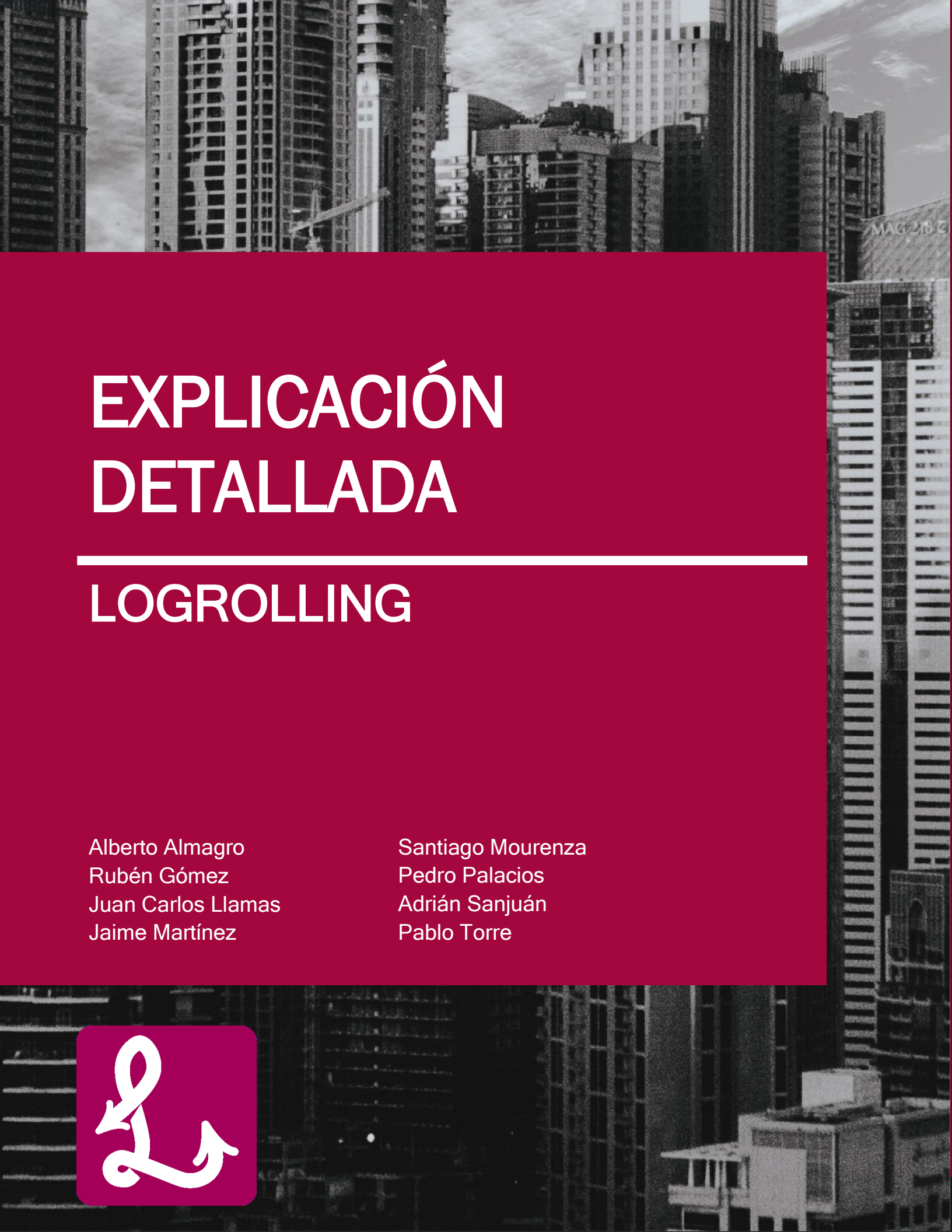

EXPLICACIÓN DETALLADA

LOGROLLING

Alberto Almagro
Rubén Gómez
Juan Carlos Llamas
Jaime Martínez
Santiago Mourenza
Pedro Palacios
Adrián Sanjuán
Pablo Torre

Índice

1. Visión general 3
2. Vista 4
3. Servidor 6
3.1 Base de datos 6
3.2 Modelo 7
3.3 Controladores 8
4. Patrones 9
4.1 MVC 9
4.2 Observer 9
4.3 Transfer 9
4.4 DAO 10
4.5 Singleton 10
4.6 Application Service 11
4.7 Abstract Factory 11
4.8 Adapter 11

1. Visión general

La arquitectura de Logrolling está estrechamente vinculada al modelo MVC. Logrolling está dividido en dos componentes independientes: El servidor, que está escrito en una aplicación *standalone* Java. Consiste en un servidor Tomcat ejecutando nuestra propia API REST, implementada mediante JAX-RS. El servidor se encarga de toda la lógica de negocio y la conexión con la base de datos. Por otra parte, está la vista, que hemos implementado como una aplicación Android. Debido a la gran independencia que tenemos entre el servidor y la vista (son componentes completamente separados), podríamos fácilmente en el futuro crear nuevas vistas, extendiendo Logrolling a aplicaciones web o iOS.

La comunicación entre la vista y el servidor se realiza mediante peticiones HTTP, y los datos están serializados en JSON (aunque la interfaz de serialización del servidor sería fácilmente extendible a otros formatos como XML).

Para garantizar la seguridad de nuestros usuarios, almacenamos las contraseñas en la base de datos *hasheadas* y *salteadas*, mediante sistemas de *hash* de contraseñas modernos y ampliamente utilizados en la comunidad. Además, los clientes no almacenan las credenciales de los usuarios en texto plano para autenticarse, sino que una vez autenticados, reciben un token (que también es almacenado *hasheado* y *salteado*) que, si es robado por un atacante, no permite obtener la contraseña original.

Precisamente por la seguridad de nuestros usuarios, nosotros no almacenamos información sensible respecto a los pagos, sino que utilizamos la plataforma de pagos Stripe certificada para almacenar este tipo de información.

2. Vista

La capa correspondiente a la vista dentro de la aplicación ha sido realizada con el programa Android Studio. La vista está basada fundamentalmente en *Activities*, donde cada *Activity* representa una pantalla de la aplicación y tiene una funcionalidad asociada. La navegación entre las distintas pantallas de la aplicación se realizará cambiando de *Activity* con el método `startActivity(new Intent(this, FavorsActivity.class))`. Cada *Activity* tendrá la funcionalidad para adquirir el contenido que mostrará. Para dicha representación, hemos utilizado las facilidades que nos otorga Android Studio, entre las que se encuentran el uso de *Material Design* y la posibilidad de añadir *Adapters* para personalizar los distintos listados de favores, mensajes y regalos.

En el *MainActivity*, se crea un *LoginService* que será el que se use mientras la aplicación esté abierta. En dicho *Activity*, se comprueba si hay un *Token* guardado en memoria persistente. En caso afirmativo, este es envidado al servidor a través de un *TransferObject* de la clase *TransferToken*. En el servidor, se comprueba si el *Token* es válido, y es entonces cuando el servidor devuelve a la vista un nombre de usuario. Seguidamente, se pasa a la pantalla de inicio que es la correspondiente a *FavorsActivity*. Si no se encuentra el *Token*, entonces el usuario es dirigido al *SignInActivity*, donde, tras introducir su nombre y su contraseña, se lanzará una petición al servidor que le devolverá a la vista el nombre de usuario correspondiente.

Las *Activities* que corresponden al resto de pantallas de la aplicación están compuestas por distintos componentes visuales que permiten la interacción del usuario, tanto para obtener información como para cambiar de *Activity*. De entre todos estos componentes, los más destacables son los *RecyclerViews*. Contamos con un total de seis *RecyclerViews* que se corresponden con: los favores que otros usuarios piden (en *FavorsActivity*), mis favores a realizar (en *MyFavorsActivity*), mis favores pedidos (en *MyFavorsActivity*), los chats abiertos (en *MessageActivity*), los mensajes de un chat (en *ChatPersonActivity*) y los regalos (en *GiftsActivity*). Los *RecyclerViews* van a funcionar como paneles que muestran al usuario la información de las distintas listas de objetos. Estos objetos *Favor*, *Gift*, *Persona* y *Mensaje* serán construidos por la vista tras peticiones al servidor que *TransferObjects* (*TransferChat*, *TransferFavor*, *TransferGift*, *TransferMessage* y *TransferUser*). Para obtener estos *TransferObjects*, desde la vista se hacen peticiones al servidor como `List<TransferFavor> lf=FavorQuerier.getAllFavors()`. Este ejemplo muestra que la comunicación entre la vista y el server es siempre a través de *Queriers*, en los que la vista solicita unos datos que le son devueltos en forma de *TransferObjects* y son tratados para poder ser mostrados.

La mayoría de los *Activities*, aquellos que podríamos considerar como pantallas de navegación principales y representan los distintos módulos de la aplicación, cuentan además con un panel inferior formado por varios componentes propios de Android Studio, como *ImageButton* o *TextView,* que permiten navegar por la aplicación yendo de un Activity a otro. Cabe destacar que en *FavorsActivity* el usuario tiene disponible una opción de búsqueda por filtro. El usuario podrá buscar un favor por nombre, por distancia o por recompensa mínima de grollies, mostrándose a continuación dicha información ordenada en el *RecyclerView* del *FavorsActivity*. Internamente, la aplicación solicita los datos al servidor mediante *Queriers* y rellena con la información recibida un array, que es pasado mediante un *Adapter* al *RecyclerView*, que lo muestra al usuario.

Una de las ventajas del uso del patrón Modelo-Vista-Controlador en nuestra aplicación es que conseguimos un mayor nivel de seguridad. La vista obtiene todos sus datos a través de *Queriers* al servidor y la comunicación entre vista y servidor es siempre mediante *TransferObjects*, que permiten el intercambio bidireccional de información entre ambos. Dichas peticiones son siempre comprobadas por el servidor lo que aporta un mayor grado de seguridad.

En cuanto al *ShopActivity*, que es el encargado de los pagos, usaremos el sistema de pagos de Stripe, por lo que no tendremos que manejar en ningún momento datos bancarios, maximizando la seguridad de la aplicación, evitando tener que almacenar información sensible como tarjetas de crédito. Además, el servidor comprueba que todos los pagos realizados por Stripe son correctos, evitando de esta manera fraudes.

3. Servidor

El servidor está compuesto por una base de datos, el modelo en sí y los controladores.

3.1 Base de datos

La base de datos de nuestra aplicación almacenará los datos de nuestros clientes. Tendremos unas clases llamadas *Migration* en las que crearemos todas las tablas necesarias en nuestra base de datos y un *Migration Manager* que se encargará de manejar todas las *Migrations*. Estas clases son muy importantes en nuestra aplicación, ya que definen en código la estructura de nuestra base de datos. Por este mismo motivo, no tenemos que tener archivos de configuración aparte, y nuestra aplicación será más manejable y ampliable a nuevos entornos, pues solo tendremos que ejecutar el jar.

Estos datos deben pasar de nuestra base de datos a la vista de algún modo. Ahí es donde entran en juego los controladores y los denominados *Managers* de nuestros objetos. Los *Managers* se encargan de implementar las operaciones CRUD y en ellos podremos ver la implementación del patrón DAO. En nuestra aplicación, tenemos 5 managers que están relacionados con cada uno de los objetos de nuestra aplicación y cada manager tendrá su correspondiente controller.

Estos managers tendrán como objetivo obedecer las órdenes del modelo y devolver la información que le pida de la base de datos. Más tarde los controladores crearán objetos *transfer* con esos datos para pasarlos a la vista. En los managers, tenemos implementadas diversas funciones para añadir un objeto a la tabla correspondiente, borrar un objeto de la misma o devolver todos los objetos de una tabla, entre un largo etcétera. Estos managers abrirán la base de datos, cogerán los datos necesarios y la volverán a cerrar. Si ocurre cualquier error en el acceso a los datos o al intentar devolverlos al modelo, se lanzarán excepciones que serán tratadas en el modelo.

La *DatabaseConnection* nos permitirá acceder a la base de datos desde los managers sin importar qué base de datos estemos utilizando (MySQL, PostgreSQL, Oracle, …), lo que disminuye la interdependencia entre clases.

En resumen, la base de datos almacenará todos los objetos que existen en nuestra aplicación. Los controladores harán de puente entre la base de datos y la vista y accederán a la base de datos a través de los managers de cada objeto. La *DatabaseConnection* nos dará esa abstracción necesaria para no depender de la base de datos que estemos utilizando y nos permitirá acceder a los datos desde los manager de los objetos.

3.2 Modelo

El modelo es una parte esencial de la aplicación al usar el patrón MVC. En el servidor de Logrolling hay un paquete dedicado exclusivamente a ello, es decir, a la representación de los datos. Hemos decidido crear diferentes clases para los diferentes aspectos de la aplicación.

Así, encontramos en primer lugar la clase *Favor*, ocupada de gestionar todo lo referente a los favores. Identificamos los favores con un id y les asociamos su título, descripción, creador, recompensa, fecha y lugar de entrega y el usuario encargado de realizar el favor. En resumen, almacenamos toda la información necesaria que la vista va a mostrar en la aplicación.

De igual manera hemos creado las clases *User*, G*ift* y *Message*. Contienen todos los datos que identifican a un usuario, a un regalo o a un mensaje, respectivamente. No es de extrañar entonces que encontremos atributos como el nombre de usuario o la contraseña para *User* o un texto, un usuario que envía y un usuario que recibe en el caso de *Message*.

En esta línea, destaca la clase *Token*, que, como ya se ha comentado en secciones anteriores, es la encargada de verificar la identidad del usuario, de manera que todas las acciones se realicen de manera segura y libre de riesgos. No entraremos de manera detallada a explicar su funcionamiento y su implementación detallada, pero sí que consideramos importante mencionarla. Sí explicaremos que hay otra clase, *Authenticator*, que es la que *hashea* y comprueba los *Token* usando librerías como `java.security` o `javax.crypto`.

Tenemos también 3 clases algo diferentes, que cumplen diversos roles. La primera *Coordinates*, la manera que hemos elegido de guardar los lugares que el usuario establezca, por ejemplo, para el lugar de entrega de un favor. Consistirá, principalmente, en una latitud y una longitud y permitirá, por ejemplo, ordenar los favores de acuerdo a la cercanía con el usuario.

Las dos clases restantes son bastante explícitas, pues se trata de *Filter* y de *ErrorMessage*. Ambas contienen funcionalidades menos imprescindibles para la aplicación que el resto de las mencionadas aquí.

3.3 Controladores

Los controladores se encargan de la transferencia de datos entre la vista y el modelo. Hay un total de 6 controladores, cada uno encargado de la comunicación de uno de los aspectos de la aplicación. 5 de ellos extienden al 6º, el *AuthenticableController*, que se encarga, entre otras cosas, de comprobar si el usuario tiene los permisos necesarios para realizar la acción que desea (por ejemplo, no se le permite borrar un favor salvo que sea el propio usuario que lo ha creado el que intenta realizar la acción).

Para gestionar esta transferencia, los controladores implementan el estilo de arquitectura software conocido como "Transferencia de Estado Representacional" (con siglas REST en inglés). Esto permite obtener los datos en formato JSON de manera sencilla y eficiente mediante la librería *JAX-RS: Java API for RESTful Web Services* de Java.

La idea es simple: Los *queries* presentes en la vista llaman al controlador del que les interese obtener información y este se la devuelve. Esta llamada se resuelve mediante los denominados métodos HTTP, cuyo funcionamiento es, en cierta medida, análoga a los métodos CRUD en bases de datos. Los 4 métodos principales son POST, GET, PUT y DELETE. De esta manera, si el usuario quiere obtener una lista de favores, el *query* asociado a los favores irá a la dirección "`/favors`" y hará una petición GET. El controlador es el que ha de procesar esta llamada y asociar un método para que cada vez que se realice esta misma petición se ejecute ese método.

Además, el controlador también gestiona cómo se produce el intercambio de datos con la vista, intentando que se produzca de la manera más segura posible. De esta manera, se implementa el patrón *transfer*, que permite enviar datos desde la base de datos de forma eficiente y segura. Hay clases *transfer* para cada uno de los datos que se quieren enviar, de manera que los controladores reciben y mandan la información usándolas. Es aquí donde se hace la conversión de los datos suministrados por la base de datos a objetos transferencia y viceversa.

Para resumir, el trabajo de los controladores es conectar el resto de la aplicación, lo que resulta imprescindible para un correcto funcionamiento de la aplicación, como se podía esperar de usar la arquitectura MVC.

4. Patrones

En Logrolling hemos usado varios patrones de diseño de software estándar, pero los hemos modificado ligeramente para así adecuarlos a las necesidades concretas de nuestro proyecto.

4.1 MVC

Al constar nuestro proyecto de dos estructuras claramente diferenciadas, el servidor y la aplicación Android, las ventajas de utilizar el patrón MVC son claras. El servidor contiene el modelo y el controlador, así como la capa de persistencia, que almacena todos los datos. De esta manera, podemos realizar actualizaciones de la aplicación Android sin tener que modificar el servidor. De la misma manera, en el futuro podremos lanzar una aplicación iOS o una aplicación web sin tener que modificar absolutamente nada. Como protocolo de comunicación entre el controlador y la vista, nos hemos decidido por utilizar HTTP, implementando REST.

4.2 Observer

El patrón *Observer* es ampliamente utilizado en la implementación de varias características de Android que utilizamos. Todos los *Listeners*, que nos permiten reaccionar a distintos eventos que ocurren en la aplicación (pulsación de botones, escritura en campos de texto, …), son registrados en una lista de observadores por Android. Cada vez que el usuario realiza alguna de esas acciones, el subsistema de Android llama a los observadores.

4.3 Transfer

En Logrolling usamos el patrón *Transfer* en todas las transferencias de datos entre el servidor y la vista. De esta manera simplificamos mucho el código de serialización y deserialización, ya que los *Transfer* son POJO (“Plain Old Java Object”), y además nos permiten separar todo el código de la vista de la capa de modelo. Aunque modifiquemos el código interno de la clase *User*, si no varía la representación de los datos necesarios para la vista, no será necesario modificar la aplicación. Esto nos permite poder actualizar rápidamente el servidor, sin tener que preocuparnos porque los usuarios deban actualizar la aplicación, que generalmente puede llevar desde días a meses.

4.4 DAO

En nuestro proyecto, necesitamos tener DAOs que realizarán todas las operaciones CRUD, pero nos conviene hacer dos importantes modificaciones:

- Los DAO no se comunican directamente con la base de datos. Como queremos que nuestra aplicación sea compatible con diversas bases de datos (MySQL, PostgreSQL, Oracle…), y no queremos que tener que volver a reescribir toda la lógica CRUD de alto nivel (los DAOs) para cada base de datos, hemos decidido añadir otra abstracción (*DatabaseConnection*), que se encarga de ofrecer comandos de base de datos a bajo nivel, que luego usan los DAO. Cada implementación de base de datos solo crear su propio *DatabaseConnection,* minimizando la repetición de código.

- Los DAO son objetos estáticos. Debido a la naturaleza de Logrolling, es normal que cada instancia del servidor reciba miles de peticiones por segundo. Por tanto, es vital reducir todo el *overhead* posible. Es bien conocido que la creación de objetos en el *heap* es pesada, por tanto, nosotros trabajaremos con objetos estáticos, que tienen toda la funcionalidad en métodos estáticos, que pueden llamarse desde distintos hilos. Hemos decidido la solución estática antes que el *Singleton* para poder utilizar los DAO en entornos multihilo y distribuidos, con lo que podremos escalar horizontalmente Logrolling de manera sencilla. Otro motivo para hacer los DAO objetos estáticos es simplificar el código. Si no fuesen objetos estáticos, o bien el código tendría que depender de una clase de los objetos estáticos, perdiendo todas las posibles ventajas que podría traer el hecho de tener objetos dinámicos, o tendríamos que inyectar a todas las clases que usasen los DAO dichos DAO, entorpeciendo la interfaz y generando una gran complicación.

4.5 Singleton

En Logrolling, utilizamos el patrón *Singleton* en el caso de que queramos una única instancia y queramos mantener los beneficios del polimorfismo asociado a tener objetos dinámicos y no estáticos. Esta ventaja es clara en los distintos *Queriers* de la vista, ya que tal vez en un futuro añadamos más proveedores distintos de la información que solicita (websockets para los mensajes, por ejemplo). Sin embargo, en otros casos, como en ciertas partes del servidor, más beneficioso que el Singleton son las clases con métodos estáticos por motivos de concurrencia, distribución y rendimiento, así como la falta de necesidad de polimorfismo y vinculación dinámica en esos casos.

4.6 Application Service

Aunque no explícitamente, en Logrolling utilizamos el patrón *Application Service*. No definimos una clase *Service* explícitamente, pero el servidor, a través de la conexión HTTP, actúa como un punto centralizado para toda la lógica de negocio. Por tanto, todo el servidor sería un gran *Application Service*. Esta centralización de la lógica de negocio simplifica la implementación de creación de distintas vistas o clientes.

4.7 Abstract Factory

Al tener distintas bases de datos posibles que necesitamos, y tener una factoría para la conexión a una base de datos específica, necesitamos una manera de obtener una conexión a la base de datos elegida para el despliegue de la aplicación. Es muy importante que esta base de datos sea la misma durante toda la aplicación, para asegurar criterios de consistencia e integridad en las transacciones, característica muy importante al trabajar con datos financieros. Un problema que no soluciona la factoría abstracta, es que es necesario pasar dicha factoría como parámetro a los distintos componentes que la usan. Es nuestro caso, tampoco es viable, por los motivos ya explicados en los DAO. Por tanto, para solucionar estos problemas específicos de nuestro proyecto, hemos decidido implementar algo similar a una factoría abstracta en una clase estática. La clase *DatabaseFactory* contiene una referencia estática a una factoría de conexiones a la base de datos. También contiene un método para cambiar dicha factoría. Así, podemos por un lado elegir la factoría que elegimos para nuestra base de datos al inicializar la aplicación, y por otro obtener en la aplicación siempre un objeto de dicha factoría, asegurando la coherencia.

4.8 Adapter

Utilizamos el patrón *Adapter* varias veces en la capa de la vista de Android. Para mostrar eficientemente una lista de muchos elementos a través de la que el usuario se pueda desplazar, utilizamos el componente de Android *RecyclerView*. Dicho *RecyclerView* necesita un *Adapter* para cada lista de objetos definidos por el usuario, que defina cómo se van a renderizar dichos elementos. Por tanto, para cada *RecyclerView* que utilizamos, tenemos su respectivo *Adapter*.