Proyecto Final

Base de Datos

UNLP

Magister en Ingeniería de Software

Profesores:

Bazzocco, Javier

Diclaudio, Federico

Integrantes:

Jurado, Abraham

Murdolo, Juan

Introducción

El proyecto de desarrollo presentado cuenta con la correcta implementación de un sistema de registro de pagos, tarjetas y promociones de distintas entidades bancarias. El mismo cuenta con diversas funcionalidades comunes para cada entidad (CRUD) y además de 10 funcionalidades específicas solicitadas como puntos del trabajo práctico.

Además de eso, en este informe se hace una pequeña investigación y descripción de los puntos claves de las relaciones de persistencia en conjunto con MySQL y MongoDB para denotar las diferencias entre ambas, y justificar las decisiones tomadas durante el desarrollo del proyecto.

Requerimientos

Para poder utilizar la aplicación se recomienda utilizar docker se debe configurar de la siguiente manera

- MySQL
    - Imagen de MySQL de la página oficial de docker.
    - Configurar el usuario “root” con password “arturito”
    - se debe crear la base de datos > CREATE DATABASE tarjetas;
    - el proyecto se encarga de crear sus propias tablas y relaciones para la persistencia
- MongoDB
    - Usuario “root” con password “arturito”
    - Crear la base de datos > use tarjetas
    - Crear un registro para inicializarla > db.user.insert({ name: “root”, age: 23})
    -

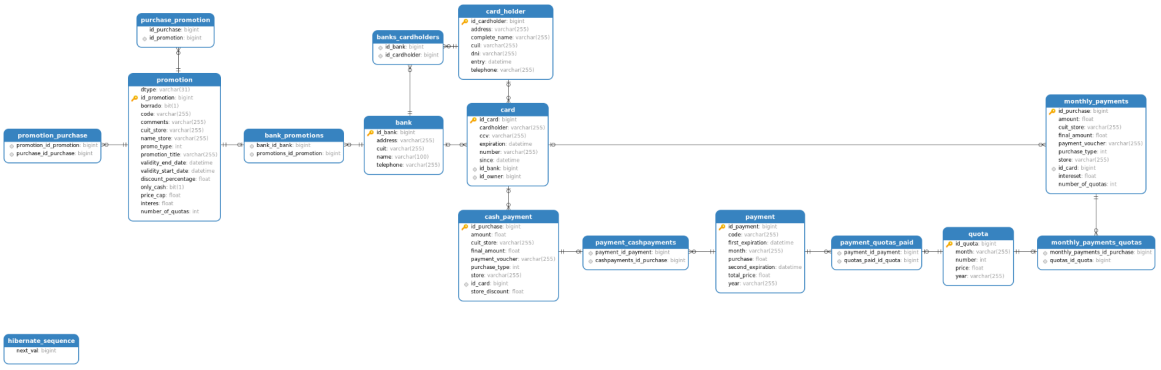
Spring

Para empezar se creó el proyecto utilizando el ***spring initializr*** ya que nos pareció mejor comenzar con un proyecto desde 0 para sentirnos más cómodos con el framework.

Una vez inicializado el proyecto creamos las clases, repositorios, servicios y controladores para ya tener al menos las clases bien definidas, durante el proceso fuimos cambiando tanto las relaciones entre las clases al principio utilizabamos los repositorios simples de spring pero después de pensarlo bastante pasamos a utilizar JpaRepository para MySQL y para mongo MongoRepository

MySQL

Diagrama de la BBDD final



Relaciones bidireccionales

Cuando tenemos relaciones entre objetos, al momento de persistirlos debemos tener en consideración el tipo de dependencia que existe entre ellos. Si tenemos una entidad Persona donde uno de sus atributos es DatosContacto y eliminamos a la Persona, seguramente vamos a eliminar los datos de contacto, ya que la Persona del cual dependen no existe.

Conceptos Claves de las relaciones

Tipos de relaciones

A través de las anotaciones que proporciona JPA cuando se utiliza Hibernate, se pueden gestionar las relaciones entre dos tablas como si de objetos se tratasen. Esto facilita el mapeo de atributos de base de datos con el modelo de objetos de la aplicación. Dependiendo de la lógica de negocio y como se modele, se podrán crear relaciones unidireccionales o bidireccionales.

- @ManyToOne
    - Existe relación Many-To-One entre las entidades donde se hace referencia a una entidad (columna o conjunto de columnas) con valores únicos que contienen de otra entidad (columna o conjunto de columnas). En bases de datos relacionales, estas relaciones se aplican mediante el uso de clave primaria clave externa entre las tablas.

- @OneToMany
    - En esta relación, cada fila de una entidad hace referencia a los muchos registros secundarios en otra entidad. Lo importante es que los registros secundarios no pueden tener varios padres. En una relación uno a varios entre la tabla A y B de la tabla, cada fila en la tabla A puede ser vinculada a una o varias filas en la tabla B.
- @OneToOne
    - En una relación uno-a-uno, un elemento puede vincularse al único otro elemento. Significa que cada fila de una entidad se refiere a una y sólo una fila de otra entidad.
- @ManyToMany
    - Relación de muchos a muchos es donde una o más filas de una entidad se asocian a más de una fila en otra entidad.

Cascada

Las operaciones en cascada con JPA permiten propagar las operaciones que se aplican en una relación entre dos entidades, definiendo el atributo cascade en las relaciones. El uso de cascade nos evita tener que pensar en operaciones previas o posteriores de entidades relacionadas al momento de realizar una operación sobre una entidad. Los valores que puede tomar cascade, son representados en el enum javax.persistence.CascadeType y son los siguientes:

- CascadeType.ALL
    - Propaga todas las operaciones de una entidad, a la entidad con la que se relaciona. Es decir, que si insertamos, actualizamos o eliminamos una entidad, también se aplican estas operaciones a la entidad que se relaciona.
- CascadeType.PERSIST
    - Propaga la persistencia de una entidad a sus entidades relacionadas. Este tipo es útil cuando ambas entidades, la principal y su relación, se crean en el mismo momento y deben ser persistidas
- CascadeType.MERGE
    - Cuando modificamos una entidad y la entidad relacionada, ambas se modifican al mismo tiempo y no se genera un nuevo registro de la entidad relacionada en la base de datos.
- CascadeType.REMOVE
    - Propaga la eliminación de la entidad relacionada a la entidad principal cuando ésta es eliminada. Este tipo de cascada nos sirve cuando una entidad solo existe a causa de la entidad que la contiene. En otras palabras, se recomienda usar solo en relaciones @OneToOne o @OneToMany, ya que las entidades relacionadas existen por la existencia de otra entidad.
    - Pero en el caso de querer usarlo en una relación @ManyToOne o @ManyToMany, la entidad relacionada puede pertenecer a más de una instancia de la entidad que la contiene.
- CascadeType.REFRESH
    - Cuando utilizamos el método refresh del EntityManager, los objetos que se encuentran en el entorno de persistencia del EntityManager se vuelven a cargar desde la base de datos, inclusive si alguno de estos sufrió alguna

modificaciones antes de ser persistido. Este tipo no suele ser de uso común, pero nos puede ser útil para que el contexto de persistencia mantenga las versiones más actualizadas de los objetos.

- CascadeType.DETACH
    - Al igual que REFRESH, suele ser poco utilizado, pero es util para sacar a un objeto del contexto de persistencia de forma manual, y de esta manera podemos cancelar cualquier cambio del objeto antes de que sea persistido en la base de datos.

En el caso de nuestra clase Bank la clase bank tiene dos relaciones una ManyToMany con cardHolders y otra OneToMany con promotions.

En el caso de los usuarios que poseen tarjetas la relación tiene sentido ya que cualquier usuario puede tener una o más tarjetas con el banco y a su vez los cardHolders pueden tener tarjetas de diversos bancos, la relación Many to Many tiene que ver con esta forma de verlo, y utilizamos la CascadeType.PERSIST y CascadeType.MERGE por lo tanto cuando se genera un nuevo card holder, automáticamente se genera su relación con el banco y al mismo tiempo si cualquiera de los dos se modifica el impacto alcanzaría a ambos lados de la persistencia.

En el caso de la relación con promociones, utilizamos el tipo all ya que en caso de que deje de existir un banco la promoción claramente dejaría de existir en consecuencia.

Fetchtype

Cuando tenemos relaciones entre objetos, los ORM al momento de traer a memoria un objeto también van a traer todos los objetos que se relacionan con él, y esto puede ser una ventaja o una desventaja dependiendo el contexto y del momento en que se haga la obtención de estos objetos.

Los dos tipos de carga que existen en JPA son los llamados Eager y Lazy:

- Eager: La carga de los objetos de la relación se produce en el mismo momento.
- Lazy: La carga de los objetos de la relación se producen a demanda, es decir, cuando un cliente los solicita.

Por defecto JPA usa EAGER para la relación @OneToOne, pero en relaciones como @OneToMany y @ManyToMany, el tipo que utiliza por defecto es LAZY ya que del otro lado de la relación hay una N cantidad de objetos a traer desde la base de datos, y eso puede disminuir el rendimiento.

El tipo LAZY conlleva un menor uso de memoria debido a que no obtiene el total de los objetos de la relación en el mismo momento, por lo que a su vez su tiempo de carga va a ser menor. Como desventaja, al diferir la obtención de los objetos si no se maneja con su debido cuidado puede llevar a nuestro código a alguna excepción durante momentos no deseados.

En cuanto al tipo EAGER, asegura tener todos los objetos de la relación, por lo que es menos probable encontrarse con algún tipo de excepción provocada por el mappeo. Como desventaja, hay un mayor tiempo de carga de los objetos en memoria y pueden obtenerse demasiados objetos innecesarios que podrían afectar el rendimiento de la aplicación.

@JsonIgnore

Se utiliza en un campo para marcar una propiedad o una lista de propiedades que se ignorarán a la hora de retornar una respuesta Json desde la base de datos.

@JsonTypeInfo

Se utiliza para indicar detalles del tipo de información que se incluirá en la serialización y deserialización

JsonTypeInfo.As

Especifica el mecanismo a usar para incluir metadatos de tipo (si los hay; para JsonTypeInfo.Id.NONE no se incluye nada); se usa al serializar y se espera al deserializar.

JsonTypeInfo.Id use

Especifica el tipo de metadatos de tipo que se utilizará al serializar información de tipo para instancias de tipo anotado y sus subtipos; así como lo que se espera durante la deserialización.

JsonTypeInfo.As include

Especifica el mecanismo a usar para incluir metadatos de tipo (si los hay; para JsonTypeInfo.Id.NONE no se incluye nada); se usa al serializar y se espera al deserializar.
Tenga en cuenta que para el tipo de metadatos de tipo JsonTypeInfo.Id.CUSTOM, esta configuración puede no tener ningún efecto.

property

Nombres de propiedad usados cuando se usa el método de inclusión de tipo (JsonTypeInfo.As.PROPERTY) (o posiblemente cuando se usan metadatos de tipo JsonTypeInfo.Id.CUSTOM). Si POJO (Plain Old Java Object) tiene una propiedad con el mismo nombre, el valor de la propiedad se establecerá con metadatos de identificación de tipo: si no existe tal propiedad, la identificación de tipo solo se usa para determinar el tipo real.

El nombre de propiedad predeterminado no se define explícitamente (o se establece en una cadena vacía) se basa en el tipo de metadatos que se va a usar.

visible

Propiedad que define si el valor del identificador de tipo se pasará como parte de la secuencia JSON al deserializador (verdadero) o si TypeDeserializer lo manejará y eliminará (falso). La propiedad no tiene ningún efecto sobre la serialización.

El valor predeterminado es falso, lo que significa que Jackson maneja y elimina el identificador de tipo del contenido JSON que se pasa a JsonDeserializer.

@JsonSubTypes

Es una anotación que pertenece al “Jackson Polymorphic Type Handling Annotations”

Se utiliza en para indicar subtipos de tipos anotados.

Uso de relaciones en el proyecto

Bank

Bank tiene distintos tipos de relaciones con CardHolders tiene una relación muchos a muchos porque varios CardHolders pueden pertenecer a 1 o muchos bancos y al mismo tiempo 1 banco posee muchos CardHolders haciendo que esa relación sea ManyToMany con el CascadeType.PERSIST Teniendo en cuenta que su relación es inmediata a la hora de la creación de un CardHolder este ya tiene que tener su banco asociado y también porque ambas entidades deben ser persistidas. y el CascadeType.MERGE siendo este tipo de cascada se utilizó principalmente porque los cambios que se generen en los datos de cualquiera de las estructuras deben impactar de manera inmediata en ambas.

Ya que la relación es muchos a muchos se crea una JoinTable llamada banks_cardholders que posee la relación entre los IDs de los objetos de ambas clases.

La otra relación que posee Bank es con las Promotion, cada banco puede tener distintas promociones, así que la relación pasa a ser uno a muchos OneToMany con los atributos CascadeType.ALL ya que se aplican todas las posibles operaciones de cascada y FetchType.LAZY esta fue una decisión de performance, ya que pocas veces se necesita leer la información de una Promotion únicamente desde su banco pero si al momento de crearse una nueva relación en la lista de bancos, por lo tanto la mejor forma de hacerlo fue desde este tipo de FetchType.

Card

Las Card tienen una relación muchos a uno con un banco ya que una tarjeta solo puede pertenecer a un solo banco, pero varias tarjetas diferentes pueden pertenecer a varios bancos diferentes posee una relación ManyToOne con los atributos FetchType.EAGER En este caso utilizamos EAGER porque la relación al ser ManyToOne solo traerá un objeto relacionado por ende no afectará la performance que la carga de objetos se produzca en el mismo momento y teniendo a columna de la tabla Card tendrá el id_bank relacionado a esta tarjeta.

Y además cuenta con una relación directa con CardHolder Siendo esta ManyToOne porque un CardHolder puede tener varias tarjetas, pero una tarjeta solo pertenece a un CardHolder. FetchType.EAGER En este caso utilizamos EAGER porque la relación al ser ManyToOne solo traerá un objeto relacionado por ende no afectará la performance que la carga de objetos se produzca en el mismo momento.

CardHolder

La única relación bidireccional que posee un CardHolder es su relación con Bank que ya se encuentra definida en Bank.

CashPayment y MonthlyPayments

CashPayment y MonthlyPayments son subclases que extienden desde Purchase, lo que encontramos en este punto y como definición es que ambas clases poseen el @Table, siendo en base de datos representadas como tablas separadas a pesar de pertenecer a la misma tabla padre. Esto permite un mayor control en las consultas, y también la gran diferencia de atributos entre ambas clases amerita que sean tablas separadas. Por ejemplo, en el caso de MonthlyPayments esta tienen una relación directa con cuotas, hablaremos de esta relación en el siguiente párrafo, pero esta relación es única para esa clase y CashPayment no la posee lo que generaría cada vez que se genera CashPayment un número importante de columnas vacías, haciendo difícil el seguimiento del resultado de la consulta, por eso nos decantamos por separar las tablas.

La tabla MonthlyPayments tiene una relación con Quota OneToMany las cuotas solo pertenecen a una entrada de MonthlyPayments pero pueden existir 2 o muchas entradas y CascadeType.ALL al crearse una instancia de MonthlyPayments se generarian también las instancias de Quota, por lo tanto este CascadeType es el indicado para el caso.

Discount y Financing

Discount y Financing son subclases que extienden desde Promotion, en este caso y a diferencia del mencionado arriba, ambas clases pertenecen en base de datos a la misma tabla la cual tiene un @DiscriminatorValue en el cual si es “Financing” o si es “Discount” nos permite generar las consultas solo a la tabla promotion y utilizar valor para identificar de que tipo de promoción se trata.

Payment

Payment es la clase que se encarga de los registros de los pagos sus relaciones son con Quota ya que un pago puede generarse en cuotas el cual despues tambien se relaciona con los MonthlyPayments o bien en efectivo teniendo una relación directa con pagos en CashPayment, ambas relaciones son **OneToMany**, se utilizó también la anotación **@JsonIgnore** para que al querer retornar las cuotas se haga desde MonthlyPayments para un mayor control a la hora de mostrar la información.

Utilizamos también el CascadeType Merge y el FetchType Eager a la hora de relacionar los pagos con sus cuotas para generar una relación en la cual los objetos se creen al mismo tiempo y su relación exista sin crear nuevas instancias.

Promotion

Promotion es la tabla que contiene las instancias de Discount y Financing que en la propiedad promoType tendrán el valor correspondiente para ver si la promoción es de un tipo o de otro.

Contiene además la relación OneToMany con la clase Purchase, la cual tiene los atributos Lazy y CascadeType.ALL relacionados. Ya que es una relación uno a muchos ya que una misma promoción puede pertenecer a más de una compra fue la elección obvia.

Purchase

En el caso de purchase, siendo similar al de arriba, vemos otra clase padre que genera un valor diferente correspondiente al tipo de clase hija que tendrá la instancia, siendo la particularidad que en este caso si existirá una tabla por cada clase hija, siendo diferente del caso anterior.

En este caso la clase Purchase cuenta con varias relaciones, primero una relación Muchos a Muchos con la tabla Promotion, generando una JoinTable, siendo la clase Purchase la encargada de definir dicha relación.

La relación con la clase tarjeta, donde vemos que habrá una relación donde muchas compras pueden estar asociadas a una misma tarjeta generando una JoinColumn con el id_card correspondiente a la compra.

Quota

Quota no posee relaciones definidas, ni es clase hija de otra.

MongoDB

Diagrama de la BBDD final

Relaciones bidireccionales

Conceptos Claves de las relaciones

El esquema flexible de MongoDB permite múltiples patrones cuando se trata de modelar relaciones entre entidades. Además, para muchos casos de uso, un modelo de datos no normalizado (almacenamiento de datos relacionados dentro de un solo documento) podría ser la mejor opción, ya que toda la información se mantiene en un solo lugar, por lo que la aplicación requiere menos consultas para obtener todos los datos. Sin embargo, este enfoque también tiene sus desventajas, como la posible duplicación de datos, documentos más grandes y el tamaño máximo del documento.

Tipos de relaciones

DBRefs

DBRef es el elemento nativo de MongoDB para expresar referencias a otros documentos con un formato explícito { $db : …, $ref : …, $id : … } que contiene información sobre la base de datos de destino, la colección y el valor de identificación del elemento de referencia, más adecuado para vincular a documentos distribuidos en diferentes colecciones.

En nuestro caso un ejemplo de este tipo de referencias es entre el Bank y el CardHolder, el traer el banco a su vez trae como referencia directa todos los cardholders de los que dispone el banco. Esto sirve para ahorrar tiempo a la hora de traer información de un solo banco ya que siempre vamos a necesitar saber sus clientes de manera directa, lo mismo sucede entre Quota y MonthlyPayments, al obtener una cuota siempre será útil tener toda la información de los pagos que tiene relacionados, así será más fácil a la hora de buscar datos cruzados.

Manual References (Referencias Manuales)

Las referencias manuales, por otro lado, tienen una estructura más simple (al almacenar sólo la identificación del documento al que se hace referencia), pero, por lo tanto, no son tan flexibles cuando se trata de referencias de colecciones mixtas.

Declarative Manual References (Enlaces declarativos/ Referencias manuales declarativas)

A partir de Spring Data MongoDB 3.3.0, las referencias manuales se pueden expresar de forma declarativa mediante la anotación @DocumentReference.
Esto le dice a la capa de mapeo que extraiga el valor de identificación de la entidad a la que se hace referencia para el almacenamiento, cargando el documento al que se hace referencia en sí mismo al leerlo.

Mediante el uso de enlaces declarativos, ahora se puede preservar la funcionalidad de mapeo mientras se optimiza el almacenamiento. Aún así, hay que tener cuidado al agregar nuevas instancias de una entidad, ya que también deben agregarse al campo de referencia de la otra entidad (por ejemplo agregar manualmente un CardHolder nuevo a la colección Bank) para establecer el enlace.

One-To-Many Style References

Se aplica una anotación @ReadOnlyPorperty adicional a una propiedad. La otra parte requiere actualizar el atributo de búsqueda de la anotación @DocumentReference con una consulta personalizada.

Uso de relaciones en el proyecto

Bank

Bank tiene dos relaciones con cardHolders y Promotion que ambas están definidas como DBRef porque facilita la obtención de datos, si queremos encontrar todos los clientes de un banco sería más sencillo que la relación fuera de esta manera, lo mismo sucede con las promociones ya que sirve para encontrar todas las promociones relacionadas con el banco.

Card

En el caso de Card card tiene una relación solo con el banco al que pertenece, al no ser una colección la referencia manual @DocumentReference nos sirve de sobra.Lo mismo sucede con CardHolder una tarjeta pertenece solo a una persona por lo que la referencia manual cumple su cometido.

CardHolder

En este caso, en ningún punto se aclara que haya que utilizar como requerimiento saber la información de un banco específico desde el id de CardHolder eso hace que con la referencia manual alcance, más allá de que sea una colección.

CashPayment y MonthlyPayments

Con el caso de Monthly Payments un pago mensual siempre va a tener relacionada una colección de cuotas, este dato es de vital importancia a la hora de generar la relación, principalmente al momento de identificar la cantidad de cuotas restantes y los montos, por ende siempre será mejor al momento de traer un Monthly payment obtener todas las cuotas relacionadas a él por ende utilizamos DBRef.

Discount y Financing

No poseen relaciones.

Payment

Payment posee dos relaciones claras, una con Quota y otra con CashPayment, es una relación necesaria ya que en el caso de SQL necesitábamos de ambas para poder obtener la información requerida para los distintos tipos de pagos. En el caso de mongo obtener la información de un pago y si tiene cuotas o es un pago en efectivo es de vital importancia tener la relación completa, esto hace que la información se pueda encontrar repetida pero la hace más fácil de localizar y más fácil de procesar, por lo tantos optamos por qué la referencia sea directa y los objetos de tipo Quota y CashPayment deberán tener los objetos embebidos.

Promotion

En el caso de Promotion también utilizamos DBRef para su relación con Purchase, pero utilizando el tipo de carga lazy, lazy lo que permite es al momento de cargar las referencia de un documento estas no serán completamente cargadas hasta que no sean accedidas

previamente. Esto aumenta la calidad de la performance a la hora de obtener las promociones pero permite tener la información de manera precisa y accesible también.

Purchase

En el caso de Purchase, pasa exactamente lo mismo con Promotion pero utilizando el atributo eager, permitiendo que toda la información de las promociones asociadas a una compra esten al momento de obtener el valor desde la bbdd, tener toda la información de todas las promociones accesibles es algo útil y mejora la calidad de los objetos retornados.
En el caso de tarjeta justificación es similar, al tener una sola tarjeta por pago será ideal que el documento retorna el objeto tarjeta y no solo su referencia, para hacer más accesible los datos a la hora de realizar la consulta se obtendrá igualmente el objeto embebido en su totalidad.

Quota

No posee relaciones

Queries

SQL a HQL

En nuestra primera entrega presentamos nuestro proyecto JPA con SQL nativas, sin embargo, después de la primera reunión, el profesor nos sugirió pasar todas nuestras consultas SQL a HQL, lo cual nos permitiría acelerar nuestra implementación en MongoDB, así lo hicimos.
Una de las cosas que más nos sorprendió fue el uso que se le podía dar a HQL por sobre SQL acá vemos dos consultas que traen el mismo resultado:

SQL

```
SELECT sum(payment.purchase) AS total_importe, bank.id_bank, card.id_card FROM payment INNER JOIN payment_cashpayments ON payment.id_payment = payment_cashpayments.payment_id_payment INNER JOIN cash_payment ON payment_cashpayments.cashpayments_id_purchase = cash_payment.id_purchase INNER JOIN card ON  cash_payment.id_card = card.id_card INNER JOIN bank ON card.id_bank = bank.id_bank GROUP BY bank.id_bank, card.id_card ORDER BY total_importe DESC LIMIT 1
```

HQL

```
SELECT sum(p.purchase) as total, b.id as bank, c.id as card FROM Payment p INNER JOIN p.cashpayment t INNER JOIN t.card c INNER JOIN c.bank b GROUP BY b.id, c.id ORDER BY sum(p.purchase) DESC
```

La simplicidad salta a la vista en la consulta HQL, esta simplicidad nos ayudó al momento de realizar el TP haciendo que salvo que algo se tuviera que hacer de manera específica con consultas nativas nos inclináramos por HQL, esto se ve en la calidad del código obtenido donde las consultas son sencillas, fáciles de leer y sus resultados son los esperados con una performance superior.

HQL a Querys MongoDB

Sin embargo, al hacer las adaptaciones sobre nuestros repositorios para trabajar con MongoDB y después hacer las primeras pruebas de peticiones con Postman no obtuvimos los resultados que esperábamos. Algunas consultas nos devolvían valores nulos y otras varios objetos cuando esperábamos uno.
Decidimos escribir nuestras consultas en formato MongoDB y funcionaron perfectamente. Aquí un ejemplo de una consulta escrita en formato MongoDB con Spring Data:

Query MongoDB

```
@Aggregation(pipeline = {
                        "{$match: { month: ?0}}",
                        "{$lookup: {from : 'cashpayment',localField :'cashpayment.$id'
,foreignField : '_id', as : 'reporter'}}",
                        "{$unwind: '$reporter'}",
                        "{$group: { _id: '', total: {$sum: '$purchase'}}}"})
   List<ArrayList>  totalCashByMonth(@Param("month") String month);
```

Conclusiones

Como trabajo de equipo fue una experiencia nueva trabajar con Java para uno de nosotros y trabajar con Mongo para el otro. Utilizar HQL también fue algo nuevo, no teníamos idea de lo potente que podía llegar a ser y de la cantidad de tiempo que ahorra, permitiendo pasar de MySQL a Mongo de manera rápida (haciendo las adaptaciones correspondientes).
Además de eso, cumplir con los puntos dados resultó algo bastante sencillo, ya que una vez que armamos toda la estructura MySQL pasar a Mongo fue algo que nos tardó como mucho 4 días, eso ya habla de la importancia de utilizar las tecnologías disponibles, no sólo como un ahorro de tiempo sino también el impacto en la performance y la calidad del código desarrollado.

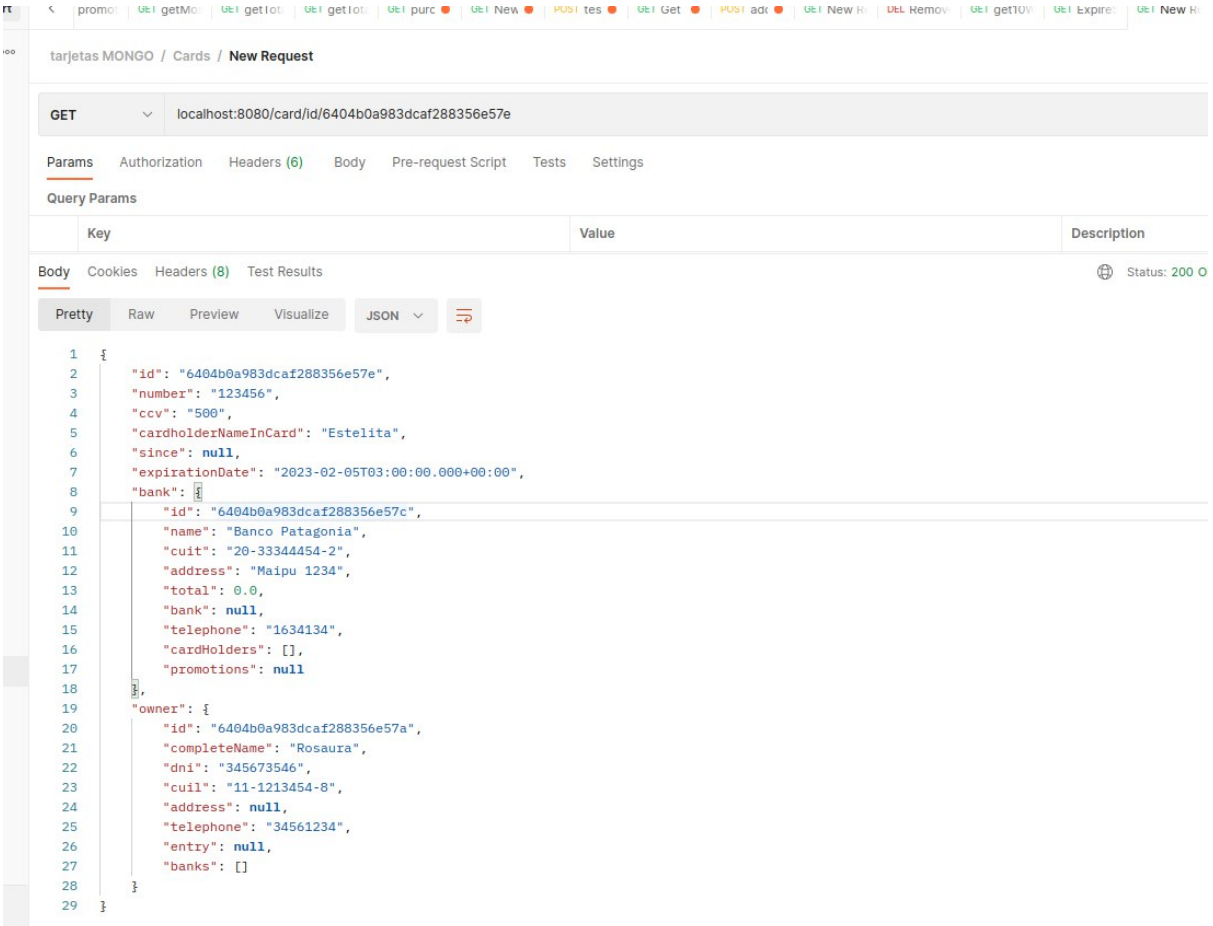
Creemos que ambas bases de datos tienen sus ventajas. Mongo a la hora de tener una mayor cantidad de información disponible al generar una sola consulta, ya que la manera en que se manejan sus referencias hace que sea posible tener objetos enteros embebidos, aunque a veces se repitan datos sigue teniendo un rendimiento superior en casos particulares. A día de hoy, como ya hemos visto en clase, la idea de la materia es utilizar diferentes métodos para obtener y administrar la información, priorizando muchas veces la

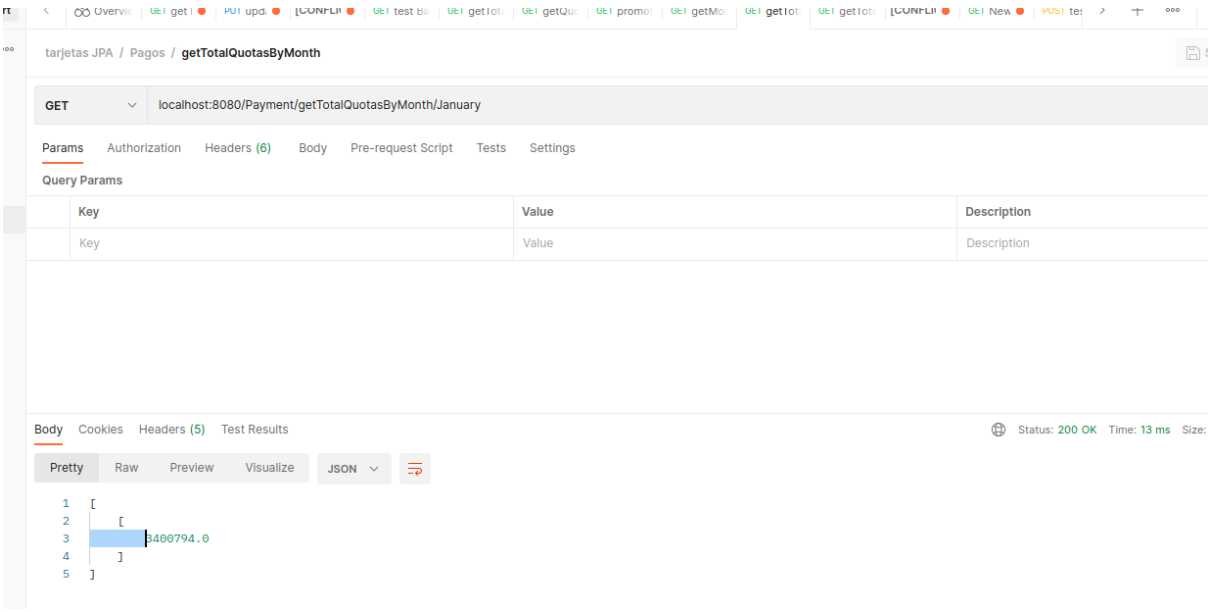
calidad de la misma, la performance a la hora de obtener la información necesaria en el momento indicado, pero por sobre todo, la cantidad de información obtenida por sobre el espacio de almacenamiento, aunque como ya dijimos arriba la misma se puede encontrar repetida algunas veces.

En el caso de MySQL, siendo ya un viejo conocido, la robustez del motor permite una amplia gama de opciones a la hora de relacionar una clase con otra y subsecuentes relaciones, permitiendo así obtener los valores buscados con una repetición nula de información, pero como vimos con Mongo esto impacta en la performance, cuando quisimos lograr una Query que necesita de más de una tabla se notaba que exige de cierto esfuerzo por parte del motor para encontrar la información solicitada en un tiempo prudente y a veces requiere de ciertos cambios en la consulta, al mismo tiempo HQL redujo en gran parte esa exigencia, pero aun así sigue siendo un motor de bbdd extremadamente poderoso si se utilizan las relaciones de manera correcta y se evita la repetición de información.

Testing de ejemplo

Una tarjeta completa con la información del banco asociada y de su propio owner, para tener una mejor lectura del objeto obtenido





Referencias

[1]https://www.tutorialspoint.com/es/jpa/jpa_entity_relationships.htm
[2]https://somospnt.com/blog/240-cascadetype-en-relaciones-con-jpa
[3]https://somospnt.com/blog/243-fetchtype-con-jpa
[4]https://spring.io/blog/2021/11/29/spring-data-mongodb-relation-modelling