Capas de Persistencia

Laboratorio IV, Claudio Zamoszczyk

Universidad de Palermo, 2012

Integrantes

Andrés Cabrera
Ignacio Gallardo
Oskarina González
Felipe Jimenez

Introducción a la persistencia

¿Que es la Persistencia?

La persistencia es la parte más crítica en una aplicación de software. Si la aplicación está diseñada bajo la orientación a objetos, la persistencia se logra por: **serialización del objeto** ó, **almacenando en una base de datos**.

> *"Persistencia es la habilidad que tiene un objeto de sobrevivir al ciclo de vida del proceso en el que reside. Los objetos que mueren al final de un proceso se llaman transitorios."*

El Modelo Relacional

Una base de datos relacional es una base de datos que cumple con el modelo relacional, el cual es el modelo más utilizado en la actualidad para implementar bases de datos ya planificadas. Permite establecer interconexiones (relaciones) entre los datos (que están guardados en tablas), y a través de dichas conexiones relacionar los datos de ambas tablas, de ahí proviene su nombre: "Modelo Relacional".

Características

- Una base de datos relacional se compone de varias tablas o relaciones.
- No pueden existir dos tablas con el mismo nombre ni registro.
- Cada tabla es a su vez un conjunto de registros (filas y columnas).
- La relación entre una tabla padre y un hijo se lleva a cabo por medio de las claves primarias y ajenas (o foráneas).
- Las claves primarias son la clave principal de un registro dentro de una tabla y éstas deben cumplir con la integridad de datos.
- Las claves ajenas se colocan en la tabla hija, contienen el mismo valor que la clave primaria del registro padre; por medio de éstas se hacen las relaciones

Ventajas

El uso de una base de datos relacional bien diseñada puede reducir mucho la cantidad de datos que debe ingresar cada vez que agrega un registro. Para un número grande de registros, una base de datos relacional puede buscar más rápido entre los registros.

La forma de trabajar con los datos persistentes en el modelo relacional es seleccionando los datos que queremos que persistan en el tiempo y grabandolos de manera explícita mediante consultas de alta/modificación de SQL, previa transformación de los datos. Los objetos trabajan de otra forma. Dependiendo de la implementación particular puede ser que haya clases persistentes, cuyos objetos siempre se almacenan en disco, marcar especiales para los objetos que permitan discriminar cuáles se almacenarán, y otras técnicas.

Desventajas

Las bases de datos convencionales fueron diseñadas para manejar tipos de datos alfanuméricos y por esto **difícilmente pueden manipular objetos y métodos** (los métodos son los comportamientos definidos de los objetos).

El auge de la Orientación a Objetos

Las opciones que se basan en imponer un único modelo teórico a toda la aplicación padecen de graves inconvenientes. En el caso de que toda la aplicación siga el modelo relacional, perdemos las ventajas de la orientación a objetos. En el caso de que toda la aplicación siga el modelo orientado a objetos, en general tenemos el problema que las bases de datos que

debemos usar están inmaduras y tienen un bajo nivel de estandarización.
Su limitación suele residir en su especialización, ya que suelen estar diseñadas para un tipo particular de objetos (por ejemplo, una base de datos para un programa de CAD).
Sin embargo en algunos escenarios específicos las bases de datos orientadas a objetos pueden aportan algunas ventajas que vale la pena mencionar:

- Manipulan datos complejos de forma rápida.
- Aporta flexibilidad y elimina por completo la necesidad de herramientas ORM's (impedancia), con su consecuente mejora en desempeño.
- Permiten que múltiples usuarios compartan objetos complejos y los manipulen en un ambiente seguro y estructurado.

Examinemos ahora la opción de que el programa sea orientado a objetos y la base de datos sea relacional, lo que, en principio, constituye opción más natural. Sin embargo, plantea el problema de cómo hacemos que dos componentes con formatos de datos muy diferentes puedan comunicarse y trabajar conjuntamente. Siguiendo la metáfora anterior, se trata de hacer que dos personas que hablan idiomas diferentes se comprendan.
La solución es la misma que se daría en la vida real.
Se debe encontrar un traductor que sepa traducir de cada idioma al otro. De esta forma, las dos personas se entenderán sin necesidad de que uno hable el idioma del otro. En el mundo de la programación este traductor no es más que un componente de software (concretamente, una capa de programación, dentro de una capa general de persistencia), al que se le dan los nombres de "capa de persistencia", "capa de datos", "correspondencia O/R" ("ORM mapping") o "motor de persistencia".

En nuestro caso, el motor de persistencia traduce entre los dos formatos de datos: de registros a objetos y de objetos a registros. Cuando el programa quiere grabar un objeto llama al motor de persistencia, este traduce el objeto a registros y llama a la base de datos para que guarde estos registros. De la misma manera, cuando el programa quiere recuperar un objeto, la base de datos recupera los registros correspondientes, los cuales son traducidos en formato de objeto por el motor de persistencia.

El programa sólo ve que puede guardar objetos y recuperar objetos, como si estuviera programado para una base de datos orientada a objetos. La base de datos sólo ve que guarda registros y recupera registros, como si el programa estuviera dirigiéndose a ella de forma relacional. Es decir, cada uno de los dos componentes trabaja con el formato de datos (el "idioma") que le resulta más natural y es el motor de persistencia el que actúa de traductor entre los dos modelos, permitiendo que los dos componentes se comuniquen y trabajen conjuntamente.

Ventajas

Esta solución goza de las mejores ventajas de los dos modelos.

- Por una parte, podemos programar con orientación a objetos, aprovechando las ventajas de flexibilidad, mantenimiento y reusabilidad.
- Por otra parte, podemos usar una base de datos relacional, aprovechándonos de su madurez y su estandarización así como de las herramientas relacionales que hay para ella.

Se calcula que un motor de persistencia puede reducir el código de una aplicación en un 40%, haciéndola menos costosa de desarrollar. Además, el código que se obtiene programando de esta manera es más limpio y sencillo y, por lo tanto, más fácil de mantener y más robusto. Por añadidura, el motor de persistencia no sólo simplifica la programación, sino que permite hacer ciertas optimizaciones de rendimiento que serían difíciles de programar por nosotros mismos.

Como conclusión y de forma general, ésta es la mejor opción en la actualidad para implementar una aplicación de software.

Imaginemos que queremos cargar en memoria un objeto persistido en la base de datos, los pasos a seguir serían: abrir la conexión a la base de datos, crear una sentencia sql parametrizada, llenar los parámetros (por ejemplo la clave primaria), recién allí ejecutarlo como una transacción y, cerrar la conexión a la base de datos.

Con un framework, la tarea se reduce a: abrir una sesión con la base de datos, especificar el tipo de objeto que queremos (y su clave primaria correspondiente), cerrar la sesión.

A continuación nos encargaremos de repasar las responsabilidades que tiene la capa de persistencia, para más adelante profundizar en las características y ventajas que aportan las herramientas y frameworks de mapeo.

Características Generales de la Capa de Persistencia

Identidad

En la siguiente porción de código veremos algunos aspectos referentes a la identidad de los objetos.

```
Persona rodrigoBueno = new Persona(”Bueno”, “Rodrigo”);
Persona ramonBueno = new Persona(”Bueno”, “Ramon”);
Persona bueno = ORM.LoadByApellido(“Bueno”);
```

Hemos creado dos instancias en memoria de la clase Persona. O sea que para el objeto tenemos dos objetos diferentes usando el mismo apellido. Si el apellido es la clave primaria en la base de datos, tendríamos problemas cuando la aplicación trate de escribir la segunda instancia en la base de datos. Por eso debemos tener cuidado con la identidad del objeto y sus diferentes notaciones en la base de datos y en el programa. La solución para esto es aplicar la identidad en la misma caché.
Imaginemos un código como el siguiente:

```
Persona bueno1 = (Persona) ORM.Load(“IDPersona”, 100);
Persona bueno2 = (Persona) ORM.Load(“IDPersona”, 100);
```

En la porción de código anterior tenemos dos variables que deberían hacer referencia al mismo objeto: Rodrigo Bueno. Si la capa de acceso no es buena, habrá cargado dos instancias del objeto en memoria de los mismos datos en la base de datos. Para evitar que suceda, necesitamos tener un mecanismo que verifique que el objeto ya está cargado en memoria desde la base de datos.
La primera vez que cargamos el objeto, la capa de acceso lo lee desde la base de datos; la segunda vez, el método Load necesita verificar si la Persona con OID igual a 100 ya está cargada en memoria. Si es así, solo retorna la referencia al objeto en memoria creado por la primera llamada.

Caché

En la mayoría de las aplicaciones, se aplica la regla del 80-20 en cuanto al acceso a datos, el 80% de accesos de lectura accede al 20% de los datos de la aplicación. Esto significa que hay un un conjunto de datos dinámicos que son relevantes a todos los usuarios del sistema, y por lo tanto accedido con más frecuencia. Las aplicaciones empresariales de sincronización de caché normalmente necesitan escalarse para manejar grandes cargas transaccionales, así múltiples instancias pueden procesar simultáneamente. Es un problema serio para el acceso a datos desde la aplicación, especialmente cuando los datos involucrados necesitan actualizarse dinámicamente a través de esas instancias. Para asegurar la integridad de datos, la base de datos comúnmente juega el rol de árbitro para todos los datos de la aplicación. Es un rol muy importante dado que los datos representan la proporción de valor más significante de una organización. Desafortunadamente, este rol también no está fácilmente distribuido sin introducir problemas significativos, especialmente en un entorno transaccional.

Es común para la base de datos usar replicación para lograr datos sincronizados, pero comúnmente ofrece una copia offline del estado de los datos más que una instancia secundaria activa. Es posible usar base de datos que puedan soportar múltiples instancias activas, pero se pueden volver caras en cuanto a performance y escalabilidad, debido a que introducen el bloqueo de objetos y la latencia de distribución. La mayoría de los sistemas usan una única base de datos activa, con múltiples servidores conectada directamente a ella, soportando un número variables de clientes.
En esta arquitectura, la carga en la base de datos incrementará linealmente con el número de instancias de la aplicación en uso, a menos que se emplee alguna caché. Pero implementando un mecanismo de caché en esta arquitectura puede traer muchos problemas, incluso corrupción en los datos, porque la caché en el servidor uno no sabrá sobre los cambios en el servidor dos.

Carga en Memoria

Uno de los problemas que debemos decidir luego del mapeo es si siempre se cargan todos los objetos relacionados a uno principal. La respuesta más probable es no, porque las redes de relaciones tienden a ser más compleja y la cadena de relación tienden a ser más larga en la vida real. Imaginemos que por defecto se carga cada relación de un objeto. En el caso de una gran base de datos se volverá grandísimo y habrá pérdida de performance.
La solución para este problema es conocida como “Carga Retardada” (*Lazy Load*) de las relaciones. Otro uso común de cargas retardada es la generación de reporte y objetos que se dan como resultados de una búsqueda, casos en los cuales se necesita sólo un subconjunto de datos del objeto.
Lo mismo sucede para aquellos campos grandes y menos usados. Por ejemplo si se almacena la foto de una Persona ocupará alrededor de 100k mientras que el resto de los atributos no llegan, en total, a 1k; y, raramente son accedidas.
Pero también, habrá veces en la que la “Carga Directa” de las relaciones se prefiera. Con ella, cada vez que se cargue un objeto, querríamos tener algunas de sus relaciones cargadas sin necesidad de volver a consultar la base de datos.

Transacción

Los datos almacenados en una base de datos necesitan ser protegidos por una transacción. Esto permite múltiples inserciones, modificaciones y borrados con la seguridad de que todo o se ejecuta o falla, como si fuera una sola entidad coherente. Las transacciones también pueden ofrecer protección de concurrencia; el bloqueo pesimista de tuplas mientras los usuarios están trabajando en ellos, evita que otros usuarios comiencen con sus cambios.
Sin embargo, el mecanismo de transacción de la base de datos tiene alguna limitaciones:

Cada transacción requiere una sesión separada, para permitir que los usuarios abran ventanas relacionadas a su trabajo requeriría de licencias extras o que las ventas estén limitada a acceso de sólo lectura.
Una alta aislación en la transacción y bloqueo basado en páginas puede evitar que otros usuarios accedan a los datos que deberían estar legítimamente permitidos.
Una alternativa es que los datos sean puestos en un buffer en la capa de persistencia, con todas las escrituras que se harán hasta que el usuario lo confirme. Esta transacción de base de datos se necesita solo durante la operación de escritura en

masa (bulk operation), permitiendo ser compartida entre múltiples ventanas. Esto requiere un esquema de bloqueo optimista donde las tuplas son chequeadas mientras se escriben.
Obviamente, esta escritura en masa corre protegida por la integridad referencial. Esas restricciones especifican los requerimientos lógicos según el caso: la tupla debe existir antes de que se la relacione y, las tuplas relacionadas a otra deben ser borradas antes de que se borre la tupla objetivo. Esto se logra mediante un mecanismo en la base de datos que mantiene correctamente las dependencias entre las tuplas.

Concurrencia

La capa de persistencia debe permitir que múltiples usuarios trabajen en la misma base de datos y proteger los datos de ser escritos equivocadamente. También es importante minimizar las restricciones en su capacidad concurrente para verl y acceder.
La integridad de datos es un riesgo cuando dos sesiones trabajan sobre la misma tupla: la pérdida de alguna actualización está asegurada. También se puede dar el caso, cuando una sesión está leyendo los datos y la otra los está editando: una lectura inconsistente es muy probable.
Hay dos técnicas principales para el problema: bloqueo pesimista y bloqueo optimista. Con el primero, se bloquea todos acceso desde que el usuario empieza a cambiar los datos hasta que COMMIT la transacción. Mientras que en el optimista, el bloqueo se aplica cuando los datos son aplicados y se van verificando mientras los datos son escritos.

Bloqueo optimista

Cuando se detecta un conflicto entre transacciones concurrentes, se cancela alguna de las transacciones. El bloqueo optimista desacopla la capa de persistencia de la necesidad de mantener una transacción pendiente, chequeando los datos mientras se escribe.
Se usan varios esquemas de bloqueo optimista. Difieren en que campos son verificados; a veces se usa un campo de estampa de tiempo (timestamp) o un simple contador (counter).

Bloqueo pesimista

Evita que aparezcan conflictos entres transacciones concurrentes permitiendo acceder a los datos a solo una transacción a la vez.
La aproximación más simple, consiste en tener una transacción abierta para todas la reglas de negocios involucradas. Hay que tener precaución con transacciones largas. Por eso se recomienda, usar múltiples transacciones en las reglas de negocios.

Otros aspectos

En esta sección veremos otros aspectos a tener en cuenta en el diseño de la persistencia.

Referencia circular

Se refiere a si el framework es capaz de detectar cual es el objeto que se está solicitando, sin hacer el roundtrip a la base de datos.

Información oculta

Hay muchas columnas de la tabla que no necesitan ser mapeadas a una propiedad del objeto. Esta columnas contienen información oculta para el modelo de objetos pero necesaria para

el modelo relacional. En esta categoría entran los mecanismos de concurrencia: estampa de tiempo y versión de objeto. Al leer el objeto, se lee esta información que es ocultada al objeto pero mantenida por el framework.

Lenguajes de consulta

La obtención de varios objetos a través de un lenguaje especial es una de las características más apreciadas.
Por ejemplo, obtener todos los Juan Pérez de una base de Personas.

```
"SELECT Persona FROM Personas WHERE Nombre = 'Juan'
```

Entre otros lenguajes de consulta y modificación de datos podemos destacar LinQ, impulsado por Microsoft para la plataforma .Net.

Hibernate en configuraciones de caché con HQL es uno de los mejores, siendo en estas configuraciones el más rápido en acceso a datos.

Actualización en cascada

La posibilidad de que modificaciones hechas a un objetos
repliquen en los objetos relacionados.

Operaciones en masa

Habrá veces que por razones de performance, se querrá hacer una operación en masa. Por ejemplo, actualizar todos los objetos con nombre Juan a J. De la manera tradicional, deberíamos leer cada objeto, modificarlos en memoria, y recién allí guardarlos de nuevo.

Tecnologías y Estándares en la Capa de Persistencia

Estándares de Conectividad a Base de Datos

OLEDB (Object Linking and Embedding for Databases)

Traducido cómo ("Enlace e incrustación de objetos para bases de datos") es una tecnología desarrollada por Microsoft usada para tener acceso a diferentes fuentes de información, o bases de datos, de manera uniforme.
OLE DB permite separar los datos de la aplicación que los requiere. Esto se hizo así ya que diferentes aplicaciones requieren acceso a diferentes tipos y almacenes de datos, y no necesariamente desean conocer cómo tener acceso a cierta funcionalidad con métodos de tecnologías específicas. OLE DB está conceptualmente dividido en consumidores y proveedores; el consumidor es la aplicación que requiere acceso a los datos y el proveedor es el componente de software que expone una interfaz OLE DB a través del uso del Component Object Model (COM).

ODBC (Open DataBase Connectivity)

Es un estándar de acceso a las bases de datos desarrollado por SQL Access Group en 1992. El objetivo de ODBC es hacer posible el acceder a cualquier dato desde cualquier aplicación, sin importar qué sistema de gestión de bases de datos (DBMS) almacene los datos. ODBC logra esto al insertar una capa intermedia (CLI) denominada nivel de Interfaz de Cliente SQL, entre la aplicación y el DBMS. El propósito de esta capa es traducir las consultas de datos de la aplicación en comandos que el DBMS entienda. Para que esto funcione tanto la aplicación

como el DBMS deben ser compatibles con ODBC, esto es que la aplicación debe ser capaz de producir comandos ODBC y el DBMS debe ser capaz de responder a ellos.

JDBC (Java Database Connectivity)
Es una API que permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java, independientemente del sistema operativo donde se ejecute o de la base de datos a la cual se accede, utilizando el dialecto SQL del modelo de base de datos que se utilice.
El API JDBC se presenta como una colección de interfaces Java y métodos de gestión de manejadores de conexión hacia cada modelo específico de base de datos.

Estándares de Acceso a Datos y Servicios

ADO, ADO.NET (ActiveX Data Objects)
Fue desarrollado por Microsoft y es usado en ambientes Windows por lenguajes de programación como Visual Basic, Visual C++, Delphi entre otros, como también en la Web mediante el uso de Active Server Pages (ASP) y el lenguaje VBScript.
ADO.NET fue desarrollado cuando se introdujo la tecnología .Net Framework, y es a veces considerado como una evolución de la tecnología (ADO), pero fue cambiado tan extensivamente que puede ser concebido como un producto enteramente nuevo.

Java Persistence API (JPA)
Es la API de persistencia desarrollada para la plataforma Java EE. La Java Persistence API, a veces referida como JPA, es un framework del lenguaje de programación Java que maneja datos relacionales en aplicaciones usando la Plataforma Java en sus ediciones Standard (Java SE) y Enterprise (Java EE).
La JPA fue originada a partir del trabajo del JSR 220 Expert Group. Ha sido incluida en el estándar EJB3.

Persistencia en este contexto cubre tres áreas:

- La API en sí misma, definida en javax.persistence.package
- La Java Persistence Query Language (JPQL)
- Metadatos objeto/relacional

El objetivo que persigue el diseño de esta API es no perder las ventajas de la orientación a objetos al interactuar con una base de datos (siguiendo el patrón de mapeo objeto-relacional), como sí pasaba con EJB2, y permitir usar objetos regulares (conocidos como POJOs).

Java Data Objects (JDO)
Es una especificación de persistencia para Java, **independiente del modelo de datos**. Una de sus características principales es la transparencia de los servicios persistentes al modelo de dominio. Los objetos persistentes JDO son clases comunes de Java (POJOs) no hay requisitos para implementar ninguna interfaz en particular ó de extender de clases especiales. Fue lanzada en 2006 y permanece en desarrollo activo en la actualidad por la fundación Apache. Se encuentra en su versión 3.0.

Java Transaction API (JTA)
JTA (API para transacciones en Java) es parte de los APIs de Java EE, JTA establece una serie de Interfaces java entre el manejador de transacciones y las partes involucradas en el

sistema de transacciones distribuidas: el servidor de aplicaciones, el manejador de recursos y las aplicaciones transaccionales, JTA fue desarrollado por SUN. JTA es una especificación construida bajo el Proceso de comunidad Java JSR 907.

Mapeo Objeto-Relacional

Los modelos ORM proporcionan grandes ventajas a proyectos que empiezan, ya que permiten tener acceso a los datos de una manera sencilla y rápida, además de proporcionar cierta abstracción sobre la base de datos que se encuentra por debajo.
Es cierto que no es la única manera de acceder a los datos, ya que soluciones específicas habitualmente tendrán mejor rendimiento, ya que no se hace una transformación de las ordenes y de los datos.

Resulta conveniente utilizarlo en los siguientes contextos:

1. Se puede diseñar fácilmente la base de datos, esto implica que podremos seguir prácticas que facilitarán la interacción ORM-DB.
2. Cuando se desea abstraerse del motor de base de datos y poder cambiarlo sin demasiado inconveniente.
3. Cuando hay que optimizar el tiempo del equipo de desarrollo.
4. Cuando hay que realizar un prototipado rápido.

No resulta conveniente utilizarlo en los siguientes contextos:

1. Cuando existen bases de datos que posean características como:
    - Tablas con varios campos como clave principal o sin claves primarias y/o foránes, ya que eso podría dificultar el mapeo y las consultas.
    - Estructuras de tablas legacy (heredadas)
2. Cuando se necesita ejecutar procesos BATCH con millones de registros.
3. Cuando el equipo de desarrollo no tiene la capacidad suficiente para trabajar con el modelo orientado a objetos.

El ORM debería resolver la mayoría de las cargas.
Un buen ORM permite:

- Mapear clases a tablas: propiedad a columna, clase a tabla.
- Persistir objetos. A través de un método Orm.Save(objeto). Encargándose de generar el SQL correspondiente.
- Recuperar objetos persistidos. A través de un método objeto = Orm.Load(objeto.class, clave_primaria).
- Recuperar una lista de objetos a partir de un lenguaje de consulta especial. A través de un método.

Entonces el ORM es una técnica de programación que me permite convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y el utilizado en una base de datos relacional, utilizando un motor de persistencia. En la práctica esto crea una base de datos orientada a objetos virtual, sobre la base de datos relacional. Esto posibilita el uso de las características propias de la orientación a objetos (básicamente herencia y polimorfismo).
En el momento de persistir un objeto, normalmente, se abre una conexión a la base

de datos, se crea una sentencia SQL parametrizada, se asignan los parámetros y recién allí se ejecuta la transacción. Imaginemos que tenemos un objeto con varias propiedades, además de varias relaciones:
-¿como las asociamos relacionalmente?
-¿Cómo las almacenamos?
-¿Automáticamente, manualmente?
-¿Qué pasa con las claves secundarias?
Ahora, si necesitamos recuperar los datos persistidos:
-¿Cargamos únicamente el objeto?
-¿Cargamos también las asociaciones?
-¿Cargamos el árbol completo?
Y si los mismos objetos están relacionados con otros:
-¿se cargan n veces hasta satisfacerlos?
Como se puede comprobar por lo que acabamos de decir, la brecha existente entre los paradigmas de objetos y relacional se vuelve mucho mayor si disponemos de modelos con objetos "grandes". De hecho, hay estudios que muestran que un 35% del código de una aplicación se produce como consecuencia del mapeado (correspondencia) entre los datos de la aplicación y el amacén de datos.
Dicho todo esto, lo que necesitamos es una herramienta ORM.
Básicamente, una ORM intenta hacer todas estas tareas pesadas por nosotros. Con una buena ORM, tendremos que definir la forma en la que estableceremos la correspondencia entre las clases y las tablas una sóla vez (indicando qué propiedad se corresponde con qué columna, qué clase con qué tabla, etc.). Después de lo cual podremos hacer cosas como utilizar POJO's (Plain Old Java Objects) de nuestra aplicación y decirle a nuestra ORM que los haga persistentes, con una instrucción similar a esta: orm.save(myObject). Es decir, una herramienta ORM puede leer o escribir en la base de datos utilizando los objetos de negocio directamente.
Más formalmente: un modelo del dominio representa las entidades del negocio utilizadas en una aplicación Java. En una arquitectura de sistemas por capas, el modelo del dominio se utiliza para ejecutar la lógica del negocio en la capa del negocio (en Java, no en la base de datos). Esta capa del negocio se comunica con la capa de persistencia subyacente para recuperar y almacenar los objetos persistentes del modelo del dominio. ORM es el middleware en la capa de persistencia que gestiona la persistencia.

Ventajas

● Rapidez en el desarrollo.
La mayoría de las herramientas actuales permiten la creación del modelo por medio del esquema de la base de datos, leyendo el esquema, nos crea el modelo adecuado y viceversa. (Ingenieria Inversa sobre la BD)
● Abstracción de la base de datos.
En general al utilizar un sistema ORM, lo que conseguimos es separarnos totalmente del sistema de Base de Datos que utilicemos, y así si en un futuro debemos de cambiar de motor de bases de datos, tendremos la seguridad de que este cambio no nos afectará a nuestro sistema, siendo el cambio más sencillo, siempre dentro de las opciones soportadas por la herramienta.
● Seguridad.
Los ORM suelen implementar sistemas para evitar tipos de ataques como pueden ser los SQL injections.

● Mantenimiento del código.
Nos facilita el mantenimiento del código debido a la correcta ordenación de la capa de datos, haciendo que el mantenimiento del código sea mucho más sencillo.
● Lenguaje propio para realizar las consultas. Estos sistemas de mapeo traen su propio lenguaje para hacer las consultas, lo que hace que los usuarios dejen de utilizar las sentencias SQL para que pasen a utilizar el lenguaje propio de cada herramienta.

Desventajas

● Configuración: la tecnología ORM requiere archivos de configuración a los esquemas en las estructuras de tabla de asignación de objetos. En los sistemas de las grandes empresas la configuración crece muy rápidamente y se convierte en extremadamente difícil de crear y administrar.
● Genera una gran dependencia con el Framework utilizado.
● Consultas personalizadas: La capacidad para asignar consultas personalizadas que no se ajustan a cualquier objeto definido o bien no es compatible o no recomendados por los proveedores del framework. Los desarrolladores se ven obligados a encontrar work-arounds escribiendo objetos ad hoc y consultas, o escribir código personalizado para obtener los datos que necesitan. Ejemplo
consultas sql directas y llamadas a stored procedures.
● Los ORM, al generar SQL dinámicamente, en muchos casos no pueden utilizar sintaxis específica de determinados motores.
● Rendimiento: Las capas ORM utilizan reflection e introspection para crear instancias de los objetos y rellenar con los datos de la base de datos. Estas son operaciones costosas en términos de procesamiento y agregan a la degradación del rendimiento de las operaciones de asignación. Las consultas de objetos sin optimizar causan pérdidas significativas en el rendimiento y sobrecargan los sistemas de bases de datos de gestión. El ajuste del rendimiento de SQL es casi imposible, ya que los frameworks proporcionan poca flexibilidad de control del SQL que se ha autogenerado.
● Caches: Este método también requiere el uso de cachés de objetos y contextos que son necesarios para mantener y realizar el seguimiento del estado del objeto y reducir ida y vuelta base de datos para los datos almacenados en caché. Estos cachés si no son cuidados y sincronizados en una implementación de múltiples niveles puede tener ramificaciones significativas en términos de datos, precisión y simultaneidad. A menudo cachés de terceros o cachés externas tienen que estar conectado para resolver este problema.

Performance

A pesar de que la performance de las ORM puede parecer, a primera vista, mucho más pobre que crear una solución especial para cada situación, lo cierto es que hay ciertas técnicas y herramientas que nos permiten disminuir estos problemas a un mínimo.
El agregado de una Cache puede mejorar mucho la performance, hibérnate (Producto comercial ORM) por
ejemplo viene con cache de dos niveles. Si ésta es configurada apropiadamente para las circunstancias particulares de un desarrollo se puede mejorar mucho la performance. Por otro lado, Si no se hace correctamente la cache puede tener problemas como:
● Modificación de la información desde afuera del cache (la información del cache

queda invalida)
● Uso excesivo de memoria
También se pueden usar técnicas como lazy-loading para mejorar la performance, de forma de no cargar en memoria todos los objetos persistidos, sino solo los que se estén usando realmente. Esto deja más memoria libre y mejora la performance dado que no se accede tanto a la base de datos.

Productos Disponibles

Aunque algunos programadores prefieren crear sus propias herramientas ORM, hay paquetes comerciales y de uso libre disponibles que desarrollan el mapeo relacional de objetos:

Doctrine

Es un framework ORM para PHP 5.2 y posterior, y entre sus puntos fuertes destaca su lenguaje DQL (Doctrine Query Language) que está inspirado en el HQL de Hibernate. Para crear el modelo, Doctrine nos da dos alternativa, hacer una clase por tabla e indicarle mediante PHP el tipo de datos que almacenaremos. O bien utilizar un esquema en formato YAML (similar a XML, pero más legible para las personas)

Propel

Es otro framework ORM para PHP 5 y superior y que está arropado por el framework Synfony. Podemos acceder y modificar los datos de la base de datos utilizando la lógica de programación orientada a objetos, en vez de utilizar los clásicos Select y Updates de SQL.

Entity Framework

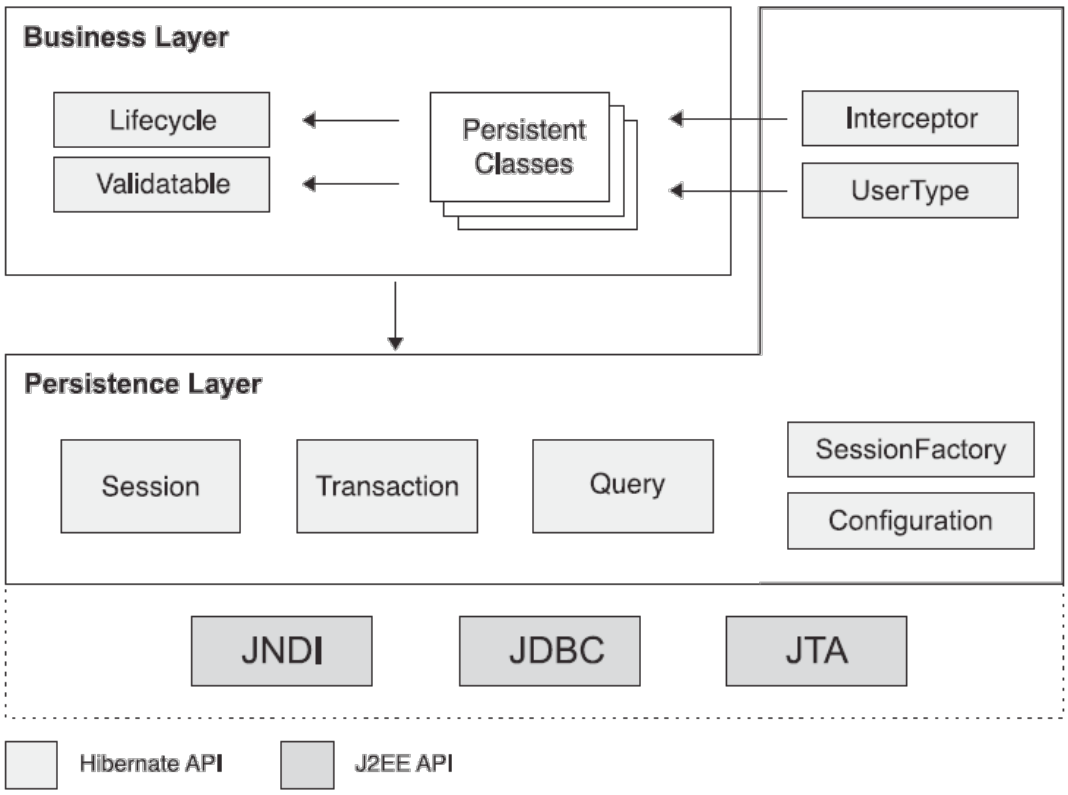
Es un ORM desarrollado por Microsoft para el mapeo objeto-relacional para los lenguajes Visual Basic. Net y C#. Incluye una herramienta llamada SQLMetal que permite la generación automática de clases directamente desde una base de datos MSSQL.

Hibernate, NHibernate

Es una ORM de libre distribución, que además, es de las más maduras y completas. Actualmente su uso está muy extendido y además está siendo desarrollada de forma muy activa. Una característica muy importante que distingue Hibernate de otras soluciones al problema de la persistencia, como los EJBs de entidad, es que la clase Hibernate persistente puede utilizarse en cualquier contexto de ejecución, es decir, no se necesita un contenedor especial para ello.

Hibernate

A continuación hablaremos de algunos de los aspectos más importantes de Hibernate.



Interfaz Session

Es una de las interfaces primarias en cualquier aplicación Hibernate. Una instancia de Session es "poco pesada" y su creación y destrucción es muy "barata". Esto es importante, ya que nuestra aplicación necesitará crear y destruir sesiones todo el tiempo, quizá en cada petición. Puede ser útil pensar en una sesión como en una caché o colección de objetos cargados (a o desde una base de datos) relacionados con una única unidad de trabajo. Hibernate puede detectar cambios en los objetos pertenecientes a una unidad de trabajo.

Interfaz SessionFactory

Permite obtener instancias Session. Esta interfaz no es "ligera", y **debería compartirse entre muchos hilos de ejecución**. Típicamente hay **una única SessionFactory** para toda la aplicación, creada durante la inicialización de la misma. Sin embargo, si la aplicación accede a varias bases de datos se necesitará una SessionFactory por cada base de datos.

Interfaz Configuration

Se utiliza para configurar y "arrancar" Hibernate. La aplicación utiliza una instancia de Configuration para especificar la ubicación de los documentos que indican el mapeado de los objetos y propiedades específicas de Hibernate, y a continuación crea la SessionFactory.

Interfaz Query

La interfaz Query **permite realizar peticiones a la base de datos** y controla cómo se ejecuta dicha petición (query). **Las peticiones se escriben en HQL** o en el dialecto SQL nativo de la base de datos que estemos utilizando. Una instancia Query se utiliza para enlazar los parámetros de la petición, limitar el número de resultados devueltos por la petición, y para ejecutar dicha petición.

Type

Un elemento fundamental y muy importante en la arquitectura Hibernate es la noción de Type. Un objeto Type Hibernate hace corresponder un tipo Java con un tipo de una columna de la base de datos. Todas las propiedades persistentes de las clases persistentes, incluyendo las asociaciones, tienen un tipo Hibernate correspondiente.
Este diseño hace que Hibernate sea altamente flexible y extendible. Incluso se permiten tipos definidos por el usuario (interfaz UserType y CompositeUserType).

Una Aplicación de Ejemplo

Bibliografía

[1] George Reese, “Database Programming with JDBC and Java”, O'Reilly Media, 1997.

[2] Todd M. Thomas, *“Java Data Access—JDBC, JNDI, and JAXP”*, M&T Books, 2002.

[3] José María Arranz Santamaría, *“Estrategias de Persistencia Orientada a Objetos en Java con JDBC: Una comparativa desde la práctica”*, Universidad Politécnica de Madrid, 2003.

[4] Pablo Pizarro, *“ORM, Object-Relational Mapping: Mapeo del modelo de objetos al modelo relacional”*, Universidad de Mendoza, 2005.

[5] Christian Bauer, Gavin King, *“Hibernate in Action. Java Persistence With Hibernate”*, Manning, 2007.

[6] Mike Keith, Merrick Schincariol, *“Pro EJB 3: Java Persistence API”*, Apress, 2006.

[7] Dave Minter, Jeff Linwood, *“Pro Hibernate 3”*, Apress, 2005

[8] Roger Jennings, *“Professional ADO.NET 3.5 with LINQ and the Entity Framework”*, Wiley Publishing, 2009.

[9] Martin Fowler, *“Patterns of Enterprise Application Architecture”*, Addison-Wesley, 2003

[10] Página Oficial de LinQ en Microsoft, Consultado en Septiembre de 2012.
http://msdn.microsoft.com/es-es/library/bb397926.aspx

[11] Página Oficial de Hibernate en JBoss Community.
http://www.hibernate.org/