MEMORIA DE ARQUITECTURA SERVIDOR

LOGROLLING

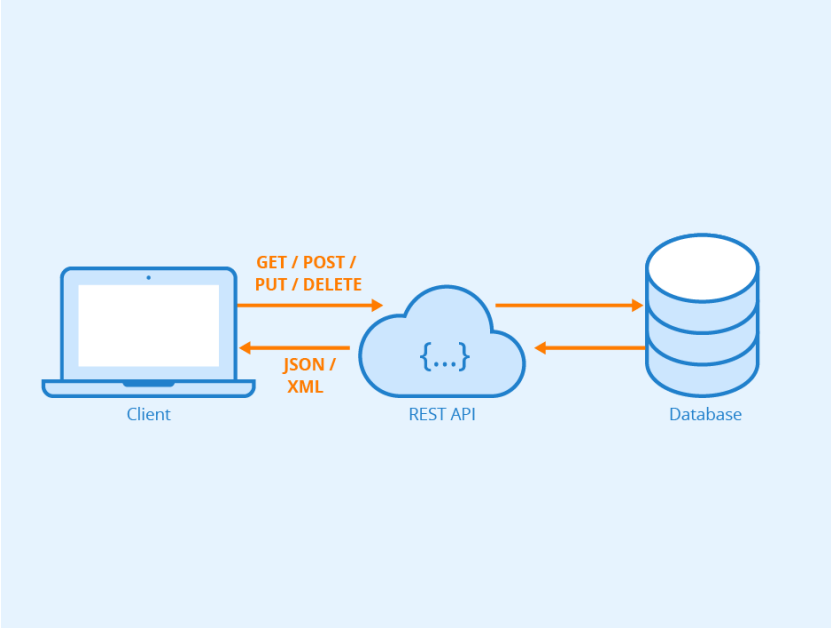
Alberto Almagro
Rubén Gómez
Juan Carlos Llamas
Jaime Martínez
Santiago Mourenza
Pedro Palacios
Adrián Sanjuán
Pablo Torre

Índice

1. Introducción

Logrolling está dividido en dos subsistemas: el servidor y el cliente.

El servidor consiste en una una base de datos que actúa como capa de persistencia y una aplicación java que actúa como servidor REST.



REST es un estilo de arquitectura software utilizado para sistemas Web. El cliente envía un mensaje HTTP a un servidor. Esta conexión entre el cliente y el servidor **no tiene estado,** es decir, entre distintas peticiones del cliente, el servidor no *recuerda* nada. El servidor REST maneja la petición, ejecuta la lógica de negocio, que se comunica con una base de datos **no accesible** al cliente, por motivos de seguridad, y devuelve una respuesta HTTP, con los códigos de estado de respuesta estándar (200, 404, 500, etc), y, si es necesario en la petición, un cuerpo JSON con la respuesta requerida por el cliente.

A grandes rasgos, viendo un esquema simplificado, una petición usual entre el cliente y el servidor sería la siguiente:

1. El usuario ejecuta alguna acción en el cliente (por ejemplo, pulsa un botón para pedir un nuevo favor).
2. El cliente manda una petición HTTP al servidor pidiéndole que cree un nuevo favor.
3. El servidor recibe la petición HTTP y se comunica con la base de datos.
4. El servidor devuelve una respuesta a la petición HTTP, o bien confirmando la creación del favor (código de respuesta 200) o devolviendo un error (por ejemplo, 401, si el usuario no estaba autenticado correctamente)

5. El cliente procesa la respuesta y reacciona asíncronamente a esta

Por tanto, el servidor define una API, un formato al que debe adherirse el cliente para hacer solicitudes. Para hacer más sencillo el manejo de dicha API, se divide por los distintos servicios que ofrece Logrolling: (anuncios, autenticación, mensajes, favores, regalos, imágenes, pagos y usuarios).

Para no obligar al cliente a leerse el código del servidor para poder implementar un cliente, hemos creado una documentación web de la API de Logrolling disponible en https://palmenros.github.io/logrolling/. En dicha documentación, separada por servicios, se indica que petición HTTP puede realizarse a qué URL y qué parámetros deben ser aportados en dicha petición.

En nuestra implementación concreta, la base de datos es *MySQL*, el servidor es una aplicación Java, ejecutable mediante un *.jar*. Y el cliente es una aplicación *Android*, distribuida mediante un *.apk*. Sin embargo, debido al diseño que hemos realizado, el servidor está completamente desacoplado del cliente, por lo que fácilmente podemos desarrollar nuevos clientes para distintas plataformas (iOS, Windows, Web…) y podemos actualizar el comportamiento del servidor, sin tener que actualizar los clientes (respetando el contrato definido por la API REST).

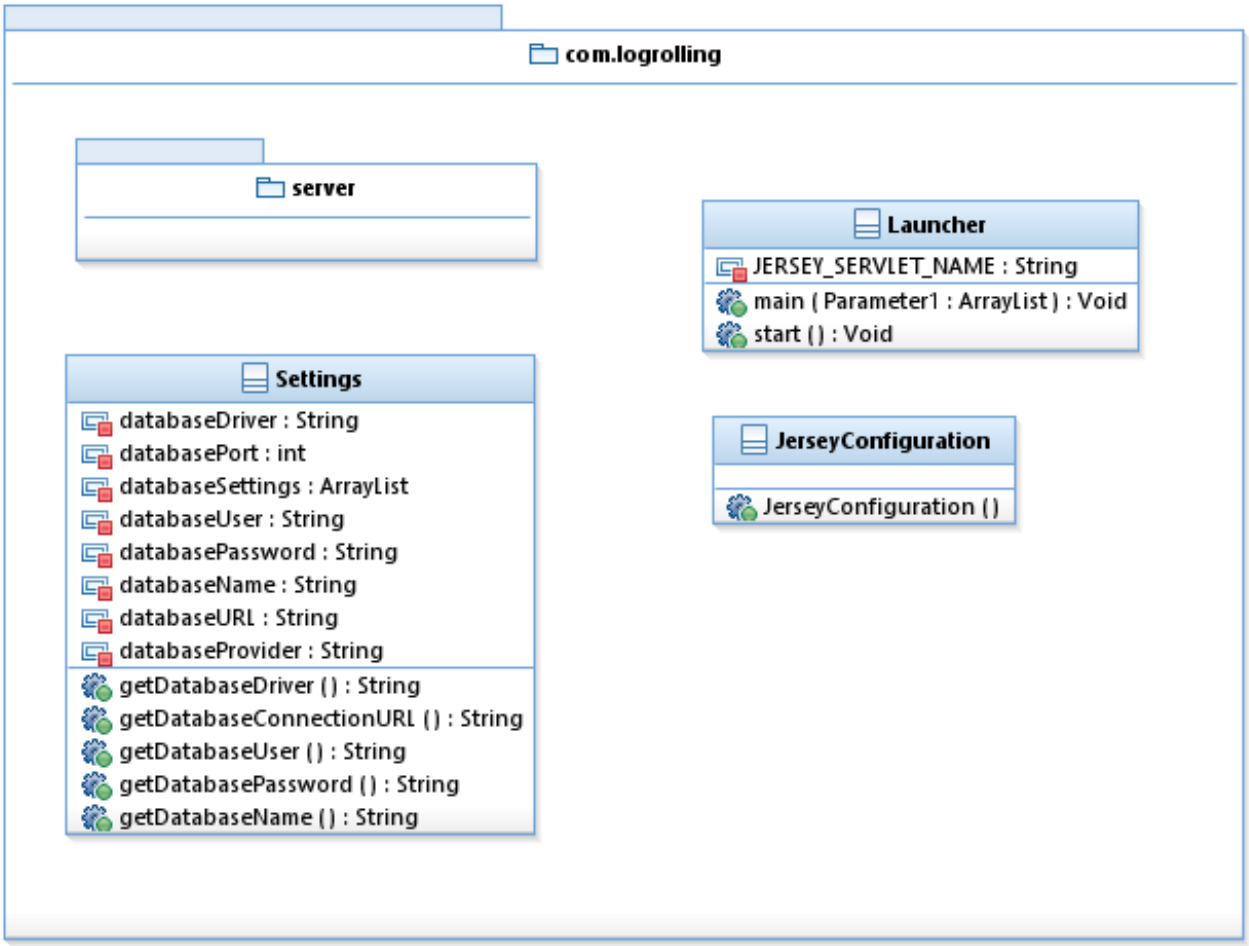
Comenzamos viendo la arquitectura del servidor. Podemos distinguir dos grandes componentes, el servidor HTTP y la base de datos.

La base de datos es la capa de persistencia de Logrolling. El cliente no tiene acceso directo a ella, solo a través del servidor HTTP. En nuestra implementación, la base de datos utilizada es *MySQL*, pero veremos más adelante como el código no depende de esta implementación, y pueden utilizarse otras implementaciones arbitrariamente como *Firebird, Oracle SQL, PostgreSQL, SQLite...*

El servidor HTTP es una aplicación Java desarrollada por nosotros que maneja la API REST, que se ejecuta sobre *Tomcat*, que es una implementación Open Source de Java Servlets, el estándar empresarial usado ahora mismo. Este componente es el que se expone al cliente a través de las distintas URL descritas en la API REST. Este componente se comunica también con la base de datos a través de distintas bases.

No implementamos desde 0 un cliente HTTP, sino que utilizamos el servidor *Tomcat*, específicamente, la librería para creación de APIs REST llamada JAX-RS. Mediante esta librería, la implementación de un *end-point* (URL a la que el cliente puede hacer una petición HTTP) se convierte en algo tan sencillo como crear una clase y anotar los distintos métodos con la URL a la que corresponde y el método HTTP al que corresponde.

A partir de este momento, y con objeto de facilitar la comprensión, usaremos el color azul cada vez que se mencione a un paquete, el color verde para las interfaces y el color granate para las clases:



Como se ve, el paquete com.logrolling es el paquete que engloba todo el código.

La clase JerseyConfiguration contiene configuración específica de JAX-RS, con información necesaria para que JAX-RS funcione correctamente, como el paquete en el que buscar las clases que recibirán los métodos de la URL, etc.

La clase Launcher es la clase que contiene el punto de entrada de la aplicación, el método *main*. Se encarga de inicializar Tomcat y JAX-RS y de llamar a nuestra clase Main dentro del paquete server.

La clase Settings es un punto centralizado donde se almacenan distintos valores de configuración necesarios por la aplicación. Por ejemplo, ahí se almacenan las credenciales necesarias para acceder a la base de datos, puerto y proveedor de la base de datos, codificación unicode a utilizar...

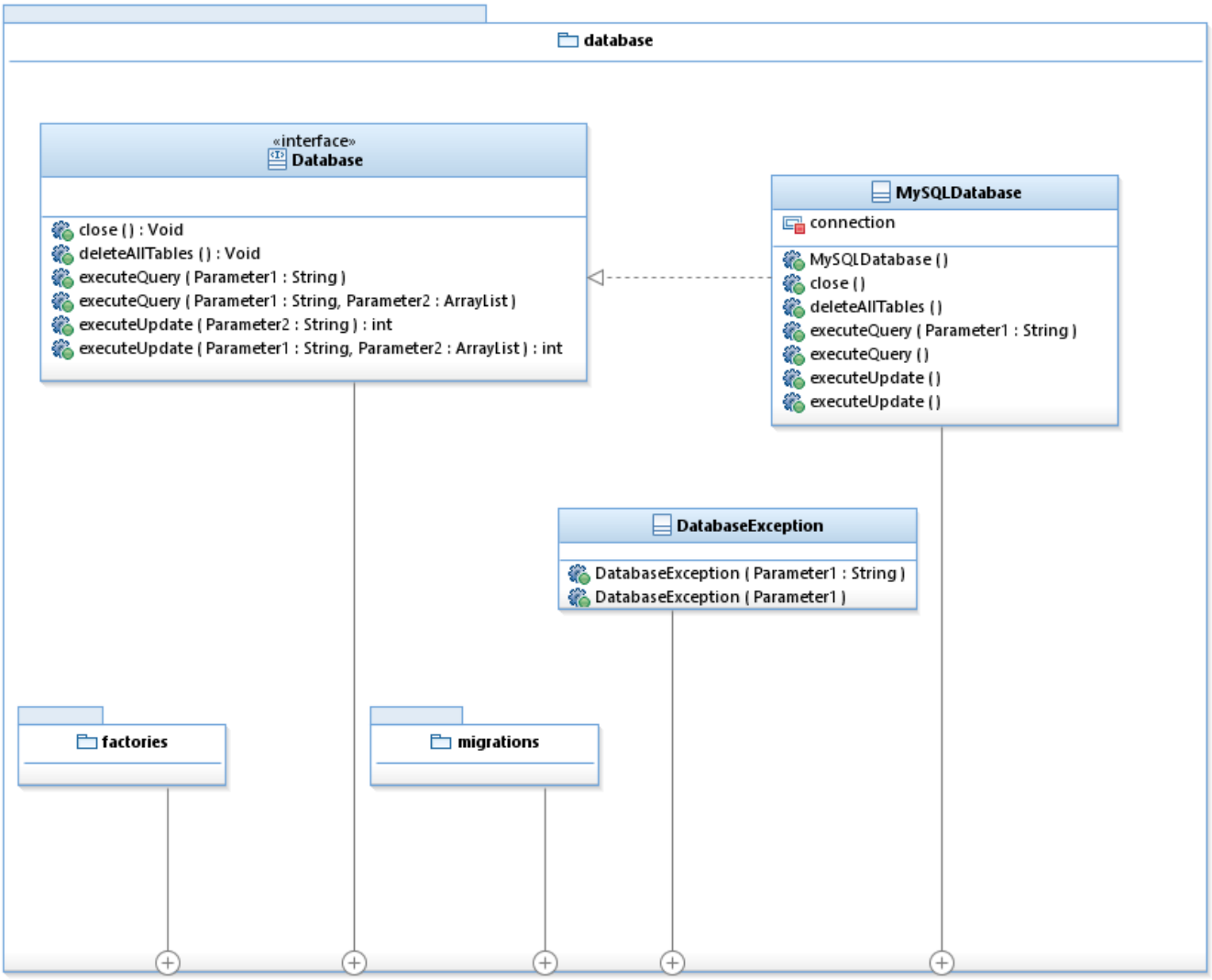
Por tanto, la función de las clases fuera del paquete *server* es puramente auxiliar. Son clases necesarias para el correcto funcionamiento de la aplicación, pero no tienen lógica de negocio.

Toda la lógica real de la aplicación está concentrada en el paquete *server.* Dentro de él se encuentra la clase **Main**, cuya función es inicializar todo lo relacionado con la base de datos; y distintos paquetes que quedan representados en el siguiente diagrama:

2. Paquete database

El paquete database contiene todo el código necesario para el acceso **a bajo nivel** a la base de datos. En este paquete no se encuentran los DAOs, sino las clases que permiten el acceso a la base de datos *a bajo nivel.* Además, este paquete es el encargado de proveer de una abstracción que puedan utilizar los DAOs que sea independiente de la base de datos utilizada realmente, para que los DAOs estén desacoplados de la ímplementación de la base de datos utilizada.

Esta abstracción que provee permite la ejecución de *SQL Statements*, pero sin tener en cuenta la implementación de la base de datos subyacente. Para eso, este paquete tiene la siguiente estructura:



La interfaz Database define la abstracción que van a utilizar los DAOs para comunicarse con la base de datos. Como hemos dicho antes, esta abstracción es completamente

independiente de la implementación de la base de datos. Esta abstracción tiene métodos para ejecutar comandos SQL (lenguaje estándar en bases de datos relacionales, que son necesarias en Logrolling por seguridad, dado que trabajamos con monedas virtuales).

Por motivos de seguridad, estos comandos están divididos en comandos que permiten devolver uno o varios valores de la base de datos (*executeQuery*) y comandos que permiten modificar los datos dentro de la base de datos (*executeUpdate*). Existen versiones que aceptan parámetros, para evitar *SQlInjection*, un error de seguridad grave.

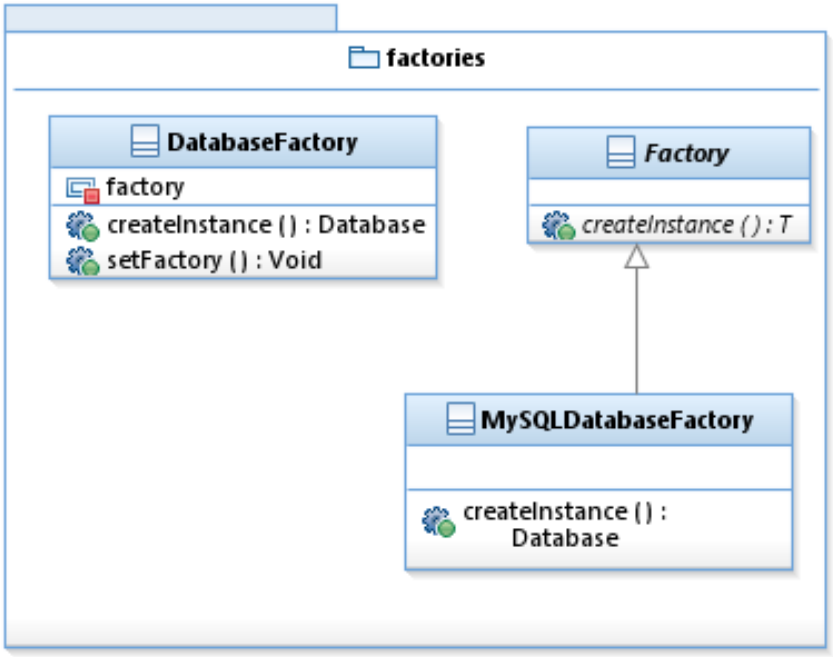
La clase DatabaseException es la única excepción que pueden lanzar los métodos de Database, y es la encargada de almacenar los datos del error. Normalmente, llevará encapsulada la verdadera excepción lanzada por cada implementación específica de la base de datos.

La clase MySQLDatabase consiste en una implementación específica para la base de datos MySQL. Implementa la interfaz Database e implementa todos los métodos de esta.

Procedemos a explicar los paquetes factories y migrations:

2.1 Paquete factories

Todo el trabajo que hemos hecho para abstraer la base de datos sería inútil si en los DAOs utilizasemos directamente la clase MySQLDatabase para acceder a la base de datos, ya que para cambiar la base de datos deberíamos modificar todos los DAOs. Para evitar esto, introducimos el paquete *factories* con la siguiente estructura:



La interfaz Factory<T>, donde T es un tipo arbitrario, tiene únicamente un método *createInstance()*, que devuelve una nueva instancia de T. Por tanto, una *Factory* es una clase que se encarga de crear objetos de un tipo T.

La clase MySQLDatabaseFactory es una implementación de Factory<Database> cuya función es devolver un objeto Database de la clase MySQLDatabase.

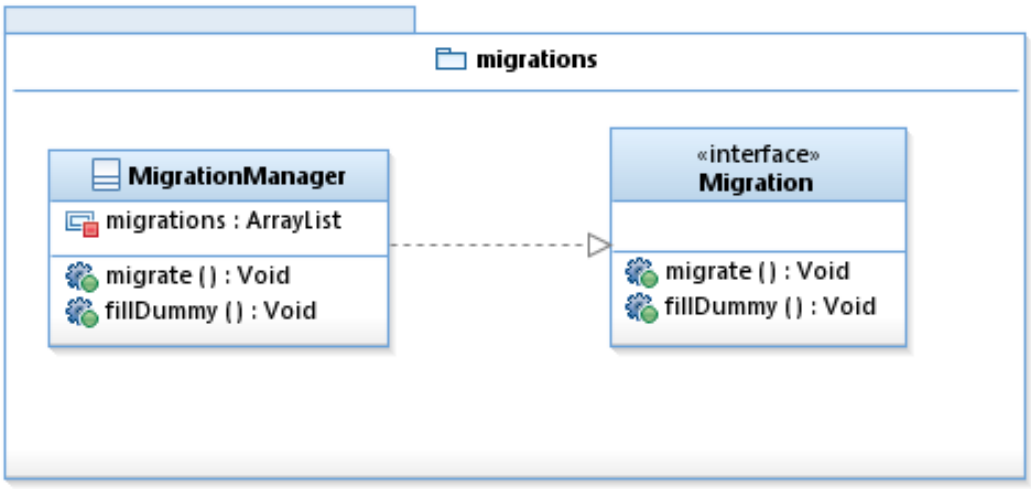
La clase DatabaseFactory es la que utilizan los DAOs. Tiene un miembro estático de tipo Factory<Database>, un método estático *createInstance*, que devuelve un nuevo objeto creado con dicha factoría, y otro un método público *setFactory* que permite cambiar la factoría utilizada para *createInstance*. Es trabajo de la clase Main inicializar DatabaseFactory con la factoría de la implementación de base de datos a utilizar. Es el único punto a modificar para cambiar la implementación de base de datos subyacente.

2.2 Paquete migrations

Además de hacer consultas para modificar datos y recibir datos a la base de datos, es necesario en algún momento definir la estructura que van a tener las tablas en las que se van a realizar dichas consultas.

Otra necesidad durante en el desarrollo de la aplicación, es que muchas veces surge la necesidad de tener valores por defecto para realizar pruebas en la aplicación, y no perder tiempo manipulando la aplicación cada vez que se borran los datos para llegar a ese estado.

Para estos propósitos existe el paquete migrations, que tiene la siguiente estructura:

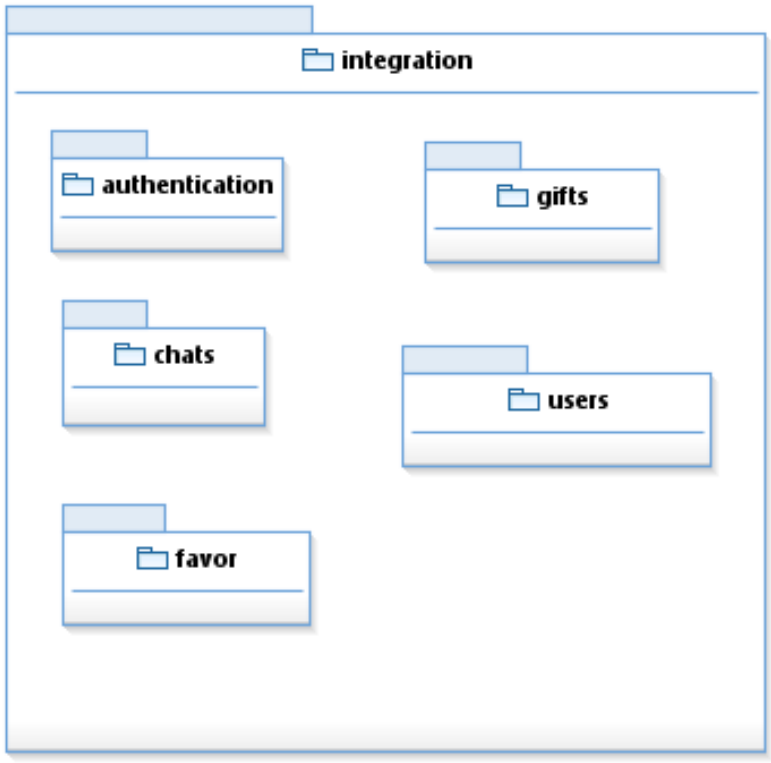


La interfaz Migration define un recurso (por ejemplo, favores, regalos, usuarios…) que es necesario almacenar en la base de datos. Por cada uno de estos recursos, se creará una clase que hereda de dicha interfaz. Esta interfaz define un método *migrate*, cuya función es crear las tablas en la base de datos necesarias para el correcto funcionamiento de los DAOs; y el método *fillDummy*, que se llamará únicamente si se ha pasado por la consola un argumento de desarrollo, y se encargará de popular las tablas con algunos datos de ejemplo definidos.

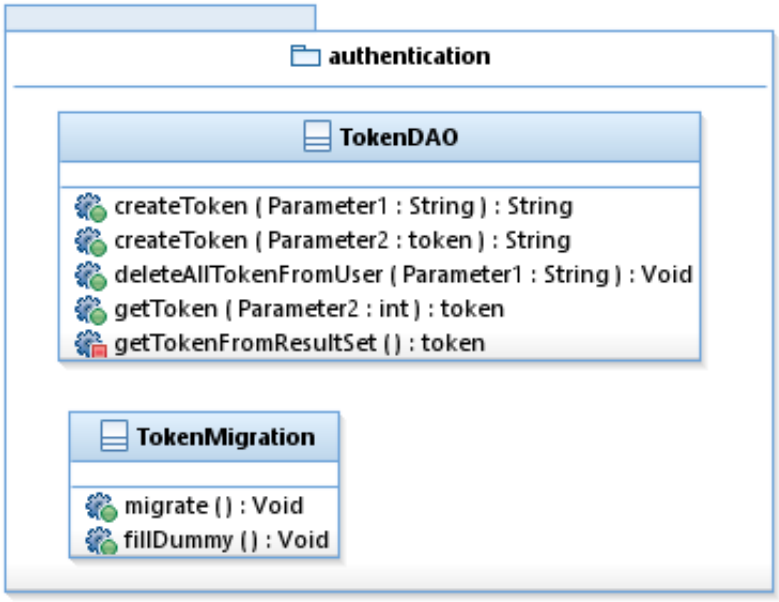
La clase MigrationManager agrupa las distintas *Migrations*. La clase Main llama a MigrationManager, y esta clase es la encargada de llamar a los métodos de cada Migration. Para esto, tiene un array de las distintas *Migrations* creadas, y llama a *migrate* para cada una de ellas y *fillDummy* para cada una de ellas, cuando se lo indique Main. Por tanto, al añadir una nueva migración, no habrá que modificar más código que añadir la nueva migración a MigrationManager.

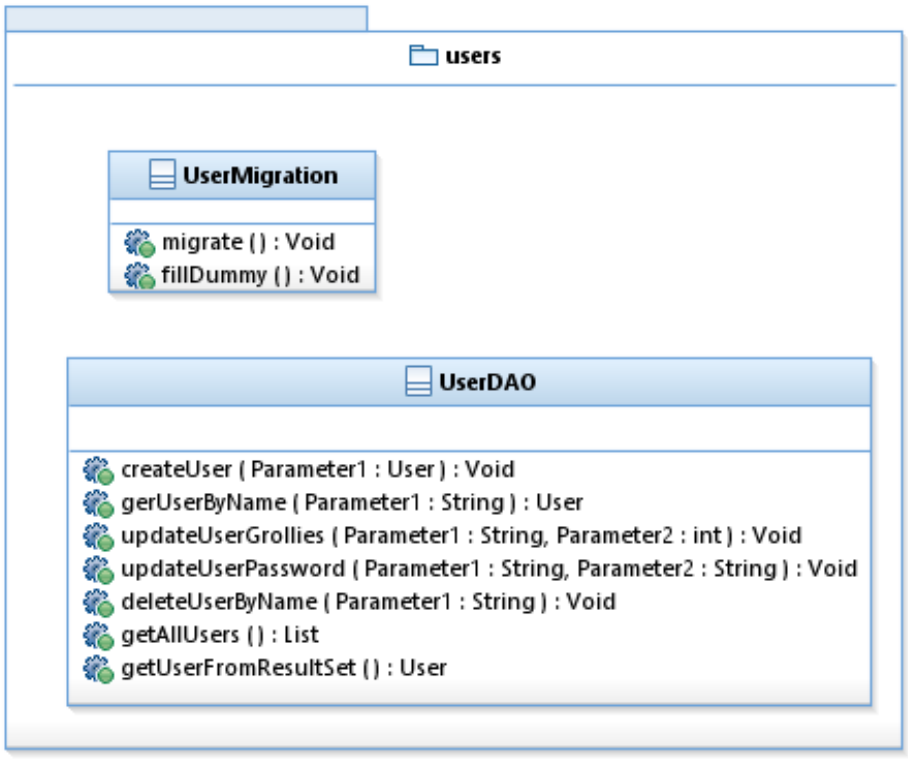
3. Paquete integration

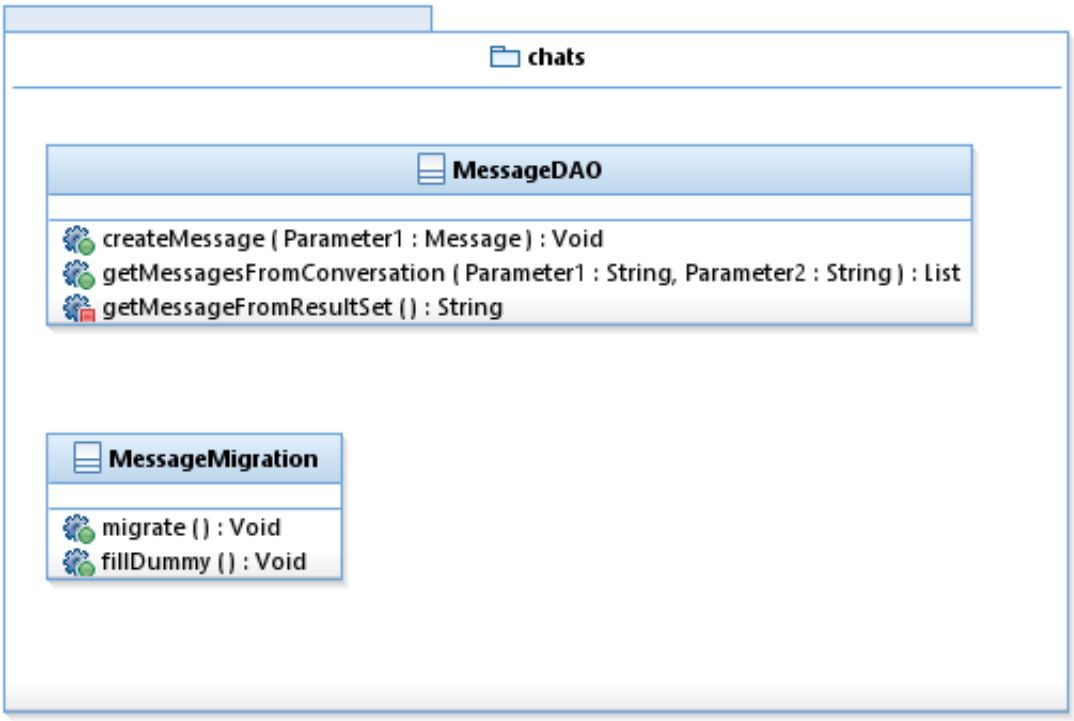
Se encarga de la interacción a alto nivel con la base de datos. Utilizan únicamente las abstracciones a bajo nivel definidas en el paquete database. La estructura es:

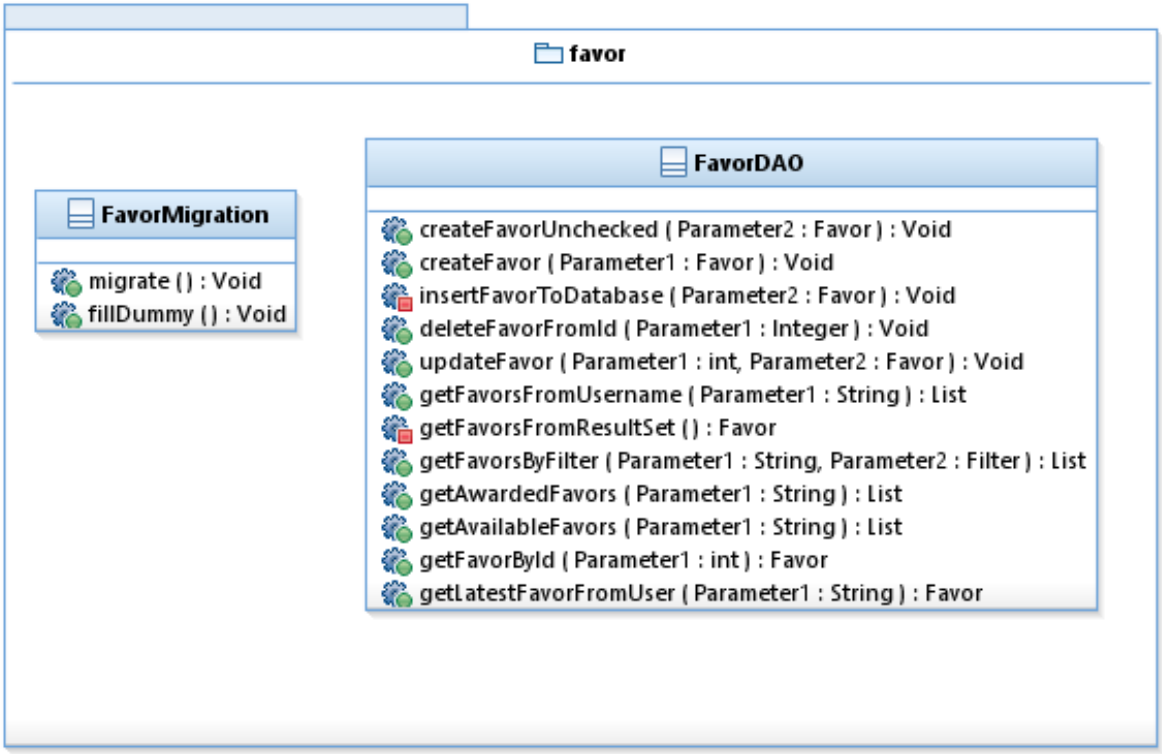


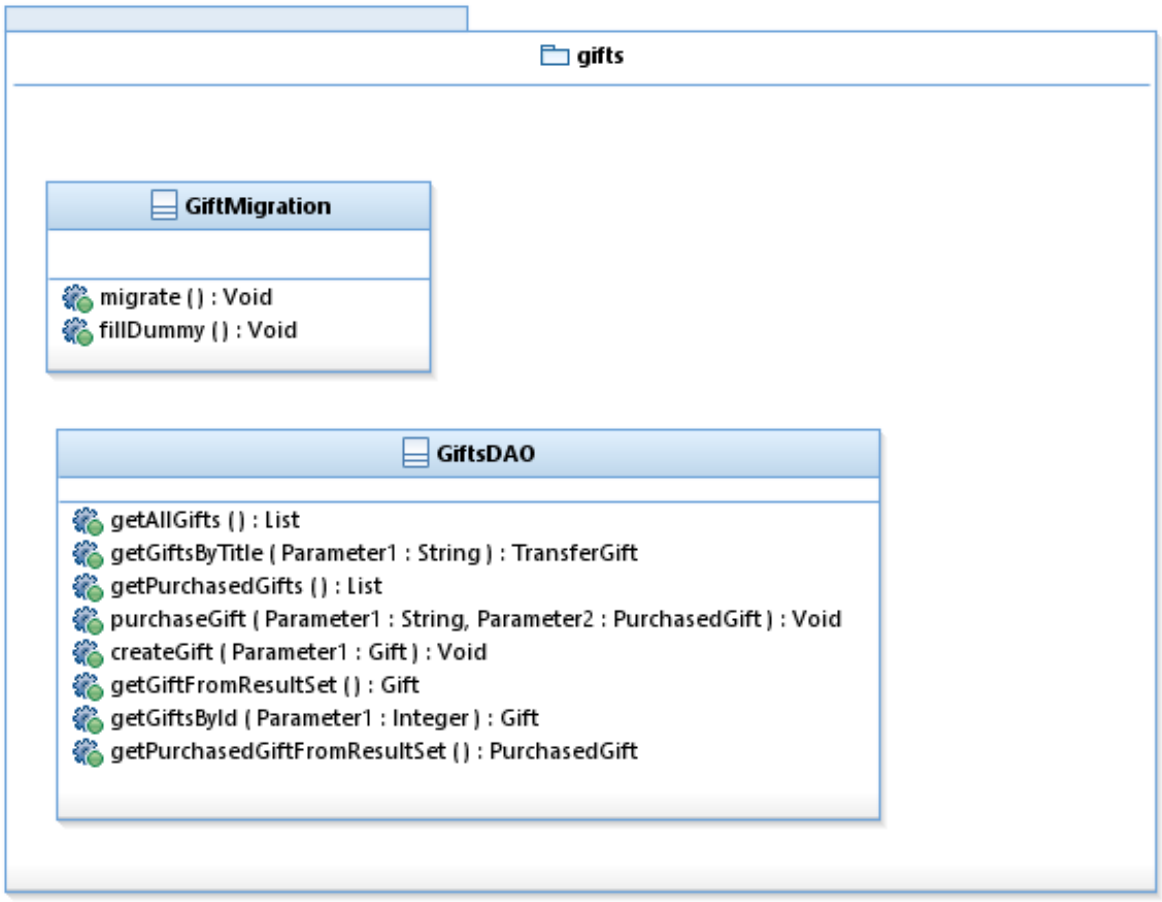
Por tanto, en este paquete, por cada uno de los *recursos* que almacenamos en la base de datos, dispone de un *DAO* y una Migration:




users
UserMigration
migrate ( ) : Void
fillDummy ( ) : Void
UserDAO
createUser (Parameter1 : User) : Void
gerUserByName (Parameter1 : String) : User
updateUserGrollies (Parameter1 : String, Parameter2 : int) : Void
updateUserPassword (Parameter1 : String, Parameter2 : String) : Void
deleteUserByName (Parameter1 : String) : Void
getAllUsers ( ) : List
getUserFromResultSet ( ) : User


chats
MessageDAO
createMessage (Parameter1 : Message) : Void
getMessagesFromConversation (Parameter1 : String, Parameter2 : String) : List
getMessageFromResultSet ( ) : String
MessageMigration
migrate ( ) : Void
fillDummy ( ) : Void


favor
FavorMigration
migrate ( ) : Void
fillDummy ( ) : Void
FavorDAO
createFavorUnchecked (Parameter2 : Favor) : Void
createFavor (Parameter1 : Favor) : Void
insertFavorToDatabase (Parameter2 : Favor) : Void
deleteFavorFromId (Parameter1 : Integer) : Void
updateFavor (Parameter1 : int, Parameter2 : Favor) : Void
getFavorsFromUsername (Parameter1 : String) : List
getFavorsFromResultSet ( ) : Favor
getFavorsByFilter (Parameter1 : String, Parameter2 : Filter) : List
getAwardedFavors (Parameter1 : String) : List
getAvailableFavors (Parameter1 : String) : List
getFavorById (Parameter1 : int) : Favor
getLatestFavorFromUser (Parameter1 : String) : Favor

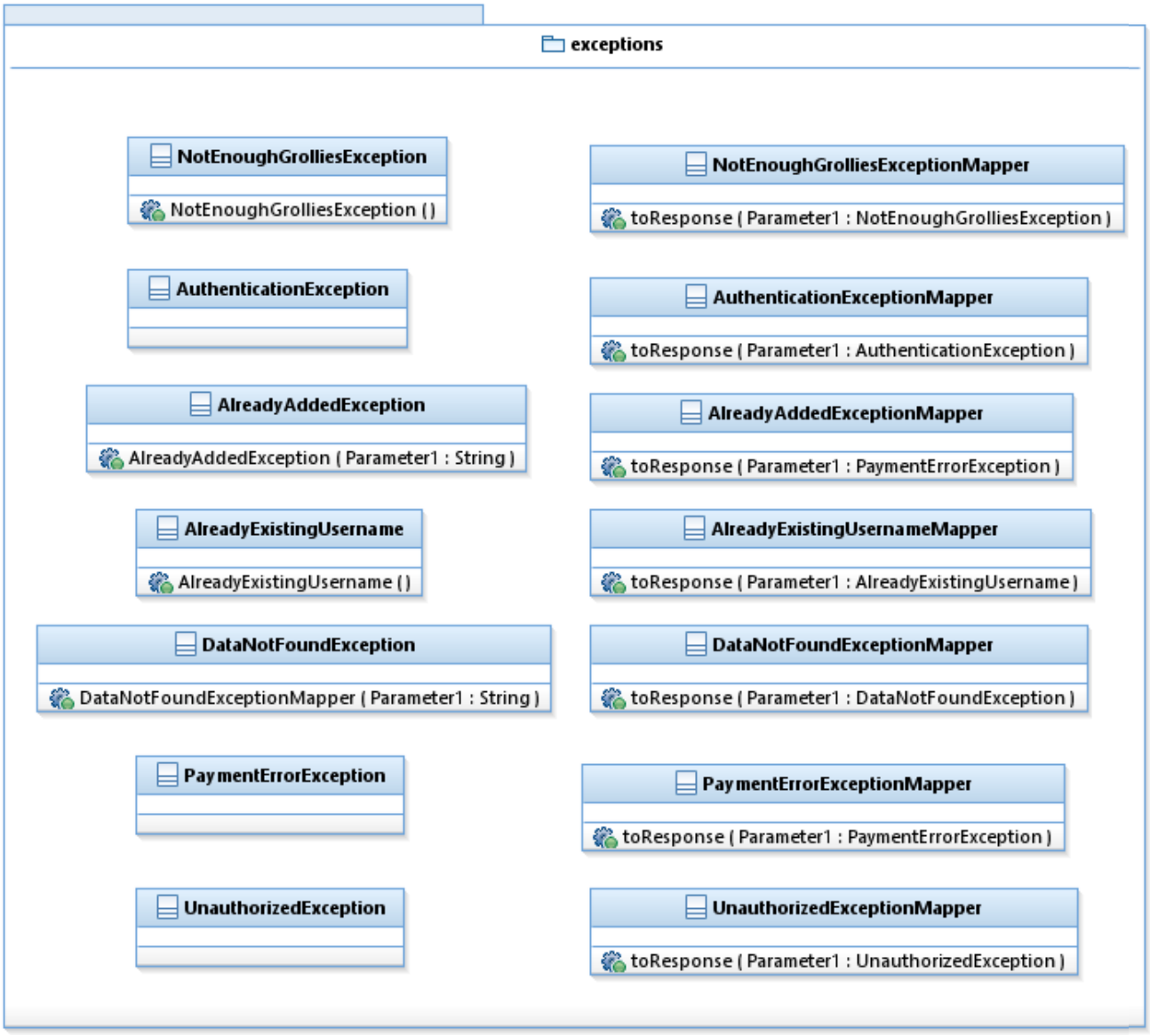

gifts
GiftMigration
migrate ( ) : Void
fillDummy ( ) : Void
GiftsDAO
getAllGifts ( ) : List
getGiftsByTitle (Parameter1 : String) : TransferGift
getPurchasedGifts ( ) : List
purchaseGift (Parameter1 : String, Parameter2 : PurchasedGift) : Void
createGift (Parameter1 : Gift) : Void
getGiftFromResultSet ( ) : Gift
getGiftsById (Parameter1 : Integer) : Gift
getPurchasedGiftFromResultSet ( ) : PurchasedGift

Cada DAO es una clase con métodos CRUD que se encarga de las operaciones de *Create, Read*, *Update* y *Delete* del recurso en la base de datos. Son clases con métodos estáticos. Estos métodos son exclusivamente los que utilizará la lógica de negocio para comunicarse con la base de datos. Por tanto, suponen una abstracción de alto nivel de la base de datos.

Cada Migration implementa de la interfaz Migration, y como hemos explicado antes, define las estructuras de las tablas necesitadas por este recurso y valores de ejemplo para el desarrollo. Desarrollaremos en más detalle estas clases en los servicios que las usan.

4. Paquete exceptions

El paquete de excepciones contiene todas las excepciones que pueden surgir al realizar determinadas acciones. La estructura es la siguiente:



Estas excepciones van acompañadas de un *ExceptionMapper*. Estos *ExceptionMapper* se encarga decirle a JAX-RS qué debería devolver en el caso de que no se capture dicha excepción. Por ejemplo, AuthenticationExceptionMapper devolverá el código de error HTTP 401 (Unauthorized). De esta manera, podemos lanzar estas excepciones durante la aplicación sin tener que preocuparnos por capturarlas, ya que el propio JAX-RS las manejará por nosotros como le hemos dicho.

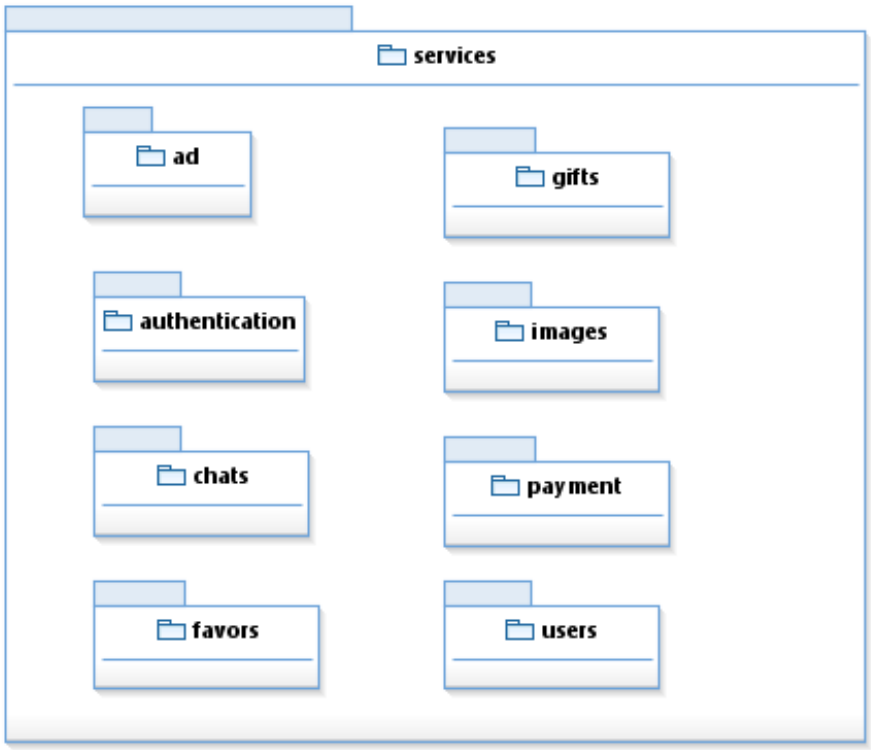
Por ejemplo, si el usuario intenta crear un favor poniéndole una recompensa mayor al número de *grollies* que tiene, se lanzará la excepción NotEnoughGrolliesException, lo que se traducirá en un mensaje de error que se enviará por la petición HTTP. Con propósitos semejantes, tenemos excepciones para gran variedad de fuentes de error, entre las que se incluyen:

- Intentar crear un usuario con un nombre de usuario ya existente (AlreadyExistingUsername).
- Intentar realizar una acción sin tener los permisos necesarios (por ejemplo, borrar un favor de otra persona) (UnauthorizedException).
- Intentar acceder a datos no existentes (DataNotFoundException).
- Errores al pagar las compras que se hagan (PaymentErrorException).
- Iniciar sesión con credenciales incorrectas (AuthenticationException).

Todas estas excepciones son lanzadas donde corresponde y evitan un incorrecto funcionamiento de la aplicación, además de dotarla de un alto nivel de seguridad.

5. Paquete services

Este es el paquete más importante y más denso del servidor al contener la lógica de negocio. Está estructurado en las diferentes funcionalidades que contiene la aplicación:



Los servicios tienen al menos la siguiente estructura común, siendo las clases que comparten todos *Modelo* y *WebServiceBroker* y *Transfer* la que solo está en algunos servicios:

Modelo es la clase del servicio que maneja la lógica de negocio. Por ejemplo, en el servicio de usuarios, la clase del modelo sería *User*, que es la clase que engloba la lógica de negocio del usuario.

Cada WebServiceBroker es la clase que brevemente dijimos antes que JAX-RS se encargaba de crear y ejecutar cada método cada vez que refería una petición a una URL. Por tanto, esta clase tiene distintos métodos, cada uno de ellos representa una combinación de URL y método HTTP. Cuando se hace una petición HTTP, JAX-RS buscará un *WebServiceBroker* que cuadre con el método y la URL de la petición, creará un objeto de esa clase y llamará al método adecuado. A cada método se le puede especificar qué URL tiene asignado utilizando la anotación *@Path,* y se especifica el método HTTP mediante las anotaciones *@GET, @POST, @PUT* y *@DELETE*.

Los *Transfer* son objetos que consisten únicamente en constructores, *setters* y *getter*s. JAX-RS permite la serialización automática de este tipo de objetos a JSON o XML (simplemente en el método del WebServiceBroker, devolviendo un objeto lo serializa automáticamente) y la deserialización automática desde JSON o XML a estos objetos (Declarándolo en el método del WebServiceBroker como parámetro de entrada, automáticamente JAX-RS se encarga de convertirlo). Por tanto, los Transfer son usados como objetos de intercambio entre el cliente y el servidor. No tienen lógica de negocio, su función es puramente transaccional.

Los servicios se han implementado de manera que sean lo más independientes posible (se explicará en el servicio de usuarios las limitaciones).

Vamos a investigar los distintos servicios en mayor profundidad. Como cada servicio que tiene comunicaciones con la base de datos tiene sus respectivas clases en el paquete integration, estudiaremos conjuntamente sus clases del paquete services y del paquete integration.

5.1 Authentication

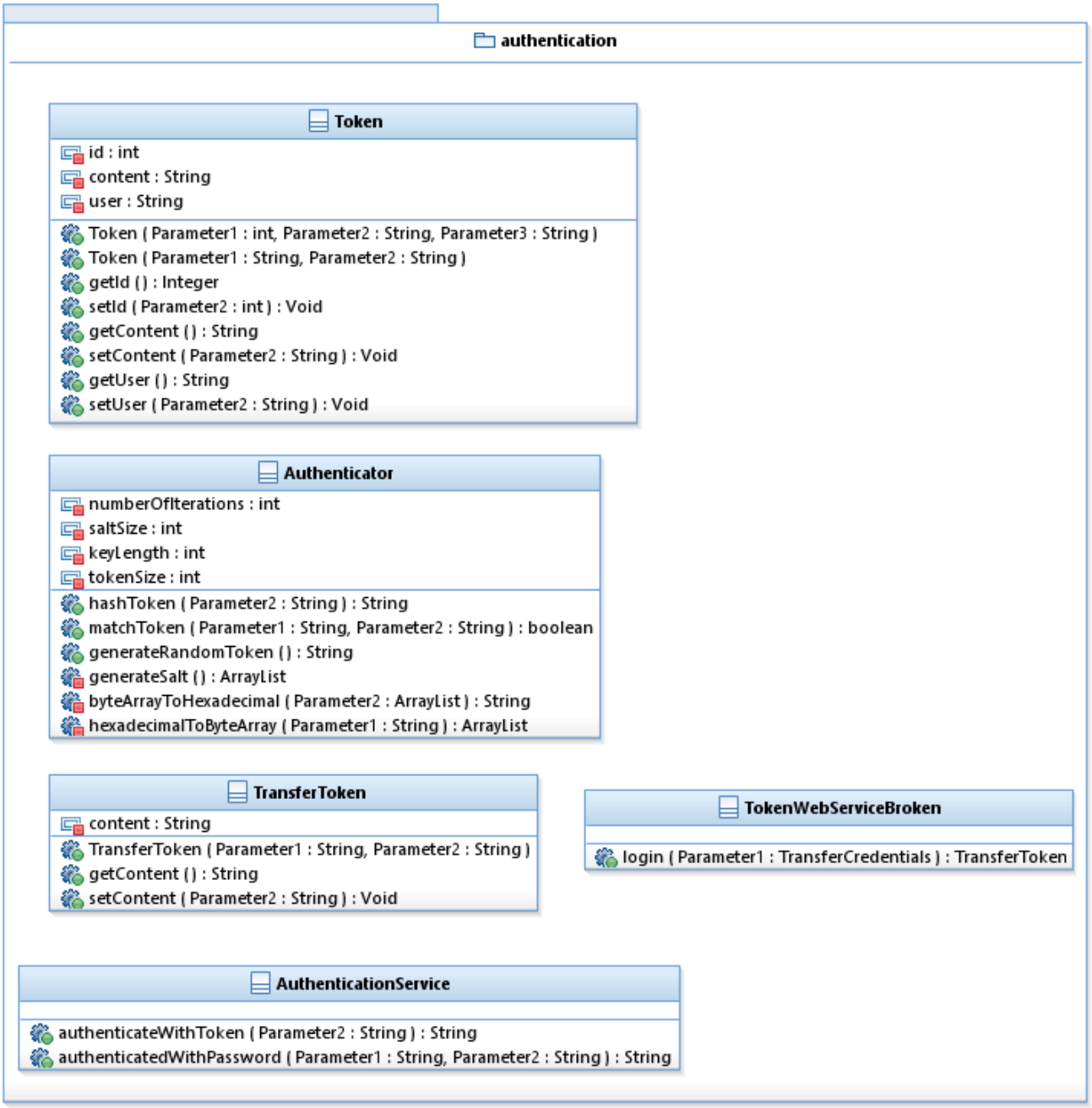
Antes hemos explicado que las peticiones HTTP no tienen estado. Por tanto, en cada petición debemos enviar los credenciales del usuario. Sería contraproducente que el cliente tuviese que almacenar la contraseña del usuario para enviarla en cada petición, ya que si el cliente es comprometido por un atacante, entonces el atacante tendría acceso a la contraseña, que podría compartir con otros servicios.

Por este motivo, introducimos el concepto de *Token.* Un token es una clave temporal que identifica al usuario y que debe mandar con cada petición que requiera de usuarios

autenticados. Estos tokens pueden ser revocados en cualquier momento, y son generados aleatoriamente de manera segura.

Para garantizar la seguridad de nuestros usuarios, almacenamos tanto las contraseñas como los tokens en la base de datos *hasheadas* y *salteadas*, mediante sistemas de *hash* de contraseñas modernos y ampliamente utilizados en la comunidad.

Analizamos el contenido del paquete:



En el paquete de autenticación tenemos varias clases, Authenticator, que se encarga de hashear y saltear las contraseñas y de generar de manera aleatoria tokens y salts; y

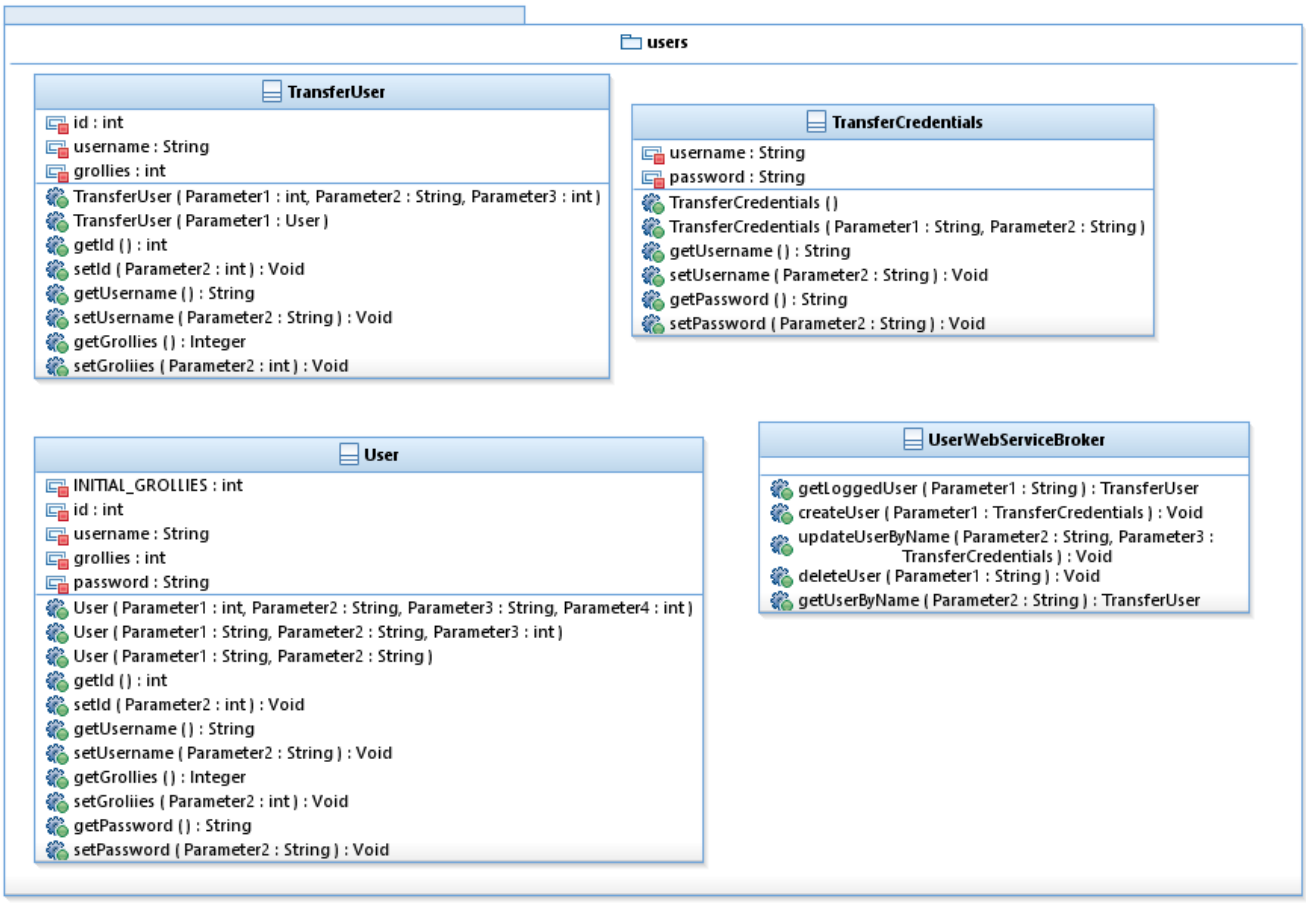
AuthenticationService, que se encarga de encapsular Authenticator, permitiendo rápidamente y con una sola línea de código autenticar a un usuario mediante un token o mediante contraseña.

Además, en este servicio, tenemos TokenWebServiceBroker, que permite generar un nuevo token a través de un usuario y una contraseña, lo que podría interpretarse como *iniciar sesión*, y es lo que se llama desde el cliente para obtener un nuevo token si no tenía ninguno almacenado.

Los tokens también cumplen la funcionalidad de permitir mantener la sesión del usuario iniciada a lo largo de distintas ejecuciones distintas del cliente.

5.2 Users

El paquete de usuarios implementa varias clases:



Tenemos en un primer lugar la clase User, con atributos como el nombre de usuario, la contraseña y su cantidad de *grollies*. También contiene un atributo con un identificador, que por el momento no cumple ninguna función específica, pues la aplicación está

implementada de manera que no pueda haber dos usuarios con el mismo nombre de usuario (lo que permite identificar de manera única a un usuario mediante su nombre). Es un aspecto pensado para implementarse en una versión futura.

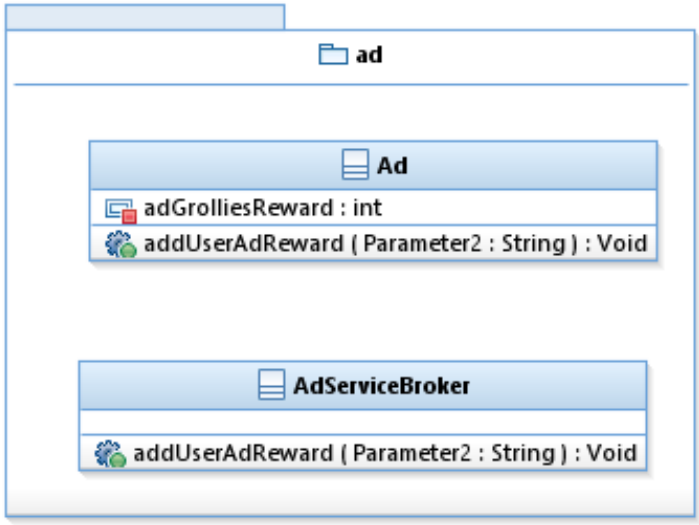
Encontramos además las clases transferencia TransferUser y TransferCredentials (usada para iniciar sesión) y la clase UserWebServiceBroker. Esta última permite gestionar las llamadas que soliciten actualizar la contraseña o registrar un usuario, por ejemplo.

Recordemos la función que cumplen aquí UserDAO y UserMigration, que se encargan de interactuar con la base de datos.

Nos parece importante resaltar que todos los demás servicios están relacionados con este (los favores están creados y son realizados por usuarios, los chats son entre usuarios, es un usuario el que hace una determinada compra para adquirir *grollies*, etc). Debido a ello, el resto de servicios necesitan mantener una determinada interdependencia con el servicio de usuarios.

5.3 Ad

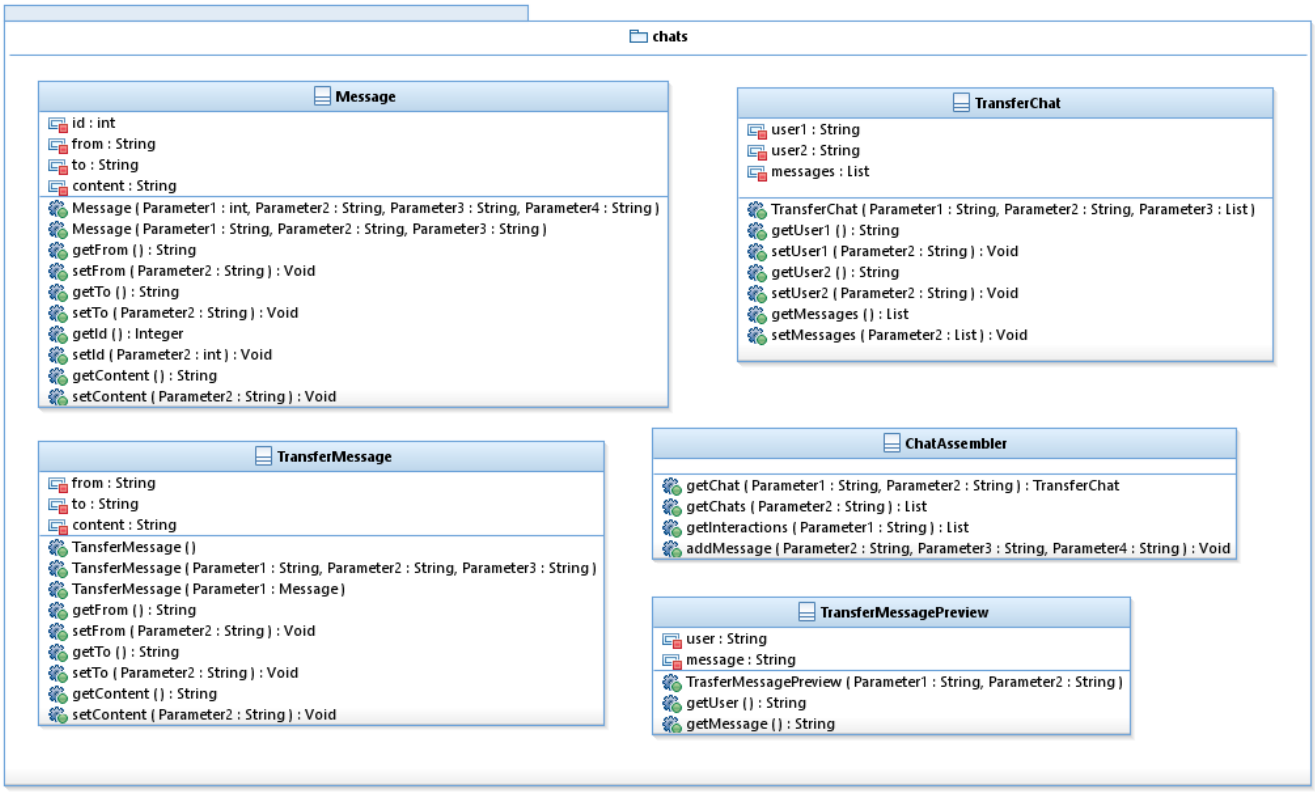
La aplicación permite al usuario obtener pequeñas cantidades de *grollies* a cambio de que este visualice anuncios de hasta 30 segundos. Esta es la funcionalidad más sencilla de implementar, pues la estructura es:



Contiene, por tanto, únicamente las clases Ad y AdServiceBroker, con un método cada una para añadir al usuario la cantidad de *grollies* correspondiente tras haber completado un anuncio. La clase Ad posee además un atributo estático para poder cambiar con facilidad el valor de esta recompensa.

5.4 Chats

Esta funcionalidad es diferente a las demás, pues implementa el patrón TOA (*Transfer Object Assembler*). Su estructura es:



Debido al uso del patrón TOA, contiene la clase ChatAssembler, encargada de gestionar todo lo que sucede en los chats de un usuario, implementando métodos para obtener el chat entre dos usuarios, todos los chats a la vez o añadir un mensaje a una conversación.

Sin embargo, en ningún momento se almacenan los chats en la base de datos, sino que se guardan únicamente los mensajes. Debido a ello, tenemos la clase ChatWebServiceBroker, pero tenemos MessageDAO (en el paquete integration) en lugar de ChatDAO.

Implementamos los mensajes mediante los atributos de quién lo ha mandado, a quién se ha mandado, el contenido del mensaje y un ID que identifica unívocamente a cada mensaje. De esta manera, podemos buscar fácilmente los mensajes que nos interesen, e incluso ordenarlos por fecha sin tener implementadas las fechas (en un chat, queremos que los mensajes se muestren en orden, de más antiguo a más reciente).

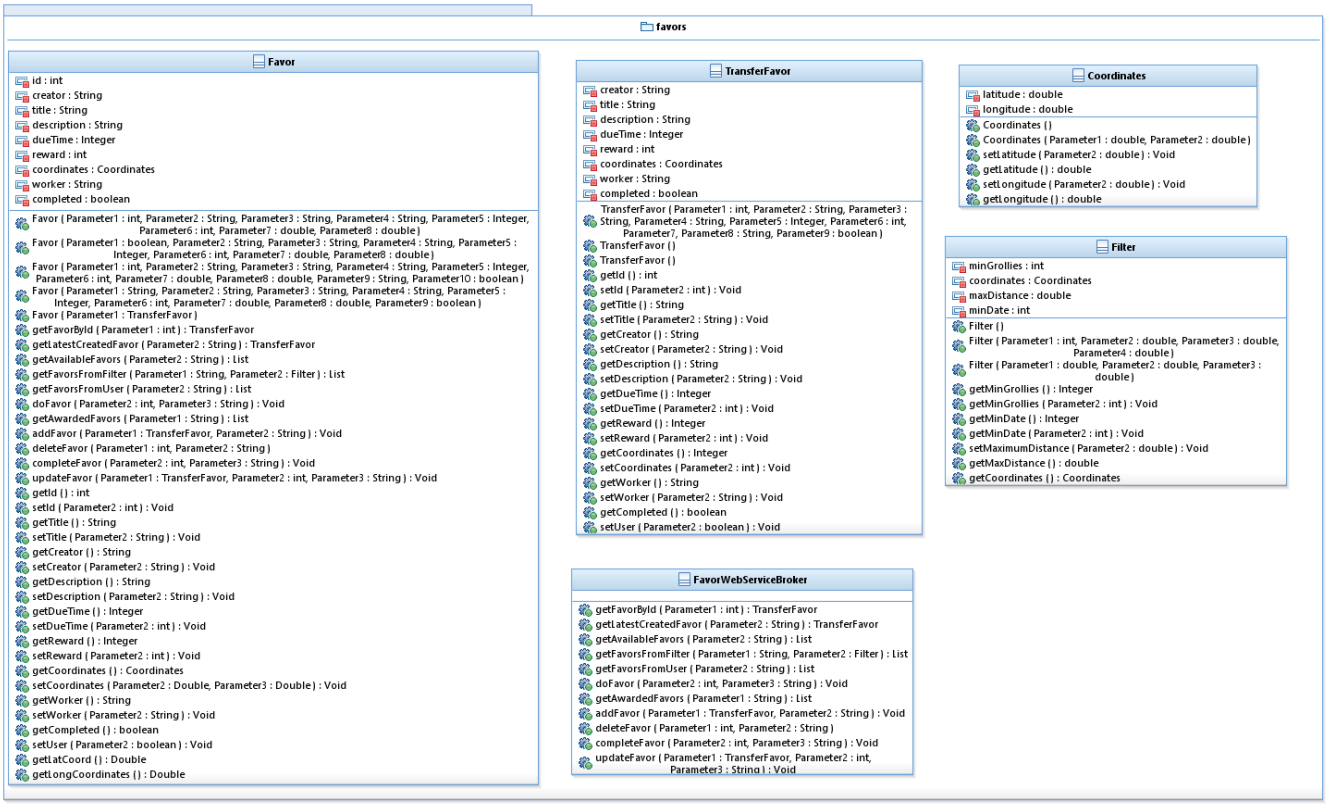
El paquete contiene también las clases *Transfer* correspondientes, como en todos los servicios que intercambian información con la vista. Entre esas clases, destaca TransferPreview, que se utiliza en el método *getInteractions*, que devuelve una lista de vista previas de mensajes que puede utilizar el cliente para mostrarle una lista de conversaciones al usuario al estilo WhatsApp o Telegram.

En el MessageDAO, al igual que en todos los DAO, implementamos las funciones CRUD que tienen relación con el objeto, en este caso con los mensajes. Como es lógico, esta clase nos permitirá crear un mensaje y devolver todos los mensajes de una conversación para mostrarlos posteriormente en la vista.

En el MessageMigration creamos una tabla en nuestra base de datos en la que guardaremos los mensajes de nuestros clientes. Como se ha explicado anteriormente, es muy importante el hecho de que cada mensaje tiene un emisor y un receptor, además de un id. Así, mediante el id, los mensajes de nuestros chats estarán ordenados de más antiguo a más reciente.

5.5 Favors

Este servicio es uno de los más completos de la aplicación, lo cual es lógico al tratarse de la funcionalidad principal, lo que queda reflejado en su estructura:



Contiene 2 clases para implementar los favores, la clase transferencia correspondiente, una clase para interactuar con el cliente, el Service Broker, el DAO, y la Migration.

Los favores están implementados con ayuda de un sistema de coordenadas con atributos que indican su creador, su título, su descripción, su fecha límite, su recompensa, el lugar de realización (con coordenadas), la persona encargada de hacerla (si hay) y si el favor está completo, además de un identificador similar al de los mensajes.

La aplicación contiene una gran cantidad de aspectos que involucran los favores. Por una parte, en la pestaña “Favores” de la aplicación podemos crear un favor nuevo, editar o borrar uno ya existente, mirar la lista de favores que hemos creado y mirar la lista de favores que tenemos que realizar. Por otra parte, en la pestaña “Explorar” podemos ver los favores de otras personas, filtrarlos de acuerdo a nuestros intereses (mediante unos determinados atributos que podemos encontrar en la clase Filter) y adjudicarnos un favor libre para realizarlo.

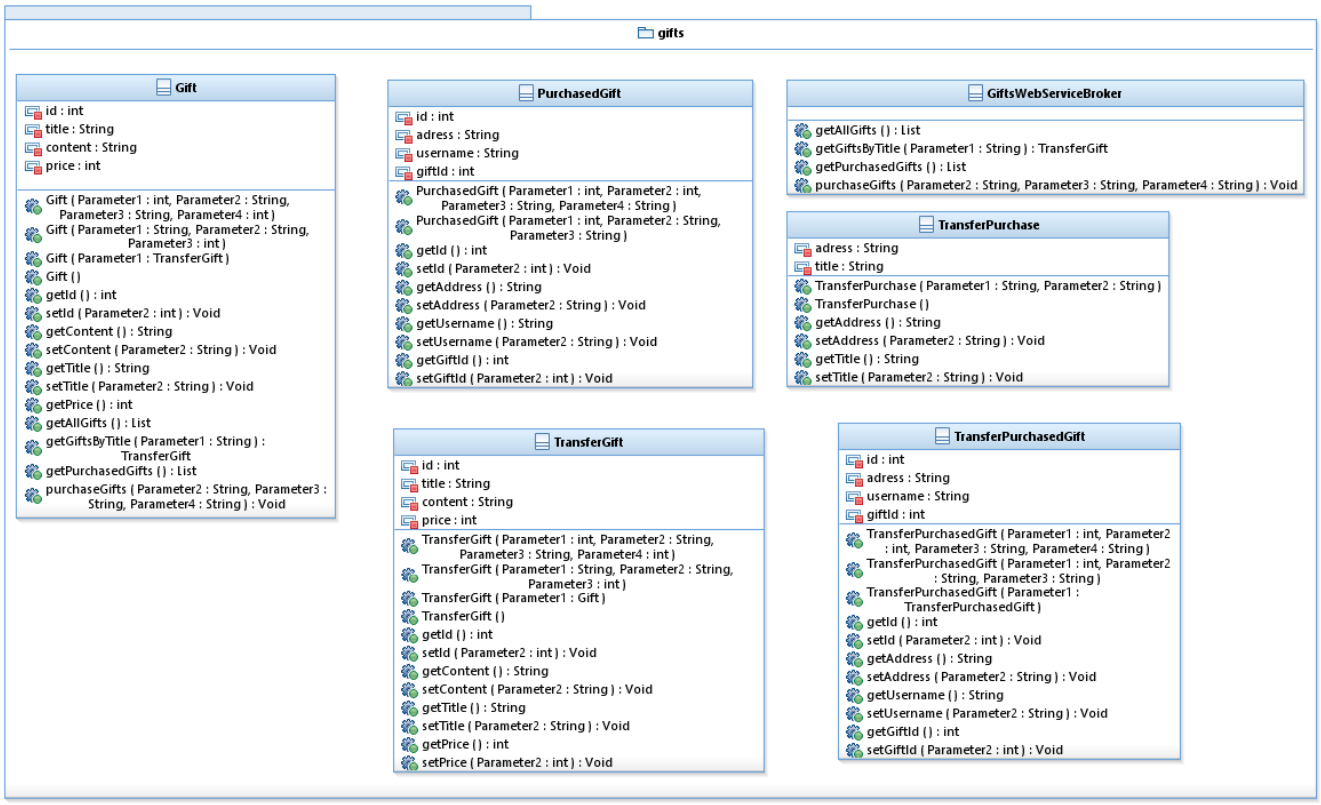
Debido a esto, podemos encontrar gran cantidad de métodos para realizar todas estas operaciones en las clases Favor y FavorWebServiceBroker.

En el FavorDAO, no solo podemos crear y borrar favores sino que también podemos actualizar un favor, ver todos nuestros favores pendientes, ver nuestros favores a realizar…
Todas estas acciones deben estar implementadas en el DAO, ya que son mucho más eficientes de realizar en la base de datos que en el servidor con Java.
También podemos buscar favores según un filtro que hemos introducido en la aplicación (distancia, recompensa y tiempo). Los favores son la principal funcionalidad de nuestra aplicación y, como es lógico, este es el DAO más completo y complejo. Otra de las funcionalidades de nuestros favores es que el usuario que lo ha creado puede “completarlo” y de esa manera da el favor por realizado, completando la transacción de grollies del creador del favor hacia la persona que ha completado dicho favor. Este DAO se encarga de implementar todas estas funcionalidades, que el cliente puede realizarlas mediante el FavorWebServiceBroker.

En el FavorMigration creamos una tabla en nuestra base de datos en la que guardaremos los favores creados por los usuarios. Estos favores tendrán, entre otras cosas, dos enteros que usaremos para ubicar el favor (coordenadas) y un parámetro para saber si se ha completado o no el favor. El resto serán datos del los favores y, si el favor está cogido por otro usuario para realizar, el nombre del usuario que realizará ese favor.

5.6 Gifts

La aplicación permite a los usuarios canjear los *grollies* que posean por productos que la aplicación enviará por correo ordinario a la dirección que soliciten, con la siguiente estructura:



Los regalos están implementados en la clase Gift mediante un identificador, un nombre, un contenido y un precio (en *grollies*). Una vez un usuario compra un regalo, se crea un objeto de la clase PurchasedGift, que contiene la información necesaria para gestionar el envío, tales como un ID de pedido, el ID del regalo que se ha comprado y la dirección a la que se ha de enviar.

La clase que gestiona todas estas operaciones actuando de puente desde que el usuario termina de comprar hasta que la información llega a la base de datos es GiftWebServiceBroker, que además permite a la vista obtener información de los regalos disponibles.

Cuando un usuario compra un regalo, se crea un objeto en la tabla de regalos comprados con el id del regalo que ha comprado, que podemos obtener de la tabla de regalos, y con el nombre del usuario que lo ha comprado. Además, los grollies del usuario que ha comprado el regalo se reducirán en la cantidad correspondiente. Tendremos un

mismo DAO y una misma Migration para los regalos de la aplicación y los regalos que han sido comprados por algún usuario.

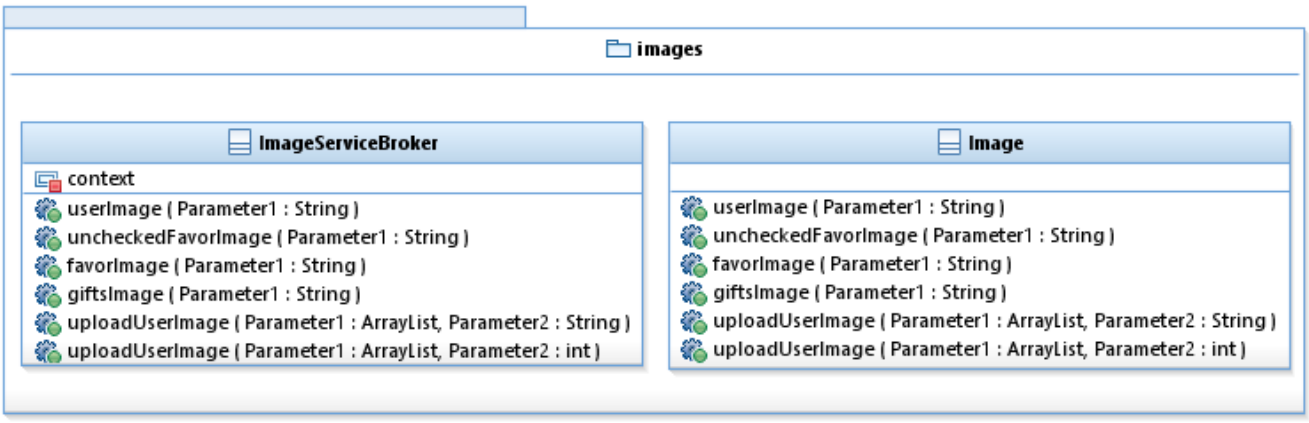
En el GiftDAO tenemos todas las funciones necesarias para guardar y mostrar los regalos. Este DAO es algo distinto del resto ya que en un mismo DAO controlamos dos tipos de objetos. Por una parte realizamos las operaciones CRUD sobre los regalos. Por otra parte, y a diferencia de los demás DAO, también realizaremos las pertinentes operaciones CRUD de los regalos comprados, que son un objeto distinto de los regalos normales.

En el GiftMigration creamos dos tablas en nuestra base de datos. Por una parte crearemos la tabla de los regalos que enseñaremos a nuestros clientes y por otro lado crearemos la tabla de los regalos que han sido comprados por algún usuario. Estos regalos tendrán asociados el nombre de usuario que, al ser único, nos facilitará el trabajo para relacionar regalos y usuarios.

5.7 Images

En Logrolling damos la opción al usuario para subir sus propias imágenes tanto como foto de perfil como para cada favor. Este servicio se encarga de manejar este aspecto.

Las imágenes son almacenadas en formato JPEG en una ruta que depende de la localización del servidor, sin depender del sistema operativo. La estructura del paquete es:

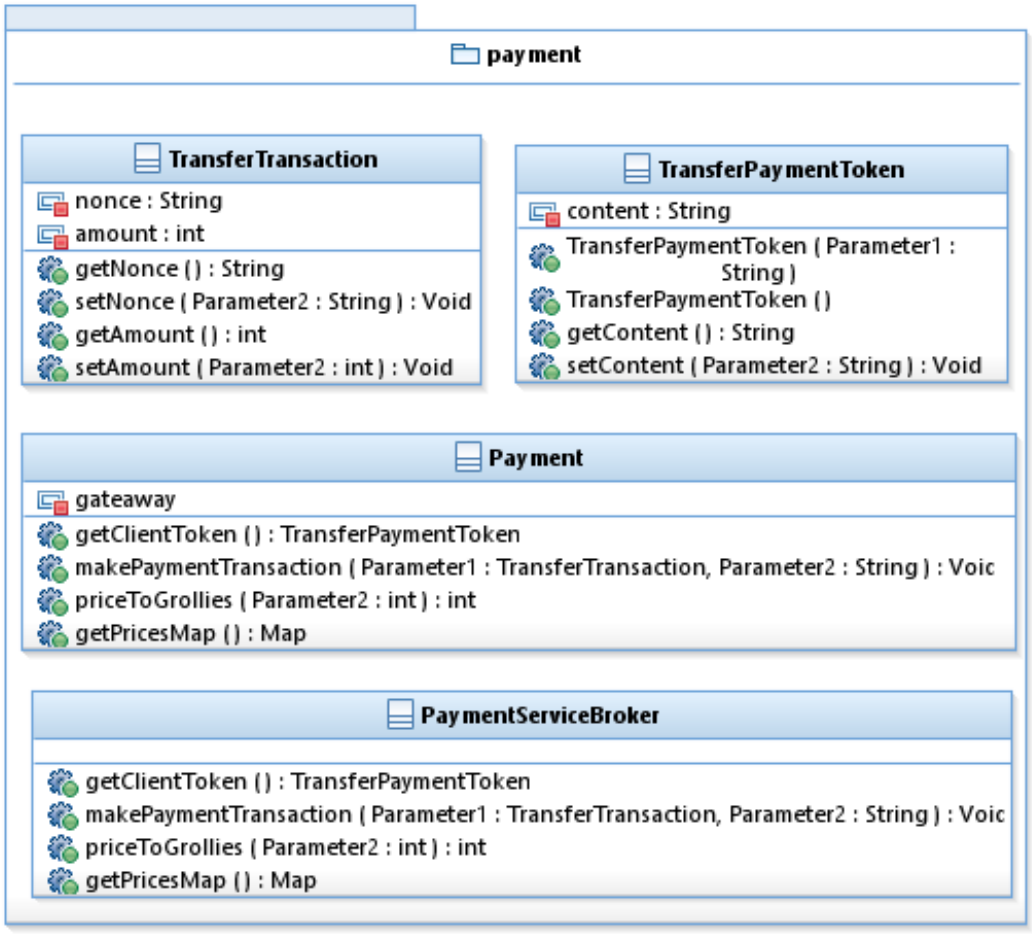


A través de ImageServiceBroker e Image, el cliente puede subir sus propias fotos o solicitar alguna imagen. Si la imagen no existe, en ciertas peticiones se devuelve un error y en otras se devuelve una imagen por defecto. Todo está documentado en la API web.

5.8 Payment

Para cumplir la normativa europea y para seguridad de nuestros usuarios, nosotros no manejamos ni almacenamos datos de tarjetas de crédito. Utilizamos una plataforma externa (Braintree) para manejar los pagos. Esta plataforma nos permite aceptar tanto tarjetas de crédito como PayPal, así como verificar desde el servidor que un pago ha sido realmente realizado, para hacer la transacción de grollies.

Desde la web de Braintree podemos además ver un resumen de las transacciones en los últimos días, realizar reportes y controlar los ingresos de nuestra aplicación. La estructura del paquete es:



En la clase Payment manejamos la integración con la API de Braintree. Desde PaymentWebServiceBroker ofrecemos al cliente la posibilidad de obtener un token de cliente de Braintree (necesario para que pueda realizar una transacción monetaria) o realizar una transacción de pago (recibimos como dato un token que enviamos a Braintree y nos indica si la transacción ha sido exitosa o no).

De esta manera, podemos trabajar con pagos de manera eficiente y segura en nuestra aplicación.

6. Notas de implementación

6.1 Sistemas distribuidos

En todo momento hemos hablado de **él** servidor, como si solamente hubiese uno. Sin embargo, una de las ventajas de los servidores que implementan API REST, es que al no tener estado, es fácilmente distribuible. Por tanto, Logrolling funciona perfectamente en entornos distribuidos.

La única necesidad sería un Load Balancer, que distribuya uniformemente las peticiones a los distintos servidores que existan.

Lo mismo ocurre con la base de datos. MySQL puede ejecutarse en entornos distribuidos. Por tanto, Logrolling escalaría bien tanto verticalmente como horizontalmente.

6.2 Plataformas de implementación

Para nuestra implementación, hemos utilizado una plataforma de Cloud Services ampliamente utilizada en la industria: Microsoft Azure. Tenemos un servidor en Azure dedicado exclusivamente a Logrolling, que ejecuta tanto la base de datos como nuestro servidor HTTP.

6.3 Sistemas de contenedores

El servidor HTTP es un Servlet de Tomcat. Por tanto, puede ser integrado perfectamente en sistemas de contenedores, como Docker o Kubernetes.

6.4 Server-Side Verification (SSV) Callbacks

Si la aplicación crece, sería posible solicitar a Facebook la utilización de la API Server-Side Verification para los anuncios, que permite confirmar que el usuario realmente ha terminado de ver un anuncio para obtener su recompensa.

6.5 JSON / XML

En todo el documento hemos hablado de como la comunicación con el servidor se produce a través de JSON, pero también podría configurarse el servidor fácilmente para aceptar o devolver tanto JSON como XML, permitiendo más formatos de intercambio.

7. Patrones utilizados

En el servidor de Logrolling hemos usado varios patrones de diseño de software estándar, pero en cada caso modificándolos ligeramente para adecuarlos a las necesidades concretas de nuestro proyecto.

7.1 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 la capa de persistencia, que almacena todos los datos, y la aplicación Android contiene la vista y el controlador que le permite comunicarse con el servidor. 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.

7.2 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. Por tanto, los *WebServiceBrokers* devuelven objetos *Transfer* y reciben objetos *Transfer* como parámetros.

7.3 TOA

En Logrolling utilizamos el patrón TOA para construir el objeto Transfer para los chats. No almacenamos directamente los chats como chats en la base de datos, sino como mensajes que hay que *ensamblar* en otro transfer object más completo. Para ello utilizamos el patrón TOA, que nos permite mantener la base de datos sencilla y poder intercambiar toda la información necesaria con el cliente de golpe.

7.4 DAO

En nuestro proyecto, necesitamos tener DAOs que realizarán todas las operaciones CRUD, pero nos conviene hacer unas pocas 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 (Database), 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 Database, 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 dependería 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.

7.5 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 debe mantenerse 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.

Por tanto, podemos, al inicializar la aplicación, elegir la factoría que elegimos para nuestra base de datos, y en la aplicación obtener siempre un objeto de dicha factoría, para asegurar la coherencia.

7.6 Aplication Service

Aunque no explícitamente, en Logrolling utilizamos el patrón Application Service. No definimos una clase *Service* explícitamente, pero los *WebServiceBroker*, a través de la conexión HTTP, actúan como puntos centralizados para acceder a la lógica de negocio de cada servicio. Además, el servidor está dividido en distintos servicios, como hemos indicado anteriormente. Esta centralización de la lógica de negocio simplifica la implementación de distintas vistas o clientes, ya que no se tienen que preocupar por cómo está implementada la lógica de negocio. Los clientes tienen un punto de acceso único a cada servicio (el *WebServiceBroker*).

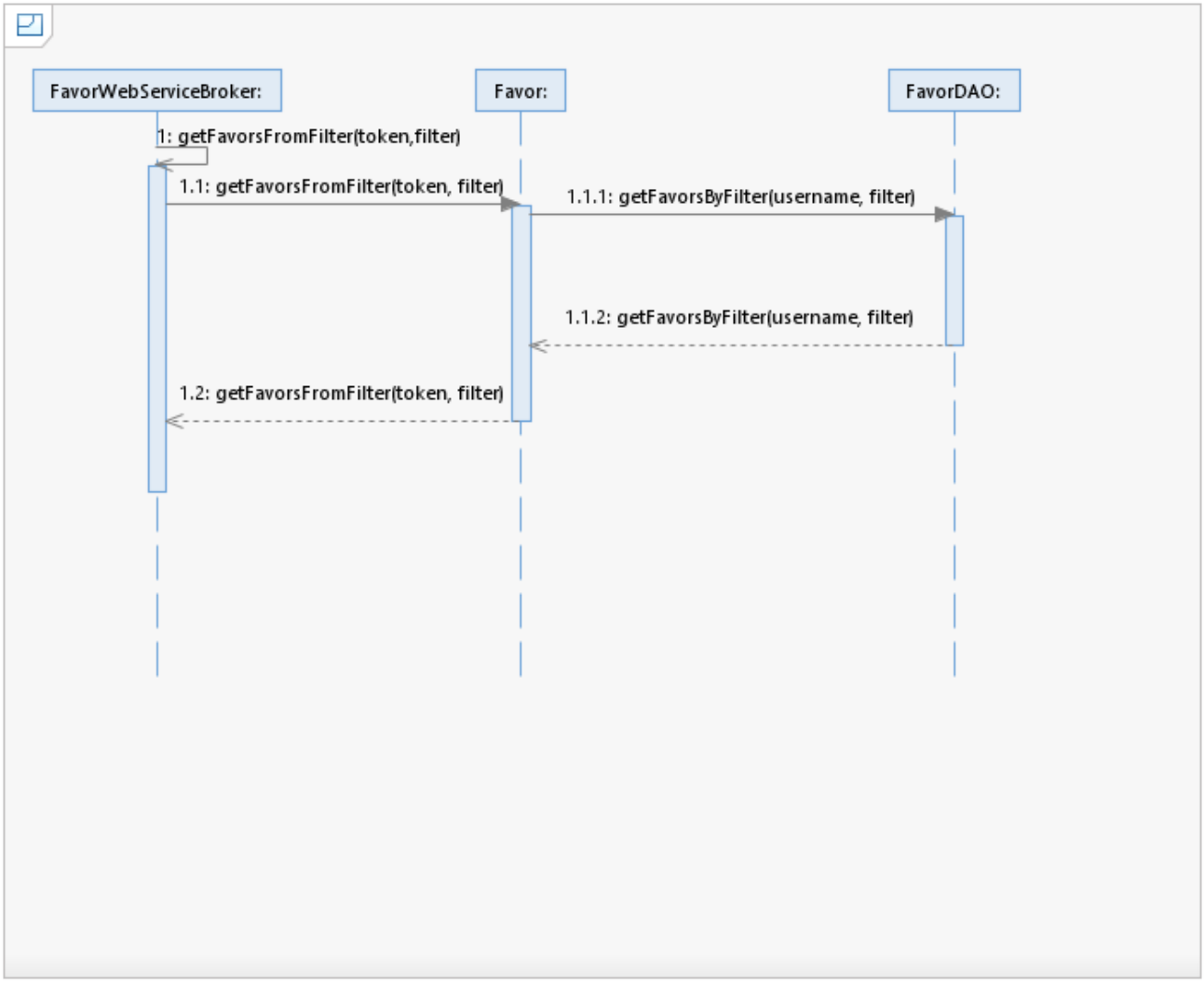
7.7 Iterator

En el servidor, la interfaz Database devuelve como resultado un ResultSet, que es un iterador, se avanza utilizando el método *next*. Utilizar iteradores para recorrer los resultados de una base de datos tiene mucho sentido, ya que el resultado es una lista de filas, que pueden ser recorridas efectivamente mediante iteradores.

8. Diagramas de secuencia

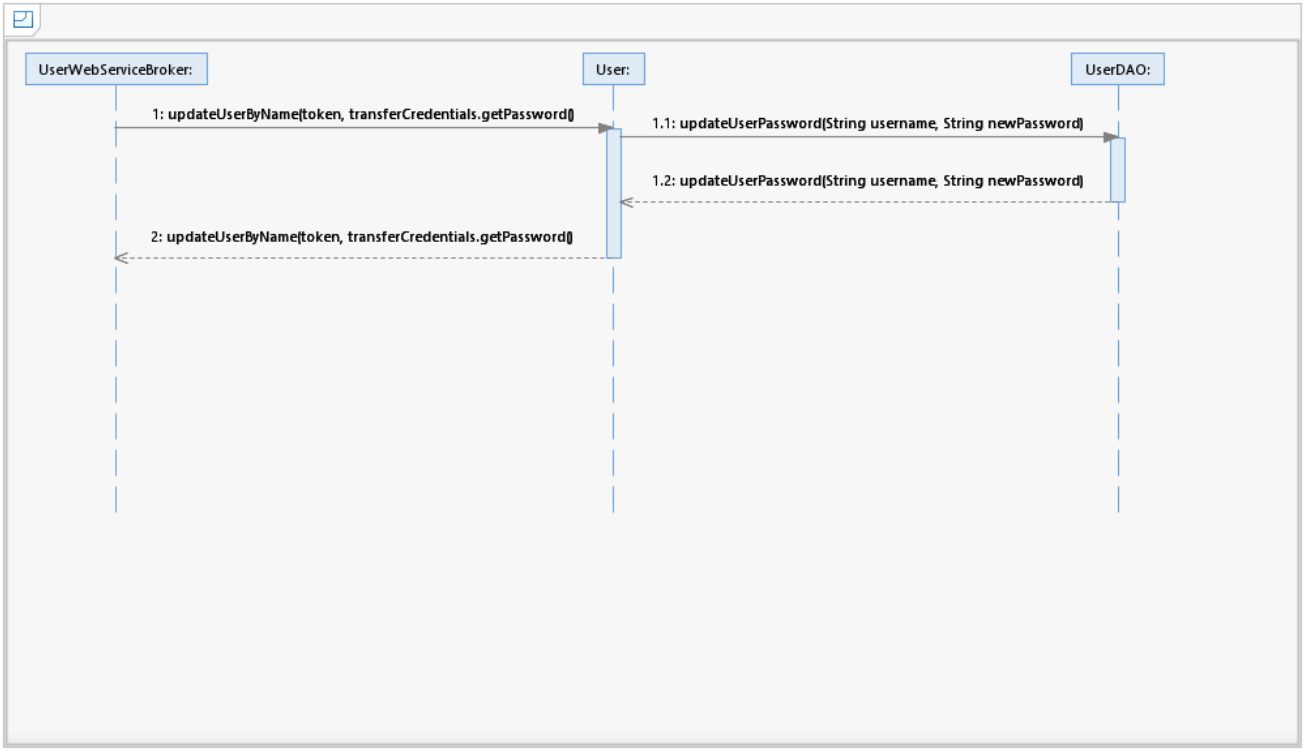
8.1 Buscar por filtro

¿Explicación?



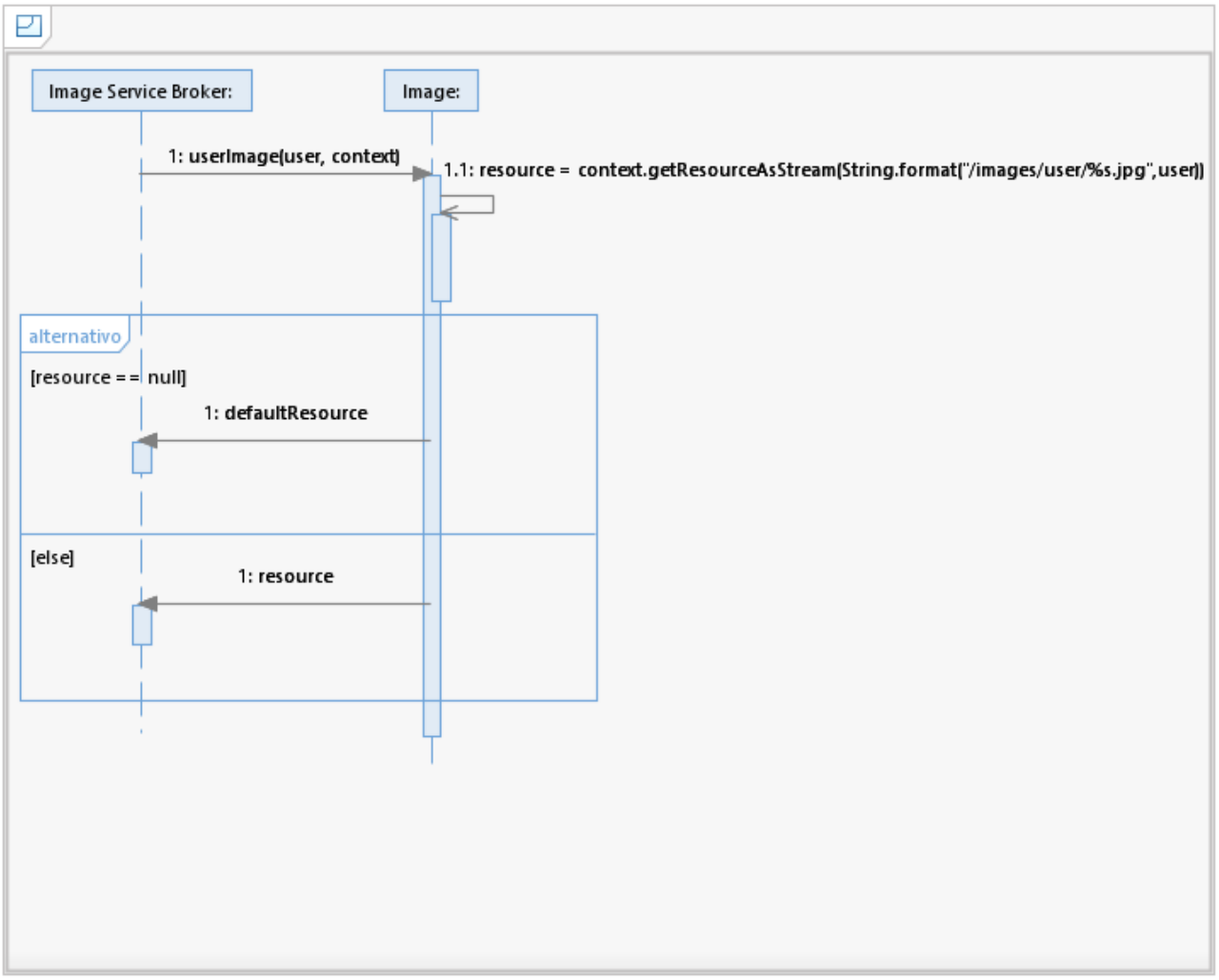
8.2 Cambiar contraseña

¿Explicación?



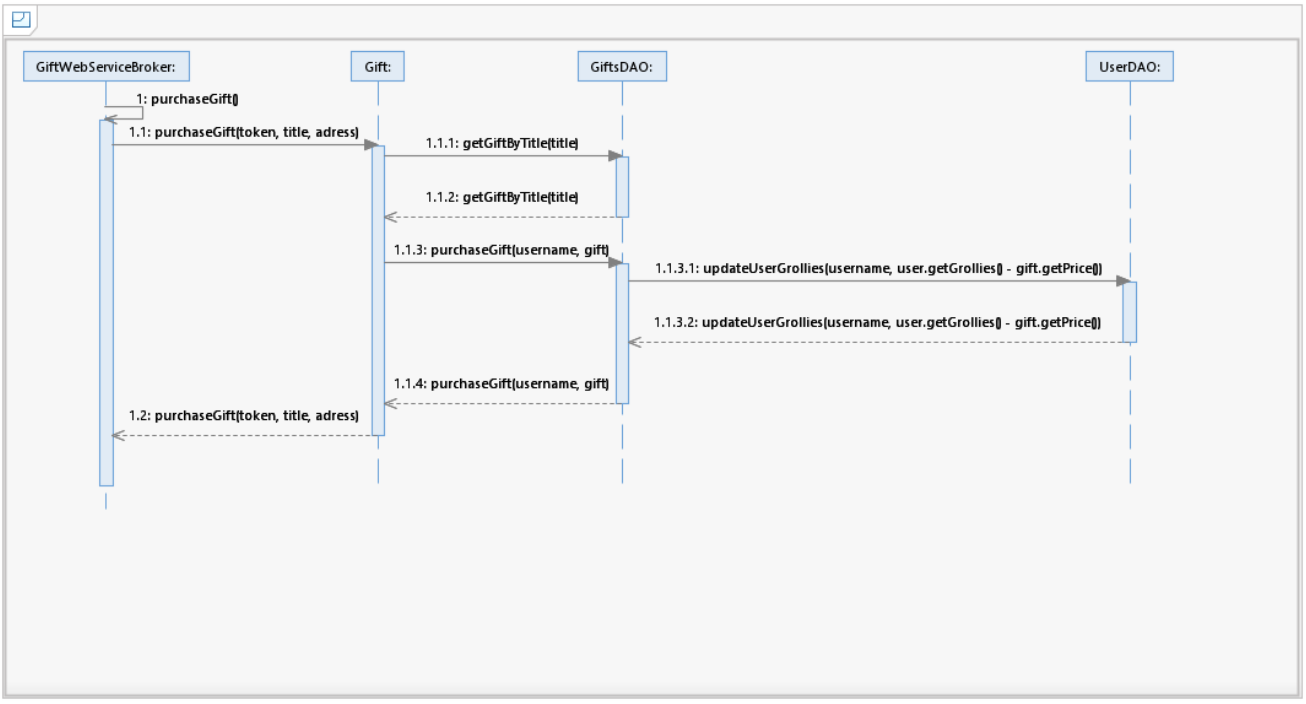
8.3 Cambiar foto de perfil

¿Explicación?



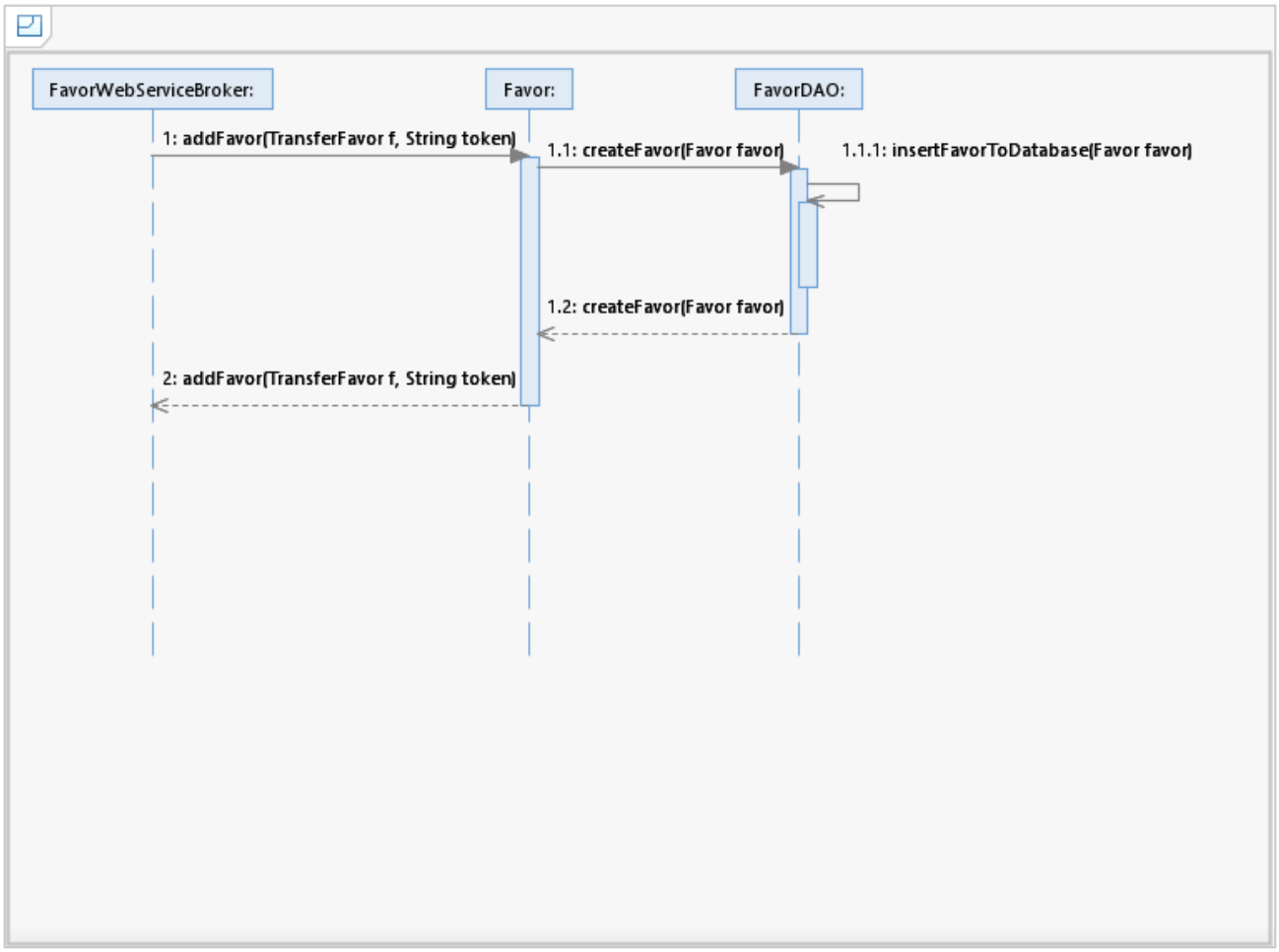
8.4 Comprar regalo

¿Explicación?



8.5 Crear favor

¿Explicación?



8.6 Iniciar sesión

¿Explicación?