Hibernate - Mapping y Relaciones

Introducción

A la hora de plasmar la información del mundo real en nuestros sistemas, podemos partir, en general, de dos enfoques:

- Podríamos comenzar con el **modelo de dominio**: una visión abstracta del mundo real en objetos, cuya misión es capturar la esencia de la información de negocio para el propósito de nuestra aplicación.
- También podríamos contar con un **modelo de datos**, formado por un conjunto de tablas relacionadas (desarrollado a partir de un DER).

Cada modelo se enfoca en ciertos aspectos:

- En nuestras soluciones de objetos generalmente estamos más preocupados en la **lógica de negocio polimórfica**, es decir, en abstracciones, generalizaciones y en el comportamiento de las entidades.
- Cuando construimos un modelo de datos nos preocupamos más acerca de la **estructura y relaciones entre entidades**, las reglas a aplicar para garantizar la integridad de los datos y la lógica empleada para manipularlos.

Como vimos, podemos desarrollar nuestra capa de persistencia, a partir del modelo de dominio, y luego plasmar esto en la BD (enfoque **Top-Down**), o comenzar utilizando un esquema preexistente (legacy) y ascender al modelo de dominio (modelo **Bottom-Up**).

De cualquier forma siempre necesitaremos unir o **"mapear"** las diferencias entre los dos mundos, sea cual fuere el modelo desde el cual partamos para nuestra solución.

A continuación veremos las restricciones que impone Hibernate sobre nuestro modelo a la hora de mapear, y cómo llevar a cabo esto.

Clases

En cuanto a las clases necesarias para desarrollar un modelo de domino, vale aclarar que Hibernate trabaja bien con **POJOs**.

Las pocas restricciones que impone sobre la clase son a su vez consideradas buenas prácticas, por lo que, en general, un POJO cumple con los requisitos para una clase persistente en Hibernate.

Un POJO consiste en:

- Propiedades: En cuanto a Hibernate vale mencionar como única restricción que las propiedades que representen colecciones deben declararse en las clases persistentes, como del tipo **List** o **Set**, y no de una implementación concreta como ArrayList. En tiempo de ejecución Hibernate wrappea las clases concretas, como HashSet, con una instancia de una clase propia de Hibernate, que no es visible a nuestra aplicación.

- Un constructor vacío (Hibernate utiliza **Reflection** para instanciar objetos a través de este constructor).

- Accesors (convención: set-get-is). No hay demasiadas cuestiones a tener en cuenta respecto a este punto. Como única limitación es importante señalar que si trabajamos con colecciones, los getters devuelvan la misma colección que se pasa al setter, ya que **Hibernate compara por identidad las colecciones cuando verifica si nuestros objetos cambiaron, y por igualdad las propiedades**. Este concepto se comprenderá mejor cuando analicemos cachés y runtime.

- Métodos de negocio: Recordemos que nuestras clases persistentes son entidades de negocio que pueden (y deberían) poseer comportamiento. El uso de Hibernate está alineado con ciertas técnicas que proponen un modelo de dominio rico en comportamiento, como **Domain Driven Design (DDD)**.

A su vez una clase persistente no requiere heredar de ninguna clase ni implementar ninguna interfaz en especial, pero puede hacerlo si así se desea, incluso de clases no persistentes.

Metadata

La idea de Hibernate en cuanto a la información de mapeo es centralizar las diferencias entre clases y tablas en un solo lugar.
El Framework nos ofrece dos posibilidades para mapear nuestras clases:

- Escribir archivos **XML**, que serán deployados junto con las clases Java, y leídos por Hibernate en tiempo de ejecución.
- Si usamos la JDK 5.0 o superior, podemos usar **annotations** directamente sobre nuestras clases. Hibernate las lee en el arranque a través de **Reflection**.

En general la segunda opción supera a la primera por lo siguiente:

- Escribir **annotations** sobre nuestras clases nos permite reducir la cantidad de líneas de código, y además la cantidad de archivos necesarios para el mapeo, lo que simplifica tanto el esfuerzo en codificar, mantener, como la probabilidad de cometer errores. Es común que al realizar proyectos complejos la cantidad de componentes intervinientes produzca un exceso de archivos de configuración, los que dificultan el mantenimiento. Por ejemplo, cuando combinamos varios frameworks.

- Las annotations son parte del lenguaje, y **permiten explotar aún más las herramientas de los IDEs** y las detecciones de los lenguajes tipados en tiempo de compilación. Así podemos hacer uso del **refactoring, y autocompletar**, entre otras.

- En el arranque es más rápido para Hibernate leer las annotations que parsear el XML en busca de la información de mapeo asociada a una clase. Esto, sin embargo, no constituye una ventaja significativa.

Por lo tanto, en este apunte optaremos por anotar las clases. Ejemplos análogos se pueden encontrar mediante mapeos con XML, que aún siguen siendo utilizados en varios sistemas.

A la hora de mapear nuestras clases mediante annotations tenemos dos posibilidades:

- Utilizar las annotations de **JPA**.
- Usar las annotations propias de Hibernate.

Al hablar de JPA, estamos refiriéndonos a una especificación: **la especificación de JaveEE para ORMs**, que busca unificar las utilidades que proveen un mapeo objeto-relacional y garantizar la interoperabilidad al cambiar de motor de ORM.
En pocas palabras, JPA no hace nada por sí mismo sin la ayuda de una implementación a las interfaces que provee. Hibernate es compatible con JPA, por lo que al utilizar Hibernate podemos usar interfaces y annotations de JPA o directamente las propias del Hibernate.
En general cualquiera de las opciones son semánticamente equivalentes (a pesar de tener ciertas diferencias sintácticas). Lo importante es tratar de ser coherente en el uso de estas annotations para lograr hacer todo de la misma forma.
La ventaja de JPA es que al amoldarnos al estándar, si el día de mañana queremos cambiar de ORM, por ejemplo, usar TopLink, en lugar de Hibernate, nuestros mapeos sobrevivirán al cambio sin requerir modificaciones adicionales.
Como desventaja, podemos mencionar que, JPA, en su intento por generalizar puede dejar de lado algún feature interesante que se pueda explotar mejor usando directamente el Framework. Por ejemplo, Hibernate provee mayores capacidades para generación de identificadores que quedan fuera de la especificación de JPA.

Una práctica recomendada sería: siempre usar JPA excepto para los casos puntuales que quedan fuera del estándar (como en el caso de las relaciones DELETE_ORPHAN, por ejemplo, los mencionados id generators, o ciertas opciones de configuración de caches sólo disponibles para Hibernate).
Para diferenciarlas en nuestro código, importamos el Package **javax.persistence.*** para annotations de JPA, pero usamos el fully qualified name para las annotations de hibernate.

Relaciones

En general en un modelo relacional podemos encontrar tres tipos de relaciones:

- One to Many / Many to One.
- One to One.
- Many to Many.

One to Many / Many to One

En este tipo de relaciones tenemos por un lado, una entidad que se relacionará con muchas (el extremo **one**) y por el otro, un conjunto de instancias de otra entidad que se relacionará cada una con una instancia como máximo en el otro extremo (el extremo **many**).

Utilizaremos un ejemplo sencillo para comprender cómo funcionan estas relaciones: en principio tendremos un modelo de dominio compuesto por **Item** y **Factura**:

- Una **Factura** podrá tener N ítems. Este es el extremo **One** de la relación.
- Cada **Item** pertenecerá como máximo a una Factura. Es el extremo **Many** de la relación.

*Desde una visión del modelo relacional, el extremo **Many** es el que posee la **ForeignKey** a la otra tabla.*

En nuestro modelo de objetos, inicialmente nuestras clases se verían como:

Factura:

```
public class Factura {

	private Long codigo = null;

	private Set<Item> items = new HashSet<Item>();

}
```

Item:

```
public class Item {

	private Long id = null;

	private Factura factura;

	private Long cantidad;

	private String nombreProducto;

}
```

Los accesors fueron eliminados por simplicidad.

Antes de comenzar a realizar los mapeos sobre nuestras clases deberíamos detenernos a analizar la naturaleza de la relación. No tenemos los datos suficientes del negocio para decidir, pero podemos intuitivamente definir algunos aspectos:

1. ¿En qué dirección agregaré elementos en mi sistema? Lo natural sería ir agregando ítems a la factura, en lugar de agregarle a cada item la factura asociada. Al menos a esto estamos acostumbrados cuando construimos un objeto complejo, como la factura en este dominio.
2. El ciclo de vida del Item está atado al de la factura. Es decir, de no existir la factura, no tendrían sentido los items. Por esta razón sería lógico que, si se elimina una determinada factura, los ítems asociados se eliminen también.
3. ¿Siempre necesitaré traer los ítems asociados a la factura cada vez que traiga una factura o sólo en ciertas oportunidades? Esto también depende del negocio, pero analizaremos las dos situaciones, proporcionando la herramienta que nos brinda Hibernate para abordarlas.

Existen otras cuestiones que iremos planteando a medida que avancemos sobre el dominio, pero por ahora podemos comenzar a abordar los mapeos considerando éstas únicamente. Como vemos, a la hora de utilizar un ORM, surgen decisiones que antes no teníamos que abordar.

Empezamos con Item:

```
@Entity
@Table(name = "ITEM")
```

```
public class Item {

    @Id @GeneratedValue
    @Column(name = "ITEM_ID")
    private Long id = null;

    @ManyToOne
    @JoinColumn(name = "FACT_ID")
    @org.hibernate.annotations.ForeignKey(name="FK_FACT_ID")
    private Factura factura;

    @Column(name = "CANTIDAD")
    private Long cantidad;

    @Column(name = "PROD_NOMBRE")
    private String nombreProducto;
}
```

Este mapeo se denomina “mapeo unidireccional muchos a uno”.
Observaciones:

1) En primer lugar, Hibernate nos permite colocar las annotations sobre nuestras properties, o sobre los accesors. En el primer caso, accederá a nuestros atributos directamente y en el segundo a través de los getters y setters.
La diferencia entre estas opciones puede verse en el siguiente ejemplo:

Supongamos que dos campos de nuestra clase se mapean sobre uno solo de la tabla correspondiente, el típico caso de Persona, con nombre y apellido. Si usamos annotations sobre accesors, podríamos lograr que el setter reciba un String, lo parsee y lo reparta entre los atributos nombre y apellido de mi clase. El getter, por su parte, concatenaría nombre + apellido y lo devolvería a Hibernate para que lo cargue en la columna correspondiente. A su vez, basándonos en los getters logramos un mayor control pudiendo agregar la información que deseemos a la columna correspondiente.

Por defecto, Hibernate accederá a los valores de nuestras propiedades de la forma en que lo indique la annotation @Id. Si ésta se encuentra en una property, lo hará siempre a través de properties. De lo contrario, a través de accesors.

2) `@Entity`: A partir de esta annotation indicamos a Hibernate que nuestra clase es persistente. Se recomienda el uso de la annotation Entity de JPA, la de Hibernate suele traer dificultades. Una vez declarada persistente nuestra clase, todos sus atributos lo serán también por default.

3) `@Table(name = "ITEM")`: A través de esta annotation especificamos un nombre de tabla en caso de diferir al de nuestra clase. De no indicarlo, Hibernate tomará como nombre de la tabla asociada, el mismo que el de la clase. Esta es una convención, y constituye un ejemplo sencillo del concepto **"convention over configuration"**.

4) `@Id`: A través de esta annotation indicamos que la property mapeada será la PK de nuestra tabla. En este caso se trata de una clave simple.

5) `@GeneratedValue`: Especifica la estrategia de generación de Ids. Hay varias opciones para generadores. En nuestro caso, al no acompañarla de parámetros adicionales, su valor será **AUTO**, que decide una estrategia de generación de Ids conveniente dependiendo del motor subyacente. (ej. **Identity**).

6) `@Column(name = "ITEM_ID")`: En este caso la annotation es opcional, si pretendemos que la columna de la tabla correspondiente se llame igual que la property. Aquí valen las mismas consideraciones que en el caso de `@Table`.

7) `@ManyToOne`: Nuestra tabla ITEM tendrá una FK a la tabla FACTURA para establecer la relación, de acuerdo al dominio. Coloquialmente podemos traducir este extremo de la relación como: "un item puede pertenecer a una única factura". Como en el otro lado "una factura puede tener más de un item", marcamos este extremo de la relación (el del item) como "Many" y el opuesto (el de la factura), como "One". Esta annotation puede estar acompañada de ciertos parámetros, como el tipo de **fetching** y el **cascading**, a los que nos referiremos más tarde.

8) `@JoinColumn(name = "FACT_ID")`: Aquí indicamos el nombre de la columna de la tabla a través de la cual estableceremos la relación . También podremos acompañar esta annotation de otros parámetros que especificaremos luego. En caso de no utilizar JoinColumn, Hibernate creará el nombre de la columna concatenando el nombre de la clase del otro extremo, un underscore y el nombre de su id. En nuestro caso sería FACTURA_FACT_ID.

9) `@org.hibernate.annotations.ForeignKey(name="FK_FACT_ID")`: Aquí utilizamos la nomenclatura de Hibernate, ya que es una annotation propia del Framework, no presente en JPA. Especificamos mediante ella el nombre de la FK, para que no nos genere una por default.

Hasta aquí seria suficiente para lograr establecer la relación entre ambas entidades. Sin embargo, en objetos, nos suele interesar recorrer las entidades en forma bidireccional. Más aún en nuestro ejemplo, ya que pedirle un conjunto de ítems a una factura resulta bastante normal.

Para lograr la bidireccionalidad, mapearemos el otro extremo de la relación.

```
@Entity
@Table(name = "FACTURA")
public class Factura {

	@Id @GeneratedValue
	@Column(name = "CODIGO")
	private Long codigo = null;

	@OneToMany
	@JoinColumn(name = "FACT_ID")
	private Set<Item> items = new HashSet<Item>();
}
```

De este mapeo podemos resaltar lo siguiente:

1) Como se observa, utilizamos la interfaz Set para la colección, ya que no podemos definir colecciones a través de clases concretas. Esto, a pesar de significar una restricción en la construcción de las clases, supone una buena práctica, por lo que incluso favorece nuestro desarrollo en lugar de dificultarlo.

2) En este caso, mapeamos el extremo **"one"** de la relación, a pesar de no tener la tabla Factura ninguna correspondencia en una columna con este campo. Aquí vale la pena hacerlo por una de las cuestiones planteadas al comienzo: sería interesante poder recorrer los ítems asociados a una factura, a partir de la misma. Si no fuera así, este extremo podría no representarse en la clase, y por consiguiente tampoco su mapeo.

3) `@OneToMany`: A través de esta annotation explicitamos la relación desde el extremo "one". Puede ir acompañada de parámetros, como los indicadores de fetching y cascade.

Con esto tenemos un mapeo básico de nuestras entidades.
Sin embargo no abordamos aún los puntos planteados al comienzo del mapeo.

El primero de ellos se refería a la dirección por la que me gustaría añadir los Ítems a la factura. Analicemos esta situación a partir del mapeo realizado, sin incluir modificaciones:

```
Factura factura = new Factura();
Item item = new Item();
factura.getItems().add(item);
```

Sería deseable que al dar la orden al Framework de persistir los objetos (más adelante veremos cómo lograr esto), a nuestro Item se le agregue una referencia a la factura (es decir, que la FK de la tabla ITEM apunte a la PK de la tabla FACTURA).

Al parecer, por nuestro negocio, en la mayoría de los casos, insertamos ítems a la factura, en lugar de asignar la factura al Item (parece lo más natural), por lo que sería correcto habilitar esta forma de asignación.

Pero, de acuerdo a los mapeos realizados, ¿qué estrategia lleva Hibernate adelante en estos casos?

Por default, Hibernate entiende que ambos extremos de una relación se sincronizan con la base de datos. Es decir, tanto si agrego un item a una factura, como si le agrego al Item una referencia a la factura, este cambio se verá reflejado en la base de datos llegado el momento que considere oportuno.

Ahora bien, por default Hibernate posee este comportamiento, sin embargo podemos ejercer cierto control sobre el mismo, e incluso lograr una solución genérica que nos ayude en la mayoría de los casos. Para ello definiremos el concepto de **Inverse** en el framework.

Inverse

La palabra nos remite a los mapeos en XML. En ellos, podemos agregar un atributo **inverse** al elemento **OneToMany**, que puede adquirir dos valores posibles:

1) True: A través de este valor decimos que los cambios que se produzcan en la colección del extremo One **no se sincronizarán con la BD**.
Decimos de esta forma, que el exremo one en una relación oneToMany actúa como un "espejo" del extremo opuesto.
De esta forma, y volviendo al ejemplo:

```
factura.getItems().add(item);
```

Sería ignorado por la base de datos a la hora de persistir.

```
Item.setFactura(factura);
```

Sería plasmado en la base de datos, llegado el momento de persistir los cambios.

En annotations, el **inverse** del xml se reemplaza por el parámetro **mappedBy="atributo"**.
En nuestro ejemplo, si no querríamos que al agregar ítems a la factura esto se viera persistido en la clase Factura haríamos:

```
@OneToMany(mappedBy="factura")
```

Donde indicamos que el atributo persistente a tener en cuenta será "factura" en la clase **Item**.

2) False (default): Es el caso que analizamos al comienzo, ya que es el default en Hibernate. Aquí ambos extremos son persistentes, sincronizándose tanto los cambios en la colección, como los cambios al asignar una factura al Item.

Con lo visto podemos lograr 2 cosas:

- Que ambos extremos sean persistentes.
- Que sólo sea persistente el extremo que posee la FK (a través de **mappedBy**).

¿Y si quiero que sólo sea persistente el extremo **One** (el que posee la colección)?

Puedo lograrlo mediante parámetros asociados a la annotation `@JoinColumn` que acompaña a `@ManyToOne`. En nuestro ejemplo, en la clase **Item**:

```
@JoinColumn(name = "FACT_ID", updatable = false, insertable = false)
```

Como vemos, **updatable** e **insertable**, impiden que estas operaciones se realicen desde este extremo, y como por default el extremo **one** es persistente, logramos que sólo éste lo sea.

Práctica recomendada para Inverse

Como mencionamos al principio, muchas buenas prácticas en objetos terminan constituyendo a la vez, buenas prácticas a la hora de utilizar un ORM como Hibernate, por la naturaleza transparente y poco invasiva del Framework.

En particular, al analizar las relaciones, surge la cuestión de las referencias cruzadas, que como sabemos, en objetos, debemos considerar adecuadamente para lograr un modelo sin inconsistencias.

Por otro lado, es considerada una buena práctica en objetos, cuando se trabaja con colecciones, no publicar los mutators de éstas, sino ofrecer una interfaz que permita al cliente añadir y quitar elementos individuales desde el objeto que posee la colección y no hacerlo directamente a través de ésta.
Java.util posee una utilidad llamada **Collections**, a la cual podemos proporcionarle una colección y con el método unmodifiableCollection, unmodifiableList, unmodifiableSet, etc, recibir una colección de sólo lectura, que devolver en caso de necesitar pasarla por medio del getter de nuestra colección.

En pocas palabras, es conveniente no publicar setters de las collections (hacerlos private o protected, o no implementarlos) y si publicamos getters, ofrecer al cliente colecciones de sólo lectura, a menos que nuestro negocio nos obligue a lo contrario.
Esto nos aporta seguridad, y a su vez obedece a la regla de **encapsulamiento** en objetos (ocultamos la representación interna, nadie sabe si tenemos una colección o qué, pero para sus fines es suficiente poder añadir y quitar elementos).
Podríamos entonces, en nuestra clase factura, proporcionar el método "addItem" de esta forma:

```
public void addItem(Item item) {
      if (item == null)
         throw new IllegalArgumentException("El item es nulo");

      if (item.getFactura() != null)
          item.getFactura().removeItem(item);

      item.setFactura(this);
      this.getItems().add(item);
}
```

De esta manera, nos aseguramos que ambos extremos de la relación son consistentes.
Hibernate no posee lo que se denomina Container Managed Relationships (CMRs), por lo que trabajar de esta forma los extremos del grafo es prácticamente obligatorio para el programador. Como se observa en el código anterior, debería proveerse a su vez un método removeItem(); que se encargue de la eliminación de las unidades.

En combinación con esta técnica, el agregar mappedBy al extremo one se sugiere en la mayoría de los casos (excepto aquellos en que no pueda utilizarse, por ejemplo, cuando se hace uso de `@IndexColumn`).
Por lo que en nuestras relaciones uno a muchos, el uso de mappedBy, junto con esta técnica para administrar colecciones resulta prácticamente obligatorio para lograr bidireccionalidad.

Podemos comprobar que, con mappedBy, la única línea de código del método anterior que se sincronizará con la BD es la de color azul. El resto del código procedural empleado, nos permite mantener coherencia a nivel del grafo de objetos. Aquí vemos una diferencia fundamental entre la forma de manejar relaciones en una base de datos que emplea foreign keys, las cuales son medios declarativos de establecer la relación, y la que debemos utilizar en objetos para lograr un resultado análogo, a través de código procedural.

Cascading

Para abordar la segunda cuestión planteada, acerca de la dependencia entre ítems y facturas, repetiremos nuestro ejemplo inicial:

```
Factura factura = new Factura();
Item item = new Item();
factura.getItems().add(item);
DB.save(factura);
```

Para el caso vale aclarar que el ciclo de vida de ambos objetos es completamente independiente. Los nuevos objetos son considerados transitorios, hasta que se persistan explícitamente. Es decir, su relación no posee influencia en su ciclo de vida.

Por esta razón, no podríamos pretender que al asociar un item a una factura, éste se persistiera automáticamente por el simple hecho de hacerlo su entidad padre.
Para lograrlo, deberíamos agregar la línea `DB.save(item);` asegurándonos de esta forma de que nuestro item deje de ser un objeto transitorio.

Sin embargo, para el caso del item, resultaría interesante contar con un mecanismo por el cual el motor de ORM comprendiera que al asociarse una nueva instancia del mismo a una factura, ambas entidades se persistieran, o mejor dicho, que un insert en la base sobre la tabla FACTURA, se propagara sobre la tabla ITEM.

Tenemos entonces, el parámetro **Cascade**, que mencionamos anteriormente, y que acompaña a la declaración de cada extremo de la relación.
Este parámetro es unidireccional, es decir, indica que los cambios se propagan de un lado hacia el otro. Puede declararse en ambos extremos, si tiene sentido hacerlo.
En nuestro caso, para el ejemplo mencionado, tiene sentido indicarlo sobre la factura, que es la que se persistirá explícitamente, y por lo tanto, propagará esta acción:

```
@OneToMany(cascade = { CascadeType.PERSIST, CascadeType.MERGE },
mappedBy = "item")
@JoinColumn(name = "FACT_ID")
private Set<Item> items = new HashSet<Item>();
```

PERSIST y **MERGE**, se refieren a **SAVE** y **UPDATE** respectivamente.
Vale aclarar, que los Cascades son por operación. ALL precisamente indica que todas operaciones posibles de propagarse desde la entidad fuente a la destino, lo hagan.

También contamos para las eliminaciones con `CascadeType.REMOVE`, que puede resultar de gran utilidad si deseamos que, al remover una factura, se eliminen también los ítems asociados (`DB.delete(factura);` en el ejemplo).

Tip:
En el ejemplo utilizamos
`factura.getItems().add(item);`

Para agregar ítems a la factura. Como mencionamos anteriormente esta práctica no es recomendada. Si vemos el resultado de esto, combinado con los cascades mencionados, comprobaremos que Hibernate envía tres sentencias al motor (un insert para la factura, un insert para el item, y un update sobre el item, para hacerlo apuntar a la factura). En caso de agregar la factura al item (item.setFactura(factura)), veremos que sólo dispara dos sentencias (dos inserts, incluyendo el correspondiente al item una referencia a la factura). Como vemos, resulta más performante sincronizar con la DB el extremo de Item y usar ***mappedBy****. Esta es otra razón por la cual continúa siendo conveniente utilizar esa práctica, proporcionando del lado de la factura una interfaz conveniente para el agregado de ítems.*

Una última cuestión respecto al cascadeo, se refiere a la existencia de objetos huérfanos.
Suponiendo que item no depende de otra entidad que de Factura, sería lógico que al remover una referencia a Item desde la colección de facturas, se elimine el registro correspondiente en la DB (como haría el Garbage Collector con nuestros objetos):

```
factura.getItems().remove(item);
```

Sin embargo, Hibernate no tiene forma de darse cuenta que el item ha quedado huérfano a menos que le indiquemos que en este tipo de situaciones ocurriría esto.

Para ello, contamos con una variante de Hibernate (no proporcionada por JPA), que se denomina **delete-orphan**:

```
@org.hibernate.annotations.Cascade(value = org.hibernate.annotations.CascadeType.DELETE_ORPHAN).
```

Fetching

La tercera cuestión a abordar se relaciona con la forma en que traemos nuestros objetos desde la base de datos. A continuación explicaremos brevemente el concepto de fetching, el cual se verá con más profundidad en apartados posteriores.

El primer ejemplo a analizar será:

```
Item item = (Item) session.get(Item.class, id);
```

La pregunta a realizar en este caso sería si al traer un objeto, deseo traer con él aquellas entidades asociadas, o posponer esto hasta el momento en que realmente vaya a utilizarlas.

Tenemos dos posibilidades para el Fetching:

Lazy: En este caso, al traer un item, no nos traerá aún la factura asociada. Para ello emplea el **pattern Proxy** y la librería **CGLIB (Code Generation Library)**, y hace apuntar a una instancia que crea en tiempo de ejecución, y que permitirá que la carga real del objeto se realice recién cuando se necesite. Para ello, al traer el item se realizará un SELECT y otro cuando se desee obtener la información de la factura.

Eager: Con esta opción, se realizará un Join entre la tabla ITEM y FACTURA, para permitir cargar ambos objetos en el momento de obtener el item desde la Base de datos.

En nuestros mapeos podemos definir esta opción de la forma siguiente:

```
@OneToMany(fetch = FetchType.EAGER/LAZY).
```

Por default, el fetching es **LAZY** en Hibernate para colecciones y **EAGER** en casos como este, en que tenemos una asociación con una clase simple.

Esto puede llevara al problema conocido como **“N + 1 Selects”**, el cual veremos mejor con un ejemplo.
Supongamos que quiero traer todos los ítems asociados a la factura “001”. Si mi estrategia de Fetching es Lazy, Hibernate realizará lo siguiente:

a) Primero traerá la factura sin Items:

SELECT * FROM FACTURA WHERE CODIGO = ‘001’

b) Luego los Items asociados a la factura:

SELECT * FROM ITEM WHERE FACT_ID = ‘001’

Si, por el contrario, tenemos seteada nuestra estrategia a **EAGER**, en lugar de realizar dos selects separados, llevará a cabo un JOIN entre las dos tablas y rellenará los objetos

con los campos resultantes del JOIN (como desventaja, cada resultado de la consulta tendrá campos repetidos, lo que introduce cierta redundancia).

Imaginemos el caso de traer más de una factura y sus campos asociados. En esa situación, tendremos 1 query para tener todas las facturas y N queries (1 por cada factura) para obtener los ítems asociados a cada una. De allí el nombre de N + 1.

Por lo tanto, hay que ser cuidadoso a la hora de seleccionar una estrategia de Fetching, de acuerdo al dominio y al problema que querramos resolver.
Vale aclarar que Hibernate limita la cantidad de colecciones maracadas con EAGER a una como máximo dentro de una clase.

Existe una forma dinámica de realizar el fetching, independientemente del mapeo globlal que realizamos de la manera expuesta. Esto se lleva a cabo a través de un lenguaje de consultas llamado HQL, que veremos luego.

One-To-One

Se trata de una simple relación uno a uno, donde una tabla posee una FK a otra, siendo este campo marcado con una restricción Unique.
Supongamos que tenemos una nueva clase **Cliente**, la cual posee una referencia a una **Dirección**, representada por una tabla independiente. (Consideremos que cada empresa cliente podrá tener una dirección y que cada dirección podrá corresponder sólo a una empresa).
De la forma siguiente podríamos llevar a cabo la relación:

```
@Entity
@Table(name = "CLIENTE")
public class Cliente {
	…
	@OneToOne
	@JoinColumn(name = "DIR_ID")
	private Direccion direccion;
	…
}

@Entity
@Table(name = "DIRECCION")
public class Direccion {
	…
	@OneToOne(mappedBy="direccion")
	private Cliente cliente;
	…
}
```

En el caso mencionado, bien podría suceder que la información de dirección se encontrara en la misma tabla de usuario. De esta manera se evitaría hacer un Join entre las dos tablas cada vez que se desea obtener la información relacionada a direcciones.
Sin embargo, en objetos puede resultar conveniente que la dirección aparezca separada de la clase Cliente, para poder pasarla como parámetro, por ejemplo. Estamos ante un problema de **granularidad**: queremos que nuestro sistema tenga más clases que tablas, porque nos resulta cómodo a la hora de programar. Hibernate aborda esta cuestión, permitiéndonos que varias clases referencien en conjunto a una sola tabla:

```
@Entity
@Table(name = "CLIENTE")
public class Cliente {
	…
```

```
        @Embedded
        private Direccion direccion;
        …
}

@Embeddable
public class Direccion {
        …
        @Column
        private String calle;
        @Column
        private Long numero;
        …
}
```

La tabla resultante, se llamará **Cliente** y tendrá tanto las columnas presentes en la clase Cliente, como aquellas que se encuentran en la clase Direccion.
El concepto mencionado responde a la denominación de **Value Types** en ORMs. Existe una clasificación en torno a las clases persistentes que se ajusta a esta situación:

- Entidades: Son clases cuyas instancias poseen su propia identidad persistente. Las entidades poseen su propio ciclo de vida. En nuestro ejemplo, Cliente es una entidad.
- Value Types: Son clases que no definen ningún tipo de identidad persistente. El ciclo de vida de un value type, está atado al de la entidad a la que pertenece. En nuestro ejemplo, Direccion es un value type.

A la hora de persitir entidades, por ejemplo, tendrá sentido hacer session.save(cliente), por ser una entidad persitente, pero nunca será posible hacer session.save(direccion), ya que es un value type, dependiente de cliente. Por otro lado, es válido ir desde un value type a una entidad (tener una asociación a ésta), pero no podremos desde esta entidad tener otra referencia al value type, ya que ***las asociaciones siempre apuntan a entidades****.*

Many-To-Many

El paradigma relacional no soporta relaciones muchos a muchos. Para lograr este efecto se utiliza una tabla intermedia que reúne las primary Keys de las tablas a relacionar, junto con la información propia de la relación.

En objetos no necesitamos de esta entidad intermedia, ya que propio paradigma soporta este tipo de relaciones. Sin embargo, podríamos mapear esa clase intermedia a través de dos relaciones **many-to-one**, de la forma ya vista, en caso de tener sentido para nuestro sistema mapear la entidad que relaciona a las otras dos (por ejemplo, en el caso de que ésta incluya información adicional, o represente una abstracción útil).

Si decidimos ignorar la tabla intermedia tendremos una verdadera relación many-to-many que mapear.

Supongamos que en nuestro negocio tenemos ahora dos clases más. Un **Producto**, asociado al item en una relación **one-to-many,** y a una clase **Categoria**, en una relación many-to-many con la clase Producto. Por ejemplo, una bebida cola, es una gaseosa pero también, en términos más generales, una bebida. A su vez, una bebida representa también a otros productos.

Si deseáramos una relación unidireccional, por ejemplo, a partir de categoría obtener los ítems asociados a la misma, bastaría con:

```
@Entity
@Table(name = "CATEGORIA")
public class Categoria {

	…
	@ManyToMany
	@JoinTable(name = "CATEGORIA_ITEM",
	joinColumns = {@JoinColumn(name = "CAT_ID")},
	inverseJoinColumns = {@JoinColumn(name = "ITEM_ID")}
	)
	private Set<Item> items = new HashSet<Item>();
	…
}
```

Si quisiéramos una relación bidireccional, como siempre, debemos asegurarnos de mantener la consistencia entre los dos extremos.

El mapeo para item sería entonces:

```
@ManyToMany(mappedBy = "items")
private Set<Categoria> categorias = new HashSet<Categoria>();
```

De esta forma, estamos ignorando la tabla intermedia y no tendremos problemas a la hora de guardar o recuperar los datos.
Sin embargo, **en general es recomendable mapear las tablas intermedias también**, ya que a medida que el negocio evoluciona, estas entidades suelen cobrar sentido.