Desarrollo de API en Java

En el siguiente documento será descrito el desarrollo de una API de acceso y modificación de una base de datos en Java, la cual hace el papel de backend en nuestro proyecto del método del caso. Se explicará tanto la arquitectura general de nuestra aplicación como los patrones de diseño que se han incorporado y aplicado para obtener el resultado final.

Antes de comenzar la discusión sobre cuestiones de diseño propiamente dichas, me gustaría hacer un breve inciso sobre nuestra decisión de usar Java. Para nuestro equipo de desarrollo Java resultó la opción evidente puesto que se trata del lenguaje de programación con el que mayor experiencia tenemos. Por otro lado, debido a la estructura global del proyecto, no monolítica, sabíamos desde un principio que no tendríamos que gestionar el servir la paǵina web, uno de los motivos que nos podrían haber impulsado a usar lenguajes como Python con servicio web mucho más sencillo. Finalmente consideramos que la rigidez de Java sería ideal para mostrar el uso de diversos patrones de diseño.

Arquitectura

Para decidir la arquitectura de nuestra aplicación es necesario primero considerar cuál es la función que esta va a desempeñar en el contexto global del proyecto. Nuestra aplicación en Java es la encargada de entrar en contacto con la base de datos, es la entidad Servidor Cudeca que ya planteamos en los diagramas de secuencia. Además, esta debe de ser capaz de recibir y responder peticiones del frontend, volviendo una vez más a los diagramas de secuencia, la entidad InterfazWeb.

Con estas consideraciones en mente nos resultó evidente y natural estructurar esta aplicación siguiendo una estructura en capas convencional, donde hay una capa de datos (acceso a la base de datos), un capa de negocio donde se gestiona la lógica del programa y una capa de presentación, que en este caso consiste en el tratamiento y envío de mensajes para el frontend.

De forma tangencial podríamos argumentar que nuestro proyecto globalmente sigue un esquema Modelo-Vista-Controlador, donde la Vista es el frontend, el actor InterfazWeb, el Controlador es la conexión a través de la API y el modelo es la capa de negocio previamente mencionada que contacta con la base de datos. No obstante, en el equipo inicialmente hubo un interés por plantear el proyecto siguiendo una arquitectura en microservicios, donde esta API desarrollada en Java no sería más que uno de esos servicios, el encargado de acceder a la base de datos, y habría otros servicios, como la pasarela de pago o incluso un hipotético servicio de login (hipotético pues nuestra propuesta de proyecto no hace uso de un sistema de login convencional sino que emplea un mecanismo más rudimentario de control de acceso en búsqueda de una mayor simplicidad), cuya implementación quedaría fuera de los objetivos del proyecto en su fase actual. En cualquier caso, estas consideraciones se escapan del objetivo del presente documento.

Además de la división horizontal en capas, de forma casi semejante a una torre de protocolos, dentro de cada capa también nos vimos impulsados a realizar ciertas divisiones para mejorar la organización. Para entender estas divisiones es necesario considerar la base de datos con la que vamos a trabajar, concretamente hemos de recordar el diagrama de entidad-relación que propusimos en la primera entrega.

En este diagrama distinguimos fundamentalmente 4 entidades: eventos, entradas, tickets y usuarios. La motivación detrás de representar los datos siguiendo este formato ya fue explicada en su respectivo documento. Por tanto, prácticamente en todas y cada una de las capas de nuestro diagrama, podemos ver clases, en ocasiones encapsuladas en subpaquetes cuando se considera conveniente, que representan una de estas entidades dentro de dicha capa.

Habiendo discutido nuestro modelo arquitectónico a un nivel teórico, veo conveniente mencionar los paquete que implementan cada capa dentro de nuestra aplicación:

- Paquete Datos: nombre autoexplicativo. Este paquete está formado por las clases, los objetos, que usaremos a lo largo de nuestra aplicación y está formado esencialmente por las clases que definimos en el diagrama de clases de la primera entrega. Este paquete alberga un paquete para cada entidad de la base de datos (Evento, Entrada, Ticket, Usuario)

- Paquete DB: este paquete se encarga de la conexión con la base de datos, para ello hacemos uso del patrón DAO y de otra serie de patrones de diseño que explicaremos en el siguiente apartado. Podemos entender esta capa como la capa de datos de nuestra aplicación, pues transforma los datos almacenados en un sistema de almacenamiento persistente (una base de datos en nuestro caso) en objetos (del paquete Datos) Java que serán usados por la lógica de negocios.

- Paquete Managers: este paquete representa la capa de negocios de nuestra aplicación. La mayoría de sus clases son sencillas debido a la naturaleza de la aplicación, pues su uso principal es mover datos desde y hasta la base de datos. No obstante, destacamos la clase CompraManagers que se encarga de gestionar la lógica propia del proceso de comprar un entrada, usando una clase Transaction como estructura de datos auxiliar

- Paquete API: paquete que supone la capa de presentación de nuestra aplicación, es el punto de contacto con el frontend. Por un lado tenemos la clase App, encargada de iniciar el servidor y definir los endpoints de la API. Para esta función hacemos uso de la dependencia Javalin Java, un framework de desarrollo web en Java que a diferencia de otras soluciones más extendidas a nivel empresarial como Spring, destaca por su ligereza y facilidad de uso, adecuada para una API como la nuestra.

  Por otro lado tenemos las clases Handlers, las cuales definen una serie de funciones con las que dar servicio a los distintos endpoints. Estos además se encargan de convertir los datos de las peticiones del servidor en objetos Java con los que la capa de negocio pueda trabajar.

- Paquete utils: utilidades misceláneas usadas a lo ancho de la aplicación. Principalmente contiene métodos estáticos relacionados con la serialización o deserialización de json. Si bien para este proceso hemos hecho uso del paquete Jackson de Java, aún así ha sido necesario encapsular ciertos métodos.

En resumidas cuentas, nuestra aplicación o bien recupera datos de la base de datos, los convierte a objetos Java, los trata adecuadamente y los envía al frontend como un json en el formato pactado entre los dos equipos de desarrollo (API signature), o bien recibe datos del frontend que deben ser tratados por la capa de presentación y que suele conllevar una inserción en la base de datos.

Aunque no toda la funcionalidad se utilice en la aplicación final, hemos procurado crear un sistema CRUD, en el que el frontend pueda crear, leer, actualizar y eliminar datos de la base de datos.

Antes de pasar a discutir los patrones de diseño empleados a los largo de la aplicación, me gustaría dedicarle una pequeña sección a la lógica de negocios asociada al proceso de comprar entradas, gestionada en la clase CompraManager dentro del paquete Managers. La mayor peculiaridad de este sistema es el uso de objetos Transaction como estructura de datos auxiliar. Cuando un usuario ha rellenado toda su información y ha elegido las entradas que quiere comprar, esta información es almacenada como una transacción, y CompraManager almacena un mapa de todas las transacciones pendientes con un ID único. De esta manera, la transacción se podrá confirmar en el futuro cuando ya se haya procesado el pago y solo entonces almacenar los tickets comprados en la base de datos.

Más allá, el almacenar de esta forma las transacciones pendientes nos permite comprobar qué entradas se pretenden comprar aunque aún no se haya confirmado la transacción a la hora de comprobar la disponibilidad de entradas. Es decir, si solo queda una entrada y un cliente la solicita comenzando una transacción, dicha entrada quedará reservada y no se la podrán quitar mientras se procesa el pago.

Patrones de diseño

Una vez tenemos una visión general de la arquitectura y funcionamiento de nuestra aplicación, podemos empezar a discutir los patrones de diseño que han sido empleados. Cabe destacar una realidad del proyecto; debido a su simplicidad lógica, siendo la lógica vinculada a la gestión de transacciones la más compleja de todas, los patrones a usar se han visto limitados a patrones creacionales, algún que otro patrón estructural y ningún patrón de comportamiento.

Sin lugar a dudas ha habido un patrón estructural que ha destacado por encima del resto: el patrón DAO (Data Access Object). Este patrón oculta la complejidad del acceso a la base de datos y ofrece una interfaz sencilla que la capa de negocios puede utilizar para acceder a los datos persistentes de la aplicación. Hemos creado una interfaz *i{ENTIDAD}DAO* para cada una de las entidades destacadas de la base de datos para poder acceder a información sobre dichas entidades de forma organizada. Más allá, contamos con la interfaz

iDatabase que se encarga de la conexión inicial con la base de datos. Realmente ambas entidades representan el mismo concepto de DAO, pero con el objetivo de modularizar el código hemos separado la inicialización de la conexión y la realización de consultas en interfaces diferentes.

El mayor beneficio de este patrón de diseño es la flexibilidad que nos ofrece de cara a acceder a la base de datos. Para el desarrollo de nuestra aplicación hemos usado MySQL como sistema de bases de datos debido a nuestro conocimiento previo de su sintaxis y a su facilidad de uso, y por tanto en el código hemos desarrollado implementaciones de las interfaces previamente descritas apropiadas para MySQL (MySQLConnection, EventoDAOMySQL, UsuarioDAOMySQL, etc). No obstante, si se nos exigiese el uso de una base de datos distinta, sería tan sencillo como realizar nuevas implementaciones de las interfaces DAO. De hecho, como demostración fue desarrollada la clase MariaDBConnection que implementa la conexión inicial con la base de datos (interfaz *iDatabase*).

Ahora bien, hemos definido una serie de interfaces para implementar el patrón DAO, y hemos desarrollado implementaciones de dichas interfaces para conectarse con una base de datos MySQL. En un caso de uso normal solamente vamos a usar un tipo de base de datos, es decir que en una misma ejecución usaremos un conjunto de implementaciones concretas de diversas interfaces. Este es el caso de uso ideal del patrón Abstract Factory, el cual lo hemos incorporado a nuestro proyecto con la interfaz *iDatabaseConnectionFactory* y su implementación concreta *MySQLAccessFactory*. Esta fábrica abstracta nos da acceso a los DAOs de las diversas entidades.

Abstract Factory no es el único patrón creacional que hemos empleado, pues también hemos visto conveniente el uso del patrón Factory convencional. En este caso el patrón se ha usado dentro del paquete Datos. Tal y como vimos en el diagrama de clases que planteamos en la primera entrega, cada clase de objetos Java cuenta con una interfaz, una clase base que la implementa y una serie de clases hijas que heredan y amplían la funcionalidad de la clase base. La estructura es perfecta para usar este patrón de diseño y de esta manera ocultar o al menos desacoplar la creación de un objeto de su uso.

Cabe destacar que no hemos aplicado el patrón factory en todas las clase de datos, solo en Evento y en Ticket. No obstante, es sencillo entender el origen de esta decisión de diseño. En el caso de Usuario, vimos redundante crear una fábrica pues solamente contábamos con una clase base y una clase hija. En el caso de Entrada, si bien sigue el mismo patrón de interfaz, clase base y 3 clases hijas, podemos apreciar en el código que no existen diferencias entre estas clases. Inicialmente se crearon por consistencia con el resto de entidades y para no generar una deuda técnica en caso de encontrar un uso a la distinción de los distintos tipos de entrada, pero al final del desarrollo podemos afirmar que se trata de un grado de complejidad que no nos ha proporcionado ninguna funcionalidad adicional pero que hemos decidido mantener por si resultase útil en un futuro más lejano.

Junto a todo lo anterior, es importante destacar cómo Evento y Ticket son los dos tipos de objetos más generados, por tanto el uso de la fábrica se ve incluso más justificado aún. Estas fábricas se usan principalmente en las clases DAO, donde se generan grandes cantidades de objetos Java a partir de los resultados de las consultas. Cabría esperar que

en los handlers ocurriese un fenómeno semejante, que se creasen muchos eventos Java a partir de las peticiones del frontend y por tanto se usasen las fábricas. Curiosamente, este no es el caso debido a las capacidades de deserialización de json que nos ofrece el paquete Jackson. Podemos pedirle a este paquete que nos convierta un json en un objeto Java directamente, por lo que la creación de objetos desde ese lado no hace uso del patrón factory, o al menos no en la mayoría de los casos.

Finalmente llegamos al último patrón que hemos usado extensiva y conscientemente en el desarrollo de nuestra aplicación: el patrón singleton. El patrón singleton nos permite crear clases de las cuales solamente exista una única instancia en todo el programa. Este patrón es adecuado para modelar la conexión a la base de datos y los objetos DAO, pues tal y como lo hemos planteado no debería haber más de un acceso simultáneo a la misma base de datos (aunque las bases de datos modernas sean capaces de gestionar este tipo de accesos sin grandes problemas de coherencia y corrupción de almacenamiento).

No solo lo hemos aplicado aquí, sino que la fábrica abstracta de bases de datos también ha sido implementada como un singleton. En otras palabras, tenemos una fábrica singleton que devuelve la instancia de otros singleton. Siguiendo esta filosofía, podríamos haber implementado todas las fábricas del programa como singletons, pues realmente no hay motivos, o al menos no los hemos encontrado, para que exista más de una instancia de cada fábrica y estas no almacenan ningún estado interno por lo que el patrón singleton sería adecuado.

Sin embargo, decidimos en contra de ellos por un par de motivos. En primer lugar, los singleton son globalmente accesibles lo cual podría llegar a ser problemático. Quizás no necesite que todas las clases de mi proyecto puedan fácilmente crear objetos. Por otro lado, los singleton, al usar métodos estáticos, son notablemente difíciles de testear, lo cual nos lleva a la siguiente sección.

Testing

Junto al desarrollo de la aplicación hemos elaborado una suite de tests con los que sistemáticamente comprobar el correcto funcionamiento de nuestra aplicación. Esta suite de testing no comenzó a realizarse hasta la mitad del desarrollo, pues inicialmente todo el equipo de backend estuvo centrado en el desarrollo de un MVP. Por tanto, no podemos decir que hayamos seguido una metodología TDD (Test Driven Development), mas eso no nos ha llevado a obviar esta parte del desarrollo.

Dada la arquitectura heredada del proyecto, que hace un uso intensivo de métodos estáticos y acceso directo a bases de datos, hemos optado por una estrategia de Testing de Caja Blanca. Para lograr un aislamiento real de las capas (API, Negocio y Datos) sin depender de la infraestructura física, ha sido necesario conocer y manipular la estructura interna de las clases.

Hemos utilizado Mockito y técnicas como Reflexión para inyectar dependencias simuladas (Mocks) en componentes que originalmente no permitían inyección. Esto nos ha permitido

validar no solo los valores de retorno, sino también el correcto flujo de ejecución y el manejo de excepciones internas.

Con esta breve descripción de la suite de tests, espero que se comprenda nuestra reticencia a aplicar el patrón singleton en más lugares a lo ancho de nuestra aplicación. Si bien podría haber una única fábrica de eventos y una única fábrica de tickets, no son necesarias más, la creación de múltiples instancias tampoco suponen un problema a diferencia de en el acceso a la base de datos, y por tanto un patrón singleton no acabaría aportando nada más que complicaciones para el testing.

Concesiones

A lo largo del presente documento se han explicado y justificado multitud de decisiones de diseño cruciales que hemos tomado como equipo de desarrollo para asegurar que el producto final presente un código funcional y a su vez elegante, un código limpio en otras palabras. No obstante, nuestra inexperiencia y falta de tiempo nos ha llevado a cometer una serie de errores o a obviar ciertas cuestiones que deberían ser pulidas si el desarrollo continuase.

En esta sección detallaremos algunas decisiones que podrían haberse tomado para mejorar el código, pero deberíamos empezar por algo que no se trata de una posibilidad de mejora, sino de un error directamente. Tenemos que hablar brevemente de las clases de datos

En la base de datos, podemos ver claramente una relación de entidad débil entre una entrada y un evento, y un ticket y una entrada respectivamente. Ahora bien, cuando ha llegado la hora de plasmar esta relación en las clases de Java, tomamos la ingenua decisión de crear una lista de entradas en los objetos de tipo evento, y una lista de tickets en los objetos de tipo entrada. Este acercamiento es ideal para añadir eventos con sus tipos de entradas que nos envíen desde la API en la base de datos, mas resulta poco adecuado para recuperar la información posteriormente. Podemos tener un ticket, pero no sabemos el tipo de entrada, y estas entradas no saben a qué evento pertenecen. Podemos hacer una secuencia de consultas encadenadas para sacar esta información, pero no somos capaces de almacenarla en una estructura de datos coherente. Necesitaríamos crear un evento con sus entradas, y estas con sus tickets para tener a mano toda la información necesaria (de hecho, esta es la táctica que empleamos en la lógica de negocios del proceso de compra).

A lo largo del desarrollo nos hemos visto obligados a refactorizar multitud de veces precisamente por este motivo, por haber considerado el flujo de información en un sentido (del frontend a la base de datos) pero no en el otro. Sin embargo no descubrimos este fallo tan fundamental en la ingeniería del software hasta una fase tardía del desarrollo y llegados a ese punto no tuvimos otra opción que paliar el problema en lugar de solucionarlo de raíz. Si el proyecto continuase después de esta entrega, realizar un proceso de refactorización para implementar estos atributos y hacer uso de ellos sería la tarea más prioritaria.

Cambiando de tema y avanzando a una cuestión mucho menos crítica, me gustaría mencionar el proceso de creación de objetos. Como hemos dicho anteriormente, hacemos

uso extensivo de fábricas para creación de objetos como pueden ser los tickets. No obstante, un problema que solo se ha hecho visible una vez empezado el desarrollo ha sido la gran cantidad de atributos opcionales de nuestros objetos. Hemos acabado con un código con muchas llamadas distintas a los constructores, pasando distintos conjuntos de argumentos. Las fábricas no simplifican este proceso, sino que solo lo complican pues cada vez que añadimos un nuevo constructor a la clase pues nos damos cuenta de la opcionalidad de un atributo, tenemos que añadirlo también a la fábrica y a sus diversas implementaciones.

Hemos pensado que este problema quizás podría aliviarse algo haciendo uso de otro patrón creacional, el patrón Builder; pero, una vez más, por falta de tiempo, hemos sido incapaces de incorporarlo en nuestro código.

Por otro lado, el acceso a la base de datos, si bien gracias al patrón DAO resulta muy flexible, tiene asociado algunas decisiones poco profesionales. En primer lugar, el programa asume el correcto funcionamiento de la conexión y si esta falla, si bien el programa no se cae, tampoco intenta restablecer la conexión por lo que la única opción es volver a lanzarlo. Más allá, las credenciales de acceso a la base de datos se encuentran en texto plano y siempre usamos las implementaciones de MySQL de los DAOs. Hubiese sido recomendable crear un archivo de configuración donde se almacenen las credenciales y el tipo de base de datos a usar, y que el programa al lanzarse leyese esta información del mismo.

Más allá de los fallos y las decisiones que podríamos haber tomado para mejorar el diseño del proyecto, es necesario mencionar otra de las limitaciones de nuestro backend, las funcionalidades que no han sido del todo incorporadas por falta de tiempo. Aunque hayamos propuesto un sistema CRUD sería deshonesto decir que este está finalizado, pues el funcionamiento de las secciones críticas para nuestro proyecto, es decir, las operaciones relacionadas con eventos y tickets, sí han sido correctamente desarrolladas y testadas para asegurar una cierta madurez del producto. Sin embargo otras funcionalidades, como pueden ser la actualización de la información, si bien han sido desarrolladas y planteadas, no han sido tan rigurosamente probadas.

Además también podemos ver carencias en pequeños detalles como la actualización de la recaudación de un evento al realizar un compra, que no sería difícil de implementar pero se ha ignorado por motivos de tiempo; o el añadir una imagen desde el frontend para ser almacenada en el backend, funcionalidad que no se ha desarrollado pues seguramente requeriría alguna adición a nuestro sistema de comunicaciones entre el frontend y el backend.

Finalmente, y esto más que un fallo es una consideración a futuro, no confío en la escalabilidad del programa. La conexión a la base de datos es única en lugar de usar una pool de conexiones, las transacciones pendientes son almacenadas en un único objeto y por tanto siempre hay que pasar por él tanto para iniciar como para terminar una transacción, y otros muchos pequeños detalles de este estilo. Dudo en la capacidad de gestionar un alto número de peticiones por parte de nuestra API. Aún entendiendo que montar un sistema de alto rendimiento se escapa de los objetivos de la signatura, considero

que quizás el código podría haber implementado una serie de pequeñas modificaciones para aumentar su escalabilidad.